Western University
Scholarship@Western

Digitized Theses | Digitized Special Collections

1991

The Kbo Model: Towards A Unified View Of Data, Behaviors, And Messages In Object-oriented Database Systems

Li Yu

Follow this and additional works at: https://ir.lib.uwo.ca/digitizedtheses

Recommended Citation
Yu, Li, "The Kbo Model: Towards A Unified View Of Data, Behaviors, And Messages In Object-oriented Database Systems" (1991). *Digitized Theses*. 2076.
https://ir.lib.uwo.ca/digitizedtheses/2076

This Dissertation is brought to you for free and open access by the Digitized Special Collections at Scholarship@Western. It has been accepted for inclusion in Digitized Theses by an authorized administrator of Scholarship@Western. For more information, please contact tadam@uwo.ca, wlswadmin@uwo.ca.

LIST OF SYMBOLS

Symbol	Description	Page
$\mathcal{I}$	set of all object-identities	41
$\mathcal{C}$	set of all class names	41
$\mathcal{A}$	set of all attribute labels	41
$\mathcal{C}_{base}$	the set of all base classes	41
$dom(D_i)$	constant domain of base class D_i	41
$\mathcal{O}$	object universe	41
$oid(o)$	identity of object o	41
$orf(o)$	set of object-identities reachable from o	41
$fis(k)$	formal input specification of knowhow k	43
$fos(k)$	formal output specification of knowhow k	43
$\mathcal{K}$	set of all non-empty knowhows	44
$\mathcal{K}^*$	knowhow universe	44
$bb(k)$	biological bearer of knowhow k	44
$kh(o)$	knowhow borne by object o	44
$\phi_\perp$	empty knowhow	44
$\leadsto$	message-sending primitive	46
ⓘ	INSTANCE-OF relationship	48
ⓑ	BEHAVIOR-OF relationship	48
$\dot{\Sigma}(c)$	interface of class c	49
$\dot{\Omega}(c)$	population of class c	49
ⓚ	KIND-OF relationship	51
ⓡ	REUSE-OF relationship	51
$ⓘ_d$	direct INSTANCE-OF relationship	56
$ⓑ_d$	direct BEHAVIOR-OF relationship	56
$ⓚ_d$	direct KIND-OF relationship	56
$ⓡ_d$	direct REUSE-OF relationship	56
$\emptyset_{if}, \emptyset_{then}, \emptyset_{else}$	pseudo part labels in bi-object classes	56
$\emptyset_{member}$	pseudo part label in set classes	56
$\emptyset_{ordered_member}$	pseudo part label in sequence classes	56
$\mathcal{P}$	set of all parts	56
ⓟ	PART-OF relationship	56
$ⓟ_d$	direct PART-OF relationship	57
$\mathcal{C}_{capsule}$	set of all capsule classes	58
$\mathcal{C}_{agg}$	set of all aggregate classes	58
$\mathcal{C}_{bio}$	set of all bi-object classes	58
$\mathcal{C}_{set}$	set of all set classes	58
$\mathcal{C}_{seq}$	set of all sequence classes	58
$dom(c)$	domain of class c	58
$\equiv$	object identity test	80
$\asymp$	object equality test	80
$\asymp_n$	object n-equality test	80
$\equiv_{kh}$	object knowhow-identity test	81

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-71961-3

Canada

ABSTRACT

One of the most distinguished features of object-oriented databases (OODBs) is their support for users to populate both data and arbitrary behaviors in the same database. Consequently, the increasingly large population of arbitrary behaviors in OODBs demands a database approach to their management and manipulation. Current OODB models offer no capability of managing arbitrary behaviors as meaningful database objects, and thus fail to meet such a demand.

The goal of this research is to unify data, behaviors, and messages into a uniform notion of objects so that a single database approach can be developed to manage and manipulate such objects. The result is a formal object-oriented data model, the KBO model, which integrates data, arbitrary behaviors, and messages into a more general notion called "Knowhow-Bearing-Objects" — objects that always bear an executable reference called "knowhow". Management of such objects is characterized by the mechanisms developed in six coherently related aspects: classification, re-utilization, identification, invocation, composition, and manipulation.

Classification of KBO objects is based on value structures, rather than on signatures or source code. Classes are related through two independent hierarchies: the KIND-OF hierarchy and the REUSE-OF hierarchy. A formal semantics is defined for the two hierarchies and theorems are developed to provide formal mechanisms for their syntactic validation. Identification of KBO objects is based on identities rather than on executable references. Invocation of KBO objects using both identities and values as behavior selectors comes naturally, thus providing the support for both monomorphism and polymorphism. Composition of KBO objects is based on four generic object structures which also imply dynamic semantics. As a result, complex objects can be directly used as complex messages to invoke complex behaviors. Theorems are developed to provide formal mechanisms to ensure the safety and success of complex messages. Manipulation of KBO objects is based on an object algebra, the KBO algebra, which can be used, not only to manipulate traditional data, but also to select, generate, maintain, and apply behaviors, in an associative manner. The properties of the algebra are formally analyzed, and theorems are developed to provide provably correct algebraic transformations.

To my wife Susan, my daughter Sarah, and my parents.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Osborn, for her support and guidance throughout my research. She was a constant source of good ideas and inspiration. In fact, this work originated as an attempt to formally define Osborn's OODB model and algebra, and to extend their power in representation and manipulation of objects.

I am also very much indebted to the other members of my research committee, Dr. Mullin and Dr. Bauer, for their review of the work and suggested improvements. In particular, Dr. Mullin provided insight and encouragement in many useful discussions while this thesis was in its formative phases, and Dr. Bauer was a great help in clarifying the problems in my research area. I would also like to thank Dr. Özsu, Dr. Yu and Dr. Nelson for their valuable comments.

Finally I would like to thank my family, but most of all, my wife Susan. Without her patience, support and encouragement, this thesis could never have been completed.

TABLE OF CONTENTS

Page

CERTIFICAATE OF EXAMINATION ii
ABSTRACT iii
DEDICATION iv
ACKNOWLEDGEMENTS v
TABLE OF CONTENTS vi
LIST OF FIGURES x
LIST OF SYMBOLS xii

CHAPTER 1 — INTRODUCTION 1

CHAPTER 2 — BACKGROUND 6
- 2.1 Introduction 6
- 2.2 Object-Oriented Programming Systems 6
- 2.3 Object-Oriented Data Models 8
- 2.4 Object-Oriented Query Algebras 14
  - 2.4.1 An Evaluation Framework 15
  - 2.4.2 Evaluating Object Algebras 20
- 2.5 Arbitrary Behaviors as Database Objects 29
  - 2.5.1 Classifying Arbitrary Behavior Objects 30
  - 2.5.2 Reusing Arbitrary Behavior Objects 34
  - 2.5.3 Identifying Arbitrary Behavior Objects 36
  - 2.5.4 Invoking Arbitrary Behavior Objects 37
  - 2.5.5 Combining Arbitrary Behavior Objects 37
  - 2.5.6 Manipulating Arbitrary Behavior Objects 38
  - 2.5.7 Summary 39

CHAPTER 3 — THE KBO MODEL: PRELIMINARIES 40
- 3.1 Introduction 40
- 3.2 Underlying Domains 40
- 3.3 A Naive Message-Sending Paradigm 45
- 3.4 A Conceptual Level Semantics 46
  - 3.4.1 Classification and Vitalization 47
  - 3.4.2 Generalization and Re-utilization 18
  - 3.4.3 Aggregations 55
  - 3.4.4 Aggregations in Structural Modeling 58
  - 3.4.5 Aggregations in Behavioral Modeling 62
- 3.5 Kernel Classes 66

CHAPTER 4 — KBO OBJECTS AND CLASSES 69
- 4.1 Introduction 69
- 4.2 KBO Objects 70
  - 4.2.1 Object Definition 70
  - 4.2.2 Graphical Representation of Objects . . . . 7[illegible]
  - 4.2.3 Object Equalities 79
  - 4.2.4 A Less Naive Message-Sending Paradigm 80
- 4.3 KBO Classes 83
  - 4.3.1 Class Definition 83
  - 4.3.2 Full Representations 85
  - 4.3.3 Value-based Interfaces 87

CHAPTER 5 — A DATABASE LEVEL SEMANTICS 89

5.1 Introduction 89
5.2 Three Interpretation Functions 90
5.2.1 Static Interpretation 90
5.2.2 Dynamic Interpretation 92
5.2.3 Reuse Interpretation 93
5.2.4 Examples 94
5.3 Orderings on KBO Classes 97
5.3.1 K-Compatibility and R-Compatibility 98
5.3.2 Syntactic Validation of K-Compatibility 99
5.3.3 Syntactic Validation of R-Compatibility 105
5.4 KBO Schemas 111

CHAPTER 6 — SIMPLE AND COMPLEX MESSAGE-SENDINGS 114
6.1 Introduction 114
6.2 Syntax of Message-Sendings 114
6.3 Semantics of Simple Message-Sendings 115
6.4 Semantics of Complex Message-Sendings 117
6.4.1 AGG Objects as Invokers 118
6.4.2 BIO Objects as Invokers 120
6.4.3 SET Objects as Invokers 121
6.4.4 SEQ Objects as Invokers 122
6.5 Shallow-Send and Deep-Send 123

CHAPTER 7 — SAFETY AND SUCCESS OF MESSAGE-SENDINGS . . . 128
7.1 Introduction 128
7.2 Well-Formed Complex Message-Sendings 128
7.3 Necessarily Unsafe Complex Message-sendings 135
7.4 Potentially Unsafe Complex Message-sendings 143
7.5 Strictly-Typed Complex Message-Sendings 146
7.6 Soundness of the $\vdash$ System 153
7.7 The $\vdash$ System Is Not Complete 159

CHAPTER 8 — THE KBO OBJECT ALGEBRA 161
8.1 Introduction 161
8.2 The Algebraic Operators 164
8.2.1 Select (σ) 166
8.2.2 Pickup (δ) 172
8.2.3 Apply (ρ) 173
8.2.4 Expression Apply (λ) 184
8.2.5 Project (π) 186
8.2.6 Combine (χ) 191
8.2.7 Union ($\cup$) 196
8.2.8 Intersect ($\cap$) 198
8.2.9 Subtract ($-$) 199
8.2.10 Collapse (ϖ) 200
8.2.11 Assimilate (α) 202
8.3 Queries for Generating Complex Invokers 203
8.3.1 Examples of Generating new AGG Invokers 204
8.3.2 Examples of Generating new BIO Invokers 205
8.3.3 Examples of Generating new SET Invokers 207
8.3.4 Examples of Generating new SEQ Invokers 209
8.4 Queries in an Interactive Application 212

CHAPTER 9 — PROPERTIES OF THE KBO ALGEBRA 217

9.1 Introduction 217
9.2 Object-Type Completeness 218
9.3 Closure Property 219
9.4 Relational Consistency 219
9.5 Formal Algebraic Transformations 221
9.5.1 Idempotence Properties 222
9.5.2 Commutativity Properties 228
9.5.3 Associativity Properties 230
9.5.4 Distributivity Properties 231
9.6 An Evaluation of the KBO Algebra 239
9.6.1 Object-Orientedness 239
9.6.2 Expressiveness 241
9.6.3 Formalness 243
9.6.4 Performance 244
9.6.5 Database Issues 245

CHAPTER 10 — CONCLUSIONS 246
10.1 Summary of Research 246
10.2 Thesis Contributions 249
10.3 Future Research Directions 252

REFERENCES 254

VITA 268

LIST OF FIGURES

2.1 Issues in Managing Arbitrary Behaviors as Database Objects 31
3.2 A Naive Message-Sending Paradigm 45
3.3 Sending object "get 1st word" to object "data model" 46
3.4 An Example of Instances and Behaviors 48
3.5 Connecting PHD-STUDENT with STUDENT and MSC-STUDENT 52
3.6 Creating BNR-EMPLOYEE from other two classes 54
3.7 All the parts of class PROJECT-BASED-CONSULTANT 59
3.8 Examples of Bi-object Classes 61
3.9 Example of an AGG Class in Behavioral Modeling 62
3.10 Example of a BIO Class in Behavioral Modeling 64
3.11 Example of a SET Class in Behavioral Modeling 65
3.12 Example of a SEQ Class in Behavioral Modeling 66
3.13 The KIND-OF and REUSE-OF Relationships among Kernel Classes . 68
4.14 The Object Graph and the Screen Display of o_1. 78
5.15 The DL-schema in Example 16 112
5.16 The DL-schema in Example 17 113
7.17 The Object Graph for Object x 137
7.18 The Set of All Potentially Unsafe Complex Message-Sendings 145
7.19 The Set of All Successful Complex Message-Sendings 160
8.20 The KIND-OF and REUSE-OF Hierarchies for an OODB System . . 163

LIST OF SYMBOLS

Symbol	Description	Page
$\mathcal{I}$	set of all object-identities	41
$\mathcal{C}$	set of all class names	41
$\mathcal{A}$	set of all attribute labels	41
$\mathcal{C}_{base}$	the set of all base classes	41
$dom(D_i)$	constant domain of base class D_i	41
$\mathcal{O}$	object universe	41
$oid(o)$	identity of object o	41
$orf(o)$	set of object-identities reachable from o	41
$fis(k)$	formal input specification of knowhow k	43
$fos(k)$	formal output specification of knowhow k	43
$\mathcal{K}$	set of all non-empty knowhows	44
$\mathcal{K}^*$	knowhow universe	44
$bb(k)$	biological bearer of knowhow k	44
$kh(o)$	knowhow borne by object o	44
$\phi_\perp$	empty knowhow	44
$\rightsquigarrow$	message-sending primitive	46
Ⓘ	INSTANCE-OF relationship	48
Ⓑ	BEHAVIOR-OF relationship	48
$\dot{\Sigma}(c)$	interface of class c	49
$\dot{\Omega}(c)$	population of class c	49
Ⓚ	KIND-OF relationship	51
Ⓡ	REUSE-OF relationship	51
Ⓘ$_d$	direct INSTANCE-OF relationship	56
Ⓑ$_d$	direct BEHAVIOR-OF relationship	56
Ⓚ$_d$	direct KIND-OF relationship	56
Ⓡ$_d$	direct REUSE-OF relationship	56
$\emptyset_{if}, \emptyset_{then}, \emptyset_{else}$	pseudo part labels in bi-object classes	56
$\emptyset_{member}$	pseudo part label in set classes	56
$\emptyset_{ordered_member}$	pseudo part label in sequence classes	56
$\mathcal{P}$	set of all parts	56
Ⓟ	PART-OF relationship	56
Ⓟ$_d$	direct PART-OF relationship	57
$\mathcal{C}_{capsule}$	set of all capsule classes	58
$\mathcal{C}_{agg}$	set of all aggregate classes	58
$\mathcal{C}_{bio}$	set of all bi-object classes	58
$\mathcal{C}_{set}$	set of all set classes	58
$\mathcal{C}_{seq}$	set of all sequence classes	58
$dom(c)$	domain of class c	58
$\equiv$	object identity test	80
$\asymp$	object equality test	80
$\asymp_n$	object n-equality test	80
$\equiv_{kh}$	object knowhow-identity test	81

$\Phi_K(c)$	set of all direct superclasses of class c	84
$\Phi_R(c)$	set of all direct supplyclasses of class c	84
$\Xi(c)$	local representation of class c	84
$\Sigma(c)$	local interface of class c	84
$\overset{\bullet}{\Xi}(c)$	representation of class c	87
$\mathring{\Sigma}(c)$	value-based interface of class c	88
$I_s(c)$	static interpretation of class c	92
$I_d(c)$	dynamic interpretation of class c	93
$I_r(c)$	reuse interpretation of class c	94
$\preceq_s$	static compatibility	99
$\preceq_d$	dynamic compatibility	99
$\preceq_k$	kind-of compatibility	99
$\preceq_r$	reuse-of compatibility	99
$\overset{1}{\rightsquigarrow}$	shallow-send operation	125
$\overset{*}{\rightsquigarrow}$	deep-send operation	126
$\overset{n}{\rightsquigarrow}$	n-level-send operation	127
$\epsilon(\mu)$	evaluation set of message-sending μ	130
$\epsilon^*(\mu)$	transitive evaluation set of message-sending μ	130
σ	Select operator	167
δ	Pickup operator	173
ρ	Apply operator	175
λ	Expression Apply operator	185
π	Project operator	187
χ	Combine operator	192
$\cup$	Union operator	197
$\cap$	Intersect operator	199
$-$	Subtract operator	200
ϖ	Collapse operator	201
α	Assimilate operator	203

The author of this thesis has granted The University of Western Ontario a non-exclusive license to reproduce and distribute copies of this thesis to users of Western Libraries. Copyright remains with the author.

Electronic theses and dissertations available in The University of Western Ontario's institutional repository (Scholarship@Western) are solely for the purpose of private study and research. They may not be copied or reproduced, except as permitted by copyright laws, without written authority of the copyright owner. Any commercial use or publication is strictly prohibited.

The original copyright license attesting to these terms and signed by the author of this thesis may be found in the original print version of the thesis, held by Western Libraries.

The thesis approval page signed by the examining committee may also be found in the original print version of the thesis held in Western Libraries.

Please contact Western Libraries for further information:
E-mail: libadmin@uwo.ca
Telephone: (519) 661-2111 Ext. 84796
Web site: http://www.lib.uwo.ca/

CHAPTER 1
INTRODUCTION

Object-oriented database (OODB) systems are currently the subject of a great deal of database research. Many OODBs have been proposed and implemented, such as GEMSTONE [Mai86], VISION [CS87], VBASE [AH87], IRIS [F*87], ORION [B*87], ENCORE [HZ87], and O2 [LRV88, LP89]. Unlike the relational data model, a single object-oriented data model has yet to emerge. Nevertheless, the existing OODBs share several high-level data-modeling concepts: (1) **Objects.** An object is an entity that has an internal state and an associated set of behaviors that operate on the state. To request that an object exhibit one of its behaviors, you send it a *message*. (2) **Complex Objects.** A complex object can be built from simpler ones by applying useful constructors, such as tuple and set. (3) **Object Identities.** Every object has a unique object identity which is immutable, and independent of the object's state. (4) **Classes.** A class is a group or categorization of objects that share the same state structure and the same set of behaviors. (5) **Encapsulation.** Encapsulation is the ability to protect an object's internal data from outside access by requiring that the object be manipulated only through its behaviors. (6) **Inheritance.** Inheritance is the ability to create new classes incrementally from existing, less specialized classes. (7) **Polymorphism.** Polymorphism is the ability to have objects of different classes respond to the same message with their own version of the behavior to achieve appropriately customized actions.

In this thesis, the word "behavior" is used as a general concept to cover what is variously called a method, function, procedure, operation, process, or capability, which is attached to a class (or a type) in an object-oriented system. Conceptually, a behavior is a computational entity which can exhibit certain "know-how" when activated (executed). Physically, behaviors may be realized in many different forms, such as functions, procedures, stored relations, programs (processes or tasks), commands (batch files), parameterized database queries, or special hardware devices.

It has been claimed by Hull and King [HK90] that the fundamental difference between object-oriented data models and semantic data models is that object-oriented data models support forms of local behaviors in a manner similar to object-oriented programming languages. This means that a database entity may locally encapsulate some complex behaviors (procedures or functions) for specifying data computations. *"This gives the database user the capability to express, in an elegant fashion, a wider class of derived information than semantic models"*[HK90].

As claimed, it is very attractive that OODB users can store and populate both data and behaviors in the same database. In particular, it is becoming a highly desired feature to allow behaviors in OODBs to be hardcoded in more than one computationally complete programming language, such as methods in O_2 [LP89], functions in IRIS [F*87], and procedures in POSTGRES [RS87]. With this capability, more and more of the intelligence can be moved out of application programs coded in different languages and into reusable "smart" objects, and much of the application execution is thus included in the database itself. As Dawson said [Daw89], *"By including more of the application code in the database (which is the locus of sharing), it becomes possible to share the application semantics embedded in the code"*. Consequently, we are witnessing an increasing proliferation of arbitrary behaviors in OODBs. This observation naturally motivates the need for direct database support for management of arbitrary behaviors as database objects in OODBs. Such a need has received increasing attention recently in OODB research field [Bee88, Bee89, CS88, ZM88, LRV88, RS89]. Most notably, Rozen and Shasha have come to the conclusion in [RS89] that one of the *"necessary features of an ideal database system"* is to *"allow procedures written in arbitrary programming languages to be stored in the database"* and treated as data.

Although a few OODBs (e.g., IRIS and VISION) made attempts to treat behaviors as database objects, their techniques are necessarily ad hoc, non-semantical, or application-specific (e.g., behavior objects of class TIME-SERIES in VISION). They are not supported by a comprehensive formal data model. Conventional OODBs and their underlying data models, in general, provide no direct support for managing arbitrary behaviors as meaningful database objects. The lack of support for such features not only makes it difficult for end-users to find (and combine, maintain, apply, etc.) the right behaviors in their queries, but, more importantly, limits the effectiveness of conventional OODBs for the following three reasons:

- Arbitrary behaviors deserve to be treated as objects. They carry knowhows about how to do things while data are knowledge about what things are. Behaviors can be grouped into classes according to their intended semantics, and have their own meaningful behaviors.

- We may want to have queries that intermix message variables (instead of constant messages) with higher-level query operations (e.g., selection, projection, etc.). Existing OODBs allow the queries that send the same message (or messages) to all the receivers in a set target. It is difficult, if not impossible, to send different messages to different receivers in a set target based on some re-

lationship between a qualified message and a qualified receiver.

- We may want to use the query language to manipulate, not only complex data, but also "complex messages". In existing OODBs, constant messages may be nested together in a query expression. However, the composition of these messages is just some quick "ad-hoc" effort to obtain the desired result. These compositions of several messages are either transient (gone at the end of the query session), or saved as "another command" which cannot be queried as "complex messages" like we do to complex objects. For example, the end-user cannot use a query facility to associatively obtain different sub-messages from different "complex messages" and combine them into a new "complex message".

There have been some proposed semantic models that are capable handling dynamic modeling. Three most notable ones are Mylopoulos and Borgida's TAXIS model [MBW80, BMW84], Brodie and Ridjanovic's SHM+ model [BR84], and King and McLeod's EVENT model [KM84]. However, their primary focus is the specification and design of transactions (in a specific, full-fledged specification language or compiler) which may require stepwise refinement and compilation. The transactions modeled are not intended to be managed, manipulated, combined or applied as (simple or complex) database objects by end-users in an associative manner. Furthermore, these models are by nature semantic models, rather than object-oriented models (for example, none of them addresses the issue of object identities). Therefore, these models could be used by code designers to design the body of arbitrary behaviors in OODBs, but they are not suitable for the study of management of arbitrary behaviors as database objects in OODBs.

We take a different approach to the problem. In our view, the knowledge about *what things are* (data) and the knowledge about *how to do things* (behaviors or messages) can be **seamlessly** modeled and manipulated in an OODB model. Compositions of behaviors and messages can be treated as "complex data objects" with behavioral semantics. In this way, they can be handled through the same query facility used for regular data objects.

What is needed is an understanding of the formal semantics of such an OODB model. It is with this goal in mind that we embarked on developing a formal object-oriented data model (the KBO model) that integrates data, arbitrary behaviors, and messages into a more general notion called "Knowhow-Bearing-Objects" — objects that always bear an executable reference called "knowhow" that knows how to do things. Management of such objects is characterized by the mechanisms developed in

six coherently related aspects: classification, re-utilization, identification, invocation, composition, and manipulation.

Classification of KBO objects is uniformly based on their value (simple or complex) structures, rather than on their signatures or source code. Classes are related through two independent hierarchies: the KIND-OF hierarchy and the REUSE-OF hierarchy. A formal semantics, both at the conceptual level and at the database level, is defined for the two hierarchies and theorems are developed to provide a formal mechanism for syntactic validation of KBO database schemas. The separation of the two hierarchies provides two orthogonal ways to organize KBO objects (as instances and as behaviors), and allows flexible knowhow reuse without interfering with the traditional "IS-A" semantics.

Identification of KBO objects is uniformly based on their object identities rather than on their executable references. Using both identities and values as behavior selectors becomes a natural way to invoke KBO objects' knowhow, and provides the support not only for polymorphic invocation but also for monomorphic invocation, which is the ability for users to pinpoint a desired knowhow.

Composition of KBO objects is uniformly based on four generic KBO complex object structures which also imply dynamic semantics. As a result, the KBO model is the first OODB model that equates its complex objects to complex messages. Four simple and useful computation control structures can now be treated as data structures inside the model. Any complex object can be directly immediately used as a complex message to invoke a complex behavior on a receiver. Theorems are developed to provide a formal mechanism to ensure the safety and success of such "data-like" complex messages.

Manipulation of KBO objects is uniformly based on an object algebra, namely the KBO algebra, which can be used, not only to manipulate traditional data, but also to select, generate, maintain, and invoke, in an associative manner, simple or complex behaviors. The properties of the algebra are formally analyzed, and theorems are developed to provide provably correct algebraic transformations.

To the best of our knowledge none of the existing OODBs have provided the flexibilities described above, and it is our conclusion that this work is a first step towards the development of a viable OODB that can directly manage and manipulate arbitrary behaviors within a unified object environment.

The thesis is organized as follows: In Chapter 2 we present an overview of object-oriented programming languages, data models and query algebras, and a closely related area of our research — management of arbitrary behaviors as database objects.

Chapter 3 presents a formal description of the basic concepts in the KBO data model. In Chapter 4, complete formal definitions are given for KBO objects and KBO classes, and resolution of attribute and behavior conflicts is discussed. Chapter 5 presents a set inclusion semantics of KBO schemas based on three interpretation functions, and formal mechanisms for syntactic validation of KBO schemas are also provided. Chapter 6 introduces the notion of complex message-sendings and their semantics based on object structures. Chapter 7 provides formal tools to guarantee the termination and success of the "data-like" complex message-sendings. The KBO object algebra is presented in Chapter 8 and its properties are formally analyzed in Chapter 9. Finally, Chapter 10 lists the main contributions of this research and outlines directions for future research.

Notational Conventions

As a notational aid, we have chosen to capitalize all names denoting classes throughout the dissertation. We also capitalize the first letter of names denoting object handles. Furthermore, throughout this thesis the terms *class* and *type* will be used synonymously.

CHAPTER 2

BACKGROUND

2.1 Introduction

This chapter includes a general discussion of current object-oriented programming systems, object-oriented data models, and object-oriented query algebras. Since a major part of our research is the development of an object algebra, we also introduce a comprehensive evaluation framework for existing object algebras. The next topic of our discussion is a directly related area of research — management of arbitrary behaviors as database objects. Various issues in this area will be examined, and the corresponding solutions in our work will be described in general terms.

2.2 Object-Oriented Programming Systems

Current OODB data models have been heavily influenced by the features from object-oriented programming languages. There are three important features that distinguish object-oriented programming languages from traditional programming languages: *encapsulation, polymorphism*, and *inheritance*. Encapsulation is the ability to protect an object's internal data from outside access and the ability to have objects that are bundled data and code. Polymorphism is the ability to have different kinds of objects respond to the same message in different (or identical) ways. Inheritance is the ability to create new classes incrementally from existing, less specialized classes. Such a class hierarchy captures (or is intended to capture) the "IS-A" semantics well known in the Artificial Intelligence area [SS77a, SS77b, SS78]. The three representative object-oriented programming systems are SMALLTALK, C++ and CLOS.

SMALLTALK [GR83] is an object-oriented programming language originally developed as a research vehicle for implementing interactive systems. Every object in SMALLTALK has an identity, a class, instance variables, and methods (behaviors). To support encapsulation, SMALLTALK uses only messages for operating on objects (i.e., to invoke methods). Classes are organized in a hierarchy, so that they can share common structure and methods in a superclass. The principal dialect of SMALLTALK uses single inheritance, but there have been a few experiments with multiple inheritance extensions [BI82]. To support polymorphism, SMALLTALK

examines the receiver object at run-time so that the message name can be bound dynamically to the address of the right method. Users are not allowed to directly access the instance variables in an object. Two special system classes, namely CompiledMethod and MethodContext, are used by SMALLTALK to gather all method bodies and their run-time executions.

C++ [Str86] is an object-oriented variant of the well-known C programming language. Because C++ places a very strong emphasis on efficiency, it avoids language features that require more than minimal support at run-time. C++ is not fully object-oriented because it also provides normal C data and because its class instances, which are a special kind of record, do not have (immutable) object identities. The original version of C++ used single inheritance. The principal reason for this seems to be to ensure that the storage representation of a class instance is identical to the storage representation (private fields) of an instance of any superclass. Each subclass (derived class) simply appends more fields to the end of the storage representation. To support encapsulation, C++ allows functions in a class to be declared private functions or friend functions. The name of a private function is available only in member functions and friend functions. A friend function of a class is a non-member function that is allowed access to the private part of the class. To support (restrict) polymorphism, C++ provides virtual member functions. Virtual member functions are messages that have more than one method and hence require dynamic binding of the right method. However, in C++ any class that is intended to be used in a polymorphic fashion has to have all its member functions virtual. The user should only use non-virtual functions where he/she is very sure that the class will never have a subclass. It is a big burden for C++ users to guess in advance which class may have subclasses or which functions may be overridden in the future. C++ utilizes compile-time type checking to some extent, but situations that can not be caught during compile-time are not taken care of during run-time.

CLOS (Common Lisp Object System) [Moo89] consists of five major concepts: classes, slots, generic functions, methods, and method combinations. Classes contain the description of instance structure and behaviors. Slots are the containers for data in an instance and hold administrative and other information about that data. Generic functions are the mechanism through which run-time, class-sensitive dispatching is accomplished. Methods are pieces of code which are associated with generic functions and contain the code the generic functions will invoke. As a result, CLOS uses generic functions, rather than messages, to operate on objects. To support polymorphism, generic functions, when called, will examine the classes of all the arguments (no one

is the receiver) passed in the call. Based on these classes they will transfer control to an appropriate body of code (a method). A special feature offered by CLOS is that programmers can specify "roles" for methods. These roles will also impact the chain of events that comprises the execution of a generic function. Some roles are predefined in CLOS, new ones may be specified by CLOS programmers. Two of the built-in roles are called :before and :after methods. They are supplied with CLOS because they cover the frequently needed case of having some pieces of code run before or after some primary method. Method combination can be achieved by selecting and running the methods with different roles in the correct order. CLOS supports multiple inheritance, and inheritance conflicts are resolved by defining an order of precedence that determines which class's characteristics dominate.

2.3 Object-Oriented Data Models

Unlike record-oriented data models, object-oriented data models provide various data structuring and manipulating mechanisms in an object-oriented manner [GR83, Str86, Moo89]. It is notable that there are some similar properties between object-oriented data models and semantic data models (see surveys [HK87, KM85]). However, as Hull and King pointed out in [HK90], *"Object-oriented database models are fundamentally different from semantic models in that they support forms of local behavior in a manner similar to object-oriented programming languages."*

The GEMSTONE data model [MP84, Mai86, PS87, B*89] merges object-oriented language concepts from Smalltalk with the capabilities of database systems. GEMSTONE's database language OPAL is similar to Smalltalk-80 [GR83] and is used for data definition, data manipulation and general computation. In GEMSTONE, every object has a unique identity that is retained through arbitrary changes in its own state. Attributes are modeled by instance variables. Since every class in GEMSTONE has to have instance variables, it seems that GEMSTONE only supports tuple-like complex objects. Unlike many other OODB models, a class definition in GEMSTONE does not specify types for instance variables (attributes); rather, types are associated with objects themselves. This offers designers more experimentability, but makes static datatype-checking impossible. GEMSTONE only supports a single inheritance hierarchy. GEMSTONE supports polymorphism/dynamic binding, because message lookups are performed by the interpreter, not the compiler. That is, messages are bound to methods at run-time. The lookup cannot be performed until the class of the receiver is known. In GEMSTONE attributes are given no type information,

and sets can have arbitrary objects as members and need not be homogeneous. Furthermore, OPAL variables are not associated with types[1]. All these features have to be supported by means of dynamic binding. GEMSTONE also intends to provide procedural interfaces to C and Pascal languages. This means that GEMSTONE programmers may be given the freedom to write methods in their favorite languages. The manipulation language OPAL is computationally complete, with assignment and flow of control constructs. GEMSTONE did not provide any special support for ad hoc queries. Instead, it was left to the user to define appropriate container types (e.g., set or relation) with iterator operations, and to use these in programs to retrieve or manipulate collections of objects.

The VISION data model [CS87, CS88] adopts a functional view of data, which is very similar to that of DAPLEX [Shi86]. Its database language is based on a Smalltalk-like syntax. All information about an object is embodied in functions which map a collection of objects into another. Attributes are modeled by enumerated functions and methods are modeled by computed functions. It is not clear whether every VISION object has a system-generated object identifier. There is no class concept in VISION. Objects are grouped into named collections. A collection is an assertion about a specified number of distinct objects. Each collection has a representative object called a prototype object. Functions are used to map one collection to another. Like GEMSTONE, VISION does not provide any ad hoc querying facilities. They have to be defined by the user as blocks, which are parameterized sequences of statements similar to those in SMALLTALK. There are two special features offered by the VISION model: (1) there are no classes or types in the database, and (2) functions are treated as objects. The function hierarchy in VISION includes these types: Function, Computed, Enumerated, FixedProperty, and TimeSeries. FixedProperty functions are insensitive to a temporal context, while TimeSeries functions maintain a series of events in an object's history. The class of a function-object determines the meta-functions that are available to it.

The VBASE data model [AH87, And90] supports strong datatyping, block structured schema definition, parameterization, an inverse mechanism, trigger methods, and other standard features. Each object in VBASE has a unique object identity generated as a part of object creation. VBASE insists on *strong reference semantics*: all objects, even integers, single characters, and booleans, are truly represented by a reference (note that this is also the approach used in our model). VBASE also

[1]So that unanticipated cases (a company car might be assigned to a department as well as an employee) can be more easily handled.

has 13 system-defined "aggregate types", including arrays, dictionaries, stacks, and sets. Complex objects can be formed by using these aggregate types. A type in VBASE has properties and operations. Operations are implemented by a series of executable code fragments. VBASE also supports parameterized types, enumeration types, union types, and variant types. Like GEMSTONE, VBASE does not allow multiple inheritance. Properties and operations are inherited via the type hierarchy in the expected manner. Besides some static type checking, VBASE uses explicit run-time type checking to achieve the same expressive capability as an untyped object system. VBASE supports an SQL query interface with object extensions. Methods in VBASE are written in a C-like language COP. An interesting feature in VBASE is the support for inverse relationships. If properties A and B are declared inverse, then whenever an update is performed on one property the other property is updated accordingly. Furthermore, VBASE allows the get/set operations for properties to be customized by implementors.

Largely based on DAPLEX [Shi86], the IRIS data model [F*87] supports high-level structural abstractions (classification, generalization and aggregation) and behavioral abstraction (encapsulation). In IRIS each object has an assigned, system-wide, unique object identifier, and has an encapsulated value. Non-atomic objects in IRIS are represented *internally* in the database by object identifiers. Atomic objects (e.g., integers and character-stings) have no user-accessable object identifiers. Therefore, they cannot be created, destroyed or updated by users. Objects serve as arguments to functions and may be returned as results of functions. Complex objects are expressed as functions which can be multi-valued. Types in IRIS are named collections of objects, which can have an atomic, tuple or set structure. Objects belonging to the same type share common functions, and all attributes are modeled as functions. An IRIS function is: (1) a stored function (a relation), (2) a derived function (a query expression), or (3) a foreign function (an arbitrary programming language implementation). It is unclear how functions of multiple arguments are attached to types. An unconventional feature in IRIS is that objects can belong to multiple types which may or may not have subtype/supertype relationships. IRIS supports user-defined foreign functions that are compiled, stored and executed under the control of the database. Any general-purpose programming language can be used to write a foreign function as long as the object code can be linked with the IRIS query processor. However, IRIS cannot optimize the body of a foreign function because IRIS does not understand the body. At execution time, when a foreign function is invoked, IRIS transfers control to the entry point of the foreign function to evaluate and return results for the

function call in IRIS. IRIS provides a query language, OSQL, which has SQL-syntax. It is unclear what is queried by OSQL, data or functions, and therefore we do not know for sure if IRIS supports encapsulation in its query language. All functions in IRIS are instances of type Function. Operations may be supported to create and/or update functions. Type Function may have subtypes like NonPredicate, Predicate, SystemFunction, and UserFunction.

The ORION data model [B*87, B*88, K*89, K*90] is often regarded as a full featured OODB model. Every object in ORION has an object identifier. Complex objects can be represented as recursively nested objects with standard constructors: tuple and set. In ORION, an attribute value is either an atomic value, an object identifier, or a set of object identifiers. Each class in ORION contains three lists: the list of superclasses, the list of attributes, and the list of methods. Multiple inheritance is supported in ORION. A subclass inherits attributes and methods from all its superclasses. Attributes and methods of a subclass can be explicitly selected from its superclasses. Otherwise, conflicts will be resolved on the basis of superclass ordering. Because ORION extends CommonLISP with object-oriented programming and database capabilities, its query syntax is CommonLISP-like. The important query primitives in ORION are **select** and **select-any**. The result of a query is always a subset of the instances of a class. Since only methods can be used in ORION queries, its query language does not violate encapsulation.

The EXTRA data model [CDV88] might be considered to be an extension of the object-oriented paradigm because of its inclusion of values which do not respect the object encapsulation and identity principles. In EXTRA, values are typed whereas objects belong to classes. Complex objects can be structured through a tuple, a set or an array constructor. EXTRA provides three different kinds of attribute value semantics: **own** attributes, **ref** attributes, and **own ref** attributes. An **own** attribute is simply a value with no identity. By default, all attributes in EXTRA are value attributes which cannot be referenced from elsewhere. A **ref** attribute is an independent object with identity. An **own ref** attribute is a dependent object which has an "as-a-whole" deletion semantics: it has to be deleted when its referencing object is deleted. Therefore, EXTRA's **own ref** attributes are similar to the composite objects in ORION. EXTRA also supports derived attributes defined via queries. A type in EXTRA does not imply the set of all instances (values) of the type. This makes it possible for a database to include more than one collection of instances of a given type, which can be very useful for non-traditional applications. Functions and procedures attached to a type have to be written in the EXTRA query language. Hence,

EXTRA does not support methods written in multiple languages. The advantage is that query optimization techniques can be applied to them. EXTRA allows multiple inheritance in which attribute conflicts are resolved by explicit renaming. Unlike ORION, no automatic resolution is provided. The EXTRA query language (called Excess) has a QUEL-like syntax and allows modification of a database as well as querying. It also provides facilities to define new functions. However, those functions cannot be associated with classes but only to types. Thus, we cannot really speak of integration. Encapsulation is not violated in the EXTRA queries because objects are queried through appropriate functions while values are queried according to their structure and appropriate functions. The result of a query is always a set value. It seems that the elements of a query result are always tuple values. We notice that the EXTRA designers decided to define a query algebra for query optimization after they already had the query language EXCESS.

Similar to the EXTRA model, O2 [LRV88, LP89, C*89a] supports both objects and values. In addition, the O2 data model is the first object-oriented data model with a formal semantics. In O2, an object is a pair *(identifier, value)*. The identifier is unique, value independent and allows reference to the object. Objects may have a tuple or set structure. Values in O2 have no identity and cannot be shared. Objects belong to classes, while values belong to types. Objects are manipulated through methods while values through operators. To every class is associated a type, describing the structure of its instances. Classes are created explicitly using commands and are organized in the inheritance hierarchy. Types are not created explicitly since they only appear as components of classes and do not appear in the inheritance hierarchy. Classes are organized in an inheritance hierarchy which is based on subtyping. Subtyping in O2 has a set inclusion semantics. The O2 data model supports multiple inheritance, but conflicts are resolved by users. In O2, the actual code of a method to be executed is not selected at compile-time but at run-time depending on the actual type of the receiver object. The query language for the O2 data model is RELOOP, which is based on an algebra [C*89a] intended for query optimization. RELOOP has an SQL-like syntax, respects encapsulation and allows querying of methods and values. The result of a query is either an existing object or an constructed value. The RELOOP compiler has a type inference mechanism. Because a RELOOP query is always a **select-from-where** block (a set filter), it only has access to set structures. In O2, one can associate a more specific method to a named object, to override an existing method in the class of the object. O2 performs type checking statically when inheritance is not used. However, when inheritance is fully used, some run-time

type-checking has to be performed.

The ENCORE data model [HZ87, Zdo88, Zdo89, SZ89] provides support for ADTs as types, type inheritance, typed collections of typed objects, and objects with identity. In ENCORE, all objects have identity and are instances of some type that describes the behavior of its instances. ENCORE has two parameterized types: Tuple[$< (A_1, T_1), \ldots, (A_n, T_n) >$] and Set[$T$]. Tuple and Set types are intended to allow the creation of new, strongly typed complex objects. A type definition includes a set of Properties (typed attributes) and a set of Operations (methods). Implementation of a property cannot have side-effects. Unlike other models, ENCORE's Tuple and Set types have fixed sets of properties and operations. For instance, Tuple[$< (A_1, T_1), (A_2, T_2) >$] can only have two properties *Get_attribute_*A_1, *Get_attribute_*A_2, and two operations *Set_attribute_*A_1, *Set_attribute_*A_2. ENCORE supports multiple inheritance in a type hierarchy. How conflicts are resolved is not addressed. Also, we are not aware whether the parameterized types Tuple and Set can have subtypes. ENCORE has an object algebra which supports ADTs and object identity while providing associative access to objects. The algebra treats properties as stored values. It can create new objects with unique identities and use object identity to manipulate objects. In ENCORE, types, operations and properties are all themselves objects. However, their classification of operation objects is neither semantics-based (such as in VISION) nor signature-based (such as in Napier88) — it is activation-based: an operation type is a procedure definition and instances of the operation type are activations of the procedure represented by the operation type. Because of this, each type in ENCORE has a set of operation types that can be instantiated and invoked on its instances. A subtype may add operation types that are not defined on its supertypes or may refine some of the operation types that are defined on its supertypes.

The OODAPLEX data model [Day89] is extended from the semantic data model DAPLEX [Shi86]. In OODAPLEX, objects are modeled by entities. Each entity has a system-generated identity which is independent of the entity's attribute values. OODAPLEX provides Tuple, Set and Multiset type constructors for building complex values. These complex values do not have a system-generated identity; they are identified only by their members. Complex objects can only be simulated by defining property functions for an abstract entity type. Each type in OODAPLEX is associated with a extent that is the set of instances of the type currently in the database. Attributes and operations in a type are indistinguishably modeled by functions which may have side-effects. A function is applied to a tuple consisting of an instance of

each of its input argument types; it then produces a tuple consisting of an instance of each of its output argument types. The set of all possible tuples of input arguments, for which the function is defined, concatenated with the corresponding tuples of output arguments, is the extent of the function. Multiple inheritance is supported in OODAPLEX. When a subtype inherits two functions of the same name, conflict is explicitly resolved by appending the corresponding supertype name to each of the conflicting functions. In OODAPLEX, only single-argument functions are attached to types; multi-argument functions are not encapsulated with any types! Therefore, polymorphism is only possible in the application of single-argument functions. An algebra called OOAlgebra is also defined for OODAPLEX. The algebra operates on complex values (tuples, sets, multisets); ADT objects are treated as 1-tuple values. Functions are also treated as sets of tuples. It seems that the only difference between the OOAlgebra and the relational algebra is that the atomic values in the relational algebra are extended to be ADT objects with identity. Also, the OOAlgebra is polymorphic in the sense that its operations can take tuples or sets as operands. Comparing with the IRIS model, the OODAPLEX model has extended DAPLEX farther by supporting the features such as manipulation of both sets of objects and individual objects, recursive queries, and an object algebra.

The CACTIS data model [HK86, HK89] emphasizes the support for a wide variety of derived data computed from functions. Every object in CACTIS has an integer identifier. Because every object in CACTIS only responds to four attribute-oriented messages (Initialize, Get_Attr_Val, Set_Attr_Val, Get_Rel_Val), it seems that CACTIS only supports the object constructor corresponding to the conventional "tuple" constructor. Conventional attributes and methods are modeled by intrinsic attributes and derived attributes, respectively. A derived attribute has attached to it an evaluation rule (written in C) which may not have side effects. CACTIS is claimed to be "active" because it recomputes derived attribute values whenever necessary. Multiple inheritance is supported and resolved at schema definition time. It is up to the designer to decide which evaluation rule to use when two same-name derived attributes are inherited by a subtype. CACTIS does not support conventional set-oriented queries. All its queries are navigational by following relationship connectors.

2.4 Object-Oriented Query Algebras

One of the common complaints against OODBs has been "*No Query Language*" [Pas89] — there is no object-oriented equivalent to SQL. It has been strongly advo-

cated by Stonebraker and others [Sto90] that a query language must be present in any kind of database system. We believe that a good first step towards an object-oriented query language is a well-defined, concise query model that accommodates the rich semantics of object-oriented data models. Recently, object-oriented query models have been developed for OODBs (e.g., IRIS and ORION), but few of them have an object algebra that underlies the query model. In this section, we are interested in developing desirable metrics for an "ideal" object algebra. Although the metrics are suggested for algebraic query models, many of them are general enough to be applied to non-algebraic query models.

In many object-oriented database applications, the advantages of using a well-chosen family of algebra operations as the basis of a query model may outweigh the restrictions imposed on the expressive power of the model. This approach supports the ability to write programs that work independently of physical structures. When arbitrary programs are used as queries, end-users may need to know about the physical data structures used (e.g., bits, bytes, registers and segments in C). They must write code which depends on the particular structure selected, leaving no opportunity for the physical structure to be tuned as database usage becomes clearer [Ull88]. In addition, using algebraic operations provides more opportunities for query optimization. Queries can be formulated in many equivalent forms and optimized by equivalence preserving transformations. The algebraic approach also provides an important property to an OODB query model — the *closure property*. A query model has the closure property if each operation on an object (objects) produces a object which has exactly the same status as the original one (ones), namely that all the operations of the object algebra are potentially applicable to it. By having this property, the result of a query can be used as the input for other queries or can be stored as a user's view. Up to now, most existing query models for OODBs do not preserve closure [ASH89].

2.4.1 An Evaluation Framework

As Brodie and Ridjannovic have said [BR88], defining and perfecting an object algebra is one of the major challenges in object-oriented data models. Although a few object algebras have been proposed, it is important to identify criteria for evaluating the relative merits of these object algebras. In the following, we present an evaluation framework for object algebras, which consists of five categories: *object-orientedness, expressiveness, formalness, performance,* and *database issues*. More detailed description of this framework can be found in [YO91a].

OBJECT-ORIENTEDNESS CATEGORY

Supports object identities and their manipulation. To support object identity, an object algebra should define its semantics over identities (i.e., its operations take object identities as input and produce an object identity as output), explicitly introduce multi-level equalities (i.e., identical, shallow equal and equal in the sense of [KC86]), and provide operations to manipulate identities. Multi-level equalities are important for eliminating duplicates [SZ89], supporting various user interfaces and studying the equivalence properties of object algebra expressions when rearranged [Osb88].

Supports encapsulation. "Encapsulation is obtained when only the operations are visible and the data and the implementation of the operations are hidden in the objects" [Atk89]. To support the encapsulation principle, the definition of an object algebra should be restricted to operate on objects only through their interfaces.

Supports inheritance hierarchy. The usual semantics of an inheritance hierarchy is that it captures the generalization (i.e., "IS-A") relationship between one class and a set of classes specialized from it [Kim89]. To support this conceptual "IS-A" semantics, queries expressed in an object algebra should be allowed to be directed against a (partial) inheritance hierarchy rooted at a target class, so that the operation scope is all the instances of all the classes in that (partial) hierarchy [Kim89].

Supports polymorphism. An object algebra is *polymorphic* if its operations are defined over object identities of any kind of allowable objects in the underlying data model. In other words, an "object" algebra should be defined across all "objects". If an object algebra is polymorphic, it looks more "full-fledged" because users will not find that some kinds of objects are apparently arbitrarily precluded from the algebra's manipulation scope.

Distinguishes classes and collections. Most query models in existing OODBs allow queries to be directed either against classes (e.g., in ORION [Kim89]) or against set objects (e.g., "collection objects" with type Set[T] in ENCORE [SZ89]). From the users' point of view, "querying against set objects" provides a simple form of privacy. For example, assume the user has a handle *MyFriends* denoting a set of persons, which is a subset of all instances of class PERSON. If querying targets can only be set objects, then the user knows for sure that no one can ever discover their "secret friends" without obtaining the handle *MyFriends*. From the implementors' point of view, using set objects as querying targets often means much smaller index structures for searching in the query model. Furthermore, sets-as-targets can always simulate classes-as-targets, but in general the opposite is not true.

Supports heterogeneous sets. Heterogeneous sets are those sets which have members of different types. When queries are directed against sets, heterogeneous sets may have to be used to support *manually* the generalization concept: if a set has member type C, we can insert into this set any object that is an instance of any subclass of C. By so doing, when we retrieve that set we will encounter not only instances of class C but also instances of other classes which are more specialized than C. A special case is when a set has a type which is the root of the class lattice, such as Object in ORION [B*87]. In this case, the set may have members that are not related through user-defined inheritance relationships.

EXPRESSIVENESS CATEGORY

Extends relational algebra consistently. For an object algebra to be at least as powerful as relational algebra, it should provide, as a minimum, an object-oriented counterpart for each of the five operations that serve to define relational algebra: *Selection, Projection, Cartesian product, Union,* and *Difference* [Ull88]. These operations work on sets rather than navigating an object at a time. It is often said that in OODBs the semantics of Join in relational algebra can always be realized by path traversals as long as all relationship types are predefined [Mai86, Zdo89]. However, it is impossible to envision all relationships between objects at the time when the database schema is designed [AG89]. Indeed, the lack of facilities to express arbitrary "join" is another major criticism of the early OODBs [NS88].

Includes object constructors. Object constructors can be algebraic expressions which denote the identity of a new transient object with a given value. We believe that object constructors are useful concepts because (1) they are explicit and free of side-effects, (2) they create temporary objects that will not survive the current querying session (unless they are explicitly made persistent), and (3) they allow users to express arbitrarily complex objects *on the fly*, in order to, for example, construct a non-atomic object to use in a comparison.

Supports sequence objects. It has been pointed out in [Atk89, Sto90] that the support for sequence objects is one of the minimal requirements for OODBs. Sequences "are important because they capture order, which occurs in the real-world, and they also arise in many scientific applications, where people need matrices or time series data"[Atk89].

Supports invocation of behaviors. An object algebra should allow invocation of arbitrary behaviors and incorporate their results into the algebra expressions. This requirement is actually a necessary condition for the criterion "Supports encapsulation" in the Object-Orientedness Category. However, the current criterion implies

more than supporting encapsulation: it allows the invocation of not only those behaviors that are intended to protect an object's internal data (typically, attribute accessors), but also those behaviors that perform arbitrary computations. Since user-defined behaviors may be arbitrarily encoded in many programming languages, an object algebra can provide (at a lower-level) full computational power to users by supporting this feature.

Includes behavior constructors. If object algebra expressions are freely intermixed with messages (which invoke user-defined methods), it is often desirable to organize several messages in a conditional, iterative, sequential, or concurrent way. In other words, there may be a need to construct useful (high-level) ways in which a set of messages is sent to a receiver. This provides a powerful tool to create dynamic method combinations out of a large number of user-defined primitive methods. This also provides opportunities for dataflow execution in a parallel architecture [Ack82, B*87b].

Supports dynamic type creation. The question of whether we should support dynamic type creation is, in a narrower sense, related to the question "Should we keep operations like projection, Cartesian product, and join used in relational algebra?". It has been argued by Osborn [Osb88] that well-understood manipulations which create new objects are necessary and useful. For instance, users in a design application may want to join a design object with a parts object in another application, to find out how much the design would cost. It is not realistic to expect that the database designers could foresee all possible relationships needed in an application.

Supports querying transitive closures. The object algebra might support transitive closure and other recursive queries directly. If this is the case, recursive queries can be immediately recognized and efficient algorithms applied to process them. Otherwise, users have to express recursive queries in methods written in a programming language. The query optimizer might not detect these recursive queries embedded in a method. These queries would be much more difficult to optimize.

Supports behaviors-as-objects. Since OODBs encapsulate processes together with data, they encourage an incremental proliferation of arbitrary behaviors in OODBs, especially in behavior-growing database application areas. Of particular interest to future behavior management systems is the ability to dynamically manipulate behavior definitions and to associatively generate and apply composite behaviors. Unfortunately, very few existing OODBs have paid attention to this aspect. Beeri has suggested [Bee89] that arbitrary functions can be treated as values and operated upon by constructors to obtain a collection of ***structured function values.***

We believe it is desirable that the object algebra can manipulate behaviors just as it can manipulate regular objects.

FORMALNESS CATEGORY

Has a formal semantics. It is crucial for the object algebra to have precise, mathematical definitions for its operations (including update operations) and generic object classes that operations handle. Without such formal definitions, the meaning of the algebra operations is unclear. Furthermore, for data consistency and query optimization, a well-understood mathematical object algebra provides a better basis for development of a higher-level query langauge [Bee89, SO90].

Is a closed algebra. This criterion is fundamental for any query model that claims to be an "algebra". Any algebra should specify the types (sorts) of objects supported and the allowable operations on objects of each defined type. In addition, all legal operations should be closed, so that no operation produces a result which lies outside of the scope of the algebra. If the algebra is one-sorted, all its operations take as input and produce as output objects of a single type. Otherwise, the algebra is many-sorted, which may introduce complexity but allows both simple and complex objects to be treated in a uniform manner [GZC89].

PERFORMANCE CATEGORY

Supports strong typing. Supporting strong typing usually implies supporting type checking at compile time [MMM90]. It is arguable whether an object algebra is more effective when based on strong typing. The languages used in PROBE, VBASE, IRIS, O2 and ENCORE are strongly typed, while the languages used in GEMSTONE (based on Smalltalk) and ORION (based on Lisp) are not. The latter group has consciously avoided strong typing to achieve better dynamic binding. Many systems provide a static type-checker, but run-time type checking is still needed because some features prevent static type checking (e.g., dynamic methods and exceptional attributes in O2). If an object algebra supports this property, it is then possible to decide whether an algebra expression is type safe. However, static type checking is usually supported at a cost in flexibility, which is significant for many applications.

Provides algebraic optimization strategies. Equivalence properties (based on various equalities supported) between the algebra operations should be studied and strategies for optimizing algebraic expressions provided.

<u>DATABASE ISSUES CATEGORY</u>

Supports both persistent and transient objects. The object algebra may provide assignment operations that distinguish persistent naming and transient naming of (intermediate or final) result objects. Persistent naming allows users to save (intermediate) results of algebra expressions if they are interested in the results or they want to use the results in the future. Transient naming allows users to retain results of algebra expressions for further manipulation during the query session; this makes it much more convenient to build up a complex query. It should be noticed that similar facilities can be provided outside of the object algebra (such as in [MD86]). When assignments are included inside the object algebra (such as in [Osb89a]), they preserve the closure property and can be freely nested in algebra expressions.

Supports schema evolution. Database schemas evolve in many application environments. Many existing OODBs have made an effort to support schema evolution [B*87, PS87]. However, most OODBs' query models do not support schema evolution, in the sense that when the database schema evolves, the applications using the database have to "evolve" as well. This is because query expressions contained in the applications have to be modified to suit the new database schema. One goal is to have an "evolution-tolerant" object algebra that "tolerates" some frequent forms of schema evolution[2], so that query expressions in the applications can still return equivalent results. With such an object algebra, users need not modify their applications when the database schema is evolving [Osb89b].

Has an equivalent object calculus. For relational databases, end-user languages are usually non-procedural (e.g., Quel, SQL), that is, based on a relational calculus. It is thus desirable to have an object calculus, whose expressive power is equivalent to that of the object algebra, as a basis for an end-user language. The object algebra can serve as the underlying evaluation mechanism for the calculus-based query language.

2.4.2 Evaluating Object Algebras

Based on the above framework, we now discuss nine query algebras proposed recently in the literature:

Manola & Dayal's Algebra (MD) [MD86, OGM86, D*85, DS85]
Osborn's Algebra (Osb) [Osb88, Osb89a, Osb89b]
Straube & Özsu's Algebra (SÖ) [SO90, Str91]
Shaw & Zdonik's Algebra (SZ) [SZ89, SZ90]

[2]It may be impossible to tolerate all kinds of schema evolution.

RELOOP Algebra for O2 (O2) [C*89]
Dayal's OOAlgebra for OODAPLEX (OO) [Day89]
Guo, Su & Lam's Association Algebra for OQL (GSL) [GSL91]
Davis & Delcambre's Algebra (DD) [DD91]
Vandenberg & DeWitt's Algebra (VD) [VD90, VD91]

Manola & Dayal's algebra called the PDM algebra, was developed for the PROBE database system. The PDM algebra is a modified relational algebra operating upon functions. In particular, an entity type (such as PERSON) is treated as a unary function which, when evaluated, returns a set of entities of that type. Formal arguments of PDM functions are labeled, and can be declared to be *in*, *out*, or both so that functions can return multiple values. PDM functions can be extensional (i.e., stored as relations) or intensional (i.e., computed from a subroutine).

Osborn's algebra was developed for a general object-oriented data model. The algebra is defined on three generic classes: atomic, aggregate and set objects. Relational algebra operations are extended. Also included are Naming, DeepCopy and Apply operations. Apply serves as an iterator on set objects. DeepCopy creates a complete copy of an object without sharing any sub-objects with the old one.

Straube & Özsu's object algebra [SO90, Str91] was developed to provide a formal basis for object-oriented query processing. To support encapsulation, the algebra allows only one comparison, namely the identity test. An object calculus is also provided. The translation from the algebra to the calculus is complete, but the translation from the calculus to the algebra is only partial.

Shaw & Zdonik's object algebra operates on objects only through the external interfaces defined by their types. Results of queries are collections of existing objects or collections of tuples built by the query. By including parameterized types, the algebra can be statically type-checked while maintaining the ability to construct dynamic relationships between existing objects.

The RELOOP Algebra is proposed to provide a basis for the query language RELOOP for the O2 data model. The RELOOP query language has a SQL-like syntax. Although RELOOP supports invocation of behaviors in its **select-from-where** blocks, such a mechanism is not explicitly defined in the semantics of its algebra. RELOOP is intended to operate only on object values, instead of object identities.

Dayal's OOAlgebra is proposed for the OODAPLEX data model. The algebra seems to be an extension of Manola & Dayal's PDM algebra. The algebra supports uniformly the formulation of associative and navigational queries over objects by

making the algebra polymorphic in a way similar to Osborn's algebra.

Guo, Su & Lam's Association Algebra is used as the basis for the design of the OQL query language and data model. Objects and their associations are uniformly represented as association patterns (e.g., linear structures, trees, lattices, networks, etc.) which can be directly manipulated by the Association Algebra. Operators of the algebra can be used to navigate along the path of interest to construct or decompose a complex pattern.

Davis & Delcambre's algebra is mainly intended to provide the functionality of relational algebra for an object-oriented data model. The algebra has a denotational semantics, and queries formulated by the algebra return classes, which retain structural connections to existing classes. The algebra also has an object calculus.

Vandenberg & DeWitt's algebra is proposed as the basis for the EXCESS query language for the EXTRA data model. The algebra is many-sorted: it provides a different set of operations for different kinds of complex objects (e.g., multisets, arrays, tuples, etc.). In particular, the algebra supplies two operations Reference and Dereference only on object identities.

OBJECT-ORIENTEDNESS CATEGORY

Supports object identities and their manipulation. Manola & Dayal's object algebra operates on functions which can be stored as relations; these functions themselves do not have identities. The tuples in a stored function contain the surrogates of entities (objects) to identify what value to return as a result when the function is invoked for a give object. Although there are no explicit multi-level equalities, the algebra does provide an identity constructor operation and an operation for eliminating duplicates. These operations manipulate identities to some extent. We thus think that their algebra partially satisfies the criterion. Osborn's object algebra supports three kinds of object-equalities: identity, shallow identity, and equality. The algebra semantics is defined on identities, but no special operations are supplied to manipulate them. Straube & Özsu's object algebra is defined on sets of identities, but supports only identity-test, and provides no operations to manipulate identities. Therefore, we consider Osborn's and Straube & Özsu's algebras also partially comply with the criterion. Shaw & Zdonik's object algebra is defined on identities of collection objects. It also supports arbitrary level identity-testing and provides operations to manipulate identities, namely DupEliminate(O, i), that keeps only one copy of i-equal objects from set-object O, and Coalesce(O, A, i), that eliminates i-equal duplicates in the attribute A of the tuples from set-object O. We consider their algebra satisfies the criterion. The RELOOP algebra, Dayal's OOAlgebra and Guo, Su & Lam's algebra

do not satisfy this criterion because their operations operate on values, not object identities, and they do not support manipulation of object identities. Although Davis & Delcambre's algebra is defined over identities (actually, collections of identities), their algebra does not supply operations for identity manipulation. Hence, their algebra partially complies with this criterion. Vandenberg & DeWitt's algebra also partially satisfies this criterion because only a subset of their operations are defined over identities (i.e., Reference and Dereference).

Supports encapsulation. Osborn's algebra, the RELOOP algebra, Dayal's OOAlgebra, Guo, Su & Lam's algebra, Davis & Delcambre's Algebra, and Vandenberg & DeWitt's algebra seems to be directly manipulating object attributes because methods are not explicitly addressed in their semantics. Hence, their algebras do not comply with the criterion. Manola & Dayal's algebra manipulates primarily the results of functions visible at the object boundary. Straube & Özsu's algebra operates only upon objects whose internal representation is inaccessible. Similarly, Shaw & Zdonik's algebra supports encapsulation; in particular, it also supports Tuple and Set objects which obey a standard interface (e.g., *Get_attribute_value* and *Set_attribute_value* for Tuple objects). Therefore, these last three algebras satisfy the criterion.

Supports inheritance hierarchy. Straube & Özsu's algebra, Guo, Su & Lam's algebra and Davis & Delcambre's algebra are the three that take advantage of the usual semantics of the inheritance hierarchy. Specifying a class in their queries refers to the class's instances and all instances of its subclasses. Particularly, in Straube & Özsu's algebra, the leaves of a query tree can be either a class name C or a class subhierarchy C^* where the latter includes all members of class C and its subclasses. Other object algebras use the inheritance hierarchy more for integrity control (e.g., any method defined for PERSON may also be applied to STUDENT, any attribute of type PERSON may also be given a STUDENT object as value, and any set whose member type is PERSON may also contain STUDENT objects, etc) than as a generalization pattern. We are not aware of how the RELOOP algebra deals with this issue.

Supports polymorphism. Only Osborn's algebra and Dayal's OOAlgebra satisfy this criterion: all their operations (except some set operations) take as input and produce as output any kind of objects. In Dayal's OOAlgebra, operations Rename, Project, Select, Product, Apply_append can all take and produce either tuple values or set values. Manola & Dayal's algebra effectively manipulates sets of entities. Straube & Özsu's algebra operations take as input "classes-as-sets" and produce as output sets of objects, and Shaw & Zdonik's algebra operations take as input and produce as output collection objects. The RELOOP algebra takes and produces only set values.

Guo, Su & Lam's algebra takes class relationships and produces new relationships. Davis & Delcambre's algebra takes and produces only classes of objects. Vandenberg & DeWitt's algebra supplies different sets of operations for different kinds of objects.

Distinguishes classes and collections. Manola & Dayal's object algebra treats entity types as collections of entities. Davis & Delcambre's algebra and Straube & Özsu's algebra treat class names as sets of objects. Guo, Su & Lam's algebra takes class relationships as an object graph (a network). Therefore, these algebras do not comply with this criterion. Osborn's algebra, Shaw & Zdonik's algebra, the RELOOP algebra, Dayal's OOAlgebra and Vandenberg & DeWitt's algebra satisfy this criterion by distinguishing between classes and set objects that have a class as the member type.

Supports heterogeneous sets. Osborn's algebra, Straube & Özsu's algebra, Shaw & Zdonik's algebra, the RELOOP algebra, Guo, Su & Lam's algebra, and Davis & Delcambre's algebra satisfy this criterion. Osborn's algebra supports both homogeneous sets (called "strong sets") and heterogeneous sets (called "weak sets"). A strong set of member type C can contain *only* objects of type C. Shaw & Zdonik's algebra supports heterogeneous sets; homogeneous sets are special cases of heterogeneous sets. Straube & Özsu's algebra supports heterogeneous sets in two different ways: (1) the leaves of a query (expression) tree can be either C or C^* where the latter stands for all members of the class (name) C and its subclasses; (2) the input of an algebra operation can be a union set of the members of two different classes (e.g., class PERSON and class BOOK). We are not aware whether Manola & Dayal's algebra, Dayal's OOAlgebra and Vandenberg & DeWitt's algebra support sets that can contain tuples of different types.

The above evaluations are summarized in the following table:

OBJECT-ORIENTEDNESS	MD	Osb	SO	SZ	O2	OO	GSL	DD	VD
Supports oids and their manipulation	◇	◇	◇	√	×	×	×	◇	◇
Supports encapsulation	√	×	√	√	×	×	×	×	×
Supports inheritance hierarchy	×	×	√	×	?	×	√	√	×
Supports polymorphism	×	√	×	×	×	√	×	×	×
Distinguishes classes and collections	×	√	×	√	√	√	×	×	√
Supports heterogeneous sets	?	√	√	√	√	?	√	√	?

√ — satisfies criterion ◇ — partial compliance

× — criterion not satisfied ? — not detailed in papers

EXPRESSIVENESS CATEGORY

Extends relational algebra consistently. Manola & Dayal's object algebra is a modified relational algebra. It contains almost unchanged relational operations. They

operate upon relation-simulated functions. For example, Select(F,P) selects those tuples of function F that satisfy a Boolean function P, and Project(F,L) projects the tuples of F on the functions in list L. The relational Join operation is replaced by the Apply_Append(F,G), which applies function G to arguments taken from the tuples of function F and returns a new function formed by appending to F's columns the results of evaluating G on the tuples of F. The algebra is extended consistently from the relational algebra. Osborn's algebra made a bigger step from the relational algebra. All its relational-like operations are defined on complex objects. The algebra consists of Choose (extended Select), Partition (extended Project), Combine (extended Cartesian product), set operations, and other useful operations. Osborn's algebra is powerful enough to simulate the relational algebra. Straube & Özsu's algebra has five operations, Select, Union, Difference, Generate and Map. Operation Map is similar to the Apply-to-all operation in functional programming. Since the algebra does not have counterparts to projection and Cartesian product, it does not satisfy the criterion. Shaw & Zdonik's algebra includes Select, Image, Project, Ojoin, set operations, Flatten (for a set of sets), Nest, UnNest, and identity operations DupEliminate and Coalesce. Operation Image is very like Straube & Özsu's Map, except that Image only applies unary methods to objects. Relational Cartesian product can be simulated by Ojoin with appropriate predicate specifications. The difference is that Ojoin always creates a set of two-attribute tuples. This is intended to preserve encapsulation. Shaw & Zdonik's algebra comes close to satisfying the criterion. The RELOOP algebra includes set operations, Selection, Projection, Cartesian product, and Reduction (which reduces a set of single-member tuples to a set of members without the tuple structure). The algebra can simulate relational algebra. Dayal's OOAlgebra and Davis & Delcambre's algebra are very similar to RELOOP in terms of the operations supplied. Guo, Su & Lam's algebra can directly simulate relational algebra by its association operators. In Vandenberg & DeWitt's algebra, selection and projection can be simulated through SET-APPLY operator, and Cartesian product, union and difference can be directly achieved by other operators.

Includes object constructors. Osborn's algebra, the RELOOP algebra, Dayal's OOAlgebra, and Davis & Delcambre's algebra include, explicitly, object constructors, e.g., constant creators of tuples and sets. In Shaw & Zdonik's algebra, tuple and set creators are methods attached to the parameterized types *Tuple* and *Set*. In Manola & Dayal's algebra, Createobj(F,L) creates a new function F1($L_1, \ldots, L_n$, L) obtained by concatenating each tuple of F($L_1, \ldots, L_n$) with a new column L. L is a surrogate. This seems to be a "set-object" constructor. There is no tuple constructor. Hence,

Manola & Dayal's algebra partially satisfies the criterion. Straube & Özsu's algebra and Guo, Su & Lam's algebra do not have object constructors.

