
# University of Alberta

# TIGUKAT: A Uniform Behavioral Objectbase Management System

by

Randal J. Peters

Technical Report TR 94–06
April 1994

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

UNIVERSITY OF ALBERTA

# TIGUKAT: A Uniform Behavioral Objectbase Management System

BY

Randal J. Peters

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

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Spring 1994

# Abstract

Object-oriented computing is influencing many areas of computer science, including database systems. Despite many advances, object-oriented computing is still in its infancy and a universally accepted definition of an object-oriented model is virtually nonexistent. In this thesis, the object model, meta-model, query model, dynamic schema evolution policies, and version control of the TIGUKAT objectbase management system are presented. An identifying characteristic of this system is that all components are uniformly modeled as objects with well-defined behavior. This is an important achievement towards advancing database technology because it unifies the components of a database within a single, clean, underlying semantics that can be easily extended to support other database services. The TIGUKAT object model is purely *behavioral*, supports full encapsulation of objects, defines a clear separation between primitive components, and incorporates a *uniform* semantics over objects. A behavioral model definition specifies the semantics of objects and this is integrated with a structural model to form a complete model definition. The meta-model is uniformly represented within the object model, giving rise to *reflective capabilities*. The query model is uniformly defined as type and behavior extensions to the base model, thus incorporating queries and query processing as extensible parts of the model. The complete query model includes a formal object calculus, formal object algebra, a definition of safety based on the *evaluable* class of queries (arguably the largest class of "reasonable" queries), proofs of completeness, and an effective algorithmic translation from the calculus to algebra. Dynamic schema evolution is a necessary feature that allows for the timely change of information and for restructuring the schema of an objectbase. Since everything is uniform, the schema evolution policies are simply behavior extensions to the base model. Temporality is incorporated to support versioning of objects and of schema. It is also used to maintain the semantic consistency of evolving behaviors.

This research leads toward the development of an extensible query optimizer, view manager, and transaction manager as uniformly integrated components of the system. This fulfills the typical gamut of database services. Temporal extensions and a seamlessly integrated database programming language are other components that this research supports.

# Acknowledgements

I would like to express my sincere thanks to Dr. Tamer Özsu, my supervisor, for his encouragement and support during my research. He created a balanced environment that allowed me the necessary freedom to pursue my ideas, while at the same time offering the right amount of guidance to keep me focused.

I would also like to thank the members of my committee Dr. Duane Szafron, Dr. William Armstrong, Dr. Paul Sorenson, Dr. Alberto Mendelzon, and Dr. Barrie Nault for their many insights and comments. I especially thank Duane Szafron for all his help on the object model and query model.

Thanks to the database research group at the University of Alberta for the many interesting discussions. I am especially grateful to those members who read earlier drafts of this work and for the useful comments they provided.

Thanks go to Dr. Özsu, the Natural Science and Engineering Research Council of Canada, the University of Alberta, and the Department of Computing Science for the funding they provided throughout this degree.

I express my sincere gratitude to my wife, Sherry, for her patience, encouragement, and love that gave me strength to carry this dream to reality.

Finally, I thank my parents, David and Tina Peters, for their love and complete support during my studies and throughout my life. This thesis is for them.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Overview

Object-oriented computing is influencing many areas of computing science, including database management. The appeal of object-orientation, from the perspective of database applications, is attributed to its higher levels of abstraction for modeling real world concepts, its support for extensibility through user-defined types, and its potential for managing interoperability.

Objectbase management systems (OBMSs)[1] are emerging as the most likely candidate to meet the complex data and information management requirements of new applications such as geographic information systems, computer aided design (CAD), computer aided manufacturing (CAM), multimedia systems, knowledge base applications, and office information systems. The general acceptance of this technology is dependent on the increased functionality it can provide. In this respect, OBMSs subsume the modeling power and expressibility of the first-generation (i.e., hierarchical and network) and second-generation (i.e., relational) systems. Unlike these earlier systems, OBMSs are well suited for handling complex information with complex relationships. Furthermore, an OBMS is better suited to integrate the components of traditional database systems such as a query model, query optimizer, schema evolution, version control, view management, transaction management, rule systems, and so on into a single, uniform system.

Despite many advances over the last decade, objectbase management technology is still in its infancy. The field is generally suffering from the absence of a universally accepted object model, along the lines of the relational model [Cod70], whose features are formally and unambiguously defined. This void makes it difficult to reason about the internal consistency of these models, investigate database features such as query models, schema evolution, views, transaction management, etc., and to generalize the results of various studies. Some standardization efforts are being pursued [ABD+89, SRL+90, FKMT91], and general descriptions of model characteristics are emerging [ZM90, Ken90a]. These have resulted in the definition of a relatively small set of core concepts that most object models share.

[1] In this thesis, the terms "objectbase" and "objectbase management system" are preferred over the more popular terms "object-oriented database" and "object-oriented database management system", since not only data in the traditional sense is managed, but also objects in general, which includes things such as code and complex information in addition to data.

## 1.2 Scope and Contributions

This thesis describes the development of the TIGUKAT[2] extensible objectbase management system (OBMS). A uniform, behavioral object model with extensible properties is developed, and this model is used to develop the foundations of the extensible OBMS, including a full-featured object query model, a uniform meta-model with reflective capabilities, dynamic schema evolution policies, and version control. This section provides an overview of the contributions in each of these areas.

Although the work in this thesis is within the context of TIGUKAT, the findings extend to any system based on a uniform, behavioral object model where behaviors define the semantics of types and are implemented within a functional paradigm.

The related areas of research touched upon in this thesis, but outside its scope, include the implementation of the object model [Ira93], the definition and implementation of a user query language [Lip93], the definition and implementation of a query optimizer and execution plan generator [Muñ94], the incorporation of temporality into the model [GÖ93], the definition and implementation of a general objectbase programming language, and distributed aspects of OBMSs.

### 1.2.1 Object Model Issues

The TIGUKAT object model is characterized by a purely behavioral semantics, a uniform approach to object modeling, and extensibility. The behavioral paradigm provides a consistent underlying operational semantics and uniformity provides a fundamental conceptual model where every concept, including types, classes, collections, behaviors, functions and meta-information, is modeled as a first-class object with well-defined behavior. The features of uniformity and the behavioral paradigm form the foundation for the extensibility of the model.

In TIGUKAT, traditional structural notions such as instance variables, method implementations, and schema definition are cast into the uniform semantics of behaviors on objects. The behavioral paradigm introduces the notion that the operations that may be performed on an object are given entirely by the behaviors defined on the type of that object. Uniformity is important in unifying the components of an OBMS into a seamless integrated system with a single underlying (behavioral) semantics.

A fundamental characteristic of object models, which differentiate them from other models, is their richer semantics. On the one hand, this enables closer modeling of complex real world applications such as geo-information and CAD/CAM systems, which makes object models more powerful. On the other hand, the richer semantics makes it more difficult to specify a clean, well-defined, universally accepted model. The power and expressibility of a general object model may prove too difficult to formalize because many important properties become intractable as the model becomes more general [Mai89]. However, certain precautions have been identified to avoid pitfalls while developing a complete object model [KW89, Bee90]. The resulting model definition may be more restrictive than a "general" model, but power and expressibility (which may not be needed anyway) must sometimes be traded for tractability.

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

The first result of this research is the development of an advanced object model through the identification and formalization of object-oriented characteristics with sufficient power and flexibility for supporting the advanced functionality demanded by OBMSs and their client applications. The TIGUKAT object model includes many of the core concepts introduced by former models, along with additional features that extend its modeling power and expressibility beyond others.

The power of the model is demonstrated in this thesis by using it to develop, in an extensible way, a uniform meta-system that is seamlessly integrated with the base model and provides reflection [PÖ93], an object query model with powerful querying facilities [PLÖS93b, PLÖS93a], plus dynamic schema evolution strategies and version control that use time to manage versions of objects and maintain semantic consistency of behaviors.

A model for objects involves the specification of two components. One part consists of the behavioral aspects, which define a universal conceptual abstraction of objects, including the relationships between objects. The other part is the structural definition, which specifies the internal organization of objects and how their relationships are organized. Subtleties, such as the difference between objects and values (hidden by the abstraction of the behavioral model) are exposed at the structural level. King [Kin89] points to the similarity between a structural object model and the semantic data modeling approach [HM81, HK87, PM88] in the sense that both are concerned with the representation of data and knowledge. A behavioral model goes further by addressing access and manipulation of objects from general purpose programming and query languages.

Behavioral and structural issues have traditionally been treated separately in the database community, with object models emphasizing one or the other. A notable exception is [Bee90], which attempts to establish a link between the two, although the behavioral and structural definitions of that model are not fully developed. Behavioral and structural aspects are both important in the development of an object model, but the two are independent, which accounts for the orthogonal directions taken by recent studies. Reconciling these approaches assists in understanding the model and forms a basis for an implementation of the model.

It has been noted that the behavioral aspects are fundamental in developing a theory of objects [Ken90a]. In this thesis, a behavioral model is coupled with a formal structural counterpart to unify the model semantics and form a complete definition. Beeri's formal structural model [Bee90] is chosen as a basis for the structural model. Several modifications are incorporated into Beeri's model in order to extend its capabilities to match the uniformity and enhanced functionality provided by the behavioral model. The integration of these two definitions results in a complete, uniform object model specification, which is a favorable platform for the implementation of TIGUKAT.

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

1. A precise specification and integration of both the behavioral and structural aspects of a uniform object model with the necessary power for handling advanced database functionality such as a powerful query model and language, schema evolution, version control, updatable views, transaction management, temporal rules, and so on.

2. A clean separation and precise definition of many object model features that are usually bundled and only intuitively defined in other studies.

3. A uniform approach to objects that models all information as first-class objects with well-defined behavior.

4. Reflective capabilities through the uniform modeling of meta-information as objects with well-defined behavior.

### 1.2.2 Query Model Issues

Two important measures of an OBMS lie in the power of its query model and the languages used to query the objectbase. User requirements of these systems demand a declarative facility to formulate queries by focusing on "what" information is needed and leaving it up to the system to determine "how" to efficiently retrieve the information. Therefore, a formal query model for these systems defines an object calculus as a theoretical framework for supporting declarative queries and a procedural (or functional) algebra to execute them efficiently. In order to support this framework, it is desirable that the calculus and algebra be equivalent in expressive power and that there be an efficient translation from calculus to algebra. Both of these properties are fulfilled by the query model presented in this thesis.

The TIGUKAT query model is defined as a uniform extension to the base object model. The formal languages include a declarative object calculus and a behavioral/functional object algebra. The query model is an extension to the object model in that queries are defined as type and behavior extensions to the base model, meaning they inherit all the characteristics of objects. One advantage of this approach is that the components of an integrated query model can be queried just like other objects. For example, one may query a collection of queries to gather statistical information about their performance, or a query on the types and behaviors of the query model may be run to analyze their definition. Another advantage is that the types and behaviors of the query model can be extended through the application of appropriate behaviors. This kind of "open-architecture" results in an extensible query model that allows advanced information processing features to be added as they are discovered using the operations provided by the base model.

Safety is an important consideration of a query model. Essentially, a query is safe if it returns a finite result in a finite amount of time [OW89]. Developing efficient methods for recognizing broader classes of safe queries and rejecting those that are unsafe is a major research issue. The TIGUKAT query model bases safety on one of the largest known class of decidable queries.

The result of a query depends on the domains referenced within that query. *Domain independence* is a property of queries that states the result of a query is not affected by changes to domains not referenced within the query. The domain independent class of queries [Mak81] has long been recognized as the largest class of "reasonable" queries. However, it is well-known that domain independence is an undecidable problem. Many decidable subclasses of the domain independent class have been proposed. The "evaluable" class of queries [GT91] is touted as the largest decidable subclass of the domain independent class. The class of safe queries in TIGUKAT is based on the evaluable class. However, the semantic characteristics of object generation introduced by the query model are exploited to extend this class and provide a broader class of safe queries. In [EMHJ93a], a similar approach is presented that extends the evaluable class with scalar functions, although that work is within the context of the relational model.

The identifying characteristics of the TIGUKAT query model that differentiates it from other object query models are the following:

1. It incorporates a formal and powerful object calculus and object algebra with a proven equivalence in expressive power and an effective (i.e., algorithmic) translation from calculus to algebra.

2. Its safety criterion is based on the *evaluable* class of queries, which is arguably the largest decidable subclass of the (undecidable) domain independent class.

3. It exploits object-oriented features to extend the *evaluable* class by introducing notions of *object generation* on equality and membership atoms, which relaxes *range* specification requirements. The result is that the broadest class of safe queries known to date is recognized by the approach.

4. It uniformly models queries as first-class objects by directly defining them as type and behavior extensions to the TIGUKAT object model. This results in an extensible query model with a consistent, uniform, underlying semantics commensurate with the object model and its behavioral semantics.

5. The extensible algebra specification forms a uniform basis for processing queries that is exploited by the extensible algebraic query optimizer and execution plan generator [Muñ94].

6. It is the most advanced object query model to be uniformly integrated with a base object model in an extensible way, thereby unifying the components of an object calculus, an object algebra, proofs of completeness between the languages, and an effective translation from calculus to algebra within a common framework.

### 1.2.3 Schema Evolution and Version Control Issues

Dynamic schema evolution is the ability for a system to make changes to the database schema while applications are running. The kinds of changes allowed, and the semantics of these changes, vary in models proposed in the past. However, there is a fundamental set of changes that is common to all models.

Schema evolution is necessary in complex applications in order to handle post-design modifications that are typical in these systems. Some examples include changes in the the way the application domain is structured, changes in the functionality of a particular application, and changes needed in order to meet performance requirements. If properly defined, schema evolution can also be used to support experimentation, or "what if" scenarios, with existing applications.

In this thesis, the full schema evolution policies in the TIGUKAT object model are presented. Everything is uniformly an object in TIGUKAT, but the schema evolution component characterizes some objects as being part of the "schema" in order to define evolutionary operations on them. Objects of other types, such as application specific types, are not considered to be part of the schema and, therefore, schema evolution policies are not defined for them.

Temporality has been introduced as a uniform extension to the TIGUKAT model [GÖ93] and is based on behaviors. A behavior is either temporal or non-temporal. By defining temporal behaviors on types, the types become temporal, and all instances of a temporal type are temporal. Actually, only the temporal behaviors defined by a type are temporal in the objects. Thus, an object may consist of both temporal and non-temporal components.

The temporal aspects are used to implicitly manage histories of behaviors. Behavior histories, in turn, are used to manage the properties of objects over time. By maintaining histories for appropriate behaviors of types, a model for versioning types is developed. This model is extended to include behavior objects and object representations as well.

Since versioning occurs implicitly through the management of behavior histories, objects are instances of a type and not instances of a version of a type. This means that objects support the full semantics of a type instead of just a portion (version) of the type. This is an identifying characteristic of the approach and has the benefit of maintaining semantic consistency between old and new versions of types and the applications that operate on their instances.

By using time to implicitly model versions of types and objects, the schema and its instances can be reconstructed at any time of interest. Each chosen time of interest is considered to be a version. Thus, the granularity of versions is based on the chosen granularity of the time scale, rather than being restricted to version numbers. Note that the granularity of the time scale could be version numbers if so desired. Using a given time reference (version number, etc.), the type lattice, type interfaces, behavior implementations, and object representations can be reconstructed as they existed at that particular time of interest. A contribution of this approach is that historical queries can be run on the objectbase quite easily.

Another identifying characteristic of version model is that object coercion occurs on a "behavior-at-a-time" basis instead of on the entire object. This means that objects can update certain behaviors to use an implementation defined by a newer version of a type, while allowing other behaviors to continue using older versions. This means that a history of the object's semantics is maintained, which helps in maintaining semantic consistency between old and new versions of types and the programs that operate on them.

## 1.3 Organization

The remainder of this thesis is organized into five chapters defining the components of the TIGUKAT objectbase management system considered in this work, plus a summary chapter with concluding remarks and future directions.

In Chapter 2, the definition of the TIGUKAT object model is presented. First, the high-level abstract behavioral definition of the model is given. This defines the primitives that form the base object model and include the primitive type lattice structure. The base model is extended through uniformity to develop other components of TIGUKAT. Second, the behavioral model is linked with a structural example model for completeness. The structural model specifies an organizational, yet implementation independent, representation of conceptual objects of the behavioral model. A simplified Geographic Information System (GIS) is defined as a client OBMS application and is used as a running example to illustrate results throughout this thesis.

In Chapter 3, the TIGUKAT query model is defined as a uniform extension to the object model and the concept of queries as objects is introduced. A formal object calculus is defined by building on the behaviors of the extended object model. A class of *safe* calculus expressions is defined as the set of "reasonable" queries considered for translation to the algebra. The operators of the formal object algebra are presented, along with a description of the type creation and inferencing mechanisms used by the algebra to derive typing information for the results of queries. Finally, the theorems and proofs of equivalence between the calculus and algebra are presented. An effective algorithm for translating safe object calculus expressions into equivalent object algebra expressions is also given.

In Chapter 4, the features of the meta-model introduced in Chapter 2 are presented. These include the ability to extend the meta-model through regular subtyping, defining

behaviors for operating on classes of objects, and the ability to provide reflection, which is the focus of the chapter.

In Chapter 5, the dynamic schema evolution policies and version control are defined. A number of invariants are defined that must be maintained over schema changes. The schema changes allowed by the model are given and their full semantics, including how they maintain the invariants, are presented. The temporality of the object model [GÖ93] is used to maintain histories of behaviors for version control. It is shown how histories are used to manage versions of types and how type versions maintain behavioral consistency between evolving types that may modify the behaviors they define. Propagation of schema changes to the instances is also considered, which results in versioned objects. A complete description of how behaviors are dispatched to versioned objects is presented to illustrate how the time model assists in maintaining behavior consistency between different versions of types.

Conclusions and contributions of this work are presented in Chapter 6. The results are summarized and a number of future research directions that the work suggests are discussed.

Since Chapters 2 through 5 are fairly diverse in subject area, each respective chapter includes a survey of related work for the topic and an overview of the chapter's contents.

Three appendices are included at the end of the thesis. Appendix A and Appendix B specify the semantics of the types and behaviors of the TIGUKAT primitive type system, respectively. These were prepared as part of the implementation of the object model. Appendix C analyzes and compares the characteristics of the TIGUKAT object model with the object-oriented manifestos [ABD+89, SRL+90] and the NIST standards report [FKMT91] as an exercise to illustrate the compliance of TIGUKAT with emerging *de facto* standardization efforts.

# Chapter 2

# The Object Model

Recent work on OBMSs has resulted in a number of object model proposals (see [Day89, MZO89, Ken90b, Sny90, Bee90] among many). Several properties of these models have emerged from the development of various prototype systems, including [GR85, CM84, BMO+89, CDV88, WLH90, Deu90, KGBW90]. Consequently, object models have some variance in the features they support. However, most of them incorporate a set of common core concepts, but the semantics of these concepts lack precise definitions and are, in general, difficult to port from one system to another. The diversity of object model definitions and the lack of a formal object model motivated the need to re-examine the qualities that object-oriented systems provide and to develop a new object model that incorporates these qualities and introduces new ones to extend the power of object models. *Uniformity* is an example of one quality that has not been pursued in other models, but is fully integrated into the object model defined here.

In this chapter[1], the TIGUKAT object model is defined. The model includes some common features of earlier proposals, along with distinctive qualities that extend its power and expressibility beyond others. The TIGUKAT object model is purely *behavioral* in nature, supports full encapsulation of objects, defines a clear separation between primitive components such as *types, classes, collections, behaviors* and *functions*, and incorporates a *uniform* semantics over objects, which makes it a favorable basis for an extensible objectbase management system. Every concept that can be modeled in TIGUKAT has the uniform semantics of a *first-class object* with well-defined behavior.

The literature recognizes two perspectives of an object model: the structural view and the behavioral view. Most object-oriented formalisms have concentrated on one or the other of these two approaches. The TIGUKAT object model includes a behavioral model definition and this is integrated with an example structural model to form a complete model definition.

## 2.1 Related Work

Codd's landmark paper in 1970 [Cod70] defined the relational model which provided a simple, but powerful, method of organizing data. The main advantages of this approach are that it offers a high degree of *data independence*, *data consistency* and language facilities based on the *first-order predicate calculus*. The success of the relational model can be

[1] Portions of this chapter are published in the 1993 *Proceedings of the Centre for Advanced Studies Conference (CASCON'93)* [ÖPI+93].

partially attributed to its precise formal specification which facilitates a systematic investigation of database management system (DBMS) functions such as query processing, views and transaction management. However, it is well recognized that the flat record based representation of the relational model results in a semantic mismatch between the entities being modeled and the underlying DBMS [Ken79].

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

An orthogonal approach to relational model extensions has been to develop a completely new data model with advanced modeling power and expressibility. One class of such models are the *semantic data models* whose key features are based on the abstraction mechanisms of *classification*, *aggregation* and *generalization* [SS77]. These features allow for complex information to be categorized and accessed in meaningful ways. The pioneering models that fall into this category are the Entity-Relationship (ER) model [Che76] and SDM [HM78, HM81]. An overview of the entire field can be found in [HK87, PM88].

Some particular semantic data models that exihibit similarities with TIGUKAT include the following:

- The functional data model and the data language DAPLEX [Shi81] which defines *entities* and *functions* as primitive modeling constructs. In DAPLEX, properties of entities and the relationships among them are modeled as functions. This places the computational power of functional languages on properties and their relationships in a uniform manner, which facilitates a better semantic expression of them. TIGUKAT adopts this uniform functional approach and builds on it with the introduction of behaviors as semantic definitions and the use of functions as the implementations of behaviors.

- SIM [JGF+88] is a commercially available DBMS based on the semantic data model SDM. Entities are defined in terms of simple *data-valued* attributes and more complex *entity-valued* attributes, which represent a binary relationship between two classes of entities. Entities are organized into meaningful collections called *classes*, each of which is either a *base class* (a class defined independently of other classes) or a *subclass* (a class defined in terms of other classes). This gives an inheritance hierarchy for entity classes. TIGUKAT separates the notions of type and class and extends the basic notion of class by supplementing classes with heterogeneous user-defined collections.

- The IFO data model [AH84] formalizes the characteristics of semantic data models and was developed to serve as a theoretical foundation for the investigation of higher-level data modeling. The TIGUKAT object model proposes a similar foundation for the investigation of object-oriented modeling.

Object-oriented models were developed to further enhance the expressiveness and abstraction that semantic data models provide. Despite the number of object-oriented models proposed, no universally accepted model exists. One reason for the absence of such a model is that object-oriented development has followed the same informal route as semantic models.

Typically, DBMS development has followed two streams in the past. The first is to extend object-oriented programming languages (OOPLs) with DBMS features such as persistence and a query facility. The resulting systems are typically a merger between object-oriented and relational systems. Out of this approach has appeared extensions to C++ (e.g., ObjectStore [LLOW91] and EXODUS [CDV88]) and Smalltalk (e.g., GemStone [BMO+89]), among others. The second approach is to develop a language-independent object model and consistently extend it with DBMS features. TIGUKAT follows the second approach as do ORION [BCG+87], $O_2$ [BBB+88], and IRIS [FBC+87], among others.

There are currently several efforts to standardize the features of object-orientation. For example, two recent manifestos have appeared [ABD+89, SRL+90] that propose various features inherent in object-oriented database management systems (OODBMSs). A side-effect of these manifestos is to outline some object-oriented concepts that have sifted through the various model proposals over the years. In addition to these, Zdonik and Maier [ZM90] define a reference model that specifies the common features that should exist in an OODBMS, Wegner [Weg90] examines the goals, concepts and paradigms of object-oriented technology in the forum of object-oriented programming. Bancilhon and Kim [BK90, Kim90b, Kim90a] discuss the issues that will be driving object-oriented research in the next few years. Kent [Ken90a] defines a framework that emphasizes behaviors and their invocations as a means of comparing the "objectness" of different models. The X3/SPARC/DBSSG/OODBTG report [FKMT91] defines an open object model architecture and recommends some standards for object management (ODM). Furthermore, several other classifications of object-oriented concepts have appeared [CW85, SB85, AC86, KC86, Ull87, Weg87, Kin89, Mai89, Nie89, Str90]. These papers serve as useful guidelines to measure the "objectness" of various models. The formal model developed in this thesis draws from all these reports and incorporates several of their core concepts. A comparison of the TIGUKAT object model with these guidelines is given in Appendix C. Other models that have influenced the design of TIGUKAT are discussed below.

Kent [Ken90b] defines a model that specifies a rigorous semantics for the existence of objects through unique object identities and has separated this from the semantics for accessing objects, which is achieved through non-unique object references. The TIGUKAT object model incorporates a semantics for object identity and object reference that is similar to the concepts presented by Kent.

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

Beeri's model proposal [Bee90] is an analysis and classification of the formal aspects and common features found in most current OODBMSs. The framework of this model includes both structural and behavioral components. The structural model deals with the representation of complex structured objects vs. atomic data values, notions of object identity, organization of inheritance graphs, and semantics of declarative languages. The

behavioral component explores higher-order concepts of object-orientation such as model uniformity, the semantics of methods, the application of methods to objects, and the semantics of inheritance. The model is mostly a sketch of ideas and is meant as a motivation for object-oriented researchers to refine the formal aspects of object models. Emphasis on logic-oriented modeling is evident throughout the paper. The structural model presented in Section 2.5 of this thesis has evolved directly from the concepts presented by Beeri.

Maier, Zhu and Ohkawa [MZO89] outline the structural object model TEDM, which encompasses prominent features of the object-oriented and logic programming worlds. From the object-oriented side, TEDM includes support for object identities, complex objects, type structures, and property inheritance. Types in TEDM have both an *intensional* and *extensional* aspect. The *intensional* view consists of the structural organization of the type in how it defines the representation of its instances. The *extensional* view denotes the collection of objects adhering to the intensional structure of the type. Thus, TEDM separates the notion of a type from its extent. However, the entire extent of a type is not automatically maintained by the model (i.e., there is no notion of a class) and in this respect resembles the structural model of Beeri [Bee90].

The TIGUKAT object model supports the separation of type and extent, but automatically maintains the extent of a type through a class. Collections are introduced to allow for general, heterogeneous user-defined groupings of objects. In this way, classes are maintained by the system to generate the entire extent of types, and there is the added flexibility of user-defined collections for customized, application-specific groupings of objects. From the separation of types and extents, the notions of *specialization* vs. *subtyping* evolved and are defined in TEDM. These notions are included in the design of TIGUKAT because of their application in type inferencing.

The PROBE Data Model (PDM) [MD86] is based on the functional data model DAPLEX [Shi81]. PDM defines *entities* that denote individual elements such as PERSONs or MATERIALs, and *functions* to represent properties of entities and the relationships among them. PDM generalizes the functional language of DAPLEX by defining a function as a relationship between collections of entities and scalar values. This generalization allows functions with zero or more inputs and one or more outputs. Furthermore, function arguments can serve as both inputs and outputs in PDM. DAPLEX functions on the other hand allow zero or more inputs and only one output, and each argument must be either an input or the single output. PDM allows functions that store values explicitly (stored functions) or that compute values through a piece of code (computed functions). However, syntactically all functions resemble computed functions. Functions in TIGUKAT are multiple input, but only single output because the result of a function must uniformly be an object. Multiple outputs can be handled by returning a single product object that is a conglomeration of other objects. The universal treatment of stored and computed functions is incorporated into TIGUKAT.

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

Iris [FBC+87, FAC+89, WLH90] is a commercial OODBMS founded on the functional data model of DAPLEX [Shi81]. The Iris model defines primitives for *objects*, *types*, and *functions*. Objects are classified into the categories of *literal* (atomic) and *non-literal* (complex) objects. Literals denote the directly system representable atomic building blocks of non-literal objects. Iris fully encapsulates object properties into behaviors (i.e., functions

or operations), which represent the only interface to objects. Thus, a high-level of data abstraction and data independence is supported by the model. Operations take objects as arguments and produce objects as results. All objects are classified into types, which define the operations applicable to objects in the extent of that type. Types may be structured into subtype/supertype relationships and multiple subtyping is supported. Classes of objects may overlap, meaning an object may belong to several heterogeneous types simultaneously unless there is an explicit declaration restricting classes to be disjoint (classes in subtype/supertype relationships must overlap). There is no support for separate user-defined collections in Iris. TIGUKAT adopts complete encapsulation of behaviors that uniformly accept objects as inputs and produce objects as results. The structural model refines this perspective by distinguishing between atomic, abstract and complex structured values. The TIGUKAT model supports heterogeneity through collections, and classes are restricted collections of objects that must be in a subset relationships with one another.

$O_2$ is a commercially available OODBMS [Deu90, Deu91, BDK92]. It consists of a formal model definition based on the framework of a set-and-tuple data model [LRV88, BBB+88] and includes set, tuple, and list constructors for modeling complex nested objects [LR89a]. The $O_2$ model supports subtyping based on the set inclusion semantics developed in [Car84] and this is used to establish classes of objects. Explicit user-defined collections are not supported. The language features of $O_2$ include an object-oriented database programming language called $CO_2$ [LR89b] with C++ like features and an SQL-like ad-hoc query language called RELOOP [BCD89, CDLR90]. The query language is tightly integrated with $CO_2$ and thus does not suffer from the "impedance mismatch" problem. Unlike TIGUKAT, the $O_2$ languages are not based on a complete formal query model that includes an object calculus and an equivalent algebra. The emergence of $O_2$ as a commercial OODBMS makes it valuable as a benchmark system for ranking other systems on their performance and industrial viability.

Smalltalk [GR85] was one of the first commercially available object-oriented languages. However, Smalltalk on its own lacks the functionality of database systems. GemStone [CM84, BMO+89] is a commercial system that extended Smalltalk with database features to form one of the first OODBMSs.

Several other systems have provided insights into the development of object-oriented features and have influenced the design of TIGUKAT. These contributions come from Encore [ZW86], Orion [BCG+87, KBC+89, KGBW90], Exodus [CDF+88, CDV88], FAD [BBKV87], LOGRES/ALGRES [CCCR+90], CACTIS [Hud86], CLASSIC [BBMR89], and EMERALD [BHJ+87].

One unconventional approach that has generated some ideas about object existence and references to objects is the formal model proposal by Wand [Wan89]. The philosophy of ontology [Bun77, Bun79] is applied to define the notion of an object. The technique introduces an intriguing philosophical perspective in defining the foundations of a formal object model. An ontological approach has applications in the design of object models because these models are expected to have high levels of abstraction, and the more abstract models become, the more likely it is that philosophical issues come into play.

## 2.2 Object Model Overview

The object model proposed here is founded on a high-level behavioral specification with object uniformity being an integral part of the definitions. The semantics of the TIGUKAT

object model is given by a complete set of definitions and is integrated with an example structural model to clarify its functionality. The model is defined *behaviorally* with a *uniform* object semantics. The TIGUKAT object model is *behavioral* in the sense that access and manipulation of objects occurs through the application of behaviors (operations) to objects, and the model is *uniform* in that every concept modelled has the status of a *first-class object* with well-defined behavior. A purely behavioral semantics, coupled with uniformity, are two major features of the TIGUKAT object model that distinguish it from other models.

The integration of the behavioral model with a structural counterpart illustrates how the behavioral concepts can be organized at a structural level. This defines a complete model that forms a basis for a clean interface to an object storage manager subsystem. The behavioral model of TIGUKAT is integrated with a structural counterpart to form a complete model definition. This is in contrast to other models that concentrate on one or the other. One exception is the model by Beeri [Bee90], which emphasizes the structural model and the integration with a behavioral model is incomplete. It is important to stress that the choice of a structural counterpart is orthogonal to the behavioral specification of TIGUKAT. The only requirement is that the structural component support the full functionality outlined by the behavioral model.

Uniformity in TIGUKAT is more complete than in other models. This is demonstrated in this thesis by uniformly defining a meta-model, a query model, schema evolution policies, and version control as extensions to the base model. Other uniform extensions include a query optimizer [Muñ94], the introduction of temporality [GÖ93], and a transaction manager.

The behavioral model evolves from the definition of several primitives. The primitives form a foundation that supplies the necessary tools from which other constructs such as user-defined and system objects may be created and extended. The primitives follow the same behavior application semantics as any other object because of the uniformity built into the model. That is, the primitive object system evolves within the same forum as other "real-world" objects through the application of behaviors. The primitive objects of the model include: *atomic entities* (i.e, reals, integers, naturals, characters, strings and booleans); *types* for defining common features of objects; *behaviors* for specifying the semantics of operations that may be performed on objects; *functions* for specifying implementations of behaviors over various types[2]; *classes* for automatic classification of objects based on their type[3]; and *collections, bags, partially ordered sets* and *lists* for supporting general, heterogeneous, user-defined groupings of objects.

The primitive type lattice of TIGUKAT is shown in Figure 2.1 with type `T_object` as the root and type `T_null` as the base. The type `T_null` binds the type lattice from the bottom (i.e., most defined type), while `T_object` binds it from the top (i.e., least defined type). `T_null` is a primitive type defined to be a subtype of all other types. `T_null` is introduced to provide, among other things, error handling and null semantics for the model. For example, there is an object `null` that is an instance of `T_null` and can be returned by behaviors that have no other result. This is the case because `T_null` (and therefore `null`) supports the behaviors of all other types and can be substituted as the result of any behavior. In a similar way, instances `undefined`, `dontknow` and other error objects of type `T_null` can be defined.

Figure 2.1 illustrates the subtyping relationships of the primitive type system. Each

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

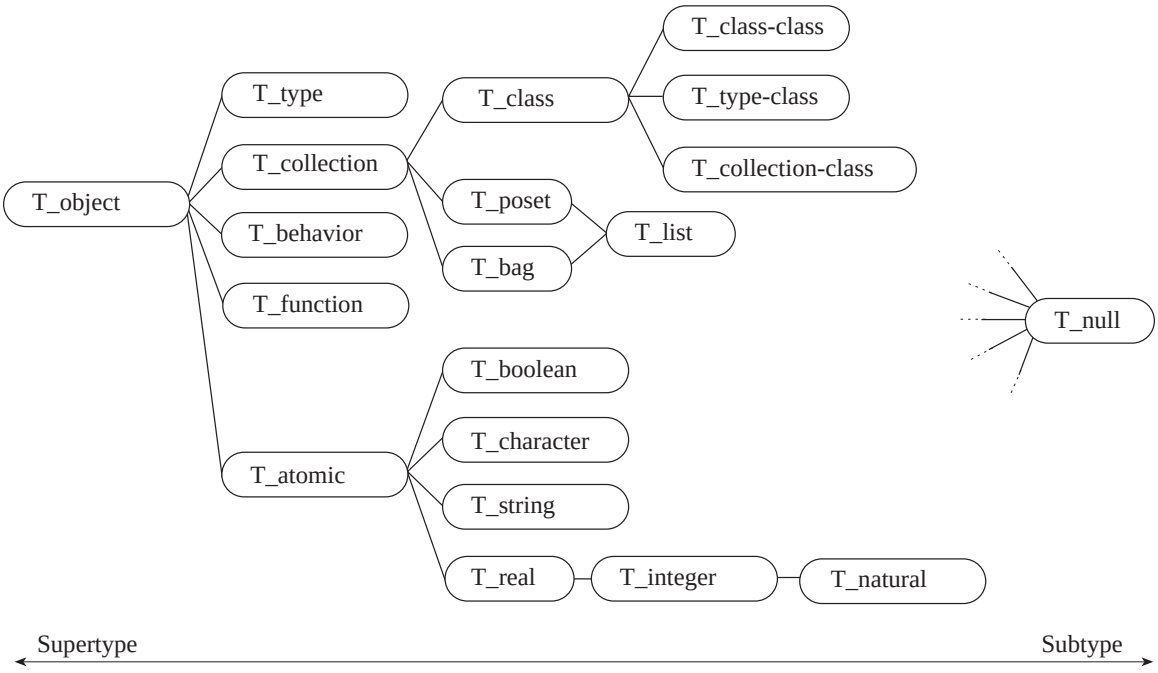



Figure 2.1: Primitive type system $\mathcal{T}$.

oval in the figure represents a primitive type and the edges between the ovals denote the well-known notion of *subtyping* (i.e., the type `T_type` is a subtype of type `T_object` and so on). Types are identified by an appropriate reference given within each oval. The semantics of the types in Figure 2.1 are formally addressed in the following sections. A brief overview is given here.

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

The type structure of Figure 2.1 is referred to as the *primitive type system* $\mathcal{T}$. Each type in $\mathcal{T}$ is associated with a unique corresponding primitive class object. Each primitive class contains instances of other primitive objects (e.g., primitive behaviors, functions, collections, strings, etc.). Types define primitive behaviors and these behaviors are associated with primitive functions that implement the semantics of the behaviors. The union of the types in $\mathcal{T}$ with the set of all primitive classes, behaviors, functions and other instance objects is defined as the *primitive object system* $\mathcal{O}$ of TIGUKAT.

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

An *object* is an abstraction for encapsulating information into a single entity that may be operated on by behaviors. An object is only accessible through the set of behaviors defined

by the type of that object, which constitutes the *interface* of the object; this is known as the *encapsulation property*. Furthermore, TIGUKAT supports *strong object identity* [KC86], meaning every object has a unique, immutable identifier associated with it, which distinguishes the object from all others.

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

Throughout this thesis, a functional programming environment is assumed as a global scope. The following prefix notations and font variations are adopted in this scope to denote object references of the various primitive kinds.

`T_name` is a type object reference,

**C_name** is a class object reference,

**L_name** is a collection object reference,

*B_name* is a behavior object reference,

*F_name* is a function object reference, and

name is some other application specific reference.

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

The means for defining the characteristics of objects (i.e., a *type*) is separated from the mechanism for grouping instances of a particular type (i.e., a *class*). A *type* is used to specify the structure and behavior of objects. The type serves as an information repository (template) of characteristics common among all objects that conform to that particular type. As shown in Figure 2.1, types are organized into a lattice structure using the notion of *subtyping*, which promotes software reuse and incremental type development.

A *class* ties together the notions of *type* and *object instance*. A *class* is a supplemental, but distinct, construct from a type that is responsible for managing the instances of a particular type. The entire collection of objects of a particular type is known as the *extent* of the type. This is separated into the notion of *deep extent* that refers to all objects of a

given type, or one of its subtypes, and the notion of *shallow extent* that refers only to those objects of a given type without considering its subtypes.

Objects of a particular type cannot exist without an associated class and every class is uniquely associated with a single type. Thus, a fundamental notion of TIGUKAT is that *objects* imply *classes* which imply *types*. Another unique feature of classes is that object creation occurs only through a class using its associated type as a template for the creation. Defining object, type and class in this manner introduces a clear separation of these concepts. This separation is important during type inferencing in the algebra which manipulates type objects into new subtype relationships and need not be concerned with the overhead of classes. Furthermore, many object-oriented systems include *abstract types* whose sole purpose is to serve as place-holders for common behaviors of subtypes and are never intended to have any instance objects. In this case, there may be no reason to manage classes for abstract types, because there are no instances of these types. The separation is also important to uniformly define the model within itself, which builds the foundation for features such as reflective capabilities.

In addition to classes, *collections* (essentially sets) are defined as a more general, user-defined, grouping construct. A *collection* is similar to a *class* in that it groups objects, but it differs in the following respects. First, object creation cannot occur through a collection; object creation occurs only through classes. This means that collections only form user-defined groupings of existing objects. Second, an object may exist in any number of collections, but it is a member of the shallow extent of only a single class. Third, the management of classes is *implicit* in that the system automatically maintains classes based on the type lattice whereas the management of collections is *explicit*, meaning that the user is responsible for their extents. Finally, a class groups the entire extension of a single type (shallow extent) along with the extensions of its subtypes (deep extent). Therefore, the elements of a class are homogeneous up to inclusion polymorphism. On the other hand, a collection may be heterogeneous in the sense that it can contain objects that may be of different types that are not in a subtype relationship with one another. A collection of objects is denoted using the standard set notation as $\{o_1, o_2, \ldots, o_m\}$ where each of the $o_i$ is an object reference.

Basic collections are supplemented with definitions for *bags* (type `T_bag`), which are collections that allow duplication of elements, *partially ordered sets* (type `T_poset`), which are collections with an ordering relation defined between pairs of elements, and *lists* (type `T_list`), which are collections that combine the properties of bags and posets by allowing both duplication and ordering of its elements.

These aggregate types may be specialized by subtyping the general types. One form of specialization is to define a subtype that restricts the elements of its instances to be of a particular type. Parameterization is used to denote this form of refinement. The syntax is given as `T_collection`$\langle$`T_X`$\rangle$, `T_bag`$\langle$`T_X`$\rangle$, `T_poset`$\langle$`T_X`$\rangle$ and `T_list`$\langle$`T_X`$\rangle$ where `T_X` represents some other type specification. This restricts the members of the aggregate type to be compatible with the type `T_X`[4]. For example, `T_collection`$\langle$`T_person`$\rangle$ represents a collection whose members are objects that are compatible with the type `T_person`. The notion of type compatibility is formally defined in Section 2.4.4.

In TIGUKAT, type `T_class` is a specialization (subtype) of `T_collection`, which introduces a clean semantics between the two and allows the model to utilize both grouping

[4]The notations `T_collection`, `T_bag`, `T_poset` and `T_list` are abbreviations for the parameterized notations `T_collection`$\langle$`T_object`$\rangle$, `T_bag`$\langle$`T_object`$\rangle$, `T_poset`$\langle$`T_object`$\rangle$ and `T_list`$\langle$`T_object`$\rangle$ respectively.

constructs in an uniform manner. For example, the targets and results of queries are typed collections of objects and since classes are a specialized collection, they may be used in queries as well. This approach provides great flexibility and expressiveness in formulating queries and gives *closure* to the query model, which is often regarded as an important feature [Bla91, YO91].

The remaining subtypes of `T_class` make up the *meta* type system. These include the types `T_class-class`, `T_type-class` and `T_collection-class`. Their placement within the type system directly supports the uniformity definition of the model. Section 2.4.6 describes the semantics of the behaviors defined on these types and the architecture of the corresponding class and instance structure of the types. This meta-model (within the model) is the foundation of reflective capabilities which is addressed in Chapter 4.

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

The semantics of every operation on an object is specified by a behavior defined on its type. A function implements the semantics of a behavior. The implementation of a particular behavior may vary over the types that support it. However, the semantics of a behavior remains consistent over all types supporting that behavior. There are two kinds of implementations for behaviors: *computed functions* and *stored functions*. A *computed function* consists of runtime calls to executable code and a *stored function* is a reference to an existing object in the objectbase. The uniformity of TIGUKAT considers each behavior application as the invocation of a function, regardless of whether the function is stored or computed.

A semantic description of a behavior may be quite complex. One approach is to define the functionality of behaviors using a *denotational semantics* [Sto77, All86, Sch88, CP89]. A simpler technique, common in many other models, is a *signature* expression. A signature defines a name (reference) used to invoke the behavior, the types of the arguments to the behavior, and the type of behavior's result. Signatures are useful and necessary for describing the semantics of behaviors, but they are inadequate for characterizing the full semantics. Describing the full semantics of behaviors is a difficult problem. In this thesis, it is assumed that a proper semantic specification mechanism exists. Only signatures are defined for behaviors to give some indication of their semantics. A more complete semantic specification is part of the future research. It should be noted that the extensibility of TIGUKAT allows the complete specification to be easily added when it is finally defined.

Functions are objects that include source and implementation components. The source component is a human readable definition of the function's operation (behavior) usually written in some object-oriented programming language, but can additionally include English commentary and further semantic descriptions. The implementation component of a function consists of executable code if the function is computed, or is simply a reference to a particular result object if the function is stored. The functional approach adopted by TIGUKAT benefits from the significant amount of research that has been done in the areas of functional programming languages and functional theory such as the lambda calculus [Bar81, Rev89] and category theory [Pie88, LS86].

As a supplement to the behavioral model, a structural model maps behavior definitions into a representation that is consistent with a storage manager level interface. The structural

level makes a cleaner distinction between atomic entities of the system and the structured objects (abstract data types (ADTs)) that are constructed from them. At this level, the domains of the atomic types are mapped into the semantics of *values*, which serve as the identity and state of atomic objects and gives them the properties of immutability.

From the user's perspective, the domains of atomic types can be assumed to exist and can be manipulated using the behaviors defined by the atomic types. In other words, they are seen as constants in the model. Exactly how this abstraction is maintained is implementation dependent. Languages for the model must provide a syntax to specify references to the constants of the atomic types. The act of specifying a constant (in a query language for example) from an atomic domain is interpreted as a request to return an object representing that constant. An implementation can either scan the objectbase and return the corresponding object constant if it exists, or create a new one if it does not. For efficiency reasons, an implementation should physically allow many duplicate instances of atomic objects, but maintain the abstraction of uniqueness and immutability. This approach is followed by the implementation of TIGUKAT [Ira93].

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

The structural aspects of the model are clarified by the introduction of an object graph representation defined in Section 2.5. An object graph is used to illustrate the structure and contents of an objectbase with application specific and primitive system objects stored uniformly. The nodes of an object graph correspond to the atomic values and abstract objects in an objectbase, while the edges represent relationships (defined as behaviors) between the various nodes (i.e., objects).

Each concept introduced in this section, although related, has a separate role in the model and each has a distinct semantics. In the sections that follow, these concepts are discussed in more detail and their semantics are formalized. First, a simplified geographic information system (GIS) is defined as a running example used throughout the thesis to demonstrate results.

## 2.3 Example Objectbase

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

technology. Specifically, a GIS requires the following capabilities:

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

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

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

A type lattice for a simplified GIS is given in Figure 2.2. The example is sufficiently complex to illustrate the functionality of the model presented in this thesis, yet simple enough to be understandable without an elaborate discussion. The example includes the root types of the various sub-lattices of the primitive type system $\mathcal{T}$ to illustrate their relative position in an extended application lattice. The additional types defined by the GIS example include:

1. Abstract types for representing information on people and their dwellings. These include the types `T_person`, `T_date`, `T_dwelling` and `T_house`. Note that `T_date` is a new atomic type introduced by the application which is used to represent dates in a form acceptable to the application.
2. Geographic types to store information about the locations of dwellings and their surrounding areas. These include the type `T_location`, the type `T_zone` along with its subtypes which categorize the various zones of a geographic area, and the type `T_map` which defines a collection of zones suitable for displaying in a window.
3. Displayable types for presenting information on a graphical device. These include the types `T_displayObject` and `T_window` which are application independent and the type `T_map` which is the only GIS application specific object that can be displayed.

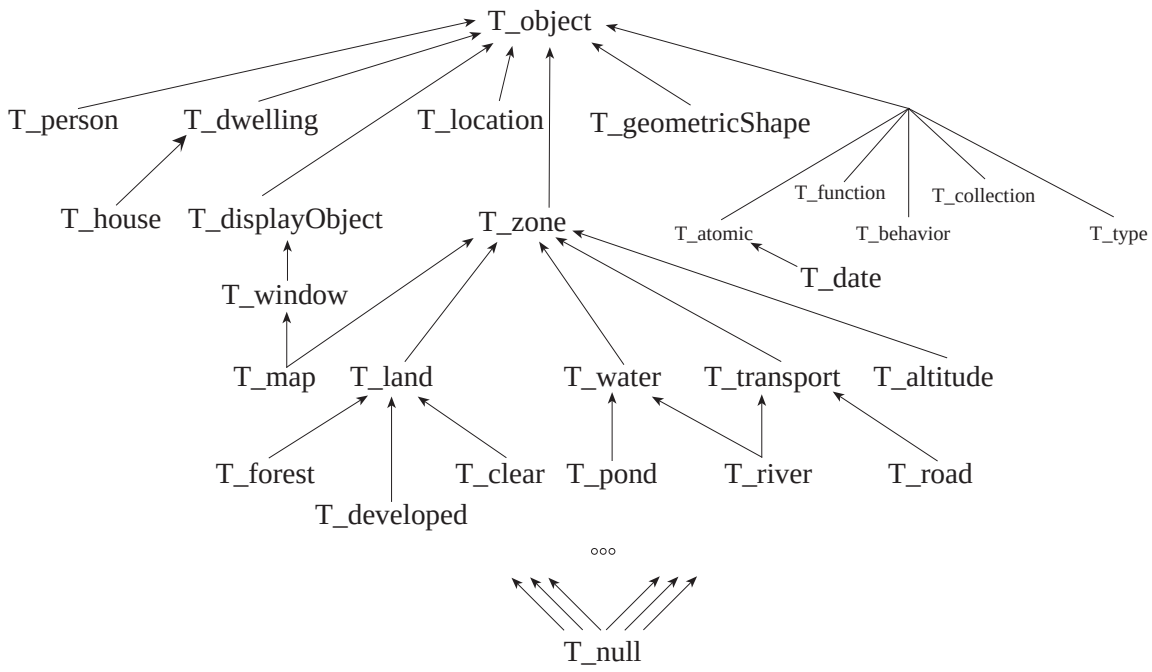



Figure 2.2: Type lattice for a simple geographic information system.

4. A type `T_geometricShape` which defines the geometric shape of the regions representing the various zones. For the purposes of this thesis, only the general type is used, but in more practical applications this type would be further specialized into subtypes representing polygons, polygons with holes, rectangles, squares, splines, and so on.

Table 2.1 defines the signatures of the GIS specific types in the lattice of Figure 2.2. The semantics of these behaviors will be clarified throughout the remainder of this thesis. Furthermore, the signatures for the types of the primitive type system $\mathcal{T}$ will also be developed.

## 2.4 The Behavioral Model

In this section, the behavioral aspects of the TIGUKAT object model are emphasized. The high-level abstract functionality of the model is described and the presentation follows a formal approach. At times structural aspects are addressed to clarify certain points raised, but these digressions are kept to a minimum. A full integration of the behavioral model with an example structural counterpart is delayed until Section 2.5.

### 2.4.1 Atomic Types, Classes and Objects

Most data models include a set of basic primitive types referred to as *atomic types*. The common types `T_boolean`, `T_character`, `T_string`, `T_real`, `T_integer` and `T_natural` are included as part of the primitive model definitions. The collection of atomic types are referred to as the *atomic type pool*. Other types may be easily added to this collection through the operation known as *subtyping*[5]. For example, the GIS application schema of

[5]Subtyping is formally defined in Section 2.4.4.

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

[a] Behavior was refined from supertype T_dwelling.

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

Section 2.3 extends the atomic types with the type T_date.

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

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

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

since these types are universally known abstractions. The full behavioral specification of these types and their objects is defined in the implementation of the model [Ira93].

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

The atomic types `T_real`, `T_integer`, `T_natural`, and `T_string` represent an infinite domain of atomic objects. A finite objectbase is assumed and therefore all classes within the model must be finite. To deal with this, there are two kinds of classes provided by the model. The one kind is called an *explicit class* because it explicitly manages its shallow extent and computes its deep extent by recursing over the shallow extents of its subclasses. The second kind is called an *implicit class* because the shallow extent is not explicitly stored, but rather is implied from the contents of the objectbase. In other words, the shallow extent of an implicit class is the (finite) collection of objects in the objectbase that belong to the class. The shallow extent of an implicit class can be computed by scanning the objectbase and returning the objects that have the same type as the type associated with the class.

Most classes are explicit classes. However, the classes for the atomic types `T_real`, `T_integer`, `T_natural` and `T_string` are implicit. Moreover, they are special in the sense that there is a built-in mechanism for creating the constants of the these classes. The act of writing down a constant of one of these classes (in a query for example) can be thought of as a request to return an object representing the constant, creating a new one if necessary. For example, the class **C_integer** is initialized with the object zero and by using the *B_succ* and *B_pred* behaviors on this object, any integer object can be theoretically created and returned. The act of writing down the integer constant 2, can be thought as a request to apply *B_succ* to the object zero and then to apply *B_succ* to the result. This either returns the existing object representing the integer 2, or creates a new one. This is an assurance that there is only one integer 2 in the objectbase. Any intermediate objects created along the way that are not stored in the objectbase are deleted. The reals and naturals have a similar semantics. The *B_succ* and *B_pred* behaviors on reals are limited to the precision of reals on a particular system. The class **C_string** is initialized with the empty string and string representations of all the characters of which there are a finite number. With these initial strings and the concat function any string can be created and returned. The act of writing down the string "joe" can be thought of as a request to apply *B_concat* to the string objects "j" and "o" and then to apply *B_concat* to the result and the string object "e". Of course, in the implementation of TIGUKAT [Ira93] it is not actually done in this way. Instead the "native" domains of the implementation language are used. The above is just a formal model that is consistent with the uniformity aspects of the object model.

The instances of the atomic types serve as both state and identity. For example, the atomic type `T_integer` draws from an infinite domain of objects whose elements serve as the identity and state of their existence. An integer reference 5 refers to an integer object

whose identity and state is the universally known abstraction of the integer 5. There is only one 5, there always has been and there always will be. Note that this does not restrict users from establishing additional references to the integer 5 such as five or V. The same argument holds for all types in the atomic type pool.

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

### 2.4.2 The Behavior and Function Primitives

Two fundamental concepts of TIGUKAT are *behaviors* and the *functions* (known as *methods* in other models) that implement them.

A *behavior* is an object that performs an operation on other objects and produces an object as a result. Behaviors are defined on types and are applicable to the object instances that are compatible with that type. Types wanting to provide a particular behavior must define that behavior object as part of their interface or have the behavior inherited through subtyping. Each behavior includes a semantic expression of its functionality. Equality for behaviors is refined to incorporate equality of semantic expression.

Behaviors are separated from their implementations (functions/methods). The benefit of this approach is that common behaviors over different types can have a different implementation in each of the types. This is referred to as *overloading* the behavior, meaning that the implementation of the behavior may vary depending on the type of the object to which it is applied. This gives the model the ability to *dynamically bind* implementations to behaviors at run time (known as *late-binding*). Overloading and late-binding are recognized as major advantages of object-oriented computing.

The semantics of every operation on an object is specified by a behavior defined on its type. A function implements the semantics of a behavior. In other words, a function provides the *operational* semantics of the behavior it implements. Due to overloading, the implementation of a particular behavior may vary over the types that support it. Nonetheless, the semantics of the behavior remains consistent over all types supporting that behavior. There are two kinds of implementations for behaviors. A *computed function* consists of runtime calls to executable code and a *stored function* is a reference to an existing object in the objectbase. Stored functions eliminate the need for instance variables, which limit reuse [WBW89b]. The uniformity of TIGUKAT conceptually transforms each behavioral application into the invocation of a function, regardless of whether the function is stored or computed. This allows designers to concentrate on responsibilities rather than data attributes [WBW89a].

Behaviors are instances of the type `T_behavior` and functions are instances of the type `T_function`. The standard arrow ($\rightarrow$) notation is used as a syntactic representation for functions and curry multiple argument function specifications. In this way, a wide variety of other representations are supportable. A general function specification is of the form

$\mathcal{A} \rightarrow \mathcal{R}$ where $\mathcal{A}$ represents the argument type expression of the function and $\mathcal{R}$ represents the result type. In general, the argument and result type expressions may consist of any other type specifications (including function specifications).

Functions as implementations of behaviors are unary (i.e., curried) in the sense that they have an argument expression $\mathcal{A}$ consisting of a single type that is compatible with the type the function is expected to be applied to (i.e., the type defining the behavior that is using the function as an implementation). The result expression $\mathcal{R}$ of a function denotes the result type of the object returned from the execution of that function.

Types have an *extent* of objects that are grouped by a corresponding *class*. Types define a set of behaviors that are applicable to the objects in its extent (i.e., its class)[6]. Behaviors represent the only means of accessing and manipulating objects in a class, and functions are the objects that implement these behaviors.

The semantic definition of a behavior can be specified in many ways. Some examples include using the code that implements the function as a specification, or using an informal English description, or possibly a more formal denotational specification [Sto77, All86, Sch88, CP89]. A simple method, common among other models, is the use of a *signature* expression for representing the meaning of a behavior. A signature defines for a behavior a name (reference) used for behavior application, the types of its arguments, and the type of its result. Signatures are useful and necessary for describing behaviors, but they are inadequate for characterizing the full semantics of behaviors. In this thesis, it is assumed that a proper semantic specification mechanism for behaviors exists and that equality testing on behavioral semantics operates reliably. There is a behavior *B_semantics* (denoted $[\![\ ]\!]$) defined on the type `T_behavior` that returns the complete semantic specification of a behavior. For example, applying *B_semantics* to a behavior, say $b$ (denoted $[\![b]\!]$), returns the semantic specification of $b$. Currently, only signatures are defined for behaviors to give some indication of their semantics. As part of the future research, a more complete specification of behavior semantics is being developed.