Includes sequence objects. Only Vandenberg & DeWitt's algebra satisfies this criterion by supplying nine array algebra operators.

Supports invocation of behaviors. Osborn's algebra, RELOOP and Davis & Delcambre's algebra do not comply with this criterion. The Apply_Append operation in Manola & Dayal's algebra and Dayal's OOAlgebra, the Image operation in Shaw & Zdonik's algebra, and the Map operation in Straube & Özsu's algebra are all specifically intended to allow invocation of arbitrary behaviors (methods), and allow incorporation of the results of those behaviors into algebra expressions. In Vandenberg & DeWitt's algebra, since all methods have to be written in the algebra, their SET_APPLY operator can achieve such a purpose.

Includes behavior constructors. It seems that only Straube & Özsu's algebra has made an attempt to satisfy this criterion. In their Map operation, one can specify a *"method-list"* which can be considered a sequential method constructor because it builds a complex operation out of a sequence of methods [SO90].

Supports dynamic type creation. Straube & Özsu's algebra and Guo, Su & Lam's algebra fail to support creation of objects of new types. In particular, Guo, Su & Lam's algebra only produces an existing object graph (or network). The other algebras support some form of dynamic type creation. In addition, Shaw & Zdonik's algebra supports strongly typed objects of new types.

Supports querying transitive closures. Manola & Dayal's algebra complies with this criterion because it includes a traversal recursion operation [D*85], which can be used when these parameters are specified: the node and edge entities of the graph to be traversed, the functions of the nodes or edges to be computed recursively, a recipe for computing each of these functions in terms of the function values computed for the immediate predecessors or successors, and any selection conditions on the nodes, edges, or paths traversed. It seems that Guo, Su & Lam's algebra can also satisfy this criterion through the combination of their association operations.

Supports behaviors-as-objects. None of these algebras complies with this criterion. To satisfy this criterion messages/behaviors should be made true "first-class" objects like other regular objects. In Manola & Dayal's algebra, functions are the leaves of a query tree, but it is unclear whether a set of arbitrary functions themselves (not the tuples they contain) can be manipulated as true objects with identity.

The above evaluations are summarized in the following tables:

EXPRESSIVENESS	MD	Osb	SO	SZ	O2	OO	GSL	DD	VD
Extends relational algebra consistently	√	√	×	√	√	√	√	√	√
Includes object constructors	◇	√	×	×	√	√	×	√	√
Supports sequence objects	×	×	×	×	×	×	×	×	√
Supports invocation of behaviors	√	×	√	√	×	√	√	×	√
Includes behavior constructors	×	×	◇	×	×	×	×	×	×
Supports dynamic type creation	√	√	×	√	√	√	×	√	√
Supports querying transitive closures	√	×	×	×	×	<	√	×	×
Supports behaviors-as-objects	×	×	×	×	×	×	×	×	×

√ — satisfies criterion ◇ — partial compliance

× — criterion not satisfied ? — not detailed in papers

FORMALNESS CATEGORY

Has a formal semantics. Straube & Õzsu's algebra, the RELOOP algebra Guo, Su & Lam's algebra, Davis & Delcambre's algebra, and Vandenberg & DeWitt's Algebra have a concise, mathematical semantics, and proofs of algebraic transformations have utilized their definitions. Shaw & Zdonik's algebra only partially satisfies this criterion because some of its operations, such as *DupEliminate*(S, i) and *Coalesce*(S, A, i), lack formal semantics.

Is a closed algebra. Manola & Dayal's algebra and Dayal's OOAlgebra fail to satisfy this criterion because their operations are sometimes undefined if the operands are arbitrary intensional functions. Also, it is not addressed in Dayal's OOAlgebra what set operations return if applied on tuples. Neither does Osborn's algebra satisfy this criterion: while its other operations can produce atom or aggregate objects, its set operations do not accept atom and aggregate objects as operands. Other algebras satisfy this criterion.

The above evaluations are summarized in the following tables:

FORMALNESS	MD	Osb	SO	SZ	O2	OO	GSL	DD	VD
Has a formal semantics	×	×	√	◇	√	×	√	√	√
Is a closed algebra	×	×	√	√	√	×	√	√	√

√ — satisfies criterion ◇ — partial compliance

× — criterion not satisfied ? — not detailed in papers

PERFORMANCE CATEGORY

Supports strong typing. In a strongly typed object algebra, every variable (name or handle) must have its type declared and can only be assigned objects of its type or of a subtype of its type [LP89]. If the types of objects are known but not the types of variables, then type checking cannot be done at compile time. Osborn's

algebra is not strongly typed because named variables have no types — a variable can be assigned values of different type at different points of execution. It seems that Manola & Dayal's algebra, Straube & Özsu's algebra, Shaw & Zdonik's algebra and Guo, Su & Lam's algebra satisfy the criterion because they only operate upon, respectively, entity types, class names, collection objects, association patterns. The RELOOP algebra and Davis & Delcambre's algebra seem to satisfy this criterion. Their operands have relation-like value and it is possible to determine all the new attributes created in new tuples statically. We are not sure whether strong typing can be guaranteed in Dayal's OOAlgebra and Vandenberg & DeWitt's algebra.

Provides algebraic optimization strategies. In Manola & Dayal's algebra, the RELOOP algebra and Dayal's OOAlgebra, relational query optimization techniques can be directly applied. However, specific optimization strategies with respect to the "object" algebra are not detailed in the literature. Optimization strategies (basically, algebraic equivalences) are given for the other algebras. Straube & Özsu go one step further, giving semantic transformations for some of their operations [SO90].

The above evaluations are summarized in the following tables:

PERFORMANCE	MD	Osb	SO	SZ	O2	OO	GSL	DD	VD
Supports strong typing	√	×	√	√	√	?	√	√	?
Provides algebraic optimization strategies	?	√	√	√	?	?	√	√	√

√ — satisfies criterion ◇ — partial compliance
× — criterion not satisfied ? — not detailed in papers

DATABASE ISSUES CATEGORY

Supports both persistent and transient objects. Manola & Dayal's algebra, Davis & Delcambre's algebra, Guo, Su & Lam's algebra, and Vandenberg & DeWitt's algebra choose to deal with persistence outside of the algebra, and thus all its results and intermediate results are assumed to be transient (saving them is outside the algebra). Osborn's algebra includes a saving facility (ReClass). The problem is that before the user saves a transient object, details (attributes, sub-attributes, etc) about that object have to be known and an appropriate class must exist. We thus consider Osborn's algebra partially complies with the criterion. Straube & Özsu's algebra only deals with persistent objects. We are not aware how Shaw & Zdonik's algebra, the RELOOP algebra and the OOAlgebra deal with this issue.

Supports schema evolution. Using some carefully defined predicates, Osborn's algebra can make certain queries behave in an equivalent way during two types of

schema modifications: (1) an atomic attribute of a class is changed to an aggregate attribute, and (2) a cluster of subclasses are added under an aggregate class in the subclass hierarchy and all the objects of that aggregate class are moved into appropriate new subclasses [Osb89b]. Osborn's algebra only partially complies with this criterion because only limited schema changes are considered (a taxonomy of schema changes are given in [B*87]). No other object algebras satisfy this criterion.

Has an equivalent object calculus. Only Straube & Özsu and Davis & Delcambre have defined a calculus for their object algebra; however, the equivalence of expressive power between the calculus and the algebra is only partial [Str91, DD91]. Hence, we consider that their algebras partially comply with this criterion. Guo, Su & Lam's algebra is equivalent to the OQL query language. However, we are not aware whether there is a formal calculus underlying the OQL query language. The same argument can be applied to Vandenberg & DeWitt's algebra. None of other algebras satisfies this criterion.

The above evaluations are summarized in the following tables:

DATABASE ISSUES	MD	Osb	SO	SZ	O2	OO	GSL	DD	VD
Supports persistent & transient objects	×	◇	×	?	?	?	×	×	×
Supports schema evolution	×	◇	×	×	×	×	×	×	×
Has an equivalent object calculus	×	×	◇	×	×	×	?	◇	?

√ — satisfies criterion ◇ — partial compliance
× — criterion not satisfied ? — not detailed in papers

2.5 Arbitrary Behaviors as Database Objects

From the data modeling point of view, *objects* in an OODB represent *conceptual entities* from the application domain being modeled. A natural observation is that conceptual entities in the real-world are clearly divided into two kinds of conceptual entities: *static conceptual entities* and *dynamic conceptual entities*. Typical examples of static conceptual entities are a person, an employee, or a student. Typical examples of dynamic conceptual entities are an "eat" behavior, a "run" behavior, and a "sleep" behavior for persons, employees or students. Each dynamic conceptual entity (let's call it a behavior) represents a unique "knowhow" that knows how to carry out an action to achieve the purpose highlighted in its name. In other words, a behavior is a knowhow exhibitor that works for its owners. For example, the behavior called "run" of persons exhibits the knowhow that makes a person "run" (e.g., moving the person's two legs back and forth again and again).

Current research in the OODB field focuses largely on modeling static conceptual entities as objects despite the fact that OODBs have the potential to be populated with many user-defined arbitrary behaviors, possibly implemented in multiple programming languages (e.g., foreign functions in IRIS, methods in O_2, and user-defined procedures in POSTGRES). Only in IRIS and VISION are user-defined arbitrary behaviors treated as database objects. Unhappily, however, they are simply indiscriminately collected under a few system-defined function types, capturing little real-world semantics of these dynamic conceptual entities. In other words, the "function objects" in IRIS and VISION are not organized in a *semantics-preserving* manner. Therefore, we argue that existing OODBs lack intuitiveness in representing arbitrary behaviors as database objects, thus leading to the loss of a direct conceptual correspondence between a real-world behavior and its representation as an true database object. We are interested in providing a mechanism to help the user organize arbitrary benaviors in a more sensible way.

In the following, we shall discuss the basic issues in managing arbitrary behaviors as database objects. To successfully manage arbitrary behaviors in OODBs as true database objects, a database system must address six fundamental problems (see Figure 2.1):

- How to classify these arbitrary behavior objects.
- How to reuse these arbitrary behavior objects.
- How to identify these arbitrary behavior objects.
- How to invoke these arbitrary behavior objects.
- How to combine these arbitrary behavior objects.
- How to manipulate these arbitrary behavior objects.

2.5.1 Classifying Arbitrary Behavior Objects

Traditionally, classes are used to organize objects according to their structure. For arbitrary behavior objects, their structure is in general either the structure of their executable code or the structure of their source code. Because arbitrary behaviors can be implemented in arbitrary programming languages (or even hardware devices) and because their implementations are encapsulated so that they can have changes transparent to end-users, it is difficult to reveal their internal structures (executable code

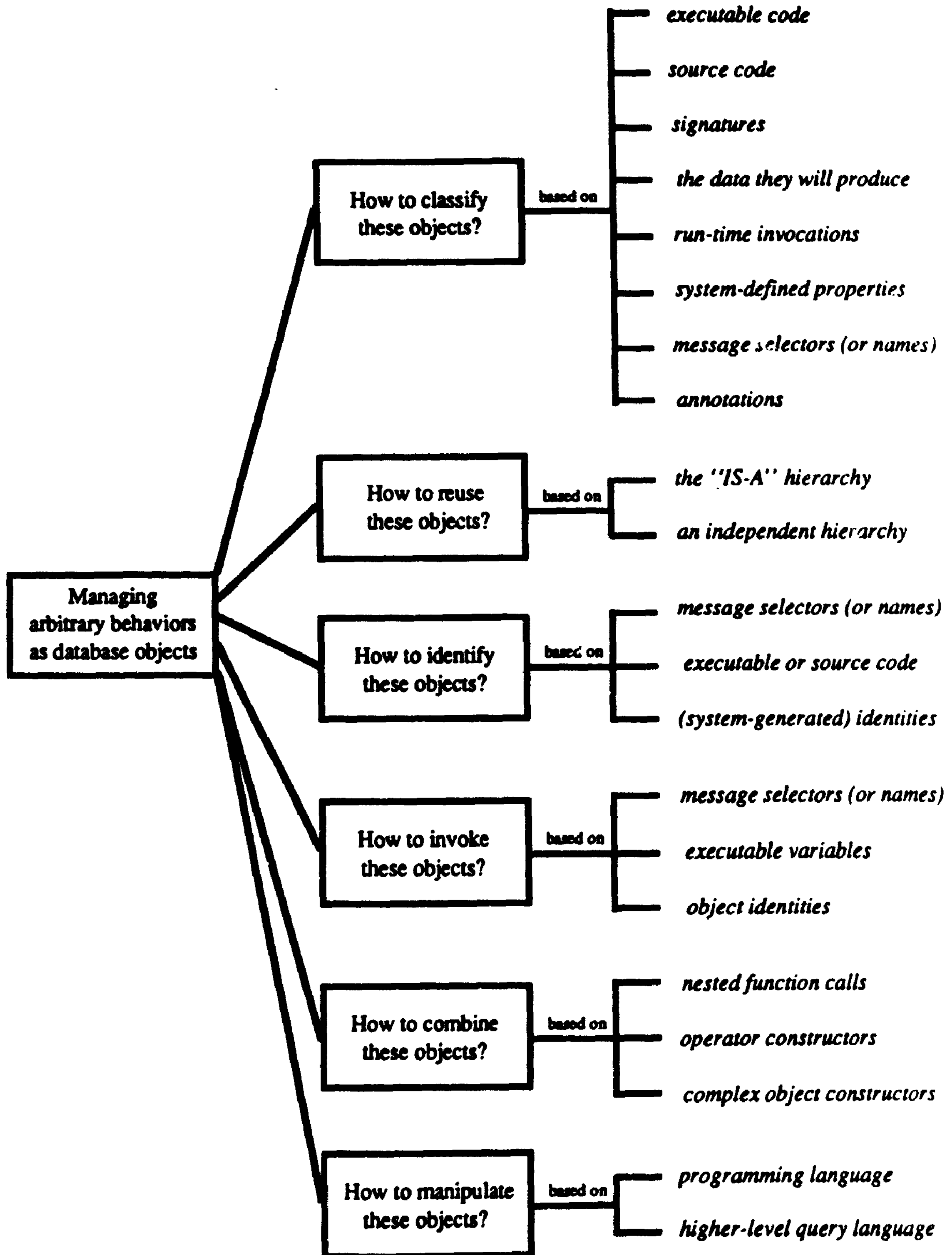


Figure 2.1: Issues in Managing Arbitrary Behaviors as Database Objects

or source code) and treat them as structured data. Furthermore, we are not interested in modeling source code construction (such as reported in [ZW88]) for arbitrary programming languages. For OODB end-users who only intermix the (ready-made) arbitrary behaviors in their queries, it is not important how the source code of an arbitrary behavior is edited and compiled or, for example, how many assignment statements it contains — it should be a secret kept by the behavior's implementor. End-users are usually interested in what "knowhow" a behavior can exhibit when used (e.g., "it knows how to rotate a window") and in how arbitrary behaviors can be organized and combined as regular objects in a "semantically related" manner (e.g., collect all behaviors that know how to display an object such as a line, a picture or a window).

Many database programming systems where procedures are "first-class values" classify procedures in terms of their signatures (e.g., [D*89, A*89, FSS89, MS89]). This facilitates static type checking because the compiler knows the signature of a procedure variable from its declared type. However, while this treatment helps the performance, it often leads to meaningless modeling of the real-world "knowhow" entities. For example, behavior b_1 *"How old are you?"* and behavior b_2 *"Take an IQ test"* attached to class PERSON would be treated as two instances of class (or type) PERSON→INTEGER. Obviously, this kind of "procedure class" is introduced more for compilers than for "real-world oriented" end-users. It may make more intuitive sense to group behavior b_2 *"Take an IQ test"* with signature PERSON→INTEGER and behavior b_3 *"Take a GRE test"* with signature PERSON×SUBJECT→SCORE-REPORT into the class TAKE-TEST, simply because they can then be naturally retrieved by compulsory test takers or simply because they are likely to share the same annotation structure (i.e., meta-semantic characteristics, such as special attributes Purpose and TestFormat in class TAKE-TEST). It must be noted here, that if we choose not to use signatures to classify behavior objects, then what is traded off is the static checking of behavior invocation; in other words, we have to rely on dynamic checking.

In the POSTGRES model [RS87], POSTQUEL procedures can be stored as values of attributes in a relation. The end-user can retrieve the result of the execution of such a procedure. Such an approach in effect classifies behaviors based on the data they will produce. For example, the class STUDENT might be defined as [Name: TEXT, Majors: MAJORS] where each instance of class MAJORS is either a set of majors or a POSTQUEL procedure that produces a set of majors. Based on this idea, the above behaviors b_1 *"How old are you?"* and b_2 *"Take an IQ test"* of class PERSON

would be treated as two instances of class INTEGER because they both produce an integer as the result. Obviously, this approach ignores the existence of the knowhows possessed by the behavior objects; it emphasizes the result of the execution of these objects rather than the "intelligent ways" the result is obtained. For the behaviors that only cause side-effects, it would be impossible to classify them (except under a class like VOID). Hence, this approach has the same problem as the approach based on signatures, namely it cannot take advantage of the intuitive semantics associated with real-world behavioral entities.

In the ENCORE model [HZ87], a behavior is treated as an "operation type", which is a procedure definition, and instances of that operation type are run-time invocations of the procedure. One probably has to introduce meta-types to classify these "operation types" in order to treat behaviors themselves as (persistent) objects. Therefore, the original problem arises again: how do we classify these behavior objects?

In a few existing OODBs, namely VISION [CS87, CS88] and IRIS [F*87, F*90], functions are treated as system objects that belong to a system-defined type Function or its system-defined subtypes, such as Computed, Enumerated, FixedProperty and TimeSeries in VISION. The subtypes of Function usually imply some kind of system-defined properties. For example, in VISION, instances of Enumerated are functions implemented by giving them explicit results [CS87]. In IRIS, one can apply some system-defined functions, such as FunctionName, FunctionBody and FunctionArgcount, to a function object. These are system function types — users are not allowed to add their own "function types". As a result, user-defined arbitrary behaviors are indiscriminately collected into one (or a few more) system-defined function type, capturing little semantics about the real-world dynamic conceptual entities. In other words, these "function objects" are not classified based on what they know how to do; they are classified based on the fact that they all have a signature and a body (and maybe a timestamp). We also notice that neither in IRIS nor in VISION are equalities of these "function objects" defined or addressed.

Two other options (see Figure 2.1) for classification of arbitrary behavior objects are to base it on their names (more precisely, message selectors) or on their annotations. The approach used in the KBO model combines these two options into one: every arbitrary behavior object has a ***descriptor*** as its value (in the traditional sense, such as in O_2) which may be an atomic value (if intended to be just an intuitive name[3], such as "How old are you?") or a tuple value (if intended to be an attribute-based

[3]The name can be regarded as the simplest form of annotation.

annotation, such as [Purpose: "How old are you?", BasedOn: "Nominal age"[4]]) or a sequence or set value (if intended to be a detailed descriptor or a multi-valued name, such as { "How old are you?", "Tell me your age!", "Retrieve age" }) or any other values allowable in the data model. Furthermore, we introduce an executable reference[5] which we call "a knowhow" as another intrinsic part of such an object (besides an object's value), which knows how to exhibit a "knowhow", whose semantic purpose is described in the value part of the object. With such a treatment, an arbitrary behavior object can be roughly considered to be a triple: (*identity*, *descriptor*, *knowhow*), where the *descriptor* is the object's value in the traditional sense.

Generally speaking, our "descriptor-based" classification approach provides the following advantages: (1) the classification of these objects imposes basically no restriction on what computations their knowhows may encapsulate, and OODB users are free to define families of *what they think are "semantically-related"* behaviors (such as the class TAKE-TEST); (2) the value of an object may serve as a detailed annotation of the knowhow of the object; (3) these objects may be used to build regular complex objects which, at the same time, imply some complex behavioral semantics, (i.e., complex objects can be viewed as complex behaviors); (4) since the values of these behaviors are effectively regular data or structured data, the query language can be used to associatively select desired behaviors or "complex behaviors" and immediately execute their knowhow or knowhows; and (5) if we do let complex objects (e.g., tuples, sets, sequences) imply some behavioral semantics, the query language can also serve as a handy "complex behavior generator" for quick application development, since most query languages in OODBs can produce new complex objects. The major disadvantage of this approach is that behavior invocations cannot be statically type-checked because behaviors are not classified by their signatures (they can still be type-checked at run-time). However, the KBO model is mainly intended for interactive database applications that desire a high experimentability, and we thus consider this disadvantage a reasonable trade-off for our purpose.

2.5.2 Reusing Arbitrary Behavior Objects

Due to the very nature of behavior objects, we cannot study their management without addressing their reuses. In true object-oriented systems, any behavior, right upon its creation, is attached to a class (the owner of the behavior), for which the behavior can exhibit its knowhow. For example, behavior b_1 ("How old are you?") is attached

[4]Chinese speak of a person's age as real age or nominal age.

[5]In practice, it can be viewed as a pointer to a fragment of executable code.

to the class PERSON. This means that when a person is asked "How old are you?", the behavior b_1 will exhibit its knowhow for that person — it *knows how* to derive or compute the age of that person. In existing OODBs, behaviors of a class can be reused by its subclasses defined in a class hierarchy. This feature allows users to create new classes incrementally from existing classes. However, the class hierarchies in existing OODBs are also considered to be a conceptual tool to capture the "IS-A" semantics [Bro84]: any instance of a subclass "is a" valid instance of its superclasses. Combining the role of code reuse and the role of modeling "IS-A" semantics in one single class hierarchy often leads to either incomprehensible conceptual semantics or inefficient code reuse in the class hierarchy [LTP86, YO91b]. Furthermore, the traditional class hierarchies often impose an automatic overriding mechanism so that a locally defined behavior of a class always overrides the other inherited behaviors with the same name. For example, if class EMPLOYEE is a subclass of PERSON and both have a locally defined behavior named "Introduce yourself", then sending the message "Introduce yourself" to an employee object will always cause it to introduce itself as an employee instead of a person (unless the message is changed). Such a receiver (object) is restricted to show only a single view, rather than multiple views, of the effect of a message.

To avoid these problems, the KBO model introduces an independent hierarchy, called the REUSE-OF hierarchy, to allow new classes to reuse behaviors of existing classes. Another hierarchy, called the KIND-OF hierarchy, is intended for the conceptual modeling of the "IS-A" semantics. For example, class EMPLOYEE can be a subclass of PERSON (i.e., EMPLOYEE KIND-OF PERSON) without reusing any behaviors from PERSON, whereas class DOG can reuse all the behaviors from PERSON (i.e., DOG REUSE-OF PERSON) without being a subclass of PERSON. Furthermore, there will be no behavior-overriding from the view point of object identity. For example, if class EMPLOYEE reuses behaviors from PERSON and both classes have a locally defined behavior named "Introduce yourself", then EMPLOYEE has two behaviors with value "Introduce yourself" which are distinguishable by their object-identities. Therefore, if we also treat messages as objects then we can also utilize identity-based comparison to select a desired behavior of the receiver. For example, if we send a message that is identical to the behavior "Introduce yourself" reused from PERSON, to an employee receiver, we would see the employee introducing himself as a person rather than as an employee (this is desirable in many situations). Consequently, messages with the same name (value) may cause different behavior objects to be invoked.

There have been several attempts to separate inheritance and subclassing in programming systems, such as CommonObjects [Sny85, Sny86], Exemplar-based Smalltalk [LTP86], POOL-I [AL90], and OROS [RWW89] (OROS is a software development environment). However, our choice is mainly motivated by how to organize and manage the knowhow-bearing entities in *database systems*. Furthermore, our approach is based on a well-defined formal semantics for both hierarchies, while those systems are not. For example, subtyping relationships in CommonObjects may not form a lattice because transitivity is not guaranteed. The approach of Exemplar-based Smalltalk seems to be aimed to separate "class variables" and "class methods" from "instance variables" and "instance methods". The approach of POOL-I uses inheritance among classes and subtyping among types. In contrast, our approach is based only on classes, and is thus closer to real-world analogies (e.g., "DOG is like a PERSON").

2.5.3 Identifying Arbitrary Behavior Objects

In conventional programming languages (such as C and Pascal), functions or procedures in a linked-in object library are uniquely identified by their names (e.g., `fseek`, `ftell` in C). In other words, a library function name linked in an application identifies a unique fragment of executable code. In true object-oriented systems, however, a function (or procedure) might have more than one version implemented (i.e., method names may be overloaded) — this is how polymorphism is supported and why dynamic binding is needed in object-oriented systems. For example, in C++, suppose a virtual function `show_profile` is defined in class PERSON and is also redefined in class EMPLOYEE which is a subclass (called a derived class in C++) of PERSON. Then, when function `show_profile` is used to send the "show profile" message to a set of persons (some of them may be employees too), one of the two implementations will be invoked by the message for each member of the set depending on whether that member is a person or an employee. As we can see, in object-oriented systems, it is inappropriate to use function (or method) names to identify different behavioral entities in their own right.

An optional approach is to directly use the executable code (e.g., function pointers in C++) or the source code (e.g., one file for one implementation) to identify arbitrary behavior objects. However, such an approach cannot guarantee an immutable identification. The KBO model chooses to use (system-generated) object-identifiers to identify arbitrary behavior objects mainly for three reasons: (1) they are guaranteed to be immutable; (2) since every regular object has such an object-identity, so should every behavior object, in order to preserve uniformity (every behavior object in the

KBO model is also a regular data object); and (3) because an executable reference (a knowhow) is made an intrinsic part of a behavior object, different objects with different values (selectors) may share the same executable reference (i.e., knowhow sharing between two different behavior objects). This feature cannot be achieved if the identification of behavior objects is based on their source or object code.

2.5.4 Invoking Arbitrary Behavior Objects

Behavior invocations are conventionally based on either method names or executable variables (e.g., function names and function pointers in C). The behavior objects in the KBO model can be invoked based on their identities or their values. This approach can effectively simulate both of the conventional approaches, since the value of such an object can be viewed as its name or selector, and the identity of such an object can be viewed as the handle accessing the actual knowhow (executable code) carried by the behavior object. We will show in the thesis that such an invocation approach naturally supports both monomorphism (when the invocation is based on object identities) and polymorphism (when the invocation is based on object values).

2.5.5 Combining Arbitrary Behavior Objects

As we observed earlier, a few existing OODBs, such as VISION [CS87, CS88] and IRIS [F*87, F*90], do treat their functions as database objects under a system-defined type Function. Following this observation, some related questions come naturally: Can we have "complex function objects" built from those "function objects" in much the same way a regular complex object can be built? If so, what behavioral semantics do these "complex function objects" imply? (Are they still directly invocable? It is only natural to expect that a complex behavior object preserves the executability.) What are the equalities for these "complex function objects"? These issues or possibilities have not been investigated in those data models.

There are basically two conventional approaches to combining arbitrary behaviors (not objects) in OODB database languages. The first approach is to simply use nested function calls in a query language. This is the approach used in RELOOP, a query language for the O_2 system [C*89]. For example, one can formulate a query like:

select *family*
from *family* **in** FAMILY
where *city*(*address*(*husband*(*family*))) = "*Paris*"

where *city*, *address* and *husband* can be arbitrary behaviors (they are unary functions in this case). The second approach is to use operator constructors that can combine several arbitrary behaviors into one single operation. FAD, a simple database language [B*87b], exhibits such an approach. For example, one of FAD's operator constructors is **pump**(f, g, set_object) where f and g are two behaviors (in this case, f has to be a unary function and g has to be a binary function). The **pump** constructor supports parallelism by a divide and conquer strategy (see [B*87b] for details).

In the KBO model, since arbitrary behaviors are treated as identity-based database objects, we use complex object constructors to build "complex behaviors". With this approach, building some common forms of complex behaviors is no more difficult for end-users to master than building complex objects. Because these complex behaviors are also regular complex objects, they can be manipulated and generated *associatively* through a query facility. Thus, end-users may be able to develop non-trivial applications without relying on application programmers. This promotes a bootstrapping approach for end-users in which complex behaviors can be progressively constructed and/or associatively selected to form more useful, immediately executable applications. Another advantage of this approach is that end-users may be able to achieve greater productivity by viewing their complex messages (they are in fact simple queries) as the production of a long-lived corporate asset rather than just some quick "ad-hoc" effort to obtain the current results. As a simple example, a complex object (a sequence object) with value $<$ "*husband*", "*address*", "*city*" $>$ is given a complex behavior semantics such that, if sent to a receiver, say *family*, it will first send "*husband*" to *family*, and then send "*address*" to the result of sending "*husband*", and finally send "*city*" to the result of sending "*address*". Intuitively, this particular complex object implies a "composite knowhow" that knows how to obtain the name of the city where the husband of a family is currently living. Obviously, such a complex object can be directly manipulated by the query facility.

2.5.6 Manipulating Arbitrary Behavior Objects

In this thesis, we are not interested in extending the power of any programming languages (they are already computationally complete). We are interested in algebraic query models for OODBs. The need for such a way of extracting information from OODBs *without having to write a program* has recently been recognized in many papers, such as [B*89, C*89, F*87, Osb88, SZ89, RS87]. Other query languages (maybe more declarative ones) can be later developed, and their queries can be mapped into equivalent algebraic expressions because query transformations are more readily dis-

covered and applied in an algebraic framework.

In particular, we are very interested in how Osborn's object algebra [Osb88, Osb89a, Osb89b] can be modified and extended to uniformly manipulate a new notion of objects that unifies regular objects, arbitrary behaviors and messages. We believe that the polymorphic nature of this algebra is an appealing feature, since it provides a single query tool for all kind of objects describable in a data model. Our ultimate goal in this aspect is to design an "end-user oriented" query language (something like SQL for relational databases) that provides a high-level, associative manipulation tool for both data, behaviors and messages in OODBs. With such a goal in mind, we provide the KBO model with a formal object algebra, called the KBO algebra, which manipulates in a uniform fashion all the objects describable in the KBO model. To the best of our knowledge, none of existing query models for OODBs supports associative manipulation (and composition, invocation) of data-like complex behaviors.

2.5.7 Summary

The following table summarizes the choices made in the KBO model concerning the six fundamental issues in managing arbitrary behaviors as database objects.

Choices of the KBO Model

Managing Arbitrary Behaviors as Database Objects	
classification of these objects	based on *annotation structures*
reuse of these objects	based on *an independent hierarchy*
identification of these objects	based on *(system-generated) object identities*
invocation of these objects	based on *object identities or object values*
combination of these objects	based on *complex object structures*
manipulation of these objects	based on *a formal object algebra*

As we will see, these basic choices allow us to achieve the very goal of this research — to unify data, arbitrary behaviors, and messages into a uniform notion of objects so that a single database approach can be developed to manage and manipulate such objects. This thesis will formally demonstrate that such an approach is possible. To the best of our knowledge, none of the existing OODBs has provided the above features for arbitrary behaviors, and it is our conclusion that this research is a first step towards the development of a viable OODB that can directly manage and manipulate arbitrary behaviors within a unified object environment.

CHAPTER 3

THE KBO MODEL: PRELIMINARIES

3.1 Introduction

In this chapter, we present the underlying domains and conceptual abstractions in the KBO model. The focus here is on a conceptual level semantics for the model. In other words, we are concerned with choosing basic constructs of ˙he model, disregarding, for the moment, the problems of their syntactic representation. The notation of first order logic will be used extensively in our definitions.

3.2 Underlying Domains

To begin a formal development of the KBO model, we introduce several basic underlying domains which the model abstractions will range over.

Definition 1: *Object Identities, Class Names, and Attribute Labels.* Let $\mathcal{I}$ be a countably infinite set of symbols called ***object-identities***, let $\mathcal{C}$ be a countably infinite set of symbols called ***class names***, and let $\mathcal{A}$ be a countably infinite set of symbols called ***attribute labels***, where $\mathcal{C} \cap \mathcal{A} = \emptyset$, $\mathcal{A} \cap \mathcal{I} = \emptyset$, and $\mathcal{C} \cap \mathcal{I} = \emptyset$. ■

Definition 2: *Base Classes.* Let $\mathcal{C}_{base} = \{D_1, \ldots, D_n\} \subset \mathcal{C}$ be a finite set of *base class* symbols, such that for each base class D_i, there is a constant domain $dom(D_i)$ associated with it, and $\forall D_i, D_j \in \mathcal{C}_{base} : dom(D_i) \cap dom(D_j) = \emptyset$. We denote $\mathcal{D} = \cup_{i=1}^{n} dom(D_i)$, and elements in $\mathcal{D}$ are called ***atomic values***. ■

In our examples, we will assume three predefined base classes: INT for integers (e.g., 530), TEXT for text strings enclosed in double quotes (e.g., "data model"), and BOOL for $\{\textbf{True}, \textbf{False}\}$.

An object-identity itself is not an object — it only identifies a unique KBO object in the object universe, which is defined as follows:

Definition 3: *Object Universe.* Let $\mathcal{O}$ be the ***object universe***, the set of all KBO objects, such that (1) there exists a bijection ***oid*** from $\mathcal{O}$ onto $\mathcal{I}$ which yields the object identity of each object in $\mathcal{O}$, and (2) there exists a total function *orf* from $\mathcal{O}$ into $2^{\mathcal{I}}$ that yields the ***object-identities reachable from*** each object in $\mathcal{O}$, such that:

(a) for any $o \in \mathcal{O}$, $orf(o)$ is a finite subset of $\mathcal{I}$

(b) $\forall o \in \mathcal{O}: orf(o) \neq \emptyset \Longrightarrow \exists id \in orf(o): orf(oid^{-1}(id)) = \emptyset$

(c) $\forall o \in \mathcal{O}, \forall id \in orf(o): orf(oid^{-1}(id)) \subseteq orf(o)$

where function oid^{-1} is the inverse of function oid. ■

According to this definition, each object has a unique object-identity that *identifies* the object, and each object also has a set (including the empty set) of object-identities reachable from the object that *constitutes* the object. Furthermore, axiom (b) implies that if an object can reach several other objects then at least one of those objects should reach no other objects. Axiom (c) implies that if an object o_1 can reach an object o_2 then o_1 can reach all the objects reachable from o_2.

The following properties of the object universe $\mathcal{O}$ can be immediately derived.

Theorem 1: *No object can reach only itself. That is,*

$$\forall o \in \mathcal{O} : orf(o) \neq \{oid(o)\}$$

Proof: Suppose there exists $o \in \mathcal{O}$ such that $orf(o) = \{oid(o)\}$. Then, there is no $id \in orf(o)$ such that $orf(oid^{-1}(id)) = \emptyset$ — a contradiction to Axiom (b) above. ■

Theorem 2: *If an object o can reach some objects, then o can always reach more objects than some of those objects can. That is,*

$$\forall o \in \mathcal{O} : orf(o) \neq \emptyset \Longrightarrow \exists i \in orf(o) : orf(o) \supset orf(oid^{-1}(i))$$

Proof: Suppose object o can reach some objects (that is, $orf(o) \neq \emptyset$) but o cannot reach more objects than any x in $orf(o)$ can, that is, $orf(o) \not\supset orf(x)$. From Axiom (c) above, we know that $orf(o) \supseteq orf(x)$ must hold. Therefore, we can only have $orf(o) = orf(x)$. Since $orf(o) \neq \emptyset$, then for all objects $oid(x) \in orf(o)$, $orf(x) \neq \emptyset$ — a contradiction to Axiom (b) above. ■

Example 1: Let symbols $Name_1, Name_2, Age_1, Age_2, Lee, Susan, NiceCouple$ be object-identities in $\mathcal{I}$. Let $o_{Name_1}, o_{Name_2}, o_{Age_1}, o_{Age_2}, o_{Lee}, o_{Susan}, o_{NiceCouple}$ be objects in $\mathcal{O}$, where

$oid(o_{Name_1}) = Name_1$
$oid(o_{Name_2}) = Name_2$
$oid(o_{Age_1}) = Age_1$
$oid(o_{Age_2}) = Age_2$
$oid(o_{Lee}) = Lee$
$oid(o_{Susan}) = Susan$
$oid(o_{NiceCouple}) = NiceCouple$

A valid reachability from these objects is given as follows:

$orf(o_{Name_1}) = \emptyset$
$orf(o_{Name_2}) = \emptyset$

$$orf(o_{Age_1}) = \emptyset$$
$$orf(o_{Age_2}) = \emptyset$$
$$orf(o_{Lee}) = \{Name_1, Age_1\}$$
$$orf(o_{Susan}) = \{Name_2, Age_2\}$$
$$orf(o_{NiceCouple}) = \{Lee, Susan, Name_1, Name_2, Age_1, Age_2\}$$

As we can see, object $o_{NiceCouple}$ can reach six objects identified by Lee, $Susan$, $Name_1$, $Name_2$, Age_1, Age_2, and these six objects constitute the object $o_{NiceCouple}$. ■

Example 2: The following reachabilities for objects o_{Lee} and o_{Susan} are invalid:

Reachability 1: $orf(o_{Lee}) = \{Lee\}$ $\quad$ $orf(o_{Susan}) = \{Susan\}$

Reachability 2: $orf(o_{Lee}) = \{Lee, Susan\}$ $\quad$ $orf(o_{Susan}) = \{Susan, Lee\}$

Reachability 1 is invalid according to Theorem 1, and Reachability 2 is invalid according to Theorem 2. ■

Based on the reachability of objects, we can further divide the object universe $\mathcal{O}$ into two disjoint sets:

Definition 4: *Capsule Objects and Complex Objects.* An object $o \in \mathcal{O}$ is a *capsule object* if $orf(o) = \emptyset$. An object o is a ***complex object*** if $orf(o) \neq \emptyset$. Consequently, we can have $\mathcal{O} = \mathcal{O}_{capsule} \cup \mathcal{O}_{complex}$ where $\mathcal{O}_{capsule}$ is the set of all capsule objects and $\mathcal{O}_{complex}$ is the set of all complex objects. ■

The intended interpretation of capsule objects is that they cannot be decomposed into other objects. That is, the contents of capsule objects are not further interpreted by the KBO model, in the sense that their representation is totally encapsulated and they are guaranteed (by their implementors) to be printable and equality-testable. Intuitively, the printable form of a capsule object is an atomic value, such as 1991 or "cat and dog".

Before we continue to introduce other underlying domains, we shall first define the notion of a *knowhow*, one of the most essential concepts in the KBO model.

Definition 5: *Knowhows.* A ***knowhow*** is a fragment of executable code implemented in an arbitrary general-purpose programming language. A knowhow k has a *formal output specification* denoted by $fos(k)$, $fos(k) \in \mathcal{C} \cup \{\text{SELF}\}$, and a *formal input specification* denoted by $fis(k)$, $fis(k) \subset \mathcal{A} \times \mathcal{C}$, such that $fis(k) = \{(L_1, c_1), \ldots, (L_n, c_n)\}$ and $\forall i, j \in [1..n] : i \neq j \Longrightarrow L_i \neq L_j$. The invocation of a knowhow k is denoted by the following abstract syntax:

$$o_{receiver}.\overline{k}(L_1\colon o_1, \ldots, L_n\colon o_n)$$

where $\overline{k}$ denotes the execution of k, and $o_{receiver}, o_1, \ldots, o_n$ $(n \geq 0)$ are objects in $\mathcal{O}$.

Here we read that a *receiver object* $o_{receiver}$ is exhibiting a knowhow k which takes as input n *argument objects* $o_1, ..., o_n$ labeled and classified by $fis(k)$, and produces as output a *result object* classified by $fos(k)$. In particular, when $fos(k)$ = SELF, the result object is $o_{receiver}$ itself. ■

The special keyword SELF implies that knowhow k is not intended to retrieve a result different from the receiver $o_{receiver}$ itself (e.g., knowhows that only display or print); it may also cause side-effects (e.g., knowhows that update).

Three comments about the above definition are in order here. First, we do not distinguish argument labels from attribute labels. Labeled arguments will facilitate argument sharing when a set of behaviors are invoked (this feature will be illustrated later in the thesis). Second, the above definition implies that each knowhow has a *strongly typed* signature; this is mainly intended to provide support for model documentation and run-time safety. Third, a low-level view of a knowhow k is to think of it as a function address (e.g., a function pointer in C or C++) which can be dereferenced (denoted by $\overline{k}$) for execution with $n + 1$ arguments $o_{receiver}, o_1, ..., o_n$.

We now continue to introduce other underlying domains in the KBO data model.

Definition 6: *Knowhow Universe.* Let $\mathcal{K}$ be a countably infinite set of *knowhows* such that there exists an injection *bb* from $\mathcal{K}$ into $\mathcal{O}$ which associates each knowhow $k \in \mathcal{K}$ with a unique "biological bearer" $bb(k) \in \mathcal{O}$. Let $\mathcal{K}^*$ be the *knowhow universe*, $\mathcal{K}^* = \mathcal{K} \cup \{\phi_\perp\}$, where $\phi_\perp$ is called *the empty knowhow*, such that there exists a surjection kh from $\mathcal{O}$ onto $\mathcal{K}^*$ which associates every object $o \in \mathcal{O}$ with a knowhow $kh(o) \in \mathcal{K}^*$. ■

Intuitively, for each knowhow k, an object $bb(k)$ must be created by k's implementor; this is intended to capture the idea that a hardcoded computation should be viewed, right upon its creation, as a true object in its own right. Also, *bb* is defined as an injection from $\mathcal{K}$ into $\mathcal{O}$ because (1) we require every knowhow, right after it is implemented (and becomes invocable), to be associated with (or "borne by") an object, which is therefore the biological bearer of the knowhow; and (2) we allow two kinds of objects that are not biological bearers — they are either regular data objects (i.e., an employee object) or behavior objects that borrow their knowhows from other behaviors (e.g., a behavior "Assign chief supervisor" attached to a class PHD-STUDENT may borrow the knowhow from a behavior "Assign advisor" attached to a class MSC-STUDENT). With the mapping *bb*, we can determine where a knowhow originally comes from. Furthermore, we define kh to be a surjection from $\mathcal{O}$ onto $\mathcal{K}^*$ because (1) there can be more than one behavior sharing the same knowhow in $\mathcal{K}$ (one of them must be the biological bearer of that knowhow); and (2) objects that

are only intended to be data will be given the same knowhow $\phi_\perp$ from $\mathcal{K}^*$. Notice that the empty knowhow $\phi_\perp$ only knows how to do nothing; its execution will result in failure.

The notion of knowhows is introduced so that we can represent the encapsulation of the implementation of a computation, because all we can see is an executable reference (the knowhow) which identifies a computation that **knows how** to do things for a receiver object (e.g., "flip" and "rotate" for a window object); how the computation is crafted is a secret kept by its implementor. In this way, the hardcoded portion of behaviors in OODBs can be completely insulated from data modeling facilities. The primary motive for pursuing this point of view is that: (1) we want to isolate the description and execution of a knowhow from its implementation details, (2) our model development can avoid certain idiosyncratic features of the specific programming language that implements knowhows, (3) a multi-language framework can be assumed for knowhow implementations, and (4) the implementation (i.e., the source code) of a knowhow can be changed or maintained presumably without affecting database applications.

Definition 7: *Objects.* An object $o \in \mathcal{O}$ is a triple $o = (id, v, k)$, where $id = oid(o)$ is the object-identity of o, $k = kh(o)$ is the knowhow of o, and v is the value of o such that $v \in \mathcal{D}$ or $v = [A_1: id_1, ..., A_n: id_n]$ or $v = \{\!| id_1, id_2, id_3 |\!\}$ or $v = \{id_1, ..., id_n\}$ or $v = <id_1, ..., id_n>$, where $A_i \in \mathcal{A}$ and $id_i \in \mathcal{I}$. When $v \in \mathcal{D}$, o is a capsule object; otherwise, o is one of the four kinds of complex objects:

- *aggregate object* (i.e., when $v = [A_1: id_1, ..., A_n: id_n]$),
- *bi-object* (i.e., when $v = \{\!| id_1, id_2, id_3 |\!\}$),
- *set object* (i.e., when $v = \{id_1, ..., id_n\}$), or
- *sequence object* (i.e., when $v = <id_1, ..., id_n>$).

■

Notice that, at this point, the four kinds of complex objects are defined only based on the forms of their values; no modeling semantics has been given to them yet. Furthermore, with this definition we can explicitly define the function orf such that

1. $orf(o) = \emptyset$ if o is a capsule object;
2. $orf(o) = \cup_{i=1}^{3} orf(oid^{-1}(id_i)) \cup \{id_1, id_2, id_3\}$ if o is a bi-object; and
3. $orf(o) = \cup_{i=1}^{n} orf(oid^{-1}(id_i)) \cup \{id_1, \ldots, id_n\}$ if o is an aggregate object, a set object or a sequence object.

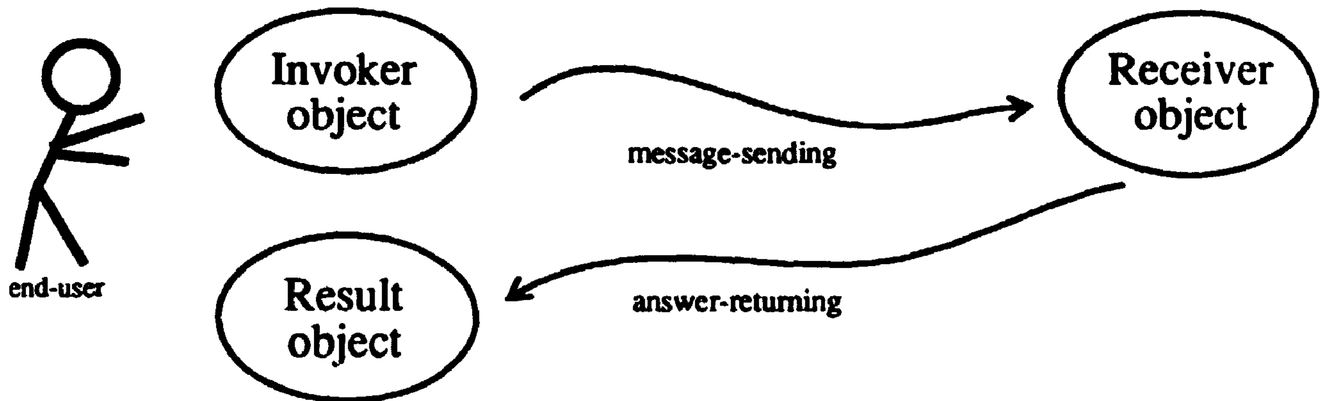


Figure 3.2: A Naive Message-Sending Paradigm

In the KBO model, because of the existence of the knowhow assignment kh, the world is viewed as composed of *"knowhow-bearing-objects"*. Conventional "data" become objects bearing the empty knowhow $\phi_\perp$, and conventional "behaviors" become objects bearing a non-empty knowhow. However, we must point out that, because of the immutability of object-identity, a "data" object can become a "behavior" object by gaining a non-empty knowhow, and, similarly, a "behavior" object can become a "data" object by losing its non-empty knowhow. For example, an integer object with value 911 may later be given a knowhow that knows how to find an emergency centre for a phone object.

3.3 A Naive Message-Sending Paradigm

In the KBO model, the execution of a knowhow is caused by a *message-sending*. To provide some background for the formal definition of the message-sending paradigm in the KBO model, in the following we describe a "naive" message-sending paradigm.

Let us assume that all knowhows take no labeled arguments. That is, $fis(k) = \emptyset$ for any $k \in \mathcal{K}$. Then, a *naive message-sending* is an expression of the form:

$$o_{invoker} \leadsto o_{receiver} \Rightarrow o_{result}$$

where the object $o_{invoker}$ (called the invoker) is sent by the end-user as a message to the object $o_{receiver}$ (called the receiver), which responds to the message by returning the object o_{result} (called the result) such that $o_{result} = o_{receiver}.\overline{kh(o_{invoker})}()$. Recall that $kh(o_{invoker})$ is the knowhow of the object $o_{invoker}$. Figure 3.2 illustrates the naive message-sending paradigm. For example, suppose we have two TEXT objects, o_1 with an atomic value "get 1st word" and o_2 with an atomic value "data model". Assume

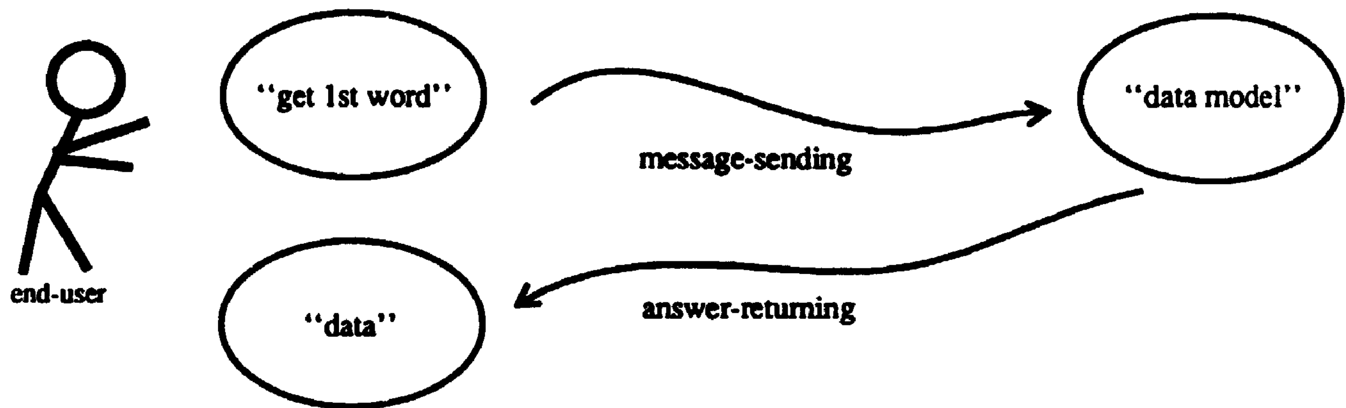


Figure 3.3: Sending object "get 1st word" to object "data model"

that the knowhow $kh(o_1)$ knows how to extract the first word in a receiver (which must be a TEXT object). Then, we would have $o_1 \leadsto o_2 \Rightarrow o_3$ where o_3 has an atomic value "data" (see Figure 3.3). Furthermore, we would also have $o_1 \leadsto o_1 \Rightarrow o_4$ where o_4 has an atomic value "get".

It should be observed that an unconventional part of this paradigm is that what is traditionally called the "message name" is replaced by an object. In doing so, we give objects the ability to invoke an execution if they are used in the invoker role of a message-sending. However, this particular message-sending paradigm is "naive" because (1) it assumes that no other arguments are needed by the execution of a knowhow (except the receiver itself); (2) it is monomorphic because it tells the receiver to execute the knowhow of the invoker, that is, the knowhow to be executed is pinpointed by the invoker; and (3) it allows *any* receiver to execute the knowhow of the invoker, that is, there is no safety checking. We thus need to further extend this message-sending paradigm. As we will see, these three problems are solved later in Chapter 4 and 6. Nevertheless, the intention here is to show, in a naive way, that our message-sending will be modeled by sending an object (the invoker) to another object (the receiver), so that our objects will not only play passive roles (as receivers) but also play active roles (as invokers).

3.4 A Conceptual Level Semantics

The KBO model provides five high-level abstractions for real-world data modeling: classification is realized by the INSTANCE-OF relationship, vitalization by the BEHAVIOR-OF relationship, generalization by the KIND-OF relationship, re-utilization

by the REUSE-OF relationship, and aggregation by the PART-OF relationship. It is very important to notice that these abstractions are *prescriptive* relationships, that is, they are explicitly declared by OODB users.

3.4.1 Classification and Vitalization

Definition 8: *INSTANCE-OF (Classification).* The INSTANCE-OF abstraction, written ⓘ, is a binary relation over $\mathcal{O} \times \mathcal{C}$ such that $\forall o \in \mathcal{O}, \exists c \in \mathcal{C} : o$ ⓘ c, where o is called an *instance* of c. ■

Definition 9: *BEHAVIOR-OF (Vitalization).* The BEHAVIOR-OF abstraction, written ⓑ, is a binary relation over $\mathcal{O} \times \mathcal{C}$ such that $\forall o \in \mathcal{O}, \exists c \in \mathcal{C} : o$ ⓑ c, where o is called a *behavior* of c. ■

The INSTANCE-OF abstraction categorizes objects into classes. It requires that every object belong to at least one class. The BEHAVIOR-OF abstraction captures the idea that knowhow-bearing objects can also be categorized by what they vitalize -- an object may exhibit its knowhow only for classes of which the object is a behavior. It requires that every object vitalize at least one class [6].

Example 3: Some instances and behaviors of the classes PERSON and PERSONAL-QUESTION are shown in Figure 3.4. It shows that two objects, o_1 and o_2, are instances of class PERSONAL-QUESTION whose values are "How old are you?" and "Are you married?", respectively. Moreover, Figure 3.4 shows that objects o_1 and o_2 are also two behaviors of class PERSON. Intuitively, what this means is that, if a message with value "How old are you?" is sent to a PERSON object, then the PERSON object will invoke the knowhow of o_1, i.e., $kh(o_1)$, to respond to the message. More precisely, $kh(o_1)$ will be executed and a result object be returned (e.g., an integer 23). A similar explanation can be given for the other behavior o_2. Notice that instances of PERSON and behaviors of PERSONAL-QUESTION are not depicted in Figure 3.4. For example, class PERSONAL-QUESTION may have its own behavior, say, o_3 with value "about marriage?" whose semantics is to return **True** if one the words "wife", "husband", "marriage" and "married" is a substring of an instance of PERSONAL-QUESTION. In such a case, we would have $o_3 \rightsquigarrow o_2 \Rightarrow$ **True**. ■

With the INSTANCE-OF and the BEHAVIOR-OF abstractions, we introduce the notions of interfaces and populations of classes.

[6] In particular, objects with the empty knowhow only "vitalize" a bottom class called BOTTOM, which is defined later in this chapter.

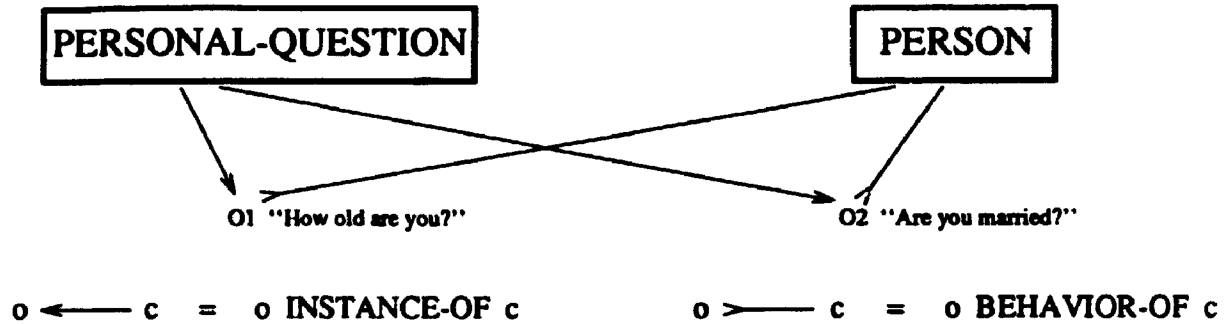


Figure 3.4: An Example of Instances and Behaviors

Definition 10: *Interfaces and Populations.* For any class $c \in \mathcal{C}$, its *interface*, denoted $\dot{\Sigma}(c)$, is the set of all its behaviors, i.e., $\dot{\Sigma}(c) = \{o \mid o \textcircled{b} c\}$, and its *population*, denoted $\dot{\Omega}(c)$, is the set of all its instances, i.e., $\dot{\Omega}(c) = \{o \mid o \textcircled{i} c\}$. ∎

The notions of interfaces and populations will be used later to develop other definitions. Particularly, the notion of an interface plays a very important role in our final definition of a message-sending in Chapter 6.

3.4.2 Generalization and Re-utilization

One of the common features of OODBs is the support of a single "inheritance hierarchy" which allows the user to create new classes incrementally from existing, less specialized classes. Although the one-hierarchy inheritance notion is simple and easy to use, it appears to have several potential problems: (1) susceptiveness to incomprehensible conceptual semantics, (2) susceptiveness to class space pollution, and (3) rigorousness of the overriding mechanism that restricts objects to present only a single view of their behaviors.

The first problem arises when users try to take advantage of the code reusability implied by the inheritance hierarchy. For example, it is very easy for a user to make a new class GORILLA a subclass of an existing class BEAR simply because they "run" and "eat" in a similar manner. However, the specialization relationship between GORILLA and BEAR is conceptually incorrect. In database systems, this often means that when a query is issued against the class BEAR, you will see some gorillas as well because a GORILLA "is a" BEAR. Similarly, you might see some cats and dogs when you query against class PERSON if the creators of CAT and DOG have chosen PERSON's behaviors to personalize cats and dogs. Another consequence is that unexpected security violations might happen. For example, instances of BNR-RESEARCHER would be retrieved by a query against IBM-RESEARCHER if BNR-

RESEARCHER were made a subclass of IBM-RESEARCHER simply for code reuse reasons.

One solution might be to move BEAR's behaviors to its superclass, say, ANIMAL and then let GORILLA inherit from ANIMAL. This is usually unrealistic both at the modeling level and at the implementation level. At the modeling level, the behavior "run" and "eat" of BEAR may not be general enough to be inherited by the majority animals. At the implementation level, moving "run" and "eat" from BEAR into ANIMAL means overwriting the behaviors "run" and "eat" attached to ANIMAL (if any) and re-implementing new "run" and "eat" in ANIMAL because they now rely on ANIMAL's representation.

Another solution might be to create a new class, say BEAR-OR-GORILLA, that has the desired "eat" and "run" behaviors and then let both BEAR and GORILLA inherit from the new class. However, because this kind of new class would have no direct instances, it is an example of the second problem we pointed out — class space pollution. If every time we encounter this situation we create a new class, very soon we will find our class hierarchy being polluted with many meaningless classes. In addition, frequent schema change could be very costly.

The third problem, the problem of a single view of object behaviors, refers to the overriding mechanism assumed by the inheritance hierarchy. In existing OODBs, if a class implements a method, then it overrides any implementations for the methods with the same name in the class's superclasses. Consider the following five classes: PROJECT-MANAGER is-a SYSTEM-ANALYST is-a PROGRAMMER is-a EMPLOYEE is-a PERSON, where "X is-a Y" means "X is a subclass of Y". Each of the five classes has a distinct implementation to support a behavior named "Introduce yourself". When the message "Introduce yourself" is sent to an instance of PROJECT-MANAGER, say *Mike*, we can only see *Mike* introducing himself as a "project manager" because the implementation invoked is the one designed for PROJECT-MANAGER (due to the overriding mechanism). In many situations, we would like a receiver to show a multiple view of the same behavior, such as the request "Introduce *Mike* as a person and forget about other details!". In most existing object-oriented systems, this requires a change in the message. For instance, the extended Smalltalk [BI82] uses the message of the form `class-name.method-name` to select a method implemented in `class-name`. The main problem with this kind of approach in OODBs is that polymorphism may not be properly supported in the case where a message is sent a set of receivers. For example, if we send "PROGRAMMER.Introduce yourself" to a set of PERSONs, those in the set who are only

employees or persons would not be able to respond to this message. The impact of this problem goes beyond mere inconvenience. Given run-time binding, an application program (or an end-user) accessing an OODB should be able to send the message "Introduce yourself" to objects whose type is not known in advance, leaving it to the object to work out how best to "introduce" itself.

In the KBO model, generalization/specialization among knowhow-bearing-objects is modeled by a "KIND-OF" hierarchy, and knowhow reusing among these objects is modeled by a "REUSE-OF" hierarchy. We will show that these two hierarchies not only provide two orthogonal ways to organize KBO objects but also effectively overcome the problems described above.

Definition 11: *KIND-OF (Generalization).* The **KIND-OF** abstraction, written Ⓚ, is a binary relation over $\mathcal{C} \times \mathcal{C}$ such that for any two classes $(c_1, c_2) \in \mathcal{C} \times \mathcal{C}$, c_1 Ⓚ c_2 if and only if both (1) and (2) hold:

(1) $\forall o \in \mathcal{O} : o$ ⓘ $c_1 \Longrightarrow o$ ⓘ c_2

(2) $c_1 \neq c_2 \Longrightarrow \exists o \in \mathcal{O} : o$ ⓘ $c_2 \wedge \neg(o$ ⓘ $c_1)$

When c_1 Ⓚ c_2, c_1 is called a ***kind*** of or a *subclass* of c_2 and c_2 is called a *superclass* of c_1. ■

Notice that $\mathcal{O}$ is the universe of all *possible* objects; it is not the set of existing objects in a particular database.

The **KIND-OF** abstraction is intended to capture the semantic database notion of "IS-A" [HK87]. Axiom (1) in the definition allows us to say, for example, that in any given database an EMPLOYEE instance "is a" PERSON instance too. Axiom (2) in the definition manifests the fact that, for example, in our possible world there may always be some PERSON instances that are not an EMPLOYEE instance. In other words, Axiom (2) implies that if c_2 is a generalization of c_1 (i.e., c_1 Ⓚ c_2) then c_2 will, conceptually, either have some instances of its own (direct instances) or have other specializations (subclasses) besides c_1 (if neither is true then c_1 and c_2 are essentially the same concepts). Here, we stress the point that a class is created to model real-world entities, rather than to provide code for other classes to inherit. In sum, a class c_1 is a kind of class c_2 if and only if every instance of c_1 is ***semantically necessarily*** an instance of c_2. For example, COMPUTER-JOURNAL is a kind of JOURNAL, but NEWSPAPER is not a kind of JOURNAL even though they may be the same structurally (e.g., both of them are composed of two attributes, Name and Publisher).

Definition 12: *REUSE-OF (Re-utilization).* The **REUSE-OF** abstraction, written Ⓡ, is a binary relation over $\mathcal{C} \times \mathcal{C}$ such that for any two classes $(c_1, c_2) \in \mathcal{C} \times \mathcal{C}$, c_1

$\textcircled{r}\ c_2$ if and only if both (1) and (2) hold:

$$(1) \quad \forall o \in \mathcal{O} : o \,\textcircled{i}\, c_2 \Longrightarrow o \,\textcircled{i}\, c_1 \vee (\exists o' \in \mathcal{O} : o' \,\textcircled{i}\, c_1 \wedge kh(o') = kh(o))$$

$$(2) \quad c_1 \neq c_2 \Longrightarrow \exists o, o' \in \mathcal{O} : o \,\textcircled{i}\, c_1 \wedge o' \,\textcircled{i}\, c_2 \wedge kh(o') \neq kh(o)$$

When $c_1 \,\textcircled{r}\, c_2$, c_1 is called a *reuse* of or a ***demandclass*** of c_2 and c_2 is called a *supplyclass* of c_1. ■

In this definition, axiom (1) states that, for each behavior of a supplyclass, either itself (i.e., with the same object-identity) becomes a behavior of a demandclass or only its knowhow (not its object-identity and value) becomes the knowhow of another behavior of the demandclass. Axiom (2) states that, conceptually, a demandclass may always have a behavior whose knowhow is not borrowed from a behavior of a supplyclass.

The REUSE-OF abstraction captures the idea of behavior or knowhow reuse. Intuitively, almost any behavior of a supplyclass will become a behavior of all its demandclasses, except that if a behavior of the supplyclass is not reused directly in a demandclass then at least its knowhow should be reused by the demandclass. It should be observed that there is no explicit "overriding" of reused behaviors (more precisely, their knowhows), because every knowhow of a supplyclass becomes a knowhow of a demandclass. The intuition here is that, since c_1 reuses c_2 as a whole (this implies that the whole data structure of c_2 must be included in c_1), any knowhow of a behavior of c_2 should be invocable on c_1's instances *at the user's discretion*, because c_1 *does have* the representation on which the knowhow depends. For example, if EMPLOYEE is a reuse of PERSON, and PERSON has a behavior "introduce yourself" with knowhow kh_1, and EMPLOYEE redefines "introduce yourself" with knowhow kh_2, then EMPLOYEE actually has two knowhows (i.e., kh_2 and kh_1) and which one to use depends on the invoker in a specific message-sending.

Example 4: Consider the three classes connected through the $\textcircled{i}$ and $\textcircled{r}$ relationships shown in Figure 3.5. In this environment, every instance of MSC-STUDENT and PHD-STUDENT is automatically an instance of STUDENT, because MSC-STUDENT and PHD-STUDENT are subclasses of STUDENT (i.e., an M.Sc. student "is a" student and a Ph.D. student "is a" student). We assume here that STUDENT was created first, MSC-STUDENT second and PHD-STUDENT third, because MSC-STUDENT was created so that it reuses all behaviors from STUDENT and PHD-STUDENT was created so that it reuses all behaviors from MSC-STUDENT. This reuse strategy is modeled by making MSC-STUDENT a demandclass of STUDENT, and PHD-STUDENT a demandclass of MSC-STUDENT. In many universities, the Ph.D. program in Computer Science is newer than the M.Sc. program (e.g., at UWO).