A *signature* specification consists of several elements. It has a *name* used to invoke the behavior, it has *argument types*, and it has a *result type*. The name for invoking a behavior is given by a standard string, and the argument types and result type are one of the types available to the user. Since behaviors are always defined on a particular type, and types can be function specifications, a behavioral specification may be thought of as a function with a single argument (an object of the type it is defined on) and a single result (an object of the type specified as the result, which may be a function). Formally, the representation of a signature is as follows:

**Definition 2.1** *Signature* $(b : R)$*:* A *signature* is a partial specification of behavior. It is denoted as $b : R$ and consists of a *name* $(b)$ that is used to apply the behavior to an object and a *result type* $(R)$ that specifies the type of the object resulting from the application of the behavior. The argument types of $b$ may be embedded as a curried function expression in $R$. □

Several primitive behaviors are defined on the type `T_behavior` for the purpose of accessing and manipulating behavior objects. The behaviors relating to signature expressions include the following:

*B_name* : `T_string` to access the name of a behavior,

[6]The relationships between *type, class* and *extent* are formally defined in Sections 2.4.4 and 2.4.5.

*B_argTypes* : `T_type` → `T_list`⟨`T_type`⟩ to return the list of argument types of a behavior for a particular type, and

*B_resultType* : `T_type` → `T_type` to return the result type of a behavior for a particular type.

The name of a behavior must be unique over types that define the behavior and are in a subtype relationship with one another. However, the result type and argument types of a behavior may vary as long as they are compatible with the types of the behavior in all supertypes that define that behavior. Type compatibility and subtyping is discussed in Section 2.4.4.

Behaviors are applied to objects. The object receiving the behavior is explicitly specified. This is similar to the classical or message-based object model outlined in [FKMT91]. The dot notation $r.b(a_1, \ldots, a_n)$ is used to denote the application of behavior $b$ to the receiver object $r$ using objects $a_1$ through $a_n$ as arguments. If no arguments are required, then the application simplifies to $r.b$. The result of this behavior application is a reference to an object in the extent of the result type specified by the signature of $b$. Since the result is a reference to an object, it may have other behaviors applied to it. Thus, the behavior application itself may be thought of as an object reference.

For example, consider the following signature defined on the type `T_person` in Table 2.1:

*B_residence* : `T_dwelling`

Applying *B_residence* to an object of type `T_person` results in the execution of the function object associated as the implementation of this behavior, which returns an object that is compatible with the type `T_dwelling`. If an expanded signature specification as in [SÖ90a] were used, the signature would be written as follows:

*B_residence* : `T_person` → `T_dwelling`

In TIGUKAT, a behavior must be defined on a type before being used and a behavior can be defined on many types[7]. Therefore, the "`T_person` →" part of the signature is omitted and is derived from the type of the receiver object instead. Consider an object Sherry as an instance of type `T_person`. The application of *B_residence* to Sherry is denoted as Sherry.*B_residence*. This invokes the function associated as the implementation of *B_residence* in type `T_person` and the result is a reference to an object in the extent of `T_dwelling`. A subtype of `T_person`, such as `T_student` for example, may have a different implementation of *B_residence*, but the behavior is semantically equivalent in both types. The signature partially supports this semantic equivalence.

An optional representation for behavior application is function invocation denoted as $b(r, a_1, \ldots, a_n)$ where one of the arguments (e.g., the first one) is special in the sense that it denotes the receiver object. This representation is equivalent to the dot notation. Referring back to a previous example, applying the behavior *B_residence* to the object Sherry using function invocation is specified as *B_residence*(Sherry). Function invocation represents an optional representation for behavior application and has a direct translation to the dot notation by moving the receiver object out of the argument list to the position before the dot.

[7] In Section 2.4.4, a behavior (*B_interface*) is defined that, when applied to a type, returns the collection of behaviors defined on that type.

In order to associate a function with a behavior for a particular type, the type `T_behavior` defines the following behavior:

$$B\_associate : \texttt{T\_type} \rightarrow (\texttt{T\_function} \rightarrow \texttt{T\_behavior})$$

This behavior accepts a type and a function as arguments. For example, the behavior application $b.B\_associate(T, f)$ will associate function $f$ with behavior $b$ in type $T$. Now, whenever $b$ is applied to an object of type $T$, the function $f$ will be invoked. Other behaviors defined on `T_behavior` include *B_implementation* for accessing the function (implementation) associated with a behavior for a particular type, *B_defines* to get a collection of types that the behavior is defined on, and *B_apply* to apply a behavior to an object with a list of arguments.

Functions have behaviors such as *B_source* and *B_executable* for accessing the source code and executable load module of a function. The source component is a human readable definition of the function's operation most likely written in some object-oriented programming language, but can include things like commentary and formal semantic specification. The implementation component of a function consists of executable code, in the case of a computed function, or is simply a reference to a particular result object, in the case of a stored function. The functional approach adopted by TIGUKAT benefits from the significant amount of research that has been done in the areas of functional programming languages and functional theory such as the lambda calculus [Bar81, Rev89] and category theory [Pie88, LS86]. Category theory is a pure theory of functions consisting of objects and *morphisms* (essentially functions) that map one object to another. In the spirit of category theory, the TIGUKAT object model is based on objects and behaviors, which act as a mapping from one object to another. The identity, composition, and associative properties of morphisms in category theory with appropriate modifications also hold for behaviors in TIGUKAT. The lambda calculus is a functional language with a simple syntax for specifying parameterized functions and function application. Lambda expressions are used in developing the predicates of the TIGUKAT algebraic operators to define the application of behaviors within queries.

The type `T_function` defines the following additional behaviors to deal with function properties and function application: *B_argTypes* for accessing the list of argument types of the function, *B_resultType* for accessing the result type of the function, *B_compile* for compiling the source code, and *B_execute* for executing the function.

In Section 2.4.4, *subtyping* (also referred to as *behavioral inheritance*) is defined as a reuse mechanism for the behaviors of types. A behavior is *inherited* in a subtype $\texttt{T\_}\tau$ if it is defined in a supertype of $\texttt{T\_}\tau$. Otherwise the behavior is *native*. Behavioral inheritance has no implication on the reuse of implementations. That is, inherited behaviors do not necessarily borrow any implementation from their supertypes (although this may be the default). For this reason, a separate reuse mechanism for implementations called *implementation inheritance* is defined. A behavior implementation (i.e., function) is *inherited* in a type if the behavior that it implements is inherited, and if the implementation is the same function as the implementation of that behavior in the supertype. Otherwise the implementation is *redefined* (or *overridden*).

The TIGUKAT object model supports *multiple inheritance* (i.e., *multiple subtyping*). Multiple subtyping means that a type can be a direct subtype of several other types. This requires a conflict resolution policy to choose an implementation when inheriting semantically equivalent behaviors with different implementations from several types. TIGUKAT

can support different conflict resolution policies because conflict resolution is not part of the base model definition. One approach is that used in Modular Smalltalk [WBW88a] where it is an error for a type to inherit two different implementations (i.e., function's) for the same behavior. The error can be avoided by explicitly redefining the implementation for that behavior. One of the two conflicting functions can be chosen as the redefined implementation. In TIGUKAT, a separate mechanism to resolve inheritance conflicts between instance variables is not required because there is no concept of instance variables. They are handled as behaviors with stored functions as the implementation and stored function conflicts are resolved in the same manner as computed function conflicts.

Conflict resolution is unnecessary for behavioral inheritance because this deals with semantics of behaviors, which are preserved over type boundaries, while the implementation of these semantics may differ over conflicting types. The inheritance mechanism, as well as the conflict resolution policy, is implementation dependent and not part of the base model definition.

### 2.4.3 The Object Primitive

An *object* is a fundamental primitive in TIGUKAT because the conceptual level of the model deals uniformly with objects. In Section 2.2, the concept of an object as an abstraction for encapsulating information and behavior into a single entity is described. The encapsulated portion of an object is referred to as its *state*, which is accessible only through a set of behaviors defined on the type for that object. The state carries the information content of the object. In addition to state, each object has an *identity*, which serves as a unique, immutable system managed identity for the object throughout its existence. Thus, the model considers an object as a pair consisting of an identity and a state.

**Definition 2.2** *Object:* An *object* is defined as the pair (*identity, state*) where *identity* is the unique, immutable identity of the object and where *state* is the information carried by the object. □

An unique *object identifier* (or *oid*) is associated with an object upon its creation and persists with that object throughout its lifetime. An *oid* serves as the identity of an object. In TIGUKAT, objects are composed of other objects because the result of behaviors applied to objects are objects themselves. Conceptually, every object in TIGUKAT is a *composite* object. By this, it is meant that every object has references/relationships (not necessarily implemented as pointers) to other objects. For example, even integers have behaviors that return objects, but they are not implemented as pointers.

If one considers the domain of all objects as the collection of pairs consisting of all possible combinations of identity and state, then an unwanted inconsistency arises. This domain will contain objects with the same identity, each associated with different states. This is obviously inconsistent because there is a single identity attempting to identify several semantically distinct states.

To eliminate this inconsistency, the following definition of a consistent set of objects is formed, which gives a basis for objectbase construction. The definition assumes the existence of an operation $oid(o)$ that takes an object $o$ as input and returns the internal identity (*oid*) of the object as its result. Note that this operation could be defined as a behavior that uniquely maps all objects (past and present) to the integers.

**Definition 2.3** *Consistent Object Set (conset):* A set of objects $O$ constitutes a *consistent object set* (*conset*) if and only if $\forall o_i, o_j \in O$, $oid(o_i) = oid(o_j) \Longrightarrow o_i = o_j$, where $\Longrightarrow$ denotes logical implication. □

The definition of a consistent object set adheres to the notion of strong object identity [KC86]. That is, every object in a *conset* has an internal identifier that is distinct from all others in the *conset*. This feature gives each object a unique existence within a *conset* and provides an unambiguous association with the state of that object. Note that with this definition two separate objects may share the same state information. This is reasonable since there are many examples of real-world objects (printed maps of a city to name one) that have identical properties, yet are distinguishable objects in their own right.

The primitive object system $\mathcal{O}$ is a *conset* of objects as laid out by the following axiom. The remainder of the model development is within the bounds of a *conset*.

**Axiom 2.1** The primitive object system $\mathcal{O}$ is a *conset*. □

Some argue [SRL+90, Bee90] that object identities should have the option of being either system or user assigned. In the TIGUKAT model, all object identities are maintained automatically by the system without any user involvement. This is in keeping with the notion of strong object identity and has additional benefits when it comes to reconciling the components of distributed object bases and the variable interpretations that may exist among them. Nevertheless, user defined identities can be supported in the presence of strong object identity. They are possible through application specific interpretations. For example, a user may choose to recognize one of the behaviors of an object (e.g., *B_social_insurance_number*) as an identifier for that object and all other objects like it. The TIGUKAT model places no restrictions on this kind of customized interpretation.

Object existence, access, and manipulation in TIGUKAT is based on the notions of *reference*, *scope* and *lifetime*. This is similar to other model proposals [Sny90, Ken90b, FKMT91] in that the only user expressible form of an object is a *reference* within a particular *scope*. A *scope* defines the visibility, access paths and lifetime of object references. A *reference* may be thought of (and actually implemented) as a *pointer* (or *handle*) to an object, which in turn leads to the object's *identity* and *state*. The notation $R_i@S_i$ denotes an object reference $R_i$ in scope $S_i$. This is shortened to $R_i$ when the scope is obvious or immaterial. The $R_i$ component is a reference name adhering to the prefix notation outlined in Section 2.2. The *lifetime* of an object is independent of the *lifetime* of a reference to that object in a particular scope. That is, when a reference disappears, the object being referenced does not necessarily disappear, but may persist past the lifetime of the reference. However, if an object no longer has any references (system or user) maintaining its existence, then the object should be selected as a candidate for storage reclamation. From the database perspective, there is also the issue of explicit deletions. Deleting an object within a particular scope should guarantee that the object is no longer visible in that scope, but how this affects its visibility within other scopes concurrently referencing the object is part of a concurrency control mechanism and is not addressed in the primitive model. The semantics of object deletion in light of schema evolution is addressed in Chapter 5. The semantics of storage reclamation is outside the scope of this thesis. Figure 2.3 is an example of an object reference model and illustrates the relationships among scope, reference, identity and state.

In Figure 2.3 there are the two scopes $S_1$ and $S_2$. The scope $S_1$ could be an application programming environment while $S_2$ may be an interactive query processor. The exact

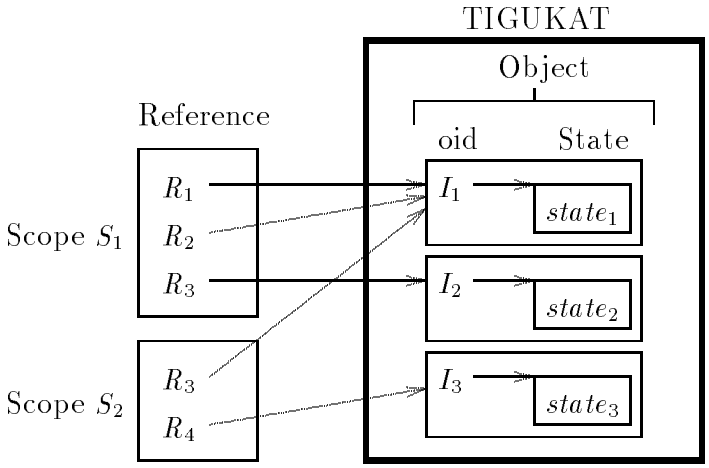



Figure 2.3: An object reference example.

semantics of the scoping rules is defined by the application accessing the objectbase and may vary over applications. Within scope $S_1$ there are the three object references $R_1, R_2$ and $R_3$. References $R_1$ and $R_2$ refer to the same object identified by $I_1$, and $R_3$ refers to the object identified by $I_2$. Within scope $S_2$ there are two object references, $R_3$ and $R_4$. In this scope, $R_3$ refers to the same object as $R_1$ and $R_2$ do in scope $S_1$, and $R_4$ refers to the object identified by $I_3$ which is unrelated to scope $S_1$. This example shows the various mappings from references over scopes $S_1$ and $S_2$ to their associated objects. The heavy dark line around the objects indicates the boundary of the TIGUKAT object model. If, for example, one considers everything within the boundary as being persistent (i.e., assuming a persistent object store), then if a reference or an entire scope disappears, the objects will persist (provided they have other references to them and won't be garbage collected). When referring to objects, the terms "object" and "object reference" are used interchangeably.

Operations on objects are performed through behaviors. Since object access is specified through references, behaviors are applied to object references within a particular scope which in turn applies the behavior to the actual objects and returns a reference to the resulting object. There are several primitive behaviors defined on type `T_object` that are inherited by all other types because the lattice is rooted at `T_object`. These behaviors represent the fundamental operations on objects.

A basic requirement in the model is a mechanism to determine if two object references are actually referring to the same object or different objects. Therefore, the following *equality* behavior is defined on `T_object`, which makes it applicable to all objects.

**Behavior 2.1** *Object Equality, (B_equal* : `T_object` → `T_boolean`*)* (=)*:* For any two object references $R_i$ and $R_j$ in their respective scopes $S_i$ and $S_j$, the result of applying $R_i@S_i.B\_equal(R_j@S_j)$ is true if and only if $R_i@S_i$ and $R_j@S_j$ map to the same object identity in the domain of object identities (i.e., $oid(R_i@S_i) = oid(R_j@S_j)$). Since the model development is within the bounds of a *conset*, the states of the objects must also be equal. The infix binary relation operator "=" is used as a shorthand for *B_equal*, and the above behavioral application of *B_equal* can be expressed as $R_i@S_i = R_j@S_j$. Similarly, the inverse relation $\neq$ is defined to test for inequality. The result of equality is an object reference to an atomic boolean object true or false. Object equality/inequality is reflexive, symmetric and transitive. □

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

Table 2.2: Object equalities of Figure 2.3.

Table 2.2 lists the equalities/inequalities that result in **true** among the references of Figure 2.3 over the two scopes. The first column shows the equalities in scope $S_1$, the second in scope $S_2$ and the third lists the equalities over both scopes.

This is the only kind of equality the primitive model defines. It is quite strong in that the only way two object references are considered equal is if they actually refer to the same object identity. This notion of object equality is the same as "identity equal" defined in [KC86] or "0-equality" defined in [LRV88]. At this level, there are no notions of *shallow* or *deep* equality found in other models [KC86, LRV88, Osb88] or extended versions of these that determine equality at various levels [SZ90]. These notions can be defined as identity equivalence relationships on the behavioral characteristics of objects and therefore should be left to customized interpretations at the behavioral level rather than being part of the primitive model definition. For example, the model may provide the classical shallow and deep equivalence through behaviors that evaluate and determine the equivalence of objects based on the identity equivalence of their component behaviors. This is strictly a design decision that should be left for the implementation phase of a particular system. Dayal [Day89] also makes this argument by stating that there are many notions of equality and those other than "identity equality" are best left for the "customizers" of the model to define the ones that are of most utility to them. For example, equality for behaviors is specialized to mean semantic equality, and equality for atomic objects is specialized to mean value equivalence.

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

Objects in TIGUKAT are strongly-typed. This means that each object is uniquely associated with a particular type, which defines the object's full semantics. Thus, object implies a type (*object* $\Longrightarrow$ *type*). A type defines the behaviors applicable to the objects of that type. It is important in type-checking and query processing to know the type of an object [SÖ90b] (or a conformance of types for an object). Therefore, a behavior on objects is defined that returns the type of the object. We say that every object *maps to* a particular

type. The *B_mapsto* behavior is defined on the type `T_object` making it applicable to all objects.

**Behavior 2.2** *Maps to (B_mapsto :* `T_type`*)* ($\mapsto$)*:* For an object reference $o$, the behavior application $o.B\_mapsto$ is defined to be the singleton type object reference $\texttt{T\_}\tau$ that represents the type of object $o$. The notation $o \mapsto \texttt{T\_}\tau$ denotes that object $o$ *maps to* type $\texttt{T\_}\tau$ (i.e. $(o \mapsto \texttt{T\_}\tau) \Longrightarrow (o.B\_mapsto = \texttt{T\_}\tau)$). □

For example, if the object Sherry is an instance of the type `T_person`, then the following behavior application returns the type of Sherry, which is `T_person`:

$$\text{Sherry}.B\_mapsto$$

Using the symbolic notation, the behavior application is specified as follows:

$$\text{Sherry} \mapsto \texttt{T\_person}$$

Extending this uniformly to types, the behavior application `T_person`.*B_mapsto* returns the type object `T_type` and `T_type`.*B_mapsto* returns `T_type` as well. Thus, `T_type` is a fix-point for the *B_mapsto* behavior. Symbolically, this is specified as follows:

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

The support of objects that have behaviors from multiple types is handled by the single type approach. For example, given types `T_student` and `T_artist`, and an object Sherry that is both a student and an artist, a new type, say `T_student-artist`, is created[8] with all the behaviors of `T_student` and `T_artist`. The object Sherry can then map to this type, thereby acquiring all the behaviors of students and artists. In Section 3.5.1, an automated type inferencing mechanism is defined for generating types during query processing so that result collections which containing objects of different types have a single type describing the common behaviors of all objects in the result. The single type approach is advocated by several type theories including Martin-Löf type theory [ML82, BCMS89] and those based on the typed lambda calculus [Car86].

A model must supply a mechanism for removing objects from the system. The TIGUKAT model allows many references to an object. Therefore, the removal of an object (within a particular scope) consists of severing the link between the reference and the object. This process does not necessarily destroy the object because other references may still be valid and in use (i.e., reference lifetime is independent of scope lifetime). When there are no references to an object, the object is *dangling*. A garbage collection policy could be employed to reclaim the storage occupied by dangling objects. Since this is an implementation issue, it is not part of the formal model definition. Concerning the primitive objects, these are system defined objects and the system always maintains a reference to them. Therefore, these objects are not endangered of becoming dangling objects and being removed by storage reclamation.

[8] This type creation can be done through subtyping as described in Section 2.4.4.

A final behavior on T_object is the identity mapping behavior *B_self* : T_object that maps every object to itself. That is, for any object $o$, $o.B\_self = o$. There are additional object behaviors whose presentation depends on other primitive concepts. These behaviors are introduced after their foundations are established.

### 2.4.4 The Type Primitive

A *type* defines behaviors and encapsulates hidden behavior implementations and state structure for objects created using the type as a template. The behaviors defined by a type describe the *interface* to the objects of that type. Types are organized into a lattice-like[9] structure using the notion of *subtyping*, which promotes software reuse and incremental type development. TIGUKAT supports multiple subtyping, so the type structure is a directed acyclic graph (DAG). However, this DAG has the root T_object, which is a supertype of all types, and the base T_null, which is a subtype of all types.

The uniformity aspects of TIGUKAT imply that types are also objects with their own state and identity along with their own type. The state of a type object consists of a structural specification of its instances (a template), references to the encapsulated behaviors it defines, references to its subtypes and supertypes, and a reference to its associated class (if it exists).

The type that describes all other type objects is the primitive type T_type, which is also a type (i.e., T_type $\mapsto$ T_type). The type T_type is a fix-point for the *B_mapsto* type referencing behavior. T_type is accessible in the same manner as any other object. Thus types, in addition to serving as descriptions of objects, are objects themselves and the type T_type serves as the description of all other types; this is known as the **type:type** property. The issue of **type:type** is controversial, particularly in the area of programming languages. Fortunately, some functional language specifications where the **type:type** property holds have emerged [Car86]. These are likely candidates to assist in the development of a programming language for the model and in expanding the semantic descriptions of behaviors.

Recall from Section 2.4.2 that behaviors are either explicitly defined by a particular type or are inherited from a supertype. Behaviors that are explicitly defined by a type and are not defined in any of its supertypes are called *native behaviors*. Other behaviors of the type that are defined by its supertypes are called *inherited behaviors*. T_type defines behaviors *B_native* for accessing the native behaviors of a type and *B_inherited* for accessing the inherited behaviors. The entire *public interface* of a type is the union of the *native* and *inherited* behaviors. The behavior *B_interface* is defined to return this union. Additional operations are defined on the interfaces to provide facilities for adding, deleting and updating the behaviors of a type. These operations address issues of *update semantics* and *schema evolution* which are covered in Chapter 5.

Two relationships among types have been identified [ÖSP94]. One is the concept of a type *specializing* another type in a manner similar to what is described in [MZO89]. The other is the more popular, and stronger, notion of explicitly creating a type as a *subtype* of another type [Car84]. *Specialize* is a binary relation defined on types that determines whether one type specializes another. A specialization is determined from the semantic characteristics of behaviors.

[9] The term "lattice" is used loosely and is common in describing the type structure of object-oriented systems. Formally, the type structure of TIGUKAT is a complete partial order with a least defined element T_object and a most defined element T_null.

**Behavior 2.3** *Specialize (B_specialize* : T_type → T_boolean*)* (⊑)*:* A *specialize* relation ⊑ between pairs of types $T\_\tau, T\_\sigma$ is a reflexive and transitive relation such that $T\_\tau.B\_specialize(T\_\sigma)$ (denoted $T\_\tau \sqsubseteq T\_\sigma$) is **true** if and only if the interface of $T\_\sigma$ is a subset of the interface of $T\_\tau$ (i.e., $T\_\sigma.B\_interface \subseteq T\_\tau.B\_interface$). This can be interpreted as, type $T\_\tau$ *specializes* type $T\_\sigma$ if and only if the behavioral interface of $T\_\tau$ subsumes the behavioral interface of $T\_\sigma$. If $T\_\tau \sqsubseteq T\_\sigma$ and $T\_\sigma \sqsubseteq T\_\tau$, then either the interfaces of $T\_\tau$ and $T\_\sigma$ are identical or $T\_\tau$ and $T\_\sigma$ refer to the same type object (i.e., $T\_\tau = T\_\sigma$). □

A type may have an associated class of objects that have been created using that type as a template. This is known as the *extent* of the type and is important in the context of *subtyping*. Subtyping, like specializing, is defined as a binary relation on types, but is stronger in the sense that it defines a partial ordering of the type lattice and a subset inclusion relationship on extents.

**Behavior 2.4** *Subtype (B_subtype* : T_type → T_boolean*)* (≼)*:* A *subtype* relation ≼ between pairs of types $T\_\tau, T\_\sigma$ is a reflexive, transitive and antisymmetric relation such that the behavior application $T\_\tau.B\_subtype(T\_\sigma)$ (denoted $T\_\tau \preceq T\_\sigma$) is **true** if and only if type $T\_\tau$ has been created as a subtype of type $T\_\sigma$. The notation $T\_\tau \preceq T\_\sigma$ is interpreted as $T\_\tau$ is a subtype of $T\_\sigma$ and implies that:

1. $T\_\tau \sqsubseteq T\_\sigma$,
2. the behaviors of $T\_\sigma$ are inherited by $T\_\tau$ (i.e., $T\_\tau.B\_inherited = T\_\sigma.B\_interface$), and
3. the extent of $T\_\tau$ is a subset of the extent of $T\_\sigma$.

It can equally be said that $T\_\sigma$ is the *supertype* of $T\_\tau$. □

Consider the simple example in Figure 2.4. The types T_person and T_house have no explicit relationship with one another, however, they do have a derived specialize relationship as indicated by the dashed arrow. On the other hand, the type T_student is explicitly denoted as a subtype of T_person as indicated by the solid arrow. According to the behaviors defined on these types (as shown in the boxes), T_person specializes T_house because T_person defines all the behaviors of T_house and more. From the definition of subtype, T_student specializes T_person (and transitively T_house), which conforms to the behavioral inclusion notion of specialize (i.e., T_student defines all the behaviors of T_person (and T_house), plus more). Conversely, T_house does not specialize T_person nor T_student. It is interesting to note that if T_person did not define the *B_name* behavior, then T_house would specialize T_person as well.

In addition to the behavioral information, the type extents are given in Figure 2.4 with ownership indicated by the double solid line. The subtype relationship between T_student and T_person insists that the extent of T_student is a subset of the extent of T_person (i.e., every student is a person). This subset relationship is shown be the dotted line. On the other hand, the specialize relationship does not demand subset inclusion of type extents. This is reasonable since a person is not a house. Specialize is important when inferring types for the results of queries. For example, if a query returns all the persons or houses that are 25 years of age, a type is needed to describe the members of the query result. By using the specialize relationship between T_person and T_house, a common type can be derived as a supertype of T_person and T_house that includes the behaviors *B_age* and

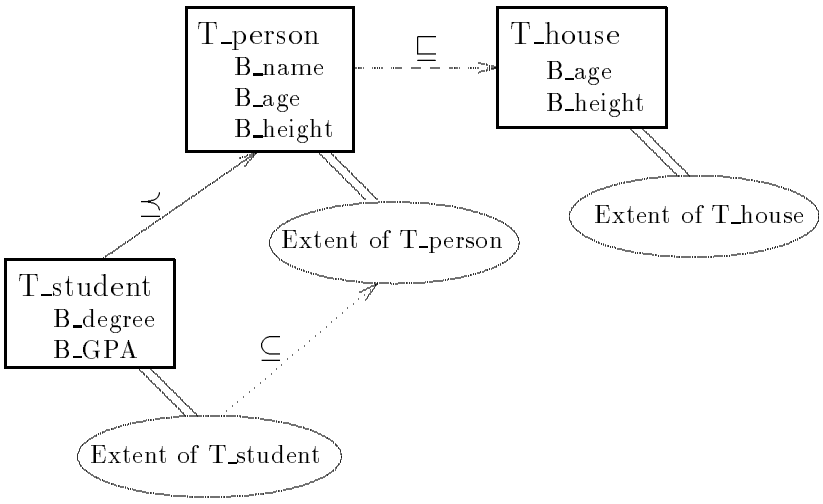



Figure 2.4: Example of subtype and specialize relationships.

*B_height*. These behaviors are applicable to all members of the query result, regardless of whether the member is a person or a house. A complete discussion of type inferencing is given in Section 3.5.1. In summary, specialize is important from the behavioral perspective, while subtype is important from the behavioral and extent inclusion perspectives.

A type is either a *direct subtype* of another type or is a subtype through transitive closure. The model defines two primitive behaviors on type `T_type` for managing subtypes. Behavior *B_subtypes* returns a collection containing all the direct subtypes of a given type and behavior *B_supertypes* returns a collection of all the direct supertypes. The type `T_object` has no supertypes.

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

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

Second, all behaviors of a supertype are automatically inherited by a subtype, which implies that these behaviors cannot be native. Note that this only refers to the *behavioral inheritance* which is different from *implementation inheritance*; the implementation of inherited behaviors may change in the subtype as long as they provide the semantics specified by the behavior. For types in a specialize relationship only, common behaviors may be redefined as native behaviors in each of the types. Lastly, subtyping defines a subset inclusion relationship on type extents while no such property is enforced for specialize. Specialize can be used to test whether two types have compatible interfaces. On the the other hand, subtyping guarantees that the interface of a type is compatible with (or conforms to) the interface of all its supertypes.

A type may be declared as a subtype of several other types, meaning that a type can have many supertypes and also many subtypes. This is usually referred to as *multiple inheritance* [Car84], but the term *multiple subtyping* is used in this thesis. It follows from this property that a type can also specialize many types and be specialized by many other types. Multiple subtyping requires a conflict resolution scheme to select a proper implementation when a type inherits semantically common behaviors (with different implementations) from

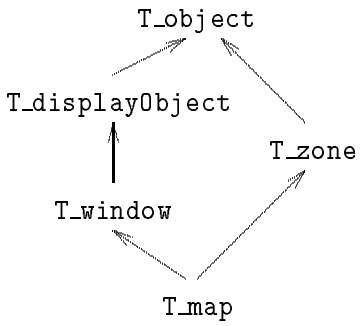



Figure 2.5: Super-lattice of type T_map.

different types. The definition of this protocol is considered to be an implementation issue and therefore is not include as part of the primitive model definition. A simple approach is to force the user to resolve the conflict by either choosing one of the possible implementations or redefining the implementation altogether. Note that conflict resolution is only a problem in implementation inheritance and is not required for behavioral inheritance due to the assumption that semantic definitions of behaviors are powerful enough to express uniqueness that persists across type boundaries.

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

**Axiom 2.2** *Root Type:* for all types $\mathtt{T\_}\tau$, $\mathtt{T\_}\tau \preceq \mathtt{T\_object}$. □

The axiom of root type states that all type objects are subtypes of the type object T_object, which forms the root of the type lattice. This axiom is important in that it forces all types in the system to support the behaviors defined on type T_object. Since types model entities in the system, the axiom ensures that everything is an object.

Every type, together with its supertypes, forms a structure called a *complete lattice*. This structure is introduced and its role in the model is established through the definition of a supertype lattice behavior on the type T_type. The following definitions reference a type system denoted $\mathcal{T}'$ that is defined to include the primitive type system $\mathcal{T}$ together with all application specific types supplementing $\mathcal{T}$.

**Behavior 2.5** *Super-lattice (B_super-lattice :* $\mathtt{T\_poset}\langle\mathtt{T\_type}\rangle$*)* ($\diamond$)*:* For a given type $\mathtt{T\_}\tau$, $\mathtt{T\_}\tau.B\_super-lattice$ (denoted $\diamond_{\mathtt{T\_}\tau}$) returns a collection of types, partially ordered by $\preceq$ (i.e., a poset), such that for all types $\mathtt{T\_}\sigma \in \diamond_{\mathtt{T\_}\tau}, \mathtt{T\_}\tau \preceq \mathtt{T\_}\sigma$ and there does not exist a type $\mathtt{T\_}\rho \in \mathcal{T}'$ such that $\mathtt{T\_}\tau \preceq \mathtt{T\_}\rho$ and $\mathtt{T\_}\rho \notin \diamond_{\mathtt{T\_}\tau}$. □

From Axiom 2.2, all types are a subtype of the type T_object. Therefore, T_object must be in $\diamond_{\mathtt{T\_}\tau}$ for all types $\mathtt{T\_}\tau$ and $\diamond_{\mathtt{T\_}\tau}$ forms a *complete lattice* of types with $\mathtt{T\_}\tau$ being the most defined element in $\diamond_{\mathtt{T\_}\tau}$ and the type T_object being the least defined one. For example, applying the super-lattice behavior to the map type T_map of Figure 2.2 (denoted as T_map.*B_super-lattice*) results in a collection of types including T_map, T_zone, T_window, T_displayObject and T_object that is partially ordered by the $\preceq$ relation. This complete lattice is represented graphically in Figure 2.5.

In addition to super-lattice, the model defines a complement behavior *B_sub-lattice* that returns the sub-lattice of a type. The sub-lattice is also a complete lattice with the receiver

type as the root and type T_null as the base. Note that *B_super-lattice* and *B_sub-lattice* include the receiver type in their result while *B_subtypes* and *B_supertypes* do not. The reason is that every type is a subtype of itself, but is not considered to be a direct subtype of itself.

By definition, any object of type T_$\tau$ must support the behaviors of all types in the super-lattice T_$\tau$. In other words, any behaviors that operate on objects of a type T_$\sigma \in \Diamond_{\texttt{T_}\tau}$ must operate on objects of type T_$\tau$. Some have called this *substitutability* [SZ89] because an object of type T_$\tau$ can be used (substituted) in any context specifying a supertype of T_$\tau$. The definition of *conformance* is refined from [Str91a] to describe this property, but first a *conforms to* relation on the type T_object is defined as follows:

**Behavior 2.6** *Conforms-to (B_conformsTo* : T_type $\rightarrow$ T_boolean*)* ($\rightsquigarrow$)*:* Given an object $o$ and a type T_$\tau$, the behavior application $o$.*B_conformsTo*(T_$\tau$) (denoted $o \rightsquigarrow$ T_$\tau$) is true if and only if $o$.*B_mapsto* $\sqsubseteq$ T_$\tau$. The term $o \rightsquigarrow$ T_$\tau$ reads object $o$ *conforms to* type T_$\tau$. □

The truth of the statement $o \rightsquigarrow$ T_$\tau$ implies that all behaviors defined on type T_$\tau$ are applicable to the object $o$. Given an object $o$ that maps to type T_$\tau$, $o$ must conform to all types that T_$\tau$ specializes. Let $S$ denote the collection containing these types. Each set in the powerset of $S$ forms what is called a *conformance* for the object $o$. A conformance is formally defined as follows:

**Definition 2.4** *Conformance* ($\approx$)*:* A *conformance* for an object $o$ is a collection of types $\Theta$ = {T_1, T_2,..., T_n} such that for all types T_i $\in \Theta$, $o \rightsquigarrow$ T_i. The notation $o \approx \Theta$ is used to indicate that object $o$ has *conformance* $\Theta$. □

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

**Definition 2.5** *Most Specific Conformance* ($MSC()$)*:* A conformance $\Theta$ for an object $o$ is a *most specific conformance* if and only if there does not exist a type T_$\tau \in \mathcal{T}'$ such that $o \rightsquigarrow$ T_$\tau$ and T_$\tau \sqsubseteq$ T_$\sigma$ for some T_$\sigma \in \Theta$, where T_$\sigma \neq$ T_$\tau$. A most specific conformance for an object $o$ is denoted by $MSC(o)$. □

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

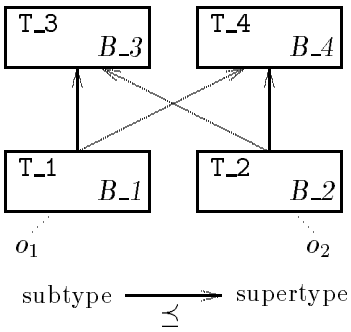



Figure 2.6: An example type schema.

**Definition 2.6** *Most Specific Set Conformance* ($MSC_{set}()$)*:* The most specific set conformance for a collection of objects $O$ (denoted $MSC_{set}(O)$) is the one and only collection of types $\Theta$ such that:

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

□

The first statement indicates that $\Theta$ is a conformance for every object in $O$. The second states that there is no type in the type lattice that more specifically defines the behavior of all objects in $O$ other than the types given in $\Theta$. For example, consider the type structure of Figure 2.6, and assume the existence of two objects $o_1$ and $o_2$ such that $o_1$ is in the extent of `T_1` and $o_2$ is in the extent of `T_2`. Because of subtyping, $o_1$ and $o_2$ are also in the extents of `T_3` and `T_4`. The $MSC(o_1)$ is {`T_1`} and the $MSC(o_2)$ is {`T_2`}. Using this schema, a query could generate and return the generic collection object $\{o_1, o_2\}$. The $MSC(\{o_1, o_2\})$ could be given as the generic collection type {`T_collection`} because of the lack of additional type information. In contrast, the $MSC_{set}(\{o_1, o_2\})$ is the collection of types that most specifically define the behaviors of the elements in $\{o_1, o_2\}$ (i.e., objects $o_1$ and $o_2$ respectively). The result of this conformance is the collection {`T_3`,`T_4`} because both $o_1$ and $o_2$ inherit the behaviors of `T_3` and `T_4` and there is no other type that more specifically defines both objects. The result could not have been {`T_1`} because $o_2$ does not conform to `T_1` and it could not have been {`T_2`} because $o_1$ does not conform to `T_2`; it also couldn't have been {`T_1`,`T_2`} for the same reason. Furthermore, {`T_3`} and {`T_4`} are also incorrect because in these cases some typing information is lost for the member objects; namely, behaviors *B_4* or *B_3* respectively.

$MSC_{set}()$ is used in the query model to perform type checking and type inferencing on the results of queries. The result of a query is a collection that may contain objects of heterogeneous types. $MSC_{set}()$ can be used on query results to determine the most typing information (i.e., behaviors) for these results. The general usefulness of $MSC_{set}()$ and an algorithm for determining the most specific set conformance for a set of objects is presented in [Str91a].

A final behavior required on types is for determining the unique class associated with a given type. In order to create objects of a particular type, there must be a class associated with the type to manage the instances of that type. However, types do not require an associated class if there are no instances of that type. For example, many object-oriented systems include *abstract types* whose sole purpose is to serve as placeholders for common

behaviors of subtypes and are never intended to have any instance objects. In this case, there may be no reason to manage classes for abstract types, because there are no instances of these types. However, a class may be formed if there is a need to categorize the objects of the subtypes by a common class. Thus, the model enforces the one way implication: *class* $\Longrightarrow$ *type*. The behavior *B_classof* is introduced to manage the class of a type:

**Behavior 2.7** *Class of (B_classof :* `T_class`*)* ($\mathcal{C}$)*:* Given a type `T_`$\tau$, the behavior application `T_`$\tau$.*B_classof* (denoted as $\mathcal{C}_{\texttt{T\_}\tau}$) returns the unique class object (if it exists) **C_**$\tau$ associated with `T_`$\tau$ that manages the extent of type `T_`$\tau$. □

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

### 2.4.5 The Collection and Class Primitives

The support of efficient query processing and storage management requires mechanisms to group related objects so that they may be managed, referenced and processed collectively. The *collection* and *class* objects serve this purpose in TIGUKAT. The relative advantages and disadvantages of providing a system-managed class as the only grouping mechanism for the extent of a type versus supporting user defined and managed collections as clusters of instances has been debated [YO91, ÖSP94]. Beeri [Bee90] shows, at a structural level, that both can be supported. The TIGUKAT model defines both classes and collections for grouping objects.

A *collection* is a general grouping mechanism. The objects managed by a collection are called the *extent*. The term "collection" and "extent" are equated, meaning a reference to a collection is a reference to its extent.

There are two ways that objects can be included in a collection. One is that objects can be explicitly added to the collection. The other is that a predicate can be defined on a collection that automatically includes objects.

The objects in a collection support a set of common behaviors; they must minimally support the behaviors of `T_object`. These common behaviors are defined by a type (called the member type) in the type lattice that is associated with the collection when it is created and can evolve as the extent changes. Every collection knows its member type.

The semantics of collection objects are given by the behaviors defined on the primitive type `T_collection`. The following behavior is defined on `T_collection` and returns the member type of a collection. The member type may be specified by the user or the system may automatically derive this type.

**Behavior 2.8** *Member Type (B_memberType :* `T_type`*)* ($\Lambda$)*:* Given a collection **L_**$\tau$, the behavior application **L_**$\tau$.*B_typeof* (denoted $\Lambda_{\mathbf{L\_}\tau}$) returns the singleton type object that represents the member type of collection **L_**$\tau$. The member type has the property $\forall o \in \mathbf{L\_}\tau, o \rightsquigarrow \Lambda_{\mathbf{L\_}\tau}$. □

Collections may be heterogeneous in the sense that the extent may contain objects that map to different types which are not in a subtype relationship with one another. The type inferencing mechanism in Section 3.5.1 guarantees that in such cases a unique type is chosen (or created) as the member type of the collection, and that this type represents the

most defined combination of the heterogeneous types. This approach allows Behavior 2.8 to always hold.

Heterogeneous collections are essential for proper handling of queries that may return objects of various types [SÖ90a]. A collection always has an associated type that specifies the behaviors supported by all objects in the extent of the collection. The maintenance of this type may require the automatic derivation of new types (during projections and joins in the algebra for example) in order to provide as much type information as possible for the objects in the collection. Type inferencing is used by the object query model defined in Chapter 3.

Other behaviors defined in `T_collection` include *B_cardinality* (denoted $|\mathbf{L\_}\tau|$) to return the number of elements in a collection, *B_elementOf* (denoted $o \in \mathbf{L\_}\tau$) to determine if a object is a member of a collection, *B_containedBy* (denoted $\mathbf{L\_}\tau \subseteq \mathbf{L\_}\sigma$) to determine subset inclusion of extents, *B_insert* and *B_delete* to add/remove objects to/from collections, and a host of other behaviors representing the algebraic operators that are introduced by the query model.

The specialized, better known, form of a collection is that of a *class*. The type `T_class` is defined as a subtype of the type `T_collection`. Therefore, classes must support all behaviors defined for collections, but these behaviors are refined (i.e., specialized) for classes. Every class is uniquely associated with a single type. This association occurs at class creation time and persists with the class throughout its lifetime. The *B_memberType* behavior for classes is defined to return this type. *B_memberType* on classes is the inverse behavior of *B_classof* on types.

The extent of a class is separated into two forms. The first form is called the *shallow extent* and is similar to the extent of a collection in that a class represents its shallow extent. The second form is called the *deep extent* and is built from the shallow extents of classes. Shallow and deep extents are well know concepts that have been discussed in other models [KC86, BCG+87, SÖ90a]. They are formally defined as follows.

**Definition 2.7** *Shallow Extent* ($^+$)*:* The *shallow extent* of a class $\mathbf{C\_}\tau$ (written $\mathbf{C\_}\tau^+$) is the collection consisting of all objects $o$ such that $o \mapsto \Lambda_{\mathbf{C\_}\tau}$. The class itself represents its shallow extent. □

**Definition 2.8** *Deep Extent* ($^*$)*:* The *deep extent* of a class $\mathbf{C\_}\tau$ (written $\mathbf{C\_}\tau^*$) is the collection consisting of all objects $o$ such that $o.B\_mapsto \preceq \Lambda_{\mathbf{C\_}\tau}$. There is a behavior *B_deepExtent* defined on `T_class` that returns the deep extent of a class. □

In a context where neither the shallow ($+$) nor deep ($*$) extent qualification is given, the deep extent is assumed.

The shallow extent of a class includes all objects created using the class member type as a template. The deep extent of a class includes the objects of the shallow extent union the shallow extents of the associated classes of all subtypes of the class member type. The shallow extent of classes are disjoint groupings of objects. That is, for all classes $\mathbf{C\_i}$, $\mathbf{C\_j}$, the collection $\mathbf{C\_i}^+ \cap \mathbf{C\_j}^+$ is empty when $\mathbf{C\_i} \neq \mathbf{C\_j}$. The definition of deep extent imposes a subset inclusion relationship on the extents of classes. This is referred to as *subclassing*, which has a direct relationship to subtyping and is in keeping with the conformance properties on types.

**Definition 2.9** *Subclass:* A class $\mathbf{C\_}\tau$ is a *subclass* of a class $\mathbf{C\_}\sigma$, meaning $\mathbf{C\_}\tau^* \subseteq \mathbf{C\_}\sigma^*$, if and only if $\Lambda_{\mathbf{C\_}\tau} \preceq \Lambda_{\mathbf{C\_}\sigma}$. One can equally say that $\mathbf{C\_}\sigma$ is the *superclass* of $\mathbf{C\_}\tau$. □

In TIGUKAT, a *type* is separated from the declaration of its *class* and subsequent *collections*. This design issue is a controversial one. Many former model proposals bundled these two concepts calling them either a "type" or a "class" [GR85, LRV88, BBB+88, Str90]. In the TIGUKAT model, special care is taken to separate the two notions and attach individual semantics to each one. We believe that a type is simply a specification mechanism that is used to describe the structure and behavior of objects. This should be separated from the grouping of objects in order to provide flexibility in defining exact grouping semantics. In the TIGUKAT model, classes group the shallow and deep extents of types, which has its basis on subtyping. In other models, this definition varies. The introduction of collections supplements classes by providing a very general grouping mechanism that has a consistent semantics with the concept of a class. The inclusion and separation of these notions provide greater modeling flexibility and expressibility than if they were bundled into a single concept. For example, in Chapter 3 queries are defined to operate on collections and return collections as results. Since classes specialize collections, queries can also operate on classes. The type checking of queries and the type inferencing of query results is a separate issue. Both classes and collections should be type checked. Since types are separate from classes, this is possible in TIGUKAT through the member type. Furthermore, a member type may be created for a collection without ever creating any objects of that type (i.e., abstract types). This new type may define the common behaviors of heterogeneous members of a collection consisting of existing objects in the objectbase that do not map to the new type. Separation of type and class allows this notion to be easily modelled as well.

A final behavior defined on the type `T_class` is that of object creation. All objects are created through a particular class using that class member type as a template. This has the side effect of automatically placing the object in the shallow extent of the class, which implies that it is in the deep extent as well. In the following signature, the notation $\Lambda_c$ denotes the type resulting from applying the *B_memberType* behavior on a *receiver* class object $c$.

**Behavior 2.9** *New (B_new : $\Lambda_c$):* Given a class $\mathbf{C\_}\tau$, the application of the behavior $\mathbf{C\_}\tau.B\_new$ has the result of creating a new object $o$ such that $o$ is consistent, $o \mapsto \Lambda_{\mathbf{C\_}\tau}$ and $o \in \mathbf{C\_}\tau^+$ (which implies $o \in \mathbf{C\_}\tau^*$). The application $\mathbf{C\_}\tau.B\_new$ denotes an object reference to the newly created object $o$ whose type is $\Lambda_{\mathbf{C\_}\tau}$ that is derived from the receiver class object $\mathbf{C\_}\tau$. □

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

### 2.4.6 Higher Level Constructs

Several of the primitives introduced in the previous sections are referred to as *meta-information* because they are objects which provide support for other objects. For example, the type `T_type` provides support for types by defining the structure and behaviors

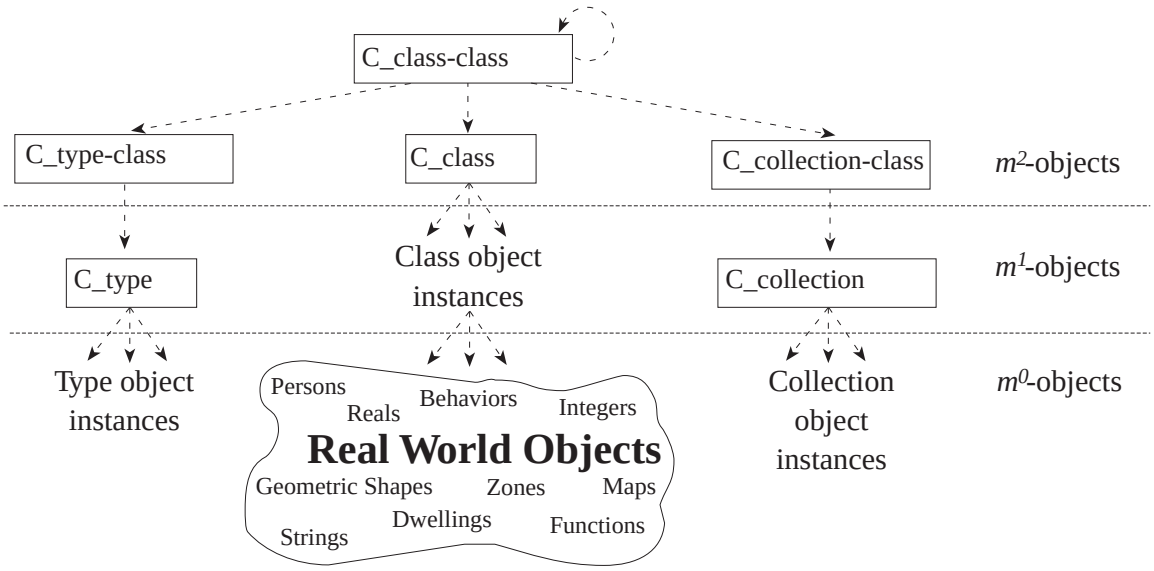



Figure 2.7: Three tiered instance structure of TIGUKAT object management.

of type objects and the class **C_class** supports classes by managing class objects in the system. In a uniform model, these meta-objects are objects themselves and are uniformly managed within the model as first-class objects. The support for this semantics lies in the introduction of higher level constructs called *meta-meta-objects* or $m^2$-objects.

The meta-system of TIGUKAT is a three tiered structure for managing objects. This structure is depicted in Figure 2.7. Each box in the figure represents a class and the text within the box is the common reference name of that class. The dashed arrows represent shallow extent instance relationships between these objects with the head of the arrow being the instance and the tail being the class to which that instance belongs.

The lowest level of the structure consists of the "normal" objects that depict real world entities such as *integers, persons, maps, behaviors* and so on, plus most of the primitive object system is integrated at this level. These include types, collections, behaviors and functions that are represented as objects, which illustrates the uniformity in TIGUKAT. This level is designated $m^0$ and its objects are $m^0$-objects.

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

The upper-most level consists of the meta-meta-information (labeled $m^2$) which defines the functionality of the $m^1$-objects (meta-information). The structure is closed off at this level because the $m^2$-object **C_class-class** is an instance of itself as illustrated by the looped instance edge. The introduction of the $m^2$-objects adds a level of abstraction to the type lattice and instance structures. The need for this three-tiered structure comes from the fact that every object belongs to a class and every class is associated with a type that defines the semantics of the instance objects in the class. Regular objects (level $m^0$) belong to some

class (level $m^1$). Since classes are objects, the class objects (level $m^1$) belong to some class (level $m^2$). The $m^2$ class objects belong to the $m^2$-class **C_class-class** which closes the lattice. The types associated with these classes are all managed as regular objects at level $m^0$. The outcome of this approach is that the entire model is consistently and uniformly defined within itself. In the following discussion, the interactions among the various levels of the structure and how they contribute to the uniformity of TIGUKAT are described. This forms the foundation of reflective capabilities.

A portion of the primitive type lattice (Figure 2.1) responsible for the meta-system is shown in Figure 2.8. Furthermore, a companion subclass lattice for this portion is shown in Figure 2.9 where **C_$x$** in Figure 2.9 is the associated class of type `T_`$x$ in Figure 2.8.

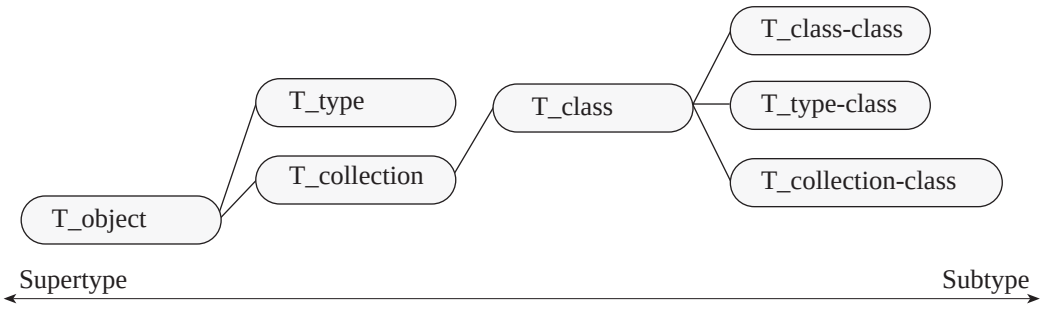



Figure 2.8: Portion of primitive type lattice responsible for meta-system.

Figure 2.9 illustrates the subset inclusion relationship and instance structure for some of the $m^0$, $m^1$ and $m^2$-objects. Starting from the left-side of the lattice structure, the relationships between these classes and their instances are described.

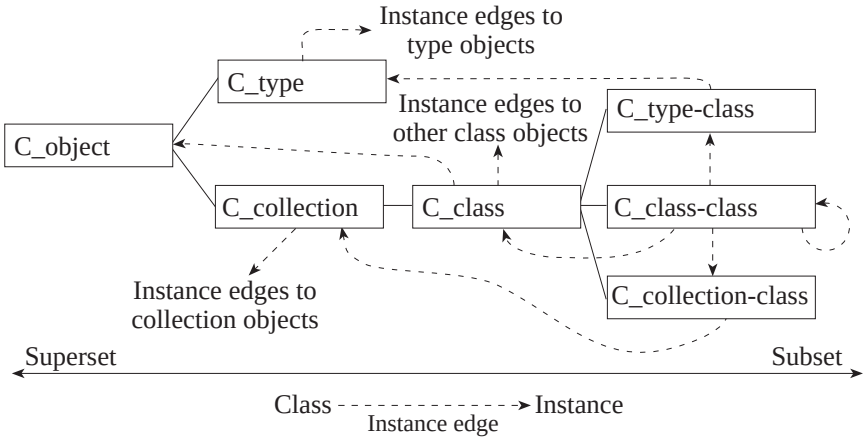



Figure 2.9: Subclass and instance structure of $m^1$ and $m^2$ objects.

The class **C_object** is an $m^1$-object that maintains all the objects in the objectbase (i.e., every object is in the deep extent of class **C_object**). Two other $m^1$-objects in the figure are subclasses of **C_object**, namely, **C_type** and **C_collection**. These two classes maintain the instances of types and collections, respectively. Class **C_collection** is further subclassed by the $m^2$-object **C_class** because every object that is a class is also a collection of objects. For example, the class **C_person** is an instance of the class **C_class** and **C_person** is a collection of person objects as well. The class **C_class** manages the instances of all classes

in the system like **C_object**, **C_person** and so on. Finally, **C_class** is subclassed by $m^2$-objects **C_type-class**, **C_class-class** and **C_collection-class**. Intuitively, **C_type-class** is a class whose instances are classes that manage type objects. Similarly, **C_class-class** is a class whose instances are classes that manage class objects and **C_collection-class** is a class whose instances are classes that manage collection objects.

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

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

For the following discussion, the reader is referred to Appendix A, which lists the signatures of the behaviors defined on the primitive types, including the meta-types. In the following, "$o \leftarrow r.B$" denotes assignment of the result of behavior application $r.B$ to an object reference $o$.

The model must have a way of consistently creating new types. Applying the generic *B_new* behavior (i.e., the one in `T_class`) on the class **C_type** is inadequate for this purpose because it simply creates new empty objects and a type must always be created as a subtype of some other type(s); minimally a subtype of `T_object`. *B_new* cannot handle these semantics because it is a generic behavior for creating any kind of object and only new type objects need supertype information; it would be inappropriate to place these semantics on *B_new*. Therefore, the *B_new* behavior must be specialized for types to allow for the addition of arguments that specify the supertype(s) of the new type, along with other arguments such as its native behaviors. To accomplish this, the type `T_class` is subtyped by type `T_type-class` (see Figure 2.8) and the behavior *B_new* is refined on this type. Now, in the primitive system, the type `T_type-class` is associated with the class **C_type-class** and the class **C_type** is created as an instance of **C_type-class** as shown in Figure 2.9. New types are created by applying the refined *B_new* behavior to **C_type**. This follows the tenet of uniformity: the behaviors defined on type `T_type-class` are applicable to the object **C_type** because it is in the extent of class **C_type-class** and **C_type-class** is associated with type `T_type-class`. In the following signature definitions, the notation $\Lambda_c$ again denotes the member type of a receiver class $c$.

**Behavior 2.10** *New Type (B_new* : `T_collection`$\langle$`T_type`$\rangle \rightarrow$ `T_collection`$\langle$`T_behavior`$\rangle \rightarrow \Lambda_c$*):* Given the class **C_type**, a set of types $T$, and a set of behaviors $B$, the behavior application **C_type**.*B_new*$(T, B)$ creates a new type as an instance of **C_type** such that it is a subtype of the types in $T$ and it defines the behaviors in $B$ as native behaviors unless they are inherited from a type in $T$. □

For example, in order to create a new type for modeling mobile homes (as a subtype of `T_dwelling`) that adds a behavior "*B_numberOfMoves*:`T_natural`" (assumed to be defined), one applies the *B_new* behavior to **C_type** and passes the appropriate arguments. The result is assigned to a standard type reference `T_mobileHome` as follows:

$$\texttt{T\_mobileHome} \leftarrow \textbf{C\_type}.B\_new(\{\texttt{T\_dwelling}\}, \{B\_numberOfMoves\})$$

A class must be associated with a type (its member type) in order to be able to create objects of that type. Furthermore, classes must be uniquely associated with a single type and no class may exist without an associated type. In order to consistently support these

semantics, the type T_class is subtyped by the type T_class-class (see Figure 2.8) and behavior *B_new* is refined for creating and associating new classes with a type.

In the primitive system, the class **C_class-class** is associated with T_class-class and maintains all the $m^2$-classes. Its instances include itself, **C_type-class**, **C_collection-class** and **C_class**. Each of these classes maintain instances of other classes. Various kinds of class structures are created by applying *B_new* to one of these classes. For the model, this means that we additional classes can be created for managing types (additional instances of **C_type-class**), for managing collections (additional instances of **C_collection-type**), for managing classes (additional instances of **C_class**), and for managing classes that manage classes (additional instances of **C_class-class**).

**Behavior 2.11** *New Class (B_new* : T_type $\rightarrow \Lambda_c$*):* Given an instance of **C_class-class** (e.g., **C_class**) and a type T_$\sigma$, the behavior application **C_class**.*B_new*(T_$\sigma$) has the result of creating a new class object **C_**$\sigma$ such that **C_**$\sigma$ is in the shallow extent of **C_class** and **C_**$\sigma$ is associated with type T_$\sigma$. If type T_$\sigma$ does not exist, or is already associated with some other class, an error condition is raised because a type may be associated with at most one class. □

For example, the following behavior application creates a new class **C_mobileHome** as an instance of **C_class** and associates this class with type T_mobileHome created above.

**C_mobileHome** ← **C_class**.*B_new*(T_mobileHome)

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

In the same way as types are associated with classes, types are also associated with collections; but a type may be the member type of any number of collections. The type T_collection-class is defined as a subtype of T_class and behavior *B_new* is refined for creating new collections similar to what was done for classes. The class **C_collection-class** is associated with T_collection-class and class **C_collection** is created as an instance of **C_collection-class** (see Figure 2.9). New collections are created by applying *B_new* to **C_collection**, passing in an appropriate member type.

**Behavior 2.12** *New Collection (B_new* : T_type $\rightarrow \Lambda_c$*):* Given class **C_collection** and type T_$\sigma$, the behavior application **C_collection**.*B_new*(T_$\sigma$) creates a new collection object **L_**$\sigma$ such that **L_**$\sigma$ is in the shallow extent of **C_collection** and **L_**$\sigma$ defines T_$\sigma$ as its member type. The type T_$\sigma$ may be omitted in which case the member type of the collection is maintained by the system and derived according to the members in the extent of the collection. If type T_$\sigma$ is given and does not exist, an error condition is raised. Types may be associated with any number of collections. □

For example, to create a new collection of map objects for mapping moblile home parks, one applies *B_new* to **C_collection** as follows:

**L_mobileHomeParks** ← **C_collection**.*B_new*(T_map)

The introduction of the $m^2$-objects complicates the type lattice and instance structures. However, the benefit of this approach is that the entire model is now consistently and uniformly defined within itself. This defines a powerful model for managing all objects, including meta-information, in a uniform way. There are several uses for this modeling capability including the ability to perform reflection. These features are presented in Chapter 4.

### 2.4.7 The Null Primitive

*Nulls* are introduced to provide a simple null semantics. The model defines a primitive type `T_null` along with its corresponding class **C_null**. This class is defined to have as primitive instance objects null, void, undefined, and dontknow. Others, such as error conditions, can be added as required.

The type `T_null` is defined to be the subtype of all other types, which is automatically maintained by the system. This gives `T_null` the opposite semantics of the type `T_object`, which is defined to be the supertype of all types. The type `T_null` *lifts* the domain of types and creates a lattice that is bounded (or pointed) at both ends. A companion axiom for the axiom of root type (Axiom 2.2) is defined to describe the type constraint of the null type.

**Axiom 2.3** *Null Type:* for all types $\texttt{T\_}\tau$, $\texttt{T\_null} \preceq \texttt{T\_}\tau$. □

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

For example, `T_null` is a subtype of the type `T_person` in the GIS example type lattice of Figure 2.2. Therefore, `T_null` can refine the behaviors of `T_person` to return an instance of `T_null` (e.g., null, undefined, etc.). Now, if for a specific instance of `T_person`, say Sherry, the result of a certain behavior, say *B_age*, is not known, it can be assigned an instance of `T_null` (e.g., null). Then, the application Sherry.*B_age* returns the object null, and all subsequent behavior applications (except for those of the primitive type system) also return some instance of `T_null`.

### 2.4.8 Definition of an Objectbase

With the modeling primitives established, the meaning of an objectbase is now defined.

**Definition 2.10** *Objectbase* ($\mathcal{OB}$)*:* An objectbase $\mathcal{OB}$ is a consistent set of objects (*conset*) such that:

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

   The elements of the *primitive object system* $\mathcal{O}$ (which is a *conset*, Section 2.4.3) are part of $\mathcal{OB}$.

2. for all objects $o \in \mathcal{OB}$, for all behaviors $B\_i \in \mathcal{OB}$, $o.B\_i \in \mathcal{OB}$.

   For all general objects and behavior objects in $\mathcal{OB}$, applying a behavior from $\mathcal{OB}$ to an object in $\mathcal{OB}$ results in an object that is also in $\mathcal{OB}$.

An objectbase defines a restricted enclosure of objects that facilitates a consistent, systematic investigation of other objectbase features such as query processing, query optimization, reflection, dynamic schema evolution, view management, transaction management, and distributed object management. An objectbase does not define the relationships of its consistent object set with external objects outside the domain of the objectbase. For now, these relationships should be considered ill-defined and inconsistent, although they may prove useful in the context of distributed environments.

## 2.5 The Structural Model

Beeri's work on formal structural object models [Bee90] has been chosen as a foundation for an example TIGUKAT structural model definition. In this chapter, Beeri's framework is followed to define a structural model that complies with the behavioral model of TIGUKAT and the integration of the two are shown.

### 2.5.1 Objects and Values

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

Beeri makes a strong case in distinguishing between the notions of "object" and "value" at the structural level. However, he does point out that in the general intuitive sense, objects and values should have the universal perception of objects. The latter perspective is defined by the behavioral model presented in Section 2.4. The structural model presented here introduces a separation of these two notions because there is an inherently different representation and semantics for values at this lower level. These differences need to be resolved eventually, and the structural model seems to be the appropriate place for this.

Beeri outlines several arguments that support the distinction of "values" from "objects." The reasons that most influence this separation are:

1. the perception that values represent universally known abstractions (such as the integers), while objects denote application specific abstractions,

2. the notion that values are built into the system and are assumed to exist, while objects need to be defined and introduced into the system,

3. the information carried by a value is itself and is immutable, while an object consists of a separate mutable state that represents the information carried by the object.

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

**Definition 2.11** *Atomic Value:* An *atomic value* is any object from the domains of the atomic types. Atomic values are predefined by the atomic types and are managed by the system. Atomic values are immutable.

Each atomic type has a standard representation for references to the atomic values of their respective domains. The act of specifying one of these references is treated as a request to return the appropriate atomic object. The system may chose to return and existing atomic object from the objectbase or may create a new one on the fly. The form of these standard references is purely syntactic and one interpretation is discussed in Section 2.4.1. Since these references are system maintained, they will never be released and will persist throughout the lifetime of the objectbase, thereby making them immutable.

Recall the definition of an object as an (*identity, state*) pair (Section 2.4.3). For atomic values, the value itself serves as identity and state all at once. This property is what makes values immutable to change. The distinguishing factor between objects and values seems to be that objects have an immutable identity separate from a mutable state, while values represent identity and state all at once, both of which are immutable. Beeri makes the distinction that values are used to describe other things, while objects are the things being described. From a mathematical perspective, one may consider values to be elements of the *built-in* domains, while objects are elements of the *uninterpreted* domains.

### 2.5.2 Abstract Objects

An *abstract object* is defined as an object that has the semantics of an immutable identity separate from a mutable state. Application specific objects and the primitive non-atomic objects all fit into this category.

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

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

In the TIGUKAT model, there is a commonality between values and objects that captures their atomicity. When referring to atomic values and abstract objects, essentially the identities of these objects are being referred. This is separate from the the state of objects. The difference between values and abstract objects is that the state of the former is immutable while the latter has a state that may change over time.

The behavioral model defines *collection, bag, poset* and *list* types for developing structured aggregation objects. The instances of these aggregate types are called *container abstract objects* (*containers* for short) in the structural model. Containers are similar to the set structured values defined by Beeri. However, containers in TIGUKAT are uniformly managed as abstract objects and may be subtyped to customize their semantics. One example is the use of parameterization to define containers whose elements are restricted to

a particular type.

Beeri also defines tuple structured values, but TIGUKAT does not. The notion of tuple is cast into the uniform concept of behaviors on types. A tuple in TIGUKAT is just a type definition with the behaviors representing the named slots (or attributes) of the tuple.

### 2.5.3 Object Graph

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

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

The graph representation presented in this section defines several kinds of nodes that may be used in an object graph. Figure 2.10 illustrates the graphical representation of these nodes and the semantics of each is defined as follows:

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

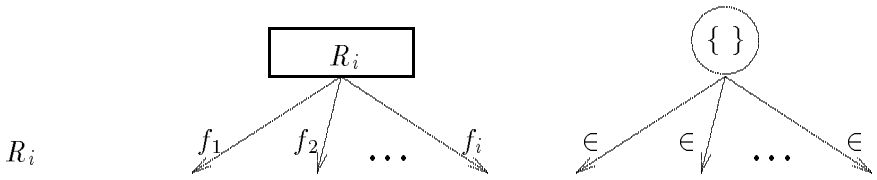
**2.10 (b)** Abstract objects consist of a box labeled with an explicit reference for identifying the object. This label can be thought of as a structural model reference and has no implications of the other scope specific object references that may exist. Abstract objects have an outgoing edge for each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

**2.10 (c)** Container abstract objects consist of an oval labeled with an explicit reference or the symbols { } if a descriptive reference is immaterial. A container has outgoing edges labeled with "$\in$" to each member object. These represent the extent of the container. Containers, like all abstract objects, have other edges to represent the behaviors specific to them.

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

### 2.5.4 Structural Example

Consider the object definitions of Figure 2.11. Each box represents a separate abstract object where the header specifies a reference for the object along with the *maps to* type




(a) Atomic value. (b) Abstract object. (c) Container abstract object.

Figure 2.10: Graphical representations of nodes in an object graph.

for that object. Following this, the behaviors for each object is listed and their associated values are given.

Figure 2.12 illustrates an object graph for the geographic objects SCounty, Notingham and Forest3 of Figure 2.11. The map object SCounty is an abstract object with several outgoing behavioral edges as shown. All nodes in the graph have a *B_self* edge that points back to the node. *B_self* is only shown for SCounty. The *B_proximity* behavior is not defined for the object and therefore points to the abstract object null. The behaviors *B_resolution*, *B_orientation* and *B_title* point to the atomic valued objects 0.5, 0 and "Sherwood County" respectively. The *B_region* behavior points to a `T_geometricShape` object that defines the geometric structure of the SCounty object. The *B_origin* behavior points to the `T_location` object $\mathsf{loc}_0$ which has *B_latitude* and *B_longitude* behaviors to the appropriate atomic valued objects 44.9 and 37.1 respectively. Finally the *B_zones* behavior points to a container comprising of the two `T_zone` element abstract objects Notingham and Forest3.

There are a few anomalies to note for the zone objects Notingham and Forest3. First, the *B_origin* behavior for Forest3 and SCounty share the same `T_location` object $\mathsf{loc}_0$ which is indicated by its two incoming edges. Second, the *B_proximity* behaviors for the two zone objects are defined and point to function abstract objects that, when given another zone object as an argument, produce the desired distance measurement representing the proximity of the argument zone to the zone on which the function is defined. For example, *B_proximity* applied to Forest3 results in the function abstract object *B_proximity*(Forest3). This abstraction can be maintained by returning the implementation function object associated with *B_proximity* with the first argument fixed to Forest3. This is sometimes referred to as a context. Context's are used in query optimization as well. The graph further indicates that an invocation of this context, when passed the argument zone Notingham, will produce the atomic valued object 25.34. This context execution is represented by the dotted line attached to Notingham in Figure 2.12. A similar application is shown on Forest3 for the *B_proximity* behavior of Notingham which shares the same result object as the previous execution.

The dotted lines do not represent behavior applications on the type `T_zone` in the normal sense, although they could. Instead, they represent the result of executing a function that has some arguments fixed (i.e., a context) and are included in this example to illustrate the power and flexibility that the functional approach provides.

### 2.5.5 Schema Objects

The structural model of TIGUKAT differs from Beeri's model [Bee90] in that Beeri makes a clear separation between the data of an objectbase and its schema, whereas TIGUKAT carries the uniformity aspects of the behavioral model into the structural model. This means

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

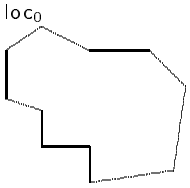
| Person9 ↦ T_person | |
|---|---|
| *B_name*: | "Robin Jr." |
| *B_birthDate*: | null |
| *B_age*: | null |
| *B_residence*: | SForest |
| *B_spouse*: | null |
| *B_children*: | null |

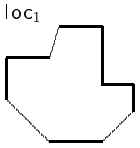
| Person7 ↦ T_person | |
|---|---|
| *B_name*: | "Maid Marion" |
| *B_birthDate*: | null |
| *B_age*: | null |
| *B_residence*: | SForest |
| *B_spouse*: | Person4 |
| *B_children*: | *B_children*(Person7) |
| *B_children*(Person4): | {Person9} |

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

| SForest ↦ T_dwelling | |
|---|---|
| *B_address*: | "Top Secret" |
| *B_inZone*: | Forest3 |

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

| Forest3 ↦ T_forest | |
|---|---|
| *B_title*: | "Sherwood Forest" |
| *B_origin*: | $\mathsf{loc}_0$ |
| *B_region*: |  |
| *B_proximity*: | *B_proximity*(Forest3) |
| *B_proximity*(Notingham): | 25.34 |

| Notingham ↦ T_developed | |
|---|---|
| *B_title*: | "City of Notingham" |
| *B_origin*: | $\mathsf{loc}_1$ |
| *B_region*: |  |
| *B_proximity*: | *B_proximity*(Notingham) |
| *B_proximity*(Forest3): | 25.34 |

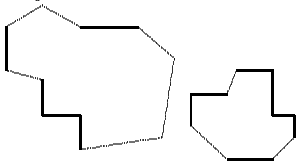
| SCounty ↦ T_map | |
|---|---|
| *B_title*: | "Sherwood County" |
| *B_origin*: | $\mathsf{loc}_0$ |
| *B_region*: |  |
| *B_proximity*: | null |
| *B_resolution*: | 0.5 |
| *B_orientation*: | 0 |
| *B_zones*: | {Forest3,Notingham} |

Figure 2.11: Objects of Sherwood County.

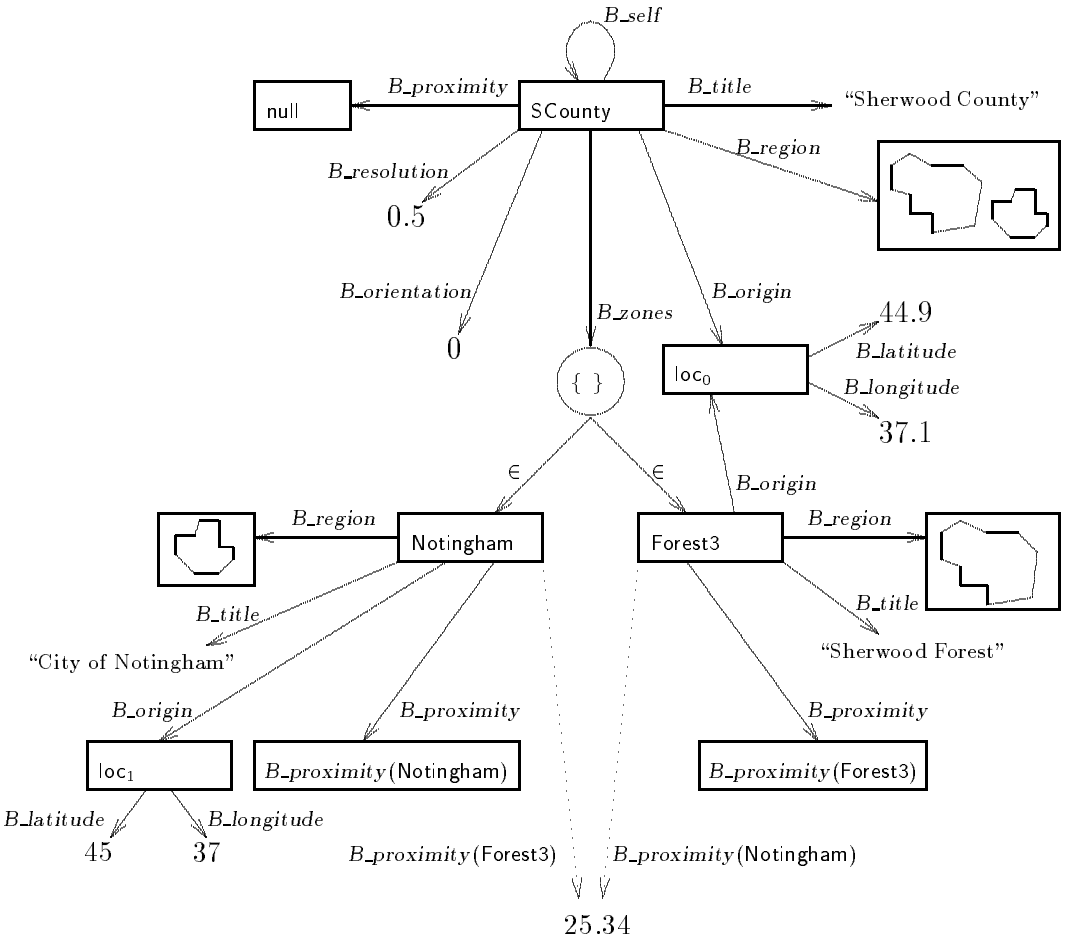



Figure 2.12: Object graph of SCounty, Notingham and Forest3 objects in Figure 2.11.

that schema objects in TIGUKAT are represented using the same graphical structures as other objects and may be integrated into a single object graph representing all information. In this way, the schema objects become part of the objectbase, which allows all database operations to be performed on them in a consistent manner.

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

Objects of type `T_type` have *B_native*, *B_inherited* and *B_interface* behavior edges pointing to containers of behaviors representing the various interface components of a type. There are *B_supertypes* and *B_subtypes* edges to containers holding the direct supertypes and direct subtypes of a type, respectively. There are *B_super-lattice* and *B_sub-lattice* edges to partially ordered containers holding the supertype lattice and subtype lattice of a type, respectively. A type has a *B_classof* edge that points to the class object that maintains the instances of the type. Finally, there are *B_subtype* and *B_specialize* edges to contexts that, when executed with another type object argument, result in a true or false object depending on whether or not the original type is in the given relationship with the second argument type.

A class object has the same outgoing edges as containers do, plus an extra edge for its deep extent behavior (*B_deepExtent*) to a container node that has an $\in$ edge to each object in the deep extent of the class. Finally, there is an edge for the *B_new* behavior to the last newly created object of the appropriate type. The side effect of applying *B_new* is to update itself to create a new object and add the object to the receiver class.

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

For example, consider the partial schema representation of the type `T_zone` as an object graph shown in Figure 2.13. The `T_zone` object indicates a *B_mapsto* behavior to the type object `T_type` of which it is also an instance. There is a *B_classof* edge to the class **C_zone**. The *B_conformsTo*, *B_subtype* and *B_specialize* behaviors result in contexts that can be applied to other `T_type` objects and determine the truth or falsity of the relationship. There is a *B_supertypes* edge to a container holding the direct supertype `T_object` of type `T_zone`. There is a *B_super-lattice* edge to a container that has element edges to the two supertypes of `T_zone` (one of which is itself). Finally, the *B_native* container of behaviors for `T_zone` is shown holding four behaviors that are defined locally by `T_zone`. The containers for *B_inherited* and *B_interface* are not shown. The container for *B_inherited* would have behaviors *B_mapsto*, *B_equality*, *B_self* and *B_conformsTo* that are inherited from `T_object` and *B_interface* would simply be the union of these two containers.

Due to the complexity of these graphs, many of the relationships are not shown. However, the previous examples give a flavor of how these links are managed and the inherent uniformity in their representation.

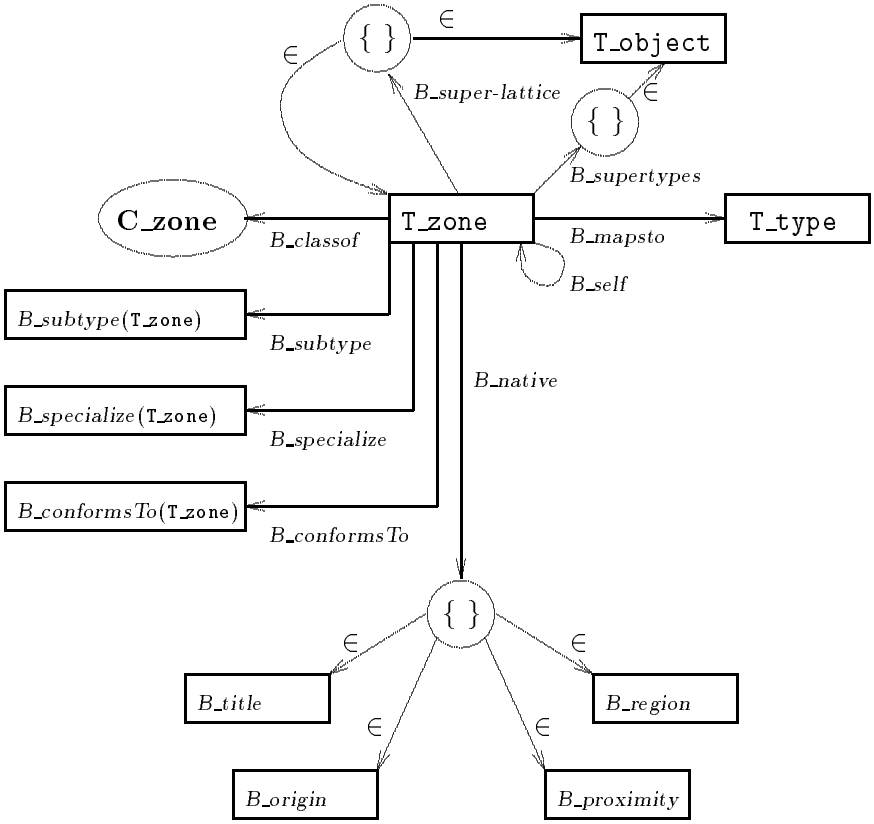



Figure 2.13: Object graph of partial schema for type `T_zone`.

# Chapter 3

# The Object Query Model

The design of a complete and uniform behavioral object model forms a basis for an extensible object query model. Following the uniform semantics of the object model, queries are modeled as type and behavior extensions to the base object model. This incorporates queries as an extensible part of the model itself. The object query model definition presented in this chapter[1] includes the type and behavior extensions to the base model, a formal object calculus with a logical foundation that is closed and incorporates the behavioral paradigm of the object model, a closed behavioral/functional object algebra with a comprehensive set of object-preserving and object-creating operators, a rigorous definition of safety based on the *evaluable* class of queries which is arguably the largest decidable subclass of the *domain independent* class, and a notion of completeness that includes reductions between the algebra and calculus that proves their equivalence. In addition to the formal aspects, a complete algorithmic translation from calculus to algebra is given.

An SQL-like user language, definition language and control language have been developed for the model and are reported elsewhere [PLÖS93a, PLÖS93b, Lip93]. Furthermore, the uniformity of the object model has been used to define an extensible query optimizer and execution plan generator. However, these components are outside the scope of this thesis.

## 3.1 Related Work

One reason for the broad acceptance of relational DBMSs is their implementation of a high level, declarative query facility, which provides an elegant and simple interface to the underlying model. One of the most popular query languages in those systems is SQL, which has become an international standard for the definition and management of relational structured data [ISO92].

In order to consistently extend the functionality of relational systems, next generation DBMSs must extend the power of the relational query model and SQL. Therefore, one of the problems facing object-oriented system designers is the definition of an object query model and its languages. The languages addressed in this thesis include a declarative calculus and a functional algebra.

[1] Portions of this chapter are published in the 1993 *Proceedings of the Second International Conference on Information and Knowledge Management (CIKM'93)* [PLÖS93a] and as a book chapter in *Emerging Landscape of Intelligence in Database and Information Systems* [ÖSP94].

The power and expressiveness of a query model is characterized by its calculus, its algebra, its notion of safety, and its completeness. In this chapter, some of the recent literature on these topics is examined. These include:

- framework papers that discuss the qualities of query models and serve as guidelines for query model development,
- complex object query models that define an object algebra, an object calculus, and link the two with a proof of equivalence, and
- specific complex algebras that introduce object-oriented operators and semantics, which are exploited and expanded on by the algebra.

### 3.1.1 Query Model Frameworks

Although there is not one single universally accepted object model, a core set of features has been identified and presented in a number of manifestos [ABD+89, SRL+90]. Similar guidelines for the design of an object query model have recently appeared as well. They are summarized below.

Yu and Osborn [YO91] define a framework for evaluating the power and expressibility of object algebras. A set of categories is proposed for measuring the object-orientedness, expressiveness, formalness, database support, and performance of an object algebra. The framework is not meant to be all inclusive. In fact, some of the recommendations are contradictory requiring compromise in a design. To illustrate the practicality of the framework, four object algebras are compared within its dimensions. The framework serves as a useful guideline for developing object algebras.

The object query module specification [Bla91] of the DARPA Open OODB project [WBT92] offers a structured discussion of language features that an object query language should provide. Some of the more general properties that distinguish object query models from others are classified into "essential" and "non-essential" categories. This is supplemented by a more detailed discussion of specific features that are organized into a framework representing an overall design space for object query languages. This framework is intended to serve as a reference model and is expected to accommodate a broad spectrum of existing and future object query model specifications. The reference model is similar to that of Yu and Osborn [YO91] and assists in understanding the dimensions of object query model design by providing a common foundation for comparing and reasoning about existing object query language definitions. This in turn helps to identify common areas of agreement which may lead to an eventual standardization of object query model features.

In [ÖSP94], several issues relating to design alternatives for an object query model within the context of knowledge base systems are examined. This work focused on presenting a general discussion of the key issues concerning query model design, how a particular set of choices are carried through to an object query model definition, and the ramifications of the choices made. Several of the alternatives outlined in that report were addressed during the development of the query model described in this thesis.

### 3.1.2 Complete Object Query Models

Several object query models have been proposed. Many focus on a particular language aspect such as a calculus, an algebra or a user language. Others define a complete model,

but in order to deal with safety they restrict their languages in certain ways. Many query models are built on the nested *set-and-tuple* style structural model. The TIGUKAT query model differs in that it is a purely behavior-theoretic approach that defines the query model as an extensible part of the base object model. Some complete query models influencing the design of the TIGUKAT query model are examined below.

The emphasis of Straube and Özsu's [SÖ90a, Str91a] work was to illustrate the viability of developing a query processor for an object-oriented database system with comparable power and expressibility available in relational systems. A formal methodology for object-oriented query processing was developed in line with the relational paradigm. That is, a high-level declarative calculus is specified, optimization techniques on the calculus are developed, an object-oriented algebra is defined, translation of conjunctive calculus formulas with limited negation into the algebra is defined, algebraic type-checking and optimization strategies based on traditional and object-oriented transformation rules are developed, and an execution plan generation mechanism is designed that translates optimized algebraic expressions into an execution plan consisting of a series of packaged object manager calls. This approach increases efficiency of query processing by reducing the number of times the query processor must cross over to the object manager.

One contribution of their work is the definition of an object algebra, an object calculus, and the linking of the two with translations between them. The algebra-to-calculus translation is complete while the calculus-to-algebra transformation is not. The algebra defines a comprehensive set of *object-preserving* operators, but lacks *object-creating* operators such as *project, product* and *join*[2]. Furthermore, the classification of "safe" queries is limited to conjunctive queries without universal quantification and without negated existential quantifiers. In effect, this means that there is no allowance for universal quantification in the translation.

Abiteboul and Beeri [AB93] define a query model for complex objects that is based on a *set-and-tuple* data model. Their model includes set and tuple type constructors that relax the common restriction of alternating set and tuple structuring. This allows for arbitrary structures with the only restriction being that the last constructor used is a set constructor. Their calculus and algebra have complete definitions that include extended set operations such *set-collapse* for collapsing sets of sets, *powerset* for forming the powerset of a given set, and a higher-order restructuring operator called *replace* that generalizes relational projection and provides set-and-tuple restructuring capabilities. Safety in their model is defined constructively similar to the (*range*) *restricted* formulas in [Ull88]. They assume that a partial order on the variables has been defined and based on this ordering, they form range terms for variables. The range terms *restrict* the domains of the variables. Constructions are defined that build safe formulas from range terms using conjunction, disjunction, quantification, and negation. With this approach, safety is dependent on how the formula is constructed from the ground up and does not take advantage of the structure of the formula to recognize a broader class of queries. The class of safe queries recognizable by this approach is a strict subset of the *evaluable* class of queries, which is the basis of safety in the TIGUKAT query model. Although the formal work of [AB93] is sound, an algorithmic definition of safety and a calculus-to-algebra translation algorithm are not given. Furthermore, an effective solution for their transformation is not apparent since it requires the formation of large *DOM* sets for each variable appearing in the formula. A *DOM* set

[2] *Object-preserving* operators are limited to returning existing objects from an objectbase while *object-creating* operators may create new objects during their execution [SS90].

consists of all possible values from the database (and the constants in the query) that the variable can possibly take on. With complex valued variables allowed in their calculus, these *DOM* sets can become quite large and impractical to manage in an algorithmic solution.

### 3.1.3 Complex Object Algebras

An algebra is usually one of the first components developed for a query model. It defines a set of procedural operators for accessing the database. The design of these operators influences the number of opportunities for optimizing database access, which determines the efficiency that data can be retrieved. Several complex object algebras have appeared in recent years. These have evolved from the nested relational models and functional language approaches. A select number of proposals related to the algebra of TIGUKAT are discussed below.

The PROBE Data Model (PDM) [MD86] builds on the functional model and language of DAPLEX [Shi81]. PDM defines an algebra-based query model that is an extension of the relational algebra. It has a functional algebra that defines the traditional relational operators, plus an "apply-and-append" operator that provides a functional notion of the join operator. Apply-and-append accepts as arguments a relation (essentially a function) and an operator function over this relation. It returns a relation containing the columns of the original relation, plus an additional column holding the result of applying the operator function to each tuple of the original relation. Thus, the relation acts as the first operand of a join and the function defines the second operand, plus the join term. A similar approach is described by the OOAlgebra of OODAPLEX [Day89]. A variant of these approaches is defined by the TIGUKAT algebra because the uniform functional approach fits in naturally with the behavioral nature of the query model.

The object algebra of Shaw and Zdonik [SZ89, SZ90] is based on a *set-and-tuple* model and consistently extends the relational algebra with both object-preserving and object-creating operators. Their algebraic operators work on collections of objects that have parameterized set types. The algebra defines traditional set operations, along with a *flatten* operator for collapsing sets of sets. For tuples, *nest* and *unnest* operators are defined to restructure the representation of tuples as flat or nested relations. In addition to these, they define a traditional *select* operator, an *image* operator that applies a function to each object of a collection and returns the results as another collection, a *project* operator as an extension of *image* that returns a newly constructed tuple object for each object of a queried collection, and an *ojoin* operator to serve as a *Cartesian product* between two collections of objects. The result of an *ojoin* is a set of object pairs with the elements of each pair containing objects from the original collections that satisfy the join condition.

Osborn [Osb88] defines an algebra for an object-oriented model based on atomic objects, strongly typed aggregates (tuples) and both homogeneous and heterogeneous sets. A fairly comprehensive set of algebraic operators is defined. The algebra is multi-sorted since the operators are defined over multiple types (sorts) of objects and as a consequence are undefined for certain combinations of these types. Operators include traditional set operations, a *combine* operator that is equivalent to Cartesian product for sets and has a similar semantics for aggregates, a *partition* operator for carving up aggregate objects only, and a *choose* operator which is a generalization of the relational *select*. The objects created by *partition*, and the types to which they belong, are all grouped under a "CreatedAggregates" class. There is no relationship between CreatedAggregates and the classes from which the new objects are derived. Furthermore, the integration of the results of *combine* with the

existing lattice is not specified.

Kim [Kim89] defines the query model for Orion. The simple form of a query in this model is restricted to a single target class. Queries always return a new class with new object instances created from the objects in the target class. Thus, the algebra is strictly object-creating. The integration of new classes into the existing lattice is achieved by hanging them off the root. Reasoning about the type of the result class to better integrate it with the existing lattice is not defined. Single operand queries are too restrictive because they do not allow explicit joins. Therefore, the model extends queries over multiple target classes. However, there is a restriction on the domains of the "join attributes" of a query in that they must be identical or in a sub/supertype relation with one another. The result of a multiple-operand query, as with single-operand ones, is a new class with new object instances that hang off the root of the lattice.

Davis [Dav90] defines a formal object algebra that includes both object-preserving and object-creating operators. The traditional object-preserving set operators, along with an object-preserving *select* operator are defined and these are closed on sets (i.e., classes). The operands of a query based on these operators are classes and the result can be a new or existing class. The relative position in the class lattice of a new class created by a query is derived from the membership properties of the operand classes. A *membership normal form* (MNF) is defined for classes that describes the properties of a class's member objects. By combining the MNF formulas of the operand classes, a new MNF formula is created that describes the new class, along with its relative position in the lattice. A *property restriction* operator, similar to select, is used to extract objects with particular properties and form a class of these objects that is a subclass of the operand class. The algebra also defines object-creating *project* and *cross product* operators for "taking apart" and "putting together" objects, respectively. However, the objects and corresponding classes created by these operators are not integrated with the classes from which they were formed. Thus, the results of these operators are not classified as they are with the object-preserving operators. The TIGUKAT algebra includes a product operator and a form of behavioral projection that integrates results into the existing lattice. Moreover, every operator of the algebra does type inferencing on the result and integrates results with the existing lattice.

## 3.2 Query Model Overview

An identifying characteristic of the TIGUKAT query model is that it is defined as type and behavior extensions to the base object model. The uniform behavioral paradigm of the object model is carried through to the query model. Queries are defined as a specialization of functions and the algebraic operators are defined as behaviors on the type `T_collection`. Thus, the query model is a collection of objects (types, behaviors, functions, etc.) uniformly integrated with the base model. This approach has several advantages. For example, the query model is itself queryable, meaning a query may be posed on a collection of query objects or on the types and behaviors making up the query model definition (i.e., schema). Another advantage is that there is a single underlying semantics for both the object and query models resulting in a clean integration of the two. The mechanics of this integration is explained in Section 3.3.

A distinction is commonly made [SS90] between *object preserving* and *object creating* operations in object query models. An *object preserving* operator is one whose result contains only existing objects. That is, it does not create or modify objects in any way, either

explicitly or by side effects. The query formalism of Straube and Özsu [SÖ90a] considered only operations of the object preserving kind. On the other hand, *object creating* operators allow for the "taking apart" and "putting together" of objects into various new structures, with new identity, that are distinct from any existing objects in the objectbase. The objects created (especially persistent objects) must be integrated into the underlying type system, including any derived types or classes necessary for the consistent existence of these new objects.

The debate over object preserving versus object creating operators has strong arguments on both sides. On the one hand, object preserving operators are important because a query language must support these kinds of queries independent of its support for object creating operators. On the other hand, object creating operators allow otherwise unrelated objects to be combined in new ways, which is important for composing new relationships among objects and reorganizing information; this is applicable, for example, in knowledge base systems where knowledge is acquired by forming new relationships from the existing facts. Object creating operators introduce several problems that need to be resolved. First, new objects require a type that may not exist and must be integrated with the existing type lattice. Questions on how this type fits into the existing lattice and the behaviors it supports must be addressed. Second, the issue of query safety becomes more complex due to the introduction of new objects during query processing. For example, consider a query that creates new objects in one of its argument collections with every iteration of its evaluation. If the semantics were such that the query would continue to process these new objects, then more objects would be created and the query could go on indefinitely.

The terms *object-preserving* and *object-creating* require further clarification in the context of a uniform object model like TIGUKAT in which everything is an object. Queries in TIGUKAT (at minimum) always create and return a new collection object that represents the objects in the result of the query. Furthermore, a query may also create a new type object to go along with the collection if a proper type does not already exist. Thus, in TIGUKAT *all* queries are object-creating in one sense. If the result collection of a query contains objects created during the execution of the query, it is called a *target-creating* query; otherwise it is called a *target-preserving* query.

The user query language (TQL) has a syntax based on the SQL *select-from-where* structure, and formal semantics defined by the object calculus. Thus, it extends the relational query languages with object-oriented features. The definition language (TDL) provides functionality to create new types, classes, collections and behaviors; to define new functions in the query language or an external language; to add and remove behavior definitions to and from types; and to associate functions with behaviors on types. The control language (TCL) consists of a few simple commands for controlling a session with the query processor.

The object calculus has a logical foundation and its expressive power is outlined by the following characteristics. It defines predicates on collections (essentially sets) of objects and returns a collection of objects as a result. This property makes the language *closed* which is important for uniformity. It incorporates the behavioral paradigm of the object model and allows the retrieval of objects using nested behavioral applications, sometimes referred to as *path expressions* or *implicit joins*. It supports both *existential* and *universal* quantification over collections. It has a rigorous definition of safety based on the evaluable class of queries that is compile time checkable. Finally, it supports controlled creation and integration of new collections, types and objects into the existing schema.

The algebra has a behavioral (or functional) basis as opposed to the logical foundation of the calculus. Like the calculus, the algebra is *closed* on collections. The algebraic

operators are modeled as behaviors on the primitive type `T_collection`. Thus, any subtype of `T_collection` (such as classes) may be used as an operand of an algebra operator.

A desirable property of an object query model is that the algebra and calculus be equivalent in expressive power, meaning that all queries expressed in one language can also be expressed in the other. The theorems and proofs that show the equivalence of algebra and calculus are given in Section 3.7. Safety of the languages is addressed in Section 3.4.4.

## 3.3 Queries as Objects

Modeling queries as objects is a natural extension to the TIGUKAT object model. A type `T_query` is defined as a subtype of `T_function` in the primitive type system as illustrated in Figure 3.1. This means that queries have the status of *first class objects* and that they inherit all the behaviors and semantics of objects. Moreover, queries are a specialized kind of function object. This means they can be used as implementations of behaviors, they can be compiled, they can be executed and so on. The specialization of function and query is not opposite (i.e., `T_function` a subtype of `T_query`) because functions are general computationally complete programs and queries have a strict safety condition (see Section 3.4.4) that functions, in general, do not satisfy. Thus, functions are a more general form of extracting and manipulating information in an objectbase.

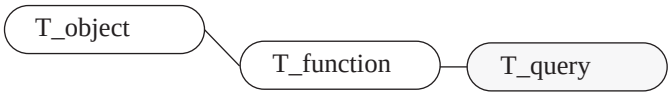



Figure 3.1: Query type extension to primitive type system.

Table 3.1 lists the signatures of behaviors defined on type `T_query`. The upper half of the table are the behaviors inherited from `T_function` and the lower half are the native behaviors defined by this type.

For example, functions have source code associated with them and the source code for a query is a query language statement such as TQL [PLÖS93a, PLÖS93b, Lip93]. The behavior *B_source* retrieves this language statement from the query object. Functions have a behavior *B_compile* that compiles the code. For a query, this involves translating the query statement into an algebra expression, optimizing it and generating an execution plan. Functions have a behavior *B_execute* that executes the compiled code. In general, for a query this means submitting the execution plan to the object manager for processing. Furthermore, queries have specialized behaviors such as *B_result*, which is a reference to the materialized query result (i.e., the actual result collection itself). If this result is made persistent, then the query is said to be *stored* and does not need to be re-evaluated the next time it is called upon to *B_execute* itself. Other behaviors relate to the extensible query optimizer [Muñ94] and include *B_initialOAPT* and *B_optimizedOAPT* for accessing the initial and optimized Object Algebra Processing Trees (OAPTs); *B_optimize* for initiating the optimization of a query using a particular search strategy; *B_searchStrategy* for accessing the search strategy used during optimization; *B_costModelType* for determining the cost model used for optimization; *B_transformations* for accessing the list of transformation rules used during optimization; *B_genExecPlan* for generating an execution plan for the compiled and optimized OAPT; *B_argMbrTypes* for accessing the membership types of the argument collections as opposed to *B_argTypes* which are the types of the collection objects

| Signatures | |
|---|---|
| *B_name*: | T_string |
| *B_argTypes*: | T_list⟨T_type⟩ |
| *B_resultType*: | T_type |
| *B_description*: | T_string |
| *B_source*: | T_string |
| *B_compile*: | T_object |
| *B_primitiveExecute*: | T_object → T_object |
| *B_executable*: | T_object |
| *B_basicExecute*: | T_list⟨T_object⟩ → T_object |
| *B_execute*: | T_list → T_object |
| *B_basicExecSave*: | T_list⟨T_object⟩ → T_object |
| *B_basicExecDontSave*: | T_list⟨T_object⟩ → T_object |
| *B_initialOAPT*: | T_algOp |
| *B_optimizedOAPT*: | T_collection⟨T_algOp⟩ |
| *B_searchStrategy*: | T_searchStrategy |
| *B_transformations*: | T_list⟨T_algEqRule⟩ |
| *B_costModelType*: | T_integer |
| *B_argMbrTypes*: | T_list⟨T_type⟩ |
| *B_resultMbrType*: | T_type |
| *B_optimize*: | T_searchStrategy → T_algOp → T_collection⟨T_algOp⟩ |
| *B_genExecPlan*: | T_algOp → T_function |
| *B_execPlanFamily*: | T_collection⟨T_function⟩ |
| *B_budgetOpt*: | T_integer |
| *B_lastOpt*: | T_date |
| *B_lastExec*: | T_date |
| *B_result*: | T_object |

Table 3.1: Behavior signatures for type T_query. Upper half are inherited from T_function. Lower half are native to this type.

themselves; *B_resultMbrType* for accessing the membership type of the result collection as opposed to *B_resultType*, which is the type of the collection; and several other behaviors, including ones for keeping various statistics about queries. As mentioned earlier, these behaviors relate to the extensible query optimizer which is reported elsewhere [Muñ94].

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

1. Queries are *first class objects*, meaning they support the uniform semantics of objects and are maintained within the objectbase as just another kind of object.

2. Since queries are objects, they can be queried and can be operated upon by other behaviors. This is useful for retrieving information about queries, generating statistics about the performance of queries and in defining extensible optimization techniques on query objects.

3. Queries are uniformly integrated with the operational semantics of the model so that

queries can be used as implementations of behaviors (i.e., the result of applying a behavior to an object can trigger the execution of a query).

4. The type `T_query` can be further specialized by subtyping. This can be useful in extending the general class of queries into additional subclasses, each with its own unique characteristics, and to incrementally develop the characteristics of new kinds of queries as they are discovered. For example, in the design of the query optimizer [Muñ94], `T_query` is subtyped by `T_adhocQuery` and `T_productionQuery`, and each defines a specialized evaluation strategy for queries. That is, *ad hoc* queries are interpreted without incurring high compile-time optimization strategies while, on the other hand, production queries are compiled once and then executed many times. Thus, more time is spent on optimizing production queries over *ad hoc* queries.

## 3.4 The Object Calculus

It is well recognized that a declarative query facility is an essential component of any database management system; object-oriented systems are no exception. In this chapter, a high-level behavioral object calculus with first-order semantics is presented.

In order to maintain the uniformity of the behavioral object model within the query model, the behavioral abstraction paradigm is carried through into the calculus. The logical foundation of the calculus includes a function symbol to incorporate the behavioral nature of the object model. This allows the use of general path expressions in the calculus. The expressive power of the calculus is equivalent to the first-order calculus, but some queries within this domain may not be *safe*. The safety of the calculus is based on the *evaluable* class of queries [GT91], which is arguably the largest decidable subclass of the domain independent class [Mak81]. The evaluable class is extended in this thesis by making use of *object generators* for equality and membership atoms, which relaxes the requirement of specifying explicit *range* expressions for each variable.

### 3.4.1 Formal Object Calculus

The *first-order theory* of the object calculus is presented, which establishes the *well-formed formulae* of the language. Following this, the augmentations to the theory that form *object calculus expressions* (OCEs) are described. These represent the class of declarative queries that can be posed on an objectbase.

The alphabet of the object calculus consists of the following symbols:

| | |
|---|---|
| Object constants: | $a, b, c, d$ |
| Object variables: | $o, p, q, u, v, x, y, z$ |
| Predicate symbols | |
| monadic: | $C, P, Q, R, S, T$ |
| dyadic: | $=, \neq, \in, \notin$ |
| $n$-ary: | $Eval$ |
| Function symbols: | $\beta$ |
| Logical connectives: | $\exists, \forall, \wedge, \vee, \neg$ |
| Delimiters: | ( ) , |

Note that the object constants, object variables, monadic predicates and function symbols may be subscripted (e.g., $a_3, o_i, C_n, \beta_1$,etc.). In addition, a vector notation $\vec{s}$ is adopted to denote a countably infinite list of symbols $s_1, s_2, \ldots, s_n$ where $n \geq 0$.

From object constants and object variables the syntax and semantics of the function symbol $\beta$ called a *behavioral specification* (Bspec) is developed. A *term* is an object constant, an object variable or a Bspec. A Bspec is an $n$+2-ary function $\beta(s, b, \vec{t})$ where $s$ and each $t_i$ denote terms and where $b$ is an object constant. For $n = 0$, $\beta(s, b)$ is used without loss of generality.

The ordered list of terms $s, b, \vec{t}$ is considered to be *behaviorally consistent* if and only if the following properties hold:

1. $b$ is an object constant denoting a behavior, meaning $b$ is not allowed to range over behaviors (functions) which ensures a first-order semantics when incorporated into a language with quantification;

2. the type of the object denoted by $s$ defines behavior $b$ as part of its interface, meaning $b$ is applicable to $s$ because it is defined on the type of $s$;

3. $\vec{t}$ is compatible with the arity of the argument list for behavior $b$, meaning the number of arguments expected by $b$ is equivalent to the number of terms in $\vec{t}$; and

4. the types of the objects denoted by $\vec{t}$ are compatible with the argument types of behavior $b$, meaning the types of the terms are compatible with the argument types of $b$.

A Bspec $\beta(s, b, \vec{t})$ is *consistent* if and only if $s, b, \vec{t}$ are *behaviorally consistent*. In TIGUKAT, every object knows its type and therefore, the consistency of a Bspec can be determined at compile time.

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

The "evaluation" of Bspecs has the following logical formation. The $n$+3-ary predicate $Eval(R, s, b, \vec{t})$ is introduced as an axiom in the language such that $Eval(R, s, b, \vec{t})$ is true if and only if $R$ denotes the "result" of applying behavior $b$ to the object denoted by term $s$ using terms $\vec{t}$ as arguments. The function symbol $\beta(s, b, \vec{t})$ is a logical representation of $R$. The $Eval$ predicate also serves as an enforcement of the consistency property of Bspecs. In the remainder of this thesis, only consistent Bspecs are considered.

Bspecs may be composed. This provides the capability of building *path expressions* in queries. For example, given the object constants emp, *B_department*, and *B_budget* with the obvious semantics, the Bspec $\beta(\beta(\mathsf{emp}, B\_department), B\_budget)$ can be composed, which denotes the object representing the annual budget of the department that employee emp works in. Also note that the example Bspec has the properties of a *ground term* (see Definition 3.1 below).

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

As shown by the above example, many path expression formations often include a series of behaviors with the semantics that the result of the first behavior be used as the input to

the second and so on. Such a sequence of **m**ultiple **op**erations is called a **mop** [SÖ90a] and is equivalent to a Bspec. The multi-operation dot notation $<\vec{s}>.b_1.b_2..b_m$ is introduced to denote a multi-operation resulting in the application of behavior object constants $b_1.b_2..b_m$ using objects denoted by terms $\vec{s}$ as arguments. Furthermore, $<\vec{s}>.\vec{b}$ is used as a shorthand to denote a multi-operation where the number and ordering of the behaviors are immaterial.

To illustrate the processing of a **mop**, consider the following multi-operation:

$$<s_1, s_2, \ldots, s_n>.b_1.b_2..b_m$$

Let $k_i$ denote the number of parameters[3] defined by behavior $b_i$, let $r_i$ designate the intermediate object denoted by the Bspec formation of behavior $b_i$ and let $r$ denote the final result of the *mop*. Procedurely, a **mop** is processed as follows where "$\leftarrow$" denotes assignment:

$$\begin{aligned}
r_1 &\leftarrow s_1.b_1(s_2, \ldots, s_{k_1+1}) \\
r_2 &\leftarrow r_1.b_2(s_{k_1+2}, \ldots, s_{(k_1+k_2+1)}) \\
&\vdots \qquad\qquad \vdots \\
r_i &\leftarrow r_{i-1}.b_i(s_{(\sum_{j=1}^{i-1} k_j)+2}, \ldots, s_{(\sum_{j=1}^{i} k_j)+1} \\
&\vdots \qquad\qquad \vdots \\
r = r_m &\leftarrow r_{m-1}.b_m(s_{(\sum_{j=1}^{m-1} k_j)+2}, \ldots, s_n)
\end{aligned}$$

The above sequence of behavioral application making up the **mop** is illustrated in Figure 3.2.

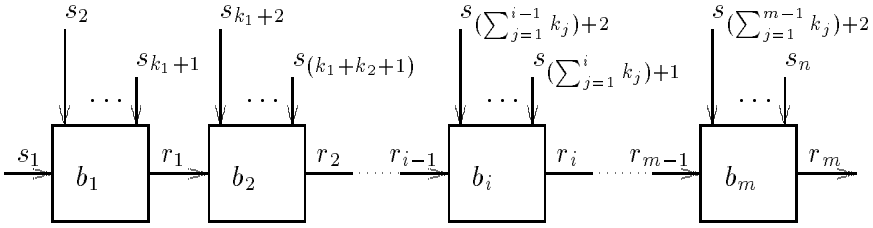



Figure 3.2: Sequence of behavioral applications making up a *mop*.

Bspecs and mops are equivalent forms of representation. One form can be freely transformed into the other and results established using one form also hold for the other. This result is important since one can transform between the formal calculus and "simpler" language notations. The equivalence is formalized by the following lemma.

**Lemma 3.1** Bspecs and mops are equivalent representations.
**Proof:** Trivial. The semantics of Bspecs and mops are defined above. Due to the following equivalence mappings between Bspecs and mops where $s$ and $t$ represent terms and $b$ represents behavior constants:

$$\beta(s, b, \vec{t}) \equiv <s, \vec{t}>.b \tag{3.1}$$

$$<\vec{t}>.b.\vec{b} \equiv <<\vec{t}>.b>.\vec{b} \tag{3.2}$$

[3]Here the *parameters* refer to the objects supplied to the behavior, not including the initial object to which the behavior is being applied.

The first mapping shows that every Bspec can be replaced by an equivalent mop over a single behavior and vice versa. The second mapping shows the unnesting of mops over multiple behaviors into an equivalent series of single behavior mops, which can be transformed by the first mapping. □

The notions of constants and variables are generalized to include Bspecs by defining *ground terms* and *variable terms* as follows:

**Definition 3.1** *Ground Term:* A *ground term* is recursively defined as follows:

1. every *object constant* is a *ground term*;
2. if $\beta(s, b, \vec{t})$ is a consistent Bspec and all of $s, \vec{t}$ are *ground terms* (note that $b$ must be a ground term by the definition of Bspec), then $\beta(s, b, \vec{t})$ is a *ground term*;
3. nothing else is a *ground term*.

From this point on, symbols defined as denoting an object constant, including symbols $a, b, c, d$, are extended to include ground terms as well. Any term that is not a ground term is called a *variable term* since it must contain at least one object variable. If $\vec{o}$ are the object variables appearing in some term $r$, then $r$ is called a *variable term over* $\vec{o}$. The variables can be thought of as the parameters of the term. If $r$ is the object variable $o$, then $r$ is a variable term over $o$. If $r$ is a term defined by Bspec $s.b(\vec{t})$ and $\vec{o}$ represents the object variables appearing in the Bspec, then $r$ is a variable term over $\vec{o}$. The notation $r\{\vec{o}\}$ is used to denote that $r$ is a variable term over $\vec{o}$. This notation is generalized to $\beta\{\vec{o}\}$ when the form of the term is immaterial. If $\vec{o}$ is empty, then $\beta\{\}$ denotes a generic ground term.

The *atomic formulas* or *atoms* are the building blocks of calculus expressions. Every atom has an equivalent Bspec (and hence **mop**) representation. Atoms are identified because they represent the fundamental predicates of the calculus and are used in translating a query to the algebra, which can then be optimized and executed. The atoms of the TIGUKAT calculus consist of the following:

**Range Atom:** $C(o)$ is called a *range atom* for $o$ where $C$ corresponds to a unary predicate representing a collection and $o$ denotes an object variable. $C$ is called the *range* of $o$. A range atom is true if and only if $o$ denotes an object in collection $C$. The semantics of this atom in a query is to have variable $o$ bind to (or range over) the objects in the collection denoted by $C$. When $C$ is defined for a class, it denotes the deep extent of the class and the notation is extended to include $C^+(o)$, which is true if and only if $o$ denotes an object in the *shallow extent* of the class. One may think of $C^+$ as a separate monadic predicate for specifying the shallow range of $o$. The Bspec representation for the range atom is $\beta(C, B\_elementOf, o)$ where $B\_elementOf$ is the collection membership behavior as defined in Appendix B. Range atom specifications of the form $C(s)$ where $s$ is a term denoting an object constant or Bspec (i.e., not an object variable) are handled by membership atoms defined below.

**Equality Atom:** $s = t$ is a built-in predicate called an *equality atom* where $s$ and $t$ are terms. The predicate is true if and only if the object denoted by term $s$ is object identity equal to the object denoted by term $t$. The semantics of this atom in a query is to test the object identity equality of $s$ and $t$ and return true if they are equal or false otherwise. This atom is type consistent for all objects since all objects must support

an object identity equality behavior. Note, as a syntactic convenience, an equality atom where both terms are boolean and where one of the terms is the object constant **true**, say $s = \textsf{true}$ where $s$ is boolean, is simplified to $s$. If one of the terms is the object constant **false**, the atom specification is simplified to $\neg s$. The Bspec representation for the equality atom is $\beta(s, B\_equal, t)$ where *B_equal* is the object equality behavior defined in Section 2.4.3. The built-in predicate $s \neq t$ is the complement of equality.

**Membership Atom:** $s \in t$ is a built-in predicate called a *membership atom* where $s$ and $t$ are terms and $t$ is a term denoting a collection. The predicate is true if and only if the object denoted by $s$ is an element of the collection denoted by $t$. The Bspec representation for the membership atom is $\beta(t, B\_elementOf, s)$. The semantics of this atom in a query is to test if $s$ is an element of $t$ and return **true** if it is or **false** otherwise. Note that a range specification of the form $C(s)$ where $s$ is an object constant or Bspec (i.e., not an object variable) is represented as a membership atom $s \in C'$ where $C'$ is a constant denoting the collection represented by predicate $C$. The built-in predicate $s \notin t$ is the complement of membership.

**Generating Atom:** An equality atom of the form $o = t$ or a membership atom $o \in t$, where $o$ is an object variable, $t$ is an appropriate term for the atom, and $o$ does not appear in $t$, are called *generating atoms* for $o$. They are so named because the object denotations for $o$ can be *generated* from $t$. $o$ is called the *generated variable* and $t$ is called the *generator*. The Bspec representations for generating atoms are $\beta(o, B\_equal, t)$ and $\beta(t, B\_elementOf, o)$. The semantics of the $o = t$ generating atom in a query is to bind $o$ to the object denoted by $t$. The semantics of the $o \in t$ generating atom is to have $o$ bind to (or range over) the objects in the collection denoted by $t$. Any atom that is not a generating atom is called a *restriction atom* and any variable that is not generated is called a *restriction variable* because they are used to restrict objects returned by a query.