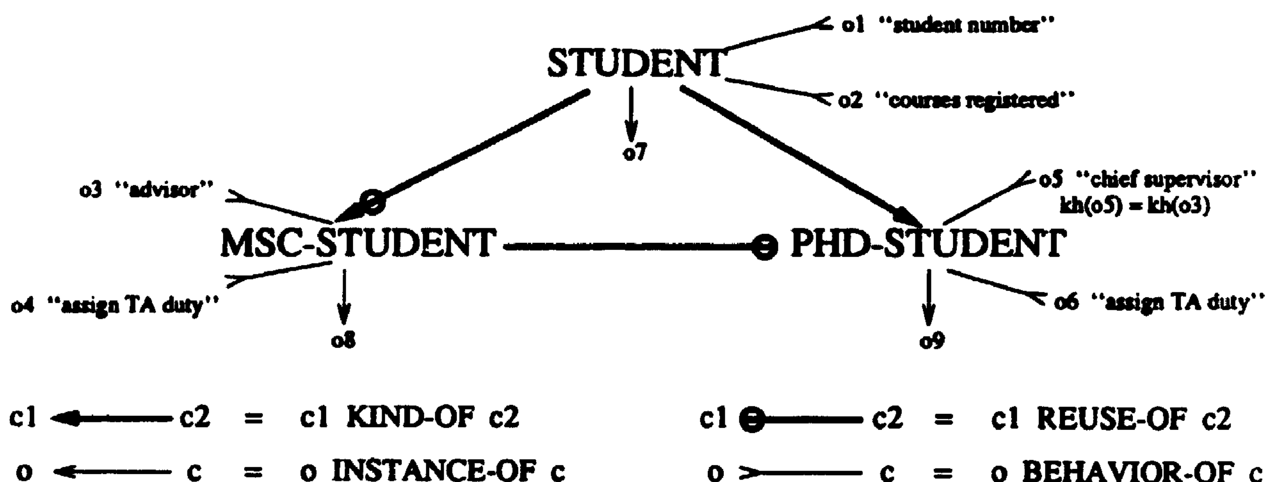


Figure 3.5: Connecting PHD-STUDENT with STUDENT and MSC-STUDENT

Hence, if we do not consider a Ph.D. student to be an M.Sc. student (which is generally true), then the approach shown in Figure 3.5 is often a better way to create the new class PHD-STUDENT, due to the strong similarity between MSC-STUDENT and PHD-STUDENT (such as the behavior "assign TA duty"). Notice that PHD-STUDENT also reuses all behaviors of STUDENT through MSC-STUDENT due to the transitivity of the REUSE-OF relation. Figure 3.5 shows that classes STUDENT, MSC-STUDENT and PHD-STUDENT each have two direct (locally defined) behaviors, namely o_1 and o_2 of STUDENT, o_3 and o_4 of MSC-STUDENT, and o_5 and o_6 of PHD-STUDENT. Also, objects o_7, o_8, o_9 are direct instances of STUDENT, MSC-STUDENT and PHD-STUDENT, respectively. Based on our previous definitions, it is not difficult to derive the following interfaces and populations:

$$\begin{array}{ll}
\dot{\Sigma}(\text{STUDENT}) = \{o_1, o_2\} & \dot{\Omega}(\text{STUDENT}) = \{o_7, o_8, o_9\} \\
\dot{\Sigma}(\text{MSC-STUDENT}) = \{o_1, o_2, o_3, o_4\} & \dot{\Omega}(\text{MSC-STUDENT}) = \{o_8\} \\
\dot{\Sigma}(\text{PHD-STUDENT}) = \{o_1, o_2, o_4, o_5, o_6\} & \dot{\Omega}(\text{PHD-STUDENT}) = \{o_9\}
\end{array}$$

In this example, two observations must be made. First, the interface of PHD-STUDENT does not include o_3 even though PHD-STUDENT is a reuse of MSC-STUDENT. This is because the knowhow of o_3 is reused by a local behavior o_5 of PHD-STUDENT, rather than o_3 itself. Hence, o_3 and o_5 share the same knowhow but are viewed by users as two different behaviors: o_3 retrieves the advisor of a master student while o_5 retrieves the chief supervisor of a Ph.D. student. Second, although behavior o_4 of MSC-STUDENT and behavior o_6 of PHD-STUDENT have the same value "assign TA duty", o_4 is not excluded from but included in the interface of PHD-

STUDENT. This means that both o_6 and o_4 are invocable on a Ph.D. student object depending on the object-identity of the message. For example, since o_9 is an instance of PHD-STUDENT, then $o_6 \rightsquigarrow o_9$ will assign o_9 a TA job based on some rules for Ph.D. students (e.g., co-teach a first-year course). However, $o_4 \rightsquigarrow o_9$ will assign o_9 a TA job only based on the rules for a master student (e.g., M.Sc. students will not be assigned to co-teach courses). ■

Example 5: Consider the four classes EMPLOYEE, IBM-EMPLOYEE, APPLE-EMPLOYEE and BNR-EMPLOYEE in an OODB (see Figure 3.6). In this environment, the classes IBM-EMPLOYEE, APPLE-EMPLOYEE and BNR-EMPLOYEE are (naturally) subclasses of EMPLOYEE. Classes IBM-EMPLOYEE and APPLE-EMPLOYEE were created by reusing all behaviors from EMPLOYEE plus their own local behaviors. Of particular interest is the creation of class BNR-EMPLOYEE; it is created by reusing all behaviors from IBM-EMPLOYEE and APPLE-EMPLOYEE, because it is a demandclass of both IBM-EMPLOYEE and APPLE-EMPLOYEE. The designer of BNR-EMPLOYEE probably thinks that the best way to describe the features of a BNR employee is to combine all the features of an IBM employee and an Apple employee. For example, an IBM-EMPLOYEE may have a (local) behavior "Novell/NetWare experience" and an APPLE-EMPLOYEE may have a (local) behavior "AppleTalk experience". In this way, BNR-EMPLOYEE is created with little effort. Notice that, if we query against IBM-EMPLOYEE (e.g., find all IBM employees who are older than 50), instances of BNR-EMPLOYEE will not be inspected simply because BNR-EMPLOYEE is only a demandclass of IBM-EMPLOYEE, not a subclass (i.e., there is no "IS-A" relationship between them). Based on the schema shown in Figure 3.6, the following interfaces can be derived:

$$\dot{\Sigma}(\text{EMPLOYEE}) = \{b_1\}$$
$$\dot{\Sigma}(\text{IBM-EMPLOYEE}) = \{b_1, b_2\}$$
$$\dot{\Sigma}(\text{APPLE-EMPLOYEE}) = \{b_1, b_3\}$$
$$\dot{\Sigma}(\text{BNR-EMPLOYEE}) = \{b_1, b_2, b_3, b_4\}$$

Here, behaviors $b_1, ..., b_4$ are also instances of class TEXT, because they all have a text value. ■

One relevant observation is that the interpretation of the KIND-OF relation is almost a reverse analogy of the interpretation of the REUSE-OF relation. That is, an instance of a subclass is always an instance of its superclasses, while a behavior of a supplyclass is always a behavior of its demandclasses (except those behaviors whose

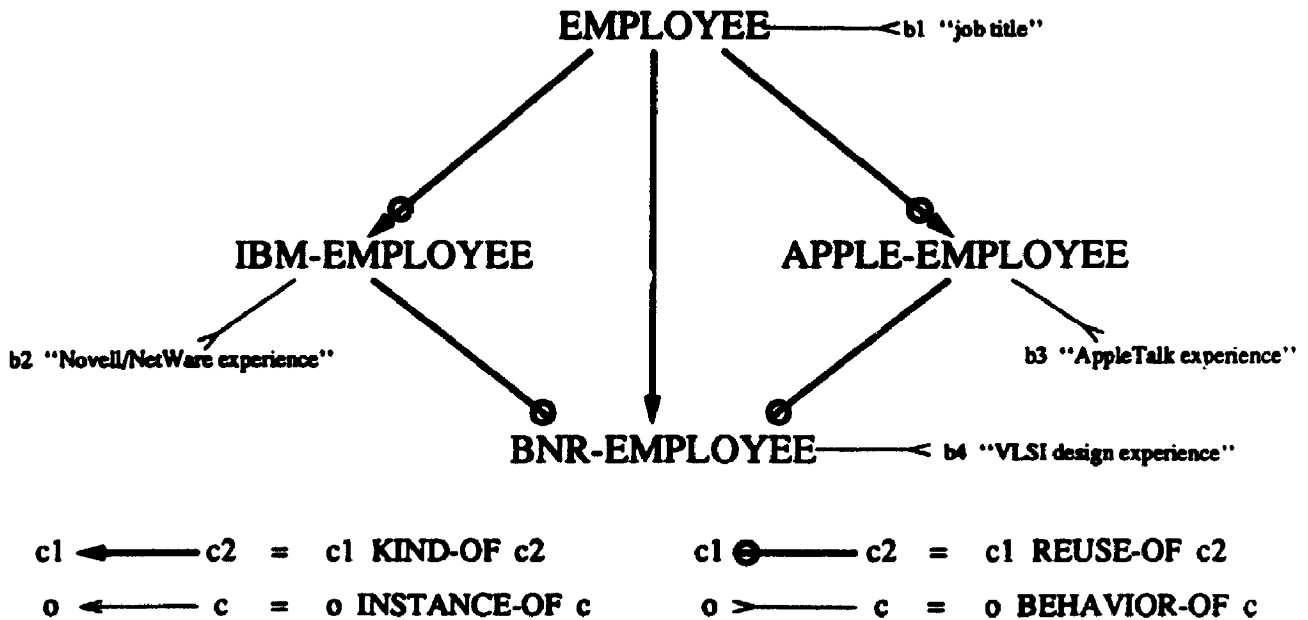


Figure 3.6: Creating BNR-EMPLOYEE from other two classes

knowhows are directly reused, rather than their identities).

From the definition of the Ⓚ and Ⓡ relations, the following properties are very straightforward:

Theorem 3: *The set $\mathcal{C}$ is partially ordered under the KIND-OF relation* Ⓚ.

Proof: From Definition 11, we can derive that c_1 Ⓚ c_2 if and only if (1) $c_1 = c_2$ or (2) $\dot{\Omega}(c_1) \subset \dot{\Omega}(c_2)$. Therefore, Ⓚ is reflexive because of (1), and is also antisymmetric and transitive because $\subset$ is antisymmetric and transitive on the set $\{ \dot{\Omega}(c) | c \in \mathcal{C}\}$. ■

Theorem 4: *The set $\mathcal{C}$ is partially ordered under the REUSE-OF relation* Ⓡ.

Proof: Let $kh(\Theta) = \{kh(o) | \, o \in \Theta\}$ where Θ is a set of objects in $\mathcal{O}$. From Definition 12, we can derive that c_1 Ⓡ c_2 if and only if (1) $c_1 = c_2$ or (2) $kh(\dot{\Sigma}(c_1)) \supset kh(\dot{\Sigma}(c_2))$. Therefore, Ⓡ is reflexive because of (1), and is also antisymmetric and transitive because $\supset$ is antisymmetric and transitive on the set $\{kh(\dot{\Sigma}(c)) | c \in \mathcal{C}\}$. ■

As we have seen, the advantage of introducing the Ⓡ relation is that *classes do not have to reuse knowhows only from their superclasses — they may reuse knowhows from whatever classes meet their needs.* As a result, the Ⓡ relation is in effect a mechanism to maximize the reuse of executable code. It must be noted that users can still enjoy the traditional single class hierarchy by simply making the KIND-OF hierarchy and the REUSE-OF hierarchy identical.

Based on the Ⓚ and Ⓡ abstractions, we define the notion of direct instances, direct behaviors, direct subclasses and direct demandclasses.

Definition 13: *Direct Instances.* An object o is a *direct instance* of a class c, denoted $o \textcircled{i}_d c$, if $o \textcircled{i} c$ and there is no class c' such that $c' \textcircled{k} c$ and $o \textcircled{i} c'$. ■

Definition 14: *Direct Behaviors.* An object o is a *direct behavior* of a class c, denoted $o \textcircled{b}_d c$, if $o \textcircled{b} c$ and there is no class c' such that $c \textcircled{r} c'$ and $o \textcircled{b} c'$. ■

Definition 15: *Direct Subclasses* Let $c_1, c_2 \in C$ be any two classes, $c_1 \neq c_2$. Then class c_1 is a *direct subclass* of class c_2 (or c_2 is a *direct superclass* of c_1), denoted $c_1 \textcircled{k}_d c_2$, if $c_1 \textcircled{k} c_2$ and there is no class c such that $c_1 \textcircled{k} c$ and $c \textcircled{k} c_2$. ■

Definition 16: *Direct Demandclasses* Let $c_1, c_2 \in C$ be any two classes, $c_1 \neq c_2$. Then, class c_1 is a *direct demandclass* of class c_2 (or c_2 is a *direct supplyclass* of c_1), denoted $c_1 \textcircled{r}_d c_2$, if $c_1 \textcircled{r} c_2$ and there is no class c such that $c_1 \textcircled{r} c$ and $c \textcircled{r} c_2$. ■

The above definitions will be used in Chapter 4 to define complete structures of KBO objects and KBO classes. We also impose two constraints on direct instances and behaviors:

Axiom 1: *Unique Direct Instance Axiom.* No object is a direct instance of two different classes. That is, $\forall o \in \mathcal{O}, \forall c_1, c_2 \in \mathcal{C} : o \textcircled{i}_d c_1 \wedge o \textcircled{i}_d c_2 \Longrightarrow c_1 = c_2$. ■

That is, when a KBO object is created, it can be directly attached as an instance to only one class.

Axiom 2: *Unique Direct Behavior Axiom.* No behavior is a direct behavior of two different classes. That is, $\forall o \in \mathcal{O}, \forall c_1, c_2 \in \mathcal{C} : o \textcircled{b}_d c_1 \wedge o \textcircled{b}_d c_2 \Longrightarrow c_1 = c_2$. ■

That is, when a KBO object is created, it can be directly attached as a behavior to only one class.

These two constraints are introduced mainly for reducing the complexity of our class definitions in the next chapter.

3.4.3 Aggregations

For the following definition, we shall introduce five pseudo attribute labels:

$$\emptyset_{if},\ \emptyset_{then},\ \emptyset_{else},\ \emptyset_{member},\ \emptyset_{ordered_member}$$

They are used to indicate some fixed or collective relationships between one class and some other classes. In practice, these pseudo attribute labels are invisible, and introduced here only for the sake of technical uniformity.

Definition 17: *PART-OF (Aggregation).* Let $\mathcal{P}$ be the set of all *parts* defined as

$$\mathcal{P} = (\mathcal{A} \cup \{\emptyset_{if}, \emptyset_{then}, \emptyset_{else}, \emptyset_{member}, \emptyset_{ordered_member}\}) \times \mathcal{C}$$

and for each part $(\alpha, c) \in \mathcal{P}$, we say that c *plays the role* α in the part. Then, the PART-OF abstraction, written $\textcircled{p}$, is a binary relation over $\mathcal{P} \times \mathcal{C}$ such that

$$\forall o_1, o_2 \in \mathcal{O}: oid(o_1) \in orf(o_2) \Longrightarrow \exists(\alpha, c_1) \in \mathcal{P}, \exists c_2 \in \mathcal{C}: o_1 \textcircled{i} c_1 \wedge o_2 \textcircled{i} c_2 \wedge (\alpha, c_1) \textcircled{p} c_2$$

and

$$\forall(\alpha_1, c_1), (\alpha_2, c_2) \in \mathcal{P}, \forall c_3 \in \mathcal{C}: (\alpha_1, c_1) \circledP c_2 \wedge (\alpha_2, c_2) \circledP c_3 \Longrightarrow (\alpha_1, c_1) \circledP c_3$$

For each assertion $(\alpha, c_1) \circledP c_2$, we call (α, c_1) a *part* of c_2. ■

Intuitively, if an object has a sub-object as a component (i.e., the object can reach the sub-object), then there must exist a PART-OF relation between a class of the sub-object and a class of the object. Furthermore, the PART-OF relation is transitive. The intended intuitive interpretation of PART-OF in the KBO model is that a class can be constructed from other classes, provided each of those classes is given an attribute label (or a pseudo attribute label) to indicate its role in the construction. The intended semantics of those pseudo attribute labels will be further explained in the next two sections.

The following example demonstrates the transitivity of the $\circledP$ relation.

Example 6: Let $Name, FirstName$ be two attribute labels in $\mathcal{A}$, and let EMPLOYEES, EMPLOYEE, FULL-NAME, TEXT be four classes in $\mathcal{C}$. Then, if we have the following PART-OF relationships:

$(\emptyset_{member}$,EMPLOYEE) $\circledP$ EMPLOYEES
$(Name$,FULL-NAME) $\circledP$ EMPLOYEE
$(FirstName$,TEXT) $\circledP$ FULL-NAME

we can also derive the following PART-OF relationships:

$(Name$,FULL-NAME) $\circledP$ EMPLOYEES
$(FirstName$,TEXT) $\circledP$ EMPLOYEES
$(FirstName$,TEXT) $\circledP$ EMPLOYEE

■

Sometimes we are only interested in the "highest-level" components of a complex relationship. Such components can be called "direct parts" of a class, defined as follows:

Definition 18: *Direct Parts.* A pair $(\alpha, c_1) \in \mathcal{P}$ is a *direct part* of a class c_2, denoted $(\alpha, c_1) \circledP_d c_2$, if $(\alpha, c_1) \circledP c_2$ and there is no pair $(\alpha', c) \in \mathcal{P}$ such that $(\alpha, c_1) \circledP c$ and $(\alpha', c) \circledP c_2$. For each assertion $(\alpha, c_1) \circledP_d c_2$, we call (α, c_1)

(1) a *attribute part* of c_2 if $\alpha \in \mathcal{A}$;

(2) a *conditional part* of c_2 if $\alpha \in \{\emptyset_{if}, \emptyset_{then}, \emptyset_{else}\}$;

(3) a *member part* of c_2 if $\alpha = \emptyset_{member}$;

(4) an *ordered member part* of c_2 if $\alpha = \emptyset_{ordered_member}$.

In other words, direct parts of a class represent the highest-level components that constitute the class, which represents a complex relationship.

Axiom 3: *Unique Direct Part Axiom.* No class has two different direct parts that have the same role. That is,

$$\forall(\alpha_1, c_1), (\alpha_2, c_2) \in \mathcal{P}, \forall c \in \mathcal{C}: (\alpha_1, c_1) \circledP_d \, c \wedge (\alpha_2, c_2) \circledP_d \, c \Longrightarrow \alpha_1 \neq \alpha_2$$

This constraint is needed to identify different roles of different direct parts in a class. In other words, two different attributes in a class cannot have the same attribute labels; otherwise, we cannot distinguish the roles they play in the relationship.

Definition 19: *Capsule, Aggregate, Bi-object, Set and Sequence Classes.*

1. A class is a *capsule class* if it does not have a part.

2. A class is an *aggregate class* if it has some parts and all of its direct parts are attribute parts.

3. A class is a *bi-object class* if it has some parts and its direct parts are three conditional parts (if-part, then-part, else-part).

4. A class is a *set class* if it has some parts and its direct part is a member part.

5. A class is a *sequence class* if it has some parts and its direct part is an ordered member part. ■

We shall let $\mathcal{C}_{capsule}$ be the set of all capsule classes, $\mathcal{C}_{agg}$ the set of all aggregate classes, $\mathcal{C}_{bio}$ the set of all bi-object classes, $\mathcal{C}_{set}$ the set of all set classes, and $\mathcal{C}_{seq}$ the set of all sequence classes.

Definition 20: *Class Domains.* For any $c \in \mathcal{C}$, its domain, denoted $dom(c)$, is defined by:

1. if c is a capsule class, then
   $dom(c) = \{v \mid \exists D_i \in \mathcal{C}_{base}: v \in dom(D_i)\}$;

2. if c is an aggregate class and $(A_i, c_i) \circledP_d \, c$ for $1 \leq i \leq n$, then
   $dom(c) = \{[A_1: id_1, \ldots, A_n: id_n] \mid \forall i \in [1..n] : id_i \in \mathcal{I} \wedge oid^{-1}(id_i) \circledI \, c_i\}$;

3. if c is a bi-object class and $(\emptyset_{if}, c_1)\ \textcircled{p}_d\ c$, $(\emptyset_{then}, c_2)\ \textcircled{p}_d\ c$ and $(\emptyset_{else}, c_3)\ \textcircled{p}_d\ c$, then
   $dom(c) = \{\{\!\!\{\ id_1, id_2, id_3\ \}\!\!\} \mid \forall i \in [1..3] : id_i \in \mathcal{I} \wedge oid^{-1}(id_i)\ \textcircled{i}\ c_i\}$;

4. if c is a set class and $(\emptyset_{member}, c')\ \textcircled{p}_d\ c$, then
   $dom(c) = \{\ \{id_1, \ldots, id_n\} \mid \forall i \in [1..n] : id_i \in \mathcal{I} \wedge oid^{-1}(id_i)\ \textcircled{i}\ c'\}$; and

5. if c is a sequence class and $(\emptyset_{ordered_member}, c')\ \textcircled{p}_d\ c$, then
   $dom(c) = \{\ <id_1, \ldots, id_n> \mid \forall i \in [1..n] : id_i \in \mathcal{I} \wedge oid^{-1}(id_i)\ \textcircled{i}\ c'\}$. ■

This definition is needed to formally define object values in Chapter 4.

3.4.4 Aggregations in Structural Modeling

The four aggregation constructs (aggregate classes, bi-object classes, set classes and sequence classes) are four mechanisms for constructing the representation of a class from other classes.

Our aggregate classes and set classes are similar to the tuple structured types and set structured types in the O_2 object-oriented data model [LRV88]. It should come as no surprise that these two aggregation constructs are supported in our model. In Figure 3.7, classes PROJECT, PROGRAMMER and CONTRACTOR are examples of aggregate classes, which are each represented as a vertex with arcs down to other vertices representing the direct parts; arcs are labeled with attribute labels. According to Figure 3.7, we have the following assertions for the three aggregate classes:

$(Name, \text{TEXT})\ \textcircled{p}_d\ \text{PROJECT}$
$(Leader, \text{PROGRAMMER})\ \textcircled{p}_d\ \text{PROJECT}$
$(Budget, \text{DOLLARS})\ \textcircled{p}_d\ \text{PROJECT}$
$(Specification, \text{REPORT})\ \textcircled{p}_d\ \text{PROJECT}$

$(Name, \text{TEXT})\ \textcircled{p}_d\ \text{PROGRAMMER}$
$(Address, \text{ADDRESS})\ \textcircled{p}_d\ \text{PROGRAMMER}$
$(Salary, \text{DOLLARS})\ \textcircled{p}_d\ \text{PROGRAMMER}$
$(Kids, \text{CHILDREN})\ \textcircled{p}_d\ \text{PROGRAMMER}$

$(Name, \text{TEXT})\ \textcircled{p}_d\ \text{CONTRACTOR}$
$(Skills, \text{SKILLS})\ \textcircled{p}_d\ \text{CONTRACTOR}$
$(BillingRate, \text{DOLLARS})\ \textcircled{p}_d\ \text{CONTRACTOR}$

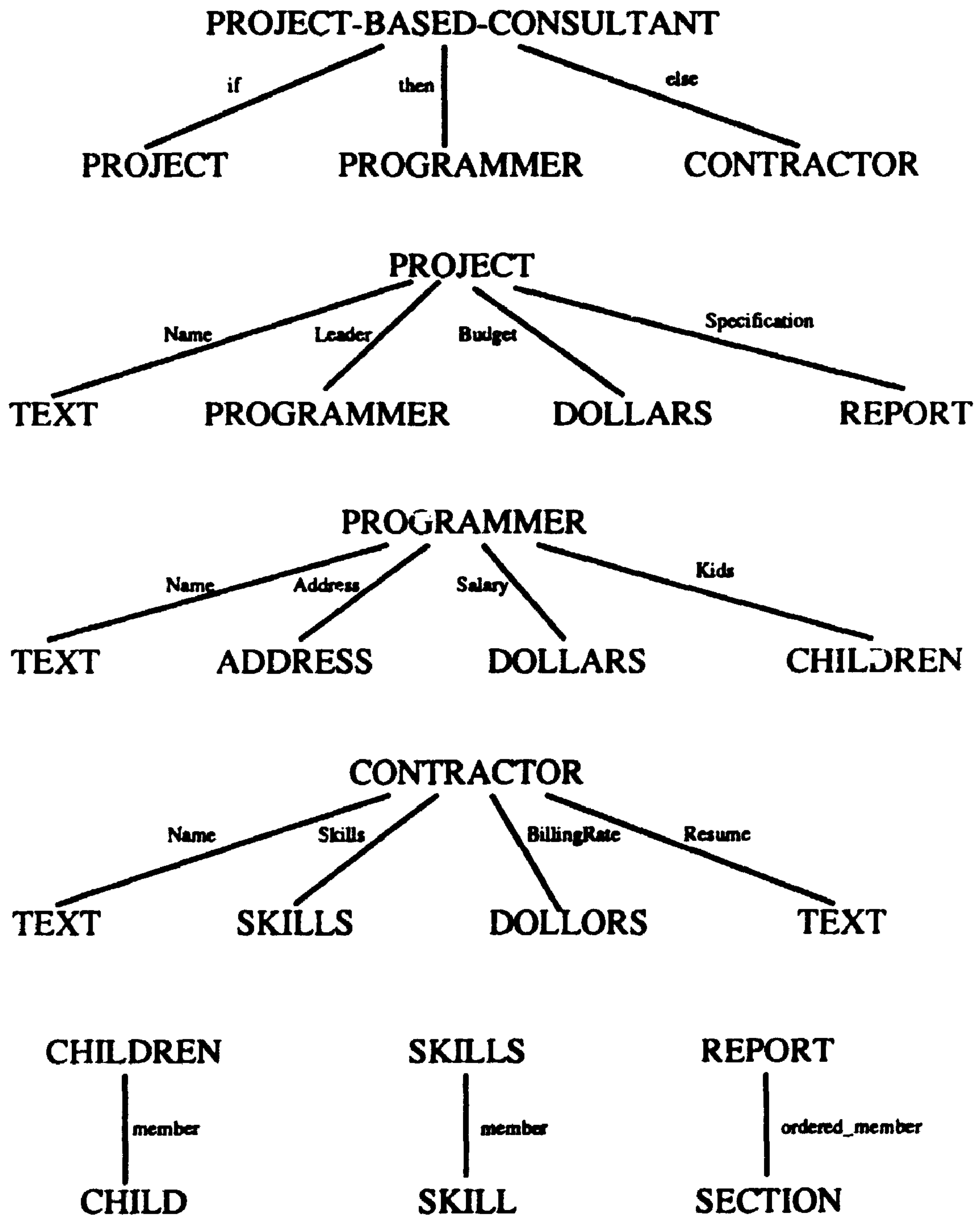


Figure 3.7: All the parts of class PROJECT-BASED-CONSULTANT

$$(\mathit{Resume}, \text{TEXT}) \textcircled{P}_d \text{ CONTRACTOR}$$

Aggregate classes are usually the most common classes in OODB applications. Also shown in Figure 3.7 are classes CHILDREN and SKILLS, which are examples of set classes: each member in a CHILDREN object is a CHILD object, and each member in a SKILLS object is a SKILL object. According to Figure 3.7, we can have the following assertions for classes CHILDREN and SKILLS:

$$(\emptyset_{member}, \text{CHILD}) \textcircled{P}_d \text{ CHILDREN}$$
$$(\emptyset_{member}, \text{SKILL}) \textcircled{P}_d \text{ SKILLS}$$

A sequence class is similar to a set class except that there is a total ordering among its member parts. For example, the graphical representation for REPORT in Figure 3.7 captures the semantics that a REPORT object contains a sequence of SECTION objects. Accordingly, we have the following assertion for REPORT:

$$(\emptyset_{ordered_member}, \text{SECTION}) \textcircled{P}_d \text{ REPORT}$$

The notion of a bi-object class is a new concept in the OODB field. Our primary motive for introducing bi-object classes is to facilitate behavioral modeling in the form of structural modeling. However, for structural modeling, this construct can capture the perception that in the real world there exist entities that are *circumstance-sensitive*. For example, a sensitive plant is normally a unfolded plant but becomes a folded plant when touched, and an independent consultant becomes a programmer when offered a project — he would become a contractor only when the project is finished. In OODBs, these circumstance-sensitive characteristics are traditionally buried in the code of the methods of the classes representing these entities (such as PLANT or CONSULTANT). As a result, there is no easy and explicit way for end-users to ask queries about these important characteristics. In our model, certain simple circumstance-sensitive entities can be represented explicitly as bi-object classes. Consider the following parts of class SENSITIVE-PLANT:

$$(\emptyset_{if}, \text{TOUCH}) \textcircled{P}_d \text{ SENSITIVE-PLANT}$$
$$(\emptyset_{then}, \text{FOLDED-PLANT}) \textcircled{P}_d \text{ SENSITIVE-PLANT}$$
$$(\emptyset_{else}, \text{UNFOLDED-PLANT}) \textcircled{P}_d \text{ SENSITIVE-PLANT}$$

Intuitively, they represent the characteristic that when a SENSITIVE-PLANT ob-

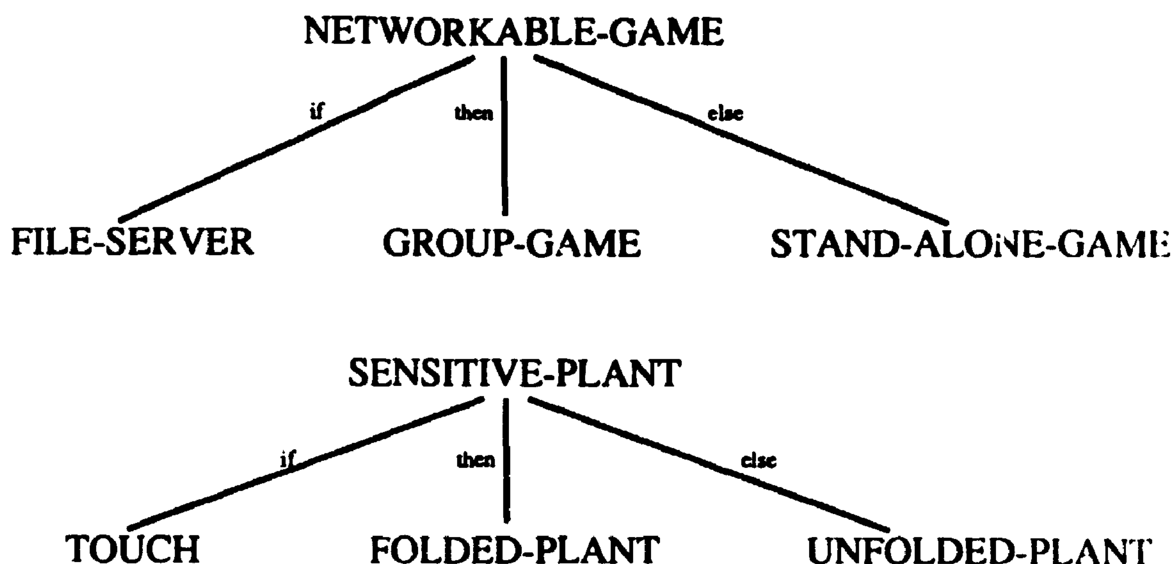


Figure 3.8: Examples of Bi-object Classes

ject is given a TOUCH object then the SENSITIVE-PLANT object behaves like a FOLDED-PLANT object; otherwise it behaves like an UNFOLDED-PLANT object. As such, a SENSITIVE-PLANT object can be viewed as containing a simple "trigger": if given a non-null touch, it switches to a folded plant; if given a null touch, it switches to an unfolded plant. As another example, consider the class PROJECT-BASED-CONSULTANT shown in Figure 3.7. This class is constructed from three other classes: PROJECT, PROGRAMMER and CONTRACTOR. The relationships appropriate to this representation are the following:

$(\emptyset_{if}, \text{PROJECT}) \; \textcircled{p}_d \; \text{PROJECT-BASED-CONSULTANT}$
$(\emptyset_{then}, \text{PROGRAMMER}) \; \textcircled{p}_d \; \text{PROJECT-BASED-CONSULTANT}$
$(\emptyset_{else}, \text{CONTRACTOR}) \; \textcircled{p}_d \; \text{PROJECT-BASED-CONSULTANT}$

Intuitively, they state that when a PROJECT-BASED-CONSULTANT is given a PROJECT then the consultant only behaves like a PROGRAMMER; otherwise the consultant behaves like a CONTRACTOR (seeking a project). In other words, a bi-object has two "spokesmen" that may take turns to answer messages sent to the bi-object, depending on whether there exists a circumstance object (such as a PROJECT object in a PROJECT-BASED-CONSULTANT object). Figure 3.8 shows some other examples of bi-object classes. The first representation captures the intuition that if a NETWORKABLE-GAME can find a FILE-SERVER, it is a GROUP-GAME; other-

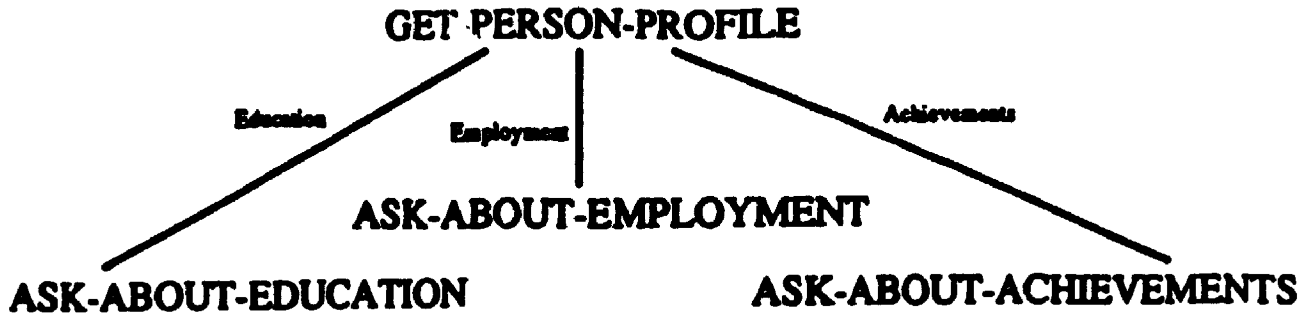


Figure 3.9: Example of an AGG Class in Behavioral Modeling

wise it is a STAND-ALONE-GAME. The second representation is for the above class SENSITIVE-PLANT.

3.4.5 Aggregations in Behavioral Modeling

For behavioral modeling, the four aggregation constructs represent four different mechanisms for sending a group of invokers (messages) to a receiver. Because these mechanisms are essentially structural mechanisms used for behavioral modeling, it becomes possible to have complex objects which are "complex invokers" when used in the invoker side in a message-sending. In the following, we describe the intended intuitive interpretations for these four constructs in behavioral modeling.

For an aggregate class with n attribute parts, an instance, when used as an invoker sent to a receiver, represents n sub-invokers that are sent at the same time (they thus could be on multiple processors), to n receivers that have the same internal state as the original receiver; the result is an instance of another aggregate class with the same n attribute labels. For example, the message-sending

$$[A_1\colon o_1, A_2\colon o_2] \rightsquigarrow o_{receiver}$$

will become

$$[A_1\colon o_1 \rightsquigarrow o_{receiver_1}, A_2\colon o_2 \rightsquigarrow o_{receiver_2}]$$

where $o_{receiver_1}$ and $o_{receiver_2}$ are copies (with new identities) of $o_{receiver}$. Semantically, this mechanism represents the *parallel* invocation of n knowhows by n sub-invokers. Note that in the above example if we simply let $o_{receiver_1}$ and $o_{receiver_2}$ be $o_{receiver}$ itself (rather than copies) then it would be impossible to have parallel invokers, and it might also produce different results when o_1 and o_2 were sent to $o_{receiver}$ in different orders. The class GET-PERSON-PROFILE in Figure 3.9 is an example of aggregate classes that can be used actively. Intuitively, the representation of this class is intended to capture a behavioral semantics: a GET-

PERSON-PROFILE invoker (instance) has three parallel sub-invokers which are respectively instances of ASK-ABOUT-EDUCATION, ASK-ABOUT-EMPLOYMENT and ASK-ABOUT-ACHIEVEMENTS. Suppose classes ASK-ABOUT-EDUCATION, ASK-ABOUT-EMPLOYMENT and ASK-ABOUT-ACHIEVEMENTS are subclasses of TEXT (recall from Definition 7 that the knowhow of an object is not the value of an object). The following is an example instance of GET-PERSON-PROFILE:

[Education: "List all your university degrees",
Employment: "List companies where you worked as a C++ programmer",
Achievements: "List all the papers you published"]

If we send this aggregate object to a receiver (say, a computer scientist), then the result object could look like this:

[Education: { "B.Sc.", "M.Sc.", "MBA", "Ph.D." },
Employment: { "AT&T", "Ontologic", "ServioLogic", "ObjectDesign" },
Achievements: $\{Paper_1, Paper_2, Paper_3\}$]

where $Paper_1, Paper_2, Paper_3$ denote three complex objects.

For a bi-object class, an instance, when used as an invoker sent to a receiver, represents a choice from two candidate sub-invokers (the then-part and the else-part): one of them will be chosen based on the result of sending a third sub-invoker (the if-part) to the receiver. For example, the message-sending

$$\{ o_{if}, o_{then}, o_{else} \} \rightsquigarrow o_{receiver}$$

will become

$$o_{then} \rightsquigarrow o_{receiver}$$

if $o_{if} \rightsquigarrow o_{receiver}$ returns a non-failure/non-false object; otherwise, the original message-sending becomes

$$o_{else} \rightsquigarrow o_{receiver}$$

Semantically, this mechanism represents the *conditional* invocation of two alternative knowhows by the sub-invoker which is the then-part or by the sub-invoker which is the else-part. The class TAKE-CARE shown in Figure 3.10 is an example of bi-object classes that can be used for behavioral modeling. Intuitively, the representation of this class can be interpreted as follows: a TAKE-CARE invoker has two alternative sub-invokers which are respectively instances of ADVICE and NORMAL-ACTIVITY, as

Figure 3.10: Example of a BIO Class in Behavioral Modeling

well as a "choice-making" sub-invoker which is an instance of DETECT-SYMPTOM. Let DETECT-SYMPTOM, ADVICE and NORMAL-ACTIVITY be subclasses of TEXT. Then the following is an example instance of TAKE-CARE:

$$\{ \text{"Are you sick?"}, \text{"Go see doctor"}, \text{"Go to work"} \}$$

where "Are you sick?" is an instance of DETECT-SYMPTOM, "Go see doctor" an instance of ADVICE, and "Go to work" an instance of NORMAL-ACTIVITY. If we send this bi-object to a receiver denoted by *Peter* and assume that *Peter*'s answer to "Are you sick?" will be **True**, then the result object from this message-sending should be the *Peter* who had gone to see a doctor (a side-effect caused by the message-sending).

For a set class, an instance with n members, when used as an invoker sent to a receiver, represents n message-sendings occurring one after another towards the same receiver, but in arbitrary order. For example, the message-sending

$$\{o_1, o_2\} \leadsto o_{receiver}$$

will either become

$$o_1 \leadsto o_{receiver} \text{ followed by } o_2 \leadsto o_{receiver}$$

or become

$$o_2 \leadsto o_{receiver} \text{ followed by } o_1 \leadsto o_{receiver}$$

Semantically, this mechanism represents an invocation of n knowhows by n sub-invokers *in any order* (or concurrently). Practically speaking, this mechanism is most useful to cause several independent (i.e., non-interacting) side-effects on the internal state of an object. The class EAT-LUNCH shown in Figure 3.11 is an example of a set class whose instances can be used meaningfully as invokers. Intuitively, the representation of EAT-LUNCH is intended to capture this behavioral semantics: an EAT-LUNCH invoker consists of a set of sub-invokers that are instances of

Figure 3.11: Example of a SET Class in Behavioral Modeling

EATING-BEHAVIORS; we don't care in what order these sub-invokers constitute the EAT-LUNCH invoker. Again, let EATING-BEHAVIOR be a subclass of TEXT. Then the following is an example instance of EAT-LUNCH:

{ "Eat sandwich", "Drink coke", "Read Toronto Star" }

where "Eat sandwich", "Drink coke", and "Read Toronto Star" are instances of class EATING-BEHAVIOR. If we send this set object to a receiver denoted by *Peter*, then the result object from this message-sending should be the *Peter* who just had his lunch by eating a sandwich, drinking coke and reading the Toronto Star, in an arbitrary order. As we can see, the concept of "set objects as invokers" enable end-users to exploit indeterminism.

For a sequence class, an instance with n ordered members, when used as an invoker sent to a receiver, represents n message-sendings occurring in succession, each with a new receiver (except the first one which has the original receiver) that is the result of the previous message-sending. For instance,

$$< o_1, o_2 > \rightsquigarrow o_{receiver}$$

will become

$$o_2 \rightsquigarrow (o_1 \rightsquigarrow o_{receiver})$$

Semantically, this mechanism represents a ***sequential*** invocation of n knowhows by n sub-invokers. The class COOK-DINNER shown in Figure 3.12 is an example of a sequence class whose instances can be used meaningfully as invokers. Intuitively, the representation of COOK-DINNER is intended to capture this behavioral semantics: a COOK-DINNER invoker consists of a sequence of sub-invokers that are instances of COOKING-STEP. Assume that "Turn stove on", "Add macaroni to boiled water" "Stir for 10 minutes", "Add Cheese sauce" and "Turn stove off" are five instances of class COOKING-STEP, and each of them has SELF as the formal output specification of their knowhows (i.e., they may cause side-effects). Then the following is an

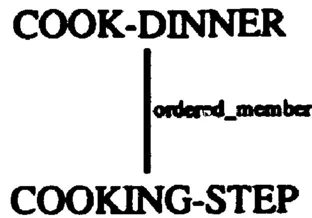


Figure 3.12: Example of a SEQ Class in Behavioral Modeling

example instance of COOK-DINNER:

< “Turn stove on”, “Add macaroni to boiled water” “Stir for 10 minutes”,
“Add cheese sauce” “Turn stove off” >

If we send this sequence object to a receiver denoted by *Peter*, then the result object from this message-sending should be the *Peter* who just cooked his dinner step by step as follows:

1. Turning a stove on:

   “Turn stove on” $\leadsto$ *Peter* $\Rightarrow$ *Peter*

2. Adding macaroni to boiled water in a pot:

   “Add macaroni to boiled water” $\leadsto$ *Peter* $\Rightarrow$ *Peter*

3. Stirring the macaroni for 10 minutes:

   “Stir for 10 minutes” $\leadsto$ *Peter* $\Rightarrow$ *Peter*

4. Adding Cheese sauce into the pot:

   “Add cheese sauce” $\leadsto$ *Peter* $\Rightarrow$ *Peter*

5. Turning the stove off:

   $“Turnstoveoff” \leadsto Peter \Rightarrow Peter$

3.5 Kernel Classes

As we have seen, relations ⓘ and ⓡ are partial orderings on C. Each of them can further become a complete lattice over classes, when the greatest class OBJECT and the smallest class BOTTOM are introduced.

Definition 21: *Kernel Classes OBJECT and BOTTOM.* There exist OBJECT and BOTTOM in $\mathcal{C}$ such that

(1) $\forall o \in \mathcal{O}, \forall c \in \mathcal{C}: o \,ⓘ\, \text{OBJECT} \wedge (o \,ⓑ\, \text{OBJECT} \Longrightarrow o \,ⓑ\, c)$

(2) $\forall o \in \mathcal{O}, \forall c \in \mathcal{C}: o \,ⓑ\, \text{BOTTOM} \wedge (o \,ⓘ\, \text{BOTTOM} \Longrightarrow o \,ⓘ\, c)$

(3) $\forall o \in \mathcal{O}: o \,ⓘ\, \text{BOTTOM} \Longrightarrow o = \textbf{\textit{null}} \vee o = \textbf{\textit{same}} \vee o = \textbf{\textit{fail}}$

■

Notice that BOTTOM has three instances ***null***, ***same*** and ***fail***, which are called *bottom objects*. The bottom object ***fail*** is mainly used to formalize run-time errors. The bottom object ***null*** is intended to represent any object whose value is unknown (but there is a value). The bottom object ***same*** is intended to return a receiver itself which receives ***same*** as an invoker. The three bottom objects have special semantics in message-sendings, which will be formally defined in Chapter 6. It should be observed that, from the above definition, bottom objects are also instances of any other classes.

The above definition implies that conventional data objects (whose knowhows are $\phi_\perp$) are also treated as behaviors of BOTTOM. There are two main reasons why we choose to do this. First, this provides theoretical elegance because this makes it possible to have a lattice under the REUSE-OF relation. Second, it is possible for regular data objects to gain a non-empty knowhow during their life time, thus becoming a behavior of some class other than BOTTOM. For example, a data object (integer) 911 can be a behavior of BOTTOM first, and may later become a behavior of class PHONE by being given a non-empty knowhow. As another example, let object *Iris* be an instance of class PROJECT with value

$$[ProName: \text{"}Iris\text{"}, Budget: 500000]$$

When object *Iris* was first created it was given an empty knowhow $\phi_\perp$, so it was treated as a behavior of BOTTOM. Suppose now object *Iris* is made a behavior of PROGRAMMER and is given a knowhow that knows how to report the progress on the Iris project for PROGRAMMER instances (programmers). More precisely, if *Peter* is a programmer, then message-sending

$$Iris \rightsquigarrow Peter$$

should cause *Peter* to return a printout of his progress report on the project *Iris*. This example shows that a regular data object like *Iris* could be promoted from a behavior of class BOTTOM to a behavior of class other than BOTTOM. In this example, if many instances of PROJECT (like *Iris*) are made behaviors of PROGRAMMER,

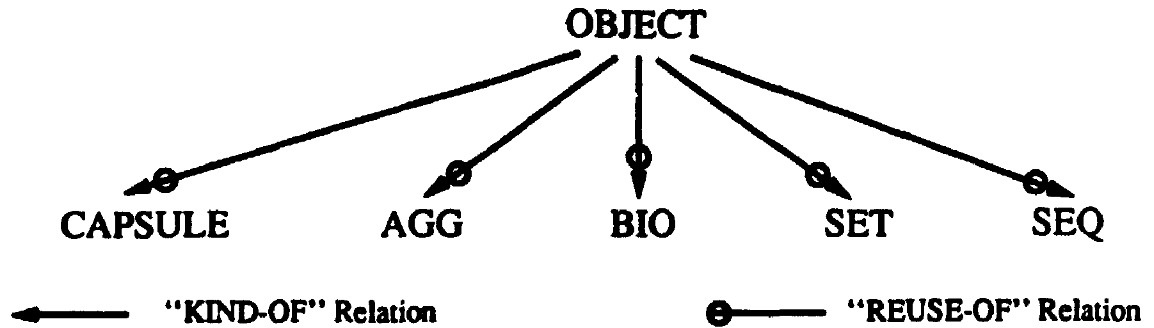


Figure 3.13: The KIND-OF and REUSE-OF Relationships among Kernel Classes

then it is possible for the end-user to ask (associatively) queries like "All programmers, please report your progress on the projects that have a budget greater than $100,000." In other words, we could select, from a set of PROJECT objects which are also behaviors of PROGRAMMER, the projects whose Budget attribute is greater than 100000, and send them directly as invokers to a set of programmers (such a query can be easily formulated in our Apply operation defined in Chapter 8). We should point out that, due to the increasing popularity of pulldown-menu driven applications, there is no reason why messages cannot be more descriptive (e.g., an aggregate, a set, a sequence) than a short name. For the sake of simplicity, most behaviors in our examples are TEXT objects bearing a non-empty knowhow.

In order to explicitly distinguish the four kinds of complex object structures in the KBO model, we also introduce the following kernel classes:

Definition 22: *Kernel Classes CAPSULE, AGG, BIO, SET and SEQ.* There exist classes CAPSULE, AGG, BIO, SET and SEQ in $\mathcal{C}$, such that:

(1) $\forall c \in \mathcal{C}_{capsule} : c \textcircled{k} \mathrm{CAPSULE} \wedge c \textcircled{r} \mathrm{CAPSULE}$

(2) $\forall c \in \mathcal{C}_{agg} : c \textcircled{k} \mathrm{AGG} \wedge c \textcircled{r} \mathrm{AGG}$

(3) $\forall c \in \mathcal{C}_{bio} : c \textcircled{k} \mathrm{BIO} \wedge c \textcircled{r} \mathrm{BIO}$

(4) $\forall c \in \mathcal{C}_{set} : c \textcircled{k} \mathrm{SET} \wedge c \textcircled{r} \mathrm{SET}$

(5) $\forall c \in \mathcal{C}_{seq} : c \textcircled{k} \mathrm{SEQ} \wedge c \textcircled{r} \mathrm{SEQ}$

■

The $\textcircled{k}$ and $\textcircled{r}$ relationships among the kernel classes are shown in Figure 3.13 (BOTTOM is omitted). It must be noticed that these kernel classes have no real-world modeling purpose, and are introduced mainly to facilitate the formalization of the other features of the KBO model.

CHAPTER 4

KBO OBJECTS AND CLASSES

4.1 Introduction

In this chapter, we shall further extend the definition of KBO objects. In Chapter 3, a KBO object o is defined as a triple

$$o = (id, v, k)$$

where

- $id = oid(o)$ is the object-identity of o.
- v is the value of o.
- $k = kh(o)$ is the knowhow of o.

Here two problems are apparent: First, the definition does not take into account the facts that each KBO object o is a direct instance of a KBO class (which we call "the class of o") and that each KBO object o is also a direct behavior of a KBO class (which we call "the owner of o"). Notice that KBO classes should correspond to *real-world* (static or dynamic) concepts. For example, a capsule object with value *"Are you married?"* may be created both as a direct instance of class PERSONAL-QUESTION and as a direct behavior of class PERSON. As another example, an aggregate object with value

$$[Do\text{: "Go jogging"}, MinWeighLoss\text{: } 10, MaxWeighLoss\text{: } 30]$$

may be created both as a direct instance of class EXERCISE and as a direct behavior of class ATHLETE. Secondly, how the knowhow (i.e., k) of an object o can be properly executed is not apparent from the above definition. One might say that the formal input/output specification of the knowhow k should tell us how to use it. This is true for an object o which is also a "biological bearer" of its knowhow, that is, $bb(kh(o)) = o$. For example, if an object x_1 with value *"Take a course"* is a biological bearer of its knowhow $kh(x_1)$ with $fis(kh(x_1)) = \{(Course, \text{COURSE})\}$ and $fos(kh(x_1)) = \text{SELF}$, then we can say that object x_1 has a signature

$$(Course, \text{COURSE}) \longrightarrow \text{SELF}$$

which tells us that its knowhow takes an instance of COURSE as a labeled input argument and produces the receiver as the result. Suppose x_1 is a direct behavior of STUDENT. Now, a direct behavior x_2 of MSC-STUDENT (with the same value *"Take a course"*) may want to borrow the knowhow from x_1 (i.e., $kh(x_2) = kh(x_1)$) but with a new signature

$$(Course, \text{500UP-COURSE}) \longrightarrow \text{SELF}$$

where class 500UP-COURSE is a subclass of class COURSE. Two observations should be made here: (1) as the knowhow of x_2 (which is not a biological bearer), $kh(x_1)$ can still be successfully executed with the new signature, because if an argument is an instance of 500UP-COURSE it is also an instance of COURSE — no typing violation (here we assume that class 500UP-COURSE is a subclass of class COURSE); (2) from the data modeling point of view, the new signature

$$(Course, \text{500UP-COURSE}) \longrightarrow \text{SELF}$$

for behavior x_2 serves as a semantic constraint which disallows a masters student to register in a course with a level less than 500. In practice, this feature is very useful since we can reuse existing executable code with the possibility to impose new semantic constraints. The above example has shown that we cannot decide how an object's knowhow is intended to be used by only looking at the formal input/output specification of that knowhow — the object may have its own input/output specification which is different from that of its knowhow.

The above problems will be dealt with in the following section. We will also define the graphical representation and equalities for KBO objects, and extend the naive message-sending paradigm, which is described in Chapter 3, to a "less naive" message-sending paradigm. A formal definition of KBO classes is then given, with a detailed discussion of the notions of local representation, local interface, full representation, full interface, and value-based interface. Also, methods are described for resolving conflicts among the values of a KBO class's behaviors.

4.2 KBO Objects

4.2.1 Object Definition

The following definition for KBO objects takes into account the direct INSTANCE-OF and direct BEHAVIOR-OF relationships, as well as the specification of how the knowhow of a KBO object can be used in the form of the signature of the object. In

particular, the definition introduces a notion called "vigor" to express the dynamic part of the object, which describes (1) which class owns this object as a direct behavior, i.e., this object represents a behavior directly implemented within the owner class; (2) what signature this object has, i.e., how this object can be used as invocable behavior; and (3) what knowhow this object has, i.e., this object may be a biological bearer bearing a newly implemented knowhow or it may be a non-biological bearer bearing a knowhow borrowed from another object.

Definition 23: *Objects Revisited.* A KBO object $o \in \mathcal{O}$ is a quadruple

$$o = (id, c, v, g)$$

where

- id is the object-identity of o such that $id = oid(o)$.
- c is the class of o such that o Ⓘ$_d$ c.
- v is the value of o such that $v \in dom(c)$.
- $g = (c_{owner}, \xi, k)$ is the *vigor* (the dynamic part) of o such that:
  1. $c_{owner} \in \mathcal{C}$ is the *owner* of o such that o Ⓑ$_d$ c_{owner}. If c_{owner} = BOTTOM, then o is a *passive object* (i.e., it does not have a real "behavioral side"), and if $c_{owner} \neq$ BOTTOM, then o is an *active object.*
  2. ξ is the *signature* of o written

$$(A_1, c_1) \times \ldots \times (A_n, c_n) \longrightarrow c_{result}$$

where $A_i \in \mathcal{A}$, $c_{result} \in \mathcal{C} \cup \{\text{SELF}\}$ and $c_i \in \mathcal{C}$ for $1 \leq i \leq n$, such that if $bb(kh(o)) = o$ (i.e., o is a biological bearer of its knowhow), then $\{(A_1, c_1), \ldots, (A_n, c_n)\} = is(kh(o))$ and $c_{result} = fos(kh(o))$. We call $(A_1, c_1) \times \ldots \times (A_n, c_n)$ the formal input specification of the object o, and c_{result} the formal output specification of the object o. In particular, ξ is written

$$() \longrightarrow c_{result}$$

when $n = 0$ (i.e., the knowhow of o does not take any labeled input arguments). When c_{owner} = BOTTOM, $\xi = () \longrightarrow$ BOTTOM (i.e., all passive objects have the same signature: $() \longrightarrow$ BOTTOM).

3. $k = kh(o)$ is the knowhow of o, such that if c_{owner} = BOTTOM then $k = \phi_{\perp}$. If $bb(k) = o$, then k is an ***inherent knowhow***; otherwise, k is a ***borrowed knowhow***. Technically, we shall denote an inherent knowhow as kh_j where j is an integer, and denote a borrowed knowhow as $kh_{o'}$ where o' is the object whose knowhow is borrowed by o. Thus if the knowhow of o is denoted as $kh_{o'}$ then $kh(o) = kh(o')$.

■

Two observations must be made here: First, a *knowhow itself is not an object — it is just another intrinsic part of a KBO object*, much like what a class or a value is to a KBO object. Second, the formal input specification of a KBO object may not be the same as the formal input specification of its knowhow; similarly, the formal output specification of a KBO object may not be the same as the formal output specification of its knowhow. However, there are some restrictions for an object to have a signature different from that of its knowhow; these restrictions are discussed in Chapter 5.

The above definition can be illustrated by the following examples.

Example 7: This example illustrates the difference between passive objects and active objects. Assume classes PHONE and EMERGENCY-CENTRE are existing classes in an OODB. In the following, o_1 and o_2 are two typical KBO objects. In particular, o_1 is a passive object because it does not have a dynamic part. In contrast, o_2 is an active object because it bears a non-empty knowhow and is created as a direct behavior of class PHONE.

$$o_1 = (id_1, \text{INT}, 911, (\text{BOTTOM}, () \longrightarrow \text{BOTTOM}, \phi_{\perp})\ )$$
$$o_2 = (id_2, \text{INT}, 911, (\text{PHONE}, () \longrightarrow \text{EMERGENCY-CENTRE}, kh_2)\ )$$

As shown above, both o_1 and o_2 are both a direct instance of class INT. But o_1 is not intended to be a meaningful behavior (i.e., no real knowhow is assigned to it), so it automatically becomes a behavior of BOTTOM — a passive object. In other words, o_1 is intended to be just a conventional data object. On the other hand, o_2 is a direct behavior of class PHONE and is given a meaningful knowhow kh_2 that knows how to find a emergency centre for a phone object. As an instance of INT, o_2 can be manipulated by all the behaviors of INT (such as comparing o_2 with another integer 800). As a behavior of PHONE, o_2 can be used as an invoker in message-sendings like

$$o_2 \leadsto MyPhone$$

return the closest emergency centre in your neighborhood (where *MyPhone* denotes an instance of PHONE). Note that, according to our naive message-sendi..g paradigm, the above message-sending pinpoints the knowhow $kh(o_2)$ for execution. ■

Example 8: This example illustrates the difference between inherent knowhows and borrowed knowhows. In particular, the example illustrates how an object containing a borrowed knowhow may have a signature that is different from the signature of the object's knowhow. Consider the two classes MSC-STUDENT and PHD-STUDENT shown in Figure 3.5, where PHD-STUDENT is declared as a reuse of MSC-STUDENT. In the following, o_3 is a behavior of MSC-STUDENT and o_4 is a behavior of PHD-STUDENT.

$$o_3 = (id_3, \text{TEXT}, \text{“\textit{Take this course}”}, \\ (\text{MSC-STUDENT}, (\text{Course}, \text{COURSE}) \longrightarrow \text{SELF}, kh_3)\ )$$
$$o_4 = (id_4, \text{TEXT}, \text{“\textit{Take required course}”}, \\ (\text{PHD-STUDENT}, (\text{Course}, \text{700-LEVEL-COURSE}) \longrightarrow \text{SELF}, kh_{o_3})\ )$$

As shown above, kh_3 is a inherent knowhow of o_3, or o_3 is the "biological bearer" of kh_3. That is, $bb(kh_3) = o_3$. Thus, according to our definition, $fis(kh_3) = \{(\text{Course,COURSE})\}$ and $fos(kh_3) = \text{SELF}$. Based on the definition of the REUSE-OF relation, because PHD-STUDENT is a reuse of MSC-STUDENT, either PHD-STUDENT automatically gets o_3 as its behavior or PHD-STUDENT can exclude o_3 by defining a new behavior that only reuses kh_3 (not o_3) from MSC-STUDENT. In this particular example, PHD-STUDENT chooses the latter. As shown in the definition of o_4, knowhow kh_3 becomes a borrowed knowhow of o_4. However, notice that o_4 has a different signature

$$(\text{Course,700-LEVEL-COURSE}) \longrightarrow \text{SELF}$$

for kh_3, whose original signature is

$$(\text{Course,COURSE}) \longrightarrow \text{SELF}$$

The purpose is to impose a semantic constraint that when a PhD student takes a course, that course must be a 700-level course. Since class 700-LEVEL-COURSE is (presumably) a kind of COURSE, the knowhow kh_3 will have no trouble taking an instance of 700-LEVEL-COURSE as the argument, because that instance is also an instance of COURSE. ■

The definition for KBO objects also implies that all objects, whether active or passive, are classified by their value structures, regardless of what vigors they have. For example, in the first example above, the passive object o_1 and the active object o_2 are both instances of INT. We could create a subclass of INT, say PHONE-NUMBER, to collect those active integer objects that are also behaviors of class PHONE. This approach allows behaviors with different signatures to be organized into uniform data structures (e.g., sets and sequences), and their descriptive part (values) to be used for associative invocations. In fact, the value of an object may be viewed as a description of the purpose of the knowhow that the object bears. Passive objects may be thought of doing nothing for their purposes. For example, an active object o with a sequence value

$$< \text{"}walk\ in\text{"}, \text{"}find\ a\ seat\text{"}, \text{"}order\ sushi\text{"}, \text{"}eat\ sushi\text{"}, \text{"}pay\ bill\text{"}, \text{"}tip\text{"}, \text{"}leave\text{"} >$$

is capable of achieving a series of actions when $kh(o)$ is activated by a person (object) visiting a Japanese restaurant. As such, this sequence value is actually a high-level specification of the encapsulated code in $kh(o)$. In this sense, the value of an active object may serve as a "complex" annotation of the object's knowhow. Notice that it is only for simplicity that most behaviors in our examples are TEXT objects.

To facilitate further discussions, we introduce some technical notation. For any object $o = (i, c, v, (c_{owner}, \xi, k))$, let $class(o) = c$, $value(o) = v$, $owner(o) = c_{owner}$, and $sig(o) = \xi$. For the sake of brevity, when an object o is a passive object, we usually denote o as a triple

$$o = (id, c, v)$$

as long as it results in no confusion (note that a passive object might become an active object later). Furthermore, throughout this thesis, we often use values instead of object-identities to denote objects for the purpose of clarity.

Example 9: More examples of KBO objects are given in the following. Note that values in the complex objects (i.e., $o_5, o_6, o_7, o_8, o_{11}$) are supposed to contain object-identities; but we use object values instead.

$o_5 = (id_5, \text{PERSON}, [Name\text{: "}JohnSmith\text{"}, Age\text{: } 35])$

$o_6 = (id_6, \text{TAKE-CARE}, \{\text{"}Are\ you\ sick?\text{"}, \text{"}Go\ see\ doctor\text{"}, \text{"}Go\ to\ work\text{"}\})$

$o_7 = (id_7, \text{HAVE-SNACK}, < \text{"}Wash\ hands\text{"}, \text{"}Eat\ crackers\text{"}, \text{"}Still\ hungry?\text{"} >)$

$o_8 = (id_8, \text{EAT-LUNCH}, \{\text{"}Eat\text{"}, \text{"}Chat\text{"}, \text{"}Drink\text{"}\})$

$o_9 = (id_9, \text{TEXT}, \text{"}Are\ you\ sick?\text{"}, (\text{PERSON}, () \longrightarrow \text{BOOL}, kh_9))$

$$o_{10} = (id_{10}, \text{TEXT}, \textit{"Do you belong to this age group?"}, (\text{PERSON}, (\text{Age1}, \text{INT}) \times (\text{Age2}, \text{INT}) \longrightarrow \text{BOOL}, kh_{10}))$$

$$o_{11} = (id_{11}, \text{EXERCISE}, [Do\text{: } \textit{"Jogging!"}, WeighLoss\text{: } 10], (\text{PERSON}, () \longrightarrow \text{SELF}, kh_{11}))$$

where

PERSON Ⓚ AGG
TAKE-CARE Ⓚ BIO
HAVE-SNACK Ⓚ SEQ
EAT-LUNCH Ⓚ SET
TEXT Ⓚ CAPSULE
EXERCISE Ⓚ AGG

In the above, only o_9, o_{10} and o_{11} are active objects. That is, they are created as meaningful behaviors of class PERSON. Notice that they all have inherent knowhows. More comments should be made about o_{11}. This object is an instance of class EXERCISE (an aggregate class) and a behavior of class PERSON (also an aggregate class). The value of o_{11},

$$[Do\text{: } \textit{"Jogging!"}, WeighLoss\text{: } 10]$$

can be thought of as an annotation of the code encapsulated in kh_{11}: the knowhow kh_{11} knows how to make a person do a kind of jogging that can lose about 10 pounds for the person. For example, if we send o_{11} to a person denoted by *Peter*, i.e.,

$$o_{11} \rightsquigarrow Peter$$

we will see *Peter* doing some kind of jogging (e.g., on an animation screen) that will get rid of at least 10 pounds from *Peter*'s belly. The significance of having invokers like o_{11} is that end-users may send messages in an associative manner. For example, if a set object denoted by *MyFavoriteExercises* contains a large number of instances of EXERCISE (including o_{11}), then we can send one or more messages to *Peter* in the following manner:

$$Select_{x.WeighLoss<15}(MyFavoriteExercises) \rightsquigarrow Peter$$

where the expression $Select_{x.WeighLoss<15}(MyFavoriteExercises)$ yields a set of instances of EXERCISE that are designed to lose less than 15 pounds for a person (obviously, o_{11} is one of them). Then, according to our informal semantics of aggregations in behavioral modeling, the selected EXERCISE objects are sent to *Peter* in an arbitrary order, causing *Peter* to do several exercises (o_{11} is one of them). ■

4.2.2 Graphical Representation of Objects

Since complex objects are hierarchically constructed objects, they can be represented graphically. The following definition provides a way to graphically represent KBO objects.

Definition 24: *Object Graph.* The ***object graph*** for a KBO object o is a directed graph such that:

- If o ⓘ CAPSULE, then o is represented by a ○ node with no outgoing edges, and the node is labeled with the value of o;
- If o ⓘ AGG and $o = (id, c, [A_1: id_1, \ldots, A_n: id_n], g)$, then o is represented by a circled *"agg"* node labeled with id, and the node has n outgoing edges labeled $A_1, \ldots, A_n$ leading respectively to the nodes representing objects whose identities are $id_1, \ldots, id_n$;
- If o ⓘ BIO and $o = (id, c, \langle id_{if}, id_{then}, id_{else} \rangle, g)$, then o is represented by a circled *"bio"* node labeled with id, and the node has three outgoing edges labeled *"if"*, *"then"*, *"else"*, leading respectively to the nodes representing objects whose identities are $id_{if}, id_{then}, id_{else}$;
- if o ⓘ SET and $o = (id, c, \{id_1, \ldots, id_n\}, g)$, then o is represented by a circled *"set"* node labeled with id, and the node has n unlabeled outgoing edges leading respectively to the nodes representing objects whose identities are $id_1, \ldots, id_n$;
- if o ⓘ SEQ and $o = (id, c, < id_1, \ldots, id_n >, g)$, then o is represented by a circled *"seq"* node labeled with id, and the node has n outgoing edges labeled with $1, 2, \ldots, n$ leading respectively to the nodes representing objects whose identities are $id_1, \ldots, id_n$.

Example 10: In the following, the (passive) object o_1 is an instance of an aggregate class POP-MENU:

$o_1 = (i_1, \text{POP-MENU}, [\,\textit{Title}: i_2,\ \textit{Residence}: i_3,\ \textit{Items}: i_4,\ \textit{ExitOptions}: i_5\,])$
$o_2 = (i_2, \text{TITLE}, [\,\textit{TopTitle}: i_6,\ \textit{BottomTitle}: i_7\,])$
$o_3 = (i_3, \text{ACCESS-MODE}, \langle \textbf{null}, i_8, i_9 \rangle)$
$o_4 = (i_4, \text{TEXT-SEQ}, < i_{10}, i_{11}, i_{12}, i_{13}, i_{14}, i_{15}, i_{16} >)$
$o_5 = (i_5, \text{TEXT-SET}, \{\ i_{16}, i_{10}, i_{15}\ \})$
$o_6 = (i_6, \text{TEXT}, \text{"\textit{Choose a direction :}"})$

$o_7 = (i_7, \text{TEXT}, \text{"\textit{Off if idle 2 minutes}"})$

$o_8 = (i_8, \text{TEXT}, \text{"\textit{Memory-based}"})$

$o_9 = (i_9, \text{TEXT}, \text{"\textit{Disk-based}"})$

$o_{10} = (i_{10}, \text{TEXT}, \text{"\textit{do nothing}"})$

$o_{11} = (i_{11}, \text{TEXT}, \text{"\textit{North}"})$

$o_{12} = (i_{12}, \text{TEXT}, \text{"\textit{West}"})$

$o_{13} = (i_{13}, \text{TEXT}, \text{"\textit{East}"})$

$o_{14} = (i_{14}, \text{TEXT}, \text{"\textit{South}"})$

$o_{15} = (i_{15}, \text{TEXT}, \text{"\textit{return}"})$

$o_{16} = (i_{16}, \text{TEXT}, \text{"\textit{quit}"})$

where

$$\text{POP-MENU} \textcircled{k} \text{AGG}$$
$$\text{TITLE} \textcircled{k} \text{AGG}$$
$$\text{ACCESS-MODE} \textcircled{k} \text{BIO}$$
$$\text{TEXT-SEQ} \textcircled{k} \text{SEQ}$$
$$\text{TEXT-SET} \textcircled{k} \text{SET}$$

and the content of o_1 can be explained as follows: (1) The *Title* attribute provides a title on the top of the pop-up menu and a message underneath the pop-up menu. (2) The *Residence* attribute indicates how the menu is popped on the screen. If *"Memory-based"*, the pop-up menu is first loaded into memory and then displayed on the screen. If *"Disk-based"*, the pop-up menu is displayed on the screen directly from the database. The former mode speeds up the subsequent displays and the latter mode saves the workstation's memory space. The *Residence* attribute is a bi-object so that the access mode can be explicitly represented and adjusted. Notice that more than two access modes can be expressed if the *Residence* attribute is a bi-object whose then-component (or else-component) is again a bi-object. (3) The *Items* attribute gives a sequence of options that should be listed in the pop-up menu from top to bottom. (4) The *ExitOptions* attribute serves as user help which explains what you could do when you do not want to make a "real" choice: you can choose *"do nothing"* so that the pop-up menu will disappear, you can choose *"return"* to return to the screen you came from, or you can choose *"quit"* to quit the application program. Notice that all the objects reachable from o_1 are passive objects. In other words, they are not intended to model any behavioral composition.

The object graph of o_1 is shown in Figure 4.14. Also shown in Figure 4.14 is the actual screen display of object o_1. One may notice that the pop-up menu o_1 is, at

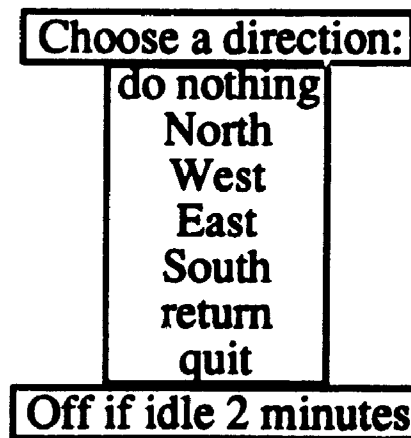


Figure 4.14: The Object Graph and the Screen Display of o_1.

this moment, disk-based, because the if-component of its *Residence* attribute is given as ***null***. Moreover, we see some object-sharing here: objects o_4 and o_5 are sharing o_{10}, o_{15}, o_{16} as their sub-objects. As a result, if we were to change the value of o_{10} from *"do nothing"* to *"do not make a choice"*, the update would be reflected immediately in both o_4 and o_5 (without object identity, o_4 and o_5 have to be updated separately). ■

4.2.3 Object Equalities

In the KBO model, any object has an object-identity, and any complex object reaches other objects via their object-identities. This leads to a requirement for multiple definitions for object equality. Especially, as we will see in Chapter 8, the KBO Algebra, an object algebra associated with the KBO model, actually allows the explicit use of any kind of object equality.