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

The choice of atoms may seem restrictive when compared to other calculi such as the tuple relational calculus that defines a greater variety of comparison predicates including $=, <, \leq, >$, and $\geq$. An identifying characteristic of the TIGUKAT calculus is that it is strictly behavioral and does not define explicit value-based comparisons of objects or their subcomponents. Thus, operations such as $<, >, \geq, \leq$ must be defined as behaviors on the respective types of objects that are to be compared. The only comparison predicates defined are object identity equality and membership. However, type implementors can specialize the behaviors for these comparison predicates in their own types (e.g., value based comparisons) that are of most utility to them. For example, a form of "structural equality" on Cartesian product types that compares two product objects based on the pairwise equality of their respective component objects can be defined.

From atoms, the definition of a *first-order well-formed-formula* or simply *formula* (abbreviated WFF) of the object calculus are built. WFFs are defined in terms of *free* and *bound* object variables. An object variable is *bound* in a formula if it has been previously introduced by the quantifier $\exists$ or $\forall$. If the variable has not been introduced with a quantifier it is *free* in the formula. WFFs are defined recursively as follows:

1. Every atom is a formula. All object variables in the atom are free in the formula.

2. If $\psi$ is a formula, then $\neg\psi$ is a formula. Object variables are free or bound in $\neg\psi$ as they are free or bound in $\psi$.

3. If $\psi_1$ and $\psi_2$ are formulas, then $\psi_1 \wedge \psi_2$ and $\psi_1 \vee \psi_2$ are formulas. Object variables are free or bound in $\psi_1 \wedge \psi_2$ and $\psi_1 \vee \psi_2$ as they are free or bound in $\psi_1$ or $\psi_2$.

4. If $\psi$ is a formula, then $\exists o(\psi)$ is a formula. Free occurrences of $o$ in $\psi$ are bound to $\exists o$ in $\exists o(\psi)$.

5. If $\psi$ is a formula, then $\forall o(\psi)$ is a formula. Free occurrences of $o$ in $\psi$ are bound to $\forall o$ in $\forall o(\psi)$.

6. Nothing else is a formula.

In the remainder of this thesis, $A, B, F, G$ and $\psi, \omega$ are used to denote formulas and subformulas. The relation "$A \stackrel{\text{def}}{=} F$" means symbol $A$ "is defined by" the expression $F$. This is used to associate formula symbols with formulas. Furthermore, $A(x)$ denotes that variable $x$ is free in formula $A$. Formulas may be enclosed in parenthesis to indicate order of precedence. In the absence of parenthesis, the following precedence hierarchy is adopted with the highest precedence at the top:

$$\begin{array}{c} \neg, \exists, \forall \\ \wedge \\ \vee \end{array}$$

### 3.4.2 Calculus Queries

Several classifications of object-oriented queries have been made. One class of queries deals only with behaviors that are *side-effect free*. A behavior is said to be *side-effect free* if it does not modify the state of any object or create new objects during its execution. This property is too restrictive in the context of the TIGUKAT model since all operations (including the algebraic operators) are uniformly managed as behaviors. At minimum, a query always returns a new collection as a result and in certain cases generates a new type for the collection as well. Thus, there is a small set of predefined behaviors that manage the controlled creation of collections (and possibly types) as their side effects. These behaviors include the algebraic operators and the primitive behaviors for collection creation and construction. The notation $newcoll(o_1, \ldots, o_n)$ is used as a shorthand to represent the creation of a collection containing objects $o_1, \ldots, o_n$. The primitive sequence of behavioral applications corresponding to this notation is as follows:

$$\textbf{C\_collection}.B\_new.B\_insert(o_1)\ldots B\_insert(o_n)$$

That is, a new empty collection is created and then each object $o_i$ is added to the collection in turn. The result is a collection containing objects $o_1, \ldots, o_n$. A compiler could optimize this series of $n+1$ behavioral applications into a single internal primitive collection creation operation since collections are part of the primitive model.

All user-defined behaviors appearing in calculus expressions are assumed to be side-effect free. In other words, all user-defined behaviors appearing in calculus expressions must be retrieval oriented.

A target-preserving query is an *object calculus expression* (OCE) of the form $\{t \mid \psi\}$ where $t$ is a target term consisting of a single variable, say $o$, possibly indexed by a set of

behaviors, $\psi$ is a WFF with $o$ as the only free variable, and *all* behaviors in the expression are side-effect free (or retrieval oriented). The semantics of a target-preserving query is to return a collection of existing objects that satisfy the formula $\psi$.

Indexed variables are of the form $o[\mathcal{B}]$ where $\mathcal{B}$ represents a subset of behaviors defined on the type of variable $o$, union the behaviors defined on type `T_object`. The union with `T_object` is necessary since every object must support the behaviors of `T_object`. The semantics of indexed terms is to *project* over the behaviors in $\mathcal{B}$ for variable $o$ creating a new type for the result. Following a projection, the membership type of the result collection will be a type that only defines the behaviors in $\mathcal{B}$. This restricts the behaviors that can, in general, be applied to the members of the result collection.

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

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

All queries that are not target-preserving are target-creating. The notation of OCEs is extended for target-creating queries to include the form $\{t_1, \ldots, t_k \mid \psi\}$ where the set of variables appearing in (possibly indexed) target terms $t_1, \ldots, t_k$ is precisely the set of free variables, say $\vec{o}$, in the WFF $\psi$. This form is a generalization of the target-preserving kind by allowing $k \geq 2$ target terms over $\vec{o}$ distinct object variables. The semantics of a target-creating query is to return a collection of product objects created by joining all permutations of $t_1$ through $t_k$ that satisfy $\psi$.

Assume in the previous example that (department, employee) pairs should be returned instead of just departments. Further assume that the employee objects are projected over the behavior *B_age*. The target-creating query that produces this result is as follows:

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

Additional examples of both target-preserving and target-creating queries are given in Section 3.6.

### 3.4.3 Expressive Power of Calculus Queries

The general expressive power of the TIGUKAT calculus is defined by the following theorem:

**Theorem 3.1** Every query expressible in the first-order calculus is expressible in the TIGUKAT calculus.

**Proof:** An object calculus expression (OCE) of the TIGUKAT calculus consists of two components: a list of (possibly indexed) target terms and a first-order well formed formula. The second component allows an OCE to express any first-order calculus expression. Thus, the general expressive power of the TIGUKAT object calculus is equivalent to the first-order calculus. Any first-order calculus formula can be translated to an OCE by simply adding target terms for every free variable in the formula. Conversely, an OCE is translated to a first-order calculus formula by dropping the target terms. There may be an additional

translation between the predicate and atom representations of the first-order calculus formula and the first-order formula of an OCE, but this can be represented with a trivial naming mapping. □

The restriction that OCEs must include only side-effect free behaviors does not pose problems since this is a universal assumption that cannot, in general, be tested and must be accepted axiomatically.

Some statements that can be expressed in a first-order language may not have any reasonable interpretation and, therefore, cannot be effectively executed by the query processor. The unreasonable statements should be identified and rejected with an indication that they cannot be processed. This raises the issue of *safety*, which involves defining a subset of the first-order statements that can be identified and processed in polynomial time. Safety and the definition of a safe subset of the TIGUKAT calculus are topics of Section 3.4.4.

### 3.4.4 Safety of Object Calculus Expressions

A traditional notion in relational database systems is that "reasonable" queries are ones whose correct answers contain values that are limited to the constants that appear in the query or the database relations that appear in the query. A corresponding notion in an object model is that reasonable queries produce correct answers that contain objects which are limited to the objects appearing the query or in the collections that appear in the query. Unary predicates $C(o)$ are defined for the finite collections and classes appearing in the objectbase. These are used to range over the elements of a collection. The collection represented by the complement of a predicate is assumed to be infinite (i.e., $\neg C(o)$ is infinite for all predicates $C$).

The object calculus is very expressive and allows for the formation of queries that have no "reasonable" interpretation. For example, the complement of a predicate $\neg C(o)$ holds for arbitrary objects $o$ that are not in the collection $C$. Another problematic query is the one that adds objects to collections over which it is ranging. This has the effect of updating the predicate on each iteration. These kinds of queries are considered "unreasonable" and an implementation should strictly avoid processing such constructs. Therefore, a criterion of *safety* is defined that consists of tests based on the structure of the formula (i.e., its syntax) to check if a formula is reasonable. Only safe queries are processed and all others are rejected. The general notion of safety is defined as follows:

**Definition 3.2** *Safety:* An expression is considered *safe* if it can be evaluated in finite time and produces finite output [OW89].

The above definition is a semantic one that raises the problem of finding an efficient solution for determining whether an arbitrary expression is safe or not. In other words, there is a need for a syntactic check that can be performed on any arbitrary formula and can determine, in polynomial time, whether the given formula is safe or not. The safe formulas are the ones translated to an algebra, optimized and executed. Since the implementations of behaviors can be arbitrary code, safety can only be guaranteed up to Bspec evaluation. That is, there are no mechanisms to guarantee the termination of a function that may be called as part of a behavior being applied to an object.

The first safety check is on the calculus formula and determines the *domain independence* of the formula. The second check is based on the operators of an equivalent algebra expression for the formula and determines the *operand finiteness* of a query, meaning it

checks that objects aren't being added to operand collections or classes of the operator. If the query fails either test, it is rejected. The domain independence form of "safety" is discussed first, followed by a discussion of operand finiteness in queries.

The class of *domain independent* formulas [Mak81, Fag82] is recognized as being the largest class of "reasonable" queries. However, the undecidability of this class is well known; Nicolas and Demolombe [ND82] have shown domain independence to be equivalent to the class of *definite* formulas defined by Kuhns [Kuh67], which has been shown to be not recursive by DiPaola [DiP69].

Many decidable subclasses of the domain independent class have been proposed. The class of conjunctive queries are those that include only $\exists$ and $\wedge$ connectives and represents one of the simplest "reasonable" subclasses shown to be decidable [Ull82]. Larger decidable subclasses augment conjunctive queries with negation and disjunction. Several object calculi proposals have defined safety in the context of conjunctive queries with disjunction and restricted forms of negation [SÖ90a, Cha92]. These proposals define a broader range of safe queries, however, more general classes have been identified. The class of *evaluable* queries as first proposed by Demolombe [Dem81] and later examined by van Gelder and Topor [GT87, GT91] is argued to be the largest decidable subclass of domain independent queries. In the TIGUKAT query model, the evaluable class is used as the base set of safe queries that can be translated into the object algebra. The class of *range restricted* queries [Dem82] has been shown to be equivalent to the evaluable class [GT91]. A strict subclass of the range restricted class (hence the evaluable class) is essentially the basis of safety in the structural query model of Abiteboul and Beeri [AB93]. Furthermore, their definition assumes the existence of a partial order on the variables in a calculus formula such that all variables are *restricted*. An indication of how to construct a proper partial ordering from a given formula is not presented. The safety model of TIGUKAT also defines a partial order and the first part of the translation from calculus to algebra (see Section 3.7.2) constructs this ordering.

The class of evaluable queries can be defined in terms of the two relations *gen* and *con* (see Figure 3.3) between variables and (sub)formulas. These relations were introduced by Gelder and Topor [GT87, GT91] in the form of logical rules.

Intuitively, $gen(x, A)$ means that formula $A$ can *generate* all the needed values of variable $x$ that contribute to making $A$ true and that there are only a finite number of these values. In other words, if $gen(x, A(x, \vec{y}))$ holds and $A(c, \vec{d})$ is true for some variable assignment $x = c$ and $\vec{y} = \vec{d}$, then one can conclude that $c$ is an element of a finite collection of objects derivable from the formula $A$ itself. If $con(x, A(x, \vec{y}))$ holds, then the variable $x$ is said to be *constrained* in $A$, meaning that $x$ is generated in every disjunct of $A$ in which $x$ appears. The *con* rules subsume the *gen* rules. Thus, it is clear that $gen(x, A)$ implies $con(x, A)$, but $con(x, A)$ does not imply $gen(x, A)$.

These rules are extended by adding a *gdb* relation that makes use of generating atoms in formulas. The *gdb* relation relies on a globally accessed partial order denoted $<_F$. This partial order consists of pairs $(x, N)$ where $x$ is a variable and $N$ is a positive integer or the symbol $\infty$. The symbol $<_F$ is used in the *gdb* rules as an infix dyadic predicate on the variables appearing in the partial order $<_F$. This predicate is defined as follows:

**Definition 3.3** *Ordering Predicate* ($<_F$)*:* For any two elements $(x, N_x)$ and $(y, N_y)$ appearing in the partial order $<_F$, the predicate $x <_F y$ is defined by the following table

where $n$ and $m$ denote positive integers and $m$ is greater than zero:

| $N_x$ | $N_y$ | $x <_F y$ |
|---|---|---|
| $\infty$ | $\infty$ | false |
| $\infty$ | $n$ | false |
| $n$ | $\infty$ | true |
| $n$ | $n+m$ | true |
| $n+m$ | $n$ | false |
| $n$ | $n$ | false |

Figure 3.3 shows the rules for the *gdb* relation and the extended *gen* and *con* relations. The partial order used by the *gdb* relation is built from the atoms in a calculus formula $F$ during the first step in the translation from the calculus to the algebra. The partial order is constructed to produce a representation of the generating atom dependencies between variables in a formula $F$. If predicate $x <_F y$ holds for the partial order $<_F$, this means that variable $x$ is not dependent on variable $y$ and that $x$ potentially generates values for $y$ in formula $F$. For example, the partial order for the formula:

$$F \stackrel{\text{def}}{=} \exists x(\mathbf{C\_emp}(x) \wedge y = x.B\_name)$$

is $<_F \stackrel{\text{def}}{=} \{(x,0),(y,1)\}$ since $x$ is generated independently of $y$ from **C_emp** and $y$ is generated using $x$ in $y = x.B\_name$. The reason $x$ "potentially" generates $y$ is clear from the following example. Consider the formula:

$$F' \stackrel{\text{def}}{=} \exists x \exists w(\mathbf{C\_emp}(x) \wedge y = x.B\_name \wedge \mathbf{C\_emp}(w) \wedge z = w.B\_age)$$

The partial order for this formula is $<_{F'} \stackrel{\text{def}}{=} \{(x,0),(w,0),(y,1),(z,1)\}$. Now, $x <_{F'} z$ holds and $x$ is not dependent on $z$, but $x$ does not generate objects for $z$ in $F'$. Thus, $x$ is only a potential generator for $z$.

The additional predicates and functions that appear within the rules of Figure 3.3 are defined as follows:

- Predicate *edb*$(A)$ holds if one of the following conditions is met:
  1. formula $A$ is a range atom of the form $C(x)$ where predicate symbol $C$ represents a finite collection;
  2. formula $A$ is an equality atom of the form $x = c$ where $c$ is a ground term; or
  3. formula $A$ is a membership atom of the form $x \in c$ where $c$ is a ground term representing a finite collection.
- Predicate *free*$(x, A)$ holds if variable $x$ appears as a free variable in formula $A$.
- Predicate *notfree*$(x, A)$ holds if variable $x$ is bound in formula $A$ or if $x$ does not appear in $A$.
- Predicate *distinct*$(x, y)$ holds if $x$ and $y$ are different variables.

| | | |
|---|---|---|
| $gdb(x, x = y)$ | **if** | $y <_F x$ |
| $gdb(x, x = \beta\{\vec{y}\})$ | **if** | $\vec{y} <_F x$ |
| $gdb(x, x \in y)$ | **if** | $y <_F x$ |
| $gdb(x, x \in \beta\{\vec{y}\})$ | **if** | $\vec{y} <_F x$ |

---

| | | |
|---|---|---|
| $gen(x, A)$ | **if** | $edb(A)$ **and** $free(x, A)$ |
| $gen(x, A)$ | **if** | $gdb(x, A)$ |
| | | |
| $gen(x, \neg A)$ | **if** | $gen(x, pushnot(\neg A))$ |
| $gen(x, \exists y A)$ | **if** | $distinct(x, y)$ **and** $gen(x, A)$ |
| $gen(x, \forall y A)$ | **if** | $distinct(x, y)$ **and** $gen(x, A)$ |
| $gen(x, A \vee B)$ | **if** | $gen(x, A)$ **and** $gen(x, B)$ |
| $gen(x, A \wedge B)$ | **if** | $gen(x, A)$ |
| $gen(x, A \wedge B)$ | **if** | $gen(x, B)$ |

---

| | | |
|---|---|---|
| $con(x, A)$ | **if** | $edb(A)$ **and** $free(x, A)$ |
| $con(x, A)$ | **if** | $gdb(x, A)$ |
| $con(x, A)$ | **if** | $notfree(x, A)$ |
| | | |
| $con(x, \neg A)$ | **if** | $con(x, pushnot(\neg A))$ |
| $con(x, \exists y A)$ | **if** | $distinct(x, y)$ **and** $con(x, A)$ |
| $con(x, \forall y A)$ | **if** | $distinct(x, y)$ **and** $con(x, A)$ |
| $con(x, A \vee B)$ | **if** | $con(x, A)$ **and** $con(x, B)$ |
| $con(x, A \wedge B)$ | **if** | $gen(x, A)$ |
| $con(x, A \wedge B)$ | **if** | $gen(x, B)$ |
| $con(x, A \wedge B)$ | **if** | $con(x, A)$ **and** $con(x, B)$ |

Figure 3.3: Logical rules that define the *gen* and *con* relations.

- Function $pushnot(\neg A)$ represents a formula $B$ (provided $edb(A)$ does not hold) that is evaluated as follows:

| $\neg A$ | $B$ |
|---|---|
| $\neg(A_1 \wedge A_2)$ | $(\neg A_1) \wedge (\neg A_2)$ |
| $\neg(A_1 \vee A_2)$ | $(\neg A_1) \vee (\neg A_2)$ |
| $\neg\exists x\, A_1$ | $\forall x \neg A_1$ |
| $\neg\forall x\, A_1$ | $\exists x \neg A_1$ |
| $\neg\neg A_1$ | $A_1$ |
| $\neg(s = t)$ | $s \neq t$ |
| $\neg(s \neq t)$ | $s = t$ |
| $\neg(s \in t)$ | $s \notin t$ |
| $\neg(s \notin t)$ | $s \in t$ |

  If $edb(A)$ holds, then $pushnot(\neg A)$ represents a formula, say $\perp$, that causes the corresponding *gen* or *con* predicate to fail.

From the relations of *gen* and *con*, the class of *evaluable* [GT91] formulas is defined

below. The class of formulas satisfying this definition (or which can be rewritten to satisfy the definition) is exactly the class of "safe" formulas of the calculus.

**Definition 3.4** *Evaluable:* A formula $F$ is *evaluable* or has the *evaluable property* if the following conditions are met:

1. For every variable $x$ that is free in $F$, $gen(x, F)$ holds.
2. For every subformula $\exists x A$ of $F$, $con(x, A)$ holds.
3. For every subformula $\forall x A$ of $F$, $con(x, \neg A)$ holds.

This definition provides an efficient, syntactic approach for determining whether a given formula is evaluable or not: simply apply the appropriate *gen* and *con* rules to the formula and subformulas. This definition is extended to object calculus expressions (OCEs) by stating that an OCE $\{\vec{t} \mid \psi\}$ where $\vec{t}$ contains at least one target term, is evaluable if the formula $\psi$ is evaluable in the sense of Definition 3.4. This establishes the decision mechanism for accepting or rejecting any arbitrary query posed as an OCE. For example, assuming all range predicates represent finite collections, the following OCE is evaluable:

$$\{o \mid C(o) \wedge \exists p(P(p) \vee \neg Q(o))\}$$

while:

$$\{o \mid C(o) \wedge \exists p(\neg P(p) \wedge p.B\_something = o.B\_something)\}$$

is not because $con(p, \neg P(p) \wedge p.B\_something = o.B\_something)$ does not hold. Note that the evaluable OCE above as given is an example of a formula that is safe in the evaluable class, but is unsafe in the (range) restricted class as defined by [AB93].

Without a partial order defined (i.e., we cannot make use of the *gdb* predicate), formulas satisfying Definition 3.4 are known as *strict-sense evaluable* [GT91] because of the conservative approach taken towards the built-in equality and membership predicates: $gen(x, x\theta y)$ and $con(x, x\theta y)$ where $\theta$ is one of $=, \in$ never hold. The strict-sense evaluable queries are the class considered in [GT91]. However, they realized that many formulas are evaluable despite this conservative approach. They presented transformations that remove some instances of equality ($=$) and yield an "equality reduced" form. However, a more general solution was needed for the TIGUKAT query model to deal with Bspecs and generating atoms that were not part of their work. The introduction of the *gdb* predicate and the formation of the partial order $<_F$ consistently extends the class of evaluable queries to a larger class recognized in [GT91]. Formulas that fail strict-sense evaluability, but can be made evaluable through transformations or rule extensions are known as *wide-sense evaluable*.

This concludes the definition of the syntactic based check for recognizing the domain independence of a formula based on the evaluable class of queries. Once it is known that an OCE is evaluable, there are a finite number of steps (described in Section 3.7 by the calculus to algebra reduction Theorem 3.3) that translates any evaluable OCE into an equivalent object algebra expression (OAE).

The second test for "safety" determines whether a query adds objects to the collections and classes that it is ranging over and to reject it if it does. This form of safety is called the check for *operand finiteness*. An example calculus expression that exhibits this problematic operation is as follows:

$$\{o \mid \exists p(\mathbf{C\_collection}(p) \wedge o = newcoll(p))\}$$

This query ranges over the entire class of collections (i.e., all collections) and for each collection $p$ it creates a new collection containing the collection $p$. The problem is that the new collections are created as instances of **C_collection**, thereby increasing the cardinality of **C_collection** for every object in **C_collection**. Since the semantics of the query is to range over all members of **C_collection**, the newly created collections should be included in the range of $p$. This results in the creation of more collections that should be included in the range and so on. The check for operand finiteness is deferred until after the generation of an equivalent algebraic expression and the check is performed on algebraic operators (see Section 3.5.3). This is done because an algebraic expression defines the procedural structure of a query and a recursive process is defined that goes through and tests each operator in turn.

## 3.5 The Object Algebra

An algebraic expression represents a typed collection of objects. The operands and result of algebraic operators are typed collections. Collections can be heterogeneous. When combining collections with certain algebra operators (e.g., product, union, intersection), a collection with a different type from those of the operand collections (or any type in the lattice) may be created. Thus, in order to integrate these new types into the existing lattice a type inferencing mechanism is introduced and used by the algebra.

There are two types to consider here: the type of the container (i.e., the type of the collection object) and the type of the objects in the container (i.e., the membership type of the collection). The types referred to in the inferencing mechanism are the membership types of collections.

### 3.5.1 Semantics of Type Inferencing

A query returns a collection as a result and every collection must have a single member type (Section 2.4.5). Thus, the algebraic operators may have to create a new type when forming a query result that contains objects of heterogeneous types or contains newly created objects. Therefore, type creation and type inferencing semantics are developed for the TIGUKAT model. Type creation and type inferencing are topics also related to schema evolution. Only the generic type creation and inferencing mechanisms are presented in this section. The complete discussion of schema evolution is presented in Chapter 5.

Let $T_i$ $(1 \leq i \leq n)$ denote types. Then, the behavioral application $T_i.B\_interface$ denotes the collection of behaviors applicable to objects of type $T_i$. The type inferencing mechanism is based on type construction operations that are modeled as behaviors on the primitive type `T_type`. They are defined as follows:

$T_1 \sqcap T_2$ (*B_tmeet*) produces the *meet* type of the argument types. The result type, say $T$, defines the behaviors that are common to types $T_1$ and $T_2$. The interface set of $T$ is defined as $T_1.B\_interface \cap T_2.B\_interface$. If $T_2$ is a subtype of $T_1$, then $T_1 \sqcap T_2$ is $T_1$. The converse is true if $T_1$ is a subtype of $T_2$. The *B_tmeet* behavior produces a result type that is integrated into the type lattice as a direct supertype of the argument types and a direct subtype of types forming the most specific set conformance of the argument types (i.e., all the common direct supertypes of the argument types before the integration is done).

$T_1 \sqcup T_2$ (*B_tjoin*) produces the *join* type of the argument types. The result type, say $T$, defines all the behaviors of $T_1$ together with all the behaviors of $T_2$. The interface set of $T$ is defined as $T_1.B\_interface \cup T_2.B\_interface$. If $T_2$ is a subtype of $T_1$, then $T_1 \sqcup T_2$ is $T_2$. The converse is true if $T_1$ is a subtype of $T_2$. The *B_tjoin* behavior produces a result type that is integrated as a direct subtype of the argument types and a direct supertype of all the common direct subtypes of the argument types before the integration is done.

$T_1 \otimes T_2$ (*B_tproduct*) produces the *product* type of the two argument types. The result type, say $T$, defines product behaviors (see below) and is integrated as a subtype of other product types according to the product behaviors defined. That is, the name and result type of product behaviors determines subtyping on product types. Objects of type $T$ are pairs with the first component being an object of type $T_1$ and the second component an object of type $T_2$. The *B_tproduct* behavior produces a product of types that does not have a sub/supertype relationship with the argument types, but is integrated with other product types. Instances of a product type are called *product objects*. They are created from objects in the extents of the types that contributed to the product type. The components of a product object are the original objects from which it was created.

The binary $\sqcap, \sqcup, \otimes$ behaviors can be naturally extended by defining them over multiple types in the following way (where $n \geq 2$):

$$\begin{aligned} \sqcap_{i=1}^{n} T_i &\equiv T_1 \sqcap T_2 \sqcap \cdots \sqcap T_n \\ \sqcup_{i=1}^{n} T_i &\equiv T_1 \sqcup T_2 \sqcup \cdots \sqcup T_n \\ \otimes_{i=1}^{n} T_i &\equiv T_1 \otimes T_2 \otimes \cdots \otimes T_n \end{aligned}$$

Parentheses may be used with the above operators. Each parenthesized subexpression represents the creation of a new type. With respect to the behaviors defined on the final type created, operators $\sqcap$ and $\sqcup$ are commutative and associative while $\otimes$ is neither. Parentheses affect the semantics of the product operator in the following way. Product types define inject behaviors ($\rho_i$) that return the $i^{th}$ component of a product object. With this in mind, the following product types are all different types that define different inject behaviors with different result types:

$$(T_1 \otimes T_2) \otimes T_3$$
$$T_1 \otimes (T_2 \otimes T_3)$$
$$T_1 \otimes T_2 \otimes T_3$$

The first type defines two inject behaviors; $\rho_1$ that returns a product object of type $T_1 \otimes T_2$ and $\rho_2$ that returns an object of type $T_3$. The second one defines two inject behaviors that differ from the first; $\rho_1$ that returns an object of type $T_1$ and $\rho_2$ that returns a product object of type $T_2 \otimes T_3$. The third type defines three inject behaviors; $\rho_1$ that returns an object of type $T_1$, $\rho_2$ that returns an object of type $T_2$ and $\rho_3$ that returns an object of type $T_3$.

The definition and integration of product types into the existing lattice and the creation of product objects is designed to be an automated process. A request is made through the application of a behavior to create a product object from a given list of objects. This may spawn the creation of a new product type and a class for the object if they don't already exist. In order to support these semantics, the following extensions are made to the primitive type system:

- T_product is defined as a subtype of T_type. T_product defines the following native behavior:

  $$B\_compTypes : \texttt{T\_list}\langle\texttt{T\_type}\rangle$$

  This behavior returns the list of component types that make up a product type. Intuitively, T_product is the type that describes the semantics of product types. The class **C_product** for this type is created as an instance of T_type-class so that the primitive type creation behavior (defined as *new* on this type) can be applied and passed a list of component types. The semantics of applying this creation behavior to **C_product** with a list of argument types is to create a product type (if one doesn't already exist) whose component types are the argument types passed, and to integrate the new type with existing product types. The behavior *B_tproduct* ($\otimes$) applies the type creation behavior to **C_product** passing along its arguments types. This defines the creation of new product types as instances of **C_product**.

- T_product-class is defined as a subtype of T_class. A product object creation behavior

  $$B\_new : \texttt{T\_list}\langle\texttt{T\_object}\rangle \rightarrow \texttt{T\_object}$$

  is defined on T_product-class. Intuitively, this type defines the semantics for the classes of product types. The class **C_product-type** is created as an instance of **C_class-class**. The type T_class-class defines a class creation behavior (*new*) that accepts a type (the type to associate a class to) as an argument. By applying this behavior to **C_product-class** and passing a product type, a class for the product type is created (if one does not already exist). Now, product objects can be created through the resulting class by applying the *B_new* behavior defined on T_product-class to the class and passing a list of objects.

For example, the following series of behavioral applications create a new product type called T_person-dwelling, a product class called **C_person-dwelling** and a product object *o* as an instance of this class. The first component of *o* is the person object joe and second component is the dwelling object apt204. The "$\leftarrow$" symbol denotes assignment and $<>$ denotes a list of objects.

$$\begin{aligned} \texttt{T\_person-dwelling} &\leftarrow \textbf{C\_product}.B\_new(<\texttt{T\_person}, \texttt{T\_dwelling}>) \\ \textbf{C\_person-dwelling} &\leftarrow \textbf{C\_product-class}.B\_new(\texttt{T\_person-dwelling}) \\ o &\leftarrow \textbf{C\_person-dwelling}.B\_new(<\texttt{joe}, \texttt{apt204}>) \end{aligned}$$

Finally, a behavior *B_newprod* is defined on T_object that accepts as arguments a list of objects and a list of corresponding behavioral projection sets. The result of applying this behavior with these arguments is as follows:

1. A product type is created (if one does not already exist) using the type of the receiver object and the types of the objects in the first argument list. The types are projected over the behavioral projections in the second argument list before the product type is formed.

2. A class for the product type is created (if one does not already exist).

3. A product object formed from the receiver and the objects in the first argument list is created as an instance of the (possibly new) product type and a reference to this object is returned.

For a given list of objects $o_1, o_2, \ldots, o_n$ and list of behavioral projection sets $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n$, the notation $newprod(o_1[\mathcal{B}_1], \ldots, o_n[\mathcal{B}_n])$ is used to denote a Bspec that represents the application of the product creating behavior with the given argument lists as:

$$o_1.B\_newprod(<o_2, \ldots, o_n>, <\mathcal{B}_1, \ldots, \mathcal{B}_n>)$$

The result is a product object $(o_1, \ldots, o_n)$ whose $i^{th}$ component is the original object $o_i$ from which it was formed. The type of each $o_i$ component object in the product type is the type of the original $o_i$ object projected over the behaviors in $\mathcal{B}_i$. When the behavioral projection list is immaterial, the notation is simplified to $newprod(o_1, \ldots, o_n)$.

In order to extract and operate on the original component objects of a product object, every product type defines an inject behavior for each of its component types. Product types are integrated into the type lattice according to the names and return types of these behaviors (more generally, their semantics). The behaviors defined on product types are the following:

**Inject:** For every product type $T_1 \otimes \cdots \otimes T_n$, there are $n$ inject behaviors defined $\rho_i$, $1 \leq i \leq n$ such that for a given object of this type, say $o$, the behavioral application $o.\rho_i$ returns the object of type $T_i$ that represents the $i^{th}$ component of $o$.

A product type $T_1 \otimes \cdots \otimes T_n$ is integrated as a subtype of a product type $T'_1 \otimes \cdots \otimes T'_m$ if $m \leq n$ and $T_i$ is a subtype of $T'_i$ for $1 \leq i \leq m$. It is integrated as a supertype of $T''_1 \otimes \cdots \otimes T''_k$ if $n \leq k$ and $T_i$ is a supertype of $T''_i$ for $1 \leq i \leq n$. If the product type cannot be integrated as a subtype of some other type, it is defined as a subtype of `T_object`.

**Equality:** The object equality behavior for `T_product` is refined to be based on pairwise identity equality of the component objects. That is, for two product objects $o$ and $o'$ of types $T_1 \otimes \cdots \otimes T_n$ and $T'_1 \otimes \cdots \otimes T'_n$, $o = o'$ is true if and only if $o.\rho_i = o'.\rho_i$ for $1 \leq i \leq n$.

### 3.5.2 Algebra Expressions

The underlying framework of the object algebra and calculus are essentially the same. However, an important difference is that the algebra can be viewed as having a functional basis as opposed to the logical foundation of the calculus. This perspective was described by Backus [Bac78] and has been exploited by several complex object models [MD86, Day89, AB93]. In the algebra, names are used as placeholders for collections with the appropriate types. The predicates $=, \neq, \in, \notin$ and connectives $\wedge, \vee, \neg$ are handled as boolean-valued functions. The object creating behaviors $newcoll(\ )$ and $newprod(\ )$ are variadic functions. There is a small set of well-defined algebraic operators (viewed as functions) that provide meaningful iterations over collections and can be composed to form more complicated queries (existential and universal quantification are handled by composing these operators). Thus, an algebraic query is a functional expression to be evaluated and the algebra is a functional language.

The basic algebra expression consists of a single collection specification. In the algebra, a base algebra expression is either a collection name or the function application $newcoll(c_1, \ldots, c_n)$ where each $c_i$ denotes a constant (i.e., a ground term). The latter is called a *collection constant*. Other algebra expressions can be constructed from the base expressions using the algebraic operators.

The basic constructs of the calculus (object constants, object variables, and Bspecs) have a functional interpretation that abstracts over the free variables in the constructs. The interpretation of these constructs is called a *functional expression.*

**Definition 3.5** *Functional Expression:* A *functional expression* is a functional abstraction of an object constant, an object variable or a Bspec defined as follows:

1. For every constant $c$, there is a unary functional expression $\lambda x.c$ that returns the constant $c$.

2. For every variable $x$, there is a unary functional expression $\lambda x.x$ that is the identity function.

3. For every Bpsec $\beta\{\vec{x}\}$, there is a functional expression $\lambda\vec{x}.\beta\{\vec{x}\}$ that represents a functional abstraction of the Bspec. If the Bspec is a ground term (i.e., is not free over any variables), then its functional expression is $\lambda x.\beta\{\}$ with the same semantics as for constants.

The variables appearing after the $\lambda$ symbol and before the first dot are called the *parameters* of the functional expression.

Since Bspecs can be abstracted into functional expressions, all behaviors have this abstraction. This means that predicates $=, \neq, \in, \notin$ and connectives $\wedge, \vee, \neg$ are boolean-valued functional expressions. The object creating behaviors *newcoll*( ) and *newprod*( ) are variadic functional expressions that produce the appropriate collection or product object. The algebraic operators (defined below) are functional expressions that operate on collections and produce collections as results.

In general, *mop* is used to denote a functional expression and is called a *mop function.* Given a mop function (*mop*) with parameters $\vec{x}$ and given objects $\vec{o}$ that are type compatible with $\vec{x}$, $mop(\vec{o})$ is used to denote the application of the mop function to the objects. That is, each $o_i$ is substituted for an $x_i$ to form a context, the context is evaluated and the result object is produced.

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

Certain algebraic operators require a functional expression (mop function) as an argument. The operator applies the mop function to permutations of elements from its operand collections and takes appropriate action on the result. Some operators require a boolean-valued functional expression (a predicate) denoted $F$. Evaluating $F$ for particular permutation of arguments produces a boolean result upon which the operator takes an appropriate action. The membership types of the operand collections must be consistent with the types expected by the mop function. Mop function qualified operators are written as $P\ \Phi_{mop}\ \langle Q_1, \ldots, Q_n\rangle$ where *mop* is a mop function (or predicate) with parameters,

say $p, q_1, \ldots, q_n$, that range over the elements of collections $P, Q_1, \ldots, Q_n$, respectively. To make the identification of arguments with parameters simpler and more explicit in algebraic operators, the $\lambda\vec{x}$ specification is dropped from mop functions and replaced by subscripting operand collections with the parameters of the mop function as $P_p$. This explicitly indicates that the range of variable $p$ (in the mop function) are the elements of the operand collection $P$. For example, $P_p\ \Phi_{mop(p,q)}\ Q_q$ is used instead of the abstract notation $P\ \Phi_{\lambda p,q.mop(p,q)}\ Q$. For operands consisting of product objects with components $\vec{x}$, the operands are subscripted with all the components as $P_{\vec{x}}$. This means that some combination of inject behaviors on the elements of $P$ will retrieve the original $x_i$ components. This is only a notational convenience to identify the ranges of variables and the components of product objects in algebra expressions.

For a collection $P$, the notation $\Lambda_P$ denotes the membership type of the objects in $P$. Furthermore, the behavioral application $\Lambda_P.B\_interface$ denotes the behaviors applicable to objects of this type. This notation and the results of Section 3.5.1 are used to infer a new membership type for the result collection produced by the operators.

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

**Difference** (denoted $P - Q$): Difference is a binary operator that produces a collection containing objects that are in $P$ and not in $Q$. The membership type of the result collection is exactly the type of $P$ (i.e. $\Lambda_P$).

**Union** (denoted $P \cup Q$): Union is a binary operator that produces a collection containing objects that are in $P$, in $Q$ or in both. The membership type of the result collection is $\Lambda_P \sqcap \Lambda_Q$. This type defines behaviors common to both $\Lambda_P$ and $\Lambda_Q$.

**Intersection** (denoted $P \cap Q$): Intersection is a binary operator that produces a collection containing objects that are both in $P$ and in $Q$. The membership type of the result collection is $\Lambda_P \sqcup \Lambda_Q$. This type defines all behaviors of both $\Lambda_P$ and $\Lambda_Q$. Note that $P \cap Q$ is derivable from difference as $P - (P - Q)$ or $Q - (Q - P)$. Even though these three operations produce result collections with identical extents, the membership type of each result may differ. The intersection operator is preferred over difference because it has the potential to produce more type information.

**Collapse** (denoted $P \Downarrow$): Collapse is a unary operator accepting a collection of collections $P$ as an argument and produces the extended union of the collections in $P$.

$$P \Downarrow \equiv \bigcup\{x \mid x \in P\}$$

The membership type of the result collection is the extended meet over the membership types of the collections in $P$.

$$\sqcap\{\Lambda_x \mid x \in P\}$$

**Select** (denoted $P\ \sigma_F\ \langle Q_1, \ldots, Q_n\rangle$): where $F$ is a predicate over the elements of collections $P, Q_1, \ldots, Q_n$, meaning $F$ expects arguments $p, q_1, \ldots, q_n$ and that they are type consistent with the membership types of the collections. Select is a higher order operation accepting a mop function, the predicate $F$, and the $n+1$ collections $P, Q_1, \ldots, Q_n$ as arguments. The select operation produces a collection containing objects from $P$ corresponding to the $p$ component of each permutation $<p, q_1, \ldots, q_n>$ that satisfies

$F(p, q_1, \ldots, q_n)$. The membership type of the result collection is exactly the type of $P$ (i.e. $\Lambda_P$).

**Example 3.1** Return the persons that are senior citizens:

$$\textbf{C\_person}_p \ \sigma_{p.B\_age \geq 65}$$

**Example 3.2** Return the maps that contain water zones:

$$\textbf{C\_map}_p \ \sigma_{q \in p.B\_zones} \ \textbf{C\_water}_q$$

**Project** (denoted $P\ \Pi_{\mathcal{B}}$): where $\mathcal{B}$ is a behavioral projection set with the restriction that it be a subset of the behaviors defined by the membership type of $P$. (i.e., a subset of $\Lambda_P.B\_interface$). The $\mathcal{B}$ collection is automatically unioned with the behaviors of type `T_object` before the project is performed in order to ensure consistency with the object model (i.e., everything is an object and therefore must support the behaviors of `T_object`). Project produces a collection containing the objects of $P$, but with a membership type coinciding with the behaviors in $\mathcal{B}$.

The new type is integrated into the sublattice rooted at `T_object` and with the base $\Lambda_P$. An abstract type definition is created that has all the behaviors defined by $\mathcal{B}$. The implementations of these behaviors are undefined, but this doesn't cause problems because no class is created and therefore no objects of this type exist. This new type has no special properties, meaning it can be subtyped, implementations for its behaviors can be defined, a class can be associated with it and objects of this type can be created.

The $\mathcal{B}$ projection set has no impact on which objects appear in the result collection of the query. It is only important during the final type assignment that occurs at type inferencing time after the extent of the query has been produced. This form of project differs from the traditional one in that it does not project over the structure of objects, but rather over their behavioral specification. The project operator is a behavioral-theoretic notion of projection that has no structural implications.

**Example 3.3** Project over behaviors *B_name* and *B_age* for class **C_person**:

$$\textbf{C\_person}\ \Pi_{B\_name, B\_age}$$

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

**Product** (denoted $Q_1 \times \cdots \times Q_n$): where $n \geq 2$. Product produces a collection containing product objects of the form $(q_1, q_2, \ldots, q_n)$ created from each permutation $<q_1, q_2, \ldots, q_n>$ such that component $q_i$ is an object from $Q_i$. Product may initiate the creation of a new type along with a new class to maintain the product objects. The membership type of the result collection is $\Lambda_{Q_1} \otimes \cdots \otimes \Lambda_{Q_n}$. Although this operator seems structural in nature, Section 3.5.1 defines a behavioral-theoretic notion of product that is commensurate with the uniformity of the object model.

There is an additional operator that fits into both the target-preserving and target-creating classification. The **map** operator produces a collection of new or existing objects depending on the mop function argument passed to it. That is, if the mop function is target-creating, the operator is target-creating, otherwise it is target-preserving. Map is defined as follows:

**Map** (denoted $Q_1 \gg_{mop} \langle Q_2, \ldots, Q_n \rangle$): where *mop* is a mop function over the elements of collections $Q_1, Q_2, \ldots, Q_n$, meaning it expects arguments $q_1, q_2, \ldots, q_n$ and that they are type consistent with the membership types of the collections. Map is a higher order operation accepting the mop function *mop* and the $n$ collections $Q_1, Q_2, \ldots, Q_n$ as arguments. For each permutation of objects $< q_1, q_2, \ldots, q_n >$ formed from the elements of the argument collections, $mop(q_1, q_2, \ldots, q_n)$ is applied and the resulting object is included in the result collection. The membership type of the result collection is the type of the mop function. Map is a generalized version of the same operator defined in [SÖ90a] and is similar to the *replace* restructuring operator in [AB93]. However, *replace* operates over a single set-valued relation in contrast to *map*, which is variadic over the number of argument collections. Map is also similar to the **image** operator of [SZ90] except that theirs is restricted to the application of single behaviors while the *mop* in a *map* operator is a general functional expression.

**Example 3.4** Return the zones that have people living in them:

$$\mathbf{C\_person}_p \gg_{p.B\_residence.B\_inZone}$$

**Example 3.5** Return the proximities of water zones to the City of Edmonton:

$$\mathbf{C\_water}_p \gg_{p.B\_proximity(\mathsf{edmonton})}$$

**Example 3.6** Return (person, person, children) triples for all combinations of people:

$$\mathbf{C\_person}_p \gg_{newprod(p,q,p.B\_children(q))} \mathbf{C\_person}_q$$

The operators defined above form the *primitive* algebra (some refer to this as a *physical* algebra). They are fundamental in supporting the expressive power of the calculus and the subsequent operators can be defined in terms of them. The following operators are added to the primitive algebra and this is called the *extended* algebra (some call this a *logical* algebra). These operators are derived from the primitive algebra, they support a useful functionality, they generalize the expressive power of the algebra and some are important for higher-level optimizations [SÖ90a]. Note that the following operators are target-creating.

**Join** (denoted $P \bowtie_F \langle Q_1, \ldots, Q_n \rangle$): where $n \geq 1$ and $F$ is a predicate over the elements of collections $P, Q_1, \ldots, Q_n$. Join produces a collection containing product objects of the form $(p, q_1, \ldots, q_n)$ created from each permutation $< p, q_1, \ldots, q_n >$ that satisfies $F(p, q_1, \ldots, q_n)$. The membership type of the result collection is $\Lambda_P \otimes \Lambda_{Q_1} \otimes \cdots \otimes \Lambda_{Q_n}$. This type and its associated class may be created if they don't already exist.

The join operator can be expressed in terms of product and selection as follows:

$$E_{x_1} \bowtie_F \langle E_{x_2}, \ldots, E_{x_n} \rangle \quad \equiv \quad (E_{x_1} \times E_{x_2} \times \cdots \times E_{x_n})_o \; \sigma_{F'}$$

where $F$ is a predicate over variables $\vec{x}$ and $F'$ is $F$ except that every occurrence of $x_i$ is replaced with $o.\rho_i$, the inject of component $x_i$ from product object $o$.

**Example 3.7** Return married couples that don't live together:

$$\mathbf{C\_person}_p \bowtie_{p.B\_spouse=q \ \wedge \ q.B\_residence \neq p.B\_residence} \mathbf{C\_person}_q$$

**Example 3.8** Return (map, water zone, water zone) triples where the given map contains two different water zone that are within 100 units from each other:

$$\mathbf{C\_map}_m \bowtie_{x \in m.B\_zones \wedge y \in m.B\_zones \wedge x \neq y \wedge x.B\_proximity(y) \leq 100} \langle \mathbf{C\_water}_x, \mathbf{C\_water}_y \rangle$$

**Generate Join** (denoted $Q_1 \ \gamma_g^o \ \langle Q_2, \ldots, Q_n \rangle$): $g$ is a generating atom of the form $o \ \theta \ mop$ where $\theta$ is either $=$ or $\in$ and $mop$ is a mop function over the elements of collections $Q_1, Q_2, \ldots, Q_n$. Generate join produces a collection of product objects created from each permutation of the $q_i$'s and extended by an object $o$ in the following way. If $\theta$ is $=$, the result contains product objects of the form $(q_1, q_2, \ldots, q_n, mop(q_1, q_2, \ldots, q_n))$ for each permutation of the $q_i$'s (i.e., each product object is a permutation of the $q_i$'s extended by the result of applying the mop function to that permutation). If $\theta$ is $\in$, the result contains product objects of the form $(q_1, q_2, \ldots, q_n, o)$ for each permutation of the $q_i$'s and each $o \in mop(q_1, q_2, \ldots, q_n)$ (i.e., for a permutation of the $q_i$'s and for each member $o$ of the collection resulting from the application $mop(q_1, q_2, \ldots, q_n)$, a product object with components $(q_1, q_2, \ldots, q_n, o)$ is created as a member of the result collection). Generate Join is similar to PDM's *apply-append* operator except theirs works on a single tuple while generate join is over an arbitrary number of collections.

The equality atom based generate join can be expressed by map as follows:

$$E_{x_1} \ \gamma^o_{o=mop} \ \langle E_{x_2}, \ldots, E_{x_n} \rangle \quad \equiv \quad E_{x_1} \gg_{newprod(x_1, x_2, \ldots, x_n, mop(\vec{x}))} \ \langle E_{x_2}, \ldots, E_{x_n} \rangle$$

The membership atom based generate join can be expressed by the following series of algebraic operations:

$$\begin{aligned} A &\stackrel{\text{def}}{=} E_{x_1} \times E_{x_2} \times \cdots \times E_{x_n} \\ B &\stackrel{\text{def}}{=} A_x \gg_{newprod(x, mop(x.\rho_1, x.\rho_2, \ldots, x.\rho_n))} \\ C &\stackrel{\text{def}}{=} (B_x \gg_{newcoll(x.\rho_1) \times x.\rho_2} ) \Downarrow \\ E_{x_1} \ \gamma^o_{o \in mop} \ \langle E_{x_2}, \ldots, E_{x_n} \rangle &\equiv C_x \gg_{newprod(x.\rho_1.\rho_1, x.\rho_1.\rho_2, \ldots, x.\rho_1.\rho_n, x.\rho_2)} \end{aligned}$$

**Example 3.9** Return (zone, proximity) pairs of each zone extended with its proximity to all water zones:

$$\mathbf{C\_zone}_p \ \gamma^o_{o=p.B\_proximity(q)} \ \mathbf{C\_water}_q$$

**Example 3.10** Return (map, zone) pairs of each map extended with the zones contained in that map:

$$\mathbf{C\_map}_p \ \gamma^o_{o \in p.B\_zones}$$

**Reduce** (denoted $P \triangle_{\rho_i}$): where $P$ is a collection of product objects and $\rho_i$ is an inject behavior defined on the membership type of $P$. The reduce operator has the effect of

discarding the $i^{th}$ component of the product objects in $P$. That is, product objects of the form:

$$(p_1, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_n)$$

with inject behaviors:

$$\rho_1, \ldots, \rho_{i-1}, \rho_i, \rho_{i+1}, \ldots, \rho_n$$

are mapped to product objects of the form:

$$(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n)$$

with inject behaviors:

$$\rho_1, \ldots, \rho_i, \rho_{i+1}, \ldots, \rho_{n-1}$$

This is similar to the relational projection operator except that the specified components are removed. If $P$ is not a product object, the empty collection is returned.

The reduce operator can be expressed by map as follows:

$$E \triangle_{\rho_i} \quad \equiv \quad E_o \gg_{newprod(o.\rho_1, \ldots, o.\rho_{i-1}, o.\rho_{i+1}, \ldots, o.\rho_n)}$$

The effect of the map is to produce product objects that contain all the original components of $o$, minus the $i^{th}$ component. Map, together with the product object creation behavior, is a generalization of the relational projection on product objects.

As a notational convenience, a series of reduce operators is coalesced into a single one and the $\rho$ symbol is dropped from the specification. The equivalence is defined as follows:

$$P \triangle_{\rho_{x_1}} \ldots \triangle_{\rho_{x_n}} \equiv P \triangle_{x_1, \ldots, x_n}$$

**Example 3.11** Let $E$ be the result of Example 3.8 above. Reduce $E$ by excluding the first water zone of the result:

$$E \triangle_x$$

The functional nature of queries is twofold. On the one hand, a query may be thought of as a function where collection names serve as variables representing the arguments. By associating these names with collections in an instantiation of an objectbase, a substitution is formed and can be evaluated. On the other hand, for a given (static) objectbase, a query denotes a constant because it will produce the same answer over and over. Thus, a query is a function only when all possible objectbases are considered. For a given objectbase (i.e., interpretation), a query is an expression resembling a 0-ary function. In contrast, behavioral compositions such as Bspecs (mops) are functions even within the instantiation of a objectbase. When they are composed with algebraic operators *select, map, join* and *generate join*, they denote functions that are applied to permutations of the elements from the operand collections.

The *powerset* operator has not been included in the TIGUKAT algebra because one of the primary concerns of the TIGUKAT project is to produce an efficient implementation of the query model. Use of powerset causes exponential growth of collections and the costs that this could incur is unacceptable for the implementation of the model.

The foundations of powerset and recursive query capability are present in the TIGUKAT query model, and since the model is extensible, they can be added by type and behavior

extensions. One extension is the addition of a primitive *powerset* algebraic operator (i.e., behavior) that accepts a collection and produces the powerset of the collection as output. Using this, a form of generate join could be derived that creates a collection of product objects – one for each element in the powerset of the *mop* function evaluation – whose components are the operand collections appended with the element from the powerset. Since a *B_containedBy* behavior (analogous to $\subseteq$) already exists on `T_collection`, only a predicate $s \subseteq t$ needs to be added in the calculus for this behavior. If the term $s$ is a variable, then this becomes another kind of generating atom in the calculus.

A clean definition of safety with respect to powerset that complies with the efficient translation of evaluable formulas (i.e., without forming large *DOM* sets) is not apparent. The powerset property has a logical derivation as follows:

$$\begin{aligned} s \subseteq t &\equiv \forall x(x \in s \Longrightarrow x \in t) \\ &\equiv \forall x(x \notin s \vee x \in t) \\ &\equiv \neg\exists x(x \in s \wedge x \notin t) \end{aligned}$$

This derivation does not satisfy the evaluable property unless $s$ and $t$ are further restricted outside the formula. This means that $s \subseteq t$ can not in general be used to generate objects for $s$ from $t$ and its only consistent use would be as a restriction atom. However, this is already handled in TIGUKAT because the derivation is a valid formula of the calculus and is safe if $s$ and $t$ are restricted outside the formula. Thus, without being able to generate values for $s$ from the derivation, no additional power is added by including a $\subseteq$ predicate and a powerset operator. On the contrary, it would make the algebra more expressive than the calculus, since the translation of the powerset operator to the calculus (i.e., the derivation above) would result in an unsafe calculus formula.

A clean incorporation of powerset capability that complies with the feasible translation properties of the evaluable class is part of the future research of the TIGUKAT project. If a compatible derivation can be found, extending the proofs of completeness will be straightforward. From algebra to calculus it is simply a matter of stating the derivation of the powerset operator and from calculus to algebra it involves carrying the $\subseteq$ predicate through the translation.

### 3.5.3 Safety of Algebra Expressions

Recall from the discussion in Section 3.4.4 that there are two forms of safety to consider. The first form checks the domain independence of the query and was defined in that section. The second form checks the safety of a query with respect to *operand finiteness*, meaning it checks that the query does not add objects to any collections or classes that it is ranging over. This check is defined on algebraic expressions and determines the operand finiteness of each operator in the expression.

Since object creation and insertion occurs through the application of behaviors, the check for operand finiteness could be combined with an algebraic type checking mechanism such as the one defined in [SÖ90b] that goes through an algebraic expression and examines the behaviors being applied in algebraic operators for type consistency.

The "problematic" operators of the algebra that can violate operand finiteness by adding objects to their operands are *select*, *map*, *join* and *generate join* because they contain mop functions that are general behavioral applications. The only side effect behaviors allowed

in mop functions are insertion into a collection (i.e., *B_insert* on a collection) and creation of a new object (i.e., *B_new* on a class). This is further restricted in that the insertion or creation behavior must be applied to a constant reference of a collection or a class (i.e., not to a variable or the result of a behavioral application) or must not occur at all.

All other behaviors in a mop function are assumed to be side-effect free (i.e., they do not create new objects or modify existing objects in any way). The reason for this assumption is that the implementations of behaviors are not examined to determine their safety with respect to operand finiteness. The exceptions to this assumption are the primitive defined *newcoll*() and *newprod*() behaviors and the algebraic operators. They can occur in mop functions, but their use is restricted as defined below.

An algebraic expression is rejected if it contains an algebraic operator that is *unsafe* with respect to operand finiteness. An algebraic operator $\Phi$ is *unsafe* with respect to operand finiteness if it is a *select*, *map*, *join* or *generate join* operator which has a mop function that contains one of the following:

- an application of *B_new* on a class that is an operand of $\Phi$,
- an application of *B_new* on a class that is a subclass of an operand of $\Phi$ and the operand is a class ranging over its deep extent,
- an application of *B_insert* on a collection that is an operand of $\Phi$,
- an application of *newcoll*() and one of the operands of $\Phi$ is the class **C_collection**,
- an application of *newprod*() that creates an object in a class that is an operand of $\Phi$,
- an application of *newprod*() that creates an object in a subclass of an operand of $\Phi$ and the operand is a class ranging over its deep extent,
- an algebraic operator and one of the operands of $\Phi$ is **C_collection**,
- an algebraic operator and this algebraic operator is *unsafe* with respect to operand finiteness.

## 3.6 Example Queries

An SQL-like language called TQL (TIGUKAT Query Language) [PLÖS93b, Lip93] has been developed for the model. The *select-from-where* clause of the language is an object-oriented extension of SQL. The basic structure of this clause is used to present some queries that illustrate the properties of the calculus and algebra. The queries are first expressed in TQL, followed by the corresponding object calculus expression and then the equivalent algebraic expression. In the algebraic expressions, operand collections are subscripted by the variables that ranges over them. If the operand consists of product objects, the variables that make up the components of these objects are listed. The indexed variables are used as a symbolic reference to the elements of the collection as described in Section 3.5.2. Furthermore, the arithmetic notation for operations like *o.greaterthan(p)*, *o.elementof(p)*, etc., is used instead of their boolean Bspec equivalents. The execution of the algebraic expression is from left-to-right, except that parenthesized expressions have higher priority and are executed first.

**Example 3.12** Return land zones valued over $100,000 or that cover an area over 1000 units.

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

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

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

**Example 3.13** Return all zones that have people living in them (the zones are generated from person objects).

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

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

Algebra: $\left(\mathbf{C\_person}_q\ \gamma^o_{o=q.B\_residence.B\_inzone}\ \right)_{q,o} \triangle_q$

**Example 3.14** Return the maps with areas where senior citizens live.

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

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

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

**Example 3.15** Return all maps that describe areas strictly above 5000 feet.

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

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

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

**Example 3.16** Return the dollar values of the zones that people live in.

TQL: **select** $p.B\_residence().B\_inzone().B\_value()$
**from** $p$ **in C_person**

Calculus: $\{\ o \mid \exists p(\mathbf{C\_person}(p) \wedge o = p.B\_residence.B\_inzone.B\_value)\}.$

Algebra: $\left(\mathbf{C\_person}_p\ \gamma^o_{o=p.B\_residence.B\_inzone.B\_value}\ \right)_{p,o} \triangle_p$
Note that this has a simplier form using the map operator as follows:
$\mathbf{C\_person}_p \gg_{p.B\_residence.B\_inzone.B\_value}$

**Example 3.17** Return the zones that are part of some map and are within 10 units from water. Project the result over *B_title* and *B_area*.