Definition 25: *Identity.* Let o_1, o_2 be two objects in $\mathcal{O}$. Then, o_1 and o_2 are *identical* (denoted $o_1 \equiv o_2$) if $oid(o_1) = oid(o_2)$. ■

Definition 26: *N-Equality.* Let the 0-equality (denoted $\asymp_0$) be the identity test $\equiv$. Then, for any two objects $o_1, o_2 \in \mathcal{O}$, o_1 and o_2 are *n-equal* (denoted $o_1 \asymp_n o_2$) for $n > 0$ if:

1. there exists $c \ \textcircled{k}\ \text{CAPSULE}$, $o_1 \ \textcircled{i}_d\ c$, $o_2 \ \textcircled{i}_d\ c$, such that $value(o_1) = value(o_2)$; or

2. there exists $c \ \textcircled{k}\ \text{AGG}$, $o_1 \ \textcircled{i}_d\ c$, $o_2 \ \textcircled{i}_d\ c$, $value(o_1) = [A_1: x_1, \ldots, A_m: x_m]$ and $value(o_2) = [A_1: y_1, \ldots, A_m: y_m]$, such that $x_i \asymp_{n-1} y_i$ for $1 \leq i \leq m$; or

3. there exists $c \ \textcircled{k}\ \text{BIO}$, $o_1 \ \textcircled{i}_d\ c$ and $o_2 \ \textcircled{i}_d\ c$, $value(o_1) = \{\!|\ x_1, x_2, x_3\ |\!\}$ and $value(o_2) = \{\!|\ y_1, y_2, y_3\ |\!\}$, such that $x_1 \asymp_{n-1} y_1$, $x_2 \asymp_{n-1} y_2$ and $x_3 \asymp_{n-1} y_3$; or

4. there exists $c \ \textcircled{k}\ \text{SET}$, $o_1 \ \textcircled{i}_d\ c$, $o_2 \ \textcircled{i}_d\ c$, $value(o_1) = \{x_1, \ldots, x_l\}$ and $value(o_2) = \{y_1, \ldots, y_m\}$, such that $l = m$ and for each x_i (y_j) there exists a single y_j (x_i), $x_i \asymp_{n-1} y_j$ for $1 \leq i \leq l$; or

5. there exists $c \ \textcircled{k}\ \text{SEQ}$ and $o_1 \ \textcircled{i}_d\ c$, $o_2 \ \textcircled{i}_d\ c$, $value(o_1) = < x_1, \ldots, x_l >$ and $value(o_2) = < y_1, \ldots, y_m >$, such that $l = m$ and $x_i \asymp_{n-1} y_i$ for $1 \leq i \leq m$.

We say that o_1 and o_2 are *equal* (denoted $o_1 \asymp o_2$) if and only if $o_1 \asymp_n o_2$ for some $n > 0$. ■

Notice that this definition is similar to the one in [SZ89], except that we have extended their definition to our BIO and SEQ objects.

A special property of KBO objects is that they all bear a knowhow. Hence, it becomes necessary to introduce an equality based on objects' knowhows. In this way, we can compare whether two KBO objects share the same knowhow.

Definition 27: ***Knowhow-Identity.*** Let o_1, o_2 be two objects in $\mathcal{O}$. Then, o_1 and o_2 are ***knowhow-identical*** (denoted $o_1 \equiv_{kh} o_2$) if $kh(o_1) = kh(o_2)$. ■

From above definitions, the following two properties are trivially true:

Theorem 5:

1. $\forall o_1, o_2 \in \mathcal{O}: o_1 \equiv o_2 \Longrightarrow o_1 \equiv_{kh} o_2$

2. $\forall o_1, o_2 \in \mathcal{O}, \forall i, j, 0 \leq i < j : o_1 \asymp_i o_2 \Longrightarrow o_1 \asymp_j o_2$

Proof: The first property holds because object identities are unique. Therefore, identical objects can only bear the same knowhow. For the second property, we need to notice that if two objects are identical then they are also equal at any level. Therefore, from the definition of n-equalities, the second property obviously holds. ■

It can be observed that two objects being identical not only means they are the same object but also means they bear the same knowhow. A special case is that all passive objects share the same knowhow $\phi_\perp$. However, if two objects are equal, they may or may not bear the same knowhow, just as two objects are traditionally considered equal if they have the same value but may not necessarily be the same object.

4.2.4 A Less Naive Message-Sending Paradigm

With the notion of object equalities, we can modify the naive message-sending paradigm proposed in Chapter 3. For this purpose, we shall first introduce the notions of monomorphic invoker and polymorphic invokers. An invoker is a ***monomorphic invoker*** if it can only result in execution of a single knowhow when sent to objects of different classes. In the other hand, an invoker is a ***polymorphic invoker*** if it may result in execution of different knowhows when sent to objects of different classes. Notice that, while in traditional OODBs an invoker is just a message name, an invoker in the KBO model is an object itself. It is easy to prove that the naive message-sending paradigm introduced in Chapter 3 only supports monomorphic invokers because the knowhow executed in a message-sending must be the knowhow of the invoker. Therefore, no matter what receiver is receiving the invoker, the knowhow executed is always the same.

With the notion of object equalities, we can now modify the naive message-sending paradigm to support not only monomorphic invokers but also polymorphic invokers. Let us again only consider the knowhows that take no labeled arguments. Then, a *less naive message-sending* is an expression of the form:

$$o_{invoker} \leadsto o_{receiver} \Rightarrow o_{result}$$

where the object $o_{invoker}$ is sent as a message to the object $o_{receiver}$, which responds to the message by returning the object o_{result} such that:

- if there is $o_{behavior} \in \overset{*}{\Sigma}(class(o_{receiver}))$ and $o_{behavior} \equiv o_{invoker}$, then

$$o_{result} = o_{receiver}.\overline{kh(o_{invoker})}()$$

  Otherwise,

- if there is $o_{behavior} \in \overset{*}{\Sigma}(class(o_{receiver}))$ and $o_{behavior} \asymp o_{invoker}$, then

$$o_{result} = o_{receiver}.\overline{kh(o_{behavior})}()$$

Intuitively, what this new paradigm has improved is that when an invoker is sent to a receiver, the receiver looks into its interface to find a behavior *identical* to the invoker (the Invoker-Identity-Sensitive Property). If the search succeeds, the receiver executes the knowhow of the behavior found. If the search fails, the receiver then tries to find a behavior *equal* to the invoker, and, if found, execute the knowhow of that behavior. The first search by the receiver is intended to support monomorphism (i.e., to pinpoint a knowhow by the invoker), and the second search is intended to support polymorphism because it effectively lets the receiver itself decide which behavior's knowhow should be executed.

We illustrate this paradigm by the following example.

Example 11: In the following, class PROFESSOR is not only a subclass but also a demandclass of class INSTRUCTOR, that is, PROFESSOR Ⓚ INSTRUCTOR and PROFESSOR ⓡ INSTRUCTOR. Let the interface $\overset{*}{\Sigma}$(PROFESSOR) = $\{o_1, o_2\}$ where o_1 is directly reused from INSTRUCTOR and o_2 is created (locally) as a direct behavior of PROFESSOR:

$$o_1 = (id_1, \text{TEXT}, \text{“}\textit{Introduce yourself}\text{”}, (\text{INSTRUCTOR}, () \longrightarrow \text{TEXT}, kh_1))$$
$$o_2 = (id_2, \text{TEXT}, \text{“}\textit{Introduce yourself}\text{”}, (\text{PROFESSOR}, () \longrightarrow \text{TEXT}, kh_2))$$

As shown above, o_1 and o_2 are instances of TEXT and behaviors of PROFESSOR. Notice that both o_1 and o_2 have an inherent knowhow. Also, kh_1 and kh_2 have different semantics. Intuitively, the way we want a professor to "introduce himself/herself" will be different from the way we want an instructor to "introduce himself/herself" (e.g., we may like kh_2 to show all the papers a professor has published, or to suppress a side-effect encapsulated in kh_1 which causes the photo of an instructor to be displayed). Now, let *Codd* denote an instance of PROFESSOR. Based on our less naive message-sending paradigm, the message-sending

$$o_1 \rightsquigarrow Codd$$

will pinpoint the knowhow kh_1 for execution, thus introducing professor *Codd* as an "instructor", while the message-sending

$$o_2 \rightsquigarrow Codd$$

will pinpoint knowhow kh_2 for execution, thus introducing professor *Codd* as a "professor". This is because in both message-sendings the invoker is identical to a behavior in the interface of PROFESSOR, and therefore the invoker effectively makes the decision about "which knowhow to execute". Suppose now we create a new (passive) TEXT object

$$o_3 = (id_3, \text{TEXT}, \text{"}\textit{Introduce yourself}\text{"})$$

and we send it to *Codd*, that is,

$$o_3 \rightsquigarrow Codd$$

Because no behavior in *Codd*'s interface (which contains o_1 and o_2) is identical to o_3 but some behaviors (i.e , both o_1 and o_2) are equal to o_3, the receiver *Codd* will decide whether o_1 or o_2 should be used to respond to the message. Note that if o_1 is the only behavior in the interface of PROFESSOR, then o_1 will definitely be chosen. In many cases (like this example), however, the receiver has to choose a behavior from more than one candidate with the same value. Polymorphism is supported in the sense that different receivers (say a instructor and a professor) may choose different knowhows to respond to the invoker o_3. ■

In the above example, we actually encounter a situation which we call a "value conflict" among candidate behaviors. Such a conflict will be discussed in the next section. Generally speaking, we will have to deal with two kinds of value conflicts: ***horizontal value conflicts*** and ***vertical value conflicts***. In the above example, class PROFESSOR has a direct behavior o_2 with value "Introduce yourself" and also reuses another behavior o_1 from INSTRUCTOR with the same value "Introduce yourself".

This is an example of vertical value conflict because PROFESSOR is "lower" than INSTRUCTOR in the corresponding REUSE-OF hierarchy. To illustrate horizontal value conflicts, let us assume that PROFESSOR does not have a direct behavior with value "Introduce yourself". Instead, PROFESSOR is a reuse of both INSTRUCTOR and RESEARCHER, and thus reuses o_1 from INSTRUCTOR and o_2 from RESEARCHER:

$$o_1 = (id_1, \text{TEXT}, \text{“}\textit{Introduce yourself}\text{”}, (\text{INSTRUCTOR}, () \longrightarrow \text{TEXT}, kh_1))$$
$$o_2 = (id_2, \text{TEXT}, \text{“}\textit{Introduce yourself}\text{”}, (\text{RESEARCHER}, () \longrightarrow \text{TEXT}, kh_2))$$

Thus, PROFESSOR has two equal behaviors (o_1 and o_2) which are reused from the "same level" — an example of a horizontal value conflict. Generally speaking, vertical value conflicts may be resolved by following the conventional approach: choose the "nearest" behavior. However, resolving horizontal value conflicts is not as obvious.

4.3 KBO Classes

4.3.1 Class Definition

Definition 28: *Classes.* A class is a quintuple

$$(\ c,\ \Phi_K(c),\ \Phi_R(c),\ \Xi(c),\ \Sigma(c)\ )$$

where

1. $c \in C$ is the class name;

2. $\Phi_K(c) = \{c' \mid c \ \textcircled{k}_d\ c'\}$ is the set of all *direct superclasses* of c;

3. $\Phi_R(c) = \{c' \mid c \ \textcircled{r}_d\ c'\}$ is the set of all *direct supplyclasses* of c;

4. $\Xi(c)$ is the *local representation* of c, such that:
   (4.1) if c $\textcircled{k}$ CAPSULE then $\Xi(c) = D_i$ where $D_i \in C_{base}$,
   (4.2) if c $\textcircled{k}$ AGG then $\Xi(c) = [A_1 \colon c_1, \ldots, A_n \colon c_n]$
   where $\{(A_i, c_i) \mid 1 \leq i \leq n\} \subseteq \{(A_x, c_x) \mid (A_x, c_x)\ \textcircled{p}_d\ c\}$,
   (4.3) if c $\textcircled{k}$ BIO then $\Xi(c) = \{\ c_1, c_2, c_3\ \}$
   where $(\emptyset_{if}, c_1)\ \textcircled{p}_d\ c$, $(\emptyset_{then}, c_2)\ \textcircled{p}_d\ c$ and $(\emptyset_{else}, c_3)\ \textcircled{p}_d\ c$,
   (4.4) if c $\textcircled{k}$ SET then $\Xi(c) = \{c_1\}$ where $(\emptyset_{member}, c_1)\ \textcircled{p}_d\ c$,
   (4.5) if c $\textcircled{k}$ SEQ then $\Xi(c) = < c_1 >$ where $(\emptyset_{ordered_member}, c_1)\ \textcircled{p}_d\ c$; and

5. $\Sigma(c)$ is the *local interface* of c, $\Sigma(c) = \{o \mid o \circledcirc_d c\}$ (i.e., all c's direct behaviors), such that:

   (5.1) $\forall o_1, o_2 \in \Sigma(c): o_1 \not\asymp o_2$

   (5.2) $\forall c_1, c_2 \in \Phi_R(c), \forall o_1 \in \dot{\Sigma}(c_1), \forall o_2 \in \dot{\Sigma}(c_2)$:

   $$o_1 :: o_2 \wedge o_1 \not\equiv_{kh} o_2 \Longrightarrow \exists o_3 \in \Sigma(c): o_3 \asymp o_1$$

We shall denote a KBO class by

$$c = (\ \Phi_K(c),\ \Phi_R(c),\ \Xi(c),\ \Sigma(c)\ )$$

■

In the above definition, axiom (5.1) states that no two local behaviors are equal, and axiom (5.2) states that if c reuses two equal behaviors with different knowhows from two different direct supplyclasses, then c must have a local behavior with the same value. This restriction will allow us to derive a unique "value-based" interface defined at the end of this chapter. Note that two behaviors in the local interface can have the same knowhow but different values. Furthermore, the local representation of an aggregate class may contain only a subset of all the direct parts of c.

Example 12: The following are the local representations of five KBO classes: a capsule class ODD-INTEGER, an aggregate class EMPLOYEE, a bi-object class SENSITIVE-PLANT, a set class PEOPLE, and a sequence class THESIS:

Ξ(ODD-INTEGER) = INT
Ξ(EMPLOYEE) = [JobTitle: TEXT, Salary: INT, Manager: EMPLOYEE]
Ξ(SENSITIVE-PLANT) = { TOUCH, FOLDED-PLANT, UNFOLDED-PLANT }
Ξ(PEOPLE) = {PERSON}
Ξ(THESIS) = <CHAPTER>
■

Example 13: The following could be the class definition for the class BNR-EMPLOYEE illustrated in Figure 3.6:

BNR-EMPLOYEE = (
    { EMPLOYEE },
    { IBM-EMPLOYEE, APPLE-EMPLOYEE },
    [Projects: VLSI-PROJECTS, FamiliarWith: VLSI-PACKAGES],
    $\{b_4\}$)

where b_4 is the only direct behavior of BNR-EMPLOYEE, and $value(b_4)$ = "VLSI design experience". The knowhow $kh(b_4)$ knows how to compute the sum of all the time a BNR employee has spent on VLSI projects and produce a number in months. ■

It can be observed that in the above definition, we impose two constraints on a class's local interface in order to *explicitly* resolve possible horizontal value conflicts. We feel that it is more accurate to place the resolution of conflicts into the hands of class designers (at least in the present framework), than to let the system do the job based on a total ordering. With these constraints, a horizontal value conflict can be resolved without being biased towards any one of c's supplyclasses that are equally "close" to the class c. This can be illustrated by the following example.

Example 14: Let class PROFESSOR be a reuse of both INSTRUCTOR and RESEARCHER, and thus reusing o_1 from INSTRUCTOR and o_2 from RESEARCHER:

$$o_1 = (id_1, \text{TEXT}, \text{"}\textit{Introduce yourself}\text{"}, (\text{INSTRUCTOR}, () \longrightarrow \text{TEXT}, kh_1))$$
$$o_2 = (id_2, \text{TEXT}, \text{"}\textit{Introduce yourself}\text{"}, (\text{RESEARCHER}, () \longrightarrow \text{TEXT}, kh_2))$$

Then, the designer of PROFESSOR will be notified that he should create a direct behavior with value "Introduce yourself" in order to resolve the horizontal value conflict caused by reusing o_1 and o_2. In fact, say the designer actually wishes to use the knowhow of o_1. This can be accomplished by creating a direct behavior o_3 in the following manner:

$$o_3 = (id_3, \text{TEXT}, \text{"}\textit{Introduce yourself}\text{"}, (\text{PROFESSOR}, () \longrightarrow \text{TEXT}, kh_{o_1}))$$

where kh_{o_1} indicates that the knowhow of o_3 is borrowed from o_1. This is allowable because PROFESSOR is a reuse of INSTRUCTOR. Therefore, the (identity-based) interface of PROFESSOR will be $\overset{*}{\Sigma}(\text{PROFESSOR}) = \{o_1, o_2, o_3\}$ while its local interface is $\Sigma(\text{PROFESSOR}) = \{o_3\}$. Behavior o_3 will also be included in a "value-based" interface (the notion of a value-based interface will be introduced shortly). ■

4.3.2 Full Representations

We now define the (full) representation of a KBO class. In particular, the representation of an aggregate class is derived from its local representation and the full

representations of all its supplyclasses.

Definition 29: ***Class Representations.*** For any class $c = (\ \Phi_K(c), \Phi_R(c), \Sigma(c), \Xi(c)\ )$, its *full representation*, denoted $\dot{\Xi}(c)$, is defined as follows:

- if c ⓚ CAPSULE or c ⓚ BIO or c ⓚ SET or c ⓚ SEQ, then $\dot{\Xi}(c) = \Xi(c)$.
- if c ⓚ AGG, then $\dot{\Xi}(c) = [A_1\text{: } c_1, \ldots, A_n\text{: } c_n, A'_1\text{: } c'_1, \ldots, A'_m\text{: } c'_m]$, where
    1. $\Xi(c) = [A_1\text{: } c_1, \ldots, A_n\text{: } c_n]$
    2. $\forall i \in [1..m]: A'_i \notin \{A_1, \ldots, A_n\}$
    3. $\forall i \in [1..m], \exists c_x \in \Phi_R(c): \dot{\Xi}(c_x) = [\ldots, A'_i\text{: } c'_i, \ldots]$
    4. $\forall i \in [1..m], \forall c_x \in \Phi_R(c): \dot{\Xi}(c_x) = [\ldots, A'_i\text{: } c_y, \ldots] \Longrightarrow c'_i$ ⓚ c_y
    5. $\forall c_x \in \Phi_R(c):$
       $\dot{\Xi}(c_x) = [\ldots, A\text{: } c_y, \ldots] \wedge A \notin \{A'_1, \ldots, A'_m\} \Longrightarrow A \in \{A_1, \ldots, A_n\}$

Some comments are needed for the definition of the full representation of an aggregate class. First, the full representation of an aggregate class is the combination of both its locally declared attributes (i.e., $A_1, ..., A_n$) and the attributes reused from its supplyclasses (i.e., $A'_1, ..., A'_m$). This is stipulated in Conditions 1, 2 and 3. Second, when there is more than one supplyclass having the same attribute name, the most specific attribute (class) is chosen among those attributes if it can be found (Condition 4); otherwise, we require the designer to explicitly declare a local attribute with the same attribute name (Condition 5). In other words, an incompatible attribute name conflict is explicitly resolved by the class designer.

Example 15: Assume we have the following representations for class STUDENT and class EMPLOYEE:

$$\dot{\Xi}(\text{STUDENT}) = [\text{Name: NAME, Friend: STUDENT}]$$
$$\dot{\Xi}(\text{EMPLOYEE}) = [\text{Name: FULLNAME, Friend: EMPLOYEE}]$$

where FULLNAME is a subclass of NAME. Assume there is no subclass relationship between STUDENT and EMPLOYEE. Let class TA be a reuse of STUDENT and EMPLOYEE, and its local representation be $\Xi(\text{TA}) = [\text{Friend: TA, Job: LABS}]$ Originally, the designer of TA only wanted to have $\Xi(\text{TA}) = [\text{Job: LABS}]$ and have the rest of the attributes supplied from its supplyclasses STUDENT and EMPLOYEE. However, the designer is required to resolve the conflict between the two Friend attributes

from STUDENT and EMPLOYEE because neither is a subclass of the other. As a result, a new local attribute Friend (with attribute type TA here) must be specified withir the local representation of TA. For the conflict between two Name attributes, because FULLNAME is a subclass of NAME, the conflict is automatically resolved. Hence, the full representation of TA is derived as:

$$\dot{\Xi}(\text{TA}) = [\text{Name: FULLNAME, Friend: TA, Job: LABS}].$$

Intuitively speaking, a class representation is the data structure portion of a class which provides attributes for the class's behaviors to access and update. In other words, the implementations of a class's behaviors rely on the representation of that class. In the above example, the implementations of the behaviors of EMPLOYEE rely on representation

[Name: FULLNAME, Friend: EMPLOYEE]

and they can be safely reused by TA, whose representation is

[Name: FULLNAME, Friend: TA, Job: LABS]

only if the class of attribute Friend (i.e., TA) is a subclass of EMPLOYEE. One should notice that only the attributes in the supplyclasses (not the superclasses) are physically reused (traditionally called "inherited") by an aggregate class. Furthermore, the representation of a class also serves as a mold, and its instances are castings from the mold. In other words, the memory for an instantiation of $\dot{\Xi}(c)$ must be allocated whenever an instance of c is created. Only the behaviors in $\dot{\Sigma}(c)$ are allowed to access and modify the representation $\dot{\Xi}(c)$.

4.3.3 Value-based Interfaces

As defined previously, the interface of a class c, i.e., $\dot{\Sigma}(c)$, is a set of (behavior) objects; some of these objects may have the same value. In the following, we introduce the notion of the value-based interface of a class, which is basically a subset of the interface of the class.

Definition 30: *Value-based Interfaces.* The *value-based interface* of a class c, written $\mathring{\Sigma}(c)$, is defined by:

$$\mathring{\Sigma}(c) = \Sigma(c) \cup \{o \mid o \in \dot{\Sigma}(c) \wedge (\forall x \in \Sigma(c): o \not\asymp x) \wedge (\exists c' \in \Phi_R(c): o \in \mathring{\Sigma}(c'))\}$$

■

We shall let $\mathring{\Sigma}(\text{OBJECT}) = \mathring{\Sigma}(\text{AGG}) = \mathring{\Sigma}(\text{BIO}) = \mathring{\Sigma}(\text{SET}) = \mathring{\Sigma}(\text{SEQ}) = \emptyset$. That is, these kernel classes do not have (user-defined) behaviors. This assumption insures that the above recursive definition for value-based interfaces terminates for every class in C.

From the definition, the value-based interface of a class is the union of its local interface and the set of all its indirect behaviors that are not equal to any behavior in the local interface and that belong to the value-based interface of a direct supplyclass. In Example 15, it is easy to derive $\mathring{\Sigma}(\text{PROFESSOR}) = \{o_3\}$. The definition of a class and the definition of a value-based interface together guarantee that behaviors in a value-based interface are pair-wise not equal. That is, there is no value conflict in a value-based interface.

With the notion of value-based interfaces, we can redefine our "less naive message-sending paradigm" in a more precise way: A ***less naive message-sending*** is an expression of the form:

$$o_{invoker} \leadsto o_{receiver} \Rightarrow o_{result}$$

where the object $o_{invoker}$ is sent as a message to the object $o_{receiver}$, which responds to the message by returning the object o_{result} such that:

1. If there exists a behavior $o_{behavior} \in \mathring{\Sigma}(class(o_{receiver}))$ and $o_{behavior} \equiv o_{invoker}$, then $o_{result} \equiv o_{receiver}.\overline{kh(o_{invoker})}()$; otherwise,

2. If there exists a behavior $o_{behavior} \in \mathring{\Sigma}(class(o_{receiver}))$ and $o_{behavior} \asymp o_{invoker}$, then $o_{result} \equiv o_{receiver}.\overline{kh(o_{behavior})}()$; otherwise,

3. $o_{result} \equiv \boldsymbol{fail}$.

Notice that this less naive message-sending paradigm has improved the naive message-sending paradigm by (i) supporting polymorphism based on the notion of a value-based interface, and (ii) supporting receiver-qualification, that is, the receiver can execute a behavior's knowhow only when that behavior is a behavior of the receiver's class (i.e., either the interface or the value-based interface needs to be searched before the execution).

CHAPTER 5
A DATABASE LEVEL SEMANTICS

5.1 Introduction

In Chapter 3, we introduced five conceptual abstractions to model real-world applications: the INSTANCE-OF, BEHAVIOR-OF, KIND-OF, REUSE-OF and PART-OF relationships. Together they provide a formal, conceptual level semantics for the KBO model. Since a database is but a representation of our perceptions of reality, the conceptual level semantics allows us to model or describe perceptions of the real world through the five abstractions. However, many practical issues are not dealt with at the conceptual level, such as what "*class*1 KIND-OF *class*2" means in terms of the syntax of the representations and the behavior signatures in the two classes. Often the user may have a correct conceptual understanding about what he/she is modeling. For example, the user thinks that "every student is also a person, so class STUDENT should be made a subclass of class PERSON". However, the representation and behaviors of class STUDENT may somehow be designed so that they are not compatible with those of class PERSON. The result is that a STUDENT object cannot be meaningfully used as a PERSON object in the database. In other words, correct conceptual understanding does not necessarily result in a correct corresponding design in the database. Therefore, our database design must be consistent with our conceptual level semantics.

In this chapter, we provide a formal database semantics for the KBO model, which can be syntactically validated (by a parser) against the users' class declarations. This database level semantics is based on three *interpretation* functions which associate three subsets of a consistent set of database objects to each KBO class. Intuitively speaking, the static interpretation of a KBO class contains all the objects which can "simulate" the representation of that class; the dynamic interpretation of a KBO class contains all the objects (behaviors) which can "simulate" the interface of that class; and the reuse interpretation of a KBO class contains all the objects (behaviors) which can be correctly reused as the behaviors of that class. Based on the inclusion of these interpretations, we then define two types of compatibilities among KBO classes — k-compatibility and r-compatibility. Syntactic mechanisms are then presented for the validation of the two compatibilities. Finally, we will formally define the notions of a KBO schema and a KBO database.

5.2 Three Interpretation Functions

A KBO database should contain a finite set of KBO objects. As in the O_2 data model [LRV88], we shall require that there be no dangling object-identities in our set of objects. Also, since our KBO objects can bear a knowhow which may be borrowed from other KBO objects, we require that if an object bears a knowhow borrowed from another object then the latter should be one of the objects in our set. This requirement may be called "no dangling borrowed knowhows". Furthermore, in order to the ensure success and safety of our complex behaviors (discussed in Chapter 7), we have to require that all the objects in our set bear a knowhow that is bound to terminate with no failure. All these requirements lead us to introduce the notion of consistency for a set of KBO objects.

Definition 31: *Consistent Set of KBO Objects.* A set Θ of KBO objects in $\mathcal{O}$ is *consistent* if all the following conditions hold:

(1) Θ is finite;

(2) $\forall o \in \Theta: orf(o) \subseteq \{oid(x) | \, x \in \Theta\}$ (i.e., no dangling object-identities);

(3) $\forall o \in \Theta: kh(o) = kh_{o'} \Longrightarrow o' \in \Theta$ (i.e., no dangling borrowed knowhows);

(4) for every $o \in \Theta$, if $kh(o)$ is not $\phi_{\perp}$ then the execution of $kh(o)$ must terminate and return a non-***fail*** result.

Notice that condition (4) suggests that we are only interested in dealing with knowhows that are professionally coded and tested. The above definition is very important, as we will later define a KBO database to be a consistent set of KBO objects.

Let Θ be a consistent set of objects. We say that a class c is *defined on* Θ if the instances and behaviors of c are only objects from Θ. In the following, we give three interpretations for all classes defined on a consistent set of objects. Each class c will be mapped to three sets of object-identities, the *static interpretation*, the *dynamic interpretation*, and the *reuse interpretation*. Note that our definition of the static interpretation follows the general philosophy of the O_2 data model [LRV88, LR88].

5.2.1 Static Interpretation

This subsection deals with the semantics of the representation of a KBO class. Unlike the population of a class, which contains objects explicitly declared by users as the instances of that class, the static interpretation of a class is, intuitively, the set consisting of all objects (precisely, object identities) with a structure "similar" to the representation of that class. In particular, the static interpretation of an aggregate

class c with representation $\dot{\Xi}(c) = [A_1\colon c_1, ..., A_n\colon c_n]$ includes all aggregate objects whose classes have a representation that "contains" $\dot{\Xi}(c)$. For example, if class PERSON has the following representation

$$\dot{\Xi}(\text{PERSON}) = [Name\colon \text{TEXT}, Address\colon \text{TEXT}]$$

then objects with values like

$$[Name\colon \text{"}John\text{"}, Address\colon \text{"}London\text{"}, Smoke\colon \textbf{True}]$$

or

$$[Name\colon \text{"}IBM\ Lab\text{"}, Address\colon \text{"}Toronto\text{"}, NumberOfEmployees\colon 56]$$

will all be included in the static interpretation of class PERSON, regardless of their intended conceptual semantics.

We now formally define the notion of static interpretation:

Definition 32: *Static Interpretation.* Let Θ be a consistent set of objects in $\mathcal{O}$. Let c be a class defined on Θ. Then, the ***static interpretation*** of class c, written $I_s(c)$, is defined as follows:

1. $I_s(\text{OBJECT}) = \Theta$
2. $I_s(\text{BOTTOM}) = \{\boldsymbol{fail}, \boldsymbol{null}, \boldsymbol{same}\}$
3. (capsule classes)
   If $\dot{\Xi}(c) = D_i$ then $I_s(c) = I_s(\text{BOTTOM}) \cup \{oid(x) \mid x \in \Theta \wedge value(x) \in dom(D_i)\}$
4. (aggregate classes)
   If $\dot{\Xi}(c) = [A_1\colon c_1, \ldots, A_n\colon c_n]$ then $I_s(c) = I_s(\text{BOTTOM}) \cup$
   $\{oid(x) \mid x \in \Theta \wedge value(x) = [A_1\colon id_1, \ldots, A_n\colon id_n, A_{n+1}\colon id_{n+1}, \ldots, A_{n+m}\colon id_{n+m}]$
   $\wedge\, m \geq 0 \wedge \forall i \in [1..n]\colon id_i \in I_s(c_i)\}$
5. (bi-object classes)
   If $\dot{\Xi}(c) = \langle\!| c_1, c_2, c_3 |\!\rangle$ then $I_s(c) = I_s(\text{BOTTOM}) \cup$
   $\{oid(x) \mid x \in \Theta \wedge value(x) = \langle\!| id_1, id_2, id_3 |\!\rangle \wedge \forall i \in [1..3]\colon id_i \in I_s(c_i)\}$
6. (set classes)
   If $\dot{\Xi}(c) = \{c'\}$ then $I_s(c) = I_s(\text{BOTTOM}) \cup$
   $\{oid(x) \mid x \in \Theta \wedge value(x) = \{id_1, \ldots, id_n\} \wedge n \geq 0 \wedge \forall i \in [1..n]\colon id_i \in I_s(c')\}$
7. (sequence classes)
   If $\dot{\Xi}(c) = <c'>$ then $I_s(c) = I_s(\text{BOTTOM}) \cup$
   $\{oid(x) \mid x \in \Theta \wedge value(x) = <id_1, \ldots, id_n> \wedge n \geq 0 \wedge \forall i \in [1..n]\colon id_i \in I_s(c')\}$

It should be observed that, from the above definition, the static interpretation of c will at *least* contain the identities of instances of class c.

5.2.2 Dynamic Interpretation

This subsection deals with the semantics of the interface of a KBO class. Informally, if a class c has a behavior $o \in \Theta$ which has, without loss of generality, a signature $(A, c_1) \longrightarrow c_2$, then the dynamic interpretation of c will include all the behaviors x in Θ which (i) are equal to o and (ii) have a signature $(A, c'_1) \longrightarrow c'_2$ such that the static interpretations of c'_1 and c'_2 are a superset of the static interpretations of c_1 and c_2 respectively. The intuition here is that, syntactically, the behavior o can be used in the context where the behaviors x are expected. More formally, we have the following definition:

Definition 33: *Dynamic Interpretation.* Let Θ be a consistent set of objects in $\mathcal{O}$. Let c be a class defined on Θ. Then, the ***dynamic interpretation*** of class c, written $I_d(c)$, is defined as follows:

$$
\begin{aligned}
I_d(c) = \{oid(x) |\ & x \in \Theta \\
& \wedge\ sig(x) = (A_1, c'_1) \times \ldots \times (A_n, c'_n) \longrightarrow c'_{out} \\
& \wedge\ (\exists o \in \overset{\circ}{\Sigma}(c) : o \asymp x \\
& \qquad \wedge\ sig(o) = (A_1, c_1) \times \ldots \times (A_n, c_n) \longrightarrow c_{out} \\
& \qquad \wedge\ \forall i \in [1..n] : I_s(c_i) \subseteq I_s(c'_i) \\
& \qquad \wedge\ I_s(\hat{c}_{out}) \subseteq I_s(\hat{c}'_{out})\ )\ \}
\end{aligned}
$$

where $\hat{c}_{out} = owner(o)$ if c_{out} = SELF otherwise $\hat{c}_{out} = c_{out}$, and $\hat{c}'_{out} = owner(x)$ if c'_{out} = SELF otherwise $\hat{c}'_{out} = c'_{out}$. ■

Intuitively, the dynamic interpretation of a class c is the set of all behaviors in Θ that are more "general" than at least one behavior of c. For example, suppose we have a behavior o_1 with value "make a match" and signature

(Person,NON-SMOKER) ⟶ NON-SMOKER

and another behavior o_2 with the value "make a match" but different signature

(Person,PERSON) ⟶ PERSON

Now, if the static interpretation of class NON-SMOKER is a subset of the static interpretation of class PERSON, i.e., I_s(NON-SMOKER) $\subseteq$ I_s(PERSON), then we consider behavior o_2 to be a more "general" version of behavior o_1.

5.2.3 Reuse Interpretation

This subsection deals with the semantics of the reusability of a KBO class. Recall that, conceptually, the behaviors of a class c can borrow any knowhows from the behaviors of other classes as long as there is a REUSE-OF relationship between c and those classes. However, the reused knowhows in class c may be given a new value or a new signature as we illustrated in the beginning of Chapter 4. In other words, the behaviors of c that reuse some existing knowhows may not be syntactically compatible with those behaviors bearing the original knowhows. Therefore, the reuse interpretation of a class is intended to include only those existing behaviors whose knowhows can be *properly* reused by the class. Formally, we have the following definition:

Definition 34: ***Reuse Interpretation.*** Let Θ be a consistent set of objects in $\mathcal{O}$. Let c be a class defined on Θ. Then, the ***reuse interpretation*** of class c, written $I_r(c)$, is defined as follows:

$$
\begin{aligned}
I_r(c) = \{oid(x) \mid x \in \Theta \\
&\wedge I_s(c) \subseteq I_s(owner(x)) \\
&\wedge (\exists y \in \dot{\Sigma}(c) : y \equiv_{kh} x) \\
&\wedge (\forall o \in \Sigma(c) : (o \asymp x \vee o \equiv_{kh} x) \\
&\qquad \wedge\, sig(o) = (A_1, c_1) \times \ldots \times (A_n, c_n) \longrightarrow c_{out} \\
&\qquad \Longrightarrow sig(x) = (A_1, c'_1) \times \ldots \times (A_n, c'_n) \longrightarrow c'_{out} \\
&\qquad\quad \wedge \forall i \in [1..n] : I_s(c_i) \subseteq I_s(c'_i) \wedge I_s(\hat{c}_{out}) \supseteq I_s(\hat{c}'_{out}) \;) \; \}
\end{aligned}
$$

where $\hat{c}_{out} = owner(o)$ if $c_{out} =$ SELF otherwise $\hat{c}_{out} = c_{out}$, and $\hat{c}'_{out} = owner(x)$ if $c'_{out} =$ SELF otherwise $\hat{c}'_{out} = c'_{out}$. ■

In other words, the reuse interpretation of c will include all the behaviors x in Θ such that (1) the owner class of x has a representation that can be supported by the representation of c, that is, $I_s(c) \subseteq I_s(owner(x))$; and (2) there is a behavior in the interface of c which is knowhow-identical to x; and (3) if there is a behavior o in the local interface of c which is either equal or knowhow-identical to x and which has, without loss in generality, a signature $(A, c_1) \longrightarrow c_2$, then x must have a signature $(A, c'_1) \longrightarrow c'_2$ such that the static interpretation of c'_1 is a superset of the static interpretation of c_1 but the static interpretation of c'_2 is a subset of the static interpretation of c_2. Notice that condition (2) implies the existence of a REUSE-OF relationship between class c and $owner(x)$. Intuitively, the reuse interpretation of a class c is the set of all behaviors in Θ whose knowhows can be safely and compatibly

reused by c's behaviors. For example, a behavior o_1 with signature

$$(\text{Person},\text{PERSON}) \longrightarrow \text{NON-SMOKER}$$

cannot borrow the knowhow of behavior o_2 with signature

$$(\text{Person},\text{PERSON}) \longrightarrow \text{PERSON}$$

if the static interpretation of PERSON is not a subset of the static interpretation of NON-SMOKER. Intuitively, this means that the encapsulated code in the knowhow of o_2 may produce a PERSON object which may not be used as a NON-SMOKER object (e.g., this PERSON object could be a smoker). Similarly, a behavior o_3 with signature

$$(\text{Person},\text{PERSON}) \longrightarrow \text{PERSON}$$

cannot borrow the knowhow of behavior o_4 with signature

$$(\text{Person},\text{NON-SMOKER}) \longrightarrow \text{PERSON}$$

if the static interpretation of NON-SMOKER is not a superset of the static interpretation of PERSON. Intuitively, this means that the encapsulated code in the knowhow of o_4 may not be able to take a PERSON argument (which could be a smoker) because it is implemented only to take a NON-SMOKER argument.

5.2.4 Examples

In the following, we give two examples to illustrate the three interpretations for KBO classes.

Example 16: In this example, we assume that $id_{10}, id_{12}, id_{15} \in I_s(\text{TEXT})$, $id_{11} \in I_s(\text{STUDENTS})$, $id_{13} \in I_s(\text{MSC-STUDENTS})$, $id_{16} \in I_s(\text{PHD-STUDENTS})$, $id_{14}, id_{17} \in I_s(\text{PROFESSOR})$, and $id_{18} \in I_s(\text{BOOL})$. Also assume that

$$I_s(\text{PHD-STUDENTS}) \subseteq I_s(\text{MSC-STUDENTS}) \subseteq I_s(\text{STUDENTS})$$

Let Θ be the following set of KBO objects (including those objects identified by $id_{10}, ..., id_{18}$).

$o_1 = (id_1, \text{STUDENT}, [Name{:}\ id_{10}, ClassMates{:}\ id_{11}])$
$o_2 = (id_2, \text{MSC-STUDENT}, [Name{:}\ id_{12}, ClassMates{:}\ id_{13}, Supervisor{:}\ id_{14}])$
$o_3 = (id_3, \text{PHD-STUDENT},$

$[Name: id_{15}, ClassMates: id_{16}, Supervisor: id_{17}, Qualified: id_{18}])$

$o_4 = (id_4, \text{TEXT}, \text{"Is your classmate?"},$
$(\text{STUDENT}, (\text{Student,STUDENT}) \longrightarrow \text{BOOL}, kh_4))$

$o_5 = (id_5, \text{TEXT}, \text{"Is your classmate?"},$
$(\text{MSC-STUDENT}, (\text{Student,MSC-STUDENT}) \longrightarrow \text{BOOL}, kh_5))$

$o_6 = (id_6, \text{TEXT}, \text{"Is your classmate?"},$
$(\text{PHD-STUDENT}, (\text{Student,PHD-STUDENT}) \longrightarrow \text{BOOL}, kh_{o_5}))$

where (representations)

$\dot{\Xi}(\text{STUDENT}) = [\text{Name: TEXT, ClassMates: STUDENTS}]$
$\dot{\Xi}(\text{MSC-STUDENT}) = [\text{Name: TEXT, ClassMates: MSC-STUDENTS,}$
$\text{Advisor: PROFESSOR}]$
$\dot{\Xi}(\text{PHD-STUDENT}) = [\text{Name: TEXT, ClassMates: PHD-STUDENTS,}$
$\text{Advisor: PROFESSOR, Qualified: BOOL}]$

and (interfaces)

$\dot{\Sigma}(\text{STUDENT}) = \{o_4\}$
$\dot{\Sigma}(\text{MSC-STUDENT}) = \{o_5\}$
$\dot{\Sigma}(\text{PHD-STUDENT}) = \{o_6\}$

Here suppose that PHD-STUDENT is a reuse of MSC-STUDENT, but neither PHD-STUDENT nor MSC-STUDENT is a reuse of STUDENT. Note that knowhows kh_4 and kh_5 know how to do the "same thing" (i.e., to decide who can be the classmate of the receiver) but *in different ways.* Furthermore, knowhow kh_5 is borrowed by the behavior o_6 since the knowhow of o_6 is denoted as kh_{o_5}. The following table gives the static, dynamic and reuse interpretations for classes STUDENT, MSC-STUDENT and PHD-STUDENT defined on Θ:

class: c	$I_s(c)$	$I_d(c)$	$I_r(c)$
STUDENT	$\{id_1, id_2, id_3\}$	$\{id_4\}$	$\{id_4\}$
MSC-STUDENT	$\{id_2, id_3\}$	$\{id_4, id_5\}$	$\{id_5\}$
PHD-STUDENT	$\{id_3\}$	$\{id_4, id_5, id_6\}$	$\{id_5, id_6\}$

Let us comment on the interpretations for class PHD-STUDENT. Object id_3 is included in the static interpretation of PHD-STUDENT, i.e., $I_s(\text{PHD-STUDENT})$, because it is the only object that contains all the attributes specified in $\dot{\Xi}(\text{PHD-STUDENT})$

and because, under our assumptions, $id_{15} \in I_s$(TEXT), $id_{16} \in I_s$(PHD-STUDENTS), $id_{17} \in I_s$(PROFESSOR), and $id_{18} \in I_s$(BOOL). Object id_4 is included in the dynamic interpretation of PHD-STUDENT, i.e., I_d(PHD-STUDENT), because $o_8 \asymp o_4$ and I_s(PHD-STUDENT) $\subseteq I_s$(STUDENT) and I_s(BOOL) $\subseteq I_s$(BOOL). Similarly, id_5, id_6 are also included in I_d(PHD-STUDENT). However, id_4 is not included in the reuse interpretation of PHD-STUDENT, i.e., I_r(PHD-STUDENT), because no behavior in the interface of PHD-STUDENT is knowhow-identical to o_4. On the other hand, behavior id_5 is included in I_r(PHD-STUDENT) because $o_8 \equiv_{kh} o_5$ and I_s(PHD-STUDENT) $\subseteq I_s$(MSC-STUDENT) and I_s(BOOL) $\supseteq I_s$(BOOL). Similarly, id_6 is also included in I_r(PHD-STUDENT).

Similar explanations can also be given for the interpretations of the other two classes. ■

Example 17: Consider another very different example. Assume that $id_{11}, id_{12}, id_{15} \in I_s$(TEXT), $id_{13}, id_{16} \in I_s$(COURSES), $id_{14}, id_{17}, id_{15} \in I_s$(DEPARTMENT), and $id_{18} \in I_s$(INT). Let Θ be the following objects (including the objects identified by $id_{11}, ..., id_{18}$).

$o_1 = (id_1, \text{PERSON}, [\mathit{Name}{:}\ id_{11}])$
$o_2 = (id_2, \text{PROGRAM}, [\mathit{Name}{:}\ id_{12}, \mathit{Courses}{:}\ id_{13}, \mathit{Dept}{:}\ id_{14}])$
$o_3 = (id_3, \text{STUDENT}, [\mathit{Name}{:}\ id_{15}, \mathit{Courses}{:}\ id_{16}, \mathit{Dept}{:}\ id_{17}, \mathit{IDnumber}{:}\ id_{18}])$
$o_4 = (id_4, \text{TEXT}, \text{"What is your name?"}, (\text{PERSON}, () \longrightarrow \text{TEXT}, kh_4))$
$o_5 = (id_5, \text{TEXT}, \text{"Program name"}, (\text{PROGRAM}, () \longrightarrow \text{TEXT}, kh_5))$
$o_6 = (id_6, \text{TEXT}, \text{"Include a new course offering"},$
$\quad (\text{PROGRAM}, (\text{Course}, \text{COURSE}) \longrightarrow \text{COURSES}, kh_6))$
$o_7 = (id_7, \text{TEXT}, \text{"Delete this course offering"},$
$\quad (\text{PROGRAM}, (\text{Course}, \text{COURSE}) \longrightarrow \text{COURSES}, kh_7))$
$o_8 = (id_8, \text{TEXT}, \text{"What is your name?"}, (\text{STUDENT}, () \longrightarrow \text{TEXT}, kh_{o_4}))$
$o_9 = (id_9, \text{TEXT}, \text{"Add a course"},$
$(\text{STUDENT}, (\text{Course}, \text{COURSE}) \longrightarrow \text{COURSES}, kh_{o_6}))$
$o_{10} = (id_{10}, \text{TEXT}, \text{"Drop a course"},$
$(\text{STUDENT}, (\text{Course}, \text{COURSE}) \longrightarrow \text{COURSES}, kh_{o_7}))$
where (representations)

$\dot{\Xi}$(PERSON) = [Name: TEXT]
$\dot{\Xi}$(PROGRAM) = [Name: TEXT, Courses: COURSES, Dept: DEPARTMENT]
$\dot{\Xi}$(STUDENT) = [Name: TEXT, Courses: COURSES, Dept: DEPARTMENT,

IDnumber: INT]

and (interfaces)

$\dot{\Sigma}(\text{PERSON}) = \{o_4\}$
$\dot{\Sigma}(\text{PROGRAM}) = \{o_5, o_6, o_7\}$
$\dot{\Sigma}(\text{STUDENT}) = \{o_8, o_9, o_{10}\}$

Here suppose class STUDENT is a reuse of class PROGRAM. Notice that o_8, o_9 and o_{10} each have a borrowed knowhow reused from a behavior of PROGRAM; this is allowable because STUDENT is a reuse of PROGRAM. Intuitively, instances of PROGRAM are like school-promoting brochures that describe the courses offered in a particular program by some department. Examples of the program Names are "MBA Program", "French as Second Language" and "Advanced Ballroom Dance". The following table gives the static, dynamic and reuse interpretations for classes PERSON, PROGRAM and STUDENT defined on Θ.

class: c	$I_s(c)$	$I_d(c)$	$I_r(c)$
PERSON	$\{id_1, id_2, id_3\}$	$\{id_4, id_8\}$	$\{id_4\}$
PROGRAM	$\{id_2, id_3\}$	$\{id_5, id_6, id_7\}$	$\{id_5, id_6, id_7\}$
STUDENT	$\{id_3\}$	$\{id_4, id_8, id_9, id_{10}\}$	$\{id_5, id_6, id_7, id_8, id_9, id_{10}\}$

Some of the above interpretations deserve further explanations. Object id_3 (an instance of STUDENT) is included in the static interpretation of PROGRAM, i.e., I_s(PROGRAM), because object id_3 contains all the attributes in the representation of PROGRAM and because, under our assumptions, $id_{15} \in I_s$(TEXT) and $id_{17} \in I_s$(DEPARTMENT). Object id_4 (a behavior of PERSON) is included in the dynamic interpretation of STUDENT, i.e., I_d(STUDENT), because $o_4 \times o_8$ and I_s(TEXT) $\subseteq I_s$(TEXT) (note that neither o_8 nor o_4 takes any labeled input arguments). Object id_6 (a behavior of PROGRAM) is included in the reuse interpretation of STUDENT, i.e., I_r(STUDENT), because $o_6 \equiv_{kh} o_9$ and I_s(COURSE) $\subseteq$ I_s(COURSE) and I_s(COURSES) $\supseteq I_s$(COURSES). ■

5.3 Orderings on KBO Classes

We now introduce two important orderings on KBO classes, *k-compatibility* ("kind-of" compatibility) and *r-compatibility* ("reuse-of" compatibility). The k-compatibility relationship will allow us to say that a user-prescribed KIND-OF relationship is *valid*

(i.e., acceptable by the database system). Similarly, the r-compatibility relationship will allow us to say that a user-prescribed REUSE-OF relationship is *valid*. K-compatibility is defined as the inclusion of static interpretations and dynamic interpretations, while r-compatibility is defined as the inclusion of reuse interpretations.

5.3.1 K-Compatibility and R-Compatibility

In order to define k-compatibility, we shall first define the notions of static compatibility and dynamic compatibility:

Definition 35: *Static Compatibility.* Let c and c' be two classes in C. We say that c is *statically compatible* with c' (denoted by $c \preceq_s c'$) if and only if $I_s(c) \subseteq I_s(c')$ for all consistent sets of KBO objects. ■

Definition 36: *Dynamic Compatibility.* Let c and c' be two classes in C. We say that c is *dynamically compatible* with c' (denoted by $c \preceq_d c'$) if and only if $I_d(c) \supseteq I_d(c')$, for all consistent sets of KBO objects. ■Intuitively, (1) if class c is statically compatible with class c' then the representation of c can "simulate" the representation of c'; and (2) if class c is dynamically compatible with class c' then the interface of c can "simulate" the interface of c'. It must be noticed that this does not imply that c and c' have to *physically* share the same representation and possess the same knowhows.

We are now ready to define k-compatibility:

Definition 37: *k-Compatibility.* Let c and c' be two classes in C. We say that c is *k-compatible* with c' (denoted by $c \preceq_k c'$) if and only if $c \preceq_s c'$ and $c \preceq_d c'$. ■

Intuitively, the partial ordering $\preceq_k$ models *context inheritance*: if $c \preceq_k c'$ then any object of class c can be used in any context (e.g., the formal input/output specification of a behavior) expecting an object of class c'; in other words, c can "inherit" all the "contexts" of c'. Thus, k-compatibility can be used to validate the "principle of substitutability" [WZ90]: ***an instance of a subclass can always be used in any context in which an instance of a superclass was expected.***

Definition 38: *r-Compatibility.* Let c and c' be two classes in C. We say that c is *r-compatible* with c' (denoted by $c \preceq_r c'$) if and only if $I_r(c) \supseteq I_r(c')$ for all consistent sets of objects. ■

Intuitively, the partial order $\preceq_r$ models *knowhow inheritance*: if $c \preceq_r c'$ then any knowhow possessed by a behavior of class c' can be "properly reused" by a behavior of class c to exhibit the same knowhow for instances of c; in other words, c can "inherit" all the "knowhows" of c'. Thus, r-compatibility can be used to validate user-declared REUSE-OF relationships.

Example 18: Examining Example 16 and Example 17, we can see that the following inequalities hold (assume the set of objects in these examples is the only consistent set).

In Example 16, we have:

MSC-STUDENT $\preceq_k$ STUDENT	MSC-STUDENT $\not\preceq_r$ STUDENT
PHD-STUDENT $\preceq_k$ STUDENT	PHD-STUDENT $\not\preceq_r$ STUDENT
PHD-STUDENT $\preceq_k$ MSC-STUDENT	PHD-STUDENT $\preceq_r$ MSC-STUDENT

In Example 17, we have:

STUDENT $\preceq_k$ PERSON	STUDENT $\not\preceq_r$ PERSON
PROGRAM $\not\preceq_k$ PERSON	PROGRAM $\not\preceq_r$ PERSON
STUDENT $\not\preceq_k$ PROGRAM	STUDENT $\preceq_r$ PROGRAM

Some remarks must be made here. First, $c_1 \preceq_k c_2$ does not imply that a user has prescribed c_1 Ⓚ c_2. For example, in Example 16 we have PHD-STUDENT $\preceq_k$ MSC-STUDENT, but the user actually never declared that PHD-STUDENT is a subclass of MSC-STUDENT. On the other hand, $c_1 \preceq_r c_2$ *does* imply that a user has prescribed c_1 ⓡ c_2. For example, in Example 17 we have STUDENT $\preceq_r$ PROGRAM, and the user did declare that STUDENT is a demandclass of PROGRAM. However, declaring c_1 ⓡ c_2 does not imply $c_1 \preceq_r c_2$ because c_1 may have a behavior that reuses a knowhow from a c_2's behavior but with a new, incompatible signature (e.g., the reused knowhow needs a INT argument but the new signature specifies a TEXT argument). ■

5.3.2 Syntactic Validation of K-Compatibility

In the previous section, we have given a semantic definition for k-compatibility. We now provide a syntactic characterization for k-compatibility. Such a characterization consists of two theorems. The first one characterizes the syntactic properties of static compatibility:

Theorem 6: *Let Θ be any consistent set of KBO objects. Let c and c' be any two KBO classes defined on Θ. Then,*

1. $c \preceq_s$ OBJECT

2. BOTTOM $\preceq_s c$

3. Let c, c' be two capsule classes, $\dot{\Xi}(c) = D_i$ and $\dot{\Xi}(c') = D_j$.

$$c \preceq_s c' \Longleftrightarrow D_i = D_j$$

4. Let c, c' be two aggregate classes, $\dot{\Xi}(c) = [A_1: c_1, ..., A_n: c_n]$ and $\dot{\Xi}(c') = [B_1: c'_1, ..., B_m: c'_m]$.

$$c \preceq_s c' \iff \forall i \in [1..m], \exists j \in [1..n]: A_j = B_i \wedge c_j \preceq_s c'_i$$

5. Let c, c' be two bi-object classes, $\dot{\Xi}(c) = \{\!\{ c_1, c_2, c_3 \}\!\}$ and $\dot{\Xi}(c') = \{\!\{ c'_1, c'_2, c'_3 \}\!\}$.

$$c \preceq_s c' \iff \forall i \in [1..3]: c_i \preceq_s c'_j$$

6. Let c, c' be two set classes, $\dot{\Xi}(c) = \{c_1\}$ and $\dot{\Xi}(c') = \{c'_1\}$.

$$c \preceq_s c' \iff c_1 \preceq_s c'_1$$

7. Let c, c' be two sequence classes, $\dot{\Xi}(c) = <c_1>$ and $\dot{\Xi}(c') = <c'_1>$.

$$c \preceq_s c' \iff c_1 \preceq_s c'_1$$

Proof: The first two statements are trivially true by the definition of static compatibility. In the following, we give the proof for the rest of the statements. We first prove the $\Leftarrow$ part by enumerating the structures of the representations of classes.

- (Capsule Classes)
  Let $\dot{\Xi}(c) = D_i$ and $\dot{\Xi}(c') = D_j$. If $D_i = D_j$ then, by definition of static interpretations, $I_s(c) = I_s(c')$, which implies $c \preceq_s c'$.

- (Aggregate Classes)
  Let $\dot{\Xi}(c) = [A_1: c_1, ..., A_n: c_n]$ and $\dot{\Xi}(c') = [B_1: c'_1, ..., B_m: c'_m]$, where $n \geq m$. Suppose $\forall i \in [1..m], \exists j \in [1..n]: B_i = A_j \wedge c_j \preceq_s c'_i$. Without loss in generality, we can let $\dot{\Xi}(c) = [B_1: c_1, ..., B_m: c_m, A''_{m+1}: c_{m+1}, ..., A''_n: c_n]$, where $c_i \preceq_s c'_i$ for $1 \leq i \leq m$. However, by definition of static interpretations, any (non-bottom) object $oid(o)$ in $I_s(c)$ must have a value
  $[B_1: id_1, ..., B_m: id_m, A''_{m+1}: id_{m+1}, ..., A''_n: id_n, L_{n+1}: id_{n+1}, ..., L_{n+k}: id_{n+k}]$
  where $k \geq 0$ and $id_i \in I_s(c_i)$ for $1 \leq i \leq n$. Because $c_i \preceq_s c'_i$ for $1 \leq i \leq m$, we know that $I_s(c_i) \subseteq I_s(c'_i)$ for $1 \leq i \leq m$. Therefore, it is also true to say that o has a value
  $[B_1: id_1, ..., B_m: id_m, A''_{m+1}: id_{m+1}, ..., A''_n: id_n, L_{n+1}: id_{n+1}, ..., L_{n+k}: id_{n+k}]$
  where $id_i \in I_s(c'_i)$ for $1 \leq i \leq m$. This implies that object $oid(o)$ is also contained in $I_s(c')$. Hence, we have $I_s(c) \subseteq I_s(c')$, which means $c \preceq_s c'$.

- (Bi-object Classes)
  Let $\dot{\Xi}(c) = \{\!\!\{ c_1, c_2, c_3 \}\!\!\}$ and $\dot{\Xi}(c') = \{\!\!\{ c'_1, c'_2, c'_3 \}\!\!\}$. By definition of static interpretations, any object $oid(o)$ in $I_s(c)$ has a value $\{\!\!\{ id_1, id_2, id_3 \}\!\!\}$ where $id_1 \in I_s(c_1)$, $id_2 \in I_s(c_2)$ and $id_3 \in I_s(c_3)$. Suppose $c_1 \preceq_s c'_1$, $c_2 \preceq_s c'_2$ and $c_3 \preceq_s c'_3$. This implies that $I_s(c_1) \subseteq I_s(c'_1)$, $I_s(c_2) \subseteq I_s(c'_2)$, and $I_s(c_3) \subseteq I_s(c'_3)$. Therefore, it is also true that $id_1 \in I_s(c'_1)$, $id_2 \in I_s(c'_2)$ and $id_3 \in I_s(c'_3)$, which implies that $oid(o)$ is also in $I_s(c')$. Hence, we have $I_s(c) \subseteq I_s(c')$, which means $c \preceq_s c'$.

- (Set Classes)
  Let $\dot{\Xi}(c) = \{c_1\}$ and $\dot{\Xi}(c') = \{c'_1\}$. By definition of static interpretations, any object $oid(o)$ in $I_s(c)$ has a value $\{id_1, ..., id_n\}$ where $id_i \in I_s(c_1)$ for $0 \leq i \leq n$. If $c_1 \preceq_s c'_1$, then $I_s(c_1) \subseteq I_s(c'_1)$, and thus $id_i \in I_s(c'_1)$ for $1 \leq i \leq n$. This implies that $oid(o)$ is also in $I_s(c')$. Hence, we have $I_s(c) \subseteq I_s(c')$, which means $c \preceq_s c'$.

- (Sequence Classes)
  Let $\dot{\Xi}(c) = <c_1>$ and $\dot{\Xi}(c') = <c'_1>$. By definition of static interpretations, any object $oid(o)$ in $I_s(c)$ has a value $< id_1, ..., id_n >$ where $id_i \in I_s(c_1)$ for $0 \leq i \leq n$. If $c_1 \preceq_s c'_1$, then $I_s(c_1) \subseteq I_s(c'_1)$, and thus $id_i \in I_s(c'_1)$ for $1 \leq i \leq n$. This implies that $oid(o)$ is also in $I_s(c')$. Hence, we have $I_s(c) \subseteq I_s(c')$, which means $c \preceq_s c'$.

We now prove the $\Longrightarrow$ part by a case study:

- (Capsule Classes)
  Let $\dot{\Xi}(c) = D_i$ and $\dot{\Xi}(c') = D_j$, and assume $c \preceq_s c'$. By definition of static interpretations, for any object o if $oid(o) \in I_s(c)$ then $value(o) \in dom(D_i)$. Because $I_s(c) \subseteq I_s(c')$ holds from $c \preceq_s c'$, we also have $value(o) \in dom(D_j)$. Under our assumption (in Chapter 3) that all domains of base classes are pairwise disjoint, it is immediate that $D_i = D_j$.

- (Aggregate Classes)
  Let $\dot{\Xi}(c) = [A_1\colon c_1, ..., A_n\colon c_n]$ and $\dot{\Xi}(c') = [B_1\colon c'_1, ..., B_m\colon c'_m]$ where $n \geq m$, and assume $c \preceq_s c'$. Because $c \preceq_s c'$ holds, we know that $I_s(c) \subseteq I_s(c')$ holds for any consistent set of KBO objects.

  Let us first assume that there is a label B_i $(1 \leq i \leq m)$ in $\dot{\Xi}(c')$ such that there no label A_j in $\dot{\Xi}(c)$ which is the same label as B_i. Then we can introduce a

new set Θ' of KBO objects, such that $\Theta' = \Theta \cup \{o\}$ where o is a new object (i.e., with a new object-identity) with $value(o) = [A_1\colon \mathbf{null}, ..., A_n\colon \mathbf{null}]$ and $kh(o) = \phi_\perp$. Obviously Θ' is still a finite set of KBO objects, with no dangling object-identities and with no dangling borrowed knowhows. Neither is there any new inherent knowhow in Θ'. Therefore, Θ' is indeed a consistent set of KBO objects. By definition of static interpretations on Θ', we have $oid(o) \in I_s(c)$ but $oid(o) \notin I_s(c')$ because B_i is not in $\dot{\Xi}(c')$. Hence, $I_s(c) \subseteq I_s(c')$ does not hold for the consistent set Θ' of KBO objects, which is a contradiction.

Now let us assume that for a label B_i $(1 \leq i \leq m)$ in $\dot{\Xi}(c')$ there is a label A_j in $\dot{\Xi}(c)$ which is the same label as B_i, but $I_s(c_j) \not\subseteq I_s(c'_i)$. This implies that in Θ there exists an object x such that $oid(x) \in I_s(c_j)$ but $oid(x) \notin I_s(c'_i)$. Again, we can introduce a new set Θ' of KBO objects, such that $\Theta' = \Theta \cup \{o\}$ where o is a new object with $value(o) = [A_1\colon \mathbf{null}, ..., A_j\colon oid(x), ..., A_n\colon \mathbf{null}]$ and $kh(o) = \phi_\perp$. Then Θ' is still a finite set of KBO objects, with no dangling object-identities and with no dangling borrowed knowhows. Therefore, Θ' is indeed a consistent set of KBO objects. By definition of static interpretations on Θ', we have $oid(o) \in I_s(c)$ but $oid(o) \notin I_s(c')$ because $oid(x) \notin I_s(c'_i)$. Therefore, $I_s(c) \subseteq I_s(c')$ does not hold for the consistent set Θ' of KBO objects, which is a contradiction to the assumption $c \preceq_s c'$.

- (Bi-object Classes)
  Let $\dot{\Xi}(c) = \langle\!\langle c_1, c_2, c_3 \rangle\!\rangle$ and $\dot{\Xi}(c') = \langle\!\langle c'_1, c'_2, c'_3 \rangle\!\rangle$, and assume $c \preceq_s c'$. Suppose there exists an object-identity $id \in I_s(c_1)$ such that $id \notin I_s(c'_1)$. Then we can introduce a new set Θ' such that $\Theta' = \Theta \cup \{o\}$ where o is a new object with $value(o) = \langle\!\langle id, \mathbf{null}, \mathbf{null} \rangle\!\rangle$ and $kh(o) = \phi_\perp$. Because Θ' is still a finite set with no dangling object-identities and with no dangling borrowed knowhows, Θ' is indeed a consistent set of KBO objects. However, by the definition of static interpretations on Θ', we have $oid(o) \in I_s(c)$ but $oid(o) \notin I_s(c')$ because $id \notin I_s(c'_1)$. Hence, $I_s(c) \subseteq I_s(c')$ does not hold for the consistent set Θ' of KBO objects, which is a contradiction. Similarly, one can show that $c_2 \preceq_s c'_2$ and $c_3 \preceq_s c'_3$ must hold too.

- (Set Classes)
  Let $\dot{\Xi}(c) = \{c_1\}$ and $\dot{\Xi}(c') = \{c'_1\}$, and assume $c \preceq_s c'$. Since $c \preceq_s c'$ holds, we know that $I_s(c) \subseteq I_s(c')$ holds not only for Θ but for any consistent set of KBO objects. Now, suppose there exists an object-identity $id \in I_s(c_1)$ such that $id \notin I_s(c'_1)$. Then we can introduce a new set Θ' of KBO objects, such

that $\Theta' = \Theta \cup \{o\}$ where o is a new object (i.e., with a new object-identity) with $value(o) = \{id\}$ and $kh(o) = \phi_\perp$. Because Θ' is still a finite set with no dangling object-identities and with no dangling borrowed knowhows, Θ' is indeed a consistent set of KBO objects. However, on Θ' we have $oid(o) \in I_s(c)$ but $oid(o) \notin I_s(c')$ from the definition of static interpretations. Hence, $I_s(c) \subseteq I_s(c')$ does not hold for the consistent set Θ' of KBO objects, which is a contradiction.

- (Sequence Classes)
  Let $\overset{\bullet}{\Xi}(c) = <c_1>$ and $\overset{\bullet}{\Xi}(c') = <c'_1>$, and assume $c \preceq_s c'$. Suppose there exists an object-identity $id \in I_s(c_1)$ such that $id \notin I_s(c'_1)$. Then we can introduce a new set Θ' of KBO objects, such that $\Theta' = \Theta \cup \{o\}$ where o is a new object with $value(o) = <id>$ and $kh(o) = \phi_\perp$. Again, Θ' is still a consistent set of KBO objects. However, by the definition of static interpretations on Θ', we have $oid(o) \in I_s(c)$ but $oid(o) \notin I_s(c')$ because $id \notin I_s(c'_1)$. Hence, $I_s(c) \subseteq I_s(c')$ does not hold for the consistent set Θ' of KBO objects, which is a contradiction.

Our second theorem characterizes the syntactic properties of dynamic compatibility.

Theorem 7: *Let Θ be any consistent set of KBO objects. Let c and c' be any two KBO classes defined on Θ. Then,*

$$
\begin{aligned}
c \preceq_d c' \iff \forall b' \in \overset{\circ}{\Sigma}(c'), \exists b \in \overset{\circ}{\Sigma}(c): b \asymp b' \\
&\wedge\ sig(b) = (A_1, c_1) \times \ldots \times (A_n, c_n) \longrightarrow c_{out} \\
&\wedge\ sig(b') = (A_1, c'_1) \times \ldots \times (A_n, c'_n) \longrightarrow c'_{out} \\
&\wedge\ \forall i \in [1..n]: c_i \preceq_s c'_i \\
&\wedge\ \hat{c}_{out} \preceq_s \hat{c}'_{out}
\end{aligned}
$$

where $\hat{c}_{out} = c$ if $c_{out} =$ SELF, otherwise $\hat{c}_{out} = c_{out}$, and $\hat{c}'_{out} = c'$ if $c'_{out} =$ SELF, otherwise $\hat{c}'_{out} = c'_{out}$.

Proof:

- ($\Longrightarrow$) : By definition of $\preceq_d$ relation, $c \preceq_d c'$ means that $I_d(c) \supseteq I_d(c')$ for any consistent set of KBO objects. Let o be any KBO object in Θ with signature

$$sig(o) = (A_1, c'_1) \times \ldots \times (A_n, c'_n) \longrightarrow c'_{out}$$

  By the definition of dynamic interpretations, $oid(o) \in I_d(c')$ if and only if there exists an object $b \in \overset{\circ}{\Sigma}(c')$ such that $o \asymp b$ and

$$sig(b) = (A_1, c_1) \times \ldots \times (A_n, c_n) \longrightarrow c_{out}$$

where $I_s(c_i) \subseteq I_s(c'_i)$ for $1 \leq i \leq n$ and $I_s(\hat{c}_{out}) \subseteq I_s(\hat{c}'_{out})$. Here, $\hat{c}_{out} = owner(b)$ if c_{out} = SELF, otherwise $\hat{c}_{out} = c_{out}$, and $\hat{c}'_{out} = owner(o)$ if c'_{out} = SELF, otherwise $\hat{c}'_{out} = c'_{out}$. Now, since every behavior in $\mathring{\Sigma}(c')$ trivially satisfies the above condition, we know that $\mathring{\Sigma}(c') \subseteq I_d(c')$ holds. Because $I_d(c) \supseteq I_d(c')$, we have $\mathring{\Sigma}(c') \subseteq I_d(c)$, which implies that for every $b' \in \mathring{\Sigma}(c')$,

$$sig(b') = (A_1, c'_1) \times \ldots \times (A_n, c'_n) \longrightarrow c'_{out}$$

there must be an object $b \in \mathring{\Sigma}(c)$ such that $b \asymp b'$ and

$$sig(b) = (A_1, c_1) \times \ldots \times (A_n, c_n) \longrightarrow c_{out}$$

where $I_s(c_i) \subseteq I_s(c'_i)$ for $1 \leq i \leq n$ and $I_s(\hat{c}_{out}) \subseteq I_s(\hat{c}'_{out})$. Here, because $owner(b) = c$ and $owner(b') = c'$, we can derive that $\hat{c}_{out} = c$ if c_{out} = SELF, otherwise $\hat{c}_{out} = c_{out}$, and $\hat{c}'_{out} = c'$ if c'_{out} = SELF, otherwise $\hat{c}'_{out} = c'_{out}$. Therefore, the right-hand side of the theorem holds.