TQL: **select** $o[B\_title, B\_area]$
**from** $p$ **in C_map**, $o$ **in** $p.B\_zones$, $q$ **in C_water**
**where** $o.B\_proximity(q) < 10$

Calculus: $\{ o[B\_title, B\_area] \mid \exists p \exists q(\mathbf{C\_map}(p) \wedge \mathbf{C\_water}(q) \wedge o \in p.B\_zones \wedge o.B\_proximity(q) < 10)\}$.

Algebra: $\left(\left(\mathbf{C}\text{\_map}_p\ \gamma^{o}_{o \in p.B\_\text{zones}}\right)_{p,o} \bowtie_{o.B\_\text{proximity}(q)<10} \mathbf{C}\text{\_water}_q\right)_{p,o,q} \triangle_{q,p}\ \Pi_{B\_\text{title},B\_\text{name}}$

**Example 3.18** Return pairs consisting of a person and the title of a map such that the person's dwelling is in the map.

TQL: **select** $p,\ q.B\_title()$
**from** $p$ **in C_person**, $q$ **in C_map**
**where** $p.B\_residence().B\_inZone() \in q.B\_zones()$

Calculus: $\{p, o \mid \exists q(\mathbf{C\_person}(p) \wedge \mathbf{C\_map}(q) \wedge o = q.B\_title \wedge p.B\_residence.B\_inZone \in q.B\_zones)\}$

Algebra: $\left(\mathbf{C}\text{\_person}_p \bowtie_{p.B\_\text{residence}.B\_\text{inZone} \in q.B\_\text{zones}} \left(\mathbf{C}\text{\_map}_q\ \gamma^{o}_{o=q.B\_\text{title}}\right)_{q,o}\right)_{p,q,o} \triangle_q$

**Example 3.19** Return (person, spouse, child) triples of all couples and their children where the first parent is homeless. The children set of a couple is "flattened" by grouping each child with their parents.

TQL: **select** $p,\ s,\ c$
**from** $p, s$ **in C_person**, $c$ **in** $p.B\_children(s)$
**where** $s = p.B\_spouse()$ **and**
**not** $p.B\_residence()$ **in** ( **select** $h$
**from** $h$ **in C_house**)

Calculus: $\{p, s, c \mid \mathbf{C\_person}(p) \wedge \mathbf{C\_person}(s) \wedge c \in p.B\_children(s) \wedge s = p.B\_spouse \wedge p.B\_residence \notin \mathbf{C\_house}\}$

Algebra: $\left(\left(\mathbf{C}\text{\_person}_p\ \sigma_{p.B\_\text{residence} \notin \mathbf{C}\text{\_house}}\right)_p \bowtie_{s=p.B\_\text{spouse}} \mathbf{C}\text{\_person}_s\right)_{p,s}\ \gamma^{c}_{c \in p.B\_\text{children}(s)}$

## 3.7 Completeness of Calculus and Algebra

A desired property of the languages of a query model is that they be equivalent in expressive power. That is, any expression formed in one language has an equivalent formation in the other. In the calculus it was shown that certain queries are not "reasonable" because there is no efficient way to process them. Thus, in defining the completeness of the languages, only the "reasonable" or "safe" expressions are considered.

In this chapter, the completeness of the reduction from the algebra to the calculus and from the calculus to the algebra is shown. This is sufficient to prove the equivalence of the formal languages. A reduction of the TIGUKAT Query Language (TQL) to the formal calculus has been reported elsewhere [PLÖS93b, Lip93].

### 3.7.1 Theorems and Proofs

**Theorem 3.2** The reduction from the object algebra to the object calculus is complete.
**Proof:** It must be shown that if $E$ is an expression in the object algebra, then there is an object calculus expression (OCE) equivalent to $E$. The proof is by structural induction on the number of operators in $E$.

**Basis. Zero Operators:** Then $E$ consists of a single collection name $C$ or a collection creating behavior application $newcoll(c_1, \ldots, c_n)$ where each $c_i$ is a constant. An equivalent OCE for $E$ in the first case is $\{o \mid C'(o)\}$ where $C'$ is the predicate for collection $C$. In the second case an equivalent OCE for $E$ is $\{o \mid o \in newcoll(c_1, \ldots, c_n)\}$.

**Induction:** Assume $E$ has at least one operator and that the theorem is true for expressions with fewer operators than $E$.

**Case 1:** $E \stackrel{\text{def}}{=} E_1\ \Pi_{\mathcal{B}}$. Since $E_1$ is an object algebra expression with fewer operators than $E$, an OCE $\{o \mid \psi_1(o)\}$ equivalent to $E_1$ can be found. Then $E$ is equivalent to $\{o[\mathcal{B}] \mid \psi_1(o)\}$.

**Case 2:** $E \stackrel{\text{def}}{=} E_1 - E_2$. By renaming of variables if necessary, OCEs $\{o[\mathcal{B}_1] \mid \psi_1(o)\}$ and $\{o[\mathcal{B}_2] \mid \psi_2(o)\}$ equivalent to $E_1$ and $E_2$ can be found (the behavioral projections $\mathcal{B}_1$ and $\mathcal{B}_2$ may be empty). Then $E$ is equivalent to $\{o[\mathcal{B}_1] \mid \psi_1(o) \wedge \neg\psi_2(o)\}$.

**Case 3:** $E \stackrel{\text{def}}{=} E_1 \cup E_2$. OCEs for $E_1$ and $E_2$ can be found as in Case 2. Then $E$ is equivalent to $\{o[\mathcal{B}_1 \cap \mathcal{B}_2] \mid \psi_1(o) \vee \psi_2(o)\}$. Note that $\mathcal{B}_1 \cap \mathcal{B}_2$ denotes the intersection of the two component behavioral projections. This intersection represents the proper behavioral projection of the result collection.

**Case 4:** $E \stackrel{\text{def}}{=} E_1 \cap E_2$. $E_1$ and $E_2$ have equivalent OCEs as in Case 2. Then $E$ is equivalent to $\{o[\mathcal{B}_1 \cup \mathcal{B}_2] \mid \psi_1(o) \wedge \psi_2(o)\}$. Here $\mathcal{B}_1 \cup \mathcal{B}_2$ denotes the union of the two component behavioral projections.

**Case 5:** $E \stackrel{\text{def}}{=} E_1 \Downarrow$. There is an equivalent OCE for $E_1$ as in Case 2. Then $E$ is equivalent to $\{o \mid \exists o_1(\psi_1(o_1) \wedge o \in o_1)\}$.

**Case 6:** $E \stackrel{\text{def}}{=} E_1\ \sigma_F\ \langle E_2, \ldots, E_n\rangle$. There are $n$ OCEs equivalent to $E_1, E_2, \ldots, E_n$. Then $E$ is equivalent to $\{o[\mathcal{B}_1] \mid \psi_1(o) \wedge \exists o_2 \ldots \exists o_n(\psi_2(o_2) \wedge \ldots \wedge \psi_n(o_n) \wedge F(o, o_2, \ldots, o_n))\}$.

**Case 7:** $E \stackrel{\text{def}}{=} E_1 \times \cdots \times E_n$. There are $n$ OCEs equivalent to $E_1, \ldots, E_n$. Then $E$ is equivalent to $\{o \mid \exists o_1 \ldots \exists o_n(\psi_1(o_1) \wedge \cdots \wedge \psi_n(o_n) \wedge o = newprod(o_1[\mathcal{B}_1], \ldots, o_n[\mathcal{B}_n]))\}$. Here $newprod(o_1[\mathcal{B}_1], \ldots, o_n[\mathcal{B}_n])$ denotes the behavioral application that creates a product object constant whose $i^{th}$ component is the object denoted by $o_i$ that is typed according to the behavioral projection set $\mathcal{B}_i$.

**Case 8:** $E \stackrel{\text{def}}{=} E_1 \gg_{mop} \langle E_2, \ldots, E_n\rangle$. There are $n$ OCEs equivalent to $E_1, E_2, \ldots, E_n$. Then $E$ is equivalent to $\{o \mid \exists o_1 \exists o_2 \ldots \exists o_n(\psi_1(o_1) \wedge \psi_2(o_2) \wedge \ldots \wedge \psi_n(o_n) \wedge o = mop(o_1, o_2, \ldots, o_n))\}$.

The other algebraic operators can be written in terms of the primitive ones above and this completes the proof. □

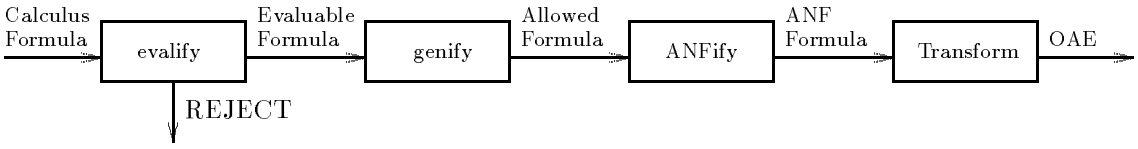



Figure 3.4: Translation steps from object calculus to object algebra.

**Theorem 3.3** The reduction from the the object calculus to the object algebra is complete.
**Proof:** The reduction from the calculus to the algebra is proven by a translation algorithm that follows the steps illustrated in Figure 3.4. The first step, called *evalify*, determines the *evaluability* (Definition 3.4) of a given object calculus formula. Recall from Section 3.4.4 that evaluability is enough for safety; this is proved by the translation algorithm in Section 3.7.2. Moreover, the class of evaluable queries being translated are *wide-sense evaluable* with respect to equality and membership, meaning a broader class of safe queries are recognized by the approach. If the input formula is not evaluable, it is rejected.

From a database point of view, only those queries considered to be safe are candidates for translation to algebra. For evaluable formulas, the rest of the translation is similar to that presented in [GT91], except that the extended definitions of the approach in this thesis are carried through.

The *genify* step converts an evaluable formula into an *allowed* form (Definition 3.6) that rewrites the formula to include range "generators" for variables in each subformula. The *ANFify* step places an allowed formula into *Allowed Normal Form* (ANF) (Definition 3.14) that makes each constructive subformula independent of atoms outside the quantifier for the subformula. The *ANFify* step makes use of *Existential Normal Form* (ENF) (Definition 3.12) and *simplified* form (Definition 3.9). The advantage of ANF is that the transformation from this form to the algebra is straightforward. The final step of the translation involves simple pattern matching to *transform* the ANF formula into a (safe) *object algebra expression* (OAE) that is equivalent to the original formula. The complete translation algorithm is presented in Section 3.7.2. □

### 3.7.2 Calculus to Algebra Translation

In this section, the complete translation algorithm for converting safe object calculus expressions into equivalent algebraic expressions is presented. The algebra expressions should be checked for type consistency before they are optimized and prior to an execution plan being generated. Since every object knows its type, this step may be performed during compilation of the query. Query optimization and execution plan generation are reported elsewhere [Muñ94].

To help understand the translation process, the following query is given as a running example. Throughout this section, the calculus expression in Example 3.20 is translated into an equivalent algebra expression with the intermediate steps shown along the way.

**Example 3.20** Return zones that are transport zones or that have people living in them.
Consider the query expressed in the following way:

$$\{\ o\ \mid\ \exists p((\mathbf{C\_person}(p)\ \wedge\ o = p.B\_residence.B\_inZone)\ \vee\ \mathbf{C\_transport}(o))\ \}$$

For brevity, predicate **C_person** is mapped to $P$, **C_transport** to $T$, and the behavior application $p.B\_residence.B\_inZone$ to $p.\alpha$. The query can then be written as:

$$\{\ o\ \mid\ \exists p((P(p)\ \wedge\ o = p.\alpha)\ \vee\ T(o))\ \}$$

Let the formula part of the query be $F \stackrel{\text{def}}{=} \exists p((P(p)\ \wedge\ o = p.\alpha)\ \vee\ T(o))$ □

First the *gen* and *con* rules of Figure 3.3 are extended by adding the notion of "generators" as described in [GT91]. The extended rules are shown in Figure 3.5. The technique adds a third argument $G(x)$ that serves as a "generator" of sorts for the variable $x$. A $G(x)$ "generator" is a disjunction of *edb* and *gdb* atoms (possibly including a placeholder $\perp$) that generates all the needed objects for $x$ in the given formula and possibly more (i.e., $G(x)$ is a range for $x$ that is at least as large as the values that $x$ can take on in the formula). Moreover, the atoms in $G(x)$ were the ones used to prove that the *gen* or *con* relation holds for variable $x$ in some formula $A(x)$. The placeholder "$\perp$" is used when $x$ is not free in the formula $A$: it may be thought of as a 0-ary predicate that always fails.

### Evalify: Syntactic Safety Check

The *evalify* algorithm (Algorithm 3.1) syntactically determines whether a given input formula $F$ is evaluable or not and returns an indicator SAFE or REJECT, respectively. Recall from the discussion in Section 3.4.4 that the *evaluable property* (Definition 3.4) is sufficient for safety. A side-effect of the algorithm is that the partial order $<_F$ for formula $F$ is defined. When *evalify* is first called, the partial order is initialized as undefined. The algorithm incrementally builds the partial order on each pass through the repeat loop; the first pass orders variables that are generated from *edb* atoms, the second pass orders variables that are generated from variables in the first pass and so on. The *gdb* predicate for the *gen* and *con* rules uses the "partially defined" partial order in each intermediate pass through the repeat loop. Thus, the results of the previous pass are used to update the partial order on the current pass. The temporary set $V$ is used to temporarily store undefined elements of the partial order that are updated after the *gen* and *con* application. This is done to avoid misorderings since the partial order is incrementally built and always used by the *gdb* predicate. If all variables in $<_F$ become ordered, the input formula is evaluable and therefore SAFE. A fixpoint of the algorithm is reached when no changes are made to the partial order. At this point the formula is REJECTed since there are variables in $<_F$ that cannot be ordered, meaning they have no "reasonable" range defined and they cannot be generated from the other variables.

The result of applying *evalify* to the formula $F$ from Example 3.20 is the indicator SAFE and the instantiation of the partial order $\{(p,0),(o,1)\}$ for $<_F$. Two passes are made through the repeat loop. The first pass updates element $(p,0)$ of the partial order and the second pass updates $(o,1)$.

### Genify: Adding Range Expressions to Subformulas

The next step of the translation process converts an evaluable formula into an *allowed* form. The definition of *allowed* is as follows:

**Definition 3.6** *Allowed:* A formula $F$ is *allowed* or has the *allowed property* if the following conditions are met:

| | | |
|---|---|---|
| $gdb(x, x = y)$ | **if** | $y <_F x$ |
| $gdb(x, x = \beta\{\vec{y}\})$ | **if** | $\vec{y} <_F x$ |
| $gdb(x, x \in y)$ | **if** | $y <_F x$ |
| $gdb(x, x \in \beta\{\vec{y}\})$ | **if** | $\vec{y} <_F x$ |

---

| | | |
|---|---|---|
| $gen(x, A, A)$ | **if** | $edb(A)$ **and** $free(x, A)$ |
| $gen(x, A, A)$ | **if** | $gdb(x, A)$ |
| | | |
| $gen(x, \neg A, G)$ | **if** | $gen(x, pushnot(\neg A), G)$ |
| $gen(x, \exists y A, G)$ | **if** | $distinct(x, y)$ **and** $gen(x, A, G)$ |
| $gen(x, \forall y A, G)$ | **if** | $distinct(x, y)$ **and** $gen(x, A, G)$ |
| $gen(x, A \vee B, G_1 \vee G_2)$ | **if** | $gen(x, A, G_1)$ **and** $gen(x, B, G_2)$ |
| $gen(x, A \wedge B, G)$ | **if** | $gen(x, A, G)$ |
| $gen(x, A \wedge B, G)$ | **if** | $gen(x, B, G)$ |

---

| | | |
|---|---|---|
| $con(x, A, A)$ | **if** | $edb(A)$ **and** $free(x, A)$ |
| $con(x, A, A)$ | **if** | $gdb(x, A)$ |
| $con(x, A, \bot)$ | **if** | $notfree(x, A)$ |
| | | |
| $con(x, \neg A, G)$ | **if** | $con(x, pushnot(\neg A), G)$ |
| $con(x, \exists y A, G)$ | **if** | $distinct(x, y)$ **and** $con(x, A, G)$ |
| $con(x, \forall y A, G)$ | **if** | $distinct(x, y)$ **and** $con(x, A, G)$ |
| $con(x, A \vee B, G_1 \vee G_2)$ | **if** | $con(x, A, G_1)$ **and** $con(x, B, G_2)$ |
| $con(x, A \wedge B, G)$ | **if** | $gen(x, A, G)$ |
| $con(x, A \wedge B, G)$ | **if** | $gen(x, B, G)$ |
| $con(x, A \wedge B, G_1 \vee G_2)$ | **if** | $con(x, A, G_1)$ **and** $con(x, B, G_2)$ |

Figure 3.5: Extended rules of $gen$ and $con$ that produce "generators".

1. For every variable $x$ that is free in $F$, $gen(x, F)$ holds.
2. For every subformula $\exists x A$ of $F$, $gen(x, A)$ holds.
3. For every subformula $\forall x A$ of $F$, $gen(x, \neg A)$ holds.

The allowed property is stronger than evaluable since every formula satisfying the allowed property satisfies the evaluable property (because $gen(x, F)$ implies $con(x, F)$), but the converse does not hold. Every evaluable formula can be translated into an equivalent allowed formula. The desired properties of allowed formulas are that all variables, free and bound, are generated from the formula and allowed formulas are more robust under certain transformations than evaluable ones. Gelder and Topor [GT91] define *conservative transformations* that include $\vee$ and $\wedge$ distribution that do not always preserve the evaluable property, but do preserve the allowed property. These transformations are used in subsequent steps of the translation to algebra and for this reason evaluable formulas are converted into an equivalent allowed form.

**Algorithm 3.1** *evalify:*

**Input:** An object calculus formula $F$

**Output:** SAFE indicating that $F$ is evaluable or REJECT

**Comments:** The algorithm incrementally builds the global partial order $<_F$ with each pass through the **repeat** loop. A temporary set $V$ is used to store elements of $<_F$ that need to be updated after each pass.

**Initialization**

1. For every variable $x_i$ appearing in $F$, initialize a pair $(x_i, \infty)$ in $<_F$. This indicates that the order for $x_i$ is undefined.
2. $order = 0$

**Procedure:**

**repeat**

$V = \{\ \}$
**foreach** undefined element $(x_i, \infty)$ in $<_F$ **do**
**if** $free(x_i, F)$ **then**
apply $gen(x_i, F)$
**else if** $x_i$ is $\exists$ bound as $\exists x A$ **then**
apply $con(x_i, A)$
**else** $x_i$ must be $\forall$ bound as $\forall x A$
apply $con(x_i, \neg A)$
**if** *gen* or *con* application succeeded **then**
$V = V \cup \{(x_i, \infty)\}$
**endfor**
**foreach** element $(x_i, \infty)$ in $V$ **do**
update element $(x_i, \infty)$ in $<_F$ to $(x_i, order)$ which defines its order
**endfor**
**if** no more undefined elements $(x_i, \infty)$ in $<_F$ **then** return SAFE
increment *order*

**until** no changes made to $<_F$

return REJECT

As an example of allowed vs. evaluable, consider the formula:

$$P(p) \ \wedge \ \exists q(Q(q) \vee (R(q) \wedge p.\alpha = q.\alpha))$$

which is allowed and the formula:

$$P(p) \ \wedge \ \exists q(Q(q) \vee (\neg R(p) \wedge p.\alpha = p.\beta))$$

which is evaluable, but not allowed because $gen(q, Q(q) \vee (\neg R(p) \wedge p.\alpha = p.\beta))$ does not hold.

Algorithm 3.2 (*genify*) follows the *con-to-gen* algorithm presented in [GT91] and translates an evaluable formula into one that is allowed. The basic procedure of the algorithm is to identify the subformulas $\exists x A$ such that $con(x, A)$ holds, but $gen(x, A)$ fails and then to rewrite these formulas as an equivalent formula, say $A'$, so that $gen(x, A')$ holds. From this point on, unless otherwise noted, it is assumed that all occurrences of $\forall x A$ in a formula have been replaced with the logical equivalent $\neg\exists x \neg A$. The *genify* algorithm is general in the sense that if the input formula is not evaluable, it can identify this and returns an error. It is necessary before applying the *genify* algorithm to check that $gen(x_i, F)$ holds for all free variables $x_i$ in $F$. The algorithm relies on the following definitions paraphrased from [GT91].

**Definition 3.7** *Truth Value Simplification:* The operation of *truth value simplification* consists of applying the following simplifications to a formula for as long as possible.

| | | | | | |
|---|---|---|---|---|---|
| $\neg$false | $\Longrightarrow$ | true | $\neg$true | $\Longrightarrow$ | false |
| $A \wedge$ false | $\Longrightarrow$ | false | $A \wedge$ true | $\Longrightarrow$ | true |
| $A \vee$ false | $\Longrightarrow$ | $A$ | $A \vee$ true | $\Longrightarrow$ | true |
| $\exists x$ false | $\Longrightarrow$ | false | $\exists x$ true | $\Longrightarrow$ | true |
| $\forall x$ false | $\Longrightarrow$ | false | $\forall x$ true | $\Longrightarrow$ | true |

Simplifications that depend on the law of the excluded middle, such as $A \vee \neg A \Longrightarrow$ true, are not part of this definition because, in general, $A$ is a formula and this part of the translation does not expend resources on recognizing formula equivalences.

**Definition 3.8** *Formula Substitution:* Let $G \stackrel{\text{def}}{=} P_1 \vee \cdots \vee P_m$ where $P_i$ are atoms in $A$. Then $A[G/\text{false}]$ denotes a formula in which each occurrence of $P_i$ in $A$ is replaced by false.

Steps 1-5 of the algorithm traverse the structure of the input formula and step 5 performs the transformations into allowed form on the subformulas that violate the *gen* property. If step 5a holds, then there is nothing to do here and the formula can continue to be traversed. Step 5b must hold in order for the formula to be evaluable and if it does not then an error is produced. If variable $x$ is not free in subformula $A$, this means that $x$ must not appear in $A$ and, therefore, the existential quantifier for $x$ can be dropped and the formula can continue to be traversed. The key step of the algorithm is 5(b)ii where $F$ is rewritten into the equivalent $\hat{F}$ form. The purpose of this step is to form a conjunction of the original subformula $A$ with a generator $G$ for the constrained variable $x$; in effect making $gen(x, G \wedge A)$ hold. The role of $\mathcal{R}$ is to act as the "remainder" of the subformula which moves copies of subformulas that are independent of $x$ (i.e., don't contain $x$) outside the existential quantifier for $x$. This is necessary to make $F$ and $\hat{F}$ equivalent because the conjunction of $G$ with $A$ changes the meaning of the subformula.

**Algorithm 3.2** *genify:*

**Input:** An evaluable formula $F$ with universal quantifiers replaced.

**Output:** An allowed formula equivalent to $F$.

**Procedure:**

1. **if** $F$ is an atom **then** return $F$
2. **if** $F$ has the form $\neg A$ **then** return $\neg genify(F)$
3. **if** $F$ has the form $A \wedge B$ **then** return $genify(A) \wedge genify(B)$
4. **if** $F$ has the form $A \vee B$ **then** return $genify(A) \vee genify(B)$
5. **if** $F$ has the form $\exists x A$ **then**
   (a) **if** $gen(x, A(x), G(x))$ holds **then** return $\exists x\ genify(A(x))$
   (b) **if** $con(x, A, G)$ holds **then**
      i. **if** $notfree(x, A)$ and hence $G = \bot$ **then** return $genify(A)$
      ii. **else** $free(x, A)$ holds and $G = P_1(x) \vee \cdots \vee P_m(x)$ where $m \geq 1$ and some of the disjuncts may be $\bot$. Let $\mathcal{R}$ be the truth value simplification of $A[G/\mathsf{false}]$. Define:
      $$\hat{F} \stackrel{\text{def}}{=} \exists x\,(G(x) \wedge A(x)) \vee \mathcal{R}$$
      and return $genify(\hat{F})$
   (c) Note that if $con(x, A, G)$ does not hold, then $F$ is not evaluable and an error is returned.

The result of applying *genify* to the example formula $F$ from Example 3.20 is the formula:

$$F' \stackrel{\text{def}}{=} \exists p(P(p)\ \wedge\ ((P(p)\ \wedge\ o = p.\alpha)\ \vee\ T(o)))\ \vee\ T(o)$$

which is allowed. The steps that produce this formula are as follows:

- The algorithm falls through to step 5 since $F$ has the form $\exists x A$ where:
$$A \stackrel{\text{def}}{=} (P(p)\ \wedge\ o = p.\alpha)\ \vee\ T(o)$$
- Step 5a fails, but step 5b succeeds with $con(p, A, G)$ where $G \stackrel{\text{def}}{=} P(p)\ \vee\ \bot$.
- Thus, the algorithm proceeds to step 5(b)ii and the result of applying this step to the example formula defines the following:
$$\begin{aligned} \mathcal{R} &\stackrel{\text{def}}{=} T(o) \\ \hat{F} &\stackrel{\text{def}}{=} \exists p((P(p)\ \vee\ \bot)\ \wedge\ ((P(p)\ \wedge\ o = p.\alpha)\ \vee\ T(o)))\ \vee\ T(o) \end{aligned}$$
$\hat{F}$ is in allowed form, and replacing all occurrences of $\bot$ with $\mathsf{false}$ and carrying out truth value simplification produces the output formula $F'$.

### ANFify: Making Subformulas Independent

The next step of translation is to normalize an allowed formula by putting it into *Allowed Normal Form* (ANF). The reason for converting a formula into ANF is that every proper

*constructive subformula* (see Definition 3.11 below) can generate objects for all the free variables in the subformula. This, in effect, makes every constructive subformula independent of atoms that appear outside the quantifier for the subformula. This means that the final translation to the algebra can translate subformulas independent of the atoms outside the quantifier for the subformula. The transformation of an ANF formula into an object algebra expression is straightforward by simple pattern matching starting with the inner subformulas and moving to the outer formula. At times the following discussion assumes a tree structured representation for a formula, where the leaves represent atoms from the calculus and the internal nodes are the connectives $\exists, \vee, \wedge, \neg$. Algorithm 3.3 (*ANFify*) and the definition of ANF depend on the following definitions that extend those presented in [GT91] by including a notion for membership.

**Definition 3.9** *Simplified Form:* A formula (with universal quantifiers replaced) is call *simplified* if the following conditions are met:

1. There is no occurrence of $\neg\neg A$. It is replaced by the logical equivalent $A$.
2. There are no occurrences of $\neg(s = t), \neg(s \neq t), \neg(s \in t), \neg(s \notin t)$. They are replaced by their logical equivalents $(s \neq t), (s = t), (s \notin t), (s \in t)$, respectively.
3. The operators $\wedge, \vee, \exists$ are made polyadic and are flattened, meaning:
   (a) in a subformula $A_1 \wedge \cdots \wedge A_n$, $n \geq 2$ and no operand $A_i$ is itself a conjunction,
   (b) in a subformula $A_1 \vee \cdots \vee A_n$, $n \geq 2$ and no operand $A_i$ is itself a disjunction,
   (c) in a subformula $\exists \vec{x} A$, operand $A$ does not begin with $\exists$.
4. In a subformula $\exists \vec{x} A$, $\mathit{free}(x_i, A)$ holds for every variable $x_i$.

An algorithm to translate a formula into simplified form follows immediately from the definition. A function *simplify* is assumed to exist and transforms an arbitrary formula into its equivalent simplified form satisfying Definition 3.9. The following three definitions formalize the notion of *Existential Normal Form* (ENF).

**Definition 3.10** *Negative/Positive Formulas:* A simplified formula is *negative* if its root is "$\neg$"; otherwise, it is *positive*. An arbitrary formula is *negative* (resp. *positive*) if its simplified form is negative (resp. positive). Atoms of a simplified formula of the form $s \neq t, s \notin t$ are negative and atoms of the form $s = t, s \in t$ are positive.

**Definition 3.11** *Restrictive/Constructive Subformulas:* A subformula $A$ of a simplified formula $F$ is *restrictive* if the parent of $A$ is "$\wedge$" and either $A$ is negative or $A$ is an atom and $\mathit{edb}(A)$ does not hold; otherwise $A$ is *constructive*.

**Definition 3.12** *Existential Normal Form:* A formula is in *Existential Normal Form* (ENF) if the following conditions hold:

1. The formula is simplified.
2. For each disjunction in the formula:
   (a) the parent of the disjunction, if it has one, is "$\wedge$", and
   (b) each operand of the disjunction is a positive formula.

3. The parent, if any, of a conjunction of negative formulas is $\exists$.

The existential normal form prohibits certain parent/child combinations illustrated by the nonblank entries in Figure 3.6. These entries specify rewrite rules that convert the prohibited combinations into permitted ones. The $s$ along the diagonal indicates a call to *simplify* on the formula and has the highest priority. The definition of ENF in [EMHJ93a, EMHJ93b] points to a shortcoming in [GT91] that does not properly transform conjunctions of negated formulas with a disjunctive parent into the algebra. For this reason, condition 3 is included in the definition of ENF and rule R1B is added as a rewrite rule in Figure 3.6.

| | Parent | | | |
|---|---|---|---|---|
| Child | $\vee$ | $\wedge$ | $\exists$ | $\neg$ |
| $\vee$ | $s$ | | R3 | R2 |
| $\wedge$ | R1B | $s$ | | R1 |
| $\exists$ | | | $s$ | |
| $\neg$ | R1A | | | $s$ |

R1 : $\neg(\neg A_1 \wedge \cdots \wedge \neg A_n) \implies A_1 \vee \cdots \vee A_n$
Only if every conjunct of $\wedge$ is negative.

R1A : $\neg A \vee B_1 \vee \cdots \vee B_n \implies \neg(A \wedge \neg B_1 \wedge \cdots \wedge \neg B_n)$

R1B : $(\neg A_1 \wedge \cdots \wedge \neg A_n) \vee B_1 \vee \cdots \vee B_m \implies \neg((A_1 \vee \cdots \vee A_n) \wedge \neg B_1 \wedge \cdots \wedge \neg B_m)$
Only if every conjunct of $\wedge$ in the formula on the left is negative.

R2 : $\neg(A_1 \vee \cdots \vee A_n) \implies (\neg A_1 \wedge \cdots \wedge \neg A_n)$

R3 : $\exists \vec{x}(A_1(\vec{x}) \vee \cdots \vee A_n(\vec{x})) \implies (\exists \vec{x_1} A_1'(\vec{x_1}) \vee \cdots \vee \exists \vec{x_n} A_n'(\vec{x_n}))$
Where variables $\vec{x_i}$ do not appear in the formula on the left and
$A_i'$ is the result of renaming $\vec{x}$ with $\vec{x_i}$.

Figure 3.6: Prohibitive parent/child combinations in ENF formulas and rewrite rules to correct the violations. The $s$ entry indicates a call to *simplify* on the formula and has highest priority.

Defining an algorithm for converting any arbitrary formula into ENF is straightforward from Figure 3.6. Algorithms are presented in both [GT91] and [EMHJ93b]. Furthermore, Lemmas are provided stating that if the input formula to the ENF algorithm is allowed, then so is the output formula. This means that an allowed formula can be converted to ENF without losing the allowed property. ENF is important for the final translation into ANF. Let *ENFify* be a function that performs ENF normalization.

The following two definitions formalize the notion of *allowed normal form*.

**Definition 3.13** *genall:* The property $genall(F)$ holds for a formula $F$ if and only if $gen(x_i, F)$ holds for every free variable appearing in $F$.

**Definition 3.14** *Allowed Normal Form:* A formula $F$ is in *Allowed Normal Form* (ANF) if it is in ENF, $genall(F)$ holds, and every constructive subformula $A$ of $F$ is in ANF.

Algorithm 3.3 (*ANFify*) transforms an allowed ENF formula into an equivalent ANF formula. The algorithm is based on the repeated application of the rewrite rules in Figure 3.6. Application of rules for Case 1 and Case 2 require the resulting formula to be

simplified before recursing on the formula. Case 3 may produce a non-ENF formula (e.g., $D \wedge \neg((A_1 \vee A_2) \wedge B)$) and so a call to *ENFify* is necessary before recursing. A fixpoint of the algorithm is reached when no changes are made to the input formula $F$ and at this point $F$ is in allowed normal form.

The purpose of the *ANFify* algorithm is to rewrite every proper constructive subformula so that all free variables in the subformula are generated by the subformula itself. This ensures that every constructive subformula is allowed and therefore can be "evaluated" independently of the atoms outside the quantifier for this formula. This motivates the following Lemma that removes the recursion in Definition 3.14, but yields the same class of ANF formulas.

**Lemma 3.2** An ENF formula $F$ is in ANF if and only if $F$ is allowed and every constructive subformula $A$ of $F$ is allowed.
**Proof:** Immediate from the definition of ANF and structural induction on $F$. The reader is referred to [GT91] for the formal proof.

The result of applying *ANFify* to the allowed formula $F'$ produced by the *genify* algorithm in the previous section is the formula:

$$F'' \stackrel{\text{def}}{=} \exists p(P(p) \ \wedge \ o = p.\alpha) \ \vee \ \exists p(P(p) \ \wedge \ T(o)) \ \vee \ T(o)$$

which is in ANF. The steps that produce this formula are as follows:

- The algorithm matches on Case 3 with the following being defined from the formula $F'$:

$$\begin{aligned} F_1 &\stackrel{\text{def}}{=} P(p) \ \wedge \ ((P(p) \ \wedge \ o = p.\alpha) \ \vee \ T(o)) \\ B_1 &\stackrel{\text{def}}{=} P(p) \\ G &\stackrel{\text{def}}{=} ((P(p) \ \wedge \ o = p.\alpha) \ \vee \ T(o)) \\ A_1 &\stackrel{\text{def}}{=} (P(p) \ \wedge \ o = p.\alpha) \\ A_2 &\stackrel{\text{def}}{=} T(o) \end{aligned}$$

  Carrying out the distribution of $B_1$ over $G$ produces two $G_i$ formulas that are in ANF and define the final result formula as follows:

$$\begin{aligned} G_1 &\stackrel{\text{def}}{=} (P(p) \ \wedge \ P(p) \ \wedge \ o = p.\alpha) \\ &\equiv (P(p) \ \wedge \ o = p.\alpha) \\ G_2 &\stackrel{\text{def}}{=} (P(p) \ \wedge \ T(o)) \\ F_2 &\stackrel{\text{def}}{=} (P(p) \ \wedge \ o = p.\alpha) \ \vee \ (P(p) \ \wedge \ T(o)) \\ F[F_1/F_2] &\stackrel{\text{def}}{=} \exists p((P(p) \ \wedge \ o = p.\alpha) \ \vee \ (P(p) \ \wedge \ T(o))) \ \vee \ T(o) \end{aligned}$$

  The call to *ENFify* on $F$ distributes the $\exists p$ over the disjunct. The resulting formula is in ANF and is the output of *ANFify* as formula $F''$.

**Algorithm 3.3** *ANFify:*

**Input:** An allowed formula $F$ in ENF.

**Output:** An ANF formula equivalent to $F$.

**Comments:**

The algorithm assumes a tree structure representation of formulas. The notation $F[A/B]$ where $A$ is a subtree (subformula) of $F$ denotes an operation that replaces the subtree of $A$ in $F$ by the tree representation of formula $B$.
In each of the cases below, $F_1$ is an allowed (not necessarily proper) subformula of $F$ to be replaced and $F_2$ is the equivalent allowed formula that replaces $F_1$. The notation "$F_1 \stackrel{\text{def}}{=} \cdots$" means that $F_1$ matches the allowed formula pattern on the right-hand side. If none of the patterns can be matched to some subformula of $F$, the algorithm falls through to the otherwise clause which causes the procedure to terminate.

**Procedure:**

**Case 1:** $F_1 \stackrel{\text{def}}{=} \exists \vec{y} A \wedge B_1 \wedge \cdots \wedge B_n$, and $genall(A)$ does not hold:

- Let $\vec{x}$ be the set of variables that are free in $A$ such that $gen(x_i, A)$ fails (since $F_1$ is allowed, this set is disjoint from $\vec{y}$).
- Let $B_1 \wedge \cdots \wedge B_k$ be a prefix (possibly after rearrangement) of $B_1 \wedge \cdots \wedge B_n$ such that $genall(A \wedge B_1 \wedge \cdots \wedge B_k)$ holds (at worst $k = n$ because $genall(F_1)$ holds).
- Let $F_2 \stackrel{\text{def}}{=} \exists \vec{y}(A \wedge B_1 \wedge \cdots \wedge B_k) \wedge B_{k+1} \wedge \cdots \wedge B_n$
- return $ANFify(simplify(F[F_1/F_2]))$

**Case 2:** $F_1 \stackrel{\text{def}}{=} \neg A \wedge B_1 \wedge \cdots \wedge B_n$, and $genall(A)$ does not hold:

- Let $\vec{x}$ be the set of variables that are free in $A$ such that $gen(x_i, A)$ fails.
- Let $B_1 \wedge \cdots \wedge B_k$ be a prefix (possibly after rearrangement) of $B_1 \wedge \cdots \wedge B_n$ such that all $\vec{x}$ are free in $B_1 \wedge \cdots \wedge B_k$ and $genall(B_1 \wedge \cdots \wedge B_k)$ holds (at worst $k = n$ because $genall(F_1)$ holds).
- Let $G \stackrel{\text{def}}{=} ANFify(B_1 \wedge \cdots \wedge B_k)$
- Let $F_2 \stackrel{\text{def}}{=} \neg(A \wedge G) \wedge B_1 \wedge \cdots \wedge B_n$
- return $ANFify(simplify(F[F_1/F_2]))$

**Case 3:** $F_1 \stackrel{\text{def}}{=} G \wedge B_1 \wedge \cdots \wedge B_n$, where $G \stackrel{\text{def}}{=} A_1 \vee \cdots \vee A_m$ and $genall(G)$ does not hold:

- Let $B_1 \wedge \cdots \wedge B_k$ be a prefix (possibly after rearrangement) of $B_1 \wedge \cdots \wedge B_n$ such that $genall(G \wedge B_1 \wedge \cdots \wedge B_k)$ holds (at worst $k = n$ because $genall(F_1)$ holds).
- Distribute $B_1 \wedge \cdots \wedge B_k$ over $G$.
- For $1 \leq i \leq m$ do: let $G_i \stackrel{\text{def}}{=} ANFify(A_i \wedge B_1 \wedge \cdots \wedge B_k)$
- Let $F_2 \stackrel{\text{def}}{=} (G_1 \vee \cdots \vee G_m) \wedge B_{k+1} \wedge \cdots \wedge B_n$
- return $ANFify(ENFify(F[F_1/F_2]))$

**Otherwise:** return $F$

### Transform: Translating into Algebra

The final step of translation involves the transformation of an ANF formula into an equivalent series of object algebra operations. This step follows immediately from the structure of an ANF formula. Every range atom $C(x)$ is translated into a name $C'$ that represents the collection of the range predicate. Atoms $x = c$ and $x \in c$ are translated as part of appropriate select or generate operations (see below) or into appropriate collections as follows:

$$\begin{aligned} x = c &\implies newcoll(c)_x \\ x \in c &\implies c_x \end{aligned}$$

The first case creates a collection containing the single constant $c$ and the second case uses $c$ as the name for the collection. Recall that the subscript $x$ is the notation from Section 3.5.2 indicates that the result collection is a range for variable $x$.

Next, the transformations shown in Figure 3.7 are applied to the remaining proper constructive subformulas and then the subformulas are combined. In the figure, $A$ and $B$ refer to subformulas, $A'$ and $B'$ refer to the algebraic equivalents of $A$ and $B$ respectively, $F$ refers to a predicate, *mop* refers to a mop function, and $\theta$ is one of $=$ or $\in$. Algebraic expressions are subscripted with the variables that they represent (or that their components represent in the case of product objects). Furthermore, $A(\vec{x})$ is used to denote that $\vec{x}$ are the only free variables in $A$. The same applies to $F(\vec{x})$ and $mop(\vec{x})$. For join terms of the form $\vec{x} = \vec{x}$, it is assumed that one set of $\vec{x}$ refer to components of $A'$ while the other set refers to components of $B'$.

Transformation (3.7) is known as a *generalized set difference* [HHT75] and could be defined as a primitive derived operator in the algebra so that efficient join techniques could be defined to process it.

Transformation (3.9) defines a join between the common variables (components) of $A$ and $B$. Transformation (3.10) defines a join using a predicate (general mop function) over the components of $A$ and $B$. Transformation (3.11) is a general case of (3.9) and (3.10) which defines a join between an $A$ and a $B$ that have some variables in common (namely, $\vec{w}, \vec{x}$), some variables not in common ($A$ has $\vec{u}, \vec{v}$ and $B$ has $\vec{y}, \vec{z}$), and a predicate over some of the common and uncommon variables of $A$ and $B$ (namely, $\vec{u}, \vec{w}, \vec{y}$). Transformation (3.13) defines a generate join over an $A$ and a $B$ that have no variables in common, and have a generating atom over some of the variables of $A$ and $B$. The reason $A$ and $B$ cannot have common variables is that the relationship between these variables would be lost in the operation. If $A$ and $B$ have common variables, then they should be joined instead. Transformation (3.12) is a special case of (3.13) where there is only one formula generating the result.

Transformations of join and generate join over two operands can be generalized over multiple operands. For example, there is the opportunity to perform the following transformations on the given formula:

$$\begin{aligned} A(\vec{x}) \wedge B(\vec{y}) \wedge C(\vec{z}) \wedge F(\vec{x}, \vec{y}, \vec{z}) &\implies (A_{\vec{x}} \bowtie_F \langle B_{\vec{y}}, C_{\vec{z}} \rangle)_{\vec{x},\vec{y},\vec{z}} \\ A(\vec{x}) \wedge B(\vec{y}) \wedge C(\vec{z}) \wedge o = mop(\vec{x}, \vec{y}, \vec{z}) &\implies (A_{\vec{x}}\ \gamma^{o}_{o=mop}\ \langle B_{\vec{y}}, C_{\vec{z}} \rangle)_{\vec{x},\vec{y},\vec{z},o} \end{aligned}$$

This groups the collections involved in the operation with the operator and may provide some opportunities for optimization such as grouping together collections that may be clustered on disk.

$$A(\vec{x}) \vee B(\vec{x}) \implies (A'_{\vec{x}} \cup B'_{\vec{x}})_{\vec{x}} \quad (3.3)$$

$$A(\vec{x}) \wedge B(\vec{x}) \implies (A'_{\vec{x}} \cap B'_{\vec{x}})_{\vec{x}} \quad (3.4)$$

$$A(\vec{x}) \wedge B(\vec{y}) \implies (A'_{\vec{x}} \times B'_{\vec{y}})_{\vec{x},\vec{y}} \quad (3.5)$$

$$A(\vec{x}) \wedge \neg B(\vec{x}) \implies (A'_{\vec{x}} - B'_{\vec{x}})_{\vec{x}} \quad (3.6)$$

$$A(\vec{x},\vec{y}) \wedge \neg B(\vec{y}) \implies (A'_{\vec{x},\vec{y}} - (A'_{\vec{x},\vec{y}} \bowtie_{\vec{y}=\vec{y}} B'_{\vec{y}})_{\vec{x},\vec{y}})_{\vec{x},\vec{y}} \quad (3.7)$$

$$A(\vec{x},\vec{y}) \wedge F(\vec{x}) \implies (A'_{\vec{x},\vec{y}}\ \sigma_F\ )_{\vec{x},\vec{y}} \quad (3.8)$$

$$A(\vec{x},\vec{y}) \wedge B(\vec{x},\vec{z}) \implies (A'_{\vec{x},\vec{y}} \bowtie_{\vec{x}=\vec{x}} B'_{\vec{x},\vec{z}})_{\vec{x},\vec{y},\vec{z}} \quad (3.9)$$

$$A(\vec{x},\vec{y}) \wedge B(\vec{w},\vec{z}) \wedge F(\vec{y},\vec{z}) \implies (A'_{\vec{x},\vec{y}} \bowtie_F B'_{\vec{w},\vec{z}})_{\vec{x},\vec{y},\vec{w},\vec{z}} \quad (3.10)$$

$$A(\vec{u},\vec{v},\vec{w},\vec{x}) \wedge B(\vec{w},\vec{x},\vec{y},\vec{z}) \wedge F(\vec{u},\vec{w},\vec{y}) \implies (A' \bowtie_{\vec{w}=\vec{w}\wedge\vec{x}=\vec{x}\wedge F} B')_{\vec{u},\vec{v},\vec{w},\vec{x},\vec{y},\vec{z}} \quad (3.11)$$

$$A(\vec{x},\vec{y}) \wedge o\theta mop(\vec{x}) \implies (A'_{\vec{x},\vec{y}}\ \gamma^{o}_{o\theta mop}\ )_{\vec{x},\vec{y},o} \quad (3.12)$$

$$A(\vec{x},\vec{y}) \wedge B(\vec{w},\vec{z}) \wedge o\theta mop(\vec{y},\vec{z}) \implies (A'_{\vec{x},\vec{y}}\ \gamma^{o}_{o\theta mop}\ B'_{\vec{w},\vec{z}})_{\vec{x},\vec{y},\vec{w},\vec{z},o} \quad (3.13)$$

$$\exists\vec{y} A(\vec{x},\vec{y}) \implies (A'_{\vec{x},\vec{y}} \triangle_{\vec{y}})_{\vec{x}} \quad (3.14)$$

Figure 3.7: Transformations from object calculus to object algebra.

The last stage of the transformation is to apply the necessary project operation using behavioral projections in the target list of the object calculus expression. This operation does not change the extent of the result collection. Rather, it has the effect of generalizing a new membership type for the collection that only includes the behaviors specified in the projection.

The result of applying the transformations to the ANF formula $F''$ output by the *ANFify* algorithm in the previous section is the algebraic expression:

$$\left((P_p\ \gamma^{o}_{o=p.\alpha}\ )_{p,o}\triangle_p\right)_o \ \cup\ ((P_p \times T_o)_{p,o}\triangle_p)_o \ \cup\ T_o$$

Written using the constructs of the original query it is:

$$\left((\mathbf{C}\_\text{person}\ \gamma^{o}_{o=p.B\_\text{residence}.B\_\text{inZone}}\ )\triangle_p\right) \cup ((\mathbf{C}\_\text{person} \times \mathbf{C}\_\text{transport})\triangle_p) \ \cup\ \mathbf{C}\_\text{transport}$$

There are opportunities for optimization on this expression, but the importance of this section was to show the correct translation from calculus to algebra. The expression should also be type checked to ensure that the behaviors used in the expression are actually defined for the objects to which they're being applied. During type checking the test for operand finiteness can also take place. The resulting example query is safe in all respects that have been considered in this thesis.

A formal complexity analysis of the entire algorithm remains open. The completion of this task may yield improvements to the algorithm. Termination of the algorithm is proven in both [GT91] and [EMHJ93b]. The object generation extension to the algorithm described in this thesis does not inhibit termination. First, only non-recursive *gdb* logical rules are added to the original *gen* and *con* rules. Second, the *evalify* algorithm is extended with a **repeat** loop with two embedded mutually exclusive **foreach** loops. The **foreach** loops are always guaranteed to terminate since the partial order $<_F$ and the intermediate set $V$ that they range over must be finite. The **repeat** loop must eventually terminate because it exits when no changes are made to the partial order $<_F$ and every iteration through

the loop only changes undefined elements in the partial order to defined elements. Thus, the number of undefined elements in the partial order (and hence possible changes to the partial order) can only decrease with every iteration, eventually reaching the fixpoint when no changes are made and terminating. The last **if** statement in the algorithm may cause earlier termination if the formula being evaluated in safe. Finally, the algorithms *genify*, *ENFify*, and *ANFify* are virtually the same as those presented in [GT91] with the ENF extension outlined in [EMHJ93b] incorporated into the *ENFify* algorithm. The interested reader is referred to those papers for the formal proofs.

The main contribution of the approach presented in this thesis is the extension of the evaluable class (and hence the allowed class) to incorporate the notion of object generation through equality and membership atoms. A second contribution is the calculation of the partial order that defines the steps in which the object generation can be performed. Furthermore, a prototype of a calculus to algebra translator based on the given algorithms has been implemented [Lip93] and the initial indications of its performance on sample queries are quite positive.

# Chapter 4

# The Meta-Model and Reflection

In this chapter[1], the features of the TIGUKAT meta-model (Section 2.4.6) are described and how it provides *reflective capabilities* is shown. *Reflection* is the ability of a system to manage information about itself and to access (or reason about) this information using the regular access primitives of the model. The ability of a model to manage information about itself is a strength because meta-information (like schema) is modeled as first-class components of the objectbase and the access primitives of the model can be uniformly used to access all information, including the meta-information like the schema. The uniformity built into the TIGUKAT object model is used to represent the meta-model and gives a clean semantics for reflection.

## 4.1 Related Work

In recent years, work on reflection in object-oriented languages (OOLs) has resulted in the identification of two basic models of reflection [Fer89]:

1. The first is called *structural reflection* and was advocated by Cointe [Coi87] in the design of ObjVlisp. The model is based on a uniform instance/class/meta-class architecture where everything is an object and meta-classes are proper classes in the sense that they can have a number of instances and can be subclassed. The discrimination between meta-classes, classes and other instances is only a consequence of inheritance and not a type distinction. This is in contrast to Smalltalk-80 [GR89] where meta-classes are anonymous objects and there is a one-to-one correspondence between a class and its meta-class.

2. The second is called *computational reflection* and was pursued by Paes [Mae87] in the development of 3-KRS. This approach essentially introduces a meta-object for each object to handle the structural and computational aspects of the object. This work was done within the context of a model that does not support the traditional class/instance structure of Smalltalk, ObjVlisp, TIGUKAT, etc., and so the structural aspects of objects are represented by the meta-objects as well. In a class/instance model, the structural aspects can be handled by the type (class) of the object and so meta-objects are only useful for computational aspects in these systems.

[1] Portions of this chapter are published in the 1993 *Proceedings of the Twelfth International Conference on Entity–Relationship Approach (ERA'93)* [PÖ93].

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

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

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

(c) the *meta-communication model*, which is based on the reification[2] of messages sent to objects.

Some work has been done on adding computational reflection to Smalltalk-80 [FJ89] and work on the ABCL/R2 language [MMWY92] is striving towards an efficient implementation of a reflective OOL with concurrency.

The TIGUKAT model supports structural reflection similar to (1) and computational reflection is handled by a meta-class model as in (2a).

A meta-object model (2b) was not chosen because of the additional overhead involved. One overhead is the introduction of a meta-object for (potentially) each object in the system. Another, more important one in light of an implementation, is the additional dispatch processing required for **every** behavior applied to an object. The application of behaviors to objects is the fundamental information access primitive of TIGUKAT. In the implementation of TIGUKAT [Ira93], measures were taken to speed up the execution of behavior application and even a trade-off of space for execution speed was made. In a meta-object approach, every behavior application needs to perform an additional check to see if the object has a meta-object and to dispatch the behavior to the meta-object if it exists. This overhead was unacceptable because we believe there are only a few occasions where objects need to support the semantics of meta-objects and the additional costs for every behavior application is too great. Besides, the semantics of meta-objects can be supported through subtyping and schema evolution (features required of an OBMSs anyway). Another anomaly with the meta-object approach is that some information is at the type level and some information is at the object level. The distribution of type information on a per object basis has implications for persistent object management (e.g., where to store the meta-object: with the type, with the object, or somewhere else?). Finally, since behaviors are objects in TIGUKAT, some form of the meta-communication model (2c) could be integrated with the system. Part of the future research is to investigate the incorporation of these semantics into TIGUKAT.

## 4.2 Overview

*Reflection* is the ability of a system to manage information about itself and to access (or reason about) this information through the regular "channels" of information retrieval. It is natural for an OBMS to manage information about itself since an OBMS is nothing more than a complex application defined by a model.

[2]Reification deals with the re-packaging and passing of messages on to other objects. It is based on the premise that messages are objects that can be sent messages to process themselves. Behaviors in TIGUKAT adhere to this semantics.

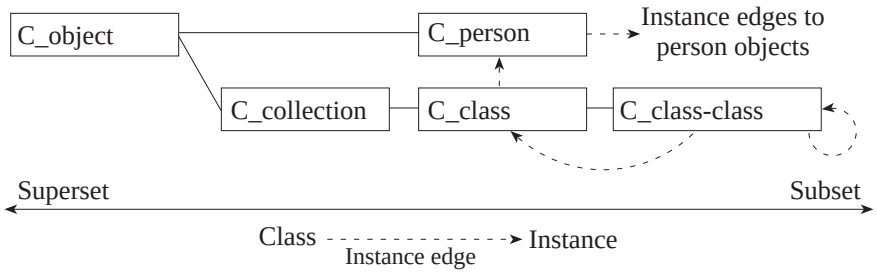



Figure 4.1: A "normal" class and instance structure for **C_person**.

There are several advantages in managing information within a model. One advantage is that the primitives of the model are used to manage all forms of information including meta-information as first-class components (*uniformity of representation*). Another advantage is that information retrieval is uniformly handled by the model's access primitives regardless of the information's type or "status" (*uniformity of access and manipulation*). With these two abilities, a system is capable of *reflection*. Relational systems provide reflective capabilities by using relations to store information (i.e., schema) about relations. However, the attributes of relations are restricted to the atomic domains of a particular system (i.e., integers, strings, dates, etc.), which limits the semantic richness of the meta-information and makes it awkward to model. With the richer type structures of object models, self management and reflection is more natural and easier to manage.

In a uniform object model like TIGUKAT, the same structures used to manage information about "normal" real-world objects such as persons, houses, maps, or complex applications (e.g., a geographic information system) are also used to manage meta-information like types, classes, behaviors, and functions. Furthermore, the access primitives to all these forms of information are uniform, meaning there is no distinction, for example, between accessing information about persons and accessing information about types. The uniformity of TIGUKAT is the basis for its reflective capabilities.

## 4.3 Features of the Meta-Model

One feature of the meta-model is that it can be used to uniformly define an $m^2$-class whose associated type includes behaviors for creating default objects of a particular type. For example, consider the GIS objectbase of Section 2.3 and assume that type `T_person` and class **C_person** are defined. The "normal" class and instance structure for this scenario is shown in Figure 4.1.

Instances of `T_person` are created by applying *B_new* to class **C_person**. However, the *B_new* behavior used in this case is the one defined on `T_class` which has a generic implementation of creating a new "empty" object as an instance of the receiver class (i.e., a new "empty" person instance of **C_person**). Most existing models allow some form of specialized *new* behavior on classes. However, they are usually defined in a roundabout and non-uniform way by stating that a class can have a *new* behavior defined that is applicable to itself (e.g., C++ [Str91b]). This is non-uniform since a class defines some behaviors that are applicable to its instances and some that are applicable to itself. Other models get around this by stating that every class is an instance of itself (e.g., Modular Smalltalk [WBW88b]), but in a uniform model this approach raises the question: is the class of persons a person?

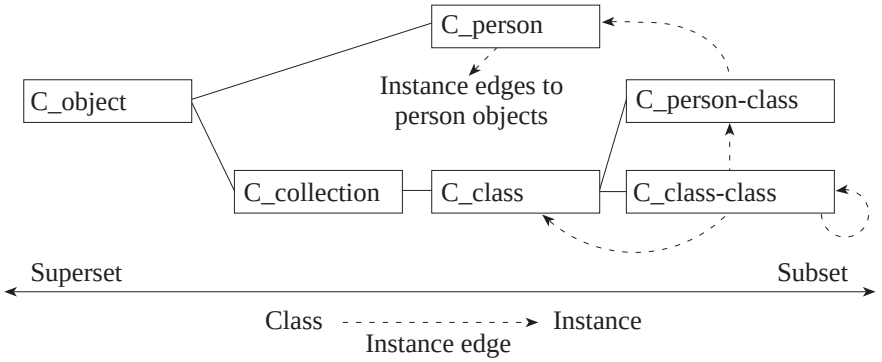



Figure 4.2: An $m^2$ class and instance structure for **C_person**.

What is needed is a uniform way of defining a behavior *B_new* for **C_person** that creates new objects of type `T_person` with some default information. It would not make sense to define this behavior on type `T_class`, since then it would be applicable to all classes and it should only be applicable to **C_person**. The solution lies in the $m^2$-objects.