- ($\Longleftarrow$) : Assume that for every $b' \in \mathring{\Sigma}(c')$,

$$sig(b') = (A_1, c'_1) \times \ldots \times (A_n, c'_n) \longrightarrow c'_{out}$$

there is $b \in \mathring{\Sigma}(c)$ such that $b \asymp b'$ and

$$sig(b) = (A_1, c_1) \times \ldots \times (A_n, c_n) \longrightarrow c_{out}$$

where $c_i \preceq_s c'_i$ for $1 \leq i \leq n$ and $\hat{c}_{out} \preceq_s \hat{c}'_{out}$. Now, suppose $c \preceq_d c'$ is not true. That is, $I_d(c) \not\supseteq I_d(c')$. This implies that there is an object-identity $oid(o)$ in $I_d(c')$ which is not in $I_d(c)$. Let

$$sig(o) = (A_1, c''_1) \times \ldots \times (A_n, c''_n) \longrightarrow c''_{out}$$

Then, $oid(o) \in I_d(c')$ implies that there exists a behavior $b' \in \mathring{\Sigma}(c')$ such that $b' \asymp o$ and

$$sig(b') = (A_1, c'_1) \times \ldots \times (A_n, c'_n) \longrightarrow c'_{out}$$

where $I_s(c'_i) \subseteq I_s(c''_i)$ for $1 \leq i \leq n$ and $I_s(\hat{c}'_{out}) \subseteq I_s(\hat{c}''_{out})$. Here $\hat{c}'_{out} = owner(b')$ if c'_{out} = SELF, otherwise $\hat{c}_{out} = c'_{out}$, and $\hat{c}''_{out} = owner(o)$ if c''_{out} = SELF, otherwise $\hat{c}''_{out} = c''_{out}$. On the other hand, $oid(o) \notin I_d(c)$ implies that there exists no a behavior $b \in \mathring{\Sigma}(c)$ such that $b \asymp o$ and

$$sig(b) = (A_1, c_1) \times \ldots \times (A_n, c_n) \longrightarrow c_{out}$$

where $I_s(c_i) \subseteq I_s(c''_i)$ for $1 \leq i \leq n$ and $I_s(\hat{c}_{out}) \subseteq I_s(\hat{c}''_{out})$. Here $\hat{c}_{out} = owner(b)$ if c_{out} = SELF, otherwise $\hat{c}_{out} = c_{out}$, and $\hat{c}''_{out} = owner(o)$ if c''_{out} = SELF, otherwise $\hat{c}''_{out} = c''_{out}$. Therefore, there cannot be any behavior $b \in \mathring{\Sigma}(c)$ such that $b \asymp b'$ and

$$sig(b) = (A_1, c_1) \times \ldots \times (A_n, c_n) \longrightarrow c_{out}$$

where $I_s(c_i) \subseteq I_s(c'_i)$ for $1 \leq i \leq n$ and $I_s(\hat{c}_{out}) \subseteq I_s(\hat{c}'_{out})$. Here $\hat{c}_{out} = owner(b) = c$ if c_{out} = SELF, otherwise $\hat{c}_{out} = c_{out}$, and $\hat{c}'_{out} = owner(b') = c'$

if c'_{out} = SELF, otherwise $\check{c}'_{out} = c'_{out}$. This is a contradiction of our initial assumption.

■

From the above theorem, the following special property is straightforward.

Theorem 8: *Let Θ be any consistent set of KBO objects. Let c and c' be any two KBO classes defined on Θ. Then,*

$$\mathring{\Sigma}(c') \subseteq \mathring{\Sigma}(c) \Longrightarrow c \preceq_d c'$$

Proof: The assertion $\mathring{\Sigma}(c') \subseteq \mathring{\Sigma}(c)$ implies that for every b' in $\mathring{\Sigma}(c')$ there is a b in $\mathring{\Sigma}(c)$ such that $b \equiv b'$. This immediately implies the right-hand side of the above theorem, and therefore also implies $c \preceq_d c'$. ■

The above theorems provide a syntactic means for checking the k-compatibility between two user-declared KBO classes. More precisely, we can check the static compatibility based on the first theorem and dynamic compatibility based on the second theorem. The third theorem can be used to deal with a special case where the dynamic compatibility between two classes can be immediately decided.

5.3.3 Syntactic Validation of R-Compatibility

In the following, we develop a theorem which is a syntactic characterization for r-compatibility. Intuitively, the theorem is intended to provide a syntactic means for validating whether a new class c is r-compatible with an existing class c', under the assumption that c' itself was already validated to be r-compatible with all its supplyclasses. This assumption actually reflects the fact that we create new classes *incrementally* from existing, compatibly connected classes. In other words, we may not be able to decide whether c is r-compatible with c', if c' itself is not r-compatible with its own supplyclasses.

Theorem 9: *Let Θ be any consistent set of KBO objects. Let c and c' are two KBO classes defined on Θ. If c' is r-compatible with all its supplyclasses defined on Θ, then*

$c \preceq_r c' \Longleftrightarrow$

- $c \preceq_s c'$ and
- $\forall x \in \dot{\Sigma}(c'), \exists y \in \dot{\Sigma}(c): y \equiv_{kh} x$ and
- $\forall x \in \dot{\Sigma}(c'), \forall y \in \Sigma(c): (y \asymp x \vee y \equiv_{kh} x)$
  $\wedge\ sig(y) = (A_1, c_1) \times \ldots \times (A_n, c_n) \longrightarrow c_{out}$

$$\Longrightarrow sig(x) = (A_1, c'_1) \times ... \times (A_n, c'_n) \longrightarrow c'_{out}$$
$$\wedge \forall i \in [1..n]: c_i \preceq_s c'_i \wedge \tilde{c}'_{out} \preceq_s \hat{c}_{out}$$

where $\hat{c}_{out} = c$ if c_{out} = SELF, otherwise $\hat{c}_{out} = c_{out}$, and $\tilde{c}'_{out} = c'$ if c'_{out} = SELF, otherwise $\tilde{c}'_{out} = c'_{out}$.

Proof:

- ($\Longrightarrow$) : Assume $c \preceq_r c'$, which means that $I_r(c) \supseteq I_r(c')$ for any consistent set of KBO objects. Let x be any KBO object in Θ with signature

  $$sig(x) = (A_1, c'_1) \times \ldots \times (A_n, c'_n) \longrightarrow c'_{out}$$

  By the definition of reuse interpretations, $oid(x) \in I_r(c')$ if and only if the following three conditions hold:

  1. $I_s(c') \subseteq I_s(owner(x))$.
  2. There exists a behavior y in $\dot{\Sigma}(c')$ such that $y \equiv_{kh} x$.
  3. If there is a behavior z in $\Sigma(c')$,

     $$sig(z) = (A_1, c_1) \times ... \times (A_n, c_n) \longrightarrow c_{out}$$

     such that $z \asymp x$ or $z \equiv_{kh} x$, then $I_s(c_i) \subseteq I_s(c'_i)$ for $1 \leq i \leq n$ and $I_s(\hat{c}_{out}) \supseteq I_s(\tilde{c}'_{out})$. Here, $\hat{c}_{out} = owner(z)$ if c_{out} = SELF, otherwise $\hat{c}_{out} = c_{out}$, and $\tilde{c}'_{out} = owner(x)$ if c'_{out} = SELF, otherwise $\tilde{c}'_{out} = c'_{out}$.

  Now we show that the right-hand side of the theorem holds in two cases:

  Case 1: Suppose all behaviors in $\dot{\Sigma}(c')$ are direct behaviors of class c'. That is, $\dot{\Sigma}(c') = \Sigma(c')$. This also implies $owner(x) = c'$ for every behavior x in $\dot{\Sigma}(c')$. Now, because all behaviors in $\dot{\Sigma}(c')$ trivially satisfy the above three conditions, all behaviors in $\dot{\Sigma}(c')$ are also in $I_r(c')$. In particular, they satisfy condition 3 simply because $\dot{\Sigma}(c') = \mathring{\Sigma}(c')$. Therefore, we have $\dot{\Sigma}(c') \subseteq I_r(c')$ holds. Since $I_r(c) \supseteq I_r(c')$, we have $\dot{\Sigma}(c') \subseteq I_r(c)$. From the definition of reuse interpretations, this implies that, for any behavior x in $\dot{\Sigma}(c')$ with signature

  $$sig(x) = (A_1, c'_1) \times \ldots \times (A_n, c'_n) \longrightarrow c'_{out}$$

  we have (1) $I_s(c) \subseteq I_s(owner(x))$, which is equivalent to $I_s(c) \subseteq I_s(c')$ because $owner(x) = c'$; (2) there exists a behavior y in $\dot{\Sigma}(c)$ such that $y \equiv_{kh} x$; and (3) if there is a behavior z in $\Sigma(c)$,

  $$sig(z) = (A_1, c_1) \times ... \times (A_n, c_n) \longrightarrow c_{out}$$

  such that $z \asymp x$ or $z \equiv_{kh} x$, then $I_s(c_i) \subseteq I_s(c'_i)$ for $1 \leq i \leq n$ and $I_s(\hat{c}_{out}) \supseteq I_s(\tilde{c}'_{out})$. Here, $\hat{c}_{out} = owner(z) = c$ if c_{out} = SELF, otherwise $\hat{c}_{out} = c_{out}$, and $\tilde{c}'_{out} = owner(x) = c'$ if c'_{out} = SELF, otherwise $\tilde{c}'_{out} = c'_{out}$. Therefore, it is immediate that the right-hand side of the theorem holds.

Case 2: Suppose some behaviors in $\overset{\bullet}{\Sigma}(c')$ are not direct behaviors of class c'. Let x be any one of these behaviors in $\overset{\bullet}{\Sigma}(c')$. Since x is not a direct behavior of c', it must be reused from another class $\bar{c}$, such that $owner(x) = \bar{c}$. Thus, there must be a REUSE-OF relationship between c' and $\bar{c}$ such that $c' \odot \bar{c}$. Since c' is r-compatible with any of its supplyclasses, we know that $c' \preceq_r \bar{c}$. Assume that all behaviors of class $\bar{c}$ are direct behaviors. Then from Case 1 above, we have the following: (1) $I_s(c') \subseteq I_s(\bar{c})$; (2) there is a behavior y in $\overset{\bullet}{\Sigma}(c')$ such that $y \equiv_{kh} x$; and (3) if there is a behavior z in $\Sigma(c')$,

$$sig(z) = (A_1, c_1) \times ... \times (A_n, c_n) \longrightarrow c_{out}$$

such that $z \asymp x$ or $z \equiv_{kh} x$, then

$$sig(x) = (A_1, c'_1) \times ... \times (A_n, c'_n) \longrightarrow c'_{out}$$

where $I_s(c_i) \subseteq I_s(c'_i)$ for $1 \leq i \leq n$ and $I_s(\hat{c}_{out}) \supseteq I_s(\tilde{c}'_{out})$. Here, $\hat{c}_{out} = owner(z) = c'$ if $c_{out} =$ SELF, otherwise $\hat{c}_{out} = c_{out}$, and $\tilde{c}'_{out} = owner(x) = \bar{c}$ if $c'_{out} =$ SELF, otherwise $\tilde{c}'_{out} = c'_{out}$. With conditions (1), (2) and (3), it follows that $oid(x) \in I_r(c')$. Now, because $I_r(c) \supseteq I_r(c')$ we thus have $x \in I_r(c)$, which in turn implies the following: (1) $I_s(c) \subseteq I_s(owner(x))$, that is, $I_s(c) \subseteq I_s(\bar{c})$; (2) there is a behavior y' in $\overset{\bullet}{\Sigma}(c)$ such that $y' \equiv_{kh} x$; and (3) if there is a behavior z' in $\Sigma(c)$,

$$sig(z') = (A_1, t'_1) \times ... \times (A_n, t'_n) \longrightarrow t'_{out}$$

such that $z' \asymp x$ or $z' \equiv_{kh} x$, then $I_s(t'_i) \subseteq I_s(c'_i)$ for $1 \leq i \leq n$ and $I_s(\hat{t}'_{out}) \supseteq I_s(\tilde{c}'_{out})$. Here, $\hat{t}'_{out} = owner(z') = c$ if $t'_{out} =$ SELF, otherwise $\hat{t}'_{out} = t'_{out}$. However, one can observe that from condition (1) we cannot derive that $I_s(c) \subseteq I_s(c')$ because $\bar{c} \neq c'$. Let us analyze two situations here. First, if c' has at least one direct behavior, then we can show that $I_s(c) \subseteq I_s(c')$ holds by the proof in Case 1 above. Second, if c' does not have any direct behaviors (i.e., all behaviors of c' are reused from other classes), then we can introduce a new set Θ' of KBO objects, such that $\Theta' = \Theta \cup \{o\}$ where o is a new object (i.e., with a new object-identity) which is shallow-identical to x (i.e., $o \asymp_1 x$), and has $owner(o) = c'$, $sig(o) = sig(x)$ and $kh(o) = kh(x)$. Two points must be noted: (a) the new object o is also included in $\overset{\bullet}{\Sigma}(c')$ because it is a direct behavior of c', and (b) x cannot be included in $\overset{\bullet}{\Sigma}(c')$ because its knowhow is borrowed by o in $\overset{\bullet}{\Sigma}(c')$. Also note that since o is only shallow-identical to x, there is only one new object (i.e., one new object-identity) that is added into Θ. Hence, Θ' is still a consistent set of KBO objects, and thus $I_r(c) \supseteq I_r(c')$ also holds for Θ'. Because o is a direct behavior of c', by the proof in Case 1 we can establish (1) $I_s(c) \subseteq I_s(c')$. Furthermore, we can see that (2) o now becomes the behavior

in $\overset{\bullet}{\Sigma}(c)$ such that $o \equiv_{kh} x$, and (3) $o \asymp x$ and $o \equiv_{kh} x$ and $sig(o) = sig(x)$. These three conditions show that the three conditions on the right-hand side of the theorem can still be satisfied by any indirect behavior of c'. Therefore, the right-hand side of the theorem holds.

Now, recall our assumption that all behaviors of class $\bar{c}$ are direct behaviors. If this is not true (i.e., $\bar{c}$ has some behaviors reused from other classes), then we can show that $c' \preceq_r \bar{c}$ implies the right-hand side of the theorem, by repeating the same process in Case 2. Because class OBJECT cannot reuse behaviors from any other classes, we can show, after a finite number of repetitions of the above process, that the right-hand side of the theorem holds.

- ($\Longleftarrow$) : Assume that the right-hand side of the theorem holds. That is, we have the following assertions:

  (P1) $I_s(c) \subseteq I_s(c')$.

  (P2) for every behavior x in $\overset{\bullet}{\Sigma}(c')$, there is a behavior y in $\overset{\bullet}{\Sigma}(c)$ such that $y \equiv_{kh} x$.

  (P3) for every behavior x in $\overset{\bullet}{\Sigma}(c')$, if there is a behavior z in $\Sigma(c)$,
  $$sig(z) = (A_1, c_1) \times ... \times (A_n, c_n) \longrightarrow c_{out}$$
  such that $z \asymp x$ or $z \equiv_{kh} x$, then
  $$sig(x) = (A_1, c'_1) \times ... \times (A_n, c'_n) \longrightarrow c'_{out}$$
  where $I_s(c_i) \subseteq I_s(c'_i)$ for $1 \leq i \leq n$ and $I_s(\hat{c}_{out}) \supseteq I_s(\hat{c}'_{out})$. Here, $\hat{c}_{out} = c$ if $c_{out} =$ SELF, otherwise $\hat{c}_{out} = c_{out}$, and $\hat{c}'_{out} = c'$ if $c'_{out} =$ SELF, otherwise $\hat{c}'_{out} = c'_{out}$.

Now, suppose the left-hand side of the theorem, i.e., $c \preceq_r c'$, is not true. That is, $I_r(c) \not\supseteq I_r(c')$. This implies that there is an object-identity $oid(x)$ in $I_r(c')$ but not in $I_r(c)$. Let
$$sig(x) = (A_1, c'_1) \times ... \times (A_n, c'_n) \longrightarrow c'_{out}$$
Knowing $oid(x) \in I_r(c')$, we can derive the following: (1) $I_s(c') \subseteq I_s(owner(x))$, and (2) there is a behavior y in $\overset{\bullet}{\Sigma}(c')$ such that $y \equiv_{kh} x$, and (3) if there is a behavior z in $\Sigma(c')$,
$$sig(z) = (A_1, c_1) \times ... \times (A_n, c_n) \longrightarrow c_{out}$$
such that $z \asymp x$ or $z \equiv_{kh} x$, then $I_s(c_i) \subseteq I_s(c'_i)$ for $1 \leq i \leq n$ and $I_s(\hat{c}_{out}) \supseteq I_s(\hat{c}'_{out})$. Here, $\hat{c}_{out} = owner(z)$ if $c_{out} =$ SELF, otherwise $\hat{c}_{out} = c_{out}$, and $\hat{c}'_{out} = owner(x)$ if $c'_{out} =$ SELF, otherwise $\hat{c}'_{out} = c'_{out}$.

On the other hand, knowing $oid(x) \notin I_r(c)$ we can derive the following: (a) $I_s(c) \not\subseteq I_s(owner(x))$, or (b) there is no behavior y' in $\dot{\Sigma}(c)$ such that $y' \equiv_{kh} x$, or (c) if there is a behavior z' in $\Sigma(c)$,

$$sig(z') = (A_1, t_1) \times \ldots \times (A_n, t_n) \longrightarrow t_{out}$$

such that $z' \asymp x$ or $z' \equiv_{kh} x$, then it is not true that $I_s(t_i) \subseteq I_s(c'_i)$ for $1 \leq i \leq n$ and $I_s(\hat{t}_{out}) \supseteq I_s(\hat{c}'_{out})$. Here, $\hat{t}_{out} = owner(z')$ if t_{out} = SELF, otherwise $\hat{t}_{out} = t_{out}$.

Now, if statement (a) is true and $owner(x) = c'$, then this is a contradiction to our assumption (P1). If statement (a) is true but $owner(x) = c''$ such that $c' \odot c''$, then statement (a) is again a contradiction to the fact that c' is r-compatible with any of its supplyclasses (i.e., $c' \odot c''$ implies $c' \preceq_r c''$, and by the "$\Longrightarrow$" part of our proof, $c \preceq_r c''$ in turn implies $I_s(c) \subseteq I_s(c'')$).

If statement (b) is true, then there will be no behavior in $\dot{\Sigma}(c)$ which can be knowhow-identical to behavior y, because $y \equiv_{kh} x$ from (1). This is thus a contradiction to our assumption (P2).

If statement (c) is true, then, for the behavior z in $\dot{\Sigma}(c')$, there will be no behavior z' in $\Sigma(c)$ such that if $z' \asymp z$ or $z' \equiv_{kh} z$ then $I_s(t_i) \subseteq I_s(c_i)$ for $1 \leq i \leq n$ and $I_s(t_{out}) \supseteq I_s(c_{out})$. This is a contradiction to our assumption (P3).

Therefore, if the right-hand side of the theorem holds, then $c \preceq_r c'$ must hold.

■

Three observations must be made about the differences between dynamic compatibility checking (Theorem 7) and r-compatibility checking (Theorem 9). First, if c is to be r-compatible with c', c must be statically compatible with c'. Intuitively, this implies that if c wishes to reuse the behaviors of c' then c should at least contain the representation required by the behaviors of c'. Dynamic compatibility does not have this condition. Second, Theorem 9 compares the signatures of two kh-identical or equal behaviors from each class, while Theorem 7 only compares the signatures of two equal behaviors from each class. Third, in Theorem 7 the result classes c_{out} and c'_{out} are compared in a way opposite to the way in which they are compared in Theorem 9. This property can be best explained by an example. Assume a class PERSON has the representation

$$\dot{\Xi}(\text{PERSON}) = [\text{Name: TEXT, Age: INT}]$$

and b_1 is a behavior of PERSON:

$$b_1 = (id_1, \text{TEXT}, \text{“}\textit{you next year}\text{”}, (\text{PERSON}, () \longrightarrow \text{PERSON}, kh_1))$$

so that, if *Lee* is a PERSON object with value [*Name*: "*Lee*", *Age*: 20], then the message-sending

$$\text{"}\textit{you next year}\text{"} \rightsquigarrow \textit{Lee}$$

returns a new PERSON object with value [*Name*: "*Lee*", *Age*: 21]. In other words, knowhow kh_1 knows how to construct a "future person" based on a current person's name and age. Now, assume another class EMPLOYEE has the representation

$\dot{\Xi}$(EMPLOYEE) = [Name: TEXT, Age: INT, Salary: INT].

To be dynamically compatible with PERSON, class EMPLOYEE can either (1) create a new behavior with value "you next year" or (2) reuse b_1 from PERSON.

- Consider option (1). Let b_2 be a behavior created for EMPLOYEE:

  $b_2 = (id_2, \text{TEXT}, \text{"}\textit{you next year}\text{"}, (\text{EMPLOYEE}, () \longrightarrow \text{EMPLOYEE}, kh_2))$

  so that, if *Mike* is an employee object with value

  [*Name*: "*Mike*", *Age*: 20, *Salary*: 20000]

  then the message-sending

  $$\text{"}\textit{you next year}\text{"} \rightsquigarrow \textit{Mike}$$

  returns a new employee object (i.e., with a new identity) with value

  [*Name*: "*Mike*", *Age*: 21, *Salary*: 22000]

  (notice the "future Mike" has a 10 percent increase in his salary). Obviously, kh_1 and kh_2 encapsulate quite different computations. The class EMPLOYEE is then dynamically compatible with PERSON because $value(b_1) = value(b_2)$ and their result classes are statically compatible, i.e., EMPLOYEE $\preceq_s$ PERSON (EMPLOYEE's representation includes PERSON's representation). Intuitively speaking, this means that, when message "you next year" is sent to a set of persons (some of them may be employees), the result will still be a set of persons. This will not be true if b_2 has a signature, say, () $\longrightarrow$ OBJECT, because OBJECT $\not\preceq_s$ PERSON.

- Consider option (2). If EMPLOYEE simply reuses b_1 from PERSON, then EMPLOYEE is obviously not only r-compatible with PERSON but also dynamically compatible with PERSON. Now, suppose EMPLOYEE only reuses the knowhow of b_1 in a behavior b_2 with a new signature:

  $b_2 = (id_2, \text{TEXT}, \text{"}\textit{you next year}\text{"}, (\text{EMPLOYEE}, () \longrightarrow \text{EMPLOYEE}, kh_{b_1}))$

  Then, although EMPLOYEE is still dynamically compatible with PERSON, EMPLOYEE is no longer r-compatible with PERSON because PERSON $\not\preceq_s$ EMPLOYEE. Intuitively, kh_{b_1} will only know how to produce a person object

based on the name and the age of an employee; in other words, its encapsulated code does not produce an employee object. Therefore, this is a safety violation because a result of the execution of b_2 cannot be used as an employee object (say, as the input of another behavior), as the signature of b_2 has indicated. In practice, this can cause memory violation and crash the system.

From Theorem 9, the following special property is straightforward.

Theorem 10: *Let Θ be any consistent set of KBO objects. Let c and c' be any two KBO classes defined on Θ. If c' is r-compatible with all its supplyclasses defined on Θ, then*

$$\mathring{\Sigma}(c') \subseteq \mathring{\Sigma}(c) \Longrightarrow c \preceq_r c'$$

Proof: The assertion $\mathring{\Sigma}(c') \subseteq \mathring{\Sigma}(c)$ states that for every b' in $\mathring{\Sigma}(c')$ there is a b in $\mathring{\Sigma}(c)$ such that $b \equiv b'$. This implies the right-hand side of Theorem 9, and therefore also implies $c \preceq_r c'$. ■

The above theorems provide a syntactic means for checking the r-compatibility between a new class c and an existing class c', provided that c' is r-compatible with its own supplyclasses. In practice, we can perform such checking each time a new user-defined class is created and connected into the existing hierarchies. In this way, we can always assume that existing classes in the database are r-compatible with their own supplyclasses. Note that used-defined classes are always r-compatible with kernel classes, whose interfaces are empty (i.e., they have no user-defined behaviors).

5.4 KBO Schemas

The previous theorems provide a syntactical means for validation of user-specified KIND-OF and REUSE-OF relationships in a KBO database schema, which is formally defined in the following:

Definition 39: *KBO Database Schemas.* A *KBO database schema* S is a triple $S = (\ \mathbf{C}, \textcircled{k}, \textcircled{r}\ )$, where $\mathbf{C}$ is a finite subset of $\mathcal{C}$, and $\textcircled{k}$, $\textcircled{r}$ are the KIND-OF and REUSE-OF binary relations on $\mathbf{C}$ satisfying the following conditions:

1. If $(c_1, c_2) \in \textcircled{k}$ then $c_1 \preceq_k c_2$.

2. If $(c_1, c_2) \in \textcircled{r}$ then $c_1 \preceq_r c_2$.

■

Visualiy, a KBO schema S can be represented by two acyclic directed graphs: (1) $(\mathbf{C}, \mathbf{K}_S)$ is an acyclic directed graph, with sharp-head edges pointing from parent

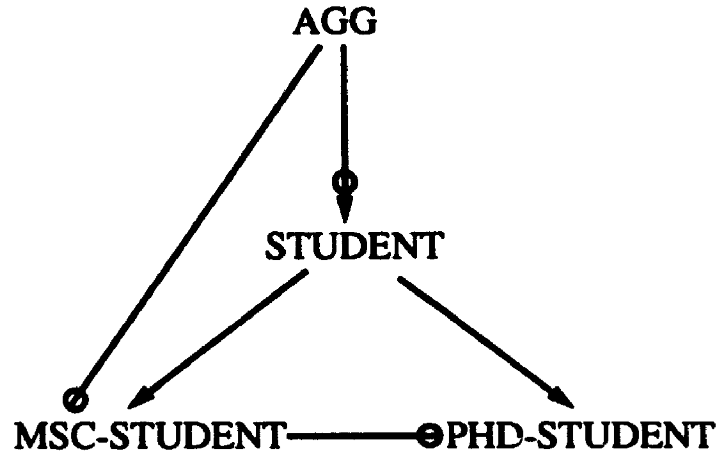


Figure 5.15: The DL-schema in Example 16

to child, such that if $< c_2, c_1 > \in \mathbf{K}_S$ then $c_1 \textcircled{k}_d\, c_2$; (2) $(\mathbf{C}, \mathbf{R}_S)$ is an acyclic directed graph, with round-head edges pointing from parent to child, such that if $< c_2, c_1 > \in \mathbf{R}_S$ then $c_1 \textcircled{r}_d\, c_2$. With such a representation, we shall call a KBO schema a *Dual-Lattice schema* or *DL-schema.*

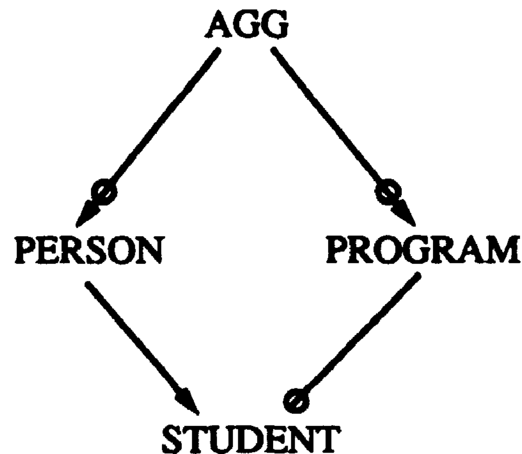
Example 19: Figure 5.15 shows the DL-schema described in Example 16. For the sake of simplicity, most kernel classes are omitted except AGG. This DL-schema is based on these prescriptive relationships: MSC-STUDENT $\textcircled{k}$ STUDENT, PHD-STUDENT $\textcircled{k}$ STUDENT, and PHD-STUDENT $\textcircled{r}$ MSC-STUDENT. Figure 5.16 shows the DL-schema described in Example 17. This DL-schema is based on these prescriptive relationships: STUDENT $\textcircled{k}$ PERSON and STUDENT $\textcircled{r}$ PROGRAM. In both DL-schemas, the prescriptive relationships can be validated to be compatible relationships (see Example 16 and 17). ■

Since two classes must be k-compatible if they have a subclass (KIND-OF) relationship between them, the signature of any behavior of the subclass, which replaces a behavior of the superclass, respects the KIND-OF hierarchy. For example, a class EVEN-INT may be a subclass of INT, where multiplication is coded in the knowhow of a behavior b_1 with signature

$$sig(b_1) = (\text{Int,INT}) \longrightarrow \text{INT}$$

Suppose class EVEN-INT is not a demandclass (reuse) of INT, and its multiplication is re-coded in the knowhow of a behavior b_2 with signature

$$sig(b_2) = (\text{Int,EVEN-INT}) \longrightarrow \text{EVEN-INT}$$



c1 ◄—— c2 = c1 KIND-OF c2 c1 ⊖—— c2 = c1 REUSE-OF c2

Figure 5.16: The DL-schema in Example 17

Thus, each class in this signature respects the KIND-OF (subclass) hierarchy. If class INT only has behavior b_1 and class EVEN-INT only has behavior b_2, then obviously EVEN-INT is k-compatible with INT.

Definition 40: *KBO Databases.* A *KBO database db* is a pair $db = (\ S, \Theta\ )$ where S is a KBO schema and Θ is a consistent set of KBO objects, such that

1. S contains the KBO kernel schema shown in Figure 3.13.

2. For all $o \in \Theta$, there exists a class c connected in S such that $o \; \textcircled{i}_d \; c$.

■

Note here, that condition 1 implies that all the constraints imposed by the kernel classes (at the end of Chapter 3) should be satisfied.

CHAPTER 6

SIMPLE AND COMPLEX MESSAGE-SENDINGS

6.1 Introduction

This chapter explores the idea of using ***sendable complex objects*** to model several common forms of behavior composition which may be manipulated regular data objects by query facilities. Because such complex objects can be viewed as a high-level semantics description of a complex behavior, end-users are able to manipulate the semantics of complex behaviors. In this way, the visibility of behavior semantics is broadened from "black-box" to "grey-box", in the sense that some high-level behavioral semantics can be captured in the forms of complex objects. A significant outcome of this approach is that it may open a brand new area for using query languages in OODBs — they can also be used as an associati ehavior chooser, generator and applier. With such a tool, OODB end-users can use complex objects, as much as possible, to construct simple queries, so that these queries become long-lived assets in the database, which may, in turn, be manipulated through the query facility to formulate new queries.

In this chapter, we first extend our less naive message-sending paradigm to support the execution of knowhows that take input arguments. We then further extend this new message-sending paradigm to include *complex message-sendings*, which model several simple but useful computation constructs. It should be noted that message-sendings are actually the simplest queries which can be issued; more complicated queries may be formulated using the standard operations offered by the KBO algebra described in Chapter 8.

6.2 Syntax of Message-Sendings

Definition 41: *Syntax of Message-Sendings.* A ***message-sending*** is an expression of the form:

$$\frac{Invoker}{Blackboard} \rightsquigarrow Receiver$$

where an object called *the Invoker* (the message) is sent to another object called *the Receiver.* The actual input arguments needed by the Invoker may be provided in an

argument container called *the Blackboard* which is a set of label-argument pairs of the form: L_1: $arg_1, \ldots, L_n$: arg_n, where $L_i \in \mathcal{A}, arg_i \in \mathcal{O}$ for $1 \leq i \leq n$ and $\forall i, j \in [1..n]$: $i \neq j \Longrightarrow L_i \neq L_j$. If no input arguments are supplied on the Blackboard, then the message-sending can be simply denoted by $Invoker \rightsquigarrow Receiver$. A message-sending always returns an object called *the Result*. ■

Example 20: The following is a message-sending:

$$\frac{\text{"Do you belong to this age group?"}}{\text{Age1:15, Age2:30}} \rightsquigarrow Mike \;\Rightarrow\; \textbf{True}$$

where the invoker is a TEXT object with value *"Do you belong to this age group?"*, the receiver is a PERSON object identified by *Mike*, and the blackboard shows two labeled arguments, 15 with label *Age*1 and 30 with label *Age*2. The result of this message-sending is an object with value **True**, meaning that *Mike* is at least 15 and at most 30. Notice that it is only for clarity that we denote capsule objects by their values (i.e., "Do you belong to this age group?", 15, 30, **True**), instead of their object-identities. ■

6.3 Semantics of Simple Message-Sendings

For the following definition we need some notation. We will use the symbol ℓ to denote a blackboard. We also let $args(\ell)$ denote the set $\{ (L_1, arg_1), \ldots, (L_n, arg_n) \}$ where $\ell = L_1$: $arg_1, \ldots, L_n$: arg_n. For example, if ℓ is the blackboard in the above message-sending, then $args(\ell) = \{(Age1, 15), (Age2, 30)\}$. Now, we are ready to define the operational semantics of simple message-sendings.

Definition 42: *Semantics of Simple Message-Sendings.* Let ℓ be a blackboard and let s, o be any two objects in $\mathcal{O}$.

1. $\frac{o}{\ell} \rightsquigarrow \textbf{\textit{fail}} \triangleq \textbf{\textit{fail}}$ $\qquad\qquad$ $\frac{\textbf{\textit{fail}}}{\ell} \rightsquigarrow o \triangleq \textbf{\textit{fail}}$
2. $\frac{o}{\ell} \rightsquigarrow \textbf{\textit{null}} \triangleq \textbf{\textit{null}}$ $\qquad\qquad$ $\frac{\textbf{\textit{null}}}{\ell} \rightsquigarrow o \triangleq \textbf{\textit{null}}$
   (when $o \neq \textbf{\textit{fail}}$)
3. $\frac{o}{\ell} \rightsquigarrow \textbf{\textit{same}} \triangleq \textbf{\textit{same}}$ $\qquad\qquad$ $\frac{\textbf{\textit{same}}}{\ell} \rightsquigarrow o \triangleq o$
   (when $o \neq \textbf{\textit{fail}}$ and $o \neq \textbf{\textit{null}}$)
4. If there exists $x \in \overset{*}{\Sigma}(class(o))$ such that $x \equiv s$ and $sig(x) = (A_1, c_1) \times \ldots \times (A_n, c_n) \longrightarrow c_r$ and $\{(A_1, a_1), \ldots, (A_n, a_n)\} \subseteq args(\ell)$ and a_i ⓘ c_i for $1 \leq i \leq n$, then

$$\frac{s}{\ell} \rightsquigarrow o \triangleq o.\overline{kh(x)}(A_1\text{: } a_1, \ldots, A_n\text{: } a_n)$$

Otherwise,

5. If there exists $x \in \mathring{\Sigma}(class(o))$ such that $x \asymp s$ and $sig(x) = (A_1, c_1) \times \ldots \times (A_n, c_n) \longrightarrow c_r$ and $\{(A_1, a_1), \ldots, (A_n, a_n)\} \subseteq args(\ell)$ and a_i Ⓘ c_i for $1 \leq i \leq n$, then

$$\frac{s}{\ell} \rightsquigarrow o \triangleq o.\overline{kh(x)}(A_1\!: a_1, \ldots, A_n\!: a_n)$$

Otherwise,

6. If s Ⓘ CAPSULE, then $\frac{s}{\ell} \rightsquigarrow o \triangleq$ ***fail***; otherwise $\frac{s}{\ell} \rightsquigarrow o$ is a *complex message-sending.*

We should point out that the bottom object ***same*** is introduced mainly because it is a very useful invoker when used in a complex invoker (defined shortly) since it returns the receiver itself. For completeness, we also defined the case where it is used as a receiver.

The above definition of simple message-sendings is similar to that of the less naive message-sending paradigm: when a message-sending $\frac{s}{\ell} \rightsquigarrow o$ occurs, the receiver o first searches its class's *interface* for a behavior which is *identical* to the invoker s and, if successful, exhibits the knowhow of that behavior (i.e., executes the knowhow). If the receiver o fails to find such a behavior, the receiver o then searches its class's *value-based interface* for a behavior which is *equal* to the invoker s and, if successful, exhibits the knowhow of that behavior. This definition, however, is more complete in the sense that: (1) it defines the special semantics for ***fail***, ***null*** and ***same*** in message-sendings; (2) it allows the execution of knowhows that take input arguments and enforces type-checking on these arguments; (3) it introduces the notion of a "complex message-sending" whose operational semantics is defined in the next section. It must be noted that when s is identical or equal to a behavior in the interface of o's class, s can still be a complex object (i.e., an AGG, BIO, SET or SEQ object); in this situation, the message-sending is still a simple message-sending, not a complex message-sending. A complex message-sending takes place only when s is a complex object which does not satisfy the conditions specified in Case 4 and 5 in the above definition. In other words, s can be either an annotated behavior or a complex invoker, depending on the receiver o.

6.4 Semantics of Complex Message-Sendings

The definition of complex message-sendings will formalize our previous intuitive discussion about aggregations in behavioral modeling in Chapter 3. The notion of complex message-sendings provides an opportunity for us to investigate how query languages can be used to manipulate the structures of "sendable complex objects", so that they can be sent to a receiver to cause a group of executions.

We will use some examples to illustrate the (navigational) usage of our complex message-sendings. These examples will be based on the following (full) interface table for a class WINDOW:

The Interface of Class WINDOW

oid	class	value	vigor		
b_1	TEXT	*"Get frame"*	WINDOW	() ⟶ RECT	kh_1
b_2	TEXT	*"Get drawable area"*	WINDOW	() ⟶ RECT	kh_2
b_3	TEXT	*"Activate"*	WINDOW	() ⟶ SELF	kh_3
b_4	TEXT	*"Deactivate"*	WINDOW	() ⟶ SELF	kh_4
b_5	TEXT	*"Is visible?"*	WINDOW	() ⟶ BOOL	kh_5
b_6	TEXT	*"Is active?"*	WINDOW	() ⟶ BOOL	kh_6
b_7	TEXT	*"Handles mouse clicks?"*	WINDOW	() ⟶ BOOL	kh_7
b_8	TEXT	*"Get title"*	WINDOW	() ⟶ TEXT	kh_8
b_9	TEXT	*"Set title"*	WINDOW	(Title,TEXT) ⟶ SELF	kh_9
b_{10}	TEXT	*"Set size"*	WINDOW	(Size,RECT) ⟶ SELF	kh_{10}
b_{11}	TEXT	*"Contains this point?"*	WINDOW	(Point,POINT) ⟶ BOOL	kh_{11}
b_{12}	TEXT	*"Supports zooming?"*	WINDOW	() ⟶ BOOL	kh_{12}
b_{13}	TEXT	*"Set zoomed size"*	WINDOW	(Size,RECT) ⟶ SELF	kh_{13}
b_{14}	TEXT	*"Show"*	WINDOW	() ⟶ SELF	kh_{14}
b_{15}	TEXT	*"Hide"*	WINDOW	() ⟶ SELF	kh_{15}
b_{16}	TEXT	*"Reverse colors"*	WINDOW	() ⟶ SELF	kh_{16}
b_{17}	TEXT	*"Highlight border"*	WINDOW	() ⟶ SELF	kh_{17}
b_{18}	TEXT	*"Double size"*	WINDOW	() ⟶ SELF	kh_{18}
b_{19}	TEXT	*"Raise to top"*	WINDOW	() ⟶ SELF	kh_{19}
b_{20}	TEXT	*"Drop to bottom"*	WINDOW	() ⟶ SELF	kh_{20}
b_{21}	TEXT	*"Flip"*	WINDOW	(Ver,BOOL) × (Hor,BOOL) ⟶ SELF	kh_{21}
b_{22}	TEXT	*"Fill color"*	WINDOW	(Color,COLOR) ⟶ SELF	kh_{22}
b_{23}	TEXT	*"Copy to"*	WINDOW	(Dest,WINDOW) ⟶ SELF	kh_{23}
b_{24}	TEXT	*"Zoom"*	WINDOW	(Direction,BOOL) ⟶ SELF	kh_{24}
b_{25}	TEXT	*"Move to"*	WINDOW	(Point,POINT) ⟶ SELF	kh_{25}
b_{26}	TEXT	*"Change size"*	WINDOW	(Xsize,INT) × (Ysize,INT) ⟶ SELF	kh_{26}
b_{27}	TEXT	*"Set coordinate type"*	WINDOW	(Coord,COORD-TYPE) ⟶ SELF	kh_{27}
b_{28}	TEXT	*"Get coordinate type"*	WINDOW	() ⟶ COORD-TYPE	kh_{28}
b_{29}	TEXT	*"Set font"*	WINDOW	(FontId,FONT) ⟶ SELF	kh_{29}

Notice that the interface of class WINDOW is also its value-based interface, that is, $\dot{\Sigma}$(WINDOW) = $\mathring{\Sigma}$(WINDOW), since there are no equal behaviors in the interface of WINDOW. As a result, sending any object with value, say, *"Get frame"* to a WINDOW receiver will cause the same execution (i.e., kh_1) as sending the object b_1 itself to that receiver.

In the above table, instances of class RECT are rectangles. The intended semantics of behaviors $b_1, ..., b_{29}$ is mostly self-explainable through their values (which are the simplest annotations). Several behaviors should be further explained: behavior b_{21} with value *"Flip"* takes two boolean arguments (labeled by Ver and Hor) to decide whether it should flip a window (the receiver) vertically or horizontally; behavior b_{24} with value *"Zoom"* takes one boolean argument (labeled by Direction) to decide whether it should zoom a window out (i.e., Direction is **True**) or zoom a window in (i.e., Direction is **False**); and behavior b_{23} with value *"Copy to"* takes one window argument (labeled by Dest) which is the destination window the behavior will copy the receiver window into.

Let *FooWindow* denote the identity of a persistent instance of class WINDOW. We also assume that the effect of these message-sendings is persistent, that is, their invokers and receivers are all persistent objects obtained from the database, and their results are also persistent objects stored back to the database. Note that, for clarity, the invokers in our examples are represented by their values, rather than by their identities. For example, the following is a simple "query" formulated as a simple message-sending: Flip the window *FooWindow* vertically.

$$\frac{\text{"}Flip\text{"}}{\text{Ver: } \mathbf{True}, \text{ Hor: } \mathbf{False}} \rightsquigarrow FooWindow$$

Notice that if the end-user wants to see the visual effect on the persistent window *FooWindow*, the message *"Show"* must be sent to *FooWindow* (which has been flipped vertically) to display the window on the screen.

We now define the behavioral semantics of complex objects in the KBO model.

6.4.1 AGG Objects as Invokers

An aggregate object can be viewed as a complex invoker with *parallel invocability.*

Definition 43: If an invoker s ⓘ AGG, then the complex message-sending $\frac{s}{\ell} \rightsquigarrow o$ is defined as

$$\frac{s}{\ell} \rightsquigarrow o \triangleq \begin{cases} \mathbf{null} & \text{if } s = [\,] \\ [A_1: \frac{s_1}{\ell} \rightsquigarrow o_1, \ldots, A_n: \frac{s_n}{\ell} \rightsquigarrow o_n] & \text{if } s = [A_1: s_1, \ldots, A_n: s_n] \end{cases}$$

where $o_j \asymp o$, $o_j \not\equiv o$ and $orf(o_j) \cap orf(o) = \emptyset$ for $1 \leq j \leq n$, and for any $i, j \in [1..n]$, if $i \neq j$ then $o_i \not\equiv o_j$ and $orf(o_i) \cap orf(j) = \emptyset$. We call $\frac{s_1}{\ell} \rightsquigarrow o_1, \ldots, \frac{s_n}{\ell} \rightsquigarrow o_n$ *sub-message-sendings caused by* $\frac{s}{\ell} \rightsquigarrow o$. ■

Notice that because $o_1, ..., o_n$ are equal to o but do not share any identities with o,

they are actually deep copies of o in the sense of [KC86]. All the sub-message-sendings occur in parallel without affecting each other's receiver.

Example 21: Find the name and region of the window *FooWindow*, and the area I can draw in this window.

$[Name\text{: "}Get\ title\text{"}, Region\text{: "}Get\ frame\text{"}, WhereToDraw\text{: "}Get\ drawable\ area\text{"}]$
$\leadsto FooWindow$

Note that the aggregate invoker does not have to be constructed on the fly; it can be an existing object obtained from the database, either navigationally or associatively through a query facility. The above complex message-sending will return a new AGG object with value

$$[Name\text{: } title_1, Region\text{: } rectangle_1, WhereToDraw\text{: } rectangle_2]$$

where $title_1$ is an instance of TEXT, and $rectangle_1, rectangle_2$ are instances of RECT. ■

Example 22: Suppose the end-user wants to choose a visual effect that can best notify him about the mouse moving into the window *FooWindow*. The following query can be used as an experiment, because it does not affect the internal state of the window *FooWindow* itself.

$[Choice1\text{: "}Reverse\ colors\text{"}, Choice2\text{: "}Highlight\ border\text{"}, Choice3\text{: "}Double\ size\text{"}]$
$\leadsto FooWindow$

This query causes three sub-message-sendings:

$$\text{"}Reverse\ colors\text{"} \leadsto window_1$$
$$\text{"}Highlight\ border\text{"} \leadsto window_2$$
$$\text{"}Double\ size\text{"} \leadsto window_3$$

where $window_1, window_2, window_3$ are equal but not identical to *FooWindow* (i.e., they are deep copies of *FooWindow*). The final result of the above complex message-sending is a new AGG object with the colors-reversed $window_1$ as attribute *Choice1*, the border-highlighted $window_2$ as attribute *Choice2*, and the size-doubled $window_3$ as attribute *Choice3*. The end-user can then use the shallow-send operator (defined

in the next section) to send messages like "Show" or "Zoom" to the new aggregate object to see how differently the three windows would react. The query illustrates two advantages for experimentally minded end-users. First, the end-user can try different experiments at once on some "fake targets" ($window_1, window_2, window_3$) instead of the real one (i.e., *FooWindow*); in this way the internal state of *FooWindow* is never affected. Second, because $window_1, window_2, window_3$ are different objects (they have different identities) with the same value, the end-user can see exactly how the original target *FooWindow* would react to each individual experiment (this cannot be achieved if $window_1, window_2, window_3$ are the same window object). ■

6.4.2 BIO Objects as Invokers

A bi-object can be viewed as a complex invoker with ***conditional invocability***.

Definition 44: If an invoker s Ⓘ BIO, then the complex message-sending $\frac{s}{\ell} \leadsto o$ is defined as

$$\frac{s}{\ell} \leadsto o \triangleq \begin{cases} \frac{s.then}{\ell} \leadsto o & \text{if } (\frac{s.if}{\ell} \leadsto o) \notin \{\textbf{False}, \textbf{\textit{fail}}\} \\ \frac{s.else}{\ell} \leadsto o & \text{otherwise} \end{cases}$$

where $s.if$, $s.then$ and $s.else$ denote the if-part, the then-part and the else-part of s, respectively. We call $\frac{s.if}{\ell} \leadsto o$ and $\frac{s.then}{\ell} \leadsto o$ (or $\frac{s.else}{\ell} \leadsto o$) *sub-message-sendings caused by* $\frac{s}{\ell} \leadsto o$. ■

Example 23: Zoom out the window *FooWindow* if possible, and hide it otherwise.

$$\frac{\{\text{"}Support\ zooming?\text{"}, \text{"}Zoom\text{"}, \text{"}Hide\text{"}\}}{\text{Direction: }\textbf{True}} \leadsto FooWindow$$

Again, the bi-object invoker here can be an existing object obtained from the database. When this query is evaluated,

$$\text{"}Support\ zooming?\text{"} \leadsto FooWindow$$

occurs first. If this returns **True** then

$$\frac{\text{"}Zoom\text{"}}{\text{Direction : }\textbf{True}} \leadsto FooWindow$$

occurs (i.e., zoom out *FooWindow*); otherwise,

$$\text{"}Hide\text{"} \leadsto FooWindow$$

occurs. Notice that only invoker "Zoom" invokes an knowhow that needs a labeled argument (i.e., Direction). ■

6.4.3 SET Objects as Invokers

A set object can be viewed as a complex invoker with ***random-order invocability***.

Definition 45: If an invoker s ⓘ SET, then the complex message-sending $\frac{s}{\ell} \leadsto o$ is defined as

$$\frac{s}{\ell} \leadsto o \triangleq \begin{cases} \textbf{\textit{null}} & \text{if } s = \{\} \\ \frac{s_1}{\ell} \leadsto o & \text{if } s = \{s_1\} \\ \frac{s-\{x\}}{\ell} \leadsto o & \text{after } \frac{x}{\ell} \leadsto o\text{, otherwise} \end{cases}$$

where $x = arb(s)$ and the function $arb(s)$ yields an arbitrarily selected element of the set object s. For each $x \in s$, we call $\frac{x}{\ell} \leadsto o$ a ***sub-message-sending caused by*** $\frac{s}{\ell} \leadsto o$. ■

It can be observed that the execution of a set object (i.e., send it as an invoker to a receiver) is non-deterministic (so that the end-user can say "I don't care how these sub-invokers are sent to the receiver; all I want is to send all of them!"), in the sense that the particular order in which each sub-object in the set is sent to the receiver is not guaranteed by the semantics. If determinism is desired, the end-user should use a sequence object as the invoker.

Example 24: Let p be an instance of class POINT and q an instance of class COLOR. Change the size of the window *FooWindow* to 30x50 and fill it with color q and reposition it at point p.

$$\frac{\{\text{“}\textit{Move to}\text{”}, \text{“}\textit{Change size}\text{”}, \text{“}\textit{Fill color}\text{”}\}}{\text{Point: } p,\ \text{Xsize: } 30,\ \text{Ysize: } 50,\ \text{Color: } q} \leadsto \textit{FooWindow}$$

Again, the set invoker may be an existing object obtained (navigationally or associatively) from the database. In this query, since the invoker is a set object, the order in which "Move to", "Change size", and "Fill color" are sent to *FooWindow* is immaterial. This is because the final state of *FooWindow* will be the same no matter which sub-invoker is sent first, second or last. For example, if *FooWindow* is filled with color q before its size is changed to 30x50, the resized *FooWindow* will still be filled with color q. As a result, the end-user does not have to worry about how the sub-invokers are organized in a set invoker. ■

Example 25: Let t be an instance of COORD-TYPE and f an instance of FONT. Set the title of *FooWindow* to "Tool Box", its coordinate type to t, and its font type to f.

$$\frac{\{\text{“}\textit{Set title}\text{”}, \text{“}\textit{Set coordinate type}\text{”}, \text{“}\textit{Set font}\text{”}\}}{\text{Title: “}\textit{Tool Box}\text{”},\ \text{Coord: } t,\ \text{FontId: } f} \leadsto \textit{FooWindow}$$

Again, the three sub-invokers in the set invoker do not affect each other's computation result. As we can see, SET invokers are particularly useful for invoking a set of non-interactive update behaviors. ■

6.4.4 SEQ Objects as Invokers

A sequence object can be viewed as a complex invoker with *sequential invocability*.

Definition 46: If an invoker s Ⓘ SEQ, then the complex message-sending $\frac{s}{\ell} \leadsto o$ is defined as

$$\frac{s}{\ell} \leadsto o \triangleq \begin{cases} \textit{null} & \text{if } s = <> \\ \frac{s_n}{\ell} \leadsto (\cdots \leadsto (\frac{s_2}{\ell} \leadsto (\frac{s_1}{\ell} \leadsto o))\ldots) & \text{when } s = <s_1, s_2, \ldots, s_n> \end{cases}$$

We call each $\frac{s_{i+1}}{\ell} \leadsto r_i$ a *sub-message-sending caused by* $\frac{s}{\ell} \leadsto o$, where $r_i \equiv \frac{s_i}{\ell} \leadsto r_{i-1}$ $(r_0 \equiv o)$ for $1 \leq i \leq n-1$. ■

Example 26: Let w be an instance of WINDOW. Zoom out window *FooWindow* and copy it to window w.

$$\frac{<\textit{“Activate”}, \textit{“Zoom”}, \textit{“Copy to”}>}{\text{Direction: }\textbf{True}, \text{ Dest: } w} \leadsto \textit{FooWindow}$$

Note that before we zoom out a window it is safer to activate that window in case it has been deactivated. Also, we have to zoom out *FooWindow* first before we copy it to window w; otherwise, if we had done the reverse, we would have copied the smaller *FooWindow* to window w. The above complex message-sending first causes

$$\textit{“Activate”} \leadsto \textit{FooWindow}$$

which activates window *FooWindow* and then returns *FooWindow* itself. Since the result of the first sub-message-sending is *FooWindow* itself, the second sub-message-sending is

$$\frac{\textit{“Zoom”}}{\text{Direction: }\textbf{True}} \leadsto \textit{FooWindow}$$

which again returns the receiver itself. So the third sub-message-sending is

$$\frac{\textit{“Copy to”}}{\text{Dest: } w} \leadsto \textit{FooWindow}$$

which returns the final result of the complex message-sending (again, the result is *FooWindow* itself). ■

Example 27: Let p be an instance of class POINT. Let *MyMove* denote a SEQ object with value $<$*“Move to”*, *“Raise to top”*$>$. Raise the window *FooWindow* to

the top if it contains point p; otherwise, move *FooWindow* to point p and then raise it to the top.

$$\frac{\{\text{``}\mathit{Contains\ this\ point?}\text{''}, \text{``}\mathit{Raise\ to\ top}\text{''}, MyMove\}}{\text{Point: } p} \leadsto FooWindow$$

Notice that messages "Contains this point?" and "Move to" (in *MyMove*) share the same labeled argument (Point: p). Also, because the effect of this query is persistent, one can only see window *FooWindow* on top of other windows when *FooWindow* is loaded onto the screen (i.e., when it must receive message "Show"). ■

As a final remark in this section, we should point out two things: (1) When the invoker is an AGG object, $\leadsto$ is an ***object generating*** operation because a new AGG object with a new oid is produced as the result; (2) When the invoker is a CAPSULE, BIO, SET, or SEQ object, $\leadsto$ is an ***object preserving*** operation in the sense that the operational semantics of $\leadsto$ does not produce a new complex object as the result. Furthermore, our example queries are not intended to demonstrate certain new power of a programming language (they may be easily hardcoded in a persistent programming language); rather, they are intended to illustrate what an OODB end-user, who has only objects and the query facility $\leadsto$ (plus creation and deletion operations), can do to query the database. Because our message-sending operation $\leadsto$ takes any object both as the receiver and as the invoker, it provides the possibility for using a conventional query language (such as an object algebra) to associatively select or generate the invoker as well as the receiver in our message-sendings.

6.5 Shallow-Send and Deep-Send

Recall in Chapter 3 we provided an intuitive discussion of the aggregations in structural modeling. To formally capture the intended semantics of aggregations in structural modeling, we introduce other two message-sending primitives, ***shallow-send*** and ***deep-send***, which define the ways an invoker may walk through the members of a complex receiver.

Definition 47: *Semantics of Shallow-Send* $\overset{1}{\leadsto}$. Let ℓ be a blackboard and let

$s, o \in \mathcal{O}$. Then, the ***shallow-send*** operator, written $\overset{1}{\leadsto}$, is defined as follows:

$$\frac{s}{\ell} \overset{1}{\leadsto} o \triangleq \begin{cases} \frac{s}{\ell} \leadsto o & \text{if } \frac{s}{\ell} \leadsto o \not\equiv \boldsymbol{fail} \\ [A_1\colon \frac{s}{\ell} \leadsto o_1, \ldots, A_n\colon \frac{s}{\ell} \leadsto o_n] & \text{if } o \, \textcircled{i} \, \text{AGG} \wedge o = [A_1\colon o_1, \ldots, A_n\colon o_n] \\ \frac{s}{\ell} \leadsto o.then & \text{if } o \, \textcircled{i} \, \text{BIO} \wedge o.if \not\equiv \boldsymbol{null} \\ \frac{s}{\ell} \leadsto o.else & \text{if } o \, \textcircled{i} \, \text{BIO} \wedge o.if \equiv \boldsymbol{null} \\ \{\frac{s}{\ell} \leadsto o_1, \ldots, \frac{s}{\ell} \leadsto o_n\} & \text{if } o \, \textcircled{i} \, \text{SET} \wedge o = \{o_1, \ldots, o_n\} \\ <\frac{s}{\ell} \leadsto o_1, \ldots, \frac{s}{\ell} \leadsto o_n> & \text{if } o \, \textcircled{i} \, \text{SEQ} \wedge o = <o_1, \ldots, o_n> \\ \boldsymbol{fail} & \text{otherwise} \end{cases}$$

■

Intuitively, the shallow-send operator is exactly the same as $\leadsto$ if s can invoke some behavior or behaviors on o. However, if that fails, the shallow operator sends invoker s into the members at the first level of o's structure (if o is a complex object). In particular, when o is a BIO object, the shallow-send operator will use the value of the if-part of o to decide which member of o should receive s.

Example 28: Assume *Phds* is a set object with value $\{Mike, Joan, Lee, Mary\}$, where $Mike, Joan, Lee$ and $Mary$ are instances of PHD-STUDENT. The set object *Phds* is an instance of PHD-STUDENTS. Suppose *Taylor* is the supervisor of *Mike* and *Joan*, and *Smith* is the supervisor of *Lee* and *Mary*, Then, the message-sending

$$\text{“}\mathit{Who\ is\ your\ supervisor?}\text{”} \overset{1}{\leadsto} \mathit{Phds}$$

will produce a new set object with the value $\{Taylor, Smith\}$. Notice that if we used the Send operator $\leadsto$ here, then ***"Who is your supervisor?"*** $\leadsto$ ***Phds*** would return ***fail*** because class PHD-STUDENTS does not have a behavior with value "Who is your supervisor?". ■

Definition 48: *Semantics of Deep-Send* $\overset{*}{\leadsto}$ Let ℓ be a blackboard and let $s, o \in \mathcal{O}$.

Then, the *deep-send* operator, written $\overset{\bullet}{\rightsquigarrow}$, is defined as follows:

$$\frac{s}{\ell} \overset{\bullet}{\rightsquigarrow} o \triangleq \begin{cases} \frac{s}{\ell} \rightsquigarrow o & \text{if } \frac{s}{\ell} \rightsquigarrow o \not\equiv \textbf{\textit{fail}} \\ [A_1: \frac{s}{\ell} \overset{\bullet}{\rightsquigarrow} o_1, \ldots, A_n: \frac{s}{\ell} \overset{\bullet}{\rightsquigarrow} o_n] & \text{if } o \text{ ⓘ AGG} \wedge o = [A_1: o_1, ..., A_n: o_n] \\ \frac{s}{\ell} \overset{\bullet}{\rightsquigarrow} o.then & \text{if } o \text{ ⓘ BIO} \wedge o.if \not\equiv \textbf{null} \\ \frac{s}{\ell} \overset{\bullet}{\rightsquigarrow} o.else & \text{if } o \text{ ⓘ BIO} \wedge o.if \equiv \textbf{null} \\ \{\frac{s}{\ell} \overset{\bullet}{\rightsquigarrow} o_1, \ldots, \frac{s}{\ell} \overset{\bullet}{\rightsquigarrow} o_n\} & \text{if } o \text{ ⓘ SET} \wedge o = \{o_1, ..., o_n\} \\ < \frac{s}{\ell} \overset{\bullet}{\rightsquigarrow} o_1, \ldots, \frac{s}{\ell} \overset{\bullet}{\rightsquigarrow} o_n > & \text{if } o \text{ ⓘ SEQ} \wedge o = < o_1, ..., o_n > \\ \textbf{\textit{fail}} & \text{otherwise} \end{cases}$$

■

Notice that this definition is recursive. Therefore, the Deep-Send operator should be used under the assumption that the object graph of the receiver o does not contain cycles. We consider this is a reasonable assumption in most cases.

Example 29: Assume a class FAMILY has the representation

$$\dot{\Xi}(\text{FAMILY}) = [\text{Father: PERSON, Mother: PERSON, Kids: PERSONS}]$$

where $\dot{\Xi}$(PERSONS) = {PERSON}. And suppose PERSON has a behavior with value "How old are you?" and signature () ⟶ INT. Let *Taylors* be an instance of FAMILY with value

$$[\text{Father:}Taylor, \text{Mother:}Lisa, \text{Kids:}\{Mike, Joan, Mary\}]$$

Then, the message-sending

$$\text{“}How\ old\ are\ you?\text{”} \overset{\bullet}{\rightsquigarrow} Taylors$$

will produce a new aggregate object with a value like

$$[\text{Father:}45, \text{Mother:}44, \text{Kids:}\{17, 15, 11\}]$$

Notice that if we used shallow-send $\overset{1}{\rightsquigarrow}$ instead of deep-send in this message-sending, that is,

$$\text{“}How\ old\ are\ you?\text{”} \overset{1}{\rightsquigarrow} Taylors$$

then the result would have the value

[Father:45, Mother:44, Kids:***fail***]

This is because class PERSONS (for attribute Kids) does not have a behavior with value "How old are you?". ■

We can further generalize the shallow-send operator to a "N-level Send" operator defined as follows:

Definition 49: *Semantics of N-level Send* $\overset{n}{\rightsquigarrow}$. Let ℓ be a blackboard and let $s, o \in \mathcal{O}$. Then, the *n-level send* operator ($n \geq 0$), written $\overset{n}{\rightsquigarrow}$, is defined as follows:

$$\frac{s}{\ell} \overset{n}{\rightsquigarrow} o \triangleq \begin{cases} \frac{s}{\ell} \rightsquigarrow o & \text{if } \frac{s}{\ell} \rightsquigarrow o \not\equiv \textbf{\textit{fail}} \\ [A_1: \frac{s}{\ell} \overset{n-1}{\rightsquigarrow} o_1, \ldots, A_n: \frac{s}{\ell} \overset{n-1}{\rightsquigarrow} o_n] & \text{if } o \; ⓘ \; \text{AGG} \wedge o = [A_1: o_1, \ldots, A_n: o_n] \\ \frac{s}{\ell} \overset{n-1}{\rightsquigarrow} o.then & \text{if } o \; ⓘ \; \text{BIO} \wedge o.if \not\equiv \textbf{null} \\ \frac{s}{\ell} \overset{n-1}{\rightsquigarrow} o.else & \text{if } o \; ⓘ \; \text{BIO} \wedge o.if \equiv \textbf{null} \\ \{\frac{s}{\ell} \overset{n-1}{\rightsquigarrow} o_1, \ldots, \frac{s}{\ell} \overset{n-1}{\rightsquigarrow} o_n\} & \text{if } o \; ⓘ \; \text{SET} \wedge o = \{o_1, \ldots, o_n\} \\ <\frac{s}{\ell} \overset{n-1}{\rightsquigarrow} o_1, \ldots, \frac{s}{\ell} \overset{n-1}{\rightsquigarrow} o_n> & \text{if } o \; ⓘ \; \text{SEQ} \wedge o = <o_1, \ldots, o_n> \\ \textbf{\textit{fail}} & \text{otherwise} \end{cases}$$

where $\overset{n}{\rightsquigarrow}$ is $\rightsquigarrow$ when $n = 0$. ■

While the 0-level-send operator $\rightsquigarrow$ allows us to say "we are only interested in sending a message to this receiver itself", the n-level-send operators (when $n > 0$) allow us to treat a receiver as a "complex receiver", that is, they capture the semantics of aggregations in structural modeling. For instance, the $\overset{n}{\rightsquigarrow}$ operator treats a set object receiver as a group of "sub-receivers", which may themselves contain "sub-receivers", and so forth. Furthermore, the $\overset{n}{\rightsquigarrow}$ operator allows us to find out how satisfactory or complete the result of a message-sending can be at the N-th level in a complex receiver. It thus provides an interesting tool for experimental queries in ad-hoc environments.

One final comment about letting an invoker "walk through" an aggregate object in our N-level Send operators. That is, when a receiver is an AGG object, the invoker is in turn sent to each attribute. This seems quite odd at first glance, especially for those who are used to relational databases where data are not associated with a set of behaviors. In OODBs, however, objects of different types may have behaviors with

the same name but different semantics. Consider the classic "supplier-part-project" example. Suppose we have an aggregate object denoted by *SupplyRecord* with value

$$[Supplier\colon s, Part\colon p, Project\colon q]$$

where s, p, q are objects of classes SUPPLIER, PART and PROJECT, respectively. Each of these three classes may have a behavior with value "get name" and a behavior with value "get description". Then, the shallow message sending

$$\text{"get name"} \overset{1}{\leadsto} SupplyRecord$$

would produce a new aggregate object with a value something like this:

[*Supplier*: *"Hewlett-Packard Canada"*,
Part: *"HP J2181A cable scanner"*,
Project: *"LAN system 3.0"*]

Similarly, the shallow message sending

$$\text{"get description"} \overset{1}{\leadsto} SupplyRecord$$

would allow the user to see an aggregation of three descriptions: a description of the supplier company s, a description of the supplied part p, and a description of the project q that needs the part. As we can see, this kind of message sending is quite useful in many situations, and, after all, it promotes overloading polymorphism on unrelated objects (because classes SUPPLIER, PART and PROJECT may not be related by KIND-OF relationships). In fact, we noticed a complaint mentioned in [RS89] that "variables in SQL cannot range over columns". What we have provided here effectively overcomes the similar problem in OODBs.

CHAPTER 7

SAFETY AND SUCCESS OF COMPLEX MESSAGE-SENDINGS

7.1 Introduction

In this chapter, we study the safety and success properties of our complex message-sendings. Our goal is to provide a formal mechanism to predict (1) whether a complex message-sending will terminate, and (2) whether a complex message-sending will succeed and return a non-***fail*** result. It is obviously desirable that complex message-sendings leading to unavoidable nontermination should be prevented from occurring. It may also be useful to warn the end-user who formulates a complex message-sending that has the potential to be nonterminating (i.e., it may or may not terminate). Success prediction is a fundamental mechanism to protect the receiver from being affected by unnecessary knowhow executions caused by a failed complex message-sending (i.e., it returns ***fail*** to the end-user). Since complex message-sendings that return ***fail*** are meaningless, it is desirable to eliminate them before their actual evaluation.

To study the termination of complex message-sendings, we must assume that, in a KBO database, all knowhows terminate when executed. To study the success of complex message-sendings, we must assume that, in a KBO database, no knowhow returns ***fail*** when its invocation is valid by the $\rightsquigarrow$ operation. In other words, these assumptions imply that simple message-sendings will either succeed to invoke a behavior (and execute its knowhow) and return a non-***fail*** result, or fail to invoke a behavior and return ***fail***. In fact, these two assumptions are part of the definition of a KBO database (in Chapter 5), which contains a consistent set of KBO objects. Without these two assumptions, the problems of termination and success would be equivalent to the halting problem, and thus be undecidable.

For convenience, throughout this chapter we will often use $o = v$ to denote an object o with value v. For instance, $x = <y_1, y_2>$ denotes a sequence object x with value $<y_1, y_2>$.

7.2 Well-Formed Complex Message-Sendings

It can be observed that the four kinds of complex message-sending implied by our four object constructors (i.e., AGG, BIO, SET, SEQ) are expressive enough to allow

the construction of complex message-sendings that have no reasonable interpretation or cause unpredictable knowhow execution (i.e., a complex message-sending returns ***fail*** even when it does successfully invoke some knowhows). To avoid such meaningless or only partially successful complex message-sendings, we wish to restrict our databases to complex message-sendings which are considered "well-formed". In order to introduce such a notion of well-formedness, we shall first define the notion of an evaluation set.

Definition 50: *Evaluation Sets.* Let μ be a message-sending. The ***evaluation set*** of μ, denoted $\varepsilon(\mu)$, is a set of message-sendings defined by

$$\varepsilon(\mu) = \{\frac{s}{\ell} \leadsto r \mid \frac{s}{\ell} \leadsto r \text{ is a sub-message-sending caused by } \mu,\ r \neq \textbf{\textit{null}} \text{ and } r \neq \textbf{\textit{fail}}\ \}$$

The transitive evaluation set of μ, denoted by ε^*, is defined as:

$$\begin{aligned}
\varepsilon^1(\mu) &= \varepsilon(\mu) \\
\varepsilon^n(\mu) &= \cup_{\nu \in \varepsilon^{n-1}(\mu)} \varepsilon(\nu) \cup \varepsilon^{n-1}(\mu) \\
\varepsilon{*}(\mu) &= \cup_{n=1}^{\infty} \varepsilon^n(\mu)
\end{aligned}$$

■

Note that $\varepsilon^*(\mu) = \emptyset$ when μ is a simple message-sending (i.e., a simple message-sending does not cause any sub-message-sendings).

Form the above definition, the following is immediate:

Theorem 11: *If a message-sending* $\frac{s}{\ell} \leadsto r$ *causes a sub-message-sending* $\frac{s'}{\ell} \leadsto r'$, *then*

$$\varepsilon^*(\frac{s}{\ell} \leadsto r) \supset \varepsilon^*(\frac{s'}{\ell} \leadsto r')$$

Proof: From the above definition, we immediately have $\varepsilon^*(\frac{s}{\ell} \leadsto r) \supseteq \varepsilon^*(\frac{s'}{\ell} \leadsto r')$. Because $\frac{s}{\ell} \leadsto r$ causes sub-message-sending $\frac{s'}{\ell} \leadsto r'$, we have $(\frac{s'}{\ell} \leadsto r') \in \varepsilon(\frac{s}{\ell} \leadsto r)$, and therefore $(\frac{s'}{\ell} \leadsto r') \in \varepsilon^*(\frac{s}{\ell} \leadsto r)$. However, message-sending $\frac{s'}{\ell} \leadsto r'$ can never be included in $\varepsilon(\frac{s'}{\ell} \leadsto r')$ because it takes place before any of its sub-message-sendings, and therefore $(\frac{s'}{\ell} \leadsto r') \notin \varepsilon^*(\frac{s'}{\ell} \leadsto r')$. Consequently, $\varepsilon^*(\frac{s}{\ell} \leadsto r) \supset \varepsilon^*(\frac{s'}{\ell} \leadsto r')$. ■

We now give some examples to illustrate evaluation sets. We assume that class WINDOW has the same interface table used for the examples in Chapter 6.4.

Example 30: Let a complex message-sending μ be the following:

$$\frac{<\text{“}\mathit{Activate}\text{”}, \text{“}\mathit{Zoom}\text{”}, \text{“}\mathit{Copy\ to}\text{”}>}{\text{Direction: }\textbf{True}\text{, Dest: } w} \leadsto \mathit{FooWindow}$$

where *FooWindow* is an instance of class WINDOW. This message-sending causes three (simple) sub-message-sendings:

μ_1 :

$$\frac{\text{"}\textit{Activate}\text{"}}{\text{Direction: }\mathbf{True},\ \text{Dest: } w} \rightsquigarrow \mathit{FooWindow} \Rightarrow \mathit{result}_1$$

μ_2 :

$$\frac{\text{"}\textit{Zoom}\text{"}}{\text{Direction: }\mathbf{True},\ \text{Dest: } w} \rightsquigarrow \mathit{result}_1 \Rightarrow \mathit{result}_2$$

μ_3 :

$$\frac{\text{"}\textit{Copy to}\text{"}}{\text{Direction: }\mathbf{True},\ \text{Dest: } w} \rightsquigarrow \mathit{result}_2 \Rightarrow \mathit{result}_3$$

Therefore, we have the evaluation set $\varepsilon(\mu) = \{\mu_1, \mu_2, \mu_3\}$. Since μ_1, μ_2, μ_3 are all simple message-sendings (i.e., they do not cause other sub-message-sendings), we also have $\varepsilon^*(\mu) = \varepsilon(\mu)$. ■

Example 31: Consider the following complex message-sending μ:

$$\frac{\{\ \text{"}\textit{Support zooming?}\text{"}, \text{"}\textit{Zoom}\text{"}, \text{"}\textit{Hide}\text{"}\ \}}{\text{Direction: }\mathbf{True}} \rightsquigarrow \mathit{FooWindow}$$

where the invoker is a bi-object with if-part "*Support zooming?*", then-part "*Zoom*" and else-part "*Hide*". This message-sending either causes these two (simple) sub-message-sendings:

μ_1 :

$$\frac{\text{"}\textit{Support zooming?}\text{"}}{\text{Direction: }\mathbf{True}} \rightsquigarrow \mathit{FooWindow} \Rightarrow \mathit{result}_1$$

μ_2 :

$$\frac{\text{"}\textit{Zoom}\text{"}}{\text{Direction: }\mathbf{True}} \rightsquigarrow \mathit{FooWindow} \Rightarrow \mathit{result}_2$$

where $\mathit{result}_1 \notin \{\mathbf{False}, \mathit{\mathbf{fail}}\}$, or causes these two (simple) sub-message-sendings:

μ_1 :

$$\frac{\text{"}\textit{Support zooming?}\text{"}}{\text{Direction: }\mathbf{True}} \rightsquigarrow \mathit{FooWindow} \Rightarrow \mathit{result}_1$$

μ'_2 :

$$\frac{\text{"}\textit{Hide}\text{"}}{\text{Direction: }\mathbf{True}} \rightsquigarrow \mathit{FooWindow} \Rightarrow \mathit{result}_2$$

where $result_1 \in \{\mathbf{False}, \boldsymbol{fail}\}$. Therefore, the evaluation set $\varepsilon(\mu)$ is either $\{\mu_1, \mu_2\}$ or $\{\mu_1, {\mu'}_2\}$. Again, since $\mu_1, \mu_2, {\mu'}_2$ are all simple message-sendings, we also have $\varepsilon^*(\mu) = \varepsilon(\mu)$. ■

Example 32: The following complex message-sending μ will cause another complex message-sending:

$$\frac{\{\text{“}\mathit{Fill\ color}\text{”}, <\text{“}\mathit{Move\ to}\text{”}, \text{“}\mathit{Raise\ to\ top}\text{”}>\}}{\text{Color: } q,\ \text{Point: } p} \leadsto \mathit{FooWindow}$$

where q is an instance of COLOR and p an instance of POINT. The message-sending μ causes two sub-message-sendings (in an arbitrary order):

μ_1 :

$$\frac{\text{“}\mathit{Fill\ color}\text{”}}{\text{Color: } q,\ \text{Point: } p} \leadsto \mathit{FooWindow}$$

μ_2 :

$$\frac{<\text{“}\mathit{Move\ to}\text{”}, \text{“}\mathit{Raise\ to\ top}\text{”}>}{\text{Color: } q,\ \text{Point: } p} \leadsto \mathit{FooWindow}$$

Hence, $\varepsilon(\mu) = \{\mu_1, \mu_2\}$. Because μ_2 is also a complex message-sending, it will cause the following two (simple) sub-message-sendings:

μ_3 :

$$\frac{\text{“}\mathit{Move\ to}\text{”}}{\text{Color: } q,\ \text{Point: } p} \leadsto \mathit{FooWindow} \Rightarrow \mathit{result}_3$$

μ_4 :

$$\frac{\text{“}\mathit{Raise\ to\ top}\text{”}}{\text{Color: } q,\ \text{Point: } p} \leadsto \mathit{result}_3$$

Hence, $\varepsilon^2(\mu) = \{\mu_1, \mu_2, \mu_3, \mu_4\}$. Because μ_3 and μ_4 are both simple message-sendings, we have $\varepsilon^*(\mu) = \varepsilon^2(\mu) = \{\mu_1, \mu_2, \mu_3, \mu_4\}$. ■

We now formally define the notion of well-formed complex message-sendings. Our approach starts by looking at what complex message-sendings can be considered to be "not well-formed". We first define the notion of nonterminating complex message-sendings.

Definition 51: *Nonterminating Complex Message-Sendings.* A complex message-sending μ is a *nonterminating* complex message-sending if and only if $|\varepsilon^*(\mu)| = \infty$. ■

Here $|S|$ denotes the cardinality of a set S. This definition implies that a nonterminating complex message-sending cannot be evaluated in finite time. The following example illustrates the nature of a nonterminating complex message-sending.

Example 33: Assume that class PERSON has two behaviors b_1 and b_2:

$$value(b_1) = \text{"drink beer"} \qquad sig(b_1) = () \longrightarrow \text{SELF}$$
$$value(b_2) = \text{"dance"} \qquad sig(b_2) = () \longrightarrow \text{SELF}$$

Let *ComeOn* denote a set object:

$$ComeOn = \{\text{"drink beer"}, \text{"dance"}, ComeOn\}$$

Let *Mike* denote an instance of PERSON. Then, the complex message-sending

$$ComeOn \leadsto Mike$$

is a nonterminating complex message-sending because it will unavoidably cause an infinite set of other message-sendings, such as the following series:

$$\text{"dance"} \leadsto Mike$$
$$\text{"drink beer"} \leadsto Mike$$
$$ComeOn \leadsto Mike$$
$$\text{"drink beer"} \leadsto Mike$$
$$\text{"dance"} \leadsto Mike$$
$$ComeOn \leadsto Mike$$
$$\vdots$$
$$\vdots$$

Notice that each time the message-sending $ComeOn \leadsto Mike$ takes place, it always causes three sub-message-sendings (in an arbitrary order). This is a typical example of a nonterminating complex message-sending. In particular, it can never terminate no matter what results "*drink beer*" $\leadsto$ *Mike* and "*dance*" $\leadsto$ *Mike* return, because the invoker *ComeOn* will unavoidably cause $ComeOn \leadsto Mike$ again. ■

Having defined the notion of nonterminating complex message-sendings, we now formally define the notion of unsuccessful complex message-sendings.

Definition 52: *Unsuccessful Complex Message-Sendings.* A complex message-sending μ is an *unsuccessful* complex message-sending if and only if μ returns ***fail***. ■

Usually, a complex message-sending is an unsuccessful complex message-sending because its receiver is not the right receiver or it cannot find the legal arguments on the blackboard[7]. We illustrate this in the following example.

[7] In interactive environments, such an error may be ignored if the interface always prompts the

Example 34: Let *FooColor* be an instance of class COLOR, which does not have behaviors with values "*Activate*", "*Zoom*", "*Copy to*" (recall they are behaviors of class WINDOW). Then the following is an unsuccessful complex message-sending:

$$\frac{<\text{"}Activate\text{"}, \text{"}Zoom\text{"}, \text{"}Copy\ to\text{"}>}{\text{Direction: \textbf{True}, Dest: } w} \leadsto FooColor$$

Notice that this message-sending does not cause any knowhow execution, that is, nothing has happened to the receiver *FooColor*. This is because the first sub-message-sending

$$\frac{\text{"}Activate\text{"}}{\text{Direction: \textbf{True}, Dest: } w} \leadsto FooColor$$

returns ***fail***, and therefore the other two sub-message-sendings have ***fail*** as their receiver and also return ***fail***. The following complex message-sending has a right receiver but an illegal argument:

$$\frac{<\text{"}Activate\text{"}, \text{"}Zoom\text{"}, \text{"}Copy\ to\text{"}>}{\text{Direction: 25, Dest: } w} \leadsto FooWindow$$

because the invoker "*Zoom*" is expecting a BOOL object as the Direction argument, not an integer. Hence, this complex message-sending also returns ***fail***. Notice that, in an interactive environment, the interface may still evaluate the first invoker "*Activate*" on *FooWindow* and prompt the end-user for the right argument for the label Direction. ■

We are particularly interested in a subset of unsuccessful complex message-sendings, which are what we call anomalous complex message-sendings.

Definition 53: *Anomalous Complex Message-Sendings.* A complex message-sending μ is an *anomalous* complex message-sending if and only if μ is an unsuccessful complex message-sending but there exists at least one simple message-sending $\mu' \in \varepsilon^*(\mu)$ which does not return ***fail***. ■

The following is an example of an anomalous complex message-sending.

Example 35: Let *FooWindow* be an instance of class WINDOW. Assume that class COLOR has a behavior with value "*Is blue color?*" and signature () ⟶ BOOL. Then the following is an anomalous complex message-sending:

$$<\text{"}Deactivate\text{"}, \text{"}Is\ blue\ color?\text{"}> \leadsto FooWindow$$

More precisely, this complex message-sending causes two sub-message-sendings. The first one is "*Deactivate*" $\leadsto$ *FooWindow*, which succeeds and executes a knowhow

end-user for the missing (or legal) arguments.

that de-activates window *FooWindow* and returns *FooWindow* itself. This result is, in turn, the receiver of the second sub-message-sending, i.e., "*Is blue color?*" $\leadsto$ *FooWindow*, which fails and returns ***fail*** because class WINDOW does not have a behavior with value "*Is blue color?*". ■

Anomalous complex message-sendings are in general not desirable because they are not intended by the end-user. More seriously, they may cause unpredictable effects on the receiver. In the above example, although the message-sending returns ***fail***, the internal state of window *FooWindow* has already been affected — *FooWindow* is de-activated. (Maybe the end-user unintentionally selected or constructed a wrong invoker or receiver.) It seems to become worse when the invoker is a deeply nested complex object, since the end-user does not know what did and what did not happen. Furthermore, undesired affects caused by anomalous complex message-sendings may be very costly to undo (if possible).

In the above two examples, ***fail*** is returned due to "type errors". In the first example, message "*Activate*" is sent to a receiver of the wrong type (i.e., class). In the second example, message "*Is blue color?*" is sent to a receiver of the wrong type. As we discussed above, it is highly desirable that unsuccessful complex message-sendings (particularly, anomalous complex message-sendings) should be detected and prevented from being evaluated. This is the task of *dynamic type-checking*. Here, an important point must be made about type checking of message-sendings. While type checking is undoubtedly a useful and essential form of data protection, there is no fundamental reason to enforce this checking during compile-time; it can be performed at run-time. Particularly, in the KBO model, we have to know the actual invoker object before we can do type-checking on a message-sending, because the class (type) of an object variable will provide no information about an actual object's signature.

With the previous definitions, we can now define the notion of well-formed complex message-sendings.

Definition 54: *Well-Formed Complex Message-Sendings.* A complex message-sending μ is a *well-formed* complex message-sending if and only if:

- μ is not a nonterminating complex message-sending, and
- μ is not an anomalous complex message-sending.

■

Notice that if we can guarantee that a complex message-sending is not an unsuccessful complex message-sending then we can also guarantee that the complex message-sending is not an anomalous complex message-sending. However, in an interpreted

query environment, we usually do not care about the unsuccessful complex message-sendings that are not anomalous, because they cause no harm and have to be checked at the simple message-sending level anyway.

The above definition of well-formedness is a semantic one. In the rest of this chapter, we analyze the syntactic properties which will help us to design techniques for detecting non-well-formed complex message-sendings, without actually evaluating them. In Section 7.3 and 7.4 we develop some formal methods for detecting nonterminating complex message-sendings. In Section 7.5 and 7.6 we develop a dynamic typing system to guarantee the success of a complex message-sending. The typing system is proven to be sound; however, it is not complete. We will discuss why we choose to make the typing system more conservative.

7.3 Necessarily Unsafe Complex Message-sendings

In this section, we study the properties of the complex message-sendings which will be unavoidably nonterminating on any non-bottom receiver. Based on these properties, this kind of *necessarily unsafe* complex message-sendings can be detected and immediately rejected, without real evaluation (i.e., execution of knowhows).

Recall that the function $orf(o)$ denotes the set of all identities of objects reachable from object o. In other words, $orf(o)$ contains all object-identities referenced within the value part of an object o, including all identities of objects recursively nested within o. Now, let $orf^1(o)$ contain only the object-identities directly referenced in $value(o)$. For example, if object $o = [A_1: o_1, A_2: o_2]$ where $o_1 = \{o_3, o_4\}$ and o_2, o_3, o_4 are capsule objects, then $orf(o) = \{oid(o_1), oid(o_2), oid(o_3), oid(o_4)\}$, but $orf^1(o) = \{oid(o_1), oid(o_2)\}$.

In the following we analyze the structure of invokers in complex message-sendings. We first define the notion of α-paths. It will be clear that the existence of α-paths in an invoker can explain why a complex message-sending is necessarily unsafe.

Definition 55: *α-paths.* Let x, y be two KBO objects. Then, an *α-path* from object x to object y is defined as follows:

1. If $oid(y) \in orf^1(x)$ and x is either an AGG object or a SET object, then xy is an α-path from x to y.

2. If $oid(y) \in orf^1(x)$ and x is a BIO object such that $x = \langle\!\langle\, y, y', y'' \,\rangle\!\rangle$, then xy is an α-path from x to y.

3. If $oid(y) \in orf^1(x)$ and x is a SEQ object such that $x = <y, y_1, y_2, \ldots>$, then xy is an α-path from x to y.

4. If α_1 is an α-path from x to z and α_2 is an α-path from z to y, then $\alpha_1\alpha_2$ is an α-path from x to y.

where y', y'', y_1, y_2, z are any KBO objects. ■

Example 36: Let x be an AGG object, x_2 a SET object, y_2 a SEQ object, and z_1 a BIO object. Let their values be as follows:

$$x = [A_1 \colon x_1, A_2 \colon x_2]$$
$$x_2 = \{y_1, y_2, y_3\}$$
$$y_2 = < z_1, z_2, z_3, z_4 >$$
$$z_1 = \{\!\!\{ w_1, w_2, w_3 \}\!\!\}$$

where other objects $x_1, y_1, y_3, z_2, z_3, z_4, w_1, w_2, w_3$ are capsule objects. The object graph of x is shown in Figure 7.17. As we can see, some of the α-paths in this graph are: xx_2y_1 is an α-path from object x to object y_1, $xx_2y_2z_1$ is an α-path from object x to object z_1, and $xx_2y_2z_1w_1$ is an α-path from object x to object w_1. However, for example, $xx_2y_2z_1w_2$ is not an α-path, and $xx_2y_2z_3$ is neither an α-path. ■

The following theorem shows that if there is an α-path from object x to object y then a complex message-sending, where x is the invoker, will *necessarily* (or unavoidably) cause another message-sending where the invoker is y.

Theorem 12: *Let x and y be any two KBO objects such that $oid(y) \in orf(x)$. Let o be any non-bottom object. Then, if there is an α-path from object x to object y then*

$$\varepsilon^*(\frac{x}{\ell} \leadsto o) \supset \varepsilon^*(\frac{y}{\ell} \leadsto o')$$

where $o' \equiv o$ or $o' \asymp o$, for any complex message-sending $\frac{x}{\ell} \leadsto o$.

Proof: Let $x_1 = x$ and let $x_1x_2\cdots x_{n-1}x_ny$ be an α-path from x to y, where n can be called the *length* of the α-path. We prove this theorem by an induction on the length n. For $n = 1$, we have $oid(y) \in orf^1(x)$, that is, $oid(y)$ is directly referenced in $value(x)$. The α-path from x to y is simply xy. Since $\frac{x}{\ell} \leadsto o$ is a complex message-sending, the invoker x and the receiver o cannot be bottom objects ***fail*** or ***null*** or ***same*** (otherwise they are simple message-sendings). We thus have the following three cases:

Case 1: Let x be either an AGG object or a SET object. That is, object x has value $[\ldots, A \colon y, \ldots]$ or $\{\ldots, y, \ldots\}$. By definition of complex message-sendings, $\frac{x}{\ell} \leadsto o$ will definitely cause the sub-message-sending $\frac{y}{\ell} \leadsto o'$, where $o' \asymp o$ if x

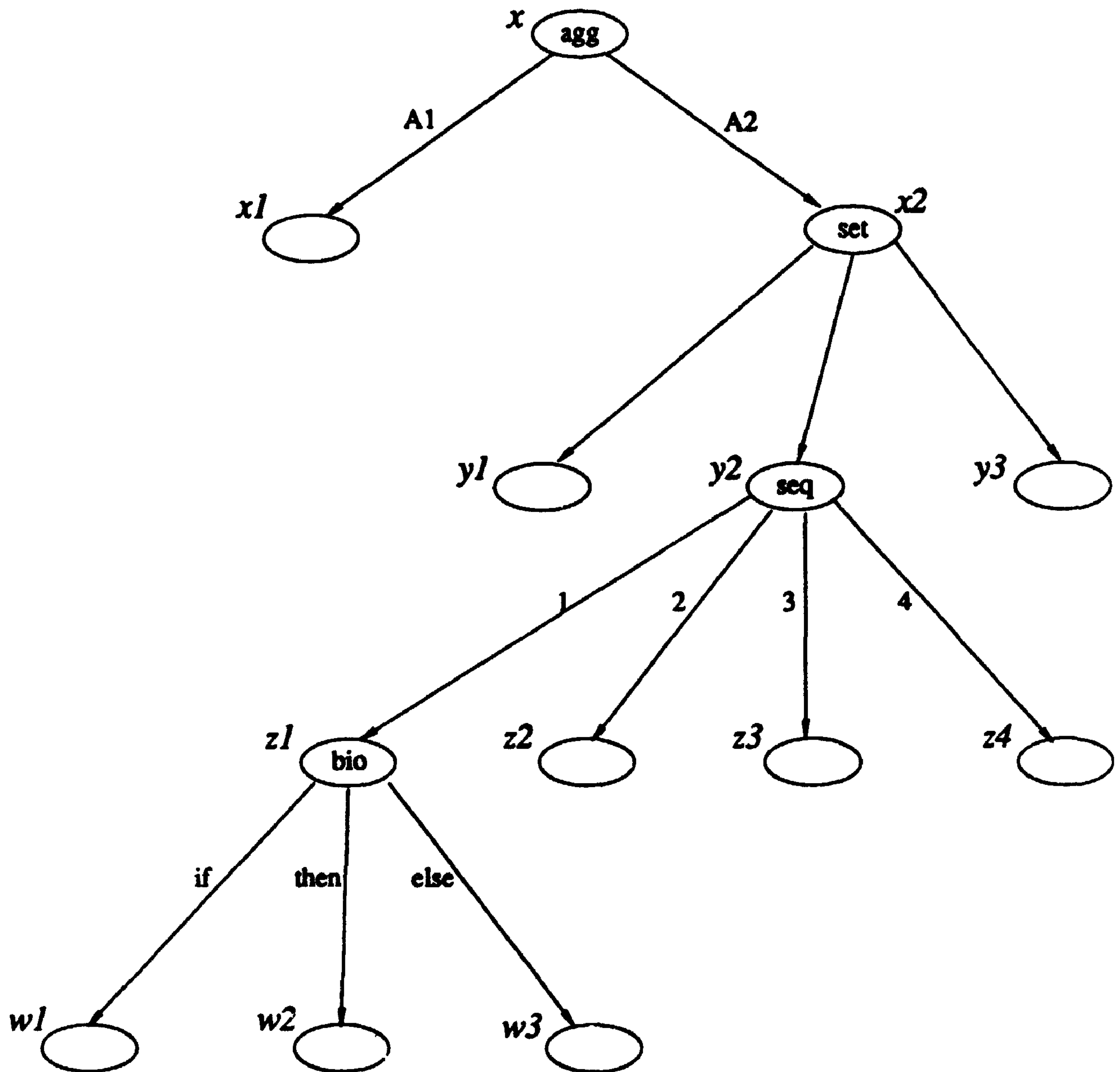


Figure 7.17: The Object Graph for Object x

is an AGG object and $o' \equiv o$ if x is a SET object, no matter what results are returned from other (if any) sub-message-sendings. Here o' cannot be a bottom object because o is not a bottom object. Therefore, from Theorem 11 we have $\varepsilon^*(\frac{x}{\ell} \leadsto o) \supset \varepsilon^*(\frac{y}{\ell} \leadsto o')$, no matter what the real evaluation of $\frac{x}{\ell} \leadsto o$ may be.

Case 2: Let x be a BIO object and $x = \{\!\{ y, y', y'' \}\!\}$. By definition of complex message-sendings where the invoker is a BIO object, $\frac{x}{\ell} \leadsto o$ will definitely cause either both $\frac{y}{\ell} \leadsto o$ and $\frac{y'}{\ell} \leadsto o$, or both $\frac{y}{\ell} \leadsto o$ and $\frac{y''}{\ell} \leadsto o$. Therefore, from Theorem 11, in either case we have $\varepsilon^*(\frac{x}{\ell} \leadsto o) \supset \varepsilon^*(\frac{y}{\ell} \leadsto o)$.

Case 3: Let x be a SEQ object and $x = <y, y_1, y_2, \ldots>$. By definition of complex message-sendings where the invoker is a SEQ object, $\frac{x}{\ell} \leadsto o$ will definitely cause $\frac{y}{\ell} \leadsto o$. In particular, $\varepsilon(\frac{x}{\ell} \leadsto o) = \{\frac{y}{\ell} \leadsto o\}$ when $\frac{y}{\ell} \leadsto o$ returns ***null*** as the result, because all subsequent sub-message-sendings will have ***null*** as the receiver. Therefore, we can only guarantee that we always have $\varepsilon^*(\frac{x}{\ell} \leadsto o) \supset \varepsilon^*(\frac{y}{\ell} \leadsto o)$.

Now hypothesize that the theorem holds for $n = k-1$. Consider the case when $n = k$. That is, there is an α-path $x_1 x_2 \cdots x_{k-1} x_k y$ from object x_1 (which is x) to object y. This implies that $x_1 x_2 \cdots x_{k-1} x_k$ is an α-path from x_1 to x_k with length $k-1$ and $x_k y$ is an α-path from x_k to y with length 1. By the induction hypothesis, we have $\varepsilon^*(\frac{x_1}{\ell} \leadsto o) \supset \varepsilon^*(\frac{x_k}{\ell} \leadsto o'')$ where $o'' \equiv o$ or $o'' \asymp o$, and $\varepsilon^*(\frac{x_k}{\ell} \leadsto o'') \supset \varepsilon^*(\frac{y}{\ell} \leadsto o')$ where $o' \equiv o''$ or $o' \asymp o''$. It is thus immediate that $\varepsilon^*(\frac{x}{\ell} \leadsto o) \supset \varepsilon^*(\frac{y}{\ell} \leadsto o')$ where $o' \equiv o$ or $o' \asymp o$. Hence, the theorem holds for all $n \geq 1$. ■

This theorem implies that we can predict what will happen *unavoidably* in a complex message-sending, by looking at the structure of the invoker without actually evaluating the message-sending. For example, consider the object graph in Figure 7.17. Since there is an α-path from x to w_1, we can confidently predict that if we sent object x to a non-bottom receiver o, i.e., $\frac{x}{\ell} \leadsto o$, then a message-sending $\frac{w_1}{\ell} \leadsto o'$ will *unavoidably* take place on a receiver o' which is either identical or equal to the original receiver o.

Now, assume that objects w_2 and w_3 shown in Figure 7.17 are identical, say, $w_2 \equiv w_3 \equiv v$. Then, although there is not an α-path from object x to object v, $\frac{x}{\ell} \leadsto o$ will definitely cause a message-sending where v is the invoker. This is because $\frac{x}{\ell} \leadsto o$ will definitely cause w_1 to be an invoker, and therefore cause either w_2 or w_3 to be an invoker. This kind of situation can be formally generalized in the following definition.

Definition 56: *α-reachability.* Let x, y be two objects. Then object y is *α-reachable* from object x if and only if one of the following holds:

1. There exists an α-path from x to y.

2. There exists a BIO object $z = \{\!\{ z_1, y, y \}\!\}$ such that z is α-reachable from x.

3. There exists a BIO object $z = \{\!\{ z_1, z_2, z_3 \}\!\}$ such that z is α-reachable from x and that y is α-reachable from both z_2 and z_3.

Here, z, z_1, z_2, z_3 are any KBO objects. ∎

Example 37: Again consider the object graph for object x shown in Figure 7.17. Suppose w_2 and w_3 are two different set objects with the same value $\{v_1, v_2\}$. Then, both v_1 and v_2 are α-reachable from x. This is because (1) there is an α-path from x to z_1, (2) there is an α-path from w_2 to v_1 (and v_2), and (3) there is an α-path from w_3 to v_1 (and v_2). ∎

With the notion of α-reachability, we can further extend Theorem 12 by giving the following theorem.

Theorem 13: *Let x, y, o be any non-bottom objects. If y is α-reachable from x then*

$$\varepsilon^*(\frac{x}{\ell} \rightsquigarrow o) \supset \varepsilon^*(\frac{y}{\ell} \rightsquigarrow o')$$

where $o' \equiv o$ or $o' \asymp o$, for any complex message-sending $\frac{x}{\ell} \rightsquigarrow o$.

Proof: This theorem can be proved by enumerating all cases in which y is α-reachable from x. We have the following three cases in which y can be α-reachable from x:

1. There is an α-path from x to y. From Theorem 12, it is immediate that $\varepsilon^*(\frac{x}{\ell} \rightsquigarrow o) \supset \varepsilon^*(\frac{y}{\ell} \rightsquigarrow o')$ where $o' \equiv o$ or $o' \asymp o$.

2. There is a BIO object z, $z = \{\!\{ z_1, y, y \}\!\}$, such that z is α-reachable from x. Thus if $\varepsilon^*(\frac{x}{\ell} \rightsquigarrow o) \supset \varepsilon^*(\frac{z}{\ell} \rightsquigarrow o')$ where $o' \equiv o$ or $o' \asymp o$, then we also have $\varepsilon^*(\frac{x}{\ell} \rightsquigarrow o) \supset \varepsilon^*(\frac{y}{\ell} \rightsquigarrow o')$ because $\frac{z}{\ell} \rightsquigarrow o'$ will definitely cause $\frac{y}{\ell} \rightsquigarrow o'$ either in the "then" case or in the "else" case.

3. There is a BIO object z, $z = \{\!\{ z_1, z_2, z_3 \}\!\}$, such that z is α-reachable from x and y is α-reachable from both z_2 and z_3. Thus if $\varepsilon^*(\frac{x}{\ell} \rightsquigarrow o) \supset \varepsilon^*(\frac{z}{\ell} \rightsquigarrow o'')$ where $o'' \equiv o$ or $o'' \asymp o$, then we also have either $\varepsilon^*(\frac{x}{\ell} \rightsquigarrow o) \supset \varepsilon^*(\frac{z_2}{\ell} \rightsquigarrow o'')$ or $\varepsilon^*(\frac{x}{\ell} \rightsquigarrow o) \supset \varepsilon^*(\frac{z_3}{\ell} \rightsquigarrow o'')$ because (by definition) complex message-sending

$\frac{x}{\ell} \leadsto o''$ will cause for sure either $\frac{z_2}{\ell} \leadsto o''$ or $\frac{z_3}{\ell} \leadsto o''$ as one of its sub-message-sendings. Furthermore, for any receiver o'', if $\varepsilon^*(\frac{z_2}{\ell} \leadsto o'') \supset \varepsilon^*(\frac{y}{\ell} \leadsto o')$ and $\varepsilon^*(\frac{z_3}{\ell} \leadsto o'') \supset \varepsilon^*(\frac{y}{\ell} \leadsto o')$ where $o' \equiv o''$ or $o' \asymp o''$, then $\varepsilon^*(\frac{x}{\ell} \leadsto o) \supset \varepsilon^*(\frac{y}{\ell} \leadsto o')$ where $o' \equiv o$ or $o' \asymp o$, since we have either $\varepsilon^*(\frac{x}{\ell} \leadsto o) \supset \varepsilon^*(\frac{z_2}{\ell} \leadsto o'')$ or $\varepsilon^*(\frac{x}{\ell} \leadsto o) \supset \varepsilon^*(\frac{z_3}{\ell} \leadsto o'')$.

■

The notion of α-reachability allows us to define formally what we call "necessarily unsafe invokers". Any objects, that are intended to be an invoker for a complex message-sending, should not be allowed to be a necessarily unsafe invoker.

Definition 57: *Necessarily Unsafe Invokers.* A complex object s is a *necessarily unsafe invoker* if one of the following holds:

1. s is α-reachable from s.
2. $s = [A_1: s_1, \ldots, A_n: s_n]$ where some of $s_1, \ldots, s_n$ is a necessarily unsafe invoker.
3. $s = \{\!\!\{ s_1, s_2, s_3 \}\!\!\}$ where s_1 is a necessarily unsafe invoker.
4. $s = \{\!\!\{ s_1, s_2, s_3 \}\!\!\}$ where both s_2 and s_3 are a necessarily unsafe invoker.
5. $s = \{s_1, \ldots, s_n\}$ where some of $s_1, \ldots, s_n$ is a necessarily unsafe invoker.
6. $s = < s_1, \ldots, s_n >$ where at least s_1 is a necessarily unsafe invoker.

■

Example 38: The following object *ComeOn* is a necessarily unsafe invoker:

$$ComeOn = \{\text{"}Dance\text{"}, \text{"}Drink\ beer\text{"}, ComeOn\}$$

because object *ComeOn* is α-reachable from itself. ■

Example 39: The following object s is a necessarily unsafe invoker:

$$s = [A_1: s_1 . A_2: s_2]$$
$$s_2 = < s_2, s_3, s_4 >$$

because object s_2 is a necessarily unsafe invoker. ■

Example 40: The following object s is a necessarily unsafe invoker:

$$s = \{s_1, s_2, s_3\}$$
$$s_3 = \{\!\!\{ s_4, s_5, s_6 \}\!\!\}$$
$$s_4 = \{\!\!\{ s_7, s_8, s_9 \}\!\!\}$$
$$s_8 = [A_1: s_3]$$

$s_9 = < s_3, s_{10} >$

because object s_3 is a necessarily unsafe invoker. ■

Now, the definition of necessarily unsafe complex message-sendings is quite straightforward:

Definition 58: *Necessarily Unsafe Complex Message-Sendings.* Let s, r be any two non-bottom objects. For any complex message-sending $\mu = (\frac{s}{\ell} \rightsquigarrow r)$, μ is a ***necessarily unsafe complex message-sending*** if and only if the invoker s is a necessarily unsafe invoker. ■

In the following, we establish the relationship between necessarily unsafe complex message-sendings and nonterminating complex message-sendings. Notice that the reverse relationship is not true.

Theorem 14: *Let s, r be any non-bottom objects. A complex message-sending $\mu = (\frac{s}{\ell} \rightsquigarrow r)$ is a nonterminating complex message-sending if μ is a necessarily unsafe message-sending.*

Proof: We shall show that $|\varepsilon^*(\frac{s}{\ell} \rightsquigarrow r)| = \infty$ always holds when the invoker s is a necessary unsafe invoker. We proceed by enumerating all cases in which an invoker can be a necessarily unsafe invoker. To help with the proof, we need some notation: if $\frac{s}{\ell} \rightsquigarrow r$ is a complex message-sending then $\frac{s^k}{\ell} \rightsquigarrow o$, where $k \geq 1$ and o is any receiver, denotes the k-th time the original invoker s appears again as the invoker in a message-sending in $\varepsilon^*(\frac{s}{\ell} \rightsquigarrow r)$ with the form $\frac{s}{\ell} \rightsquigarrow o$. Obviously, the first time s appears is in the original message-sending $\frac{s}{\ell} \rightsquigarrow r$. Notice that when $\frac{s^1}{\ell} \rightsquigarrow o_1$ causes, say, two other message-sendings with the forms $\frac{s^2}{\ell} \rightsquigarrow o_2$ and $\frac{s^3}{\ell} \rightsquigarrow o_3$, all these "other" message-sendings are considered different message-sendings even if o_1, o_2, o_3 are identical, because they take place at different time points.

Case 1: Let s be α-reachable from s. Let r_1 be r. From Theorem 13, we immediately have $\varepsilon^*(\frac{s^1}{\ell} \rightsquigarrow r_1) \supset \varepsilon^*(\frac{s^2}{\ell} \rightsquigarrow r_2)$ where $r_2 \equiv r_1$ or $r_2 \asymp r_1$. Similarly, we can have $\varepsilon^*(\frac{s^2}{\ell} \rightsquigarrow r_2) \supset \varepsilon^*(\frac{s^3}{\ell} \rightsquigarrow r_3)$, $\varepsilon^*(\frac{s^3}{\ell} \rightsquigarrow r_3) \supset \varepsilon^*(\frac{s^4}{\ell} \rightsquigarrow r_4)$, etc. Now, since there is always at least one message-sending $\frac{s^{k+1}}{\ell} \rightsquigarrow r_{k+1}$ in $\varepsilon^*(\frac{s^k}{\ell} \rightsquigarrow r_k)$, where $r_{k+1} \equiv r_k$ or $r_{k+1} \asymp r_k$, we have $\varepsilon^*(\frac{s^k}{\ell} \rightsquigarrow r_k) \neq \emptyset$ for all $k \geq 1$. Hence, $|\varepsilon^*(\frac{s^1}{\ell} \rightsquigarrow r_1)| = |\varepsilon^*(\frac{s}{\ell} \rightsquigarrow r)| = \infty$.

Case 2: Let $s = [A_1\colon s_1, \ldots, A_n\colon s_n]$ where at least one of $s_1, \ldots, s_n$ is a necessarily unsafe invoker. Without loss in generality we can let s_j, where $1 \leq j \leq n$, be the necessarily unsafe invoker. By definition of complex message-sendings, the message-sending $\frac{s}{\ell} \rightsquigarrow r$ will always cause the sub-message-sending $\frac{s_j}{\ell} \rightsquigarrow r'$ where $r' \asymp r$. Immediately, we have $\varepsilon(\frac{s}{\ell} \rightsquigarrow r) \supset \varepsilon(\frac{s_j}{\ell} \rightsquigarrow r')$ and therefore

$\varepsilon^*(\frac{s}{\ell} \rightsquigarrow r) \supset \varepsilon^*(\frac{s_2}{\ell} \rightsquigarrow r')$. Hence, if $|\varepsilon^*(\frac{s_2}{\ell} \rightsquigarrow r')| = \infty$ then $|\varepsilon^*(\frac{s}{\ell} \rightsquigarrow r)| = \infty$.

Case 3: Let $s = \{\!\{ s_1, s_2, s_3 \}\!\}$ where s_1 is a necessarily unsafe invoker. By definition of complex message-sendings, message-sending $\frac{s}{\ell} \rightsquigarrow r$ will cause either sub-message-sendings $\frac{s_1}{\ell} \rightsquigarrow r$ and $\frac{s_2}{\ell} \rightsquigarrow r$, or sub-message-sendings $\frac{s_1}{\ell} \rightsquigarrow r$ and $\frac{s_3}{\ell} \rightsquigarrow r$. Therefore, no matter how the message-sending $\frac{s}{\ell} \rightsquigarrow r$ is actually evaluated, we always have $\varepsilon(\frac{s}{\ell} \rightsquigarrow r) \supset \varepsilon(\frac{s_1}{\ell} \rightsquigarrow r)$, which implies $\varepsilon^*(\frac{s}{\ell} \rightsquigarrow r) \supset \varepsilon^*(\frac{s_1}{\ell} \rightsquigarrow r)$. Hence, if $|\varepsilon^*(\frac{s_1}{\ell} \rightsquigarrow r)| = \infty$ then $|\varepsilon^*(\frac{s}{\ell} \rightsquigarrow r)| = \infty$.

Case 4: Let $s = \{\!\{ s_1, s_2, s_3 \}\!\}$ where both s_2 and s_3 are necessarily unsafe invokers. By definition of complex message-sendings, the message-sending $\frac{s}{\ell} \rightsquigarrow r$ will definitely cause either the sub-message-sending $\frac{s_2}{\ell} \rightsquigarrow r$ or the sub-message-sending $\frac{s_3}{\ell} \rightsquigarrow r$. We thus have either $\varepsilon(\frac{s}{\ell} \rightsquigarrow r) \supset \varepsilon(\frac{s_2}{\ell} \rightsquigarrow r)$ or $\varepsilon(\frac{s}{\ell} \rightsquigarrow r) \supset \varepsilon(\frac{s_3}{\ell} \rightsquigarrow r)$, and therefore $\varepsilon^*(\frac{s}{\ell} \rightsquigarrow r) \supset \varepsilon^*(\frac{s_2}{\ell} \rightsquigarrow r)$ or $\varepsilon^*(\frac{s}{\ell} \rightsquigarrow r) \supset \varepsilon^*(\frac{s_3}{\ell} \rightsquigarrow r)$. Hence, if both $|\varepsilon^*(\frac{s_2}{\ell} \rightsquigarrow r)| = \infty$ and $|\varepsilon^*(\frac{s_3}{\ell} \rightsquigarrow r)| = \infty$, then, in either case, $|\varepsilon^*(\frac{s}{\ell} \rightsquigarrow r)| = \infty$.

Case 5: Let $s = \{s_1, \ldots, s_n\}$ where at least one of $s_1, \ldots, s_n$ is a necessarily unsafe invoker. Without loss in generality we can let s_j, where $1 \leq j \leq n$, be the necessarily unsafe invoker. By definition of complex message-sendings, the message-sending $\frac{s}{\ell} \rightsquigarrow r$ will always cause the sub-message-sending $\frac{s_j}{\ell} \rightsquigarrow r$, where the receiver r may (or may not) have been affected by some of the other sub-message-sendings, i.e., $\frac{s_1}{\ell} \rightsquigarrow r, \ldots, \frac{s_{j-1}}{\ell} \rightsquigarrow r, \frac{s_{j+1}}{\ell} \rightsquigarrow r, \ldots, \frac{s_n}{\ell} \rightsquigarrow r$. Therefore, it always holds that $\varepsilon(\frac{s}{\ell} \rightsquigarrow r) \supset \varepsilon(\frac{s_j}{\ell} \rightsquigarrow r)$, which implies $\varepsilon^*(\frac{s}{\ell} \rightsquigarrow r) \supset \varepsilon^*(\frac{s_j}{\ell} \rightsquigarrow r)$. Hence, if $|\varepsilon^*(\frac{s_j}{\ell} \rightsquigarrow r)| = \infty$ then $|\varepsilon^*(\frac{s}{\ell} \rightsquigarrow r)| = \infty$.

Case 6: Let $s = < s_1, s_2, \ldots, s_n >$ where at least s_1 is a necessarily unsafe invoker. By definition of complex message-sendings, message-sending $\frac{s}{\ell} \rightsquigarrow r$ will always cause the first sub-message-sending $\frac{s_1}{\ell} \rightsquigarrow r$. It should be noted that the sub-message-sending $\frac{s_2}{\ell} \rightsquigarrow r'$, where r' is the result of $\frac{s_1}{\ell} \rightsquigarrow r$, may not be included in $\varepsilon(\frac{s}{\ell} \rightsquigarrow r)$, because it is possible to have $r' = \textbf{\textit{null}}$ (it is allowable for a knowhow to return ***null***, like the functions in C, which represents a non-informative or unknown datum). However, it is always true that $\varepsilon(\frac{s}{\ell} \rightsquigarrow r) \supset \varepsilon(\frac{s_1}{\ell} \rightsquigarrow r)$, and therefore $\varepsilon^*(\frac{s}{\ell} \rightsquigarrow r) \supset \varepsilon^*(\frac{s_1}{\ell} \rightsquigarrow r)$. Hence, if $|\varepsilon^*(\frac{s_1}{\ell} \rightsquigarrow r)| = \infty$ then $|\varepsilon^*(\frac{s}{\ell} \rightsquigarrow r)| = \infty$.

The above theorem is one of the main results of this chapter. It provides the basis for development of techniques that can predict unavoidable nontermination of a complex message-sending.

7.4 Potentially Unsafe Complex Message-sendings

In this section, we study the properties of complex message-sendings which have the potential to be nonterminating on any non-bottom object (receiver). In other words, they may or may not terminate. Our intention in detecting this kind of complex message-sendings is to offer the ability (if desired) to raise a warning about them before their evaluation, and thus let end-users decide whether to reject, modify or simply carry on with the message-sending.

Definition 59: *Potentially Unsafe Invokers.* A complex object s is considered a ***potentially unsafe invoker*** if (1) $oid(s) \in orf(s)$ or (2) there exists $oid(x) \in orf(s)$ such that x is a potential unsafe invoker. ■

Notice that this is a recursive definition.

It can be observed that if object x is a potentially unsafe invoker then the object graph of x contains at least one cycle. With the notion of potentially unsafe invokers, it is straightforward to define the notion of potentially unsafe complex message-sendings.

Definition 60: *Potentially Unsafe Complex Message-Sendings.* Let s, r be any non-bottom objects. For any complex message-sending $\mu = (\frac{s}{\ell} \rightsquigarrow r)$, μ is a ***potentially unsafe complex message-sending*** if s is a potentially unsafe invoker. ■

Example 41: Let $s = < s_1, s, s_2 >$. Then, for any non-bottom receiver r, the complex message-sending $\frac{s}{\ell} \rightsquigarrow r$ is a potentially unsafe complex message-sending. More precisely, $\frac{s}{\ell} \rightsquigarrow r$ will cause the following message-sendings (in sequential order): $\frac{s_1}{\ell} \rightsquigarrow r \Rightarrow result_1$, $\frac{s}{\ell} \rightsquigarrow result_1$, $\frac{s_1}{\ell} \rightsquigarrow result_1 \Rightarrow result_2$, $\frac{s}{\ell} \rightsquigarrow result_2$, $\frac{s_1}{\ell} \rightsquigarrow result_2 \Rightarrow result_3, \ldots\ldots$, where $result_i$ $(i \geq 1)$ is not a bottom object. ■

Example 42: Let Foo denote a BIO object, $Foo = \{ Foo_1, \mathbf{null}, Foo \}$. And let Foo_1 denote a SEQ object, $Foo_1 = <$ "*Activate*", "*Deactivate*", "*Is active?*" $>$. Let *FooWindow* be an instance of WINDOW. Then, the complex message-sending $Foo \rightsquigarrow FooWindow$ is a potentially unsafe complex message-sending. In fact, if behaviors *"Activate"*, *"Deactivate"* and *"Is active?"* are correctly implemented for class WINDOW, message-sending $Foo \rightsquigarrow FooWindow$ will never terminate because $Foo_1 \rightsquigarrow FooWindow$ will always returns **False** and therefore $Foo \rightsquigarrow FooWindow$

will always be repeated. Notice, however, that this message-sending is not a necessarily unsafe message-sending since the invoker *Foo* is not a necessarily unsafe invoker. ■

We have introduced the notions of nonterminating, necessarily unsafe, and potentially unsafe complex message-sendings. The relationship among them can be formally established through the following theorem:

Theorem 15: *Let* $M_{non} = \{\mu | \ \mu$ *is a nonterminating complex message-sending* $\}$
$M_{pot} = \{\mu | \ \mu$ *is a potentially unsafe complex message-sending* $\}$
$M_{nec} = \{\mu | \ \mu$ *is a necessarily unsafe complex message-sending* $\}$

Then,

$$M_{nec} \subset M_{non} \subset M_{pot}$$

Proof:

1. We first show $M_{nec} \subset M_{non}$. In other words, we have to show that every necessarily unsafe message-sending is a nonterminating message-sending, and that there are some nonterminating message-sendings which are not necessarily unsafe message-sendings. The former is the result of Theorem 14. To show the latter holds, we can again give a trivial example. Let $Foo = \{\!\!\{ \textbf{\textit{null}}, Foo, \textbf{\textit{same}} \}\!\!\}$ be a bi-object, which is not a necessarily unsafe invoker by our definition because the else-part (i.e., ***same***) does not reference *Foo*. (Note if $Foo = \{\!\!\{ \textbf{\textit{null}}, Foo, Foo \}\!\!\}$ or $Foo = \{\!\!\{ Foo, \textbf{\textit{null}}, \textbf{\textit{null}} \}\!\!\}$ then *Foo* is a necessarily unsafe invoker.) However, for any non-bottom receiver o, $Foo \rightsquigarrow o$ is a nonterminating message-sending because it always causes itself as a sub-message-sending. More precisely, $Foo \rightsquigarrow o$ causes $\textbf{\textit{null}} \rightsquigarrow o \Rightarrow \textbf{\textit{null}}$, and then since $\textbf{\textit{null}} \notin \{\textbf{False}, \textbf{\textit{fail}}\}$ we will see $Foo \rightsquigarrow o$ occurring again and again, indefinitely.

2. We now show $M_{non} \subset M_{pot}$. In other words, we have to show (1) every nonterminating message-sending is a potentially unsafe message-sending, and (2) there are some potentially unsafe message-sendings which are not nonterminating message-sendings. To show (1) holds, hypothesize that μ is a nonterminating message-sending but not a potentially unsafe message-sending. This implies that the invoker in μ is not a potentially unsafe invoker, that is,

$$\forall x: oid(x) \in orf(invoker(\mu)) \cup \{oid(invoker(\mu))\} \Longrightarrow oid(x) \notin orf(x)$$

where $invoker(\mu)$ denotes the invoker in the message-sending μ. This means that no message-sending μ', where $\mu' \in \epsilon^*(\mu)$ or $\mu' = \mu$, can cause another

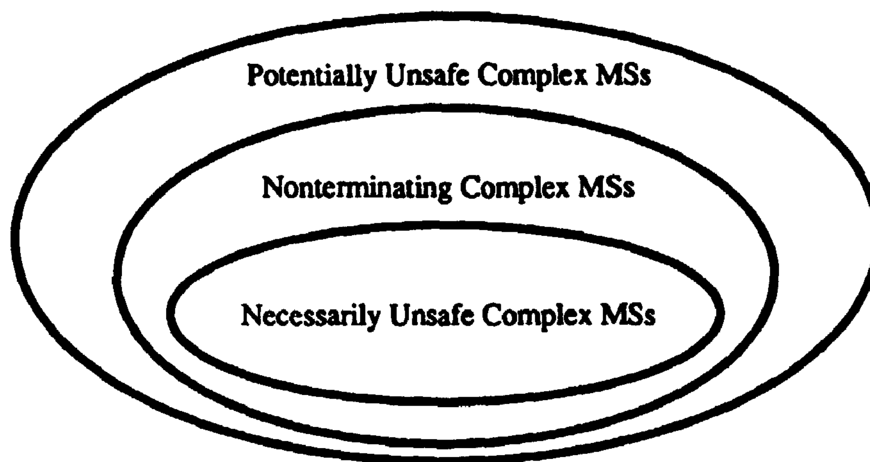


Figure 7.18: The Set of All Potentially Unsafe Complex Message-Sendings

message-sending μ'' such that $invoker(\mu') \equiv invoker(\mu'')$. Furthermore, object $invoker(\mu)$ or any complex object referenced in $invoker(\mu)$ can only reference a finite set of other objects (i.e., sub-objects), because objects in a KBO database are a consistent set of KBO objects. Therefore, all the other complex message-sendings caused by μ will eventually cause one or more simple message-sendings. These simple message-sendings either fail and return ***fail***, or invoke a behavior and execute a knowhow which is bound to terminate (again, due to the consistency of a KBO database). This implies that $\varepsilon^*(\mu)$ is a finite set of message-sendings and, hence, $|\varepsilon^*(\mu)| \neq \infty$. This contradicts to the hypothesis that μ is a nonterminating message-sending. Therefore, a nonterminating message-sending must be a potentially unsafe message-sending.

To show (2) holds, we can simply give a trivial example. Let *Foo* denote a bi-object, $Foo = \langle\!\langle\ \mathbf{null}, \mathbf{same}, Foo\ \rangle\!\rangle$, which is obviously a potentially unsafe invoker because *Foo* is reachable from *Foo*. However, for any receiver o, $Foo \rightsquigarrow o$ is not a nonterminating message-sending because it always returns the receiver o itself. More precisely, $Foo \rightsquigarrow o$ causes $\mathbf{null} \rightsquigarrow o \Rightarrow \mathbf{null}$, and then since $\mathbf{null} \notin \{\text{False}, \mathbf{fail}\}$ we will have $\mathbf{same} \rightsquigarrow o \Rightarrow o$.

The above theorem may be illustrated by Figure 7.18. As a conclusion, from a practical point of view we can choose one of the following alternatives to prevent nontermination caused by complex message-sendings: (1) We can impose constraints that restrict KBO complex objects to be acyclic; or (2) We can allow KBO complex

objects to be cyclic but check (and thus eliminate) necessarily unsafe invokes from complex message-sendings. The first approach provides less expressive power of KBO complex objects, but guarantees the termination of complex message-sendings. The second approach provides more expressive power of KBO complex objects (i.e., they can simulate certain recursive control structures), but requires run-time (or compile-time, if the invoker is a constant) detection. Moreover, the second approach still accepts potentially unsafe complex message-sendings. Hence one has to choose between limiting the expressive power of KBO complex objects, or living with possibly nonterminating complex message-sendings. It seems, however, that if a potentially unsafe complex message-sending, which is not a necessarily unsafe complex message-sending, happens to be nonterminating, then in most cases it is a rare mistake because practically it does not make any sense (as shown in some of our previous examples). If the second approach is adopted, then Theorem 14 provides a formal basis for the development of a nontermination detector.

7.5 Strictly-Typed Complex Message-Sendings

In this section, we study the success properties of complex message-sendings. In other words, we are interested in the circumstances in which a complex message-sending will be guaranteed to return a result other than ***fail***. In this way, we may be able to predict that a complex message-sending is *definitely* not an anomalous complex message-sending before its evaluation.

Our approach is to provide some success criteria in the form of inference rules, which can be used to prevent some complex message-sendings from occurring while allowing others. In a conventional sense, these rules can be called a "type-checking system", and, in our context, it is also a *dynamic type-checking system* since we have to know the actual object-identity of the invoker in a message-sending. We feel that such a system is practically feasible as well as desirable, since in emerging database applications like engineering designs, such a checking process seems to be always much cheaper than the actual evaluation of a complex message-sending. For example, for a complex message-sending like the following

$$\{\text{"}Rotate\text{"}, \text{"}Reverse\ color\ map\text{"}\} \rightsquigarrow AirPlaneDesignWindow$$

checking whether *both* messages *"Rotate"* and *"Reverse color map"* contained in the set invoker can be understood by the receiver *AirPlaneDesignWindow* of DESIGN-WINDOW (i.e., Does class DESIGN-WINDOW have two behaviors with values *"Rotate"* and *"Reverse color map"*?) is obviously much faster (in fact, such checking time

can almost be ignored) than the actual successful evaluation of this message-sending (i.e., executing two knowhows, one to rotate *AirPlaneDesignWindow* and one to reverse the color map in *AirPlaneDesignWindow*).

In the sequel, we shall restrict our attention only to complex message-sendings that are not nonterminating complex message-sendings (they can be eliminated based on the mechanisms described previously in this chapter).

Definition 61: *Dynamic Typings.* The complete set of typing rules for complex message-sendings is defined by a deduction system $\vdash$ for ***dynamic typings*** of the form

$$(\frac{s}{\ell} \leadsto r) :: c$$

which can be read as ***"a message-sending*** $(\frac{s}{\ell} \leadsto r)$ ***is typed by a result class c".*** ■ The deduction system $\vdash$ will be given in the next definition. Notice that this is dynamic typing, in that it has to refer to the identity of the actual invoker s at run-time. However, it can also be used as a static typing system when the invoker s is expressed as a constant (as opposed to an untyped object variable).

For the following definition, we shall let C_{kernel} denote the set containing these kernel classes: OBJECT, CAPSULE, AGG, BIO, SET, and SEQ. We also let o_c denote any object such that $class(o_c) = c$, that is, object o_c is a direct instance of class c. Moreover, recall that $args(\ell)$ denotes the set of all label-argument pairs appearing on the blackboard ℓ.

Definition 62: *Strictly-Typed Complex Message-Sendings.* A complex message-sending is a ***strictly-typed*** complex message-sending if and only if it is typed by a result class derivable from the following inference rules (the long horizontal line is read "implies"):

RULE 1: *null* As Receiver

$$\vdash (\frac{s}{\ell} \leadsto \textbf{\textit{null}}) :: \text{BOTTOM}$$

RULE 2: *null* As Invoker

$$\vdash (\frac{\textbf{\textit{null}}}{\ell} \leadsto r) :: \text{BOTTOM}$$

RULE 3: *same* As Receiver

$$\vdash (\frac{s}{\ell} \leadsto \textbf{\textit{same}}) :: \text{BOTTOM}$$

RULE 4: *same* As Invoker

$$\vdash (\frac{same}{\ell} \rightsquigarrow r) \ :: \ class(r)$$

RULE 5: Monomorphic Invocation

$$\vdash (\frac{s}{\ell} \rightsquigarrow r) \ :: \ c$$

where $s \in \dot{\Sigma}(class(r))$, $sig(s) = () \longrightarrow c$, or $sig(s) = (L_1, c_1) \times \cdots \times (L_n, c_n) \longrightarrow c$ and $args(\ell) \supseteq \{(L_1, a_1), \ldots, (L_n, a_n)\}$ such that $a_i \textcircled{i} c_i$ for $1 \le i \le n$.

RULE 6: Polymorphic Invocation

$$\vdash (\frac{s}{\ell} \rightsquigarrow r) \ :: \ c$$

where $s \notin \dot{\Sigma}(class(r))$ but $\exists s' \in \mathring{\Sigma}(class(r)) : s' \asymp s$, $sig(s') = () \longrightarrow c$, or $sig(s') = (L_1, c_1) \times \cdots \times (L_n, c_n) \longrightarrow c$ and $args(\ell) \supseteq \{(L_1, a_1), \ldots, (L_n, a_n)\}$ such that $a_i \textcircled{i} c_i$ for $1 \le i \le n$.

RULE 7: Translating SELF

$$\frac{\vdash (\frac{s}{\ell} \rightsquigarrow r) \ :: \ \text{SELF}}{\vdash (\frac{s}{\ell} \rightsquigarrow r) \ :: \ class(r)}$$

RULE 8: Substitutability

$$\frac{\vdash (\frac{s}{\ell} \rightsquigarrow r) \ :: \ c_1 \qquad c_1 \textcircled{k} c_2 \qquad c_2 \notin C_{kernel}}{\vdash (\frac{s}{\ell} \rightsquigarrow r) \ :: \ c_2}$$

RULE 9: AGG Invokers

$$\frac{\vdash (\frac{s_1}{\ell} \rightsquigarrow r) \ :: \ c_1 \qquad \cdots\cdots \qquad \vdash (\frac{s_n}{\ell} \rightsquigarrow r) \ :: \ c_n}{\vdash (\frac{[A_1 : s_1, \ldots, A_n : s_n]}{\ell} \rightsquigarrow r) \ :: \ c_{tmp}}$$

where $c_{tmp} \textcircled{k}_d$ AGG such that $\dot{\Xi}(c_{tmp}) = [A_1 \colon c_1, \ldots, A_n \colon c_n]$ and $\dot{\Sigma}(c_{tmp}) = \{b_1, \ldots, b_n\}$ with $value(b_i) =$ "*get* A_i" and $sig(b_i) = () \longrightarrow c_i$ for $1 \le i \le n$.

RULE 10: Then-Recursive BIO Invokers

$$\frac{\vdash (\frac{s_1}{\ell} \rightsquigarrow r) \ :: \ c \qquad s = \langle s_1, s, s_3 \rangle}{\vdash (\frac{s}{\ell} \rightsquigarrow r) \ :: \ c}$$

RULE 11: Else-Recursive BIO Invokers

$$\frac{\vdash (\frac{s_2}{\ell} \leadsto r) :: c \qquad s = \{\!\!\{ s_1, s_2, s \}\!\!\}}{\vdash (\frac{s}{\ell} \leadsto r) :: c}$$

RULE 12: Non-Recursive BIO Invokers

$$\frac{\vdash (\frac{s_2}{\ell} \leadsto r) :: c \quad \vdash (\frac{s_3}{\ell} \leadsto r) :: c \quad s_2 \not\equiv s \quad s_3 \not\equiv s \quad s = \{\!\!\{ s_1, s_2, s_3 \}\!\!\}}{\vdash (\frac{s}{\ell} \leadsto r) :: c}$$

RULE 13: SET Invokers

$$\frac{\vdash (\frac{s_1}{\ell} \leadsto r) :: c \qquad \cdots\cdots \qquad \vdash (\frac{s_n}{\ell} \leadsto r) :: c}{\vdash (\frac{\{s_1, \ldots, s_n\}}{\ell} \leadsto r) :: c}$$

RULE 14: SEQ Invokers

$$\frac{\vdash (\frac{s_1}{\ell} \leadsto r) :: c_1 \quad \vdash (\frac{s_2}{\ell} \leadsto o_{c_1}) :: c_2 \quad \cdots\cdots \quad \vdash (\frac{s_n}{\ell} \leadsto o_{c_{n-1}}) :: c_n}{\vdash (\frac{\langle s_1, \ldots, s_n \rangle}{\ell} \leadsto r) :: c_n}$$

RULE 1 (***null*** As Reccıver), RULE 2 (***null*** As Invoker), RULE 3 (***same*** As Receiver), and RULE 4 (***same*** As Invoker) are the basic judgments of the $\vdash$ system, and their assertions are immediately clear from the semantics defined for simple message-sendings. Note that the $\vdash$ system does not include rules for ***fail*** because it is meaningless for end-user to use ***fail*** as an invoker or receiver in any message-sending.

RULE 5 (Monomorphic Invocation) and RULE 6 (Polymorphic Invocation) are also the basic judgments of the $\vdash$ system. RULE 5 and RULE 6 assert that a simple message-sending that successfully invokes a behavior (and thus executes its knowhow) will have that behavior's formal output specification (i.e., the output class in the signature specified for that behavior) as the result class. One point should be noted here: The formal output specification of a behavior is the formal output specification of its knowhow if the behavior is a biological bearer. However, if the behavior is not a biological bearer (i.e., its knowhow is borrowed from another behavior), then its formal output specification may be changed to a more generalized class (this is guaranteed by the r-compatibility).

RULE 7 (Translating SELF) asserts that if a message-sending μ is typed by SELF (which may be used only as the formal output specification of a behavior) then μ is also typed by the class of the receiver in μ. This is also immediately clear, since SELF indicates that the message-sending will return the receiver itself (possibly with some side-effects). Hence the result class is the class of the receiver.

The assertion made in RULE 8 (Substitutability) is quite straightforward by definition of the Ⓚ relationship, except the constraint $c_2 \notin C_{kernel}$. We impose this constraint mainly for two practical reasons. First, kernel classes are introduced to facilitate the formal development of the KBO model (e.g., they are widely used in our formal definitions), and they have no *user-defined* behaviors (at least in the present framework). Note that users can always create other generic subclasses of kernel classes, such as MY-SET or MY-SEQ, and define behaviors for them (such as "Number of members" or "Add one member"). If desirable, these behaviors may even be implemented to return only ***null***, so that they can serve as an "abstract interface" for the subclasses to follow and to implement useful knowhows. Second, because the message-sending $\rightsquigarrow$ operation *does not* "penetrate" the structure of the receiver (i.e., the $\rightsquigarrow$ operation is only interested in the current receiver itself, not its components or members), no use can be made of the results of kernel classes by other message-sendings. As an example, the following complex message-sending

$$\{\text{"}How\ old\ are\ you?\text{"}, \text{"}Are\ you\ married?\text{"}\} \rightsquigarrow SomePerson$$

may return an INT object if *"How old are you?"* is last sent to *SomePerson*, or return a BOOL object if *"Are you married?"* is last sent to *SomePerson*. Therefore, without the constraint $c_2 \notin C_{kernel}$ in RULE 8, our best typing prediction for this complex message-sending would be CAPSULE, the superclass of INT and BOOL. Such typing information (i.e., CAPSULE as the result class for the above message-sending) is practically useless because CAPSULE itself does not have any user-defined behaviors. In other words, typings like CAPSULE become very misleading. Therefore, in order to be more practical, we have decided to simply let the type checker report an error in this kind of message-sending. It is for such a practical purpose that our typing system is defined to be more conservative, resulting in the incompleteness of the system. For example, the above message-sending does not return ***fail*** (it returns an INT object or a BOOL object); however, our typing system will prohibit this message-sending from occurring by reporting an error.

RULE 9 (AGG Invokers) describes the typing of a complex message-sending where the invoker is an aggregate object s with value $[A_1 : s_1, ..., A_n : s_n]$. If the typing c_i can be derived for each message-sending $\frac{s_i}{t} \rightsquigarrow r$, then we construct the typing c_{tmp} for $\frac{s}{t} \rightsquigarrow r$, which c_{tmp} is a direct subclass of AGG such that its local behaviors "*get* A_i" (attribute accessors) are automatically generated. Here, we assume that c_{tmp} is a system-generated, temporary class which, if desired, can be renamed and installed persistently in the database; once it is installed in the database, users may create other arbitrary behaviors for this class.

RULE 10 (Then-Recursive BIO Invokers) describes the typing for a complex message-sending where the invoker is a bi-object. If the invoker can repeat itself as the then-part, then the typing for this complex message-sending is the typing of sending the else-part to the receiver. RULE 11 (Else-Recursive BIO Invokers) makes the similar assertion except that the recursion can only happen at the else-part. Note that recursion cannot happen at both the then-part and the else-part; otherwise s would be α-reachable from s and thus $\frac{s}{t} \rightsquigarrow r$ would be a nonterminating message-sending. Furthermore, for RULE 10 and RULE 11, we implicitly assume that the if-part of s is well coded such that it semantically makes sense. For example, let *BecomeAdult* denote a bi-object:

$$BecomeAdult = \langle\, s_1, BecomeAdult, \text{“}\textit{Set adult status}\text{”} \,\rangle$$

where $s_1 = <$ "*Increase your age*", "*Are you a teenager?*" $>$, and suppose class PERSON does have three behaviors with values "*Increase your age*", "*Are you a teenager?*" and "*Set adult status*", and with signatures $() \longrightarrow$ SELF, $() \longrightarrow$ BOOL, and $() \longrightarrow$ SELF, respectively. Now, if the behavior with value "*Are you a teenager?*" has a knowhow which is implemented so that it is consistent with the behavior's intended semantics (i.e., return **True** if a person is under 18, and return **False** otherwise), then the complex message-sending

$$BecomeAdult \rightsquigarrow SomePerson$$

will eventually terminate and return the receiver *SomePerson* itself (who is now over 18 and has been given adult status). However, if the behavior "*Are you a teenager?*" was carelessly implemented such that it always returns **True** no mater what age a person is, then the complex message-sending *BecomeAdult* $\rightsquigarrow$ *SomePerson* can never terminate. This is because there is no chance for the else-part "*Set adult status*" to be sent to *SomePerson*. In fact, since we take a "black box" view of knowhow implementations, we have to assume that they are well coded and thoroughly tested by professional programmers.

RULE 12 (Non-Recursive BIO Invokers) describes how to derive the typing of a non-recursive complex message-sending from the then-part and the else-part of the invoker. If we can derive the same typing for both $\frac{s_2}{t} \rightsquigarrow r$ and $\frac{s_3}{t} \rightsquigarrow r$, then that typing is the typing for $\frac{s}{t} \rightsquigarrow r$. Note that the rule has nothing to do with the typing of $\frac{s_1}{t} \rightsquigarrow r$.

RULE 13 (SET Invokers) determines the typing of a complex message-sending where the invoker is a set object $s = \{s_1, \ldots, s_n\}$. The rule asserts that if we can

derive the same typing c for each $\frac{s_i}{\ell} \rightsquigarrow r$ for $1 \leq i \leq n$, then c is also the typing of $\frac{s}{\ell} \rightsquigarrow r$.

RULE 14 (SEQ Invokers) determines the typing of a complex message-sending where the invoker is a sequence object $s = < s_1, \ldots, s_n >$. The rule asserts that if the typing of $\frac{s_1}{\ell} \rightsquigarrow r$ is c_1 and the typing of $\frac{s_i}{\ell} \rightsquigarrow o_{c_{i-1}}$ is c_i for $2 \leq i \leq n$, then c_n is the typing of $\frac{s}{\ell} \rightsquigarrow r$. Here, each o_{c_i} denotes any object which is a direct instance of class c_i. It can be observed that, from RULE 1 to RULE 13, we need nothing from receiver r other than its class (i.e., in RULE 4, RULE 5, RULE 6 and RULE 7 we need to know $class(r)$) to derive the typing. Therefore, using variables o_{c_i} in RULE 14 does not affect the applicability of our typing system.