First, a new type called `T_person-class` is created as a subtype of `T_class` and will specialize *B_new*. The following behavior application performs this task:

$$\texttt{T\_person-class} \leftarrow \textbf{C\_type}.B\_new(\{\texttt{T\_class}\},\{\ \})$$

Following this, the implementation of the inherited behavior *B_new* is redefined to create person objects with some default information (i.e., age set to 0, birthdate set to current date, etc.). To accomplish this, a new function is created with the appropriate code that performs the necessary actions, and this function is associated with *B_new* on type `T_person-class`. In the following discussion, this task is assumed to be completed. Next, an $m^2$-class **C_person-class** is created and associated with type `T_person-class` so that an instance of this type can be created. The following step creates the $m^2$-class:

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

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

$$\textbf{C\_person} \leftarrow \textbf{C\_person-class}.B\_new(\texttt{T\_person})$$

This series of behavior applications results in a class and instance structure shown in Figure 4.2. Now, the class **C_person** is an instance of **C_person-class** and thus the *B_new* behavior (the one defined on `T_person-class`) may be applied to it to create a new person with default information (i.e., **C_person**.*B_new*() creates a new person with defaults as dictated by the particular implementation). This gives a uniform semantics for the creation and management of objects. Furthermore, the example meta-system for persons was created in a uniform way using the primitives of the TIGUKAT object model.

Another feature of the meta-model is that the $m^2$-classes support a uniform definition of *class behaviors* (i.e., behaviors that are applicable to classes). For example, a behavior *B_averageAge* can be defined on type `T_person-class` that computes the average age of persons in a class. Now, this behavior is applicable to the class **C_person** and applying

it as **C_person**.*B_averageAge*() yields the average age of the persons in the objectbase. If `T_person` is subtyped by a `T_student` and the same semantics should be associated with class **C_student**, then **C_student** is created as an instance of **C_person-class**[3]. Then *B_averageAge* is applicable to **C_student** and computes the average age of the students in the objectbase. Any number of "person-like" classes (employee, teaching assistant, etc.) can be created in this way and have these semantics attached to them. A similar approach can be used to generalize this concept to collections. That is, define *collection behaviors*, such as *B_averageAge*, which are applicable to collections and can be used to compute various results from the elements of collections.

The meta-system architecture of TIGUKAT is similar to the meta-class structure in ObjVlisp [Coi87] and it is a generalization of the Smalltalk-80 [GR89] parallel one-to-one class/meta-class lattice because it is entirely uniform. Every class, including the $m^2$ classes is a proper class which, in general, have multiple instances and can be subclassed (i.e., their associated types can be subtyped). Furthermore, the TIGUKAT meta-architecture is closed, unlike the TAXIS [MBW80, LM79] and Telos [KMSB89] models which handle meta modeling by allowing the definition of an arbitrary number of meta-class levels where each subsequent meta-class level models the level below it. The uppermost meta-class level is not modeled within these models since that would require another meta-class level to be added which would not be modeled in the model, and so on.

One advantage of this approach is that there is less overhead for those classes that don't need additional class behaviors or don't need to specialize class behaviors. For example, both **C_person** and **C_student** can be defined as instances of **C_person-class** if **C_student** doesn't require additional class behaviors or specialization of existing ones. Furthermore, those classes that don't require any class behaviors can be instances of the general **C_class**. This illustrates that $m^2$ classes are classes in general whose instances are class objects.

A (potential) disadvantage is that the schema needs to be reorganized if at a later time it is decided that additional class behaviors are needed for certain classes that were grouped as instances of one meta-class (e.g., if additional behaviors are needed which are applicable to **C_student**, but not applicable to **C_person**). This kind of "evolution" can be viewed as correcting design problems of an application (i.e., it was a design mistake to create **C_student** as an instance of **C_person-class**). The problem is corrected by subtyping `T_person-class` with `T_student-class`, defining the new behaviors and specializations on this type, creating an associated class **C_student-class**, and migrating **C_student** as an instance of **C_student-class**. This reorganization is necessary because both structural and computational reflection are handled by the type. The frequency of this kind of schema reorganization in existing systems seems to be low. Nonetheless, with the development of the schema evolution policies in Chapter 5, these kinds of changes follow naturally since some form of them must be supported in a full-featured OBMS anyway.

Another approach is to introduce a *meta-object* for each object to handle the object's computational aspects [Mae87, Fer89]. This avoids schema reorganization by allowing behaviors to be redefined in the meta-objects instead of the type. However, it requires some additional dispatch processing to determine if an object has a meta-object and if it does, to tell the meta-object to handle the behavior. If the object doesn't have a meta-object, then the regular type dispatch should occur. Furthermore, there are additional space require-

[3]Alternatively, a type `T_student-class` could be created as a subtype of `T_person-class`, a class **C_student-class** could be created and associated with `T_student-class`, and **C_student** could be created as an instance of **C_student-class**. This approach requires the creation of additional objects, but has the benefit of allowing the behaviors applicable to **C_student** to be specialized.

ments since every object can potentially have a meta-object. The drawback in an OBMS application environment is that efficient query processing is a must and the overhead of the additional dispatch processing for **every** behavior application can become quite significant in queries where many behaviors are being applied. Thus, the flexibility of meta-objects (that can be supported through subtyping instead) is traded for speed.

The goal of uniformity is that the representation and semantics of the meta-system (and beyond) should be no different than it is for the "normal" real-world objects. TIGUKAT achieves this goal through the meta-system architecture described in this chapter.

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

In the same way as different "flavors" of object equality can be defined, different kinds of *new* behaviors for $m^2$-objects can also be defined. For example, `T_person-class` can define several different kinds of *new* behaviors that accept variations of arguments (such as name, age, address, etc.) and create person objects with the given arguments as initial information Furthermore, a variety of default *new* behaviors can be defined that create person objects with various defaults (e.g., *B_newBorn*, *B_newYouth*, *B_newSenior*, etc.). This illustrates another feature of the uniform meta-system architecture.

The beauty of a uniform approach is that the results of this chapter generalize over all objects in TIGUKAT, including the meta-system architecture and beyond.

## 4.4 Reflective Capabilities

Recall that *reflection* is the ability of a system or model to manage information about itself and to access this information using the regular "channels" of information retrieval in a uniform way. The architecture of the meta-system described in Chapter 2 is consistent with the modeling capabilities of the TIGUKAT object model and therefore the meta-system is uniformly defined within the model itself. The access primitives of the model (which in TIGUKAT is the application of behaviors to objects) can be uniformly applied to all objects in the system, including the meta-system, to retrieve information about objects. Thus, uniformity in TIGUKAT is a support mechanism for reflection.

The *select-from-where* clause of TQL is used to present some queries that illustrate the reflective capabilities of TIGUKAT. First, to recap its syntax, some example queries on "normal" real-world objects are given. These examples also serve to show that the method of querying real-world objects is uniform with the method for querying schema and meta-information (i.e., the syntax of the clause does not change with schema objects).

**Example 4.1** Return land zones valued over $100,000 or that cover an area over 1000 units.

**select** o
**from** o **in C_land**
**where** (o.*B_value*() > 100000) **or** (o.*B_area*() > 1000)

**Example 4.2** Return all zones that have people living in them (the zones are generated from person objects).

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

**Example 4.3** Return all maps that describe areas strictly above 5000 feet.

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

**Example 4.4** Return pairs consisting of a person and the title of a map such that the person's dwelling is in the map.

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

The above queries introduce variables (i.e., o,p,q) that range over classes and collections. The queries apply behaviors to the variables and other object references to extract information about the objects and return the information (in the form of objects) as part of the query. Since everything in the model has the status of a first-class object, the paradigm of applying behaviors to objects carries through to all objects which provides the *reflective capabilities* of the model.

The behavior application paradigm can be uniformly used on meta-objects. For example, information about types can be retrieved by querying the class **C_type**. This follows directly from the tenet of uniformity. Types are objects that are instances of the class **C_type**. The class **C_type** is associated with type `T_type`. The behaviors defined on `T_type` are applicable to types. Some example *reflective* queries on types are given below.

**Example 4.5** Return the types that have behaviors *B_name* and *B_age* defined as part of their interface.

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

**Example 4.6** Return the types that define behavior *B_age* with the same implementation as one of the supertypes.

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

**Example 4.7** Return all types that inherit behavior *B_age*, but define a different implementation from all types in the super-lattice that define behavior *B_age*.

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

**Example 4.8** Return all subtypes of `T_person`.

**select** r
**from** r **in** `T_person`.*B_sub-lattice*()

**Example 4.9** Return pairs consisting of a subtype of `T_person` and the native behaviors that the subtype defines.

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

**Example 4.10** Return pairs consisting of an object in collection **L_stuff** together with the type of the object, but only if it is a subtype of `T_zone`.

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

Carrying through the uniformity to class and collection objects, the following queries are *reflective* on classes and collections.

**Example 4.11** Return all the classes in the objectbase.

**select** o
**from** o **in C_class**

**Example 4.12** Return the classes that make up the meta-meta-system.

**select** o
**from** o **in C_class-class**

**Example 4.13** Return the collections that contain the object David. Furthermore, restrict the result to collections with a membership type of `T_person` or one of its subtypes.

**select** o
**from** o **in C_collection**
**where** o.*B_memberType*() **in** `T_person`.*B_sub-lattice*() **and** David **in** o

**Example 4.14** Return the classes that have a greater cardinality than any collection in the system without considering other classes.

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

**Example 4.15** Return pairs consisting of an $m^2$-class and the collection of native class behaviors defined by the $m^2$-class.

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

**Example 4.16** Return the objects in **L_things** that exist in at least one other collection without considering their existence in a class.

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

The uniform paradigm of behavioral application can be consistently applied to all objects in TIGUKAT since every object belongs to the extent of some class and every class is associated with a type and every type defines behaviors that are applicable to the objects in the extent of the associated class. Notice that some of the examples intermix access to "normal" objects with access to schema objects like types, classes and collections within the same query. Accessing information about any object, regardless of its "status', 'is simply a matter of applying behaviors defined by a type to the objects of that type.

The object model approach differs from relational systems that use relations to store information about relations in that the attributes of relations are limited to the atomic domains of a particular system (i.e., integers, strings, dates, etc.) while the object model has a rich type system for representing complex objects and a sophisticated execution model for applying behaviors to objects. Thus, representing schema information in a uniform object model is more natural and easier to manage. As a consequence, the access primitives apply naturally to all forms of information as well. In this chapter, it is shown how TIGUKAT supports this uniform semantics and how it is used to provide reflection.

# Chapter 5

# Schema Evolution and Versioning

In this chapter, the schema evolution policies and version control management in TIGUKAT are presented. A time domain is proposed as a foundation for managing schema changes and for tracking versions of objects. Temporality has been introduced into the TIGUKAT object model [GÖ93] and is founded on behaviors. A behavior is created to be either temporal or snapshot oriented. If a type defines a temporal behavior, then the type is temporal and all of its instances are temporal on the temporal behaviors. Thus, temporality of objects is dependent on the temporality of their type. In this chapter, only a brief overview of temporality in TIGUKAT is presented since it is part of another doctoral thesis [Gor96]. The focus of this chapter is how the temporal extensions are used to manage schema evolution and version control in TIGUKAT.

Typical client applications of OBMSs experience changes to the way in which information is organized (i.e., evolving schema). Moreover, historical tracking of the changes is usually a requirement for these applications. For example, in an engineering design application many components of an overall design may go through several modifications in order to produce a final product. Furthermore, each intermediate *version* of the component may have certain properties that need to be retained as a historical record of that particular component (e.g., the different versions may have been used in other products). The inter-connection of the various versions of components also gives rise to versions of an overall design, and the resulting designs may be part of others and so on. Efficiency considerations is another example of why an application may be modified to change the way in which it organizes information. The evolutionary characteristic of these applications requires sophisticated dynamic schema evolution policies for managing changes in schema and ensuring the overall consistency of the system.

## 5.1 Issues of Schema Evolution

Typical schema changes include adding and dropping types, adding and dropping subtype relationships between types, adding and dropping behaviors defined on a type, and, in the context of TIGUKAT, adding and dropping classes. A typical schema change can affect many aspects of a system. There are two fundamental problems to consider:

1. the effects of the change on the overall way in which the system organizes information (i.e., the effects on the schema), and

2. the effects of the change on the consistency of the underlying objects (i.e., the propagation of the changes to the existing instances). The *object migration* problem can also be considered in this context. Object migration deals with properly updating objects that change their type (i.e., migrate from one type to another). This can be perceived as a change in the object's type (i.e., a schema change) that only affects the single object. Object migration is not specifically addressed in this thesis. An additional problem to consider is the effects of the change on behaviors that access migrated instances. For example, if a behavior is dropped and the affected objects no longer respond to that behavior, then other behaviors that use the dropped behavior in their implementation will no longer work on those objects. This secondary problem has received some attention [SZ87], but more work is required. Version control based on temporality as described in this chapter is a good basis for providing solutions to this problem.

Some particular systems that have proposed solutions to these problems are examined in more detail in Section 5.3. For the first problem, the basic approach has been to define a number of invariants that must be satisfied by the schema and then to define rules and procedures for maintaining these invariants for each schema change that can occur.

For the second problem, one solution is to explicitly *coerce* objects to coincide with the new definition of the schema. This technique updates the affected objects, changing their representation as dictated by the new schema. Unless a versioning mechanism is used in conjunction with coercion, the old representations of the objects are lost. *Screening* and *conversion* are two techniques for defining when coercion actually takes place. Orion [BKKK87, KC88] is a system that uses the screening approach and GemStone [PS87] uses conversion. Other systems are discussed in Section 5.3.

In *screening*, schema changes generate a conversion program that is independently capable of converting objects into the new representation. The coercion is not immediate, but rather is delayed until an instance of the modified schema is accessed. That is, object access is monitored by the system, and whenever an outdated object is accessed, the system invokes the conversion program to coerce the object into the newer definition. Conversion programs resulting from multiple independent changes to a type are composed, meaning access to an object may invoke the execution of multiple conversion programs where each one handles a particular change to the schema. Screening causes delays during access to objects.

In *conversion*, each schema change initiates an immediate conversion of all objects affected by the change. In contrast to screening, this approach causes delays during the modification of schema, but no delays are incurred during access to objects.

A second solution for handling change consistency of instances is to introduce a new version of the schema with every modification and to supplement each schema version with additional definitions that handle the semantic differences between versions. These additional definitions are known as *filters* and the technique is called *filtering*. Error handlers are one example of *filters*. They can be defined on each version of the schema to trap inconsistent access and produce error and warning messages. The Encore model [SZ86, SZ87] uses type versioning with error handlers as a filtering mechanism.

In the filtering approach, changes are never propagated to the instances. Instead, objects become instances of particular versions of the schema. When the schema is changed, the old objects remain with the old version of the schema and new objects are created as instances of the new schema. The filters define the consistency between the old and new versions of

schema and handle the problems associated with behaviors written according to one version accessing objects of a different version. For example, if a behavior is dropped from a type, then a filter can be defined on the new version of the schema that produces a default value if a behavior written according to the old version applies the dropped behavior to an object created according to the new version.

A hybrid model combines two or more of the above methods. For example, a system could use filtering as the underlying mechanism and allow explicit coercion to newer versions of types, either through screening or conversion. Another example is a system that takes a more active role by using screening as the default and switching to conversion whenever the system is idle.

## 5.2 Issues of Version Control

Version control is the ability to manage different versions of objects. Usually, this is a selective feature that may be set to only track versions of certain objects. In a uniform model like TIGUKAT, where everything is an object, all forms of information are candidates for versioning. The selectivity of versioning in TIGUKAT is based on the behaviors defined on types. Basically, the temporal behaviors defined on a type are the aspects of all instances of that type that are versioned over time. The non-temporal behaviors are not versioned. Thus, entire objects are not versioned in TIGUKAT, but only the components relating to temporal behaviors.

Several approaches to versioning have been identified and explored. These include the *versions of objects* (VOO), *versions of types* (VOT), *versions of schema* (VOS), and *views of schema* (WOS) approaches.

In the versions of objects approach, it is the individual objects that are versioned. This approach has been explored in the context of models that do not carry uniformity to the extent that TIGUKAT does. Thus, the schema in these models are not objects and are not versioned.

The inability to version the schema means that objects that existed before a schema change are irreversibly modified when updated to coincide with the new schema. This shortfall has led to the development of techniques for versioning individual types (or classes) [SZ86] and a broader approach of versioning the entire schema [KC88]. The former manages schema changes on a per type basis, while the latter treats the entire schema as an object that is versioned.

In the views of schema approach, there is a single underlying schema and objects are instances of this schema. Any number of views can be defined on the schema and a schema view defines the visibility of objects and their properties under that view.

The version control mechanism described in this chapter introduces another way of managing versions called the *versioned behaviors* (VDB) approach. This approach stems directly from the temporality of the object model in that temporal objects are exactly the versioned objects. An object is temporal if its type defines at least one temporal behavior. Temporal and non-temporal behaviors are primitive elements of the temporal model. One advantage of this approach is that entire objects are not versioned – only the components defined by the temporal behaviors are versioned. Another advantage is that temporality is selective on a behavioral basis. This means temporality can be turned on or off for behaviors by defining the appropriate temporal or non-temporal behaviors, respectively. Furthermore, objects can be coerced to newer versions of the schema one behavior at a time. This means

that different temporal behaviors of an object can correspond to different versions of the schema. This provides great flexibility in managing versions.

With the VDB approach, objects are instances of a single type. This is in contrast to the VOT and VOS approaches where objects are instances of a version of a type. Using VDB, subtype relationships between types can be modeled over time by defining behaviors *B_subtypes* and *B_supertypes* as being temporal behaviors. Now, at a given time of interest[1], the sub/supertypes of all types can be found, and by combining these results, a version of the entire schema can be constructed at that time of interest.

Since everything is uniformly an object in TIGUKAT, the VDB approach is similar to VOO with schema support, but differs in that entire objects are not versioned – only the temporal behaviors of objects are versioned. By defining temporal behaviors on type objects, VOT is supported, and by specifying a particular time of interest, a version of the schema can be generated and, thus, VOS is supported as well.

## 5.3 Related Work

In recent years, several researchers have addressed the problem of defining schema evolution policies and version control for OBMSs. Some systems are described below in relation to the concepts introduced in the previous section.

The Orion [BKKK87, KC88] model is the first system to introduce the invariants and rules approach as a more structured way of describing schema evolution in OBMSs. Orion defines a complete set of invariants and a set of twelve accompanying rules for maintaining the invariants over schema changes. The allowed schema changes are classified into several categories, each of which affects different parts of the schema. These changes represent the typical schema modifications allowed in most systems today. The changes supported in TIGUKAT are similar to those of Orion, but vary to deal with uniformity, which is not part of Orion. For example, stored properties and computed methods are separate concepts in Orion and need to be handled separately, while in TIGUKAT they are treated uniformly as behaviors and, therefore, a single mechanism suffices for both.

Schema evolution in GemStone [PS87] is similar to Orion in its definition of a number of invariants. The GemStone model is less complex than Orion in that multiple inheritance and explicit deletion of objects are not permitted. As a result, the schema evolution policies in GemStone are simpler and cleaner. For example, while Orion defines twelve rules for disambiguating the effects of schema modification, GemStone requires no such rules. It is now generally accepted that multiple inheritance is a necessity in advanced OBMSs and, therefore, is part of the TIGUKAT model and is considered in schema evolution. Explicit deletion is another operation that is typical in database systems. In TIGUKAT, deletion is addressed in the context of the temporal model extensions. The existence of an object in its class is managed by a behavior *B_lifespan* that returns the interval in which the object is valid. When an object is "deleted", it is not removed from the system. Instead, the lifespan of the object in its class is timestamped with the deletion time and this "effectively deletes" the object from subsequent time. Conversion is used in GemStone to propagate changes to the instances. Literature on GemStone mentions the possibility of a hybrid approach

[1] Note that the time reference used to specify a "time of interest" is determined by the structure of the temporal behaviors. This is flexible and could be an absolute time point, a relative time point, a version number, or some other relevant time reference. Only the generic "time of interest" reference is used in this thesis, but one may replace this with "version number" to bring additional meaning to the concepts introduced.

that allows both conversion and screening, but it is not clear if such a system has yet been developed. The emphasis of GemStone is to provide schema evolution without the use of versioning. Thus, version control is not part of the system.

Skarra and Zdonik [SZ86, SZ87] define a framework for versioning types in the Encore object model as a support mechanism for evolving type definitions. A generic type consists of a collection of individual versions of that type. This is known as the *version set* of the type. Every change to a type definition results in the generation of a new version of that type. Since a change to a type can also affect its subtypes because of specialization requirements, new versions of the subtypes may also need to be generated. By default, objects are bound to a specific type version and must be explicitly coerced to a newer version in order to be updated. Since objects are bound to a specific type version, a problem of missing information can arise if programs (i.e., methods) written according to one type version are applied to objects of a different version. For example, if a property is dropped from a type, programs written according to an older type version may no longer work on objects created with the newer version because the newer object is missing some information (i.e., the dropped property). Similarly, if a property is added to a type, programs written with the newer type version in mind may not work on older objects because of missing information. For this reason, type versions include additional definitions, called *handlers*, that manage the semantic differences between versions – such as the missing information problem. This approach is one of the first to address the issue of maintaining behavioral consistency between versions of types.

One result of Skarra and Zdonik's work is a design methodology for defining *handlers*. A *handler* is defined on a type version and specifies an "on condition" that traps read and write access to a particular property that is undefined or invalid in that particular type version, but is valid in the generic type. Furthermore, a handler defines an appropriate action to take if such an access occurs. Consider the missing information example above. A handler can be defined on the type version that is the missing property so that it returns a default value, a nil value, or simply generates an error. Using this approach, a handler can be defined for each semantic difference between type versions in order to filter object access and to trap any inconsistent accesses that may occur. This is the filtering approach to change propagation. A filtering approach is also used in TIGUKAT, but the temporality of the object model, instead of handlers, is used to manage behavioral consistency between versions.

Skarra and Zdonik go a long way towards maintaining the semantics of behaviors between different versions of types. However, it is clear that defining handlers on various type versions can become confusing and unmanageable in systems with a large number of types that change often. In response to this problem, a more fundamental approach that uses temporal behaviors to model versions of objects is proposed in this thesis. Since the TIGUKAT model is uniform, types are objects with well-defined behavior and by defining appropriate temporal behaviors of types (e.g., subtype and supertype relationship behaviors), types are naturally versioned in TIGUKAT. Versions of the schema extend naturally from this by simply specifying a particular time of interest and then using this time reference to index the correct versions of types. The temporal subtype and supertype relationship behaviors at the given time reference define the structure of the particular version of the schema at this time. Semantic consistency of behaviors between old and new versions of types is also supported in TIGUKAT. Instead of defining handlers on the various versions of types, pre-existence and post-existence implementations can be defined for the temporal behaviors on these types. These implementations can return, similar to Encore, a default

value, a nil value, or generate an error.

Nguyen and Rieu [NR89] discuss schema evolution in the Sherpa model and compare their work to Encore, Gemstone, Orion, and one of their earlier models for CAD systems called Cadb. The emphasis of this work is to provide equal support for evolving schema definitions and for propagating changes to instances. The schema changes allowed in Sherpa follow those of Orion. Schema changes are propagated to instances through conversion or screening, which is selected by the user. However, only the conversion approach is discussed. Change propagation is assisted by the notion of *relevant classes*. A *relevant class* is a semantically consistent partial definition of a complete class and is bound to the class. A relevant class is similar to a type version in [SZ86] and a complete class resembles a version set.

The properties of relevant classes are characterized automatically by selecting from the powerset of instance variables and constraints defined in a complete class definition. The selection is restricted to only those combinations that are meaningful with respect to certain semantic rules [NR87]. Objects are instances of exactly one relevant class, which characterizes a partial definition of that object. The purpose of relevant classes is to evaluate the side-effects of propagating schema changes to the instances and to guide this propagation.

Relationships between relevant classes can be characterized as a graph where the nodes are relevant classes and the edges are labeled with schema changes that take one relevant class definition to another. As the schema evolves, relevant classes are used to evaluate the changes and test their semantic consistency. Objects are migrated between relevant classes to effect the changes made to them. This migration is essentially object coercion. The propagation of objects within a set of relevant classes can have a large overhead, but it is argued that relevant classes group objects into smaller sub-classifications so that the number of objects affected by a change within a class is reduced, thereby increasing performance. This approach is valid in systems that consider partial definitions of objects within a class.

In the Farandole 2 model [ALP91], a structure called a *context* and the maintenance of versions within contexts are proposed as a basis for schema evolution and versioning. A *context* is a partial view of the overall schema that serves a dual purpose: it defines a subset of objects in the database, and a subset of operations that can be performed on these objects. Versions can be derived from the visible schema within a given context. Thus, a views of schema approach is used to define contexts (views) and this is combined with a versions of schema approach for each context to define versions of the schema within the scope of a given context. Thus, the approach is close to managing versions of views. A global database schema can be derived from the set of all contexts. The typical schema changes are allowed. A context is represented by a connected graph where the nodes are classes and the edges are attributes denoting relationships between classes. Thus, contexts are similar to entity-relationship diagrams. Schema changes are characterized into graph operations and rules for maintaining graph integrity are defined.

Elements of versions and contexts can be shared by other versions and contexts. Thus, objects must maintain information about the contexts and versions in which they participate. One must consider the amount of extra space needed to store this information in the objects rather than the types. The focus of the work is on managing changes to schema and no propagation technique is explicitly stated, although it seems that conversion or screening could be used. There is a brief discussion on how the model improves independence between programs and changing schema, which suggests a filtering approach, but it is unclear how the model achieves this feature. Like relevant classes, it is argued that a context provides a smaller group of objects that need to be modified as a result of schema changes, which is

intended to improve performance.

Osborn [Osb89] describes an algebra that utilizes inclusion polymorphism to define equivalence of queries on different versions of schema. The work does not describe how schema changes are propagated to the instances. Two kinds of schema modifications are considered. The first involves changing simple atomic attributes like strings and integers to more complex aggregates of these simple types (the opposite direction of changing aggregates to simple types is also discussed). Only one level of aggregation is considered. That is, the aggregation of aggregate types is not discussed.

The second modification considered is that of specializing aggregate types (the opposite direction of generalizing aggregate types is also discussed). Several example queries using strings and integers are presented. The schema is modified by specializing previous types and it is shown how the equivalence of queries are preserved (or not preserved) through polymorphism. The results are interesting, but the full scope of schema evolution is not considered.

In OTGen [LH90], the focus shifts from dynamic schema evolution to database reorganization. The invariants and rules approach is used, and the typical schema changes are allowed. The invariants are used to define default transformations for each schema change. Schema changes produce a transformation table that describes how to modify affected instances. Multiple schema changes are usually grouped and released as a package called a *transformer*. Screening is used to apply the transformer and propagate changes to the instances. Multiple releases are composed and, thus, access to an older object can invoke multiple transformers to bring the object up to date. One result of the database reorganization approach is that multiple changes are packaged into a single release and this is expected to reduce the number of screening operations that need to be invoked for each object access. Another result is that transformers are represented as tables that are initialized by OTGen. A simple language is provided to describe transformations. Before releasing a transformer, a database administrator can edit the entries in the table to override the default transformations. Each release can be thought of as a separate version of the entire database. Thus, this is similar to the versions of schema approach. Since the focus of the paper is on database reorganization, the details of invoking and accessing individual versions is not discussed.

Reiter [Rei92] discusses a formal approach to defining database updates using techniques from artificial intelligence. A *situational calculus*[2] for a transaction model is defined and a solution to the *frame problem*[3] within this model is described. This requires the introduction of second-order operations and details are not given. In a uniform model like TIGUKAT, the schema is part of the objectbase and thus can be part of updates to the objectbase. It seems likely that Reiter's formal model could be adapted to describe schema evolution in a uniform model. A form of versioning is already part of his model since he describes how a transaction modifies a database within a particular state taking it to a new state. Thus, old states are preserved and each state is like a version of the database. This approach is appealing because it moves from the traditional procedural treatment of updates to a declarative one.

In the systems discussed above, if an object is coerced to coincide with a new definition

[2]The *situational calculus* [McC68] is a first order language designed to represent dynamically changing worlds in which changes are the result of applying named *actions* within a particular state taking the world to a new state.

[3]The *frame problem* stems from the need for specifying the *invariants* of a particular action or update within a world of which there are usually a large number.

of the schema, the entire object must be converted. In systems that don't define versioning, the old state of the object is lost. The approach in TIGUKAT differs in that the granularity of object coercion is based on individual behaviors. That is, individual behaviors defined on the type of an object can be coerced to a new definition for that object, leaving the other behaviors to retain their old definitions. Furthermore, a historical record of the coerced behaviors is maintained for each object so that older definitions of the behaviors can still be accessed for each object. Complete object coercion can be done by explicitly coercing all the behaviors of an object.

Substantial research has been ongoing in the past decade to support the notion of time in various systems [Soo91, TCG+93]. Time has been introduced recently in the context of object models [RS91, KS92, DW92, WD92]. These studies have concentrated on extending the object model to facilitate various notions of time. Furthermore, query models have been extended by adding new operators and constructs that range over time values and allow for the execution of queries on temporal and non-temporal objects to be carried out in a uniform manner. Using time to model schema evolution in an object model has not received much attention. Given the application domains that TIGUKAT is expected to support, temporal extensions to the TIGUKAT object model have been introduced [GÖ93]. In this thesis, it is show how time is used to model temporal behaviors, which is turn models versions of objects, types, and schema.

## 5.4 Overview of Schema Evolution and Versioning

In this thesis, a linear model of time is proposed as a foundation for managing schema evolution and version control. Temporality is based on behaviors and is consistently extended to include schema information like types, plus all forms of objects as well. Since temporality is behavior based, an object is temporal if and only if it's type defines at least one temporal behavior. Otherwise, the object is non-temporal. Therefore, temporal and non-temporal objects co-exist in the model. Temporal behaviors are a specialization of the primitive, non-temporal behaviors. Thus, temporality is transparent in the model (i.e., if the user is not concerned with temporality, then the temporal behaviors act just as regular, non-temporal behaviors do).

Temporal behaviors manage histories of changes to objects and therefore a version of a temporal object can be constructed at any time of interest by indexing into these histories. By defining appropriate temporal behaviors on the meta-architecture, versions of types and versions of schema are supported. That is, changes to the schema involve updating the history of certain behaviors. For example, adding a new behavior to a type changes the history of the type's interface to include the new behavior. The old interface of the type is maintained and can be accessed through temporal language features that allow behavior applications to be qualified by a time reference point. One need only specify a time reference in the past when applying the *B_interface* behavior to get an older version of the interface of a given type. This is effectively versions of types. Similarly, the subtype relationship behavior is defined to be temporal and, therefore, the structure of the type lattice can be reconstructed at any time of interest. This is effectively versions of schema.

Coercion of objects to a newer version of a type is optional in TIGUKAT. Since different versions of types are maintained through temporality, all the schema information of older objects is available and can be used to continue processing these objects in the old way. If coercion is desired, the entire object does not need to be updated. Objects can be coerced

to a newer version one behavior at a time. This means that some behaviors of the object may work with newer versions, while others may work with older ones. This is in contrast to other models where an object is converted in its entirety to a newer version, thereby losing the old information of the object. Since the old information of the object is available, even if objects are coerced to a newer version, historical queries can be run by giving an appropriate time point in the past history of the object.

Even though this work is within the context of the TIGUKAT object model, the results reported here extend to any system that uses time to model histories of behaviors. Currently, we are unaware of any other systems that use this approach.

## 5.5 Temporality of the Object Model

Most of the applications that OBMSs are expected to support exhibit some form of temporality. Some examples are the following: in engineering databases, there is a need to identify different versions of a design as it evolves; in multimedia systems, the video images are timed and synchronized with audio; in office information systems, documents are ordered based on their temporal relationships. Thus, a temporal domain is a very natural part of an OBMS and in many cases simplifies advanced management facilities such as schema evolution and version control.

Temporality has been introduced into the TIGUKAT object model [GÖ93]. A brief overview is presented in this section to establish the foundation for using time to manage schema evolution and versioning.

Time is added to TIGUKAT by extending the base model with time-related types and behaviors. Figure 5.1 shows the types added by the temporal extensions. Some of the time-related behaviors defined on these types are discussed below.

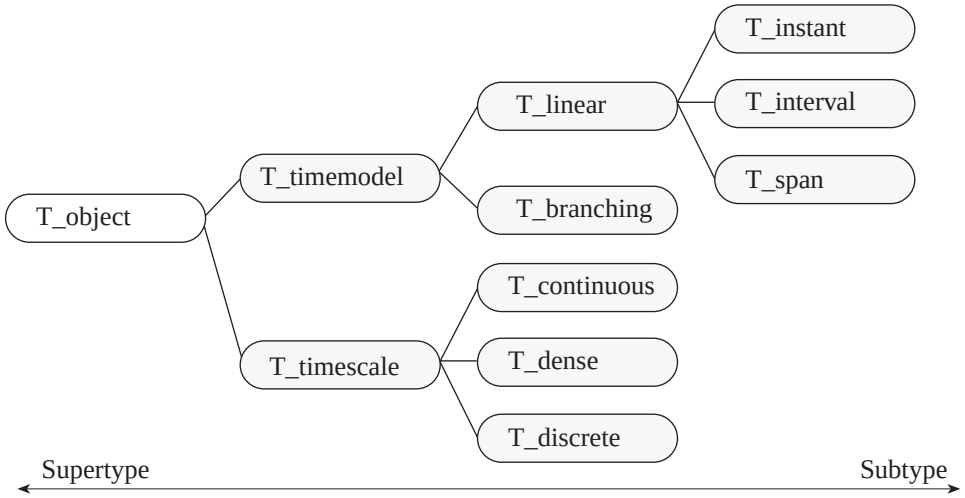



Figure 5.1: The abstract time types.

Two aspects of modeling time are considered: the *structural models* of time and the *density* of these models. Two structural models are represented in TIGUKAT. The first is a *linear* model where time flows from past to future in a totally ordered manner. The second is a *branching* model where time flows linearly until a certain point where it can *branch* into several independent, parallel linear models that can go on branching indefinitely. The

structure of a branching model is a directed tree with the root being the start of time, the leaves being the current time at the various branches, the nodes being the branch points, and the edges being linear models that connect nodes. The type `T_timemodel` represents structural models in general and the types `T_linear` and `T_branching` represent the two specific structural models considered in TIGUKAT. Linear models are further specialized into instantaneous models (`T_instant`) consisting of a single time point (e.g., 32, $t_4$), interval models (`T_interval`) consisting of specific lower and upper bound time points (e.g., [2-16], $[t_4 - t_9)$), and spanning models (`T_span`) that consist of durations (e.g., 4 days, 2 months, annually, quarterly).

The density of a structural model defines the domain over which time is perceived or referenced in that model. In other words, it defines a scale for time in the model. Three basic *scales* (i.e., *domains*) of time are considered in TIGUKAT. The general density of time models is represented by the type `T_timescale`. The subtypes `T_continuous`, `T_dense`, and `T_discrete` represent the three basic time scales. *Discrete* domains map time to the set of integers, *dense* domains map time to the set of rational numbers, and *continuous* domains map time to the set of real numbers.

For the purpose of developing schema evolution and versioning, this thesis concentrates on the `T_interval` and `T_discrete` types, which suffice for its design.

Since temporality is integrated with the base object model, it can be extended. Additional structural models and densities can be easily introduced by building on the established types. This is a direct result of the uniformity of the model. For example, to model *dates*, a type `T_date` can be defined as a subtype of `T_instant`. To model years, months, or day spans (i.e., durations), appropriate subtypes of `T_span` can be created. These can be further subtyped to model a finer granularity of time.

To manage temporal information about the behaviors of objects, the type `T_temporalBhv` is introduced as a subtype of `T_behavior`. This type defines additional functionality for representing the semantics of temporality on behaviors. An instance of `T_temporalBhv` is called a *temporal behavior*. Temporal behaviors are prefixed by *BT_*. The associated class **C_temporalBhv** is introduced to manage the temporal behavior instances.

The additional functionality of `T_temporalBhv` allows its instances to maintain a history of updates with respect to objects they are applicable to. The history of updates is modeled by the *B_history* behavior defined on `T_temporalBhv`[4]. The signature is *B_history* is as follows:

$$B\_history\text{: } \texttt{T\_object} \rightarrow \texttt{T\_collection}\langle \texttt{T\_timemodel}, \texttt{T\_object} \rangle$$

*B_history* requires a temporal behavior as the receiver. It accepts an object as an argument, and returns a collection of <`T_timemodel`, `T_object`> pairs as a result. The result represents the history of the receiver behavior with respect to the given argument object. If the receiver behavior is not defined on the type of the argument object, an error condition is raised. For example, assume theArctic is an instance of `T_land` and *B_value* is defined as a temporal behavior (denoted *BT_value*). The behavior application theArctic.*BT_value* returns the current value of the land and *BT_value*.*B_history*(theArctic) returns the entire history of the land value as it has changed over time.

The following definitions formally establish the semantics of temporal and non-temporal objects according to behaviors and types.

[4]Note that *B_history* is an instance of `T_behavior` and not `T_temporalBhv`.

**Definition 5.1** *Temporality of Behaviors:* A behavior $b$ is temporal if and only if it is an instance of `T_temporalBhv` (i.e., $b \in$ **C_temporalBhv**).

**Definition 5.2** *Temporality of Types:* A type $t$ is temporal if and only if it defines at least one behavior in its interface. That is, the following condition is met:

$$\exists b \mid b \in t.B\_interface \wedge b \in \textbf{C\_temporalBhv}$$

**Definition 5.3** *Temporality of Objects:* An object $o$ is temporal if and only if the type of $o$ (i.e., $o.B\_mapsto$) is temporal.

From these definitions it is clear that, in TIGUKAT, temporality of objects is not orthogonal to their type. In other words, if a type is temporal, then all of its instances are temporal, and if a type is non-temporal, then none of its instances are temporal. This approach is reasonable since certain aspects (i.e., behaviors) of a similar group of objects (i.e., of a particular type) are usually temporally maintained. For example, to track the value of land zones (i.e, objects of type `T_land`), *B_value* would be defined as a temporal behavior. According to the definitions, this means that a value history would be kept for each land zone. This is reasonable since a land value history is something that would typically be tracked for all units of land or for none.

To demonstrate the notion of timestamping objects, the type `T_DiscInterval` is introduced as a subtype of `T_interval`. The behaviors of `T_DiscInterval` are specialized by fixing the time scale to be discrete. In the following discussion, the term *interval* is used to mean an instance of **C_DiscInterval**. Intervals are represented as pairs of the form $[l, u)$ where $l$ and $u$ are time instants that denote the *lower* and *upper* bounds of the interval, respectively. An interval is closed on $l$ and open on $u$. Occasionally, such as with history termination, an interval will be closed on both ends in which case it is represented as $[l, u]$. The interval [ ] denotes the empty interval and can be used in time comparison operations. The time instant *now* is introduced as the marking symbol for the current time. An interval whose upper bound is *now* expands as the clock ticks. The specification of particular units of time is left to the user or application. This is flexible and could be given by specific time points, relative time points, version numbers, and so on.

The time model component of the $<$`T_timemodel`, `T_object`$>$ pairs is assumed to be the *interval* in which the object is valid. Consequently, the history of temporal behaviors is represented by sets of pairs of the form $<[l, u), o>$ where $[l, u)$ is an interval as described above and $o$ is the object that is valid (or exists) over the given interval. The interval serves as a timestamp for the validity of object $o$.

Now, the result of *B_history* is a collection of $<$`T_DiscInterval`, `T_object`$>$ pairs (that is, `T_collection`$\langle$`T_DiscInterval`, `T_object`$\rangle$). In other words, the result collection consists of objects whose type is `T_DiscInterval` $\times$ `T_object`. This type is automatically created as a subtype of `T_product` (see Chapter 3) and thereby inherits all its native behaviors. Recall that the inject behavior ($\rho_i$) of `T_product` returns the $i^{th}$ component of a product object. Hence, if $e$ is an element from a history collection, then $e.\rho_1$ returns the `T_DiscInterval` component of $e$ and $e.\rho_2$ returns `T_object` component.

Another important behavior introduced by the temporal extensions is the *B_lifespan* behavior defined on `T_object`. The signature of *B_lifespan* in the context of the model discussed above is as follows:

$$B\_lifespan : \texttt{T\_collection} \rightarrow \texttt{T\_DiscInterval}$$

This behavior is applied to an object, accepts a collection as an argument, and returns a discrete interval representing the time in which the object exists in the given collection. For example, the following behavior application returns the lifespan of the object theArctic in class **C_land**:

$$\text{theArctic}.B\_lifespan(\textbf{C\_land})$$

Rules are defined in [GÖ93] to ensure the semantic consistency of lifespans in the context of classes and inclusion polymorphism. For example, the lifespan of an object in a class is contained within the lifespan of that object in any superclass. That is, if an object ceases to exist in a certain class, then it must also cease to exist in the subclasses. This is reasonable since, for example, if a certain house is demolished and ceases to be a dwelling, then it should also cease to be a house.

An object is effectively deleted from a collection (or class) by timestamping its lifespan in that collection with the current time. Objects that currently exist in a collection have the upper bound of their lifespan interval set to *now*.

A noteworthy point is the *temporal transparency* built into the model. The distinction between temporal and non-temporal behaviors is based on type[5]. The specification of a signature for a temporal behavior and its application to objects is no different from a non-temporal one. This is important from the user's perspective since the utilization of temporal and non-temporal behaviors is transparent. The history of a temporal behavior with respect to a certain object can be retrieved by applying the *B_history* behavior to it.

Two basic aspects of time are considered in databases that incorporate temporality. These are the *valid* and *transaction* times. The former denotes the time when an object becomes effective (begins to model reality), while the latter represents the time when a transaction was posted to the database. The need to distinguish between valid and transaction times arises when an update to an object is posted to the database at a time that is different from the time when the update becomes valid. In this work, only valid times are considered, however, the concepts introduced also apply to transaction times and can easily be carried through to them as well.

## 5.6 Semantics of Schema Evolution

### 5.6.1 Definition of Schema

There are different kinds of objects modeled by TIGUKAT, some of which are classified as schema objects. All objects managed by TIGUKAT can be placed into one of the following categories: *type, class, behavior, function, collection* or *other*. These characterizations are used to define the "schema" of the model and the changes that affect the schema. First, the definition of what constitutes schema objects is proposed. This is followed by the definition of the "schema."

**Definition 5.4** *Schema Objects:* The following classifications of schema objects are primitive to the model:

- The class **C_type** forms the collection of type schema objects denoted $TSO$.

[5]The prefix *BT_* was introduced to denote temporal behaviors, but this is only a notational convenience to improve readability and could be dropped.

- For all types $t \in TSO$, the extended union over the behavior application $t.B\_interface$, that is:

$$\bigcup t.B\_interface$$

  forms the collection of behavior schema objects denoted $BSO$. Only those behaviors defined in the interface of some type are considered to be behavior schema objects. Note that $BSO \subseteq$ **C_behavior**.

- For all behaviors $b \in BSO$, for all types $t \in TSO$, the extended union over the behavior application $b.B\_implementation(t)$, that is:

$$\bigcup b.B\_implementation(t)$$

  forms the collection of function schema objects denoted $FSO$. Only those functions defined as the implementation of some behavior for some type are considered to be function schema objects. Note that $FSO \subseteq$ **C_function**.

- The class **C_collection** forms the collection of collection schema objects denoted $LSO$.

- The class **C_class** forms the collection of class schema objects denoted $CSO$. Note that $CSO \subseteq LSO$.

**Definition 5.5** *Schema:* The *schema* of a TIGUKAT objectbase is equivalent to the union of all schema object collections. That is:

$$schema \equiv TSO \cup BSO \cup FSO \cup LSO \cup CSO$$

Note that $CSO$ is included for completeness. It is unnecessary since $CSO$ is a subset of $LSO$.

There are three basic operations that can be performed on objects: *add, drop* and *modify*. In the context of the temporal model, *adding* refers to creating an object and beginning its lifespan in its class, *dropping* refers to terminating the lifespan of an object in its class, and *modifying* refers to updating the object, which in turn leads to versioning the temporal aspects (i.e., temporal behaviors) of the object.

Table 5.1 shows the combinations between the various object categories and the different kinds of operations that can be performed. The **bold** entries represent combinations that implicate schema evolution modifications, while the *emphasized* entries denote other changes that are not considered to be part of the schema evolution problem.

For the purpose of performing the operations in the *Drop* (D) column of Table 5.1, a generic drop behavior *B_drop* is added to type `T_object`. The signature of *B_drop* is as follows:

$$B\_drop : \texttt{T\_object}$$

The implementation of the behavior is redefined in the types of the various schema objects affected by the operation. The details of each refinement are given in the sections that follow.

Before considering each schema change in turn, the invariants of schema evolution that must be maintained over schema modifications are presented.

| | Operation | | |
|---|---|---|---|
| Objects | Add (A) | Drop (D) | Modify (M) |
| Type (T) | **subtyping** | **type deletion** | **add behavior**(AB)<br>**drop behavior**(DB)<br>**add supertype link**(ASL)<br>**drop supertype link**(DSL) |
| Class (C) | **class creation** | **class deletion** | *extent change* |
| Behavior (B) | *behavior definition* | **behavior deletion** | **change association**(CA) |
| Function (F) | *function definition* | **function deletion** | *implementation change* |
| Collection (L) | **collection creation** | **collection deletion** | *extent change* |
| Other (O) | *instance creation* | *instance deletion* | *instance update* |

Table 5.1: Classification of schema changes.

### 5.6.2 Invariants of Schema

The following invariants have been identified for maintaining the semantics of schema modifications in TIGUKAT. The invariants are used to gauge the consistency of a schema change in that the invariants must be satisfied both before and after a schema change is performed. The **type lattice**, **full inheritance**, **domain compatibility**, and **distinct behavior** invariants are similar to those presented in other models such as Orion and GemStone. The **full implementation** and **direct supertype** invariants are unique to the design of this approach, and the **temporal** invariant is required due to the introduction of temporality into the model.

**Type Lattice Invariant:** *The type lattice is a connected, directed acyclic graph (DAG).* The nodes of the lattice are types and the directed edges are subtype relationships with tail of the edge being the subtype of the type pointed to by the head. The lattice has the single system defined type `T_object` as its root and the system defined type `T_null` is its base. Since the lattice is connected, there are no isolated types and all types are a subtype of the root `T_object`.

A *chain* in the type lattice is a collection of types, totally ordered by subtyping, such that they form a single connected path through the lattice. A chain is identified as a collection of types that are connected by sub/supertypes relationships such that they form a connected path through the lattice. A chain of length one from a type `T_a` to a supertype `T_b` is called a *direct supertype link* from `T_a` to `T_b` or a *direct subtype link* from `T_b` to `T_a`. For example, in Figure 2.2 the types {`T_pond`, `T_water`, `T_zone`} form a chain and so do {`T_land`, `T_zone`} which is a direct supertype link from `T_land` to `T_zone`. A single type such as {`T_map`} forms a chain of length zero. On the other hand, {`T_map`, `T_land`, `T_zone`} does not form a chain because `T_map` and `T_land` are not in a sub/supertype relationship with one another. As well, {`T_forest`, `T_zone`} is not a chain because its connectivity is broken by the exclusion of type `T_land`.

**Full Inheritance Invariant:** *A type inherits all behaviors defined by its supertypes.* The behaviors inherited by a type are called the *inherited behaviors* of the type. A type can define additional behaviors that are not part of its inherited behavior set. These are called the *native behaviors* of the type. The union of the inherited and native behaviors is called the *interface* of the type. A type's interface is a superset of the union of interfaces of its supertypes.

**Full Implementation Invariant:** *A type that has an associated class must have functions associated with all its behaviors.* If a type has an associated class, then objects of that type may already exist and new objects can be created. In order for these objects to have their full meaning, all behaviors defined on the type must have functions (i.e., implementations) associated with them.

**Direct Supertype Invariant:** *A direct supertype link between two types is the only chain linking the types.* If another chain links the types, the direct supertype link is dropped. This means that for any two types, say `T_a` and `T_b`, if there exists a chain from `T_a` to `T_b` greater than length one, then there are no direct supertype links (i.e., chains of length one) from `T_a` to `T_b`. Furthermore, this implies that there is at most one direct supertype link between any two types.

**Domain Compatibility Invariant:** *The result type of a behavior in a type must generalize the result type of that behavior in all subtypes.* That is, the result type of a behavior defined on a type, say `T_a`, must generalize the result type of that behavior in all subtypes of `T_a`. Note that the result types may be the same. This invariant ensures substitutability.

**Distinct Behavior Invariant:** *The behaviors in the interface of a type are unique.* That is, the semantics of the behaviors must be unique. Since a name is part of a behavior's semantics, the names of behaviors in the interface of a type must be unique.

**Temporal Invariant:** *The behaviors defined in the interface of a type at a given time are applicable to all instances of that type that exist at that time.* That is, if a behavior exists in the interface of a type at a given time $t$, and $t$ is within the lifespan of an object of that type, then the behavior is applicable to the object. The temporal invariant is managed automatically by temporal model through the timestamping of temporal behaviors.

### 5.6.3 Semantics of Change

In this section the modifications that affect the schema (i.e., the bold entries of Table 5.1) are described. The basic operations affecting the schema include adding behaviors to a type definition, dropping behaviors from a type definition, changing the implementation of a behavior in a type, and adding and dropping classes. The other schema changes, namely, adding and dropping types, adding and dropping supertype links, dropping behaviors and dropping functions can be defined in terms of the type-related basic operations.

Type modifications are separated into changes affecting the behaviors defined on a type and changes affecting the relationships between types such as adding and dropping direct supertype links. The semantics of these changes are discussed in the following sections.

#### Modify Type - Add Behavior (MT-AB)

This operation adds a native behavior to a type. In order to satisfy the distinct behavior invariant, the operation is rejected if the behavior is already defined on the type either natively or through inheritance. The full inheritance invariant requires that the added behavior is inherited by all subtypes of the type to which it is added. Behavior *B_addBehavior* defined on `T_type` performs this schema change. The signature of *B_addBehavior* is as follows:

$$B\_addBehavior : \texttt{T\_behavior} \rightarrow \texttt{T\_function} \rightarrow \texttt{T\_type}$$

*B_addBehavior* is applied to a type object and accepts a behavior and a function as arguments. The behavior argument is the behavior to add to the receiver type and the function argument is the implementation to associate with the behavior for that type. For example, the following behavior application adds a behavior *B_PHlevel* to the type `T_water` (see Table 2.1, page 21) and associates this behavior with a stored function:

$$\texttt{T\_water}.B\_addBehavior(B\_PHlevel, STORED)$$

The function argument may be omitted if the receiver type does not have an associated class and if all subtypes of the receiver type that have an associated class already define the behavior being added. This restriction is imposed to satisfy the full implementation invariant.

In order to satisfy the domain compatibility invariant, the result type of the behavior in the type to which it is added must generalize the result type of the behavior in all the subtypes of that type. All other invariants are satisfied.

### Modify Type - Drop Behavior (MT-DB)

This operation drops a native behavior from a type. The operation is rejected if the behavior is not defined on the type or if it is inherited by the type. Thus, only native behaviors can be dropped. Dropping an inherited behavior would mean that the behavior must also be dropped from all the supertypes, otherwise the behavior would be re-inherited because of the full inheritance invariant. With the restriction of only dropping native behaviors, the supertypes of a type retain all their original behaviors and are unaffected by the change.

Behavior *B_dropBehavior* defined on `T_type` performs this operation. The signature of *B_dropBehavior* is as follows:

$$B\_dropBehavior : \texttt{T\_behavior} \rightarrow \texttt{T\_type}$$

*B_dropBehavior* is applied to a type and accepts the behavior to be dropped as an argument.

When a native behavior is dropped, its native definition is propagated to all the subtypes, unless the behavior is inherited by the subtype through some other chain in which case the behavior will be inherited instead of native. With this approach, the interface of the subtypes retain all their original behaviors and only the single type directly involved in the operation actually drops the native behavior.

The reason for using this approach is that it is a fundamental approach in the sense that other forms of behavior dropping can be defined in terms of it. For example, in ORION the semantics of behavior dropping (i.e., attribute and method dropping in their model) is to recursively drop the behavior from all the subtypes as well. With the approach taken in TIGUKAT, a behavior can be defined (e.g., *B_dropBhvDeep*) that recursively performs *B_dropBehavior* on all the subtypes, which effectively drops the behavior from the subtypes (unless the behavior is inherited through some other chain). Other forms of behavior dropping can be defined in terms of the fundamental *B_dropBehavior*. An interesting approach would be to allow behaviors in a type to be flagged as being *semi-native* in the sense that they should not be dropped by a recursive decent drop process i.e., by *B_dropBhvDeep*), but instead should persist as native definitions in those types.

### Modify Type - Add Supertype Link (MT-ASL)

This operation effectively adds a subtyping relationship between two types. The addition of a type, say $S$, as a direct supertype of another type, say $T$ is rejected if (a) it introduces a cycle into the lattice, (b) $T$ is already linked to $S$ through some chain, or (c) there exists a behavior, say $b$, defined on both $S$ and $T$ and the result type of the behavior in $S$ does not generalize the result type in $T$. Behavior *B_addSupertype* defined on `T_type` performs this operation. The signature of *B_addSupertype* is as follows:

$$B\_addSupertype : \texttt{T\_type} \rightarrow \texttt{T\_type}$$

*B_addSupertype* is applied to a type and accepts a type as an argument. Its semantics is to add the argument type as a supertype of the receiver. To add $S$ as a supertype of $T$ we apply $T.B\_addSupertype(S)$. The behaviors of $S$ are inherited by $T$ and all the subtypes of $T$. This is equivalent to propagating the inheritance of added behaviors defined above and follows all the rules established for that operation.

### Modify Type - Drop Supertype Link (MT-DSL)

This operation drops a direct supertype link between two types. A direct supertype link to `T_object` cannot be dropped. Behavior *B_dropSupertype* defined on `T_type` performs this operation. The signature of *B_dropSupertype* is as follows:

$$B\_dropSupertype : \texttt{T\_type} \rightarrow \texttt{T\_type}$$

The receiver of *B_dropSupertype* is a type and a direct supertype of the receiver is passed as an argument. The semantics of this operation is to drop the direct supertype link between the receiver and the argument, reestablish links between the receiver and the supertypes of the argument, and reestablish links between the subtypes of the receiver and the argument.

Formally, let $R_i$ be the state of the receiver type before the change and $R_j$ be its state after the supertype link has been dropped. Similarly, let $T_i$ and $T_j$ be the before and after states of other general types. Furthermore, let $A$ denote the argument type (the before and after states are not important for the argument). The following super-lattice properties hold as a result of dropping a direct supertype link[6]:

$$R_j.B\_super\text{–}lattice = R_i.B\_super\text{–}lattice - \{A\}$$
$$\forall T_i \in (R_i.B\_super\text{–}lattice - \{R_i\}),\ T_j.B\_super\text{–}lattice = T_i.B\_super\text{–}lattice$$
$$\forall T_i \in (R_i.B\_sub\text{–}lattice - \{R_i\}),\ T_j.B\_super\text{–}lattice = T_i.B\_super\text{–}lattice$$

The semantics of this operation is clarified by the example lattice shown in Figure 5.2. Assume that the direct supertype link from $T$ to $S$ is to be removed. The behavior application $T.B\_dropSupertype(S)$ removes the direct supertype link between $T$ and $S$ and modifies the type lattice in the following way:

[6]The corresponding sub-lattice, supertypes and subtypes properties are also updated accordingly. Only the super-lattice properties are given, but these are sufficient for describing the effects of the schema change.

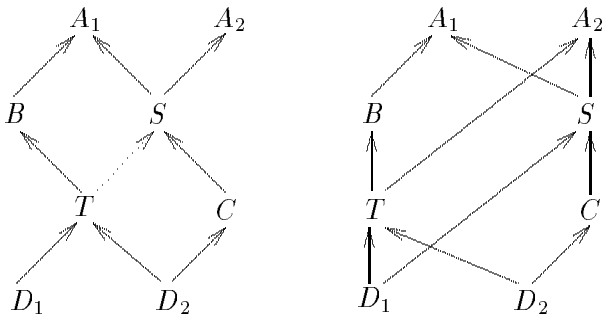



Figure 5.2: Effects of dropping a direct supertype link from type $T$ to type $S$.

- It adds a supertype link from $T$ to every supertype of $S$, unless $T$ is linked to the supertype(s) through another chain. In Figure 5.2, $T$ is re-linked to $A_2$, but not to $A_1$ since $T$ is already linked to $A_1$ through the chain containing $B$. This ensures that the interface of $T$ does not change by more than the native behaviors defined on $S$.
- It adds a supertype link from each subtype of $T$ to $S$, unless the subtype is linked to $S$ through another chain. In Figure 5.2, $D_1$ is re-linked to $S$, but $D_2$ is not since it is already linked to $S$ through the chain containing $C$. This ensures that the interface of $T$'s subtypes are not affected by the change.
- It drops the native behaviors of $S$ from the interface of $T$. These behaviors are not dropped from the subtypes of $T$ because the subtypes are re-linked to $S$ by the step above and therefore inherit its behaviors. Furthermore, the behaviors inherited by $S$ are not dropped from $T$ because $T$ was re-linked to the supertypes of $S$ and therefore inherits these behaviors.

With this approach, only the interface of $T$ is affected by losing the native behaviors of $S$. The interfaces of all other types remain unchanged.

The remaining type-related operations of adding and dropping types are discussed in the following sections. Since the temporal model is used for the dropping operation, types are not actually deleted. Instead, the lifespan of a dropped type in the class **C_type** is timestamped with the current time. This "effectively deletes" the type from subsequent time.

### Add Type (AT)

This operation creates a new type and integrates it with the existing lattice. Creating a type adds it to $TSO$, which is turn adds it to the schema. Type creation is supported through regular *subtyping* which is an operation provided by the primitive model.

Chapter 2 describes the behavior *B_new* as part of the meta-system and how it can be applied to the system supplied class **C_type** to create a new type. The *B_new* behavior accepts a collection of types as the first argument and a collection of behaviors as the second one. The result of applying the behavior is that a new type is created as a subtype of the types in the first argument collection and the behaviors in the second argument collection

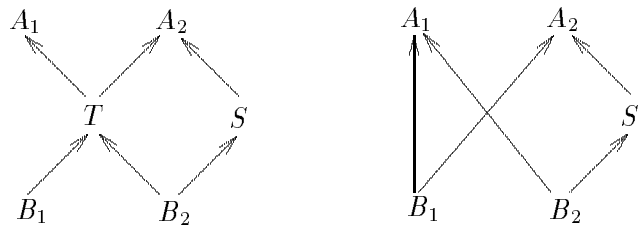



Figure 5.3: Effects of dropping a type $T$.

are defined natively on the new type, unless they are inherited from one of the argument types.

Subtyping (and thus the AT operation) can be defined in terms of creating a new type with the given behaviors defined natively, adding appropriate supertype links from the type to each argument type (which will update the native definitions appropriately), followed by adding a supertype link from `T_null` to the new type.

### Drop Type (DT)

This operation drops a given type, removing it from $TSO$ and, therefore, removing it from the schema as well. Dropping a type from the lattice terminates the lifespan of the type in the class **C_type**. This effectively deletes the type from subsequent time. The general *B_drop* behavior defined on `T_object` is refined in type `T_type` to perform type dropping.

The primitive types of the model (i.e., those in the primitive type system $\mathcal{T}$) cannot be dropped. When a type is dropped, the type's associated class and all the instances in the shallow extent of the class are dropped as well. If object migration techniques were introduced into the model, the instances could be ported to some other type prior to being dropped in order to preserve their existence. Object migration is outside the scope of this thesis.

Every direct subtype $B_j$ of a dropped type $T$ is re-linked to every direct supertype $A_i$ of $T$ unless there is a chain from $B_j$ to $A_i$ that does not include $T$. Furthermore, the native behaviors of $T$ are propagated to the direct subtypes so that they become native in the subtypes unless the behavior is inherited through some other chain. For example, Figure 5.3 shows the effect of dropping a type $T$. The subtype $B_1$ is re-linked to both supertypes $A_1$ and $A_2$, while $B_2$ is re-linked to $A_1$ but not $A_2$ because it is already linked to $A_2$ through the chain that includes $S$. If $T$ and $S$ both define a native behavior $b$, then a native definition of $b$ would be propagated to $B_1$, but not to $B_2$ because $B_2$ inherits the behavior from $S$.

The implementation of *B_drop* can be defined in terms of other operations. For example, to drop the type $T$, the following sequence of operations can be performed:

1. Drop supertype links from each subtype $B_j$ to $T$ (i.e., apply $B_j$.*B_dropSupertype*($T$)).
2. Add supertype links from each $B_j$ to each supertype $A_i$ of $T$ – if not already linked through some other chain (i.e., apply $B_j$.*B_addSupertype*($A_i$)).
3. Drop supertype links from $T$ to each $A_i$ (i.e., apply $T$.*B_dropSupertype*($A_i$)).

4. Effectively delete the type $T$, its associated class $C$, and all instances in the shallow extent of $C$ by timestamping their lifespan in the appropriate class.

Using this approach, dropping a type does not affect the interface of any other type and the operation is uniform in the sense that a series of type drops will produce the same resulting lattice regardless of the order in which the types are dropped. In contrast to this approach, Orion only links a subtype of a dropped type to the supertypes if the subtype becomes isolated. As a result, a series of type drops may produce a different resulting lattice depending on the order in which the types are dropped. For example, in Figure 5.3 consider dropping $T$ followed by dropping $S$ as opposed to first dropping $S$ and then dropping $T$. In Orion, the resulting lattices are different in both cases. In the first case, $B_1$ has supertype links to both $A_1$ and $A_2$ while $B_2$ is linked only to $A_2$. In the second case, $B_1$ and $B_2$ have links to both $A_1$ and $A_2$. With our approach, the two resulting lattices are the same with $B_1$ and $B_2$ linked to both $A_1$ and $A_2$.

The semantics of schema changes affecting classes is described in the following sections. The only two changes considered are adding and dropping classes.

### Add Class (AC)

Class addition is class creation as defined by the primitive model. Creating a class adds it to $CSO$, which in turn adds it to the schema. The behavior *B_new* defined for classes can be applied to **C_class** to create a new class object. The *B_new* behavior accepts a type argument to be associated with the new class. The operation is rejected if the type already has an associated class or if the type defines a behavior that does not have an associated implementation – a class can only be created if the type has implementations defined for all its behaviors. A class manages the instances of a type. The creation of a class allows instances of its associated type to be created.

### Drop Class (DC)

This operation drops a given class removing it from $CSO$ and, therefore, removing it from the schema as well. Dropping a class terminates its lifespan in the class **C_class**. The *B_drop* behavior defined on `T_object` is refined in type `T_class` to perform class dropping.

The instances of a dropped class are also dropped. As mentioned above, if the model includes object migration techniques, instances can be migrated to another class before dropping the class in order to preserve their existence before dropping the class.

### Drop Behavior (DB)

Since explicitly dropping behaviors from a type definition (operation MT-DB) is a schema change, dropping a behavior in its entirety is also a schema change because the behavior may be defined on one or more types.

The DB operation drops a given behavior, which could possibly remove it from $BSO$ and, therefore, remove it from the schema as well. Dropping a behavior terminates its lifespan in the class **C_behavior**. The *B_drop* behavior defined on `T_object` is refined in type `T_behavior` to perform behavior dropping.

A dropped behavior is also dropped from all types that define the behavior either natively or through inheritance. The semantics of this operation follows dropping behaviors from types (operation MT-DB) defined above. Therefore, the implementation of *B_drop* in type

| Old Implementation | New Implementation |
|---|---|
| *computed*$_i$ | *computed*$_j$ |
| *computed*$_i$ | *stored*$_j$ |
| **stored**$_i$ | **stored**$_j$ |
| *stored*$_i$ | *computed*$_j$ |
| *undefined* | *stored*$_j$ |
| *undefined* | *computed*$_j$ |

Table 5.2: Valid implementation changes of a behavior in a type.

`T_behavior` can be defined in terms of dropping the given behavior from all types that define it, followed by timestamping the lifespan of the behavior in class **C_behavior**.

### Modify Behavior - Change Association (MB-CA)

The modification to a behavior that is considered to be a schema change is re-associating a different function with a behavior in a particular type. Behavior *B_associate* defined on `T_behavior` is provided as part of the primitive system to perform user-level behavior/function association changes. The signature of *B_associate* is as follows:

$$B\_associate : \texttt{T\_type} \rightarrow \texttt{T\_function} \rightarrow \texttt{T\_behavior}$$

*B_associate* is applied to a behavior and accepts a type and a function as arguments. The behavior must be defined on the type argument and the result is to associate the function argument as the implementation of the behavior in the given type.

Recall that stored and computed functions represent the implementations of behaviors. The valid association changes are shown in Table 5.2. The notation *computed*$_i$ and *stored*$_i$ refer to computed and stored functions respectively. The subscripts $i$ and $j$ are used to denote distinct functions. The term *undefined* is for the case when the behavior is not associated with any function. The combinations *computed*$_i$ to *computed*$_i$ and *stored*$_i$ to *stored*$_i$ are not included in the table because these do not reflect changes in function association. The *emphasized* rows represent user-level changes and the **bold** row is a system-level change for reorganizing the internal representation of objects.

A system defined primitive function called *F_STORED* is provided to associate a behavior with a stored function. The details of the stored location that the function accesses (e.g., slot number in the object) is transparent to the user and is handled internally by the system. One example of using the bold entry in the table is during multiple inheritance. It is usually necessary to reorganize the order of slots in the subtype because of slot number conflicts between the multiple supertypes. Changing a behavior from accessing one slot to accessing another is conceptually a change in implementation. This approach is uniform and is easily perceived to be the case if one considers each slot to have a separate stored function defined for it. Obviously, it is not implemented in this way (see [Ira93] for details on implementation), but it serves as a uniform framework for characterizing implementation changes.

Since changing the association of a function with a behavior is considered a schema change, dropping a function in its entirety must also be a schema change because the function may be associated as the implementation of a behavior in some type.

### Drop Function (DF)

This operation drops a given function, which could possibly remove it from $FSO$ and, therefore, from the schema. Dropping a function terminates its lifespan in the class **C_function**. The *B_drop* behavior defined on `T_object` is refined in type `T_function` to perform function dropping.

Only user-defined computed functions can be dropped. The operation is rejected if the function is associated as the implementation of a behavior in a type that has an associated class. These behaviors must be re-associated to other functions prior to dropping the given function. For those behaviors associated to the function in types that don't have an associated class, the behaviors become undefined in these types when the function is dropped.

### Drop Collection (DL)

This operation drops a given collection removing it from $LSO$ and, therefore, removing it from the schema as well. Dropping a collection terminates its lifespan in the class **C_collection**. The *B_drop* behavior defined on `T_object` is refined in type `T_collection` to perform collection dropping which simply drops the collection and nothing else. That is, the instances of a dropped collection are not affected because of their existence in a class.

### Add Collection (AL)

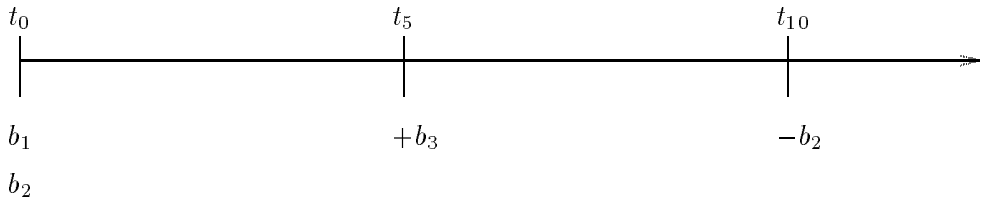
Collection addition is collection creation as defined by the primitive model. Creating a collections adds it to $LSO$, which in turn adds it to the schema. The behavior *B_new* defined classes can be applied to **C_collection** to create a new collection object. The *B_new* behavior accepts a type argument that denotes the membership type of the new collection. A collection is a user-defined and user-managed grouping of objects. Thus, modification of collections is left to the user and is not considered as part of schema evolution.

### Other changes

The remaining entries in Table 5.1 represent changes that are not considered part of the schema evolution problem. Each is discussed in this section to describe why it is not included as part of schema evolution.

Creating, dropping, and updating object instances (operations AO, DO, and MO) other than the schema instances discussed above clearly are operations concerned with the real-world concepts modeled in the objectbase and, therefore, do not have an affect on the schema. Defining a new behavior (operation AB) does not affect the schema because behaviors don't become part of the schema until after they are added to the interface of some type. Defining a new function (operation AF) does not affect the schema because functions don't become part of the schema until after they are associated as the implementation of a behavior defined on some type. Modifying a function (operation MF) does not affect the semantics of the behaviors it may be associated with and, therefore, this operation does not affect the schema.

Collections are groupings of objects that are defined and maintained by the user. Modifying a collection involves changing the membership of its extent and changing its membership type. These are operations related to the contents of the collection and, therefore, are not part of the schema evolution problem.




Interface history of type T:

$\{<[t_0, t_5), \{b_1, b_2\}>,\ <[t_5, t_{10}), \{b_1, b_2, b_3\}>,\ <[t_{10}, now], \{b_1, b_3\}>\}$

Figure 5.4: History of the interface of type $T$.

## 5.7 Versions of Types with Time

In this section, the incorporation of time to model versions of a type interface and implementation histories of behaviors is presented. The various changes that can occur on types, how these changes are reflected in the time model to manage type versions, and how the changes affect the instances of the type are described. The changes considered include: adding a behavior to a type, dropping a behavior from a type, and changing the implementation of a behavior for a particular type. These three changes were shown in the previous section to be the basis of most other schema changes.

### 5.7.1 Adding/Dropping Behaviors

As specified in Chapter 2, every type has an *interface*, which is the collection of behaviors that are applicable to the objects of that type. Recall that an interface consists of both native and inherited behaviors. Also recall that there are three behaviors defined on `T_type` that return the various components of a type's interface: *B_native* returns the collection of native behaviors, *B_inherited* returns the inherited behaviors, and *B_interface* returns the entire interface of a type.

In order to version the aspects of schema evolution that deal with adding behaviors to a type and dropping behaviors from a type, the three interface behaviors are redefined to be temporal behaviors. Thus, to keep with naming conventions, they will be referred to as *BT_native*, *BT_inherited*, and *BT_interface*.

Note that separate histories for each of these behaviors need not be explicitly maintained. For example, in an implementation one can choose to only maintain the native behaviors of a type. The entire interface of a type can be derived by unioning the native behaviors of all the supertypes of the type. The inherited behaviors can be derived by taking the difference of the interface and the native behaviors of the type. As another alternative, one may choose to maintain the interface of a type and derive the native and inherited behaviors. In this approach, the native behaviors of a type can be derived by unioning the interfaces of the direct supertypes and subtracting the result from the interface of the type. The inherited behaviors can be derived in the same way as above. Throughout the remainder of this thesis, histories of interfaces in the abstract sense are considered and the actual maintenance of them is left as an implementation detail.

With the time-varying interface extensions, the various aspects of a type's interface can be determined at any time of interest. For example, Figure 5.4 shows the history of the entire interface for a type $T$. A timeline representation and the result of $T.B\_interface$ are shown. The notation $+b_i$ and $-b_i$ are used to indicate the adding and dropping of some behavior $b_i$, respectively. At time $t_0$, behaviors $b_1$ and $b_2$ are defined on $T$ and the initial history of $T$'s interface is $\{<[t_0, now], \{b_1, b_2\}>\}$. At time $t_5$, a behavior $b_3$ is added to $T$. To reflect this change, the interface history is updated to $\{<[t_0, t_5), \{b_1, b_2\}>, <[t_5, now], \{b_1, b_2, b_3\}>\}$. This shows that between $t_0$ and $t_5$ only behaviors $b_1$ and $b_2$ are defined and between $t_5$ and *now* behaviors $b_1, b_2$ and $b_3$ exist. Next, at time $t_{10}$, behavior $b_2$ is dropped from type $T$. The final history of the interface of $T$ is shown in Figure 5.4. The difference from the previous history is that the second entry is timestamped with the open time of $t_{10}$ and a third entry $<[t_{10}, now], \{b_1, b_3\}>$ is added to reflect the change of dropping behavior $b_2$. The native and inherited behaviors would contain similar histories. Using this information, the interface of a type at any time of interest can be reconstructed. For example, at time $t_3$ the interface of type $T$ was $\{b_1, b_2\}$, at time $t_5$ it was $\{b_1, b_2, b_3\}$, and at time *now* it is $\{b_1, b_3\}$.

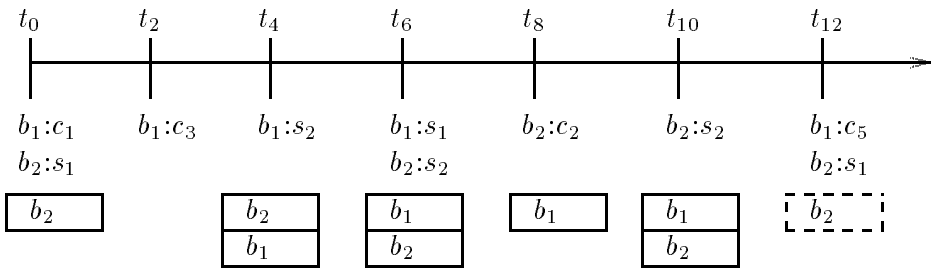
### 5.7.2 Changing Implementations of Behaviors

Each behavior defined on a type has a particular implementation for that type. The *B_implementation* behavior defined on `T_behavior` accepts a type as an argument and returns the implementation (function) of the receiver behavior for the given type. In order to model the aspect of schema evolution that deals with changing the implementations of behaviors on types, the implementation behavior is redefined to be a temporal behavior *BT_implementation*.

With this behavior being temporal, the implementation of a behavior on a particular type at any time of interest can be determined. For example, Figure 5.5 shows the history of the implementations for behaviors $b_1$ and $b_2$ on type $T$. A timeline representation and histories of $BT\_implementation.B\_history(b_1)$ and $BT\_implementation.B\_history(b_2)$ are shown. The interface history of $T$ is also shown for clarity. The notation $c_i$ denotes a computed function, $s_i$ a stored function, and $b_j{:}c_i$ or $b_j{:}s_i$ denotes the association of a computed or stored function with behavior $b_j$. Moreover, for stored functions, the subscript $i$ refers to a location (e.g., a slot number) in an object representation that the stored function accesses. An object representation (i.e., the state of an object) consists of a number of slots for holding information carried by the object. The representations of objects at different times according to the stored functions associated with behaviors at those times are depicted by the boxes labeled with behaviors. For example, at time $t_4$, the object representation consists of two slots – the first slot is for the stored implementation of behavior $b_2$ and the second is for $b_1$. At time $t_8$, the object representation consists of only one slot which is for $b_1$.

Figure 5.5 is used to describe how the implementation changes in Table 5.2 are maintained by implementation histories. At time $t_2$, the implementation of $b_1$ changed from the computed function $c_1$ to the computed function $c_3$. At time $t_4$, the implementation of $b_1$ changed from the computed function $c_3$ to the stored function $s_2$. At time $t_6$, the implementation of $b_1$ changed from the stored function $s_2$ to the stored function $s_1$. At the same time, $b_2$ changed from $s_1$ to $s_2$. At time $t_8$, the implementation of $b_2$ changed from the stored function $s_2$ to the computed function $c_2$. All these changes are reflected in the implementation histories of behaviors $b_1$ and $b_2$.

Note that at time $t_{12}$ the behavior $b_1$ was changed from the stored behavior $s_1$ to the




Implementation history of behavior $b_1$ for type T:

$\{<[t_0, t_2), c_1>,\ <[t_2, t_4), c_3>,\ <[t_4, t_6), s_2>,\ <[t_6, t_{12}), s_1>,\ <[t_{12}, now], c_5>\}$

Implementation history of behavior $b_2$ for type T:

$\{<[t_0, t_6), s_1>,\ <[t_6, t_8), s_2>,\ <[t_8, t_{10}), c_2>,\ <[t_{10}, t_{12}), s_2>,\ <[t_{12}, now], s_1>\}$

Interface history of type T:

$\{<[t_0, now], \{b_1, b_2\}>\}$

Figure 5.5: Implementation histories of behaviors $b_1$ and $b_2$ for type T and object representations

computed behavior $c_5$. Since all object representations at time $t_{12}$ require only one slot, the change to $b_1$ implies a change to $b_2$ so that at time $t_{12}$ behavior $b_2$ accesses slot one instead of slot two. Furthermore, note that the implicit implementation change of $b_2$ was from a stored function to a stored function which is a system managed change and therefore is transparent to the user. The implicit implementation change of $b_2$ is reflected in its history by the two entries $<[t_{10}, t_{12}), s_2>$ and $<[t_{12}, now], s_1>$. In general, the slots of an object representation are reorganized (meaning an implicit change occurs) whenever a stored to computed implementation change removes a slot other than the last slot of an object's representation. The system can also rearrange slots as part of an implementation change.

By tightly integrating temporal aspects of the TIGUKAT object model with schema changes, the behaviors, their implementations, and the object representations for any type can be reconstructed at any given time $t$. For example, the interface of type $T$ at time $t_7$ is given by the behavior application $T.[t_7]B\_interface$, which results in the collection $\{b_1, b_2\}$. The syntax $o.[t]b$ denotes the application of behavior $b$ to object $o$ at time $t$. The implementation of $b_1$ at time $t_7$ is given by $b_1.[t_7]B\_implementation(T)$, which is $s_1$. Similarly, the implementation of $b_2$ at time $t_7$ is given by $b_2.[t_7]B\_implementation(T)$, which is $s_2$. Since there are two stored functions, this implies a two slot representation for objects at time $t_7$. That is, $b_1$ accesses slot one using stored function $s_1$ and $b_2$ accesses slot two using stored function $s_2$.

## 5.8 Change Propagation

Propagation of changes in TIGUKAT uses a filtering approach with explicit coercion of behaviors. That is, when a change is made to the schema, the change is not automatically

propagated to the instances. Instead, the old version of the schema is maintained and the change is recorded in the proper behavior histories. The objects continue to maintain the characteristics of the older schema. New objects correspond to the semantics of the newer schema. Objects from the older schema can be coerced to the newer schema one behavior at a time. Thus, portions of an object (i.e., some behaviors) may correspond to older schema, while other portions correspond to newer schema. This is a novel characteristic of the approach. Note that an object can be coerced to a newer version in its entirety by coercing all the behaviors of that object.

When coercing an object to a newer version, if the object has temporal characteristics (i.e., there are temporal behaviors defined on it), the old version of these temporal aspects are maintained. In this case historical queries can be run on the object.

Recall that an object is created as an instance of a particular type. The creation time of every object is recorded by the behavior *B_created* defined on `T_object`. Applying the behavior returns the time that the object was created. The signature of *B_created* is as follows:

$$B\_created : \texttt{T\_instant}$$

Note that *B_created* is not a temporal behavior. Also note that the behavior is introduced for convenience and is equivalent to the lower bound of the lifespan of an object in its class. That is, for a given object $o$, the following equivalence holds, where *B_lb* is a behavior defined on intervals that returns the lower bound of an interval:

$$o.B\_created \equiv (o.B\_lifespan(o.B\_mapsto.B\_classof)).B\_lb$$

The behaviors applicable to an object are those that exist in the interface of its type at the creation time of the object. The implementations of these behaviors are those that exist in the implementation histories for the type at the creation time of the object. The stored functions at the creation time of the object determine the representation of the the initial state of the object.

As time progresses and types evolve, the interface of a type and the implementations of its behaviors may change. Any behavior applicable to an object can be explicitly coerced to a newer implementation of the behavior. The change is recorded in the *B_changes* behavior defined on `T_object`. The signature for *B_changes* is as follows:

$$B\_changes : \texttt{T\_list}\langle\texttt{T\_timemodel}, \texttt{T\_behavior}\rangle$$

The result of *B_changes* is a list of (*time, behavior*) pairs. When a behavior for a particular object is coerced into a newer implementation, the time of the coercion and the behavior coerced is recorded in the *B_changes* list of the object.

The *B_changes* list is used in the behavior dispatch routine (defined in Section 5.9) to determine the most recent coercion time of a behavior being applied to an object. This time is used as a reference point for determining the appropriate implementation of the behavior at that time.

An object can be coerced to a behavior with a newer implementation that changes from computed to stored, stored to computed, computed to computed, and stored to stored. The first three are user-level changes, while the last is a system-level change that is strictly internal and not accessible to the user. The change from computed to stored and vice versa require a change to the state of an object by either adding or dropping a slot representing the stored information. A system managed change that requires state changes is the

reorganization of slots (i.e., stored to stored). In order to maintain the old state versions of temporal objects, the notion of an object as an (*identity, state*) pair is extended to one where an object is an (*identity, state-history*) pair. Since states are not objects, the state-history of an object is managed internally by the system. It is similar to other histories in that time-intervals are used to record changes in the state. Whenever a change to the representation of an object occurs due to the coercion of one of its temporal behaviors, the change is recorded in the state-history of the object. Thus, a temporal object is generic in the sense that it consists of all its representations over time. This is called the *generic instance* of the object. The default representation of a generic instance is the most current representation in the state-history. The individual representations of an object denote how the object existed at certain times in the past. Each of these representations is called a *version instance* of the object. Thus, the generic instance is the most current version instance of an object. Each version instance is an object in its own right in that it contains the state-history of the object up until the given version representation.

The primitive behavior *B_self* defined on `T_object` is refined to accept a time argument and returns the *version instance* of an object at the given time. That is, for an object $o$ and a time $t$, the behavior application $o.B\_self(t)$ returns the the version instance $o'$ of $o$ such that the most current representation (i.e., default) representation is the one at time $t$, which includes the entire history prior to $t$. Using this construct, historical states of an object's "self" can be retrieved.

## 5.9 Temporal Behavior Dispatch

The previous sections established the mechanism for versioning behaviors of a type, the implementations of behaviors for a type, and the states of objects. In this section, the behavior dispatch process for applying a behavior $b$ to an object $o$ at given time $t$ is described. The syntax $o.[t]b$ is used to denote this application. The time component is optional and if left out, the current time *now* is assumed.

### 5.9.1 Overview

Figure 5.6 provides an overview of the dispatch process. A behavior application is first checked for validity. It is considered valid if the object $o$ exists at time $t$ and behavior $b$ is defined in the interface of $o$'s type at time $t$. An invalid behavior application produces an error. For a valid application, an appropriate time reference point $r$ is found. The time reference point is either the time component of the most recent coerced entry for $b$ in the *B_changes* list of $o$ going back in the history starting from time $t$ or it is the *B_created* time of $o$ if there is no appropriate entry in *B_changes*. The time reference point $r$ is used as an index into the *B_implementation* history of $b$ for the type of $o$ to retrieve the appropriate implementation $f$. If there is no implementation defined at time $r$, then $o$ is coerced to the first defined implementation of $b$. The implicit coercion is an internal operation that may or may not be transparent to the user. In an interactive environment, the system could ask the user to choose an appropriate implementation for the behavior. If the implementation is a computed function, then the function is simply applied to object $o$. However, if the implementation is a stored function, then the time reference point $r$ is used to retrieve the object $o$ that has the appropriate state representation at time $r$. We denote this representation of the object by $o'$. The object $o'$ is the same object as $o$, but the state of $o'$ is the state at time $r$. The stored function $f$ is applied to $o'$ and accesses

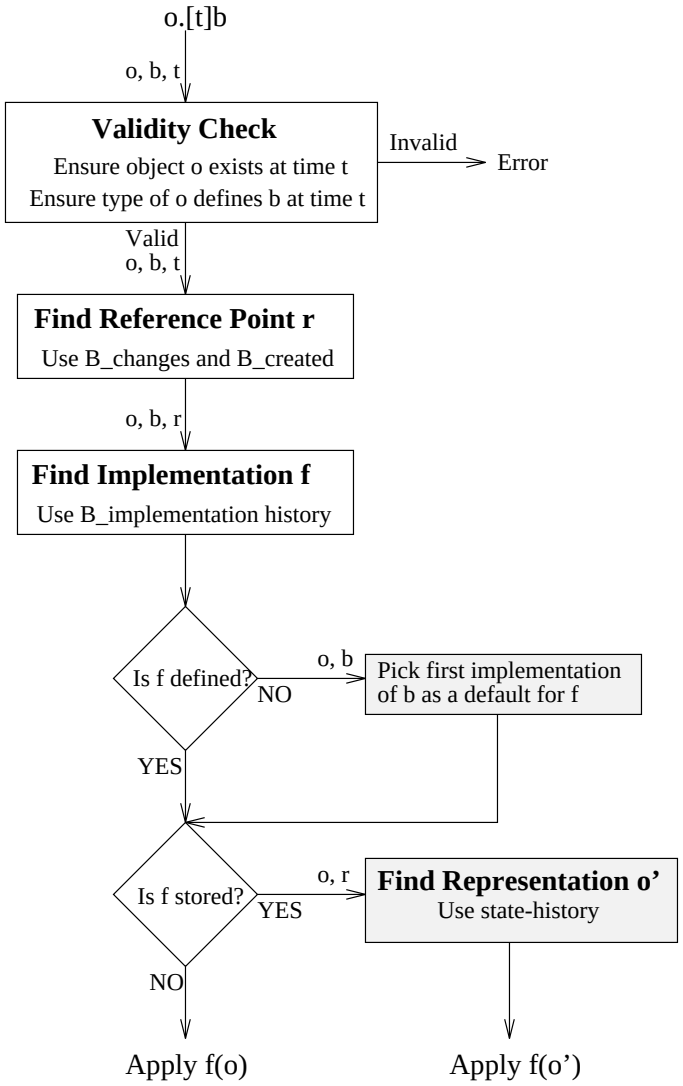



Figure 5.6: Dispatch process for applying a behavior $b$ to an object $o$ at time $t$.

the appropriate state of $o$. The implicit coercion and representation retrieval operations are grayed in Figure 5.6 to highlight that they are internal system operations.

### 5.9.2 Dispatch Semantics

In order for a behavior application to be valid, object $o$ must exist at time $t$ and the behavior $b$ must be defined in the interface of the type of $o$ at time $t$. The validity check algorithm, Algorithm 5.1 (*Validity*) performs this test in the form of a logical expression.

The first part of the expression (5.1) checks that $o$ exists at time $t$ by testing whether time $t$ lies within the lifespan of $o$ in the class of its associated type. In the second part of the expression (5.2), $B\_interface.B\_history(o.B\_mapsto)$ returns the interface history for the type of object $o$. This history is searched for an entry $x$ that satisfies the third part (5.3) of the expression, which checks that time $t$ lies within the time interval of entry $x$, and the fourth part (5.4), which checks that behavior $b$ is part of the interface of the type at this time. If all conditions are satisfied, the behavior application is valid and `true` is returned.

**Algorithm 5.1** *Validity:*

**Input:** An object $o$, a behavior $b$ and a time $t$

**Output:** True if the application is valid, false otherwise

**Procedure:**

$$\begin{aligned}
\textbf{return} \quad & (t.B_within(o.B_lifespan(o.B_mapsto.B_classof)) && (5.1)\\
& \wedge\ \exists x(x \in BT_interface.B_history(o.B_mapsto) && (5.2)\\
& \quad \wedge\ t.B_within(x.\rho_1) && (5.3)\\
& \quad \wedge\ b \in x.\rho_2)) && (5.4)
\end{aligned}$$

Otherwise, the behavior application is invalid and false is returned.

If the validity test is satisfied, the next step is to determine the proper time reference point so that the appropriate implementation can be retrieved. Algorithm 5.2 (*Reference-Point*) performs this operation and returns the proper time reference point.

Algorithm 5.2 indexes into the *B_changes* list of the argument object $o$ for the most recent entry containing behavior $b$ with a time point less than or equal to the time $t$. If an entry is found, the time component of the entry is returned as the time reference point. If an entry is not found, the created time of $o$ is returned.

Using the time reference point $r$, the proper implementation of the behavior is found. Algorithm 5.3 (*Implementation*) finds and returns this implementation. Note that an implementation may not be defined at the given reference point $r$. This can occur if a behavior has been added to the interface of the type at a time later than $r$ and older objects have not been coerced to the new interface. In this case, the object is implicitly coerced to the first implementation of the behavior and this implementation is returned. The user can later coerce the behavior to a newer implementation if desired. In an interactive environment, the system could give the user the option of choosing which implementation to coerce the behavior to or may allow the user to leave the behavior's implementation undefined. This gives the user the flexibility to coerce the behavior to any implementation desired.

If the function returned by Algorithm 5.3 is a stored function, the representation of object $o$ at reference point $r$ must also be found since the stored function was defined for the representation at this time. Algorithm 5.4 (*Representation*) performs the simple task of returning the version instance of object $o$ at time reference point $r$. This is done using the *B_self* behavior and passing the time point $r$ as an argument.

If the function returned by Algorithm 5.3 is a computed function, then there is no need to determine a specific representation since computed functions apply behaviors to other objects and do not depend on any particular representation. The behavior applications inside computed functions go through the same behavior dispatch process and therefore appropriate version instances will be determined as required.

As the final step, if the function returned from Algorithm 5.3 is stored, then it is applied to the version instance returned from Algorithm 5.4. Otherwise, the function must be computed and is simply applied to generic instance $o$. The relationships between the algorithms are shown in Algorithm 5.5 (*Dispatch*).

**Algorithm 5.2** *ReferencePoint:*

**Input:** An object $o$, a behavior $b$ and a time $t$

**Output:** A time reference point

**Procedure:**

Index into the *B_changes* list of $o$ for an entry $E$ that satisfies the following conditions:

- the behavior element of $E$ matches $b$,
- the time element of $E$ is $\leq t$,
- there does not exist another entry $E'$ satisfying the above two conditions whose time element is greater than the time element of $E$.

**if** an entry $E$ found **then**
    **return** time element of $E$
**else**
    **return** $o.B\_created$

**Algorithm 5.3** *Implementation:*

**Input:** An object $o$, a behavior $b$ and a time reference point $r$

**Output:** The function that implements behavior $b$ for object $o$ at time $r$

**Procedure:**

**if** $b.B\_implementation(o.B\_mapsto)$ has an entry at time $r$ **then**
    **return** the implementation element associated with this entry
**else**
    **return** the first implementation of $b$ as a default

**Algorithm 5.4** *Representation:*

**Input:** An object $o$ and a time reference point $r$

**Output:** An object with its representation at time reference point $r$

**Procedure:**
**return** $o.B\_self(r)$

**Algorithm 5.5** *Dispatch:*

**Input:** An object $o$, a behavior $b$ and a time $t$

**Output:** An object resulting from the application $o.[t]b$

**Procedure:**

**if** $Validity(o, b, t)$ **then**
  $r \leftarrow ReferencePoint(o, b, t)$
  $f \leftarrow Implementation(o, b, r)$
  **if** $f$ is a stored function **then**
    $o' \leftarrow Representation(o, r)$
    **return** $f(o')$
  **else**
    **return** $f(o)$
**else**
  **INVALID:** object $o$ does not exist at time $t$
    or behavior $b$ not defined in the interface of $o$'s type at time $t$

### 5.9.3 Examples

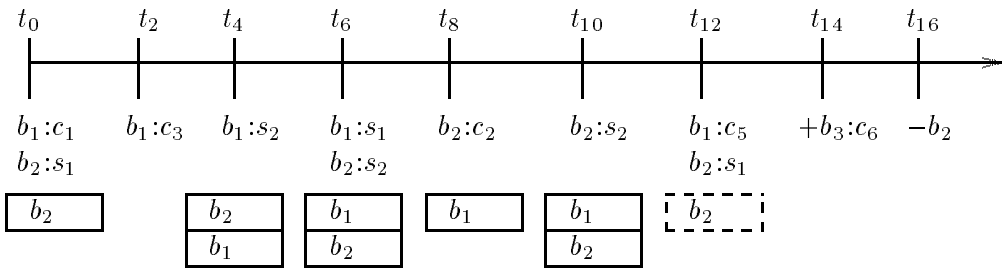
For the following examples, consider Figure 5.7, which extends the timeline of type $T$ in Figure 5.5 by adding a behavior $b_3$ with the computed implementation $c_6$ at time $t_{14}$ and dropping the behavior $b_2$ at time $t_{16}$. Note that the object representation will not change by adding behavior $b_3$ and the representations will be empty after behavior $b_2$ is dropped.

Furthermore, consider Figure 5.8, which contains two example objects created as instances of type $T$. The figure shows the created time, the changes list and the internal state-history of the objects. For the state-histories the notation $rep@t_i$ is used to denote the version instance of an object at time $t_i$. Object $o_1$ was created at time $t_0$. The default behaviors and implementations for this object are those that exist at time $t_0$. Namely, $b_1 : c_1$ and $b_2 : s_1$ (see Figure 5.5). The behavior $b_1$ for this object was coerced to a version at time $t_9$, behavior $b_2$ was coerced to a version at time $t_{11}$ and behavior $b_3$ was coerced to time $t_{14}$. The internal state-history of $o_1$ has three different version instances that correspond to the entries in the changes list. Object $o_2$ was created at time $t_6$. Its default behaviors and implementations are $b_1 : s_1$ and $b_2 : s_2$. It has no entries in its changes list and, therefore, has only one version instance in its state history.

Several example behavior applications using time are presented to show how the dispatch process is followed in order to determine the proper implementation and version instance that are appropriate at the given time of interest.

**Example 5.1** Behavior application $o_1.[t_7]b_1$

**Validity:** Object $o_1$ was created at time $t_0$ and exists at time $now$. Therefore, the lifespan of $o_1$ is the time interval $[t_0, now]$. Since $t_7$ in within this interval (i.e., lifespan), the object part of the behavior application is valid.




Implementation history of behavior $b_1$ for type T:

$\{<[t_0, t_6), s_1>,\ <[t_6, t_8), s_2>,\ <[t_8, t_{10}), c_2>,\ <[t_{10}, t_{12}), s_2>,\ <[t_{12}, t_{16}], s_1>\}$

Implementation history of behavior $b_2$ for type T:

$\{<[t_{14}, now], c_6>\}$

Interface history of type T:

$\{<[t_0, t_{14}), \{b_1, b_2\}>,\ <[t_{14}, t_{16}), \{b_1, b_2, b_3\}>,\ <[t_{16}, now], \{b_1, b_3\}>\}$

Figure 5.7: Example showing effects on implementation histories of first adding and then dropping a behavior.

Object $o_1$

| | | |
|---|---|---|
| *B_created* | = | $t_0$ |
| *B_changes* | = | $\{<t_9, b_1>, <t_{11}, b_2>, <t_{14}, b_3>\}$ |
| state-history | = | $\{<[t_0, t_9), rep@t_0>, <[t_9, t_{11}), rep@t_9>,$ $<[t_{11}, now], rep@t_{11}>\}$ |

Object $o_2$

| | | |
|---|---|---|
| *B_created* | = | $t_6$ |
| *B_changes* | = | $\{\ \}$ |
| state-history | = | $\{<[t_6, now], rep@t_6>\}$ |

Figure 5.8: Two example objects of type $T$.

The type of $o_1$ is $T$. The interface of $T$ at time $t_7$ is $\{b_1, b_2\}$. Since $b_1$ is part of this interface, the behavior part of the application is valid and thus the validity test is satisfied.

**Reference Point:** The next step is to find an appropriate time reference point with respect to $t_7$. Searching through the *B_changes* list of $o_1$, we find there is no entry that satisfies the criteria in Algorithm 5.2. Thus, the *B_created* time $t_0$ is returned as the reference point.

**Implementation:** Using the time reference point $t_0$, we pick out the appropriate implementation of $b_1$ for type $T$ at time $t_0$, which is the computed function $c_1$.

**Representation:** Since the function returned in the previous step is a computed function, this step is skipped.

**Dispatch:** To complete the dispatch of the behavior, the computed function $c_1$ is executed

using object $o$ as an argument.

**Example 5.2** Behavior application $o_1.[t_{10}]b_1$
The validity test is satisfied. There is an entry $<t_9, b_1>$ in the changes list of $o_1$ that satisfies the criteria of Algorithm 5.2. Thus, we use $t_9$ as the time reference point for finding the appropriate implementation of $b_1$ for type $T$. The implementation chosen is the stored function $s_1$. Since this is a stored function, we also get the object $o_1'$ with the appropriate representation at time $t_9$ which is $rep@t_9$. We can now apply $s_1$ to $o_1'$. The function and representation are correct for $o_1$ since behavior $b_1$ was coerced to the new version at time $t_9$ for this object.

**Example 5.3** Behavior application $o_1.[t_{10}]b_2$
The validity test is satisfied. There is no entry in the changes list for $b_2$ with a lesser time than $t_{10}$. Therefore, we use the created time $t_0$ as the time reference point. This gives the implementation $s_1$ and the object $o_1'$ with representation $rep@t_0$. We can now apply $s_1$ to $o_1'$. Note that this example and the previous one both apply $s_1$ for different behaviors (namely, $b_1$ and $b_2$). The reason they are valid is that they are applied to different representations of object $o_1$ as well.

**Example 5.4** Behavior application $o_1.[t_{15}]b_3$
The validity test is satisfied. Since there is an appropriate entry $<t_{14}, b_3>$ in the changes list of $o_1$, we use $t_{14}$ as the time reference point which gives the implementation $c_6$. Since this is a computed function, we simply apply $c_6$ to $o_1$. This is correct for $o_1$ since behavior $b_3$ was coerced to the new version at time $t_{14}$ for this object.

**Example 5.5** Behavior application $o_1.[now]b_2$
This fails the validity test because behavior $b_2$ is not part of the interface of $T$ at time $now$.

**Example 5.6** Behavior application $o_2.[t_7]b_2$
The validity test is satisfied. There are no entries in the changes list for $o_2$ so we use the created time $t_6$ as the time reference point. This gives the implementation $s_2$ and the object $o_2'$ with representation $rep@t_6$.

**Example 5.7** Behavior application $o_2.[t_{13}]b_2$
The validity test is satisfied. Again, because there are no entries in the changes list for $o_2$ we use the created time $t_6$ as the time reference point. This gives the implementation $s_2$ and the object $o_2'$ with the representation $rep@t_6$.

**Example 5.8** Behavior application $o_2.b_1$
Since no time point is specified, the default time $now$ is assumed. The validity test is satisfied. There are no entries in the changes list for $o_2$ so we use the created time $t_6$ as the time reference point. This gives the implementation $s_1$ and the object $o_2'$ with representation $rep@t_6$.

**Example 5.9** Behavior application $o_2.b_3$
The validity test is satisfied. There are no entries in the changes list for $o_2$ so we use the created time $t_6$ as the time reference point. There is no implementation defined for $b_3$ on type $T$ at time $t_6$. Therefore, we implicitly coerce $o_2$ to the first implementation of $b_3$ which is at time $t_{14}$. This adds the entry $<t_{14}, b_3>$ to the changes list of $o_2$. Now, the implementation chosen is $c_6$. Since this is a computed function, no particular representation is required and we simply apply it to $o$.

# Chapter 6

# Conclusions

## 6.1 Summary and Contributions

The first result of the thesis is the definition of a uniform behavioral object model with sufficient power and expressibility for supporting the data and information management requirements of advanced applications such as geographic information systems, engineering databases, office information systems, knowledge base systems, and multi-media databases. These applications require the management of complex objects with complex relationships. User access to such systems is characterized by long-running, interactive transactions that involve large and semantically diverse units of data. Thus, the functionality required of objectbase management systems (OBMSs) subsumes the functionality of their predecessors.

A high-level abstract behavioral object model is integrated with a formal structural counterpart to form a complete model definition. The reconciling of these two components helps in understanding the semantics of the model and is a favorable basis for an implementation.

The fundamental contributions of the object model are the following:

1. A precise specification and integration of both the behavioral and structural aspects of an object model with sufficient power for handling advanced database functionality.

2. A clean separation and precise definition of many object model features which are usually bundled and only intuitively defined in other studies.

3. A uniform approach to objects which models all information as first-class objects with well-defined behavior. The result is an extensible model capable of defining other components of an OBMS within itself. It is shown in this thesis how uniformity is used to define an object query model, provide reflection and define schema evolution strategies, all within the model itself. Other work has extended this approach to an extensible query optimizer [Muñ94] and this could be extended to the view manager as well.

In keeping with the uniformity aspects of the object model, the query model is defined in a consistent way as type and behavior extensions to the base object model. Thus, queries are objects with well-defined behavior. This is a uniform object-oriented approach to developing an extensible query model that is seamlessly integrated with the object model. This kind of natural extension is possible due to the uniformity built into the object model which treats everything as a first-class object and allows the consistent abstraction of an object's

"attributes" into the uniform semantics of behaviors. This specification has been used as a foundation for implementing the query model and its user language.

The formal object calculus is a powerful declarative object creating language that incorporates the behavioral paradigm of the object model. Safety is based on the evaluable class of queries [GT91] which is arguably the largest decidable subclass of the domain independent class [Mak81]. The class of evaluable queries defined is *wide-sense evaluable* with respect to equality and membership atoms, meaning a broader class of safe queries is recognized by the approach. The object algebra includes a powerful, complete set of the behavioral/functional operators that fully support the object-creating nature of the calculus. A novel operator is *behavioral projection*, which is a form of type generalization and has applications in view support. Other notable operators include a generalized *map* for applying behaviors to elements of collections, a *select* and the derived *join* and *generate join* operators. The calculus and algebra are proven to be equivalent in expressive power. Furthermore, a feasible translation algorithm from calculus to algebra is presented that does not depend on the formation of (potentially) large *DOM* domains. Object creating languages require the ability to perform type inferencing because newly created objects may not correspond to any type in the lattice. As part of the algebra, the relationship of the operators to the schema in terms of the creation and integration of new types is defined.

The contributions and novelty of the query model are the following:

1. It incorporates a formal and powerful object calculus and object algebra with a proven equivalence in expressive power and a complete feasible algorithmic translation from calculus to algebra.

2. Its safety criterion is based on the *evaluable* class of queries [GT91] which is arguably the largest decidable subclass of domain independent queries [Mak81]. An additional form of safety with respect to the closure of a query is also defined. The class of safe queries defined in this thesis is the largest class of any object model to date.

3. It exploits object-oriented features to extend the *evaluable* class by introducing notions of *object generation* on equality and membership atoms which relaxes *range* specification requirements. The result is that a broader class of safe queries are recognized by the approach.

4. It uniformly models queries as first class objects by directly defining them as type and behavior extensions to the TIGUKAT object model. This makes for an extensible query model that has a consistent uniform underlying semantics commensurate with the object model. It is the most complete model that has defined the database functionality of a query model and temporal schema evolution as a uniform extension to the base object model. The uniformity extends to other components such as the query optimizer, view manager and object manager.

5. The extensible algebra specification forms a uniform basis for processing queries and is exploited by an extensible algebraic query optimizer and execution plan generator which are reported elsewhere [Muñ94, Ira93].

6. It is the most advanced extensible, uniform, behavioral object query model to formally bring together the components of an object calculus, an object algebra, proofs of completeness between the languages, and an effective algorithmic translation from the calculus to the algebra.

The uniform meta-architecture of the TIGUKAT object model is capable of managing information about itself and the access primitive of applying behaviors to objects is uniform over all forms of information, including the meta-information. Another result of this thesis is how the model's uniformity provides a basis for reflective capabilities. Types in the model support both structural and computational reflection which are seen as the two major forms of reflection.

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

Using an SQL-like query language, several "regular" queries on real-world objects are compared with queries on meta-information and it is shown that in a uniform model, there is no distinction between "normal" objects and meta-objects because everything has the status of a first-class object. Queries can access information about types, classes and collections (parts of the schema) by applying behaviors to objects in a uniform way. Queries can even mix regular and meta-objects in a single query.

The meta-system design has similarities to ObjVlisp [Coi87] and is a uniform extension to the Smalltalk-80 [GR89] meta-class architecture. It is more general in the sense that it can mimic the parallel meta-class structure of Smalltalk-80, but does not force this semantics. Other differences are that any class in TIGUKAT can have many instances and any type can be subtyped. Thus, the *metaness* of an object is a consequence of inheritance and gives rise to a uniform model. One advantage is reduced overhead since not all classes require a meta-class. However, some subtype reorganization is required if later it is decided that a particular class needs to specialize some other meta-class. These changes can be seen as application design corrections and the schema evolution policies make these changes natural since some form of them must be supported in a full-fledged OBMS anyway. Since behaviors are objects in TIGUKAT, some form of the meta-communication model of computational reflection could be integrated with the system. This is part of the future research of the TIGUKAT project.

The novelty and contributions of the meta-model design in TIGUKAT are as follows:

1. The meta-model is a uniform component that is integrated with the design of the base model. This means that the meta-objects such as types, classes, collections, behaviors, and functions are uniformly objects in TIGUKAT.

2. The uniformity of the meta-model provides a basis for reflective capabilities, which emerge naturally out of its uniform design. It was shown that the existing primitive features of TQL and the formal query model were sufficient for performing reflective queries, and that both "regular" and meta objects could be retrieved by these queries.

3. Types in TIGUKAT provide support for both structural and computational reflection, which are regarded as the two major forms of reflection. This thesis focused mainly on structural reflection.

4. The meta-model provides support for other features such as multiple *new* behaviors for creating various default forms of new objects and *class behaviors* for defining

behaviors that are applicable to an entire class of objects and perform an operation on certain properties of all objects in the class (e.g., average volume, total age, etc.) .

Schema evolution in the TIGUKAT model consists of a number of invariants that must be maintained over schema changes. A classification of all schema changes was made and the semantics of each change was defined. Since the model is uniform, schema evolution is the result of updating certain behaviors and its development was just a matter of identifying the semantics of these updates. By adding temporality to these behaviors, a history of schema changes is easily maintained and the entire schema can be reconstructed at any time of interest. This lays the foundation for developing versions of types, versions of schema, and versions of instances within the single framework of temporality.

A unique feature of the version model is that a temporal domain is introduced to implicitly manage histories for behaviors. Behavior histories are used to manage the properties of objects over time. Since everything in TIGUKAT is uniform, the schema are objects with well-defined behavior. By maintaining histories for appropriate behaviors of types, a model for versioning types is developed. This model is extended to behavior objects and object representations (state) as well. Since versioning occurs implicitly through the management of behavior histories, objects are instances of a type and not instances of a version of a type. This means that objects support the full semantics of a type instead of just a portion (version) of the type. This approach has the major benefit of maintaining semantic consistency between old and new versions of types and the programs that operate on their instances.

By using time to implicitly model versions of types and objects, the schema and its instances can be reconstructed at any time of interest. That is, the type lattice, type interfaces, behavior implementations and object representations can be recreated as they existed at a particular time of interest. One benefit of this approach is that historical queries can be run on the objectbase.

Another unique feature of the version model is that object coercion occurs on a "behavior at a time" basis instead of on the entire object. This means that objects can update certain behaviors to use those defined by a newer version of a type while allowing other behaviors to use older versions. This means that a history of the object's semantics is maintained which helps in maintaining semantic consistency between old and new versions of types and the programs that use them. Complete object coercion is possible by coercing all the behaviors of an object.

The novelty and contributions of the design of schema evolution and version control in TIGUKAT are as follows:

1. The integration of schema evolution and version control using a temporal domain is a new approach in object management.

2. Temporality based on behaviors, together with the uniformity of the model, unifies the various approaches of versioning proposed in the past. That is, by versioning behaviors (i.e., defining temporal behaviors on types in general) one gets versions of objects, by versioning behaviors on `T_type` one gets versions of types, and by versioning behaviors on type relationships such as *B_subtypes*, *B_supertypes*, etc., one gets versions of schema.

3. The temporal framework supports a filtering approach where objects are not updated to newer versions of the schema, but rather the semantic differences between the versions are maintained through interface and implementation histories of the schema.

Objects can be explicitly coerced to newer versions of the schema one behavior at a time. This means that an object may have some characteristics of older schema, some characteristics of newer schema, and may "skip" certain generations of schema changes. This is in contrast to other approaches where an object must be converted in its entirety to a newer version of the schema in a stepwise fashion from generation to generation.

## 6.2 Future Research

The work presented in this thesis suggests a number of interesting directions for future research. The uniformity of the model makes it an excellent candidate for developing an extensible view manager that is seamlessly integrated with the base model. As with the query model, views are objects whose semantics are defined by type and behavior extensions to the base model. This brings views into the model, meaning they can be operated on by behaviors, they can be queried, and they can be uniformly used to derive other views. The definition of a view restricts the objects that an application or user can see. Each view must consistently maintain all the properties of the model. Therefore, a view is like a sub-objectbase of the overall system that defines a *conset* of objects. A view definition may contain other views so that applications can easily switch from one view to another. Defining the semantics of a consistent view manager and developing a design methodology for creating views are major areas of research that can extend the functionality of the TIGUKAT objectbase management system.

One interesting direction to explore in the context of views relates to extending the temporal model to include a branching model of time and investigate how this can be used to support views. For example, each branch of time could be seen as a separate view of the objectbase with different objects, types, behaviors, collections, etc., visible along the various lines. The semantics of how these lines split, interact, and possibly merge are very interesting topics of future research.

The object-oriented approach is a suitable candidate for facilitating an integration of the data abstraction and computation model of object-oriented programming languages with the performance and consistency of an object query model. Traditionally, these two areas have developed orthogonally to one other. An integration would alleviate many problems (e.g., impedance mismatch) associated with embedded languages in use today. An interesting direction for future research lies in investigating how a uniform behavioral model like TIGUKAT could lead to a merger of these two disciplines. The definition of a uniform programming language is one possibility for bridging this gap in a seamless fashion.

Developing an object manager is another important area of research. An object manager design must address many related issues including object representation, physical partitioning of logical entities such as classes and their extents, clustering of complex objects, object caching, indexing, and how and when functions are bound to objects. The design is also affected by the underlying hardware architecture (e.g., uni-processor vs. multi-processor), and the available operating system services.

The issues related to object storage management are quite complex and require a significant amount of research. The advent of distributed object management complicates matters. Current approaches rely on simple client/server type architectures where there is (usually) only one server and many clients. With interoperability of autonomous heterogeneous systems becoming a big issue is database systems research, the development of an

OBMS with an architecture that is "open" to other systems is an active area of research and is a direction that this research could take. The uniformity of TIGUKAT could be of great help in this area since it may be possible to define other models as type and behavior extensions to the base model. This would give a seamless integration with these systems. The research opportunities along these lines are very promising.

In this thesis, signatures were defined and used as a partial semantics for behaviors. The development of a specification technique for defining the complete semantics of behaviors is left for future research. This is currently an open research topic with several candidate approaches being identified, including the use of denotational semantics and predicative specification techniques. Much research is required in this area. The extensible design of the TIGUKAT object model makes it primed and ready to incorporate any advancement in this area. Once defined, a full specification technique can easily be incorporated as part of the *B_semantics* behavior of type `T_behavior`.

The development of the TIGUKAT object model is more precise and formal than other object model definitions in order to clarify its properties and the semantics of its operations. However, an interesting and challenging exercise would be to define the features of the model using a formal mathematical theory of functions such as category theory or typed lambda calculus. This is sure to provide insight into the semantics of modeling objects and the effects on other database functionality such as view management, transaction management, distribution, and so on. It may also provide a theoretical foundation for object models in the same way as relational theory did for the relational model. An advancement in this area would clearly strengthen the object modeling approach and assert the limitations of its modeling capability.

# Bibliography


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

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

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

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

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

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

[ALP91] J. Andany, M. Léonard, and C. Palisser. Management of Schema Evolution in Databases. In *Proc. of the 17th Int'l Conf. on Very Large Databases*, pages 161–170, September 1991.

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

[Bac78] J. Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and it's Algebra of Programs. *Communications of the ACM*, 21(8):613–641, 1978.

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

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

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

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

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

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

[BCMS89] R. Backhouse, P. Chisholm, G. Malcolm, and Erik Saaman. Do-it-Yourself Type Theory. *Formal Aspects of Computing*, 1:19–84, 1989.

[BDK92] F.A. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building and Object-Oriented Database System: The Story of $O_2$*. Morgan Kaufmann Publishers, 1992.

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

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

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

[BKKK87] J. Banerjee, W. Kim, H-J. Kim, and H.F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 311–322, May 1987.

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

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

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

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

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

[Car86] L. Cardelli. A Polymorphic $\lambda$-calculus with Type:Type. Research Report 10, DEC Systems Research Center, May 1986.

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

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

[CDLR90] S. Cluet, C. Delobel, C. Lécluse, and P. Richard. RELOOP: An Algebra Based Query Language for an Object-Oriented Database System. *Data & Knowledge Engineering*, 5:333–352, 1990.

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

[Cha92] E.P.F. Chan. Containment and Minimization of Positive Conjunctive Queries in OODB's. In *Proc. of the 11th ACM Symposium on Principles of Database Systems*, pages 202–211, June 1992.

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

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

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

[Cod79] E.F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, 4(4):397–434, December 1979.

[Coi87] P. Cointe. Metaclasses are First Class: the ObjVlisp Model. In *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 156–167, October 1987.

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

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

[Dav90] K.C. Davis. *A Formal Foundation for Object-Oriented Algebraic Query Processing*. PhD thesis, University of Southwestern Louisiana, 1990.

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

[Dem81] R. Demolombe. Assigning Meaning to Ill-Defined Queries Expressed in Relational Calculus. In *Advances in Database Theory*. Plenum Press, 1981.