Since the above rules are mainly intended to be used as a dynamic typing system, the type checking for simple message-sendings are quite trivial. We gain little by run-time checking the success of simple message-sendings. More precisely, dynamically checking simple message-sendings is equivalent to actually evaluating them, in the sense that type checking takes place right before the simple message-sending occurs — if the checking succeeds then a knowhow is immediately executed; otherwise ***fail*** is returned. The true advantage of using such a dynamic typing system can only be appreciated when a complex message-sending is typechecked. Since we check the complex message-sending as a whole (instead of checking each sub-message-sending), either a set of knowhows will be executed (when the checking succeeds) or none of them will be executed (when the checking fails). In this way, there will be no undesirable affects on the receiver when the typechecker reports a failure (i.e., nothing has happened to the receiver yet!). In other words, ***our dynamic typing system is intended to protect the receiver from being affected by anomalous complex message-sendings.*** Let us illustrate this through an example.

Example 43: Assume that class WINDOW and class COLOR are two direct subclasses of AGG. Recall that class WINDOW (in Chapter 6) has a behavior with value *"Deactivate"* and signature () $\longrightarrow$ SELF, whose semantics is to de-activate a window. Suppose class COLOR has a behavior with value *"Is blue color?"* and signature () $\longrightarrow$ BOOL. Now, let *FooWindow* be an instance of class WINDOW, and let *Foo* be a sequence object

$$Foo = < \text{"}Deactivate\text{"}, \text{"}Is\ blue\ color?\text{"} >$$

Then, the following is an anomalous complex message-sending:

$$Foo \rightsquigarrow FooWindow$$

because, although the message-sending returns ***fail***, it already executed a knowhow

that de-activates the window *FooWindow*. This can be prevented from happening if we perform a typechecking on $Foo \leadsto FooWindow$ before it is evaluated. More precisely, using the $\vdash$ system, we can only deduce the following:

(using RULE 5 or RULE 6):	$\vdash$ ("*Deactivate*" $\leadsto$ *FooWindow*) :: SELF
(using RULE 7):	$\vdash$ ("*Deactivate*" $\leadsto$ *FooWindow*) :: WINDOW

Now, according to RULE 14, we must derive a typing for

$$\text{"}Is\ blue\ color?\text{"} \leadsto o_{WINDOW}$$

However, neither RULE 5 nor RULE 6 can obtain a typing for this message-sending. Nor can we apply RULE 8 to obtain a more general typing for

$$\text{"}Deactivate\text{"} \leadsto FooWindow$$

because WINDOW, its current typing, is already a direct subclass of AGG (a kernel class). That is, there is no class other than AGG which is more general than WINDOW. Therefore, no typing can be derived by the $\vdash$ system for

$$Foo \leadsto FooWindow$$

and a failure is thus reported. ■

7.6 Soundness of the $\vdash$ System

As we have seen, the $\vdash$ system is intended to predict whether a complex message-sending can return a meaningful result (i.e., an object other than ***fail***), without actually executing any knowhow. In this section, we shall discuss the soundness of the $\vdash$ system. The $\vdash$ system is *sound* if any message-sending that has a result class derivable from the $\vdash$ system will always return an instance of that result class.

Theorem 16: *The $\vdash$ system is sound. That is, for any message-sending $\frac{s}{\ell} \leadsto r$, any object o, and any class c, if $\frac{s}{\ell} \leadsto r \Rightarrow o$ and $\vdash (\frac{s}{\ell} \leadsto r)$ :: c, then o is an instance of class c.*

Proof: The theorem is proved by enumerating all cases in which a typing can be derived by the $\vdash$ system.

1. Let s be ***null*** or let r be ***null*** or ***same***. Then by definition of the $\leadsto$ operation, $\frac{\mathbf{null}}{\ell} \leadsto r \Rightarrow \mathbf{null}$, $\frac{s}{\ell} \leadsto \mathbf{null} \Rightarrow \mathbf{null}$, and $\frac{s}{\ell} \leadsto \mathbf{same} \Rightarrow \mathbf{same}$. Obviously,

their results are BOTTOM objects. On the other hand, their typings are derived directly by RULE 1, RULE 2 and RULE 3, respectively, as the class BOTTOM. Hence, the theorem holds in these three cases.

2. Let s be ***same***. Then by definition of the $\rightsquigarrow$ operation, $\frac{s}{\ell} \rightsquigarrow r \Rightarrow r$. On the other hand, the typing of $\frac{\mathbf{same}}{\ell} \rightsquigarrow r$ is derived directly by RULE 4, that is, $\vdash (\frac{\mathbf{null}}{\ell} \rightsquigarrow r) :: class(r)$. Hence, the theorem holds in this case because r is a (direct) instance of its own class $class(r)$.

3. Let s be a capsule or complex object such that s is identical to a (behavior) object in the interface of the class of the receiver r, i.e., $s \in \dot{\Sigma}(class(r))$. Let $sig(s) = (L_1, c_1) \times \cdots \times (L_n, c_n) \longrightarrow c$ where $n \geq 0$. Suppose $\vdash (\frac{s}{\ell} \rightsquigarrow r) :: c$. According to RULE 5, this implies that the blackboard ℓ provides all the labeled (legal) arguments needed (if any) by the execution of $kh(s)$. Now, suppose $\frac{s}{\ell} \rightsquigarrow r \Rightarrow o$. By the definition of simple message-sendings, if the blackboard ℓ provides all the labeled (legal) arguments needed (if any) by the execution of $kh(s)$, the result o must be an instance of class c, the formal output specification of behavior s.

4. Let s be a capsule or complex object such that s is equal, but not identical, to a (behavior) object in the value-based interface of the class of the receiver r, i.e., $s \asymp s'$ and $s' \in \mathring{\Sigma}(class(r))$. Let $sig(s') = (L_1, c_1) \times \cdots \times (L_n, c_n) \longrightarrow c$ where $n \geq 0$. Suppose $\vdash (\frac{s}{\ell} \rightsquigarrow r) :: c$. According to RULE 6, this implies that the blackboard ℓ provides all the labeled (legal) arguments needed (if any) by the execution of $kh(s')$. Now, suppose $\frac{s}{\ell} \rightsquigarrow r \Rightarrow o$. By the definition of simple message-sendings, if the blackboard ℓ provides all the labeled (legal) arguments needed (if any) by the execution of $kh(s')$, the result o must be an instance of class c, the formal output specification of behavior s'.

5. Let $\frac{s}{\ell} \rightsquigarrow r \Rightarrow o$ and $\vdash (\frac{s}{\ell} \rightsquigarrow r) :: class(r)$, where the typing $class(r)$ is derived using RULE 7. Since we have used RULE 7, we must have $\vdash (\frac{s}{\ell} \rightsquigarrow r) ::$ SELF, and o must be an instance of SELF, which in turn implies that $\frac{s}{\ell} \rightsquigarrow r$ returns the receiver r itself. It follows that o is the receiver r itself and therefore o is also an instance of $class(r)$.

6. Let $\frac{s}{\ell} \rightsquigarrow r \Rightarrow o$ and $\vdash (\frac{s}{\ell} \rightsquigarrow r) :: c$, where the typing c is derived using RULE 8. That is, there is another class c_1 such that $\vdash (\frac{s}{\ell} \rightsquigarrow r) :: c_1$ and c_1 Ⓘ c. Therefore, o is an instance of c_1. Since c_1 is also a subclass of c, by the definition of the Ⓘ relationship it is immediate that o is also an instance of c.

7. Let s have value $[A_1: s_1, \ldots, A_n: s_n]$, and let s be neither identical nor equal to any behavior of $class(r)$. Let $\frac{s}{\ell} \rightsquigarrow r \Rightarrow o$. From the definition of complex message-sendings, we know that o must be an aggregate object with value $[A_1: o_1, \ldots, A_n: o_n]$ where, for $1 \leq i \leq n$, o_i is the result of $\frac{s_i}{\ell} \rightsquigarrow r_i$, and $r_i \asymp r$ and $r_i \not\equiv r$ (by definition). Now, let $\vdash (\frac{s}{\ell} \rightsquigarrow r) :: c$. From RULE 9 we know that c is a direct subclass of AGG and $\dot{\Xi}(c) = [A_1: c_1, \ldots, A_n: c_n]$, where $\vdash (\frac{s_i}{\ell} \rightsquigarrow r) :: c_i$ for $1 \leq i \leq n$. Because r_i are deep copies of r with the same class, we have $class(r_i) = class(r)$ for $1 \leq i \leq n$. Because the $\vdash$ system only depends on the classes of receivers (except ***null*** and ***same***, which result in the typing BOTTOM) and because the class of r_i $(1 \leq i \leq n)$ is the same as the class of r, we also have $\vdash (\frac{s_i}{\ell} \rightsquigarrow r_i) :: c_i$ for $1 \leq i \leq n$, Therefore, o_i is an instance of c_i for $1 \leq i \leq n$. Since $\dot{\Xi}(c) = [A_1: c_1, \ldots, A_n: c_n]$ and $o = [A_1: o_1, \ldots, A_n: o_n]$, it follows that o is a legal instance of c.

8. Let s have value $\langle\!\{ s_1, s, s_3 \}\!\rangle$, and let s be neither identical nor equal to any behavior of $class(r)$. Let $\frac{s}{\ell} \rightsquigarrow r \Rightarrow o$. From the definition of complex message-sendings, we know that o must be the result of either $\frac{s}{\ell} \rightsquigarrow r$ (in the "then" case) or $\frac{s_3}{\ell} \rightsquigarrow r$ (in the "else" case). However, the former (i.e., $\frac{s}{\ell} \rightsquigarrow r$) does not return a result but simply repeats the original message-sending. Hence, because we assume that $\frac{s}{\ell} \rightsquigarrow r$ is not a nonterminating message-sending, o can only be the result of $\frac{s_3}{\ell} \rightsquigarrow r$. Now, let $\vdash (\frac{s}{\ell} \rightsquigarrow r) :: c$. Since RULE 10 is the corresponding derivation, we must have $\vdash (\frac{s_3}{\ell} \rightsquigarrow r) :: c$. Since this implies that $\frac{s_3}{\ell} \rightsquigarrow r$ returns an instance of class c, the actual result o is an instance of c, which is also the typing of $\frac{s}{\ell} \rightsquigarrow r$.

9. Let s have value $\langle\!\{ s_1, s_2, s \}\!\rangle$, and let s be neither identical nor equal to any behavior of $class(r)$. Let $\frac{s}{\ell} \rightsquigarrow r \Rightarrow o$. From the definition of complex message-sendings, we know that o must be the result of either $\frac{s_2}{\ell} \rightsquigarrow r$ or $\frac{s}{\ell} \rightsquigarrow r$. However, the latter (i.e., $\frac{s}{\ell} \rightsquigarrow r$) does not return a result but simply repeats the original message-sending. Hence, because $\frac{s}{\ell} \rightsquigarrow r$ is assumed to be terminating, o can only be the result of $\frac{s_2}{\ell} \rightsquigarrow r$. Now, let $\vdash (\frac{s}{\ell} \rightsquigarrow r) :: c$. Since RULE 11 is the corresponding derivation, we must have $\vdash (\frac{s_2}{\ell} \rightsquigarrow r) :: c$. Since this implies that $\frac{s_2}{\ell} \rightsquigarrow r$ returns an instance of class c, the actual result o is an instance of c, which is also the typing of $\frac{s}{\ell} \rightsquigarrow r$.

10. Let s have value $\langle\!\{ s_1, s_2, s_3 \}\!\rangle$, and let s be neither identical nor equal to any behavior of $class(r)$. Let $\frac{s}{\ell} \rightsquigarrow r \Rightarrow o$. From the definition of complex message-sendings, we know that o must be the result of either $\frac{s_2}{\ell} \rightsquigarrow r$ or $\frac{s_3}{\ell} \rightsquigarrow r$. Now,

let $\vdash (\frac{s}{\ell} \leadsto r)$ :: c. Since $s \not\equiv s_2$ and $s \not\equiv s_3$, the derivation must be RULE 12. Thus we must have $\vdash (\frac{s_2}{\ell} \leadsto r)$ :: c and $\vdash (\frac{s_3}{\ell} \leadsto r)$ :: c. This means that o, which is either the result of $\frac{s_2}{\ell} \leadsto r$ or the result of $\frac{s_3}{\ell} \leadsto r$, is an instance of c, which is also the typing of $\frac{s}{\ell} \leadsto r$.

11. Let s have value $\{s_1, \ldots, s_n\}$, and let s be neither identical nor equal to any behavior of $class(r)$. Let $\frac{s}{\ell} \leadsto r \Rightarrow o$. From the definition of complex message-sendings, we know that o must be the result of one of these message-sendings: $\frac{s_1}{\ell} \leadsto r, \ldots, \frac{s_n}{\ell} \leadsto r$. (But we do not know which one, until $\frac{s}{\ell} \leadsto r$ is actually evaluated.) Now, let $\vdash (\frac{s}{\ell} \leadsto r)$ :: c. Since the derivation must be RULE 13, we know that, for $1 \leq i \leq n$, $\vdash (\frac{s_i}{\ell} \leadsto r)$ :: c. This means that all message-sendings $\frac{s_i}{\ell} \leadsto r$ $(1 \leq i \leq n)$ return an instance of c. Therefore, no matter which $\frac{s_i}{\ell} \leadsto r$ actually returns o, o is always an instance of c, which is also the typing of $\frac{s}{\ell} \leadsto r$.

12. Let s have value $< s_1, \ldots, s_n >$, and let s be neither identical nor equal to any behavior of $class(r)$. Let $\frac{s}{\ell} \leadsto r \Rightarrow o$ and $(\frac{s}{\ell} \leadsto r)$ :: c_n. Since RULE 14 is the required derivation, we have

$$\vdash (\frac{s_1}{\ell} \leadsto r) :: c_1, \quad \vdash (\frac{s_2}{\ell} \leadsto o_{c_1}) :: c_2, \quad \ldots, \quad \vdash (\frac{s_n}{\ell} \leadsto o_{c_{n-1}}) :: c_n$$

Now we prove that o is an instance of c_n by induction on n. For $n = 1$, this statement is trivially true because $\frac{s}{\ell} \leadsto r$ becomes $\frac{s_1}{\ell} \leadsto r$, whose result o will be an instance of c_1 since $\vdash (\frac{s_1}{\ell} \leadsto r)$ :: c_1 holds. Hypothesis that the statement is true for all $k < n$. By the definition of complex message-sendings, it is easy to see that $\frac{s}{\ell} \leadsto r$ is semantically equivalent to $\frac{s_n}{\ell} \leadsto (\frac{<s_1, \ldots, s_{n-1}>}{\ell} \leadsto r)$, i.e., they return the same result. By the induction hypothesis on n, we know that $\frac{<s_1, \ldots, s_{n-1}>}{\ell} \leadsto r$ returns an instance o' of c_{n-1}. Now if we can prove that $\vdash (\frac{s_n}{\ell} \leadsto o')$ :: c_n, then by the inductive hypothesis on the value of s, we know that $\frac{s}{\ell} \leadsto r$ must return instance of c_n. However, at this point we only have $\vdash (\frac{s_n}{\ell} \leadsto o_{c_{n-1}})$ :: c_n, instead of $\vdash (\frac{s_n}{\ell} \leadsto o')$ :: c_n. Here, $o_{c_{n-1}}$ is an arbitrary direct instance of c_{n-1}, i.e., $class(o_{c_{n-1}}) = c_{n-1}$. We thus have to study two cases. (1) $class(o') = c_{n-1}$, or (2) $class(o') \textcircled{i} c_{n-1}$. If case (1) is true, we immediately have $\vdash (\frac{s_n}{\ell} \leadsto o')$ :: c_n. If case (2) is true, then the derivations through RULE 4, RULE 5 and RULE 6 may be different due to their dependence on the information about the receiver's (direct) class.

   (a) If $\vdash (\frac{s_n}{\ell} \leadsto o_{c_{n-1}})$ :: c_n is derived by RULE 4, then we know that s_n is ***same*** and $c_n = c_{n-1}$. Because $\frac{\textbf{same}}{\ell} \leadsto o'$ returns o' itself and because

$class(o') \textcircled{k} c_n$, we can conclude that o' is an instance of c_n.

(b) If $\vdash (\frac{s_n}{\ell} \leadsto o_{c_{n-1}}) :: c_n$ is derived by RULE 5, then we know that $s_n \in \dot{\Sigma}(c_{n-1})$ such that the formal output specification of behavior s_n is c_n. Since we have $class(o') \textcircled{k} c_n$, $class(o')$ must be k-compatible with c_n. This implies that the behavior s_n (invoked by s_n itself on receiver o') from the interface of $class(o')$ must have c_n or a subclass of c_n as its formal output specification. Therefore, if $\frac{s_n}{\ell} \leadsto o'$ returns o then o is still an instance of c_n.

(c) If $\vdash (\frac{s_n}{\ell} \leadsto o_{c_{n-1}}) :: c_n$ is derived by RULE 6, then we know that there is $b \in \mathring{\Sigma}(c_{n-1})$ such that $b \asymp s_n$ and the formal output specification of behavior b is c_n. Again, since we have $class(o') \textcircled{k} c_n$, $class(o')$ must be k-compatible with c_n. This implies that b (invoked by s_n on receiver o') must have c_n or a subclass of c_n as its formal output specification. Therefore, if $\frac{s_n}{\ell} \leadsto o'$ returns o then o is still an instance of c_n.

■

Since objects in a KBO database are a consistent set of objects. bottom object ***fail*** cannot be returned as a result by any knowhow in the database. Therefore, the following is immediate:

Theorem 17: *For any message-sending* $\frac{s}{\ell} \leadsto r$ *where* $r \neq \boldsymbol{fail}$ *and* $s \neq \boldsymbol{fail}$ *and* $\boldsymbol{fail} \notin orf(s)$, *for any object* o *and any class* c, *if* $\frac{s}{\ell} \leadsto r \Rightarrow o$ *and* $\vdash (\frac{s}{\ell} \leadsto r) :: c$, *then object* o *is not bottom object* ***fail***.

Proof: According to Theorem 16, if $\frac{s}{\ell} \leadsto r \Rightarrow o$ and $\vdash (\frac{s}{\ell} \leadsto r) :: c$, then object o is an instance of c. Suppose o is ***fail***. Because $r \neq \boldsymbol{fail}$, $s \neq \boldsymbol{fail}$ and $\boldsymbol{fail} \notin orf(s)$, and because the $\vdash$ system derives the typing c by rules that guarantee successful invocation of any behaviors required by $\frac{s}{\ell} \leadsto r$, o can only be produced by the execution of a behavior with class c as its formal output specification. However, since a KBO database contains a consistent set of KBO objects, no behaviors in the database can bear a knowhow whose execution may return ***fail*** — a contradiction. Therefore, if $\frac{s}{\ell} \leadsto r \Rightarrow o$ and $\vdash (\frac{s}{\ell} \leadsto r) :: c$, then o cannot be ***fail***. ■

To gain more insight in the $\vdash$ system as a dynamic type checker, we give an example of complex message-sendings which have a typing derivable from the inference rules.

Example 44: Recall we gave an example query in Chapter 6: "Raise the window *FooWindow* to the top if it contains point p; otherwise, move *FooWindow* to point

p and then raise it to the top", where *FooWindow* is an instance of class WINDOW (see Chapter 6.4). Assume we have the following bi-object:

$RaiseIt = \langle\!\langle$ *"Contains this point?"*, *"Raise to top"*, < *"Move to"*, *"Raise to top"* > $\rangle\!\rangle$

Then the above query can be formulated as the following complex message-sending:

$$\frac{RaiseIt}{\text{Point: } p} \rightsquigarrow FooWindow$$

Now we show that a typing can be derived for this complex message-sending. Using the $\vdash$ system, we can deduce the following:

(1) Applying RULE 6:

$$\vdash (\frac{\text{"}Move\ to\text{"}}{\text{Point} : p} \rightsquigarrow FooWindow) :: \text{SELF}$$

(2) Applying RULE 7:

$$\frac{\vdash (\frac{\text{"}Move\ to\text{"}}{\text{Point: } p} \rightsquigarrow FooWindow) :: \text{SELF}}{\vdash (\frac{\text{"}Move\ to\text{"}}{\text{Point: } p} \rightsquigarrow FooWindow) :: \text{WINDOW}}$$

(3) Applying RULE 6:

$$\vdash (\frac{\text{"}Raise\ to\ top\text{"}}{\text{Point} : p} \rightsquigarrow FooWindow) :: \text{SELF}$$

(4) Applying RULE 7:

$$\frac{\vdash (\frac{\text{"}Raise\ to\ top\text{"}}{\text{Point: } p} \rightsquigarrow FooWindow) :: \text{SELF}}{\vdash (\frac{\text{"}Raise\ to\ top\text{"}}{\text{Point: } p} \rightsquigarrow FooWindow) :: \text{WINDOW}}$$

(5) Applying RULE 14:

$$\frac{\begin{array}{c}\vdash (\frac{\text{"}Move\ to\text{"}}{\text{Point: } p} \rightsquigarrow FooWindow) :: \text{WINDOW} \\ \vdash (\frac{\text{"}Raise\ to\ top\text{"}}{\text{Point: } p} \rightsquigarrow {}^{o}\text{WINDOW}) :: \text{WINDOW}\end{array}}{\vdash (\frac{<\text{"}Move\ to\text{"}, \text{"}Raise\ to\ top\text{"}>}{\text{Point: } p} \rightsquigarrow FooWindow) :: \text{WINDOW}}$$

(6) Applying RULE 12: (Let s_3 = < *"Move to"*, *"Raise to top"* >)

$$\frac{\begin{array}{c}\vdash (\frac{\text{"}Raise\ to\ top\text{"}}{\text{Point: } p} \rightsquigarrow FooWindow) :: \text{WINDOW} \\ \vdash (\frac{s_3}{\text{Point: } p} \rightsquigarrow FooWindow) :: \text{WINDOW}\end{array}}{\vdash (\frac{\langle\!\langle \text{"}Contain\ this\ point?\text{"}, \text{"}Raise\ to\ top\text{"}, s_3 \rangle\!\rangle}{\text{Point: } p} \rightsquigarrow FooWindow) :: \text{WINDOW}}$$

Notice that in (6) we omit other obviously satisfied conditions. ■

The following theorem summarizes the main result of this section.

Theorem 18: ***In a KBO database, a complex message-sending is not an anomalous complex message-sending if it is a strictly-typed complex message-sending.***

Proof: This is a direct result of Theorem 17. ■

Theorem 18 assures us that if a complex message-sending can pass our dynamic checking, then it will succeed and return a meaningful result.

7.7 The $\vdash$ System Is Not Complete

The $\vdash$ system is ***complete*** if it can derive a typing for every message-sending that returns a result other than ***fail***. Completeness seems to be impossible because it is possible to have some successful complex message-sendings that are not strictly-typed complex message-sendings. This can be illustrated by some simple examples.

Example 45: Assume that class PERSON has the following three behaviors:

b_1 with: $value(b_1)$ = "*Live in London?*" and $sig(b_1) = () \longrightarrow$ BOOL
b_2 with: $value(b_2)$ = "*Get home address*" and $sig(b_2) = () \longrightarrow$ TEXT
b_3 with: $value(b_3)$ = "*Get phone number*" and $sig(b_3) = () \longrightarrow$ INT

Let $Where$ denote a bi-object,

$Where = \{\!|$ "*Live in London?*", "*Home address*", "*Phone number*" $|\!\}$

Then, the complex message-sending

$$Where \leadsto SomePerson$$

may be useful for the end-user, if the intention is to get in touch with $SomePerson$ either by visiting this person (if he/she lives in London) or by telephone calls. However, the message-sending is not a strictly-typed complex message-sending, because INT and TEXT are two direct subclasses of CAPSULE, and therefore, the $\vdash$ system is not able to derive a typing for the above message-sending. ■

Example 46: Assume that class PERSON also has the following two behaviors:

b_4 with: $value(b_4)$ = "*Set home address*" and
$sig(b_4) = (Address, \text{TEXT}) \longrightarrow \text{TEXT}$
b_5 with: $value(b_5)$ = "*Set phone number*" and
$sig(b_5) = (PhoneNumber, \text{INT}) \longrightarrow \text{INT}$

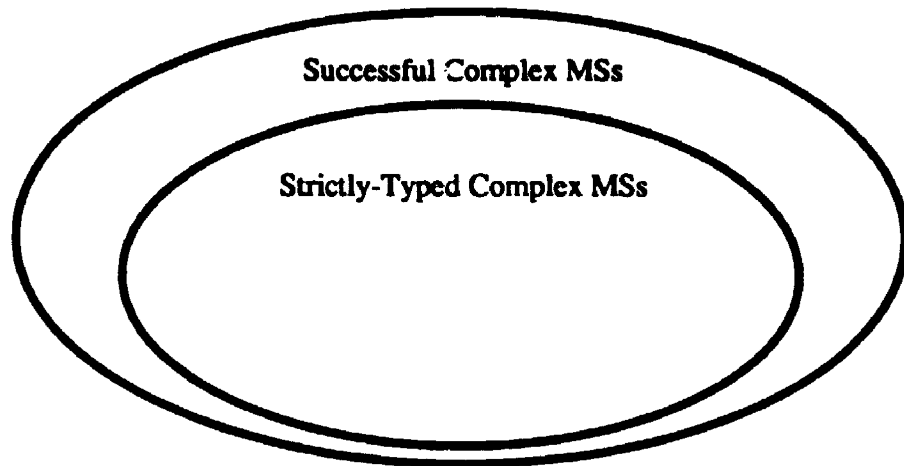


Figure 7.19: The Set of All Successful Complex Message-Sendings

where the semantics of b_4 is to update a person's home address and also return the new address to the end-user (for re-assuring or confirmation), and similarly, the semantics of b_5 is to update a person's phone number and also return the new address to the end-user. In other words, although b_4 and b_5 do not return the receiver itself, they still have side-effects (the updates). Let *SetAll* denote a set object,

$$SetAll = \{\text{"}Set\ home\ address\text{"}, \text{"}Set\ phone\ number\text{"}\}.$$

Then, the complex message-sending

$$\frac{SetAll}{\overline{\text{PhoneNumber} : 6722105}, \ \overline{\text{Address} : \text{"230 } Oxford\ Street,\ London\text{"}}} \leadsto SomePerson$$

may be useful for the end-user who only wants to update *SomePerson*'s phone number and home address, and does not care what the message-sending returns. However, for the similar reason as in the previous example, if we use the $\vdash$ system to check this message-sending, we will get a failure report. ■

Clearly, the set of all strictly-typed complex message-sendings is only a subset of all successful complex message-sendings, as shown in Figure 7.19. As we have seen from above examples, the $\vdash$ system does not allow some (possibly useful) complex message-sendings. The tradeoff here is that more type information can be gained by disallowing such complex message-sending as we explained previously. It is hoped that the rules of the $\vdash$ system will be of use in developing more flexible type inference procedures in our future research.

CHAPTER 8
THE KBO OBJECT ALGEBRA

8.1 Introduction

This chapter presents the KBO algebra, an object algebra associated with the KBO model. The KBO algebra defines a higher-level manipulative semantics for the KBO model and at the same time serves as a basis for developing other forms of higher-level query languages for KBO databases. The choice of operators for the KBO algebra has been partially influenced by the usefulness of those operations in relational databases. A distinct feature of the KBO algebra is that it takes into account the notion of sendable objects in the KBO model. For example, the operands of some operators are defined to be invokers in message-sendings. Furthermore, the KBO algebra provides a uniform set of operations over all five types of classes (i.e., CAPSULE, AGG, BIO, SET and SEQ classes). As a consequence, end-users can enjoy a truly generic query facility, without having to be aware of what kind of targets they are querying against. Another interesting feature of the KBO algebra is that it can be used to produce and maintain complex behaviors on a large scale without involving hardcoding programmers. We believe that this feature opens a new application area for using database query facilities. We will demonstrate this feature later in the chapter.

A very important point must be emphasized here: Because our algebra operators are defined to be as polymorphic as possible (i.e., applicable on all kinds of operands), some of them (mainly Apply and Combine) have a tedious definition which enumerates all possible cases. However, in practice, for a specific application or a specific user, only a few cases of those operators would be used often. For example, most conventional users may stick with queries against sets, and hence, for each of our operators, they only need know how the operators work on sets. Our intention is to define a complete formal object algebra, thus providing a theoretical foundation for further studies, such as query transformations.

Throughout this chapter, we will often use the notion of an "object handle" in our examples. In the real world, we use names, such as *Mike* or *UWO*, to refer to real-world objects. Thus, we assume that there is an *object handle space* denoted by $\mathcal{H}$, which is a countably infinite set of generic (or untyped) object variables called *object handles*. The handle assignment is an expression of the form

$$h := exp$$

where the result of an algebra expression *exp* is assigned to an object handle $h \in \mathcal{H}$. More precisely, the object handle h is now pointing to the result object evaluated from the algebra expression *exp*. It must be noted that object handles are not objects — they just denote objects. In other words, they are intended to be used only as a mnemonic name for an object identity. Each handle assignment will cause an object handle to denote a new object-identity. We say that an object is a *named object* if it can be reached through an object handle.

As we introduce the operators in the KBO algebra, we will illustrate their application with example queries on an OODB system shown in Figure 8.20 (for simplicity, we assume that all REUSE-OF relationships are the same as all KIND-OF relationships.) The classes INT, TEXT and TEXT-SET can be considered to be predefined by the system. The example OODB is based on the setting of an imaginary large corporation: the company has EMPLOYEEs, some of whom are SCIENTISTs; employees have a FAMILY which may include other PERSONs; SCIENTISTs may write technical REPORTs which are a sequence of CHAPTERs, which may cite other PAPERs published in the literature; the company offers a wide variety of LI-PLANs (Life Insurance PLANs) to its employees; in order to keep a high morale among employees, PERSONAL-QUESTIONs are designed to study (and help solve) employees' personal problems; employees are also assisted in calculating their tax credits using GET-CREDIT methods; every now and then the CEO of the company may make some quick CEO-DECISIONs. Two points must be noted in this database. First, any new data types, which the user does not wish to be represented as a subclass of AGG, BIO, SET or SEQ (such as BITMAP, SONG-CLIP, etc.) can always be created as a subclass of CAPSULE, which may have its own subclasses later. In this way, the database supports extensibility [Atk89]. Second, all subclasses (except TEXT-SET which may be predefined) under SET and SEQ are created by database users either because they want to implement some (user-defined) behaviors for these classes or because they want to use these classes to specify an attribute or a signature. For example, class PERSON has a counterpart PERSONS under SET either because PERSONS itself also has some behaviors, such as "Get average age" from an instance of PERSONS (a set of persons) or because class PERSONS is needed to define an attribute in other classes. As another example, class FAMILY does not have a counterpart class under SET or SEQ, because the users have not felt the need either to create some special behaviors for a set of families or to use FAMILIES to specify an attribute or a signature. Nevertheless, if desired, a set of FAMILY objects can still be stored in a direct instance of class SET, which can serve as an operand

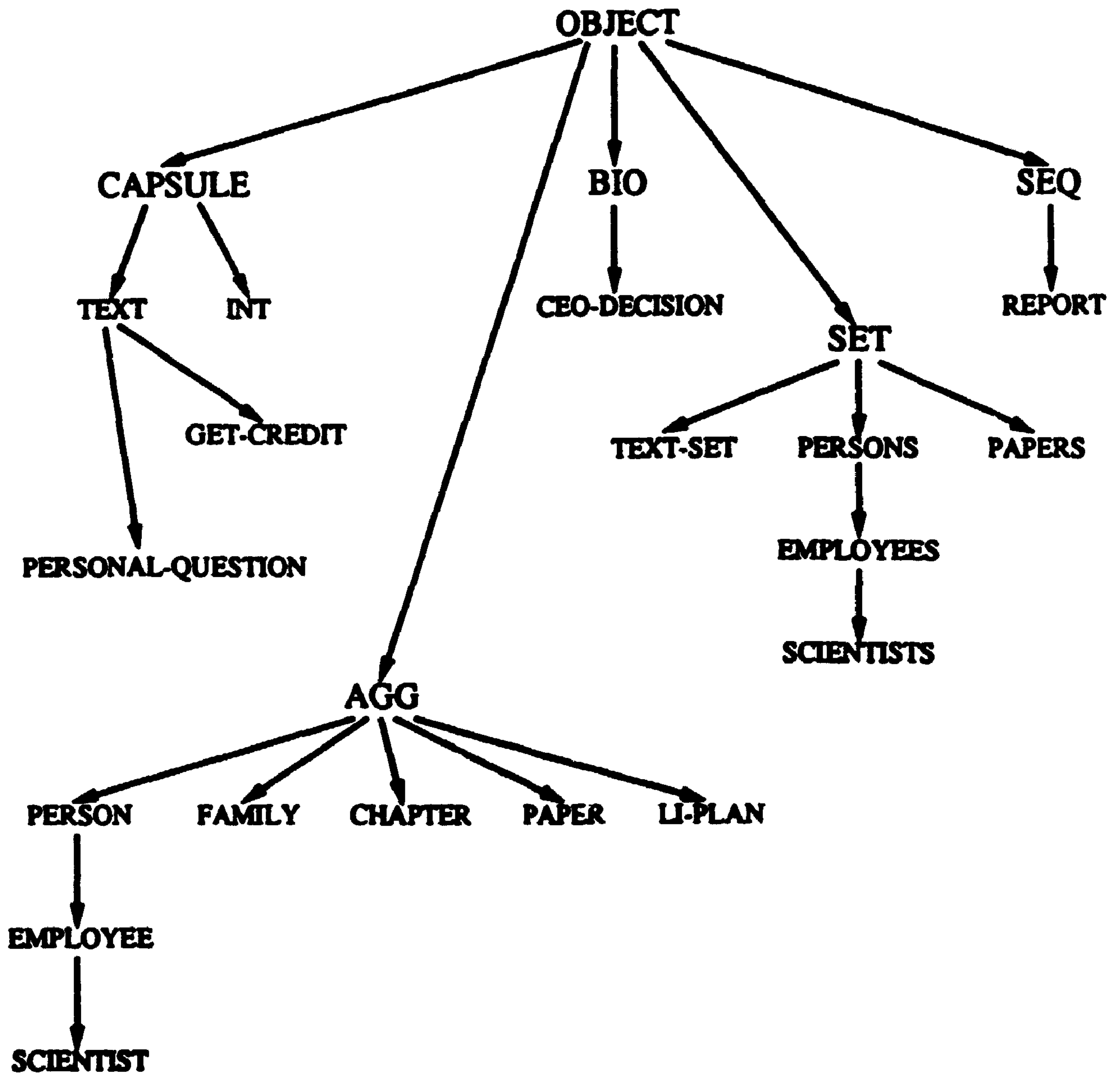


Figure 8.20: The KIND-OF and REUSE-OF Hierarchies for an OODB System

for any of the operations in the KBO algebra (such as "select families that have a single parent from a set of families"). Note that in some existing OODBs such as ORION [B*87], any user-defined class always automatically causes the creation of a counterpart class under a generic class Set.

8.2 The Algebraic Operators

Each operator of the KBO Algebra takes one or more KBO objects as input and produces a single KBO object as output. Precisely speaking, operands and results in the KBO algebra are *object-identities* of KBO objects. In particular, any KBO object can be taken by any of our operators as an operand. Therefore, the KBO algebra maintains the closure property where the result of a query can be used as the input to another.

The algebra has eleven operators: Select σ, Pickup δ, Apply ρ, Expression Apply λ, Project π, Combine χ, Union $\cup$, Intersect $\cap$, Subtract $-$, Collapse ϖ, and Assimilate α. Some of the these algebra operators are qualified by a predicate $p(x_1, \ldots, x_n)$ defined on object variables $x_1, \ldots, x_n$. Such a predicate is a first-order Boolean formula with variables representing KBO objects. These object variables can be bound to one or more algebraic operations. The formula consists of one or more *atoms* connected by $\wedge$, $\vee$, or $\neg$ using parenthesis as desired. Legal atoms are defined in the following:

Definition 63: *Atoms.* A legal *atom* in the KBO algebra has one of the following forms:

- $x\theta y$, where x and y

    1. are object variables or constants, or

    2. are "dotted" object variables (i.e., $\dot{x}$ or $\dot{y}$) representing a "deep-copied" object variable, that is, $\dot{x}$ copies all the values but no identities from x such that $\dot{x} \asymp x$ but $\dot{x} \not\equiv x$ (the notion of deep-copy was first proposed in [KC86]), or

    3. denote a message-sending of the form

$$\frac{s}{L_1 : o_1, \ldots, L_n : o_n} \rightsquigarrow r$$

or

$$\frac{s}{L_1 : o_1, \ldots, L_n : o_n} \overset{\nu}{\rightsquigarrow} r$$

where ν is either an integer $n \geq 1$ or $*$, and $s, r, o_1, \ldots, o_n$ are object variables or constants, or

4. denote an attribute extraction of the form $z.A$ where z is an object variable bound to aggregate objects, or
5. denote a component extraction of the form $z.if, z.then$, or $z.else$ where z is an object variable bound to bi-objects,

and θ is one of the object comparison operators $\equiv$, $\asymp$, $\asymp_n$ $(n \geq 1)$, $\equiv_{kh}$, $\in$, $\in_{\asymp}$, $\in_{\asymp_n}$ $(n \geq 1)$, or $\in_{\equiv_{kh}}$. Here, $\in$, $\in_{\asymp}$, $\in_{\asymp_n}$ and $\in_{\equiv_{kh}}$ denote the set/sequence membership test using equalities $\equiv$, $\asymp$, $\asymp_n$ and $\equiv_{kh}$, respectively.

- $x := y$, where x is an object variable and y denotes a message-sending. This atom causes x to denote the same object denoted by y (e.g., the result of a message-sending) and is always evaluated to **True**.
- $x\theta y$, where x and y are object variables or constants bound to class INT, and θ is one of the usual integer comparison operators $<, >, \leq$ or $\geq$.

In above, comparison operators for INT objects should be thought of as special forms of message-sendings. Because they are used so often in our example queries, we simplify the syntax of these message-sendings by expressing them in their usual form (i.e., as operators that compare the values of two INT objects). Furthermore, throughout this chapter, capsule objects in our query examples that are (constant) instances of INT or TEXT will often be denoted by their values (e.g., 130 or "Mary") instead of by their identities, only for the sake of clarity.

Example 47: Let x, y, z be object variables. Then the following are examples of legal atoms and their semantics:

1. $(x \equiv y)$ — Are the objects denoted by x and y identical objects ?
2. $(x.then \equiv_{kh} y.else)$ — Do the then-part of x and the else-part of y share the same knowhow?
3. $(x \asymp_2 y)$ — Are the objects denoted by x and y 2-level equal objects?
4. $((y \rightsquigarrow \dot{x}) \asymp x.A)$ — Is the result of sending y to a copy of x (i.e., a pretended x) equal to the attribute A of the object x? Note here, that if y causes some side-effects on $\dot{x}$, we don't have to worry about x being affected because $\dot{x}$ and x do not share any object-identities at any level of nesting.

5. $((x \rightsquigarrow y) \in z.A)$ — Is the identity of the result of sending x to y contained in the set (or sequence) value of the attribute A of the object z?

6. $(x.if \in_{\times} (y \rightsquigarrow z))$ — Is the if-part of x equal to an object contained in the result of sending y to z?

7. $(x \asymp \text{"}hello\text{"})$ — Is text object x equal to "hello"?

8. $(50 < (\frac{x}{L:y} \rightsquigarrow z))$ — Is 50 less than the integer value of the result of sending x to z with an argument y labeled L?

Before we define our algebra operators, we shall introduce a notation $\star$ to denote the concatenation of sequence values such that

$$<x_1,\ldots,x_m> \star <y_1,\ldots,y_n> \; = \; <x_1,\ldots,x_m,y_1,\ldots,y_n>$$

where $m \geq 0$ and $n \geq 0$. Now we are ready to define the operators in the KBO algebra. Note that the special notation $\triangleq$ is read "is defined as".

8.2.1 Select (σ)

The Select operator, σ, allows one or more existing objects to be selected based on a selection predicate $p(x_1,...,x_n,x)$. The Select operator will always take at least one operand, and, in this case, the selection predicate is expressed as $p(x)$. The main purpose of introducing extra operands is to increase the expressiveness of the selection predicate so that its expression may have other variables ranging over a set or sequence operand which may be expressed as another nested algebra expression.

Definition 64: *The Select Operator σ.* Let $s_1,\ldots,s_n,r$ be any KBO objects (precisely, their object-identities). The Select operation, denoted

$$\sigma_{p(x_1,\ldots x_n,x)}(s_1,\ldots,s_n,r)$$

where $n \geq 0$, is defined as follows:

1. If r ⓘ CAPSULE, AGG or BIO, then

$$\sigma_{p(x_1,\ldots x_n,x)}(s_1,\ldots,s_n,r) \triangleq \begin{cases} r & \text{if } p(s'_1,\ldots,s'_n,r) \\ \textbf{\textit{null}} & \text{otherwise} \end{cases}$$

   The result is an instance of $class(r)$. Here, variables $s'_1,...,s'_n$ will be defined shortly.

2. If $r \; \textcircled{i}$ SET, then

$$\sigma_{p(x_1,\ldots x_n,x)}(s_1,\ldots,s_n,r) \triangleq \{o \mid o \in r \wedge p(s'_1,\ldots,s'_n,o)\}$$

The result is an instance of $class(r)$.

3. If $r \; \textcircled{i}$ SEQ and $r = <r_1,\ldots,r_m>$, then

$$\sigma_{p(x_1,\ldots x_n,x)}(s_1,\ldots,s_n,r) \triangleq r'_1 \star \cdots \star r'_m$$

where

$$r'_i = \begin{cases} <r_i> & \text{if } p(s'_1,\ldots,s'_n,r_i) \\ <> & \text{otherwise} \end{cases}$$

The result is an instance of $class(r)$.

In the above, $s'_1,\ldots,s'_n$ in the predicate $p(s'_1,\ldots,s'_n,x)$ are defined as follows:

$$s'_i = \begin{cases} s_i & \text{if } s_i \; \textcircled{i} \text{ CAPSULE, AGG or BIO} \\ o & \text{if } s_i \; \textcircled{i} \text{ SET or SEQ and there exists } o \in s_i \end{cases}$$

In other words, without loss in generality, if $s_1,\ldots,s_k$ are instances of CAPSULE, AGG or BIO, and $s_{k+1},\ldots,s_n$ are instances of SET or SEQ, then

$$p(s'_1,\ldots,s'_n,x) \Longleftrightarrow \exists o_{k+1} \in s_{k+1},\ldots,\exists o_n \in s_n : p(s_1,\ldots,s_k,o_{k+1},\ldots,o_n,x)$$

■

The Select operator is similar to that of [SO90] which also allows multiple (set) operands. However, our Select operator is more powerful in that (1) it is "polymorphic" — it takes any kind of operands (not only sets), and (2) if there are message-sendings in the predicate $p(s_1,\ldots,s_n,x)$ then their invokers can be selected from the multiple operands and qualified by the predicate (this feature will be illustrated shortly). Multiple operands permit explicit joins in the sense that the set/sequence objects are selected based on their relationships with objects in other sets/sequences.

Example 48: Let handle *MyGroup* denote an instance of class PERSONS which has the following representation:

$\dot{\Xi}$(PERSONS) = {PERSON}

$\dot{\Xi}$(PERSON) = [Name: TEXT, Address: TEXT, Age: INT, Income: INT, Spouse: PERSON, Children: PERSONS, ...]

That is, *MyGroup* is a set (object) of PERSON objects.

Query 1: Find all the persons in *MyGroup* who are older than 30 and live on Queen St.

$$\sigma_{x\ Age>30\ \wedge\ x.Address \asymp \text{"Queen St."}}(MyGroup)$$

where the object variable x ranges over the set *MyGroup*. This query produces a new instance of PERSONS with a new object-identity.

For the next query, let handle *AIgroup* denote another instance of PERSONS, i.e., another set (object) of persons.

Query 2: Find all married persons in *MyGroup* who are younger than some person in *AIgroup* who is not married. (Let x_1 range over *AIgroup* and x range over *MyGroup*)

$$\sigma_{x_1\ Spouse \equiv \mathbf{null}\ \wedge\ \neg(x.Spouse \equiv \mathbf{null})\ \wedge\ -\ Age < x_1.Age}(AIgroup, MyGroup)$$

Here, if a person is not married then the object representing this person should have a ***null*** Spouse attribute. This query produces a new instance of PERSONS which is a subset of *MyGroup*. Note that both operands can be replaced by another algebra expression.

For our third query, suppose class PERSON has the following behavior with:

value "*On unemployment insurance*", signature () $\longrightarrow$ SELF

This behavior knows how to put a person on unemployment insurance (for people who are laid off). For example, the knowhow of this behavior might be implemented as an expert system to calculate (and update) a person's income based on his/her current personal data. However, since knowhows are encapsulated code, end-users only know, from the signature of this behavior, that this behavior will cause some side-effects on a person object's internal state (to put him/her on unemployment insurance).

Query 2: Find all the persons in *MyGroup* who, if laid off, would still earn more than what their spouses are earning now[8].

$$\sigma_{t:=(\text{"On unemployment insurance"} \rightsquigarrow \dot{x})\ \wedge\ t.Income > x.Spouse.Income}(MyGroup)$$

Some interesting features are demonstrated in this example:

- First, variable $\dot{x}$ ranges over the *deep copies* of the members in set object *MyGroup*. That is, for each $x \in MyGroup$, $\dot{x} \asymp x$ but $\dot{x} \not\equiv x$ and $orf(\dot{x}) \cap orf(x) = \emptyset$ (i.e., they do not share any identities at all!). This implies that any side-effects that happen within $\dot{x}$ never happen within x because they do not share any data values. For instance, even if the Income of $\dot{x}$ was updated from 30000 to 18000, the Income of x is still 30000.

[8]We assume that the atoms in the predicate are evaluated from left to right.

- Second, the auxiliary variable t only serves as a convenience for denoting the result of the message-sending "*On unemployment insurance*" $\leadsto \dot{x}$. One might say that attribute extraction $t.Income$ could be replaced by

$$(\text{"On unemployment insurance"} \leadsto \dot{x}).Income$$

However, we prefer to use $t :=$ ("*On unemployment insurance*" $\leadsto \dot{x}$), because it is clearer and simplifies the atoms in the predicate, particularly when t is used in more than one atom. Notice that this message-sending returns the receiver $\dot{x}$ itself; in other words, t denotes a person $\dot{x}$ who has just been put on unemployment insurance.

- Third, as we said, all selected persons in the result of this query are actually never affected by the message-sending "*On unemployment insurance*" $\leadsto \dot{x}$ because $\dot{x}$ and x do not share any object-identities at all. In other words, all selected persons in the result are still the same persons in the input set *MyGroup* — what we can know from this query is that these selected people might still earn more than their spouses even if they were laid off. We believe that such queries would have many applications. In existing object algebras, such queries are not explicitly supported.

The following example illustrates selections performed on sequence objects.

Example 49: Let handle *MyReport* denote an instance of class REPORT which has the following representation:

$\dot{\Xi}$(REPORT) = <CHAPTER>

$\dot{\Xi}$(CHAPTER) = [Content: TEXT, Done: BOOL, PapersCited: PAPERS]

That is, class REPORT is created to represent sequences of CHAPTER objects. Hence *MyReport* is a sequence of chapters. Assume that there is also a class SPELL-CHECKER which has a behavior with:

value "*Check article*", signature (Article,TEXT) $\longrightarrow$ INT

This behavior produces an integer indicating the number of spelling errors found in an (input) article. Let handle *SmartChecker* denote an instance of class SPELL-CHECKER. As an example of how to use *SmartChecker*, the message-sending

$$\frac{\text{"Check article"}}{\text{Article : "A very very shorrt artecil."}} \leadsto SmartChecker$$

should return integer 2 indicating that two spelling errors have been found in the input article (a TEXT object). Let us now consider the following query.

Query: Find all the chapters in *MyReport* which are done and have no spelling errors when checked by the spelling-checker *SmartChecker*. (Let x range over *MyReport*.)

$$\sigma_{x.Done \asymp \textbf{True} \wedge (\frac{\text{"Check article"}}{\text{Article} \quad x.\text{Content}} \rightsquigarrow SmartChecker) \asymp 0}(MyReport)$$

Note that for each chapter x in *MyReport*, *x.Content* is a TEXT object. This query produces a new instance of REPORT, which may have fewer members than *MyReport* but preserves the ordering of *MyReport*. ■

The following example shows that the selection predicates involving message-sendings can have invokers selected from sets/sequences of objects.

Example 50: Assume that class EMPLOYEE not only reuses all attributes from PERSON but also adds more new attributes, including [... LifeIns: INT ...], which represents the lump sum of the life insurance held by an employee. Suppose the company offers a wide variety of life insurance plans to its employees. For this purpose, the class LI-PLAN (Life Insurance PLAN) is created with the following representation:

$\dot{\Xi}$(LI-PLAN) = [Hold: TEXT, Protection: TEXT, BestForAgeUnder: INT]

Upon the creation of each LI-PLAN object, it is given a knowhow (hardcoded computation) which knows how to calculate a specific type of life insurance policy for an employee based on some company rules as well as the employee's personal data (from an EMPLOYEE object). In other words, these LI-PLAN objects are created to be the behaviors of class EMPLOYEE. Each LI-PLAN object is not represented by a TEXT object; instead, it is an aggregate object so more detailed annotation can be given to the knowhow the object bears. Precisely, attribute Hold describes what this life insurance plan is called (e.g., "Freedom55"), attribute Protection describes what protection this plan is mainly intended to offer (e.g., "short term"), and attribute BestForAgeUnder suggests an age limit of people who may benefit the most from this plan. Note that these LI-PLAN objects are not intended to be complex behaviors because their attributes are not intended to model a sub-behavior; rather, their attributes are intended to describe some meta-semantic characteristics of the knowhow they bear. The following are two examples of LI-PLAN objects:

(id_1, LI-PLAN,
  [*Hold*: *"Freedom55"*, *Protection*: *"short term"*, *BestForAgeUnder*: 50],
  (EMPLOYEE, () $\longrightarrow$ SELF, kh_1))

(id_2, LI-PLAN,
  [*Hold*: *"Comprehensive"*, *Protection*: *"long term"*, *BestForAgeUnder*: 30],

(EMPLOYEE, () $\longrightarrow$ SELF, kh_2))

Here, object id_1 is a direct instance of class LI-PLAN, has value

$$[Hold\text{: "}Freedom55\text{"}, Protection\text{: "}shortterm\text{"}, BestForAgeUnder\text{: } 50]$$

and is a direct behavior of EMPLOYEE with signature () $\longrightarrow$ SELF and knowhow kh_1. The structure of object id_2 can be similarly explained. The knowhows kh_1 and kh_2 implement two different calculation methods which calculate the value for the LifeIns attribute of an employee based on some company rules and the employee's current personal data (e.g., age, income, single or married, with or without children, etc.). Note that these knowhows may cause other side-effects, like updating the spouse's life insurance, printing out a policy report, etc. In the following, let handle *HoldLifeIns* denote a set of LI-PLAN objects, and handle *Mike* denote an object of EMPLOYEE.

Query 1: Select *Mike* if it is possible for him to hold a short-term life insurance policy worth more than \$100000 through a plan from *HoldLifeIns*. (Let x_1 range over *HoldLifeIns*.)

$$\sigma_{x_1.Protection \asymp \text{"short term"} \wedge t:=(x_1 \rightsquigarrow \dot{x}) \wedge t.LifeIns > 100000}(HoldLifeIns, Mike)$$

Note that since *Mike* is an aggregate object, variable x in the predicate will be directly substituted by object *Mike*. This expression can be read "If there is a company life insurance plan x_1 in *HoldLifeIns* which offers short-term protection and by holding which employee x can can obtain insurance worth more than \$100,000, then select employee x ". It must be noted that during the evaluation of this query, the real target *Mike* is never affected by the message-sending $x_1 \rightsquigarrow \dot{x}$, which only causes side-effects (some updates) on a "pretended target" $\dot{x}$ (a deep copy of *Mike*). As a comparison, the following query

$$\sigma_{x_1.Protection \asymp \text{"short term"} \wedge t:=(x_1 \rightsquigarrow x) \wedge t.LifeIns > 100000}(HoldLifeIns, Mike)$$

will return, if successful, the object *Mike* who has already taken a selected life insurance plan x_1 from *HoldLifeIns*. This means, the *Mike* after this query may have a different internal state from the *Mike* before the query.

For the next query, let handle *MyGroup* denote a set of EMPLOYEE objects.

Query 2: Find all the employees in *MyGroup* who have life insurance worth less than \$50,000 but can increase their life insurance to more than \$200,000 by holding a life insurance plan from *HoldLifeIns* that is most suitable for them. (Let x range over *MyGroup* and x_1 range over *HoldLifeIns*.)

$$\sigma_{x.LifeIns < 50000 \wedge x.Age \leq x_1.BestForAgeUnder \wedge t:=(x_1 \rightsquigarrow \dot{x}) \wedge t.LifeIns > 200000}(HoldLifeIns, MyGroup)$$

Once again, all the employees selected by this query are not actually affected by the message-sending $x_1 \rightsquigarrow \dot{x}$ (they have not taken a plan from *HoldLifeIns* yet). Also, an interesting feature illustrated by this example is the atom $x.Age \leq x_1.BestForAgeUnder$ in the above predicate, which specifies a relationship between the desired invoker x_1 and the desired receiver x in the message-sending $x_1 \rightsquigarrow \dot{x}$. In other words, different employees x in *MyGroup* may be sent different invokers in $x_1 \rightsquigarrow \dot{x}$ depending whether they have the relationship $x.Age \leq x_1.BestForAgeUnder$. Note that since x_1 from *HoldLifeIns* has signature () $\longrightarrow$ SELF, message-sending $x_1 \rightsquigarrow \dot{x}$ returns $\dot{x}$ itself, which is in turn denoted by t. The result of this query is a new instance of EMPLOYEES, which is also a subset of *MyGroup*.

The next query illustrates a Select operation with three operands.

Query 3: Find all the employees in *MyGroup* who have life insurance worth less than that of an employee in *AIgroup* but can increase their life insurance to more than that employee by holding a life insurance policy from *HoldLifeIns*. (Let x range over *MyGroup*, x_1 range over *HoldLifeIns* and x_2 range over *AIgroup*.)

$$\sigma_{x.LifeIns<x_2.LifeIns \ \wedge \ t:=(x_1 \rightsquigarrow \dot{x}) \ \wedge \ t.LifeIns>x_2.LifeIns}(AIgroup, HoldLifeIns, MyGroup)$$

This query also returns a subset of *MyGroup*. Note that, if the result is an empty set, then it tells us that every one in *AIgroup* has probably got the best deal they can get from the company. ■

8.2.2 Pickup (δ)

The Pickup operator, written δ, picks up an object that satisfies a predicate. In particular, when it is applied to a set or sequence, it will evaluate each member of the set/sequence against the predicate, until a member that satisfies the predicate is found. Then, this member is "picked up" and produced as the result. Hence, the main difference between Pickup and Select operators is that Select operator produces a subset/subsequence of the operand r as output if r is a SET or SEQ object, whereas Pickup produces at most one member.

Definition 65: *The Pickup Operator δ.* Let $s_1, \ldots, s_n, r$ be any KBO objects (precisely, their object-identities). The Pickup operation, denoted

$$\delta_{p(x_1,\ldots x_n,x)}(s_1, \ldots, s_n, r)$$

where $n \geq 0$, is defined as follows:

1. If r ⓘ CAPSULE, AGG or BIO, then

$$\delta_{p(x_1,\ldots x_n,x)}(s_1,\ldots,s_n,r) \triangleq \begin{cases} r & \text{if } p(s'_1,\ldots,s'_n,r) \\ \mathbf{null} & \text{otherwise} \end{cases}$$

2. If r ⓘ SET, then

$$\delta_{p(x_1,\ldots x_n,x)}(s_1,\ldots,s_n,r) \triangleq \begin{cases} o & \text{if } o \in r \wedge p(s'_1,\ldots,s'_n,o) \\ \mathbf{null} & \text{otherwise} \end{cases}$$

The result is an existing instance of class c such that $\dot{\Xi}(class(r)) = \{c\}$.

3. If r ⓘ SEQ and $r = <r_1,\ldots,r_m>$, then

$$\delta_{p(x_1,\ldots x_n,x)}(s_1,\ldots,s_n,r) \triangleq \begin{cases} r_k & \text{if } r_k \in r \wedge p(s'_1,\ldots,s'_n,r_k) \\ & \quad \wedge \forall i \in [1..k-1] : \neg p(s'_1,\ldots,s'_n,r_i) \\ \mathbf{null} & \text{otherwise} \end{cases}$$

The result is an existing instance of class c such that $\dot{\Xi}(class(r)) = <c>$.

where the predicate $p(s'_1,\ldots,s'_n,x)$ is defined the same way as in the Select operator.
■

Note that when operand r is a sequence, the Pickup operator picks up the first member in r that satisfies the predicate.

Example 51: Using the context of Example 43, we can formulate the following query:

Query: Find an employee in *MyGroup* who has life insurance worth less than $50,000 but can increase his/her life insurance to more than $200,000 by holding a life insurance plan from *HoldLifeIns* that is best suited to him/her. (Let x range over *MyGroup* and x_1 range over *HoldLifeIns*.)

$\delta_{x.LifeIns<50000 \,\wedge\, x.Age \leq x_1.BestForAgeUnder \wedge\, t:=(x_1 \rightsquigarrow x) \,\wedge\, t.LifeIns>200000}$
$(HoldLifeIns, MyGroup)$

If successful, this query returns a single EMPLOYEE object from the set *MyGroup*.
■

8.2.3 Apply (ρ)

The Apply operator, ρ, is a powerful tool to apply a single invoker, a set of invokers, or a sequence of invokers, to a single receiver, a set of receivers, or a sequence of

receivers. The Apply operator is qualified by a binary predicate which specifies certain relationships between an invoker candidate and a receiver candidate. It must be observed that in general this operation cannot be achieved by performing an operation like *Image* (which applies a behavior to each member of a set) in Shaw and Zdonik's algebra [SZ89] or *Map* (which applies a sequence of behaviors to each member of a set) in Straube and Ozsu's algebra [SO90], followed by a selection operation (or vice versa), because, for each member x_r (receiver) in a set object, our Apply operator may choose a different invoker x_s to apply, based on the binary predicate $p(x_s, x_r)$. In other words, each member in a set of receivers may not have the same invoker — a major difference from operators like *Image* and *Map*. This is, in fact, a very distinct advantage of the KBO algebra over the existing object algebras. We will further elaborate this feature later in our examples.

Definition 66: *The Apply Operator* ρ. Let ℓ be a blackboard and s, r be any two KBO objects (precisely, their object-identities). The Apply operation, denoted

$$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell}$$

is defined as follows:

1. If s ⓘ CAPSULE, AGG or BIO, then

   (a) if r ⓘ CAPSULE, AGG or BIO, then

   $$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell} \triangleq \begin{cases} \frac{s}{\ell} \rightsquigarrow r & \text{if } p(s,r) \\ \textbf{null} & \text{otherwise} \end{cases}$$

   (b) if r ⓘ SET, then

   $$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell} \triangleq \{\frac{s}{\ell} \rightsquigarrow o \mid o \in r \wedge p(s,o)\}$$

   The result is a direct instance of class SET.

   (c) if r ⓘ SEQ and $r = <o_1, \ldots, o_n>$, then

   $$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell} \triangleq o'_1 \star \cdots \star o'_n$$

   where

   $$o'_i = \begin{cases} <\frac{s}{\ell} \rightsquigarrow o_i> & \text{if } p(s,o_i) \\ <> & \text{otherwise} \end{cases}$$

   The result is a direct instance of class SEQ.

2. If $s \textcircled{i}$ SET, then

   (a) if $r \textcircled{i}$ CAPSULE, AGG or BIO, then

   $$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell} \triangleq \{\frac{o}{\ell} \rightsquigarrow r \mid o \in s \wedge p(o,r)\}$$

   The result is a direct instance of class SET.

   (b) if $r \textcircled{i}$ SET, then

   $$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell} \triangleq \{\ \{\frac{o_s}{\ell} \rightsquigarrow o_r \mid o_s \in s \wedge p(o_s,o_r)\} \mid o_r \in r\}$$

   The result is a direct instance of class SET, which in turn is a set of SET objects.

   (c) if $r \textcircled{i}$ SEQ and $r = <o_1, \ldots, o\ >$, then

   $$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell} \triangleq o'_1 \star \cdots \star o'_n$$

   where

   $$o'_i = <\{\frac{o_s}{\ell} \rightsquigarrow o_i \mid o_s \in s \wedge p(o_s,o_i)\}>$$

   The result is a direct instance of SEQ, which in turn is a sequence of SET objects.

3. If $s \textcircled{i}$ SEQ and $s = <s_1, \ldots, s_n>$, then

   (a) if $r \textcircled{i}$ CAPSULE, AGG or BIO, then

   $$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell} \triangleq o'_1 \star \cdots \star o'_n$$

   where

   $$o'_i = \begin{cases} <\frac{s_i}{\ell} \rightsquigarrow r> & \text{if } p(s_i,r) \\ <> & \text{otherwise} \end{cases}$$

   The result is a direct instance of SEQ.

   (b) if $r \textcircled{i}$ SET, then

   $$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell} \triangleq \{\ \psi(s_1,o_r) \star \cdots \star \psi(s_n,o_r) \ \mid o_r \in r\}$$

   where

   $$\psi(s_i,o_r) = \begin{cases} <\frac{s_i}{\ell} \rightsquigarrow o_r> & \text{if } p(s_i,o_r) \\ <> & \text{otherwise} \end{cases}$$

   The result is a direct instance of SET, which in turn is a set of SEQ objects.

(c) if r ⓘ SEQ and $r = <r_1, \ldots, r_m>$, then

$$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell} \triangleq r'_1 \star \cdots \star r'_m$$

where

$$r'_i = <\psi(s_1, r_i) \star \cdots \star \psi(s_n, r_i)>$$

and

$$\psi(s_i, r_i) = \begin{cases} <\frac{s_i}{\ell} \leadsto r_i> & \text{if } p(s_i, r_i) \\ <> & \text{otherwise} \end{cases}$$

The result is a direct instance of SEQ, which in turn is a sequence of SEQ objects.

■

Note that the predicate $p(x_s, x_r)$ can be omitted if it is simply the constant True. We should point out that when a query result is a direct instance of kernel classes SET or SEQ (or AGG and BIO), it can be either used immediately as input for other algebra operations or saved under a new subclass of SET or SEQ, so that in the future some *user-defined* behaviors can be implemented for that new class. Although the saving facility is not explicitly provided within our algebra (such as an operator like ReClass in Osborn's algebra [Osb88, Osb89a, Osb89b]), it can be easily provided outside the algebra (i.e., create a new class and attach the query result to that class as a new instance). Direct instances of a kernel class (i.e., AGG, BIO, SET or SEQ), are very useful in three situations: (1) as operands in algebra operations that do not directly invoke user-defined behaviors, mainly as Select, Pickup, Combine, Union, Intersect, Subtract, Collapse and Assimilate; (2) as the operands that are defined to be invokers in message-sendings, such as the first operand of the Apply operator; and (3) as invokers in the message-sendings specified in the predicate of Select, Pickup and Apply operators. Notice that when objects are only used in the invoker side in message-sendings, they themselves do not have to have user-defined behaviors. Since direct instances of kernel classes do not have user-defined behaviors, they cannot be used as the receiver in the send operation $\leadsto$.

We shall make some remarks about each case in the above definition and give some examples. In case 1 (a) where s ⓘ CAPSULE, AGG or BIO, and r ⓘ CAPSULE, AGG or BIO, the Apply operation

$$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell}$$

returns the result of sending s to r when $p(x_s, x_r)$ is evaluated to **True**. This case can be thought of as a message-sending that can only occur if the invoker and the receiver can be qualified by predicate $p(x_s, x_r)$. This case can be illustrated by the following example.

Example 52: Suppose handle *SharePlan*1 denotes an instance of CEO-DECISION, which is a bi-object created by the CEO of the company to give a share of sales to employees.
Query: Use *SharePlan*1 to give *Mike* profit-sharing if *SharePlan*1 differentiates managers from average employees.

$$\rho_{x_s . if \asymp \text{"Is manager?"}}(SharePlan1, Mike)$$

where variable x_s will be substituted by the operand *SharePlan*1. As an illustration, if *SharePlan*1 has the value

{ *"Has seniority?"*, *"Share based on projects"*, *"Share based on hours"* }

then the predicate in above query will be evaluated to **False**, and *SharePlan*1 will not be applied to *Mike*. However, if *SharePlan*1 has the value

{ *"Is manager?"*, *"Share based on projects"*, *"Share based on hours"* }

then the predicate in above query will be evaluated to **True**, and *SharePlan*1 will be applied to *Mike*. Note that, since *SharePlan*1 is an object-identity which remains the same even when its value is changed, the above query evaluated at different points of time may return different results. ■

In case 1 (b) where s $\textcircled{i}$ CAPSULE, AGG or BIO, and r $\textcircled{i}$ SET, the Apply operation

$$\frac{\rho_{p(x_s,x_r)}(s, r)}{\ell}$$

returns a set object in which each member is the result of sending s to each member o in r when $p(s, o)$ is evaluated to **True**. Without the predicate, this case is similar to the *Image* operator in Shaw and Zdonik's algebra [SZ89], except that our invoker s (which they call a function) not only can invoke a single knowhow (when s $\textcircled{i}$ CAPSULE, such as *"Increase salary"*) but also can invoke more than one knowhow (when s $\textcircled{i}$ AGG or BIO, such as { *"Is manager?"*, *"Increase salary"*, *"pay overtime"* }). This case can be illustrated by the following example.

Example 53: Assume that class EMPLOYEE has these three behaviors with:

value *"Is manager?"*, signature () ⟶ BOOL
value *"Increase salary"*, signature (*Percent*, INT) ⟶ SELF

value *"Pay overtime"*, signature () $\longrightarrow$ SELF

Suppose the CEO of the company *Company* (which denotes a set of employees) cannot offer to give every employee a big raise except managers; but instead the CEO decides to pay non-manager employees the overtime they recorded. So the CEO creates a new bi-object under class CEO-DECISION denoted by handle *MakeEveryOneHappy*:

$$MakeEveryOneHappy := \{ \text{"}Is\ manager?\text{"}, \text{"}Increase\ salary\text{"}, \text{"}Pay\ overtime\text{"} \}$$

A very important point must be noted: the bi-object *MakeEveryOneHappy* is now stored as a persistent object in the database. The object thus becomes long lived knowledge which can be manipulated or reused (directly or associatively through queries) in the future. Furthermore, such objects may be saved under a user-created class which has user-defined behaviors, such as class CEO-DECISION which may have behaviors with values like "decision about hiring?". In this sense, such objects are fundamentally different from those stored SQL statements supported by some relational databases.

Now, the following query can be issued by the CEO:

Query: Give all managers in *Company* a 15% raise and all other employees overtime pay.

$$\frac{\rho(MakeEveryOneHappy, Company)}{\text{Percent} : 15}$$

Note that the predicate $p(x_s, x_r)$ in this operation is omitted because it is simply the constant **True**. Since both *"Increase salary"* and *"Pay overtime"* returns the receiver itself, the query returns a new set object containing all the employees in *Company*, each of whom has got either a 15% raise or some overtime pay. ■

In case 1 (c) where s ⓘ CAPSULE, AGG or BIO, and r ⓘ SEQ, the Apply operation

$$\frac{\rho_{p(x_s,x_r)}(s, r)}{\ell}$$

returns a sequence object in which each member is the result of sending s to each member o in r when $p(s, o)$ is evaluated to **True**. This case can be illustrated by the following example.