[Dem82] R. Demolombe. Syntactical Characterization of a Subset of Domain Independent Formulas. Technical report, ONERA-CERT, 1982.

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

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

[DiP69] R.A. DiPaola. The Recursive Unsolvability of the Decision Problem for the Class of Definite Formulas. *Journal of the ACM*, 16(2), 1969.

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

[DW92] U. Dayal and G. Wuu. A Uniform Approach to Processing Temporal Queries. In *Proc. of the 8th Int'l. Conf. on Data Engineering*, pages 407–418, August 1992.

[EMHJ93a] M. Escobar-Molano, R. Hull, and D. Jacobs. Safety and Translation of Calculus Queries with Scaler Functions (Extended Abstract). In *Proc. of the 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 253–264, May 1993.

[EMHJ93b] M. Escobar-Molano, R. Hull, and D. Jacobs. Safety and Translation of Calculus Queries with Scaler Functions. Technical report, Computer Science Department, University of Southern California, March 1993.

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

[Fag82] R. Fagin. Horn Clauses and Database Dependencies. *Journal of the ACM*, 29(4), 1982.

[FBC+87] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat,

T.A. Ryan, and M.C. Shan. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[Fer89] J. Ferber. Computational Reflection in Class Based Object-Oriented Languages. In *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 317–326, October 1989.

[FJ89] B. Foote and R.E. Johnson. Reflective Facilities in Smalltalk-80. In *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 327–335, October 1989.

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

[GÖ93] I. Goralwalla and M.T. Özsu. Temporal Extensions to a Uniform Behavioral Object Model. In *Proc. of the 12th Int'l Conf. on Entity–Relationship Approach*, pages 115–127, December 1993.

[Gor96] Iqbal Goralwalla. *A Temporal Active OBMS*. PhD thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1996. Forthcoming.

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

[GR89] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[GT87] A.V. Gelder and R.W. Topor. Safety and Correct Translation of Relational Calculus Formulas. In *Proc. of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, pages 313–327. ACM Press, March 1987.

[GT91] A.V. Gelder and R.W. Topor. Safety and Translation of Relational Calculus Queries. *ACM Transactions on Database Systems*, 16(2):235–278, June 1991.

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

[HHT75] P. Hall, P. Hitchcock, and S. Todd. An Algebra of Relations for Machine Computation. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pages 225–232, 1975.

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

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

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

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

[Ira93] B. Irani. Implementation Design and Development of the TIGUKAT Object Model. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1993. Available as University of Alberta Technical Report TR93-10.

[ISO92] ISO. *Information Technology – Database Languages – SQL.* International Organization for Standardization, 1992. ISO/IEC 9075:1992 (E).

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

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

[KBC+89] W. Kim, N. Ballou, H.T. Chou, J.F. Garza, and D. Woelk. Features of the ORION Object-Oriented Database System. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. Addison Wesley, 1989.

[KC86] S.N. Khoshafian and G.P. Copeland. Object Identity. In *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 406–416, September 1986.

[KC88] W. Kim and H-T. Chou. Versions of Schema for Object-Oriented Databases. In *Proc. of the 14th Int'l Conf. on Very Large Databases*, pages 148–159, 1988.

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

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

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

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

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

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

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

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

[KMSB89] M. Koubarakis, M. Mylopoulos, M. Stanley, and A. Borgida. Telos: Features and Formalization. Technical Report KRR-TR-89-4, University of Toronto, February 1989.

[KS92] W. Kafer and H. Schoning. Realizing a Temporal Complex-Object Data Model. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 266–275, June 1992.

[Kuh67] J.L. Kuhns. Answering Questions by Computer: A Logical Study. Technical Report RM-5428-PR, Rand Corp., 1967.

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

[LH90] B.S. Lerner and A.N. Habermann. Beyond Schema Evolution to Database Reorganization. In *ECOOP/OOPSLA '90 Proceedings*, pages 67–76, October 1990.

[Lip93] A. Lipka. The Design and Implementation of TIGUKAT User Languages. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1993. Available as University of Alberta Technical Report TR93-11.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[LM79] H. Levesque and J. Mylopoulos. A Procedural Semantics for Semantic Networks. In N. Findler, editor, *Associative Networks*. Academic Press, 1979.

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

[LR89b] C. Lécluse and P. Richard. The $O_2$ Database Programming Language. In *Proc. of the 15th Int'l Conf. on Very Large Databases*, pages 411–422, August 1989.

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

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

[Mae87] P. Maes. Concepts and Experiments in Computational Reflection. In *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 147–155, October 1987.

[Mai89] D. Maier. Why isn't there an Object-Oriented Data Model. In *Proc. of the IFIP 11th World Computer Conference*, August-September 1989.

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

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

[MBW80] J. Mylopoulos, P.A. Bernstein, and H.K.T. Wong. A Language Facility for Designing Database-Intensive Applications. *ACM Transactions on Database Systems*, 5(2):185–207, June 1980.

[McC68] J. McCarthy. Programs with Common Sense. In *Semantic Information Processing*, pages 403–418. MIT Press, 1968.

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

[ML82] P. Martin-Löf. Constructive Mathematics and Computer Programming. In *Proc. of the Sixth Int'l Conf. for Logic, Methodology and Philosophy of Science*, pages 153–175, 1982.

[MMWY92] H. Masuhara, S. Matsuoka, T. Wantanabe, and A. Yonezawa. Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently. In *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 127–144, October 1992.

[Muñ94] A. Muñoz. "an extensible query optimizer architecture for the tigukat object-base management system". Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1994. Available as University of Alberta Technical Report TR94-01.

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

[ND82] J.M. Nicolas and R. Demolombe. On the Stability of Relational Queries. Technical report, ONERA-CERT, 1982.

[Nie89] O. Nierstrasz. A Survey of Object-Oriented Concepts. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 3–21. Addison Wesley, 1989.

[NR87] G.T. Nguyen and D. Rieu. Expert Database Support for Consistent Dynamic Objects. In *Proc. of the 13th Int'l Conf. on Very Large Databases*, pages 493–500, September 1987.

[NR89] G.T. Nguyen and D. Rieu. Schema Evolution in Object-Oriented Database Systems. *Data & Knowledge Engineering*, 4:43–67, 1989.

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

[ÖPI+93] M.T. Özsu, R.J. Peters, B. Irani, A. Lipka, A. Muñoz, and D. Szafron. TIGUKAT Object Management System: Initial Design and Current Directions. In *Proc. of the Centre for Advanced Studies Conference (CASCON)*, pages 595–611, October 1993.

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

[Osb89] S.L. Osborn. The Role of Polymorphism in Schema Evolution in an OODB. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):310–317, September 1989.

[ÖSP94] M.T. Özsu, D.D. Straube, and R.J. Peters. Query Processing Issues in Object-Oriented Knowledge Base Systems. In F.E. Petry and L.M. Delcambre, editors, *Emerging Landscape of Intelligence in Database and Information Systems*. JAI Press, 1994. In press.

[OW89] G. Ozsoyoglu and H. Wang. A Relational Calculus with Set Operators, Its Safety, and Equivalent Graphical Languages. *IEEE Transactions on Software Engineering*, SE-15(9):1038–1052, September 1989.

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

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

[PLÖS93a] R.J. Peters, A. Lipka, M.T. Özsu, and D. Szafron. An Extensible Query Model and Its Languages for a Uniform Behavioral Object Management System. In *Proc. of the Second Int'l. Conf. on Information and Knowledge Management*, pages 403–412, November 1993. A full version of this paper is available as University of Alberta Technical Report TR93-01.

[PLÖS93b] R.J. Peters, A. Lipka, M.T. Özsu, and D. Szafron. The Query Model and Query Language of TIGUKAT. Technical Report TR93-01, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, June 1993.

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

[PÖ93] R.J. Peters and M.T. Özsu. Reflection in a Uniform Behavioral Object Model. In *Proc. of the 12th Int'l Conf. on Entity–Relationship Approach*, pages 37–49, December 1993.

[PS87] D.J. Penney and J. Stein. Class Modification in the GenStone Object-Oriented DBMS. In *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 111–117, October 1987.

[Rei92] R. Reiter. On Formalizing Database Updates: Preliminary Report. In *Proc. of the 3rd Int'l Conf. on Extending Database Technology*, pages 10–20, March 1992.

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

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

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

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

[RS91] E. Rose and A. Segev. TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints. In *Proc. of the 10th Int'l Conf. on Entity–Relationship Approach*, pages 205–229, October 1991.

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

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

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

[Shi81] D.W. Shipman. The Functional Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

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

[Sny90] A. Snyder. An Abstract Object Model for Object-Oriented Systems. Technical Report HPL-90-22, Hewlett Packard Labs, April 1990.

[SÖ90a] D.D. Straube and M.T. Özsu. Queries and Query Processing in Object-Oriented Database Systems. *ACM Transactions on Information Systems*, 8(4):387–430, October 1990.

[SÖ90b] D.D. Straube and M.T. Özsu. Type Consistency of Queries in an Object-Oriented Database System. In *ECOOP/OOPSLA '90 Proceedings*, pages 224–233, October 1990.

[Soo91] M.D. Soo. Bibliography on Temporal Databases. *ACM SIGMOD Record*, 20(1):14–23, 1991.

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

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

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

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

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

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

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

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

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

[Str91a] D.D. Straube. *Queries and Query Processing in Object-Oriented Database Systems*. PhD thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1991.

[Str91b] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1991. Second edition.

[SZ86] A.H. Skarra and S.B. Zdonik. The Management of Changing Types in an Object-Oriented Database. In *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 483–495, September 1986.

[SZ87] A.H. Skarra and S.B. Zdonik. Type Evolution in an Object-Oriented Database. In *Research Directions in Object-Oriented Programming*, pages 393–415. MIT Press, 1987.

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

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

[TCG+93] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, 1993.

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

[Ull82] J.D. Ullman. *Principles of Database Systems*. Computer Science Press, 1982. 2nd. Edition.

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

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988. Volume 1.

[Wan89] Y. Wand. A Proposal for a Formal Model of Objects. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 537–559. Addison Wesley, 1989.

[WBT92] D. Wells, J.A. Blakeley, and C.W. Thompson. Architecture of the Open Object-Oriented Database Management System. *IEEE Computer*, 25(10):74–82, October 1992.

[WBW88a] A. Wirfs-Brock and B. Wilkerson. An Overview of Modular Smalltalk. In *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 123–134, September 1988.

[WBW88b] A. Wirfs-Brock and B. Wilkerson. An Overview of Modular Smalltalk. In *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 123–134, October 1988.

[WBW89a] A. Wirfs-Brock and B. Wilkerson. Object-Oriented Design: A Responsibility-Driven Approach. In *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 71–75, October 1989.

[WBW89b] A. Wirfs-Brock and B. Wilkerson. Variables Limit Reusability. *Journal of Object-Oriented Programming*, 2(1):34–40, May/June 1989.

[WD92] G. Wuu and U. Dayal. A Uniform Model for Temporal Object-Oriented Databases. In *Proc. of the 8th Int'l. Conf. on Data Engineering*, pages 584–593, Tempe, USA, February 1992.

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

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

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

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

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

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

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

[ZW86] S. Zdonik and P. Wegner. Language and Methodology for Object-Oriented Database Environments. In *Proc. of the 19th Annual Hawaii Int'l Conference on System Sciences*, January 1986.

# Appendix A

# Primitive Type System

Table A.1 shows the signatures of the behaviors for the non-atomic types (except the container types). Table A.2 shows the signatures of the behaviors for the container types. Table A.3 shows the signatures of the behaviors for the atomic types. The receiver type of a behavior is excluded because the receiver must be an object of a type that is compatible with the type defining the behavior. The notation `T_collection`⟨`T`⟩ is used to define a collection type whose members are of type $T$. The type specifications for the behaviors are the *most general* types. Types for some of the behaviors are revised in the subtypes that inherit them. For example, the result type of *B_self* is always the type of the receiver object and the result type of *B_new* is always the membership type of the receiver class.

| Type | Signatures | |
|---|---|---|
| T_object | *B_self*: | T_object |
| | *B_mapsto*: | T_type → T_boolean |
| | *B_conformsTo*: | T_type → T_boolean |
| | *B_equal*: | T_object → T_boolean |
| | *B_notequal*: | T_object → T_boolean |
| | *B_drop*: | T_object |
| T_type | *B_interface*: | T_collection⟨T_behavior⟩ |
| | *B_native*: | T_collection⟨T_behavior⟩ |
| | *B_inherited*: | T_collection⟨T_behavior⟩ |
| | *B_specialize*: | T_type → T_boolean |
| | *B_subtype*: | T_type → T_boolean |
| | *B_subtypes*: | T_collection⟨T_type⟩ |
| | *B_supertypes*: | T_collection⟨T_type⟩ |
| | *B_sub-lattice*: | T_poset⟨T_type⟩ |
| | *B_super-lattice*: | T_poset⟨T_type⟩ |
| | *B_classof*: | T_class |
| | *B_addBehavior*: | T_behavior → T_function → T_type |
| | *B_dropBehavior*: | T_behavior → T_type |
| | *B_addSupertype*: | T_type → T_type |
| | *B_dropSupertype*: | T_type → T_type |
| T_behavior | *B_name*: | T_string |
| | *B_argTypes*: | T_type → T_list⟨T_type⟩ |
| | *B_resultType*: | T_type → T_type |
| | *B_description*: | T_string |
| | *B_semantics*: | T_object |
| | *B_associate*: | T_type → T_function → T_behavior |
| | *B_implementation*: | T_type → T_function |
| | *B_primitiveApply*: | T_object → T_object |
| | *B_apply*: | T_object → T_list → T_object |
| | *B_defines*: | T_poset⟨T_type⟩ |
| T_function | *B_name*: | T_string |
| | *B_argTypes*: | T_list⟨T_type⟩ |
| | *B_resultType*: | T_type |
| | *B_description*: | T_string |
| | *B_source*: | T_object |
| | *B_compile*: | T_function |
| | *B_primitiveExecute*: | T_object → T_object |
| | *B_executable*: | T_object |
| | *B_basicExecute*: | T_list → T_object |
| | *B_execute*: | T_list → T_object |

Table A.1: Behavior signatures of the non-atomic types of the primitive type system.

| Type | Signatures |
|---|---|
| T_collection | *B_memberType*: T_type |
| | *B_cardinality*: T_natural |
| | *B_elementOf*: T_object → T_boolean |
| | *B_insert*: T_object → T_collection |
| | *B_remove*: T_object → T_collection |
| | *B_containedBy*: T_collection → T_boolean |
| | *B_setEqual*: T_collection → T_boolean |
| | *B_isEmpty*: T_boolean |
| | *B_union*: T_collection → T_collection |
| | *B_difference*: T_collection → T_collection |
| | *B_intersect*: T_collection → T_collection |
| | *B_collapse*: T_collection |
| | *B_select*: T_string → T_list⟨T_collection⟩ → T_collection |
| | *B_project*: T_collection⟨T_behavior⟩ → T_collection |
| | *B_map*: T_string → T_list⟨T_collection⟩ → T_collection |
| | *B_product*: T_list⟨T_collection⟩ → T_collection |
| | *B_reduce*: T_collection⟨T_natural⟩ → T_collection |
| | *B_join*: T_string → T_list⟨T_collection⟩ → T_collection |
| | *B_genjoin*: T_string → T_string → T_list⟨T_collection⟩ → T_collection |
| T_bag | *B_occurrences*: T_object → T_natural |
| | *B_count*: T_natural |
| | *B_dropAll*: T_object → T_bag |
| | Behaviors from T_collection refined to preserve duplicates |
| T_poset | *B_ordered*: T_object → T_object → T_boolean |
| | *B_ordering*: T_function |
| | Behaviors from T_collection refined to preserve ordering |
| T_list | *B_insertAt*: T_object → T_natural → T_list |
| | *B_dropAt*: T_natural → T_list |
| | *B_append*: T_object → T_list |
| | *B_getAt*: T_natural → T_list |
| | *B_setAt*: T_object → T_natural → T_list |
| | *B_positions*: T_object → T_list⟨T_natural⟩ |
| | *B_currPosn*: T_natural |
| | *B_current*: T_object |
| | *B_first*: T_object |
| | *B_last*: T_object |
| | *B_next*: T_object |
| | *B_previous*: T_object |
| | *B_dropCurr*: T_list |
| | *B_outOfBounds*: T_boolean |
| | Behaviors refined to preserve duplicates and ordering |
| T_class | *B_new*: T_object |
| | *B_deepExtent*: T_collection |
| T_class-class | *B_new*: T_type → T_class |
| T_type-class | *B_new*: T_collection⟨T_type⟩ → T_collection⟨T_behavior⟩ → T_type |
| T_collection-class | *B_new*: T_type → T_collection |

Table A.2: Behavior signatures of the container types of the primitive type system.

| 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 |
| | *B_stringOf*: | T_string |
| T_string | *B_car*: | T_character |
| | *B_cdr*: | T_string |
| | *B_concat*: | T_string → T_string |
| T_real | *B_succ*: | T_real |
| | *B_pred*: | 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 | |

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

# Appendix B

# Behavior Definitions

In this appendix, we define the full behavioral specification of the primitive type system of TIGUKAT. The primitive type lattice is shown in Figure 2.1 on page 14. A summary of the behaviors is shown in Appendix A.

In the following specifications, we use variables $o$, $p$ and $q$ in examples as references to objects of various particular types. We use the dot notation $o.B\_something(a_1, \ldots, a_n)$ for the behavior application where $o$ is the receiver of behavior *B_something* that uses arguments $a_1$ through $a_n$. Behavior applications assume left associativity in the absence of qualifying parenthesis. That is, the following two behavior applications are equivalent:

$$o.B\_one(p).B\_two(q) \equiv (o.B\_one(p)).B\_two(q)$$

The type specifications are divided into the following components: the name of the type, its corresponding class, its supertypes, its subtypes, the native behaviors defined by the type and the derived behaviors defined by the type. Native behaviors are those which are introduced by the type (i.e., they are not inherited). Derived behaviors are those which are defined in terms of existing behaviors (i.e., they are not primitive to the type system, but are defined for brevity and ease of use). The implementations for some of the inherited behaviors are refined in the subtypes and their extended semantics are given in the refined behaviors section.

# T_object

Supertypes: *none*

Subtypes: T_type, T_collection, T_behavior, T_function, T_atomic

Native Behaviors:

**self**

*B_self*: T_object

Example: *o.B_self*

Symbol: $I_o$

Returns the receiver object *o*. This is the mathematical *identity* operation for objects.

**mapsto**

*B_mapsto*: T_type

Example: *o.B_mapsto*

Symbol: $o \mapsto$

Returns the type of the reciever object *o* (i.e., the most defined type). Every object in the system has a *mapsto* type.

**conformsTo**

*B_conformsTo*: T_type → T_boolean

Example: *o.B_conformsTo(p)*

Symbol: $o \rightsquigarrow p$

If the receiver *o* conforms to the type argument *p*, the object true is returned, otherwise false is returned.

**equal**

*B_equal*: T_object → T_boolean

Example: *o.B_equal(p)*

Symbol: $o = p$

If the receiver *o* is identity equal to the argument object *p*, the object true is returned, otherwise false is returned.

**drop**

*B_drop*: T_object

Example: *o.B_drop*

Symbol:

Drops the receiver object *o*, which "effectively deletes" the object. The object is dropped from its class and all collections in which it appears. All references to the object become invalid. When considering the temporality of the object model, the lifespan of the object in its class, and all collections, is termintated.

Derived Behaviors:

**notequal**

*B_notequal*: T_object → T_boolean

Example: *o.B_notequal(p)*

Symbol: $o \neq p$

Derivation: $\neg(o = p)$

This is the complement of *B_equal*.

# T_type

Supertypes: `T_object`

Subtypes: *none*

Native Behaviors:

**classof**

*B_classof*: `T_class`

Example: *o.B_classof*

Symbol: $\mathcal{C}_o$

Returns the class object that has been associated with the receiver *o*. Types are associated with at most one class. For those types not associated with a class, the object undefined is returned.

**native**

*B_native*: `T_collection⟨T_behavior⟩`

Example: *o.B_native*

Symbol:

Returns the set of behaviors that are defined by the receiver *o* and not defined by any supertypes of *o*. The set is empty if *o* doesn't define any native behaviors.

**inherited**

*B_inherited*: `T_collection⟨T_behavior⟩`

Example: *o.B_inherited*

Symbol:

Returns the collection of behaviors that are inherited by the receiver *o*. This set is a superset of the interface set of `T_object`.

**subtypes**

*B_subtypes*: `T_collection⟨T_type⟩`

Example: *o.B_subtypes*

Symbol:

Returns the set of type objects that are a direct subtype of the receiver *o*. The result set does not include the object *o* itself. For the types that do not have any subtypes, the empty set is returned.

**supertypes**

*B_supertypes*: `T_collection⟨T_type⟩`

Example: *o.B_supertypes*

Symbol:

Returns the set of type objects that are a direct supertype of the receiver *o*. The result set does not include the object *o* itself. Every type object except `T_object` has a non-empty supertype set. The supertype set for `T_object` is the empty set.

**addBehavior**

*B_addBehavior*: `T_behavior` → `T_function` → `T_type`

Example: *o.B_addBehavior*$(p, q)$

Symbol:

Adds the behavior object *p* as a native behavior of the receiver type *o*. The operation is rejected if *o* already defines *p*. If *o* has an associated class or if any subtype of *o* has an associated class and does not already define *p*, then a function *q* must be given as the implementation of the *p* in these types.

**dropBehavior**

*B_dropBehavior*: `T_behavior` → `T_type`

Example: *o.B_dropBehavior*$(p)$

Symbol:

Drops the native behavior $p$ from the receiver type $o$. The operation is rejected if $p$ is not natively defined on $o$. The native definition of $p$ is propogated to the subtypes, unless inherited from some type other than $o$.

**addSupertype**

*B_addSupertype*: `T_type` → `T_type`
Example: $o.B\_addSupertype(p)$
Symbol:

Adds the type argument $p$ as a supertype of $o$. The operation is rejected if it introduces a cycle into the type lattice or if $p$ is already an element of the super-lattice of $o$.

**dropSupertype**

*B_dropSupertype*: `T_type` → `T_type`
Example: $o.B\_dropSupertype(p)$
Symbol:

Drops the type argument $p$ as a supertype of $o$. Type $o$ is relinked to the supertypes of $p$ and type $p$ is relinked to the subtypes of $o$.

Derived Behaviors:

**interface**

*B_interface*: `T_collection`⟨`T_behavior`⟩
Example: $o.B\_interface$
Symbol:
Derivation: $o.B\_native \cup o.B\_inherited$

Returns the set of behavior objects resulting from the union of the native and inherited behaviors of receiver $o$. This set is a superset of the interface set of `T_object`.

**super-lattice**

*B_super-lattice*: `T_poset`⟨`T_type`⟩
Example: $o.B\_super - lattice$
Symbol:
Derivation: Derived by recursively applying *B_supertypes* until `T_object` is reached, partially ordering the intermediate results, and adding the receiver type object $o$.

Returns the set of all type objects, partially ordered by $\preceq$, that are supertypes of the receiver $o$. The result set includes the type object $o$ itself. The result lattice has `T_object` as the root and $o$ as the base. Every type object has a non-empty super-lattice.

**sub-lattice**

*B_sub-lattice*: `T_poset`⟨`T_type`⟩
Example: $o.B\_sub - lattice$
Symbol:
Derivation: Derived by recursively applying *B_subtypes* until `T_null` is reached, partially ordering the intermediate results, and adding the receiver type object $o$ as the root.

Returns the set of all type objects, partially ordered by $\preceq$, which are subtypes of the receiver $o$. The result set includes the type object $o$ itself. The result lattice has $o$ as the root and `T_null` as the base. Every type object has a non-empty subtype-lattice.

**specialize**

*B_specialize*: `T_type` → `T_boolean`
Example: $o.B\_specialize(p)$

Symbol: $o \sqsubseteq p$
Derivation: $p.B\_interface \subseteq o.B\_interface$

| |
|---|
| Returns true if the receiver $o$ specializes of the type argument object $p$, false otherwise. |

**subtype**

$B\_subtype$: `T_type` $\rightarrow$ `T_boolean`
Example: $o.B\_subtype(p)$
Symbol: $o \preceq p$
Derivation: $o \in p.B\_sub - lattice$

| |
|---|
| Returns true if the receiver $o$ is a subtype of the type argument object $p$, false otherwise. |

# T_behavior

Supertypes: `T_object`
Subtypes: *none*
Native Behaviors:

**name**
*B_name*: `T_string`
Example: *o.B_name*
Symbol:
Returns the signature name of the receiver *o*.

**argTypes**
*B_argTypes*: `T_type` → `T_list`⟨`T_type`⟩
Example: *o.B_argTypes*(*p*)
Symbol:
Returns the list of types that are the argument types of the signature for the behavior *o* in the type *p*.

**resultType**
*B_resultType*: `T_type` → `T_type`
Example: *o.B_resultType*(*p*)
Symbol:
Returns the type that is the result type of the signature for the behavior *o* in the type *p*.

**description**
*B_description*: `T_string`
Example: *o.B_description*
Symbol:
Returns a short description of behavior *o*.

**semantics**
*B_semantics*: `T_object`
Example: *o.B_semantics*
Symbol: $[\![o]\!]$
Returns the full semantics of the behavior *o*.

**associate**
*B_associate*: `T_type` → `T_function` → `T_behavior`
Example: *o.B_associate*(*p*, *q*)
Symbol:
Associates the function object of the argument *q* with the behavior *o* for the given type object *p*. The behavior has the side-effect of modifying the behavior *o* so that it executes the associated function *q* when applied to an object of type *p*.

**implementation**
*B_implementation*: `T_type` → `T_function`
Example: *o.B_implementation*(*p*)
Symbol:
Returns the function object associated with the behavior *o* for the argument type object *p*.

**primitiveApply**
*B_primitiveApply*: `T_object` → `T_object`
Example: *o.B_primitiveApply*(*p*)
Symbol:
Applies the behavior object *o* to the argument object *p*. One of the requirements is that the type of *p* must define behavior *o* as part of its interface.

**defines**
*B_defines*: `T_poset`⟨`T_type`⟩
Example: *o.B_defines*

Symbol:

| |
|---|
| Returns the partially ordered set of type objects (i.e., lattice) that define the behavior $o$ as part of their interface. |

Derived Behaviors:

**apply**

$B\_apply$: `T_object` $\rightarrow$ `T_list` $\rightarrow$ `T_object`

Example: $o.B\_apply(p, q)$

Symbol:

Derivation: If the argument list $q$ is empty, the apply works the same as the primitive apply. If there are arguments, they are passed directly to the execution of the function associated with this behavior.

| |
|---|
| Applies the behavior object $o$ to the object $p$ using the objects in the list $q$ as arguments. The requirements are that the type of $p$ must define behavior $o$ as part of its interface and the type of the objects in $q$ must conform to the argument types defined by the signature of behavior $o$ in the type of $p$. |

# T_function

Supertypes: `T_object`
Subtypes: *none*
Native Behaviors:

**name**
*B_name*: `T_string`
Example: *o.B_name*
Symbol:
Returns the name of the function object *o*.

**argTypes**
*B_argTypes*: `T_list`⟨`T_type`⟩
Example: *o.B_argTypes*
Symbol:
Returns a list of types that denote the types and ordering of the argument objects for the function *o*.

**resultType**
*B_resultType*: `T_type`
Example: *o.B_resultType*
Symbol:
Returns the result type of the function *o*.

**description**
*B_description*: `T_string`
Example: *o.B_description*
Symbol:
Returns a description of the function object *o*.

**source**
*B_source*: `T_string`
Example: *B_source*(*o*)
Symbol:
Returns the source code of the function *o*.

**compile**
*B_compile*: `T_function`
Example: *o.B_compile*
Symbol:
Compiles the function *o* and produces an executable that is returned by *B_executable* below.

**primtiveExecute**
*B_primtiveExecute*: `T_object` → `T_object`
Example: *o.B_primitiveExecute*(*p*)
Symbol:
Executes the function *o* using the object *p* as an argument and returns a result object. This requires that the argument *p* is compatible with the argument type of the function *o*.

**executable**
*B_executable*: `T_object`
Example: *o.B_executable*
Symbol:
Returns the executable of the function *o*.

Derived Behaviors:

**basicExecute**
*B_basicExecute*: `T_list` → `T_object`
Example: *o.B_basicExecute*(*p*)
Symbol:
Derivation: Function currying of the *B_primitiveExecute* is abstracted as a list of arguments.

| | |
|---|---|
| | Executes the function $o$ using the list of objects in $p$ as arguments and returns a result object. This requires that the list of arguments in $p$ is compatible with the argument type list for the function $o$. |
| **execute** | *B_execute*: `T_list` $\rightarrow$ `T_object`<br>Example: $o.B\_execute(p)$<br>Symbol:<br>Derivation: Function currying is abstracted as a list of arguments. |
| | For this general function type the behavior performs the same operation as *B_basicExecute* above. |

# T_collection

Supertypes: `T_object`

Subtypes: `T_class`

Native Behaviors:

**memberType**

*B_memberType*: `T_type`

Example: *o.B_memberType*

Symbol: $\Lambda_o$

Returns the type of the members in the collection *o*. Every collection is associated with exactly one member type, but a type object may be associated with many collections.

**cardinality**

*B_cardinality*: `T_natural`

Example: *o.B_cardinality*

Symbol: $|o|$

Returns the number of elements in collection *o*.

**elementOf**

*B_elementOf*: `T_object` → `T_boolean`

Example: *o.B_elementOf*(*p*)

Symbol: $p \in o$

Returns `true` if the object *p* is a member of collection *o*, `false` otherwise.

**insert**

*B_insert*: `T_object` → `T_collection`

Example: *o.B_insert*(*p*)

Symbol:

Adds the object *p* to the collection *o* if *p* is not already a member of *o*. This cannot be defined in terms of union since union returns a new collection and this behavior modifies the extent of *o*.

**remove**

*B_remove*: `T_object` → `T_collection`

Example: *o.B_remove*(*p*)

Symbol:

Removes the object *p* from the collection *o*. This cannot be defined in terms of difference since difference returns a new collection and this behavior modifies the extent of *o*.

**union**

*B_union*: `T_collection` → `T_collection`

Example: *o.B_union*(*p*)

Symbol: $o \cup p$

Returns the set union of collections *o* and *p*.

**difference**

*B_difference*: `T_collection` → `T_collection`

Example: *o.B_difference*(*p*)

Symbol: $o - p$

Returns the set difference of collections *o* and *p*.

**intersect**

*B_intersect*: `T_collection` → `T_collection`

Example: *o.B_intersect*(*p*)

Symbol: $o \cap p$

Returns the set intersection of collections *o* and *p*.

**collapse**

*B_collapse*: `T_collection`

Example: *o.B_collapse*

Symbol: $o \Downarrow$

| | |
|---|---|
| | Receiver $o$ is a collection of collections. The result is to take the extended union of the element collections in $o$. |
| **select** | *B_select*: `T_string` $\rightarrow$ `T_list`$\langle$`T_collection`$\rangle$ $\rightarrow$ `T_collection`<br>Example: $o.B\_select(p, q)$<br>Symbol: $o\ \sigma_p\ q$ |
| | The argument $p$ is a predicate over the collections in $q$ and the receiver collection $o$. The result is to return objects from $o$ that satisfy the predicate $p$. |
| **project** | *B_project*: `T_collection`$\langle$`T_behavior`$\rangle$ $\rightarrow$ `T_collection`<br>Example: $o.B\_project(p)$<br>Symbol: $o\ \Pi_p$ |
| | The argument $p$ is a collection of behaviors defined by the membership type of $o$. The result is a new collection containing all the objects of $o$, but with a membership type that only defines the behaviors in $p$, plus those defined on `T_object`. In other words, the operator projects over the behaviors in $p$. |
| **map** | *B_map*: `T_string` $\rightarrow$ `T_list`$\langle$`T_collection`$\rangle$ $\rightarrow$ `T_collection`<br>Example: $o.B\_map(p, q)$<br>Symbol: $o \gg_p\ q$ |
| | The argument $p$ is a mop function over the collections in $q$ and the receiver collection $o$. The result consists of the objects returned by applying the mop function $p$ to the objects in $o$ using the objects in the collections of $q$ as arguments. |
| **product** | *B_product*: `T_list`$\langle$`T_collection`$\rangle$ $\rightarrow$ `T_collection`<br>Example: $o.B\_product(p)$<br>Symbol: $o \times p_1 \times \cdots \times p_n$ |
| | The argument $p$ is a list of $n$ collections. The result collection contains product objects drawn from each permutation of objects in $o$ and objects in the collections of $p$. The first component is an object from $o$, the second is an object from the first collection in $p$ (i.e., $p_1$), the third from the second collection in $p$ (i.e., $p_2$), and so on. |
| Derived Behaviors: | |
| **containedBy** | *B_containedBy*: `T_collection` $\rightarrow$ `T_boolean`<br>Example: $o.B\_containedBy(p)$<br>Symbol: $o \subseteq p$<br>Derivation: $\forall x(x \in o \longrightarrow x \in p)$ |
| | Returns **true** if all elements in collection $o$ are also members of collection $p$, **false** otherwise. |
| **isEmpty** | *B_isEmpty*: `T_boolean`<br>Example: $o.B\_isEmpty$<br>Symbol:<br>Derivation: $o =$ **C_collection**.*B_new* |

Returns true if $o$ is an empty collection. The application of *B_new* in the derivation ensures that a new empty collection is created. This is only done to demonstrate one derivation of the behavior. Any known empty collection would suffice.

**setEqual**

*B_setEqual*: `T_collection` → `T_boolean`
Example: $o.B\_setEqual(p)$
Symbol: $o =_{\{\}} p$
Derivation: $o \subseteq p \wedge p \subseteq o$

Returns true if collections $o$ and $p$ contain the same elements, false otherwise.

**reduce**

*B_reduce*: `T_collection`⟨`T_natural`⟩ → `T_collection`
Example: $o.B\_reduce(p)$
Symbol: $o\triangle_p$
Derivation: Derived in terms of *B_map* as shown in Chapter 3

The receiver $o$ is a collection of product objects and the argument $p$ is a list of naturals denoting components of the product objects. The result is the objects of $o$ with the components specified by $p$ removed.

**join**

*B_join*: `T_string` → `T_list`⟨`T_collection`⟩ → `T_collection`
Example: $o.B\_join(p, q)$
Symbol: $o \bowtie_p q$
Derivation: Derived in terms of *B_product* and *B_select* as shown in Chapter 3

The argument $p$ is a predicate over the collections in $q$ and the receiver collection $o$. The result is to return product objects formed (i.e., joined) from the objects in $o$ and the objects in the collections of $q$ such that the predicate $p$ is satisfied by the component objects.

**genjoin**

*B_genjoin*: `T_string` → `T_string` → `T_list`⟨`T_collection`⟩ → `T_collection`
Example: $o.B\_genjoin(g, p, q)$
Symbol: $o\ \gamma_p^g\ q$
Derivation: Derived in terms of *B_map* as shown in Chapter 3

The argument $g$ is the variable to be generated and $p$ is a generating atom (i.e, mop function) that generates $g$. The mop function $p$ operates over the collections in $q$ and the receiver collection $o$. The result is to return product objects formed (i.e., joined) from the objects in $o$ and the objects in the collections of $q$, and to append to each product object the result of applying the generating atom to the corresponding component objects. In other words, new objects are generated and joined to each permutation of product objects formed from the objects in $o$ and the objects in the collections of $q$.

# T_bag

Supertypes: `T_collection`

Subtypes: `T_list`

Native Behaviors:

**occurrences**

*B_occurrences*: `T_object` → `T_natural`

Example: $o.B\_occurrences(p)$

Symbol: $\delta_o p$

| Returns the number of times that argument object $p$ appears in the bag $o$. |
|---|

**count**

*B_count*: `T_natural`

Example: $o.B\_count$

Symbol:

| Returns the total number of elements contained within the bag $o$. Each duplicate is counted separately. |
|---|

Derived Behaviors:

**dropAll**

*B_dropAll*: `T_object` → `T_bag`

Example: $o.B\_dropAll(p)$

Symbol:

Derivation: for all $p \in o$, $o.B\_drop(p)$

| Drops all occurrences of $p$ in $o$. |
|---|

Refined Behaviors:

**cardinality**

*B_cardinality*: `T_natural`

Example: $o.B\_cardinality$

Symbol: $|o|$

| Returns the cardinality of the bag $o$. The cardinality of a bag does not take duplicates into account. Cardinality returns the total number of unique elements in a bag. |
|---|

# T_poset

Supertypes: `T_collection`

Subtypes: `T_list`

Native Behaviors:

**ordered**

*B_ordered*: `T_object` → `T_object` → `T_boolean`

Example: $o.B\_ordered(p, q)$

Symbol: $p \leq_o q$

This behavior uses the ordering relation defined on the receiver poset $o$ that returns **true** if the argument object $p$ occurs before the argument object $q$ in the poset $o$ or if $p$ and $q$ are equal in the poset. The behavior returns **false** if $p$ does not occur before $q$ or is not equal to $q$. The behavior returns **unknown** if no ordering of $p$ and $q$ is known.

**ordering**

*B_ordering*: `T_function`

Example: $o.B\_ordering$

Symbol: $\leq_o$

Returns the ordering relation defined on the receiver poset $o$. An ordering relation is a function of the form `T_object` → `T_object` → `T_boolean` and returns **true** if the two argument objects are ordered, **false** if they are not, or **unknown** if no ordering of the arguments is known.

The behaviors inherited from `T_collection` are refined to always maintain the ordering of objects in a poset. The behaviors that returned a collection are refined to return a poset.

# T_list

Supertypes: T_bag, T_poset
Subtypes: *none*
Native Behaviors:

**insertAt**
*B_insertAt*: T_object → T_natural → T_list
Example: $o.B\_insertAt(p, q)$
Symbol:
Inserts the object $p$ into the list $o$ at position $q$.

**dropAt**
*B_dropAt*: T_natural → T_list
Example: $o.B\_dropAt(p)$
Symbol:
Drops the object at position $p$ from list $o$.

**append**
*B_append*: T_object → T_list
Example: $o.B\_append(p)$
Symbol:
Append the object $p$ to the end of list $o$.

**getAt**
*B_getAt*: T_natural → T_list
Example: $o.B\_getAt(p)$
Symbol:
Return the object at position $p$ in list $o$.

**setAt**
*B_setAt*: T_object → T_natural → T_list
Example: $o.B\_setAt(p, q)$
Symbol:
Set position $q$ in list $o$ to the object $p$.

**positions**
*B_positions*: T_object → T_list⟨T_natural⟩
Example: $o.B\_positions(p)$
Symbol:
Return a list containing the positions where object $p$ occurs in list $o$.

**currPosn**
*B_currPosn*: T_natural
Example: $o.B\_currPosn$
Symbol:
Returns the current list position for list processing.

**current**
*B_current*: T_object
Example: $o.B\_current$
Symbol:
Returns the object at the current list position.

**first**
*B_first*: T_object
Example: $o.B\_first$
Symbol:
Returns the first object in the list and sets the current list position to the beginning of the list.

**last**
*B_last*: T_object
Example: $o.B\_last$
Symbol:

| | |
|---|---|
| | Returns the last object in the list and sets the current list position to the end of the list. |
| **next** | *B_next*: `T_object`<br>Example: *o.B_next*<br>Symbol: |
| | Returns the object that follows the current object and increments the current list position. If the behavior proceeds past the end of the list, an "out of bounds" condition is raised. |
| **previous** | *B_previous*: `T_object`<br>Example: *o.B_previous*<br>Symbol: |
| | Returns the object that precedes the current object and decrements the current list position. If the behavior proceeds past the beginning of the list, an "out of bounds" condition is raised. |
| **dropCurr** | *B_dropCurr*: `T_list`<br>Example: *o.B_dropCurr*<br>Symbol: |
| | Drop the current object of list *o*. |
| **outOfBounds** | *B_outOfBounds*: `T_boolean`<br>Example: *o.B_outOfBounds*<br>Symbol: |
| | Returns `true` if an "out of bounds" condition has been raised, `false` otherwise. |

The behaviors inherited from `T_poset` and `T_bag` are refined to maintain the ordering and duplication of objects in a list.

# T_class

Supertypes: `T_collection`

Subtypes: `T_class-class`, `T_type-class`, `T_collection-class`

Native Behaviors:

**new**

*B_new*: `T_object`

Example: *o.B_new*

Symbol:

| Creates and returns a new object with a unique identity from all other objects in the system. The object is created in accordance with the member type of the class *o* and becomes part of the shallow extent of this class. This has the effect of also including the object in the deep extent of the class. |
|---|

Derived Behaviors:

**deepExtent**

*B_deepExtent*: `T_collection`

Example: *o.B_deepExtent*

Symbol: $o^*$

Derivation: This is the union of the class with all its subclasses

| Returns a collection containing the objects in the deep extent of class *o*. The deep extent of a class consists of the objects created using the associated member type of the class or any of its subtypes. |
|---|

Refined Behaviors:

**memberType**

*B_memberType*: `T_type`

Example: *o.B_memberType*

Symbol: $\Lambda_o$

| Returns the type object associated with the class *o*. Every class is associated with exactly one type and every type is associated with at most one class. |
|---|

# T_class-class

Supertypes: `T_class`

Subtypes: *none*

Refined Behaviors:

**new**

*B_new*: `T_type` → `T_class`

Example: $o.B\_new(p)$

Symbol:

*B_new* is refined from `T_class` to create a new instance of the class $o$ and associate the new instance with the type object $p$. If the type $p$ does not exist, or it is already associated with another class object, an error condition is raised. The type of the resulting instance is the type associated with the receiver of the behavior. The receiver $o$ is a class object that manages other class objects.

# T_type-class

Supertypes: `T_class`

Subtypes: *none*

Refined Behaviors:

**new** *B_new*: `T_collection`⟨`T_type`⟩ → `T_collection`⟨`T_behavior`⟩ → `T_type`

Example: $o.B\_new(p, q)$

Symbol:

*B_new* is refined from `T_class` to create a new instance of the class $o$. The class $o$ manages type objects, thus a new type is created. The argument $p$ represents a non-empty collection of supertypes for the newly created type. The newly created type inherits all the behaviors of these supertypes. The argument $q$ is a collection (possibly empty) of behaviors to be defined natively on the newly created type. The type of the resulting instance is the member type associated with the receiver $o$.

# T_collection-class

Supertypes: `T_class`

Subtypes: *none*

Refined Behaviors:

**new** *B_new*: `T_type` → `T_collection`

Example: $o.B\_new(p)$

Symbol:

*B_new* is refined from `T_class` to create a new instance of the class $o$ and associate the new instance with the type object denoted by the argument $p$. If the type argument $p$ is omitted, the type of the collection is derived and maintained by the system according to the member objects of the collection. If the type object $p$ is given and does not exist, an error condition is raised. The type of the resulting instance is the member type associated with the receiver $o$. The argument $o$ is a class object which manages collection objects.

# T_atomic

Supertypes: T_object
Subtypes: T_boolean, T_character, T_string, T_real

# T_boolean

Supertypes: `T_atomic`

Subtypes: *none*

Native Behaviors:

**not**

*B_not*: `T_boolean`

Example: $o.B\_not$

Symbol: $\neg o$

| Returns the boolean complement of the receiver $o$. |
|---|

**or**

*B_or*: `T_boolean` → `T_boolean`

Example: $o.B\_or(p)$

Symbol: $o \vee p$

| Returns the boolean **OR** of the receiver $o$ and argument $p$. |
|---|

**if**

*B_if*: `T_object` → `T_object` → `T_object`

Example: $o.B\_if(p, q)$

Symbol: $o \longrightarrow p \Box q$

| If the receiver $o$ is `true`, the argument $p$ is returned, otherwise the argument $q$ is returned. |
|---|

Derived Behaviors:

**and**

*B_and*: `T_boolean` → `T_boolean`

Example: $o.B\_and(p)$

Symbol: $o \wedge p$

Derivation: $\neg(\neg o \vee \neg p)$

| Returns the boolean **AND** of the receiver $o$ and argument $p$. |
|---|

**xor**

*B_xor*: `T_boolean` → `T_boolean`

Example: $o.B\_xor(p)$

Symbol: $o \oplus p$

Derivation: $(o \wedge \neg p) \vee (\neg o \wedge p)$

| Returns the **EXCLUSIVE OR** of the receiver $o$ and arugment $p$. |
|---|

# T_character

Supertypes: `T_atomic`

Subtypes: *none*

Native Behaviors:

**ord**

*B_ord*: `T_natural`

Example: *o.B_ord*

Symbol:

| Returns the ordinal value of the receiver character *o*. |
|---|

**stringOf**

*B_stringOf*: `T_string`

Example: *o.B_stringOf*

Symbol:

| Returns the string representation of the receiver character *o*. |
|---|

# T_string

Supertypes: `T_atomic`

Subtypes: *none*

Native Behaviors:

**car**

*B_car*: `T_character`

Example: *o.B_car*

Symbol:

| Returns the first character of the string *o*. If *o* is the empty string, null is returned. |
|---|

**cdr**

*B_cdr*: `T_string`

Example: *o.B_cdr*

Symbol:

| Returns the remainder of the string *o* with the first character removed. If *o* is the empty string, null is returned. The resulting string is always different from the receiver string. |
|---|

**concat**

*B_concat*: `T_string` → `T_string`

Example: *o.B_concat*(*p*)

Symbol: *o* || *p*

| Returns the concatenation of the receiver string *o* and argument string *p*. If one of the strings is the empty string, the other string is returned. The result string is always different from the receiver and argument strings unless one of them is the empty string. |
|---|

Derived Behaviors:

**substr**

*B_substr*: `T_natural` → `T_natural` → `T_string`

Example: *o.B_substr*(*p*, *q*)

Symbol:

Derivation: Apply *B_cdr* *p* number of times to skip over the first *p* characters of the string. Then, beginning with an empty string, apply *B_car* *q* number of times and *B_concat* the string representation of the resulting characters to the result.

| Returns the substring of *o* starting at position *p* and continuing for *q* number of characters. The first character is at position zero. |
|---|

Other string related behaviors can be easily defined in terms of the primitve ones.

# T_real

Supertypes: `T_atomic`
Subtypes: `T_integer`
Native Behaviors:

**succ**
*B_succ*: `T_real`
Example: *o.B_succ*
Symbol:

| Returns the floating point number that follows *o*. The successor is rounded up to the precision of a particular system. |
|---|

**pred**
*B_pred*: `T_real`
Example: *o.B_pred*
Symbol:

| Returns the floating point number that precedes *o*. The predecessor is truncated to the precision of a particular system. |
|---|

**add**
*B_add*: `T_real` → `T_real`
Example: *o.B_add(p)*
Symbol: $o + p$

| Returns the floating point addition of the two reals *o* and *p*. |
|---|

**subtract**
*B_subtract*: `T_real` → `T_real`
Example: *o.B_subtract(p)*
Symbol: $o - p$

| Returns the floating point subtraction of the two reals *o* and *p*. |
|---|

**multiply**
*B_multiply*: `T_real` → `T_real`
Example: *o.B_multiply(p)*
Symbol: $o * p$

| Returns the floating point multiplication of the two reals *o* and *p*. |
|---|

**divide**
*B_divide*: `T_real` → `T_real`
Example: *o.B_divide(p)*
Symbol: $o \div p$

| Returns the floating point division of the two reals *o* and *p*. |
|---|

**trunc**
*B_trunc*: `T_integer`
Example: *o.B_trunc*
Symbol:

| Returns the integer resulting from the truncation of the fractional part of the real *o*. |
|---|

**round**
*B_round*: `T_integer`
Example: *o.B_round*
Symbol:

| Returns the integer resulting from rounding the fractional part of the real *o*. |
|---|

**lessThan**
*B_lessThan*: `T_real` → `T_boolean`
Example: *o.B_lessThan(p)*
Symbol: $o < p$

| Returns true if the *o* is less than the *p* and false otherwise. This behavior defines a total ordering on the domain of reals. |
|---|

Derived Behaviors:

**lessThanEQ** $B\_lessThanEQ$: `T_real` $\rightarrow$ `T_boolean`

Example: $o.B\_lessThanEQ(p)$

Symbol: $o \leq p$

Derivation: $(o < p) \vee (o = p)$

| Returns **true** if the real $o$ is less than or equal to the real $p$, **false** otherwise. |
|---|

**greaterThan** $B\_greaterThan$: `T_real` $\rightarrow$ `T_boolean`

Example: $o.B\_greaterThan(p)$

Symbol: $o > p$

Derivation: $\neg((o < p) \vee (o = p))$

| Returns **true** if the real $o$ is greater than the real $p$, **false** otherwise. |
|---|

**greaterThanEQ** $B\_greaterThanEQ$: `T_real` $\rightarrow$ `T_boolean`

Example: $o.B\_greaterThanEQ(p)$

Symbol: $o \geq p$

Derivation: $\neg(o < p)$

| Returns **true** if the real $o$ is greater than or equal to the real $p$, **false** otherwise. |
|---|

# T_integer

Supertypes: `T_real`

Subtypes: `T_natural`

Native Behaviors:

The behaviors inherited from `T_real` are refined to produce integer results when both of the arguments are integer objects.

# T_natural

Supertypes: `T_integer`

Subtypes: *none*

Native Behaviors:

The behaviors inherited from `T_integer` are refined to produce results of type `T_natural` when both of the arguments are naturals and results of type `T_integer` when the argument is an integer.

# Appendix C

# Object Model Analysis

In this chapter, a brief discussion of TIGUKAT's conformance with the guidelines outlined in the two manifesto papers [ABD+89] and [SRL+90] is given. Futhermore, TIGUKAT's compliance with the recommendations n [FKMT91] is considered. These references are slightly outdated in terms of current object technology. Nevertheless, they contain many core concepts important to the development of an object model.

## C.1 Conformance to Manifestos

Following [MB90], the discussion is organized along the structure of [ABD+89] and refers to [SRL+90] periodically. The characteristics of an OBMS are separated in [ABD+89] into *mandatory* and *optional* sections. There are also a number of features that the authors were unable to agree on a classification at the time. Furthermore, they specify several open design decisions that they thought were best handled by the model designer because no consensus had been reached on them by the scientific community and it was uncertain at the time which of the alternatives were more or less object-oriented. Each of their issues are considered in turn.

### C.1.1 Mandatory requirements

**Complex objects.** The TIGUKAT model supports complex objects. TIGUKAT is functional in that objects (and their properties) are only accessible through the applications of behaviors. The model is uniform in that everything is an object including behaviors and their implementations. Since behaviors are mappings from objects into other objects, every object may be considered as a complex object. The TIGUKAT model does not explicitly incorporate the notion of constructors. Instead, a type that exhibits the behavior of a desired constructor is defined, which is uniform. For example, the TIGUKAT model defines an atomic *integer* type whose instances are integers and whose behaviors are the typical operations on integers.

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

**Encapsulation.** The TIGUKAT model fully encapsulates the state of objects whose only access is through a set of public behaviors defined on its type. Objects may be viewed as instances of abstract data types that define an interface for the objects.

**Types and Classes.** The notions of type and class are separated in TIGUKAT and a different semantics is attached to each one. A type is defined as a specification tool (template) for objects, whereas a class as a grouping construct for instances of a type. A class has a number of restrictions defined on it that imposes a subset inclusion structure on the groupings of objects. Futhermore, TIGUKAT defines collections, not mentioned in [ABD+89], that serve as a general user-specified grouping mechanism. We feel that the clear separation of these concepts clarifies their roles in an object model.

**Class or Type Hierarchies.** First of all, the term "hierarchy" is inappropriate here since not only strict hierarchies are supported, but general directed acyclic graph structures such as lattices as well. The TIGUKAT model defines two categories of "inheritance." The first refers to the inheritance of behavior specifications on types (called *behavioral inheritance*) and is defined by subtyping relationships on types. The second is an inheritance mechanism for the methods (functions in TIGUKAT) that implement behaviors (called *implementation inheritance*). We are careful to attach individual semantics to each one. The reason being that behaviors and functions represent two different aspects of a type and their inheritance semantics is orthogonal.

**Overriding, overloading and late binding.** These notions are supported in TIGUKAT through the separation of the *behavioral* and *implementation* inheritance hierarchies. The semantics of behaviors are separated from their possible implementations (i.e., functions). This means that behaviors may be defined on many types (i.e., *overloading*) and that the implementation of the behavior may be different (redefined) for each type (i.e., *overriding*). *Late binding* is more a language support issue and is not part of the formal model definition. Whether an implementation is bound to a behavior for a particular application is up to a compiler for a particular access language. In general, late binding support is a necessity and is a good idea. However, in certain cases a compiler may choose to bind implementations to behaviors at compile time for efficiency reasons.

**Computational completeness.** Since the TIGUKAT model is functional and uniform, any computable function can be defined and attached to any behavior of a type in the system. Furthermore, a database programming language for the model is being developed. We feel that this satisfies the computational completeness requirement.

**Extensibility.** The TIGUKAT model is fully extensible through the operations provided by the meta-system as described in Chapters 2 and 4. The additional benefit is that these operations are uniformly provided as behaviors on primitive types, thus the same behavior application principles are used to apply them and create new types, classes, behaviors, functions and so on.

**Persistence.** The TIGUKAT model integrates persistent and transient objects. Persistent is a characteristic of individual objects, meaning persistence is orthogonal to type. The manner in which objects can be made persistent or transient is a language issue that is considered to be part of the database language methodology. The different storage and

management requirements of persistent and transient objects is an implementation issue that is outside the object model considerations.

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

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

**Ad hoc query facility.** The query model of TIGUKAT is defined as a uniform extension to the object model, thus cleanly integrating the two. The algebraic operations are developed as behavior extension to the model. A calculus is defined for declarative access to objects and has a complete translation to the algebra for processing from within the model itself. An SQL-like query language, TQL, has been developed [Lip93] for user-level declarative access to objects. This satisfies the query facility requirement and provides the unique contribution of a uniform, integrated query model.

### C.1.2 Optional Features

**Multiple inheritance.** The TIGUKAT model provides multiple inheritance as explained in the manifesto papers. However, it is called multiple subtyping in this thesis. A different meaning is attached to the term *inheritance*, which refers to the reuse of behaviors and implementations. The general consensus at present is that multiple subtyping is a mandatory feature of an OBMS and, thus, we feel this feature should be included as part of the previous section.

**Type checking and type inferencing** It has already been proven [SÖ90a] that much of the type checking involved in query processing can be performed at compile time. The query model definition supports type inferencing and dynamic schema creation for deriving type information of queries that return objects of heterogeneous types.

**Distribution** Distribution is an issue related to the implementation of the model and should be transparent within the model definition itself. The problems associated with distributed OBMSs are part of the future research.

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

**Versions** A versioning mechanism using time as a supplement of schema evolution has been developed in this thesis (see Chapter 5). The results of behaviors are defined by their histories as they change over time. These histories allow us to version objects and since the model is uniform, it allows us to version types, classes, behaviors, functions and so forth. The contribution in this area is that the *versions of behaviors* approach is new and gives a better integration of versions with other objects.

### C.1.3 Undetermined Mandatory or Optional

**View definition and derived data.** Views are part of the future work of this research. A view mechanism with update semantics is being developed for the object model.

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

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

**Schema Evolution.** A complete classification of schema changes has been identified and developed for the model in Chapter 5. Time is used to record schema changes, which helps us to maintain semantic integrity of behaviors and provide versions.

### C.1.4 Open Choices

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

**Representation system.** The TIGUKAT model defines a basic primitive type system that includes the functionality to uniformly extend all parts of the type system. This makes for a powerful and fully functional representation system.

**Type system.** As indicated in the point above, the primitive model definition includes a basic type system that is fully extensible.

**Uniformity** The TIGUKAT model uniformly treats all entities as objects. This includes all the primitives such as *object, type, class, collection, behavior* and *function*. Uniformity is an important feature in several respects. From the modeling perspective, a clean, self-contained description of the model with no dependence on external meta information can be defined (see Chapter 2). From a language point of view, a single uniform approach in accessing and manipulating all information in the system can be defined (see Chapters 3 and 4). In the query model this means the efficient query operators may be uniformly applied to the modeling primitives, thereby providing a powerful, ad-hoc access mechanism to what is essentially meta-information. In Chapter 4, it is shown how this provides reflection in the model. Uniformity has also been used to extend the base model with a query optimizer [Muñ94], temporality [GÖ93], schema evolution and version control (see Chapter 5), and work has begun on an extensible transaction manager. I feel uniformity is a major contribution of this work.

## C.2 Conformance to OODB Task Group Recommendations

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

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

The other components of the ODM reference model comply with those covered in Section C.1 or are related to non-data model issues such as storage management, query models, transaction management and programming languages.