Example 54: Recall that class REPORT has the following representation:

$\dot{\Xi}$(REPORT) =<CHAPTER>

$\dot{\Xi}$(CHAPTER) = [Content: TEXT, Done: BOOL, PapersCited: PAPERS]

Let handle *MyReport* denote an instance of REPORT, that is, *MyReport* is a sequence of chapters. Assume that class CHAPTER has a behavior with

value *"Number of papers cited"*, signature () $\longrightarrow$ INT

which returns the number of papers cited in a chapter.

Query: Find out how many papers are cited in each finished chapter in *MyReport*. (Let x_r range over *MyReport*.)

$$\rho_{x_r.Done \times \mathbf{True}}(\text{"}\textit{Number of papers cited}\text{"}, \textit{MyReport})$$

This query produces a new sequence (object) of integers. Note that such a result often provides more information than a set of objects. For example, if this query returns the sequence $<32, 12, 7, 16, 28>$, then it also indicates that chapters at the beginning and the chapters at the end in *MyReport* seem to cite more papers than those chapters in the middle. ■

In case 2 (a) where s ⓘ SET, r ⓘ CAPSULE, AGG or BIO, the Apply operation

$$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell}$$

returns a set object in which each member is the result of sending each member o in s to object r (which is neither a set nor a sequence) when $p(o,r)$ is evaluated to **True**. This case can be illustrated by the following example.

Example 55: In Figure 8.20, class GET-CREDIT is a class whose instances are behaviors of class PERSON. Each instance of GET-CREDIT bears a hardcoded knowhow that knows how to calculate a type of tax credit for a person based on his/her personal information. Let handle *GetCredits* denote a set object containing the following instances of GET-CREDIT:

GetCredits = { *"Get child tax credit",*
"Get federal sales tax credit",
"Get refund of investment tax credit",
"Get forward averaging tax credit",
"Get provincial tax credit",
"Get Tax paid by installments",
"Get UI overpayment",
"Get CPP overpayment",
"Get GST credit" }

And all behaviors in *GetCredits* have the same signature () ⟶ INT. Assume that class GET-CREDIT itself also has some behaviors, one of which is the behavior with:

value *"About overpayment?"*, signature () ⟶ BOOL

whose semantics is to check whether the value of an instance of GET-CREDIT contains the keyword "overpayment", and, if so, return **True** otherwise **False**. Let *Mike*

denote an instance of PERSON.

Query: Find all the tax credits from $Mike$'s overpayment to the government. (Let x_1 range over $GetCredits$.)

$$\rho_{(\text{"About overpayment?"} \rightsquigarrow x_1) \asymp \textbf{True}}(GetCredits, Mike)$$

This query will produce a new set object containing two INT objects: $\{o_1, o_2\}$ where

$$\text{"Get UI overpayment"} \rightsquigarrow Mike \Rightarrow o_1$$

and

$$\text{"Get CPP overpayment"} \rightsquigarrow Mike \Rightarrow o_2$$

because the predicate

$$(\text{"About overpayment?"} \rightsquigarrow x_1) \asymp \textbf{True}$$

only qualifies two objects from $GetCredits$, one with value "*Get UI overpayment*" and one with value "*Get CPP overpayment*". So only these two objects are sent to the second operand $Mike$. ■

In case 2 (b) where s ⓘ SET and r ⓘ SET, the Apply operation

$$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell}$$

returns a set of set objects, each of which contains the results of sending each member o_s in s to a member o_r in r when $p(o_s, o_r)$ is evaluated to **True**. Practically speaking, this case may be the most useful case for the Apply operator, since it supports set-oriented behavior invocation with qualified receivers and invokers. This is one of the most important features that make the KBO algebra so unique when compared with existing OODB query models. This case can be illustrated by the following example.

Example 56: Let us use the previous example. Let handle $MyGroup$ denote a set of PERSON objects.

Query: For every person in $MyGroup$, find all his/her tax credits which can also be claimed, if over \$1000, by his/her spouse. (Let x_s range over $GetCredits$ and x_r range over $MyGroup$.)

$$\rho_{(x_s \rightsquigarrow x_r.Spouse) > 1000}(GetCredits, MyGroup)$$

This query returns a set (object) of set objects, each of which contains a set of integers representing the tax credits for each person in $MyGroup$ which can also be claimed,

if over \$1000, by the person's spouse. It must be noted that each (person) receiver in *MyGroup* may have received a different message or messages because messages are selected associatively based on the predicate $(x_s \rightsquigarrow x_r.Spouse) > 1000$. For example, if $MyGroup = \{Mike, John\}$, and if for *Mike* only the following is true:

$$(\text{"Get federal sales tax credit"} \rightsquigarrow Mike.Spouse) > 1000$$

and if for *John* only the following are true:

$$(\text{"Get child tax credit"} \rightsquigarrow John.Spouse) > 1000$$

$$(\text{"Get tax paid by installments"} \rightsquigarrow John.Spouse) > 1000$$

then the above query would return a set object with the following value:

$$\{\ \{o_1\},\ \{o_2, o_3\}\ \}$$

where

$$\text{"Get federal sales tax credit"} \rightsquigarrow Mike \Rightarrow o_1$$

$$\text{"Get child tax credit"} \rightsquigarrow John \Rightarrow o_2$$

$$\text{"Get tax paid by installments"} \rightsquigarrow John \Rightarrow o_3$$

This example clearly demonstrates the advantages of our Apply operator. In applications (like Income Tax) where there may exist a large number of hardcoded behaviors, queries like the above one are obviously desirable, because invokers to be applied can be dynamically adjusted based on certain relationships between the desired message and the desired receiver, such as the predicate $(x_s \rightsquigarrow x_r.Spouse) > 1000$ in the above query. To the best of our knowledge, the KBO algebra is the first OODB query model to support this kind of query. Furthermore, our queries can also avoid undesirable side-effects caused by message-sendings in the predicate. For example, suppose some behaviors in *GetCredits* have side-effects (but still return an integer). Then, we can modify the above query to be the following one:

$$\rho_{(x_s \rightsquigarrow \dot{x}_r.Spouse) > 1000}(GetCredits, MyGroup)$$

where the dotted variable $\dot{x}_r$ ensures that no receiver from *MyGroup* is actually affected by the possible side-effects caused by message x_s. ■

In case 2 (c) where s ⓘ SET and r ⓘ SEQ, the Apply operation

$$\frac{\rho_{p(x_s, x_r)}(s, r)}{\ell}$$

returns a sequence of set objects, each of which contains the results of sending each member o_s in s to a member o_r in r when $p(o_s, o_r)$ is evaluated to **True**. This case is quite similar to case 2.(b) except that all the results of message-sendings are contained in a new sequence object. Note that the new sequence object preserves the ordering in the sequence object r. For example, if $s = \{s_1, s_2, \}$ and $r = < r_1, r_2 >$, then the Apply operation

$$\frac{\rho(s,r)}{\ell}$$

returns a new sequence object (i.e., a new identity) with value

$$< \{o_1, o_2, \}, \ \{o_3, o_4\} >$$

where $\frac{s_1}{\ell} \rightsquigarrow r_1 \Rightarrow o_1$, $\frac{s_2}{\ell} \rightsquigarrow r_1 \Rightarrow o_2$, $\frac{s_1}{\ell} \rightsquigarrow r_2 \Rightarrow o_3$, and $\frac{s_2}{\ell} \rightsquigarrow r_2 \Rightarrow o_4$. The Apply operator in this case is useful for sending a set of qualifiable invokers to a sequence of qualifiable receivers.

In case 3.(a) where s ⓘ SEQ and r ⓘ CAPSULE, AGG or BIO, the Apply operation

$$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell}$$

returns a new sequence of objects, each of which is the result of sending each member s_i in s to the single receiver r when $p(s_i, r)$ is evaluated to **True**. The new sequence object preserves the ordering of the sequence operand s. This case can be illustrated by the following example.

Example 57: Assume the class SCIENTIST has three behaviors with:

value *"Bachelors from"*, signature () ⟶ TEXT
value *"Masters from"*, signature () ⟶ TEXT
value *"Doctorate from"*, signature () ⟶ TEXT

Let handle *Mary* denote an instance of SCIENTIST, and handle *GetDegrees* denote a sequence object with value < *"Bachelors from"*, *"Masters from"*, *"Doctors from"* >.

Query: List all degrees of scientist *Mary* from the lowest one to the highest one.

$$\rho(GetDegrees, Mary)$$

This query returns a new sequence object with a value something like this:

< *"BMath Waterloo"*, *"MSc Harvard"*, *"PhD Carnegie Mellon"* >

In case 3.(b) wher $\cdot$;) SEQ and r ⓘ SET, the Apply operation

$$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell}$$

returns a new set of sequence objects, each of which is the sequence of the results of sending each member s_i (from s_1 to s_n) in s to a member o_r in r when $p(s_i, o_r)$ is evaluated to **True**. In this way, each such sequence object still preserves the ordering of the sequence object s, and all these sequence objects are contained in a new set object produced by the operation. This case allows us to use a sequence of invokers to invoke a sequence of results for each object in a set (object). This case can be illustrated by the following example.

Example 58: Let handle *AIgroup* denote a set of AI scientists. Let handle *GetDegrees* denote the same (sequence) object in the previous example.
Query: List all degrees of scientists in *AIgroup* from the lowest one to the highest one.

$$\rho(GetDegrees, AIgroup)$$

This query returns a new set object with a value something like this:

{ < *"BSc Waterloo", "MSc Toronto",* **null** >,
< *"BMath Waterloo", "MSc Harvard", "PhD Carnegie Mellon"* >,
$\ldots, \ldots$ }

Such a result can be used to create the staff document of a research lab. ■

In the last case 3 (c) where s ⓘ SEQ and r ⓘ SEQ, the Apply operation

$$\frac{\rho_{p(x_s,x_r)}(s,r)}{\ell}$$

returns a new sequence of sequence objects, each of which is the sequence of the results of sending each member s_i (from s_1 to s_n) in s to a member o_r in r when $p(s_i, o_r)$ is evaluated to **True**. This case is very similar to case 3 (b), except that here the result object is a sequence object that preserves the ordering of the sequence object r. In the above example, if *AIgroup* was a sequence of scientists ordered by their highest degrees, then the query

$$\rho(GetDegrees, AIgroup)$$

might have returned a result like this:

< < *“BMath Waterloo”, “MSc Harvard”, “PhD Carnegie Mellon”* >,
< *“BSc Waterloo”, “MSc Toronto”*, **null** >,
. . . , . . . >

where the first sub-sequence is the information extracted from a scientist with a doctorate and the second sub-sequence is the information extracted from a scientist with a masters. In this way, the ordering of *GetDegrees* is preserved in sub-sequences, and the ordering of *AIgroup* is preserved in the sequence containing those sub-sequences. ■

The Apply operator can be extended based on different message-sending primitives, denoted by the following expression:

$$\frac{\overset{\nu}{\rho}_{p(x_s,x_r)}(s,r)}{\ell}$$

where ν is either an integer $n \geq 1$ or $*$, and the semantics of these operators is defined by simply replacing all the $\rightsquigarrow$ primitives in the above definition with the primitive $\overset{\nu}{\rightsquigarrow}$.

8.2.4 Expression Apply (λ)

The Expression Apply operator, λ, is derived from the Apply operator proposed in Osborn's object algebra [Osb88, Osb89a, Osb89b] with the extension to sequence operands. The Expression Apply operator is mainly intended to support the application of a single KBO *algebra* expression to each member of a set object or a sequence object. However, to make the operator more "polymorphic", we also allow this operator to apply an algebra expression to a single target (a capsule object, an aggregate object or a bi-object). The advantage provided by the Expression Apply operator is that it allows us to first build an arbitrary algebra expression and then iterate it though a set/sequence to get a set/sequence of desired results. While the Apply operator applies an invoker or a set of invokers to a receiver or a set of receivers, the Expression Apply operator applies an algebra expression (not an object) to an operand or members in an operand.

Definition 67: *The Expression Apply Operator* λ. Let r be an KBO object (precisely, an object-identity). Let $exp(x)$ denote a KBO algebra expression that takes an object x as input. The Expression Apply operation, denoted

$$\lambda_{exp(x)}(r)$$

is defined as follows:

1. If r ⓘ CAPSULE, AGG or BIO, then

$$\lambda_{exp(x)}(r) \triangleq exp(r)$$

The result is the object produced by the algebra expression $exp(r)$.

2. If r ⓘ SET, then

$$\lambda_{exp(x)}(r) \triangleq \{exp(o) \mid o \in r\}$$

The result is a direct instance of the kernel class SET.

3. If r ⓘ SEQ, $r = <r_1, \ldots, r_n>$, then

$$\lambda_{exp(x)} \triangleq <exp(r_1)> \star \cdots \star <exp(r_n)>$$

The result is a direct instance of the kernel class SEQ.

Example 59: The class PERSONAL-QUESTION shown in Figure 8.20 is a class whose instances are also behaviors of class PERSON. For example, an object with value *"Married?"* is an instance of PERSONAL-QUESTION, and sending *"Married?"* to a person can cause the person to answer **True or False**. Assume class PERSONAL-QUESTION also has its own behaviors, one of which is the behavior with:

value *"About what?"*, signature () ⟶ TEXT

whose semantics is to return the category (such as "Children", "Marriage", "Health") of a personal question. Let handle *PersonalQuestions* denote a set of instances of PERSONAL-QUESTION. Let the algebra expression $exp(x)$ be built as follows:

$$exp(x) = \rho_{(\text{"About what?"} \rightsquigarrow z_s) \asymp \text{"Children"}}(PersonalQuestions, \delta_{(\text{"Married?"} \rightsquigarrow y) \asymp \mathbf{True}}(x))$$

where z_s ranges over *PersonalQuestions* in the Apply operation and y ranges over x in the Pickup operation. This expression says "pick up a married person from set object x and ask this person some personal questions about children". More precisely, the Pickup operation chooses a married person y from set x and the Apply operation applies all invokers z_s from set *PersonalQuestions* that is about children (e.g., *"Which schools do your children go to?"*) to y. The result from this expression will be a set object containing all the answers from person y about his/her children. Now, let handle $MyLab = \{group_1, group_2, group_3\}$ where $group_1, group_2, group_3$ all denote a set of PERSON objects.

Query: Find a married person from each group in *MyLab* and collect all his/her answers for personal questions about children.

$$\lambda_{exp(x)}(MyLab)$$

This operation applies the expression $exp(x)$ to each member of $MyLab$. That is, the evaluation would be equivalent to evaluating

$$\{exp(group_1),\ exp(group_2),\ exp(group_3)\}$$

and the result of this query is a new SET object with value

$$\{answers_1,\ answers_2,\ answers_3\}$$

where $answers_i$ denotes a set (object) of answers from $person_i$ about his/her children. ■

The Expression Apply operator is most useful for manipulating the objects that are a set of sets, a set of sequences, a sequence of sets, or a sequence of sequences, because it can apply an algebra expression to each operand's member (which is also a set or a sequence). In the above example, the operand $MyLab$ is a set of sets.

8.2.5 Project (π)

The Project operation allows us to extract any information derivable through the behaviors of a single object (objects from a set or a sequence). Relationships between extracted results are represented in the form of aggregate objects or bi-objects. Our Project operator is similar to the Project operator proposed in [SZ89], with four important extensions: (1) Invokers in our Project are not a single function name (or a lambda expression representing a computation [SZ89]); instead, they can be any KBO objects or results from any KBO algebra expressions. (2) the if-then-else relationship can also be created through our Project operator in the form of bi-objects. (3) Invokers in our Project can take, besides the receiver, some labeled arguments provided in the blackboard ℓ. (4) Our Project operator not only operates on a set of receivers, but also operates on a single receiver or a sequence of receivers.

Definition 68: *The Project Operator* π. Let ℓ be a blackboard and $s_1, \ldots, s_n, r$ be any KBO objects (precisely, their object-identities). Let $A_1, \ldots, A_n \in \mathcal{A}$ be any attribute labels such that $\forall i, j \in [1..n] : i \neq j \Longrightarrow A_i \neq A_j$. The Project operation, denoted

$$\frac{\pi_{A_1,\ldots,A_n}(s_1, \ldots, s_n, r)}{\ell} \quad \text{or} \quad \frac{\pi_{if,then,else}(s_1, s_2, s_3, r)}{\ell}$$

where $n \geq 1$, is defined as follows:

1. If r ⓘ CAPSULE, AGG or BIO, then

$$\frac{\pi_{A_1,\ldots,A_n}(s_1,\ldots,s_n,r)}{\ell} \triangleq [A_1\colon \frac{s_1}{\ell} \leadsto r,\ldots,A_n\colon \frac{s_n}{\ell} \leadsto r]$$

$$\frac{\pi_{if,then,else}(s_1,s_2,s_3,r)}{\ell} \triangleq \{\!|\ \frac{s_1}{\ell} \leadsto r, \frac{s_2}{\ell} \leadsto r, \frac{s_3}{\ell} \leadsto r\ |\!\}$$

The result is a direct instance of the kernel class AGG or BIO.

2. If r ⓘ SET, then

$$\frac{\pi_{A_1,\ldots,A_n}(s_1,\ldots,s_n,r)}{\ell} \triangleq \{[A_1\colon \frac{s_1}{\ell} \leadsto o,\ldots,A_n\colon \frac{s_n}{\ell} \leadsto o] \mid o \in r\}$$

$$\frac{\pi_{if,then,else}(s_1,s_2,s_3,r)}{\ell} \triangleq \{\{\!|\ \frac{s_1}{\ell} \leadsto o, \frac{s_2}{\ell} \leadsto o, \frac{s_3}{\ell} \leadsto o\ |\!\} \mid o \in r\}$$

The result is a SET object containing AGG or BIO objects.

3. If r ⓘ SEQ, $r = <o_1,\ldots,o_n>$, then

$$\frac{\pi_{A_1,\ldots,A_n}(s_1,\ldots,s_n,r)}{\ell} \triangleq o'_1 \star \cdots \star o'_n$$

where

$$o'_i = <[A_1\colon \frac{s_1}{\ell} \leadsto o_i,\ldots,A_n\colon \frac{s_n}{\ell} \leadsto o_i]>$$

and

$$\frac{\pi_{if,then,else}(s_1,s_2,s_3,r)}{\ell} \triangleq o'_1 \star \cdots \star o'_n$$

where

$$o'_i = <\{\!|\ \frac{s_1}{\ell} \leadsto o_i, \frac{s_2}{\ell} \leadsto o_i, \frac{s_3}{\ell} \leadsto o_i\ |\!\}>$$

The result is a SEQ object containing AGG or BIO objects.

Example 60: Let class FAMILY have the following representation:

$\dot{\Xi}$(FAMILY) = [Father: PERSON, Mother: PERSON, Children: PERSONS]

And class FAMILY has five behaviors with:

value *"Get family income"*, signature () ⟶ INT
value *"Get family size"*, signature () ⟶ INT
value *"get Father"*, signature () ⟶ PERSON
value *"get Mother"*, signature () ⟶ PERSON
value *"get Children"*, signature () ⟶ PERSONS

The first behavior calculates the total of everyone's income in the family, and the second behavior calculates the number of people in a family. The other three behaviors can be simply attribute accessors. Let handle $StaffFamilies$ denote a set of FAMILY objects:

Query 1: Find the family income and size of all families in $StaffFamilies$.

$$\pi_{Income,Size}(\text{"}Get\ family\ income\text{"}, \text{"}Get\ family\ size\text{"}, StaffFamilies)$$

Here the blackboard is omitted because no argument is needed. The query returns a direct instance of the kernel class SET, which is a set of AGG objects with two attributes Income and Size, such as the set

$$\{[Income\!: 40500, Size\!: 5],\ [Income\!: 35800, Size\!: 4]\}$$

Notice that, although such new AGG objects do not have user-defined behaviors yet, they can still be used as input for other algebra operations like Select and Pickup, or as invokers (if they are intended to be invokers) in message-sendings (including those caused by Apply and Project operators).

Query 2: Find the family-heads of of all families in $StaffFamilies$ such that (from now on) the family-head of a family will always be the mother if the family has children and the father otherwise.

$FamilyHeads := \pi_{if,then,else}(\text{"}get\ Children\text{"}, \text{"}get\ Mother\text{"},$
$\text{"}get\ Father\text{"}, StaffFamilies)$

The result denoted by handle $FamilyHeads$ is a set of bi-objects. We shall use this result to explain why such bi-objects are indeed desirable in real-world applications. Suppose $FamilyHead1$ is one of the bi-objects in $FamilyHeads$, and it has value

$$(\ id_1, id_2, id_3\ )$$

where id_1 is the identity of a set object extracted from the Children attribute of $Family_1$, id_2 is the identity of an aggregate object extracted from the Mother attribute of $Family_1$, and id_3 is the identity of an aggregate object extracted from the Father attribute of $Family_1$. It must be noted that we, as end-users, do not know whether identity id_1 has an empty value (i.e., $\{\}$, the ***null*** object for sets) or not. Futhermore, after $FamilyHead1$ is created by the above query, object id_1, during its life time, may be updated every now and then from having an empty value to having a non-empty value (e.g., a child is born in the family), or from having a non-empty

value to having an empty value (e.g., the only child is killed in a car accident). However, no matter how its value is changed, the identity of this object remains the same, that is, id_1. Therefore, message-sendings like

$$\text{“}Claim\ spouse's\ tax\ credits\text{”} \overset{1}{\rightsquigarrow} FamilyHead1$$

will always send the message to the "right family-head" no matter what has happened to the value of id_1. In this particular message-sending, if id_1 currently has an empty value (i.e., the family currently has no children), then the message *"Claim spouse's tax credits"* is sent to the current family head id_3 (the father); if id_1 currently has a non-empty value (i.e., the family currently has some children), then the message *"Claim spouse's tax credits"* is sent to the current family head id_2 (the mother). Such automatic adjustable features of bi-objects as receivers are not explicitly supported in any other OODB models. Note that we can use the extended Apply operator $\overset{1}{\rho}$ to send this message to each family-head in the set $FamilyHeads$:

$$\overset{1}{\rho}\,(\text{“}Claim\ spouse's\ tax\ credits\text{”}, FamilyHeads)$$

The Project operator can be extended based on different message-sending primitives, denoted by the following expression:

$$\frac{\overset{\nu}{\pi}_{A_1,\ldots,A_n}(s_1,\ldots,s_n,r)}{\ell} \quad \text{or} \quad \frac{\overset{\nu}{\pi}_{if,then,else}(s_1,s_2,s_3,r)}{\ell}$$

where ν is either an integer $n \geq 1$ or $*$, and the semantics of these operators is defined by simply replacing all the $\rightsquigarrow$ primitives in the above definition with the primitive $\overset{\nu}{\rightsquigarrow}$. We illustrate such extended operators with an example.

Example 61: Let us continue the above example with the following objects:

$StaffFamilies = \{Family_1, Family_2\}$
$Family_1 =$ [Father: o_1, Mother: o_2, Children: $\{o_3, o_4, o_5\}$]
$Family_2 =$ [Father: o_6, Mother: o_7, Children: $\{o_8, o_9\}$]
$o_{10} = <\text{“get Father”}, \text{“get Age”}>$
$o_{11} = <\text{“get Mother”}, \text{“get Age”}>$
$o_{12} = <\text{“get Children”}, \text{“get Age”}>$

Note that objects o_{10}, o_{11}, o_{12} may share the same object with value "get Age". Assume that class PERSON has a behavior with value *"get Age"* and signature

$() \longrightarrow$ INT.

Query: Find ages of all people in all families in *StaffFamilies*.

$$\overset{1}{\pi}_{FatherAge,MotherAge,KidsAges}\,(o_{10}, o_{11}, o_{12}, StaffFamilies)$$

Suppose this query returns a new SET object with value:

{ [FatherAge: 46, MotherAge: 44, KidsAges: { 18, 16, 13 }],
[FatherAge: 34, MotherAge: 33, KidsAges: { 10, 9 }] }

More precisely, the first aggregate object

$$[FatherAge : 46, MotherAge : 44, KidsAges : \{18, 16, 13\}]$$

would be the result of evaluating

$$[FatherAge\text{: } o_{10} \overset{1}{\leadsto} Family_1, MotherAge\text{: } o_{11} \overset{1}{\leadsto} Family_1, KidsAges\text{: } o_{12} \overset{1}{\leadsto} Family_1]$$

Here, $o_{10} \overset{1}{\leadsto} Family_1$ is equivalent to

$$\text{“}get\ Age\text{”} \overset{1}{\leadsto} (\text{“}get\ Father\text{”} \overset{1}{\leadsto} Family_1)$$

which is evaluated to

$$\text{“}get\ Age\text{”} \overset{1}{\leadsto} o_1$$

and returns the age (46) of the father o_1. Similar evaluation happens to $o_{11} \overset{1}{\leadsto} Family_1$. However, for message-sending $o_{12} \overset{1}{\leadsto} Family_1$, which is equivalent to

$$\text{“}get\ Age\text{”} \overset{1}{\leadsto} (\text{“}get\ Children\text{”} \overset{1}{\leadsto} Family_1)$$

the result of "*get Children*" $\overset{1}{\leadsto} Family_1$ is a set object $\{o_3, o_4, o_5\}$, which is a set of persons. This set object itself (i.e., an instance of PERSONS) does not understand the message "*get Age*", because class PERSONS does not have a behavior with value "*get Age*". Therefore, the shallow-send operator $\overset{1}{\leadsto}$ causes "*get Age*" $\overset{1}{\leadsto} \{o_3, o_4, o_5\}$ to become

$$\{\text{“}get\ Age\text{”} \overset{1}{\leadsto} o_3, \text{“}get\ Age\text{”} \overset{1}{\leadsto} o_4, \text{“}get\ Age\text{”} \overset{1}{\leadsto} o_5\}$$

which returns the set object $\{18, 16, 13\}$ as the attribute KidsAges. ■

As we have seen in this example, the extended Project operators can provide more expressive power for our queries.

8.2.6 Combine (χ)

Many real-world relationships between objects can be represented by objects and their sub-objects (i.e., the objects they reference through identities). However, it is impossible to envision all relationships (that are needed during a query session) between objects at the time when their classes are created [AG89]. Hence there is a need for an explicit way to handle cases when a relationship is not defined in a class. The Combine operator is an explicit operator used to create relationships between an object or a set/sequence of objects and another object or a set/sequence of objects. Practically, the Combine operation is most useful for combining information from two sets/sequences of objects.

The Combine operator is defined essentially as a Cartesian product of (1) two single objects, (2) two sets of objects, (3) two sequences of objects, (4) a single object and a set of objects, (5) a single object and a sequence of objects, and (6) a set of objects and a sequence of objects. In such a way, our Combine operator can operate on any two KBO objects, regardless of what classes they belong to. Just as the Project operator, we also extend the Combine operator to combine information in a bi-object. As an analogy to the definition of the regular Combine operator, this new version of the Combine operator is basically a Cartesian product among three operands.

Definition 69: *The Combine Operator* χ. Let ℓ be a blackboard and s, r, t be any three objects (precisely, two object-identities). Let A, B be two attribute labels such that $A \neq B$. The Combine operation, denoted

$$\chi_{A,B}(s,r) \quad \text{or} \quad \chi_{if,then,else}(t,s,r,)$$

is defined as follows (only for the first Combine operator):

1. If s ⓘ CAPSULE, AGG or BIO, then

   (a) if r ⓘ CAPSULE, AGG or BIO, then

   $$\chi_{A,B}(s,r) \triangleq [A: s, B: r]$$

   The result is a new AGG object.

   (b) if r ⓘ SET, then

   $$\chi_{A,B}(s,r) \triangleq \{[A: s, B: o] \mid o \in r\}$$

   The result is a new SET object containing AGG objects.

(c) if r ⓘ SEQ and $r = <r_1, \ldots, r_n>$, then

$$\chi_{A,B}(s,r) \triangleq <[A\colon s, B\colon r_1], \ldots, [A\colon s, B\colon r_n]>$$

The result is a new SEQ object containing AGG objects.

2. If s ⓘ SET, then

(a) if r ⓘ CAPSULE, AGG or BIO, then

$$\chi_{A,B}(s,r) \triangleq \{[A\colon o, B\colon r] \mid o \in s\}$$

The result is a new SET object containing AGG objects.

(b) if r ⓘ SET, then

$$\chi_{A,B}(s,r) \triangleq \{[A\colon o_s, B\colon o_r] \mid o_s \in s \wedge o_r \in r\}$$

The result is a new SET object containing AGG objects.

(c) if r ⓘ SEQ and $r = <r_1, \ldots, r_n>$, then

$$\chi_{A,B}(s,r) \triangleq \{<[A\colon o_s, B\colon r_1], \ldots, [A\colon o_s, B\colon r_n]> \mid o_s \in s\}$$

The result is a new SET object containing sequences of AGG objects.

3. If s ⓘ SEQ and $s = <s_1, \ldots, s_n>$, then

(a) if r ⓘ CAPSULE, AGG or BIO, then

$$\chi_{A,B}(s,r) \triangleq <[A\colon s_1, B\colon r], \ldots, [A\colon s_n, B\colon r]>$$

The result is a new SEQ object containing AGG objects.

(b) if r ⓘ SET, then

$$\chi_{A,B}(s,r) \triangleq <\{[A\colon s_1, B\colon o_r] \mid o_r \in r\}, \ldots, \{[A\colon s_n, B\colon o_r] \mid o_r \in r\}>$$

The result is a new SEQ object containing sets of AGG objects.

(c) if r ⓘ SEQ and $r = <r_1, \ldots, r_m>$, then

$$\begin{aligned}\chi_{A,B}(s,r) \triangleq <&[A\colon s_1, B\colon r_1], [A\colon s_1, B\colon r_2], \ldots, [A\colon s_1, B\colon r_m],\\ &[A\colon s_2, B\colon r_1], [A\colon s_2, B\colon r_2], \ldots, [A\colon s_2, B\colon r_m],\\ &\ldots,\\ &[A\colon s_n, B\colon r_1], [A\colon s_n, B\colon r_2], \ldots, [A\colon s_n, B\colon r_m]>\end{aligned}$$

The result is a new SEQ object containing AGG objects.

■

Note that the definition for the other Combine operator $\chi_{if,then,else}(t, s, r,)$ is too tedious to be given here. Basically speaking, this Combine operator is defined in a manner very similar to the definition of $\chi_{A,B}(s, r)$. As an illustration, if t, s, r are all instances of CAPSULE, AGG or BIO, then

$$\chi_{if,then,else}(t, s, r,) \triangleq \{\!\!\{ t, s, r \}\!\!\}$$

and if t, s, r are all instances of SET, then

$$\chi_{if,then,else}(t, s, r,) \triangleq \{\{\!\!\{ o_t, o_s, o_r \}\!\!\} \mid o_t \in t \wedge o_s \in s \wedge o_r \in r\}$$

These two cases are probably most useful in practice. The main purpose of this Combine operator is to support associative generation of new bi-objects.

In the following, we further explain the cases of the Combine operator $\chi_{A,B}(s, r)$. In case 1 (a), the Combine operator allows us to simply generate a new AGG object that combines the two operands in its two attributes For example,

$$\chi_{MyFriend,Speaks}(Mike, \text{“}French\text{”})$$

simply creates a new AGG object [*MyFriend*: *Mike*, *Speaks*: “*French*”]. Note that the Combine operator could be extended to operate on n operands instead of two operands. In the present KBO algebra, however, we will stay with two operands for the sake of simplicity.

In case 1 (b) and case 2 (a), the Combine operator allows us to combine a single object with a set of objects. A new SET object is generated, which contains a set of two-attribute aggregate objects. For example, if $MyGroup = \{Mike, Mary\}$, then

$$\chi_{MyFriend,Speaks}(MyGroup, \text{“}French\text{”})$$

returns a new SET object with value:

{ [*MyFriend*: *Mike*, *Speaks*: “*French*”], [*MyFriend*: *Mary*, *Speaks*: “*French*”] }

Note that the TEXT objects *“French”* in both aggregate objects are the same object, that is, they are identical. This implies that if we later update the TEXT object from *“French”* to *“French and Spanish”*, then all aggregate objects in the above set object will be updated at once.

In case 1.(c) and case 3.(a), the Combine operator allows us to combine a single object with a sequence of objects. A new SEQ object is generated, which contains a sequence of two-attribute aggregate objects. For example, if $MyReport = <$

$chapter_1, chapter_2 >$, then

$$\chi_{Chapter, ProofReadBy}(MyReport, Mary)$$

returns a new SEQ object with value

$$<[Chapter\colon chapter_1, ProofReadBy\colon Mary], [Chapter\colon chapter_2, ProofReadBy\colon Mary]>$$

where $Mary$ is an object handle denoting a PERSON object.

The Combine operator in case 2.(b) is quite similar to the conventional Cartesian product — it creates relationships between objects from two set of objects. A new SET object is generated, containing a set of two-attribute aggregate objects. For example, if we have two set objects: $MyGroup = \{Mike, Mary\}$ and $Languages = \{\text{"English"}, \text{"French"}\}$, then

$$\chi_{MyFriend, Speaks}(MyGroup, Languages)$$

returns a new SET object with value

$$\{\ [MyFriend\colon Mike, Speaks\colon \text{"English"}], [MyFriend\colon Mary, Speaks\colon \text{"English"}],$$
$$[MyFriend\colon Mike, Speaks\colon \text{"French"}], [MyFriend\colon Mary, Speaks\colon \text{"French"}]\ \}$$

A natural extension of case 2 (b) is case 3 (c) where the Combine operator creates relationships between objects from two sequences of objects. In case 3 (c), a new SEQ object is generated, containing a sequence of two-attribute aggregate objects. The overall ordering of this new sequence is based on the first operand's ordering, and the ordering of the second operand is preserved by each of n sub-groups within the result sequence. For example, if we have $MyReport = < chapter_1, chapter_2 >$ and $TopSpellers = < Mike, Mary >$ (i.e., $Mike$ is a better speller than $Mary$), then

$$\chi_{Chapter, ProofReadBy}(MyReport, TopSpellers)$$

returns a new SEQ object with value

$$<[Chapter\colon chapter_1, ProofReadBy\colon Mike],$$
$$[Chapter\colon chapter_1, ProofReadBy\colon Mary],$$
$$[Chapter\colon chapter_2, ProofReadBy\colon Mike],$$
$$[Chapter\colon chapter_2, ProofReadBy\colon Mary]>$$

Both the first two AGG objects and the last two AGG objects preserve the ordering of *TopSpellers* (i.e., *Mike* is used before *Mary*). The ordering of *MyReport* is preserved by all the AGG objects (i.e., $chapter_1$ is taken before $chapter_2$). If we switch the two operands, then the query

$$\chi_{ProofReadBy,Chapter}(TopSpellers, MyReport)$$

returns a new SEQ object with value

$$\begin{aligned} < &[Chapter\colon chapter_1, ProofReadBy\colon Mike], \\ &[Chapter\colon chapter_2, ProofReadBy\colon Mike], \\ &[Chapter\colon chapter_1, ProofReadBy\colon Mary], \\ &[Chapter\colon chapter_2, ProofReadBy\colon Mary] > \end{aligned}$$

Notice that the order of the two attributes in AGG objects is immaterial.

The Combine operator in case 2.(c) or case 3.(b) creates relationships between objects from a set of objects and a sequence of objects. Case 2.(c) allows us to generate a set of sequences such that these sequences still preserve the ordering of the sequence operand. Case 3.(b) allows us to generate a sequence of sets such that the new sequence still preserves the ordering of the sequence operand. In this way, the end-user can choose the representation of the query result that best suits their intuition. In other words, we allow end-users to decide how they wish to preserve the ordering of the input sequence. These two cases are useful in situations where the end-user otherwise has to convert a sequence to a set (thus losing the ordering) in order to create relationships between objects from a sequence and a set. To the best of our knowledge, our algebra is the first object algebra to provide such capabilities. The following example illustrates the application of these two cases.

Example 62: Let handle *MyReport* denote an instance of REPORT, i.e., a sequence of CHAPTER objects. Recall that class CHAPTER has the following representation:

$\dot{\Xi}$(CHAPTER) = [Content: TEXT, Done: BOOL, PapersCited: PAPERS]

Assume that class PAPERS has a behavior with:

value *"Get all authors"*, signature () $\longrightarrow$ TEXT-SET

which returns a set of TEXT objects that are the names of all the authors whose papers are contained in an instance of PAPERS. Let $exp(x)$ be an algebra expression built as follows:

$$exp(x) \doteq \sigma_{y.Scientist.Name \in_{=}(\text{"All authors"} \leadsto y.Chapter.PapersCited)}(x)$$

which says "for each AGG object in set x if the name of the Scientist (attribute) is among the authors whose papers are cited in Chapter (attribute), then select this AGG object. Let handle *AIgroup* denote a set of scientists.

Query: For each chapter in *MyReport*, find all scientists in *AIgroup* whose paper or papers are cited in that chapter. For this query, if we want to list all the scientists for each chapter (from the first one to the last one), we can formulate the following query:

$$\lambda_{exp(x)}(\chi_{Chapter,Scientist}(MyReport, AIgroup))$$

This query returns a sequence of sets, each of which is a set of Chapter-Scientist pairs in which (each) Scientist's name is among the authors whose papers are cited in (the same) Chapter. As a comparison, if we want to list all the chapters (from the first one to the last one) for each scientist, we can formulate the following query:

$$\lambda_{exp(x)}(\chi_{Scientist,Chapter}(AIgroup, MyReport))$$

This query returns a set of sequences, each of which is a sequence of Chapter-Scientist pairs in which (the same) Scientist's name is among the authors whose papers are cited in (each) Chapter. ■

8.2.7 Union ($\cup$)

The Union operator, $\cup$, is mainly used to create new sets or sequences of objects. However, our Union operator is different from those Union operators in existing object algebras (such as [SZ89, SO90]), in that our Union operator can also be used to take the union of a set and a sequence or a sequence and another sequence. Note that the membership test $\in$ is based on the identity test $\equiv$.

Definition 70: *The Union Operator* $\cup$. Let s, r be any two objects (precisely, two object-identities). The Union operation, denoted

$$\cup(s, r)$$

is defined as follows:

1. If s Ⓘ SET and r Ⓘ SET, then

$$\cup(s,r) \triangleq \{o \mid o \in s \vee o \in r\}$$

   The result is a direct instance of SET.

2. If s ⓘ SET and r ⓘ SEQ, then

$$\cup(s,r) \triangleq \{o \mid o \in s \vee o \in r\}$$

The result is a direct instance of SET.

3. If s ⓘ SEQ, $s = <s_1, \ldots, s_n>$, and r ⓘ SEQ, $r = <r_1, \ldots, r_m>$, then

$$\cup(s,r) \triangleq <s_1, \ldots, s_n, r_1, \ldots, r_m>$$

The result is a direct instance of SEQ.

4. If s ⓘ SEQ, $s = <s_1, \ldots, s_n>$, and r ⓘ SET, then

$$\cup(s,r) \triangleq <s_1, \ldots, s_n, r_1, \ldots, r_m>$$

where $r_i \in r$ for $1 \leq i \leq m$ and the ordering of $r_1, \ldots, r_m$ is arbitrary. The result is a direct instance of SEQ.

5. In any other cases, $\cup(s,r) \triangleq$ ***null***.

■

The definition in case 1 is the conventional one. Union in case 2 destroys the ordering of the sequence operand. This case can be used to convert a sequence r to a set by performing $\cup(\{\ \}, r)$. Union in case 3 is a very useful one since it is essentially a concatenation operator for sequences. For example,

$$\cup(<chapter_1, chapter_2>, <chapter_3, chapter_4>)$$

returns a new SEQ object with value $< chapter_1, chapter_2, chapter_3, chapter_4 >$. Union in case 4 generates a new sequence in which at least the ordering of the sequence operand is preserved. Such a sequence result can be used as the input to a sorting behavior implemented in a sequence class. Union in case 4 can also be used to convert a set r to a sequence by performing $\cup(<>, r)$.

The Union operator can be extended to

$$\cup_{\asymp_n}(s,r) \qquad \text{or} \qquad \cup_{\asymp}(s,r)$$

and its semantics is defined by replacing all the (identity-based) membership tests $\in$ in the above definition with $\in_{\asymp_n}$ or $\in_{\asymp}$. For example, let $s = \{id_1, id_2\}$ and $r = \{id_3, id_4\}$. If $id_1 \not\equiv id_3$ but $id_1 \asymp id_3$, then $\cup(s,r) = \{id_1, id_2, id_3, id_4\}$ while $\cup_{\asymp}(s,r) = \{id_1, id_2, id_4\}$.

8.2.8 Intersect ($\cap$)

The Intersect operator, $\cap$, is also mainly used to create new sets or sequences of objects. This operator is different from the conventional Intersection operators, in that our Intersect operator can also operate on a set and a sequence or a sequence and another sequence.

Definition 71: *The Intersect Operator* $\cap$. Let s, r be any two objects (precisely, their identities). The Intersect operation, denoted

$$\cap(s,r)$$

is defined as follows:

1. If $s \textcircled{i}$ SET and $r \textcircled{i}$ SET, then

$$\cap(s,r) \triangleq \{o \mid o \in s \wedge o \in r\}$$

2. If $s \textcircled{i}$ SET and $r \textcircled{i}$ SEQ, then

$$\cap(s,r) \triangleq \{o \mid o \in s \wedge o \in r\}$$

3. If $s \textcircled{i}$ SEQ, $s = <s_1, \ldots, s_n>$, and $r \textcircled{i}$ SEQ, then

$$\cap(s,r) \triangleq s'_1 \star \cdots \star s'_n$$

   where

$$s'_i = \begin{cases} <s_i> & \text{if } s_i \in r \\ <> & \text{otherwise} \end{cases}$$

4. If $s \textcircled{i}$ SEQ, $s = <s_1, \ldots, s_n>$, and $r \textcircled{i}$ SET, then

$$\cap(s,r) \triangleq s'_1 \star \cdots \star s'_n$$

   where

$$s'_i = \begin{cases} <s_i> & \text{if } s_i \in r \\ <> & \text{otherwise} \end{cases}$$

5. In any other cases, $\cap(s,r) \triangleq$ ***null***.

■

Case 1 is the conventional definition. Like the Union operator, Intersect in case 2 destroys the ordering of the sequence operand. In other words, the sequence operand is treated as if it were a set operand. In case 3 and case 4, the Intersect operation

drops any object in the first operand (a sequence) that is not identical to any object in the second operand (a sequence or set). In both case 3 and case 4, the ordering of the first operand is preserved. For example,

$$\cap(<chapter_1, chapter_2, chapter_3, chapter_4>, <chapter_2, chapter_3>)$$

returns a new SEQ object with value $<chapter_2, chapter_3>$.

The Intersect operator can be extended to

$$\cap_{\asymp_n}(s,r) \qquad \text{or} \qquad \cap_{\asymp}(s,r)$$

and its semantics is defined by replacing all the (identity-based) membership tests $\in$ in the above definition with $\in_{\asymp_n}$ or $\in_{\asymp}$. For example, let $s = <id_1, id_2>$ and $r = <id_3, id_4>$. If $id_1 \not\equiv id_3$ but $id_1 \asymp id_3$, then $\cap(s,r) = <>$ while $\cap_{\asymp}(s,r) = <id_1>$.

8.2.9 Subtract (−)

Like the Union and Intersect operators, the Subtract operator, −, is mainly used to create new sets or sequences of objects. Again this operator can operate on a set and a sequence, or a sequence and another sequence.

Definition 72: *The Subtract Operator* −. Let s, r be any two objects (precisely, their identities). The Subtract operation, denoted

$$-(s,r)$$

is defined as follows:

1. If s ⓘ SET and r ⓘ SET, then

$$-(s,r) \triangleq \{o \mid o \in s \wedge o \notin r\}$$

2. If s ⓘ SET and r ⓘ SEQ, then

$$-(s,r) \triangleq \{o \mid o \in s \wedge o \notin r\}$$

3. If s ⓘ SEQ, $s = <s_1, \ldots, s_n>$, and r ⓘ SEQ, then

$$-(s,r) \triangleq s'_1 \star \cdots \star s'_n$$

where

$$s'_i = \begin{cases} <> & \text{if } s_i \in r \\ <s_i> & \text{otherwise} \end{cases}$$

4. If s $\textcircled{i}$ SEQ, $s = \quad s_1, \ldots, s_n >$, and r $\textcircled{i}$ SET, then

$$-(s,r) \triangleq s'_1 \star \cdots \star s'_n$$

where

$$s'_i = \begin{cases} <> & \text{if } s_i \in r \\ <s_i> & \text{otherwise} \end{cases}$$

5. In any other cases, $-(s,r) \triangleq$ ***null***.

Case 1 is the conventional definition. Case 2 allows us to drop objects from a set that are identical to some object in a sequence. Case 3 and case 4 allow us to drop objects from a sequence that are identical to some object in another sequence (case 3) or a set (case 4), while still preserving the ordering of the first sequence operand. For example

$$-(<chapter_1, chapter_2, chapter_3, chapter_4>, <chapter_2, chapter_3>)$$

returns a new SEQ object with value $<chapter_1, chapter_4>$.

The Subtract operator can be extended to

$$-_{\asymp_n}(s,r) \qquad \text{or} \qquad -_{\asymp}(s,r)$$

and its semantics is defined by replacing all the (identity-based) membership tests $\in$ in the above definition with $\in_{\asymp_n}$ or $\in_{\asymp}$. For example, let $s = <id_1, id_2>$ and $r = \{id_3, id_4\}$. If $id_1 \not\equiv id_3$ but $id_1 \asymp id_3$, then $-(s,r) = <id_1, id_2>$ while $-_{\asymp}(s,r) = <id_2>$.

8.2.10 Collapse (ϖ)

The Collapse operator is used to restructure sets of sets, sets of sequences, sequences of sequences, and sequences of sets. The Collapse operator extends the Flatten operator in [SZ89] or Set-Collapse operator in [VD91], in that it can "collapse" not only a set of sets but also a set of sequences, a sequence of sequences, a sequence of sets, and an aggregate of aggregates.

Definition 73: *The Collapse Operator* ϖ. Let r be any object (precisely, an object-identity). The Collapse operation, denoted

$$\varpi(r)$$

is defined as follows:

1. If r Ⓘ AGG and $r = [A_1{:}\ o_1, ..., A_k{:}\ o_k, ..., A_n{:}\ o_n]$ where $0 \leq k \leq n$ and (without loss of generality) for all i such that o_i Ⓘ AGG and $o_i = [B_{1_i}{:}\ o_{1_i}, ..., B_{m_i}{:}\ o_{m_i}]$, then

$$\varpi(r) \triangleq [B_{1_1}{:}\ o_{1_1}, ..., B_{m_1}{:}\ o_{m_1}, ..., B_{1_k}{:}\ o_{1_k}, ..., B_{m_k}{:}\ o_{m_k}, A_{k+1}{:}\ o_{k+1}, ..., A_n{:}\ o_n]$$

The result is a direct instance of AGG.

2. If r Ⓘ SET and $\forall o \in r : o$ Ⓘ SET, then

$$\varpi(r) \triangleq \{x \mid \exists o \in r : x \in o\}$$

The result is a direct instance of SET.

3. If r Ⓘ SET and $\forall o \in r : o$ Ⓘ SEQ, then

$$\varpi(r) \triangleq \{x \mid \exists o \in r : x \in o\}$$

The result is a direct instance of SET.

4. If r Ⓘ SEQ, $r = <r_1, \ldots, r_n>$, and $\forall i \in [1..n] : r_i$ Ⓘ SEQ, then

$$\varpi(r) \triangleq r_1 \star \cdots \star r_n$$

The result is a direct instance of SEQ.

5. If r Ⓘ SEQ, $r = <r_1, \ldots, r_n>$, and $\forall i \in [1..n] : r_i$ Ⓘ SET, then

$$\varpi(r) \triangleq r'_1 \star \cdots \star r'_n$$

where $r'_i = <r_{i_1}, ..., r_{i_m}>$ such that $r_{i_j} \in r_i$ and the ordering of r'_i is arbitrary. The result is a direct instance of SEQ.

6. In any other cases, $\varpi(r) \triangleq$ ***null***.

■

The first case, i.e., the collapse of aggregates in an aggregate, is useful in conjunction with the Combine operator. For example, if $o_1 = [A_1 : x_1, A_2 : x_2]$ and $o_2 = [B_1{:}\ y_1, B_2{:}\ y_2]$, the Combine operation

$$\chi_{A,B}(o_1, o_2)$$

returns a result with value

$$[A{:}\ [A_1{:}\ x_1, A_2{:}\ x_2], B{:}\ [B_1{:}\ y_1, B_2{:}\ y_2]]$$

which represents a new relationship between two other relationships (i.e., o_1 and o_2). In many situations, such a new relationship semantically makes a lot of sense. However, if this new relationship is not the intention of the end-user, a Collapse operation may be performed on this result, that is,

$$\varpi(\chi_{A,B}(o_1, o_2))$$

which returns a result with value

$$[A_1: x_1, A_2: x_2, B_1: y_1, B_2: y_2]$$

which represents a relationship among four objects (they may or may not be complex objects).

8.2.11 Assimilate (α)

The Assimilate operation eliminates the k-equal duplicates in a SET object or a SEQ object, whenever they are undesirable to the user. When applying this operator on a set/sequence object-identity, it returns a new object (identity) with a new set/sequence value in which members are pairwise not k-equal.

Definition 74: *The Assimilate Operator* α. Let r be any object (i.e., an object-identity). The Assimilate operation, denoted

$$\alpha_k(r)$$

where $k \geq 0$, is defined as follows:

1. If r ⓘ SET and $r = \{r_1, \ldots, r_n\}$, then

$$\alpha_k(r) \triangleq \{r_1, \ldots, r_m\}$$

   where $m \leq n$ and $\forall i, j \in [1..m]: i \neq j \Longrightarrow r_i \not\asymp_k r_j$.

2. If r ⓘ SEQ and $r = \langle r_1, \ldots, r_n \rangle$, then

$$\alpha_k(r) \triangleq r'_1 \star \cdots \star r'_n$$

   where

$$r'_i = \begin{cases} \langle\rangle & \text{if } \exists r_j, j < i : r_j \asymp_k r_i \\ \langle r_i \rangle & \text{otherwise} \end{cases}$$

3. In any other cases, $\alpha_k(r) \triangleq r$.

■

Example 63: Given a set object handle $Pals = \{id_1, id_2, id_3, id_4, id_5\}$, where objects $id_1, id_2, id_3, id_4, id_5$ are TEXT objects and have values "Lee", "Andy", "Lee", "Tom", and "Lee", respectively. Then the algebra expression

$$\alpha_1(Pals)$$

returns a new set object with value $\{id_1, id_2, id_4\}$. ■

Note that it is assumed that an instance of SET does not contain two identical objects. This means that if r is a set object then $\alpha_0(r)$ should return r itself.

8.3 Queries for Generating Complex Invokers

In the previous section, our query examples have demonstrated the advantage of sending qualified invokers to qualified receivers. Another advantage of the KBO algebra is its support for *associative generation of new complex behaviors that are immediately executable.* Intuitively, this statement can be justified by combining the following factors: (1) Invokers in our message-sending operation $\rightsquigarrow$ can be any KBO objects; in particular, whenever a complex object (the invoker) fails to invoke a single knowhow, it is then interpreted as a complex invoker which can invoke several knowhows based on the behavioral semantics implied by its data structure (aggregate, bi-object, set or sequence). Hence, such complex message-sendings can be viewed as complex behaviors. (2) Our Project and Combine operators can generate new AGG and BIO objects, and our Select, Union, Intersect and Subtract operators can generate new SET and SEQ objects. Since any new complex object implicitly represents a complex form of invoking ability when sent to receivers, we can say that our algebra operators generate new complex behaviors (or invokers). (3) If new complex behaviors are generated in sets or in sequences (e.g., by Project or Combine), then the Pickup operator allows us to choose the right complex invoker (i.e., the one satisfying the predicate) from the set/sequence. (4) Since any result produced by our algebra operations can be immediately used as the invoker in our message-sendings, we can say that our algebra supports associative generation of new complex behaviors that are immediately executable.

We must point out that the full power of this feature offered by the KBO algebra can be best explored in databases where a great number of complex invokers have been created (maybe over the years of usage of the database). Hence, we give the

following examples under the assumption that a large number of complex invokers have been created by end-users.

8.3.1 Examples of Generating new AGG Invokers

The following examples illustrate the generation of some new AGG invokers. Note that all the following algebra expressions can be nested together as one single expression; we separate them only for the sake of clarity. In this example, the end-user wishes to generate a set of new behaviors from *PersonalQuestions* (a set of instances of PERSONAL-QUESTION) that can return multiple answers about a single topic when sent to an employee. First, a Combine operation is performed:

$$h_1 := \chi_{Q1,Q2}(PersonalQuestions, PersonalQuestions)$$

The result is now denoted by handle h_1. Note that we can always use Collapse and Combine operators repeatedly to get more than two questions in one aggregate object. For simplicity, in this example we only combine two questions into each aggregate object in the result h_1. Next we select all the aggregate objects from h_1 that are questions about the same topic.

$$h_2 := \sigma_{\neg(x.Q1 \equiv x.Q2) \wedge \text{“About what?”} \leadsto x.Q1 \times \text{“About what?”} \leadsto x.Q2}(h_1)$$

The result, denoted by h_2, is a set of AGG objects that can be used as complex invokers sent to EMPLOYEE objects to obtain more than one answer about the same topic. For example, we choose a complex invoker from h_2 about children:

$$h_3 := \delta_{(\text{“About what?”} \leadsto x.Q1) \times \text{“Children”}}(h_2)$$

and h_3 may have a value which looks like this:

$$[Q1\text{: “Child care cost”}, Q2\text{: “Children ages”}]$$

We can send h_3 immediately to employee *Mike*:

$$h_3 \leadsto Mike$$

According to the behavioral semantics of an AGG invoker, this message-sending returns a two-attribute aggregate, one holding the result of $x.Q1 \leadsto Mike_1$ and one holding the result of $x.Q2 \leadsto Mike_2$, where $Mike_1$ and $Mike_2$ are deep-copies of *Mike*. For example, the whole result may have a value like this

$$[Q1\text{: } 2000, Q2\text{: } \{6, 3, 2\}]$$

Such a result reflects a relationship between the answers for Q1 and Q2 from *Mike* (e.g., $2000 is spent last year for the day-care of the three children). We can also combine the questions and answers together by performing the following query:

$$\chi_{Questions,Answers}(h_3, h_3 \leadsto Mike)$$

which returns a result with the value:
[*Questions*: [$Q1$: *"Child care cost"*, $Q2$: *"Children ages"*],
Answers: [$Q1$: 2000, $Q2$: $\{6,3,2\}$]]
As a more powerful tool, we can build the following expression:

$$exp(x) = \chi_{Questions,Answers}(h_3, h_3 \leadsto x)$$

and then apply this expression to a set of employees in *MyGroup*:

$$\lambda_{exp(x)}(MyGroup)$$

The result is a set of AGG objects combining the two questions and their answers from each employee in *MyGroup*. As we can see, the KBO algebra has opened a brand new area for using query facilities in OODBs. The above examples demonstrated the usefulness and flexibility of our algebra.

8.3.2 Examples of Generating new BIO Invokers

The following examples illustrate the generation of some new BIO invokers. Assume that class BIO has three system-defined behaviors with values *"get if-part"*, *"get then-part"*, and *"get else-part"*, respectively. For example, if o is a bi-object with value $\langle id_1, id_2, id_3 \rangle$ then *"get if-part"* $\leadsto o \Rightarrow id_1$, *"get then-part"* $\leadsto o \Rightarrow id_2$, and *"get else-part"* $\leadsto o \Rightarrow id_3$. From Figure 8.20, we know that class CEO-DECISION is a subclass and a demandclass of class BIO. Now, let handle *RaiseDecisions* denote a set of CEO-DECISION objects. Suppose the CEO changes his/her mind about some of the decisions made in *RaiseDecisions* and wants to reverse the decisions that are based on whether a receiver is a manager. Then the following query can be issued by the CEO (for clarity we again separate the whole query into two expressions):

$$h_1 := \sigma_{s.if \times \text{"Is manager?"}}(RaiseDecisions)$$

$$h_2 := \pi_{if,then,else}(\text{"get if-part"}, \text{"get else-part"}, \text{"get then-part"}, h_1)$$

The result h_1 from the Select operation contains all the bi-objects from *RaiseDecisions* whose if-part is equal to *"Is manager?"*. The result h_2 from the Project operation

contains a new set of CEO-DECISION objects such that each bi-object in h_2 swaps the then-part and the else-part of a bi-object in h_1. For example, if the bi-object

$$\{\!\!\{ \text{“}\mathit{Is\ manager?}\text{”}, \text{“}\mathit{Increase\ salary}\text{”}, \text{“}\mathit{Pay\ overtime}\text{”} \}\!\!\}$$

is contained in set h_1, then there should be a new bi-object in h_2 with value:

$$\{\!\!\{ \text{“}\mathit{Is\ manager?}\text{”}, \text{“}\mathit{Pay\ overtime}\text{”}, \text{“}\mathit{Increase\ salary}\text{”} \}\!\!\}$$

In our next example, the CEO wishes to add one more condition to each decision in *RaiseDecisions*, such that for each decision its then-part should be carried out if the receiver also has a PhD degree. Assume that the class EMPLOYEE has behavior with value *"Has PhD?"* which returns a BOOL object. Then, an expression $exp(x)$ containing a Combine operator can be built as follows:

$$exp(x) = \chi_{if,then,else}(\text{“}\mathit{Has\ PhD?}\text{”}, x, \text{“}\mathit{get\ else\text{-}part}\text{”} \leadsto x)$$

which, if applied to a bi-object x, creates a new bi-object which has value *"Has PhD?"* as its if-part, and the whole bi-object x as its then-part, and the else-part of the bi-object x as its else-part. Now, the following query can be issued, which adds one more condition to each decision in *RaiseDecisions*:

$$h_3 := \lambda_{exp(x)}(RaiseDecisions)$$

The result contains a set of bi-objects with value $\{\!\!\{ id_1, \{\!\!\{ id_2, id_3, id_4 \}\!\!\}, id_4 \}\!\!\}$ such that id_1 has value *"Has PhD?"* and there is a bi-object in *RaiseDecisions* with value $\{\!\!\{ id_2, id_3, id_4 \}\!\!\}$. For instance, if the bi-object

$$\{\!\!\{ \text{“}\mathit{Is\ manager?}\text{”}, \text{“}\mathit{Increase\ salary}\text{”}, \text{“}\mathit{Pay\ overtime}\text{”} \}\!\!\}$$

is contained in *RaiseDecisions* then there should be a new bi-object contained in h_3 with value

$$\{\!\!\{ \text{“}\mathit{Has\ PhD?}\text{”}, \{\!\!\{ \text{“}\mathit{Is\ manager?}\text{”}, \text{“}\mathit{Increase\ salary}\text{”}, \text{“}\mathit{Pay\ overtime}\text{”} \}\!\!\}, \text{“}\mathit{Pay\ overtime}\text{”} \}\!\!\}$$

If this new bi-object is selected and sent to employee *Mike*, then *Mike* not only has to be a manager but also has to have a PhD degree in order to get his salary increased.

The following query is formulated such that the end-user hopes to give a 15% salary increase to those employees in *MyGroup* based on a decision from h_3 which does not require that an employee be a manager.

$$\frac{\rho(\delta_{\neg(x.then.if \times \text{“}\mathit{Is\ manager?}\text{”}) \wedge (x.then.then \times \text{“}\mathit{Increase\ salary}\text{”} \vee x.else \times \text{“}\mathit{Increase\ salary}\text{”})}(h_3), MyGroup)}{\text{Percent}: 15}$$

The query first picks up a bi-object from h_3 based on the desired predicate, and then applies the bi-object to a set of employees in *MyGroup*.

8.3.3 Examples of Generating new SET Invokers

The following examples illustrate the generation of new SET invokers. Suppose the company is committed to provide its employees with a great variety of benefits packages. All these packages are stored in a set object denoted by *BenefitsPackages* with the following value:

BenefitsPackages =
{ {*LifeInsPlan*1, *Dental*50, *StockPurchase*, *PensionPlan*3, *VacationPlan*},
{*EaringsProtection*, *VacationPlan*, *GroupLifeIns*, *Dental*80},
{*CreditUnion*, *PensionPlan*2, *VacationPlan*, *FitnessSubsidy*, *HomeInsPlan*},
{*TuitionRefund*, *LifeInsPlan*4, *GroupRRSP*, *DayCarePlan*6, *Dental*30},
{*SalesShare*, *TravelAccIns*, *OnSiteMedical*, *LifeInsPlan*1},
⋮
}

where each sub-set contains some object handles denoting the identity of a behavior of class EMPLOYEE (or an object with the same value). For example, *LifeInsPlan* denotes an instance of class LI-PLAN, and *Dental*50 may be just a TEXT object with value *"50% dental plan"*. Suppose that individually all these objects can invoke a behavior of EMPLOYEE with signature () ⟶ SELF (note that if a behavior takes some labeled arguments, these arguments can always be prompted for from the end-user when needed). In other words, each object denoted by those handles can invoke a hardcoded knowhow that does some complex calculation based on an employee's current internal state and updates the employee's state to a new state (such as changing the values of some attributes). For example, the behavior *Dental*50, when sent to an employee object, will calculate a specific dental plan based on the policy of this plan and the employee's current data (such as the ages of the employee's dependent children and the like), and the generated dental plan (i.e., the one tailored to this particular individual) is then saved as the value of attribute DentalPlan in the employee object.

In the set *BenefitsPackages*, each sub-set in *BenefitsPackages* represents a package deal which is carefully designed to provide an overall balanced program to meet the various needs of company employees. Giving a specific package to an employee is realized by choosing a set object from *BenefitsPackages* and sending this set object (as the invoker) to the object representing that employee.

In the following example, we wish to create a new set of benefits packages, which

includes all the packages from *BenefitsPackages* except that the object denoted by *GroupLifeIns* will be added into the packages that includes *GroupRRSP*, to replace (if any) other old life insurance plans. Again, for clarity we separate this query into several un-nested expressions. First, we select all packages that include *GroupRRSP* into the set h_1:

$$h_1 := \sigma_{GroupRRSP \in_{\mathbf{s}} x}(BenefitsPackages)$$

Then, we collect the packages that do not include *GroupRRSP* into the set h_2:

$$h_2 := -(BenefitsPackages, h_1)$$

Assume that class LI-PLAN has a behavior with value *"Is a life insurance plan?"* which always returns **True** when sent to a LI-PLAN object (such as *LifeInsPlan*1 and *LifeInsPlan*4). We now build an algebra expression as follows:

$$exp(x) = \cup(\{GroupLifeIns\}, \sigma_{\neg((\text{"Is a life insurance plan"} \leadsto y) \times \mathbf{True})}(x))$$

which produces the union of the set {*GroupLifeIns*} and a set containing all the members from x except those which are life insurance plans. We apply this expression to the set h_1:

$$h_3 := \lambda_{exp(x)}(h_1)$$

which gets rid of any old life insurance plan from each package in h_1 and adds the life insurance plan *GroupLifeIns*. Now we can generate the new set of benefits packages by the following query:

$$NewBenefitsPackages := \cup(h_2, h_3)$$

The result is the one we wish to generate.

The next example demonstrates the usage of the packages in *NewBenefitsPackages*. Let handle *Package*25 (say, an often used package) denote one of the objects contained in *NewBenefitsPackages*. Suppose we wish to give salesperson *Mike* a benefits package that includes the benefits offered both by *Package*25 and by a package that has the travel accident insurance plan *TravelAccIns*. This is accomplished by the following message-sending:

$$\cap(Package25, \delta_{TravelAccIns \in_{\mathbf{s}} x}(NewBenefitsPackages)) \leadsto Mike$$

where the Pickup operation selects a package from *NewBenefitsPackages* that has the travel accident insurance plan *TravelAccIns*, and the Intersect operation collects the benefits that must be offered both by *Package*25 and by the package selected by

the Pickup operation. If we wish give the same package to everyone in $MyGroup$, we can do the following:

$$\cap(Package25, \delta_{TravelAccIns \in_{\#} x}(NewBenefitsPackages)) \overset{1}{\rightsquigarrow} MyGroup$$

The example has demonstrated the usage of the Subtract, Union, Intersect operators for generating new SET invokers.

8.3.4 Examples of Generating new SEQ Invokers

In the following, we give some examples to illustrate the generation of new SEQ invokers. Assume that the following set has been populated by the end-users who frequently retrieve information from FAMILY objects.

$$GetInfo = \{< \text{"get Father"}, \text{"get Address"} >, < \text{"get Mother"}, \text{"get Address"} >, ..., ...\}$$

Also assume that class SEQ has a system-defined behavior with value *"Last member"* which returns (if any) the last member in a sequence. Suppose in class FAMILY the attribute Address now has its type changed from TEXT to ADDRESS, which has the following representation

$$\dot{\Xi}(\text{ADDRESS}) = [\text{OfficeAddress: TEXT, HomeAddress: TEXT}]$$

and has two accessors with values *"get OfficeAddress"* and *"get HomeAddress"*. Therefore, we wish to update all sequences in $GetInfo$ that have a value $< ..., \text{"get Address"} >$ to $< ..., \text{"get Address"}, \text{"get HomeAddress"} >$. We first select all sequences from $GetInfo$ that have *"get Address"* as the last member:

$$h_1 := \sigma_{(\text{"Last member"} \rightsquigarrow x) \asymp \text{"get Adress"}}(GetInfo)$$

We then build an algebra expression:

$$exp(x) = \cup(x, < \text{"get HomeAddress"} >)$$

which concatenates sequence x and $< \text{"get HomeAddress"} >$. Then we apply this expression to each sequence contained in h_1:

$$h_2 := \lambda_{exp(x)}(h_1)$$

The result is what we desired. That is, the result contains a set of new sequence invokers. Note that the query can be formulated in one expression:

$$\lambda_{\cup(x, < \text{"get HomeAddress"} >)}(\sigma_{(\text{"Last member"} \rightsquigarrow y) \asymp \text{"get Address"}}(GetInfo))$$

This example also demonstrates how complex behaviors can be adjusted to schema changes by using the object algebra.

For the next example, let set object *SetupProject* contain a set of strategies that have been used (and accumulated over the years) for different scientists to set up a project. When a scientist wishes to start a project, he/she can choose a desired strategy from *SetupProject* (or design a new one if none in *SetupProject* is desirable), and send it to the object that represents this scientist. For example, if Mike is a scientist working with the company, then there is an instance of class SCIENTIST that represents Mike. Let this instance be denoted by handle *Mike*. Note that a SCIENTIST object may contain information about how many programming employees work under this scientist, whether they are available at this time, what projects they are currently working on, whether they are on leave of absence, what programming experience they have, what funding situations are now, who is this scientist's boss, etc. This information may be utilized (for calculation) and updated by those behaviors of SCIENTIST contained in sequences in *SetupProject*. Assume that all these behaviors have the signature () $\longrightarrow$ SELF.

SetupProject =
{< *"Meeting with CEO", "Budget estimate", "Get budget approved"* >,
< *"Prepare proposal", "Budget estimate", "Verify budget", "Get budget approved"* >
,
< *"Marketing study", "Group scheduling", "Manpower planing", "Get budget approved"* >
,
⋮
}

Note that some of the behaviors contained in these sequences may, when invoked, interact with the end-user during its execution. For example, behavior *"Meeting with CEO"* may start a LAN conferencing facility on the screen so that the end-user (a scientist) can talk to the CEO about the project, and behavior *"Group scheduling"* may invoke a group-scheduler on a LAN that allows the scientist to schedule the project progress cooperatively with his/her programmers.

Query: Based on the existing project-setup strategies, create a new set of strategies that include marketing study and require a meeting with the CEO at the end. We first select all strategies from *SetupProject* that contain an invoker with value

"Marketing study":

$$h_1 := \sigma_{\text{"Marketing study"} \in_\times x}(SetupProject)$$

We then build an algebra expression $exp(x)$ as follows:

$$exp(x) = \cup(-_\times(x, <\text{"Meeting with CEO"}>), <\text{"Meeting with CEO"}>)$$

where the Subtract operation drops any object with value *"Meeting with CEO"* (if any) from a sequence x and the Union operation appends an object with value *"Meeting with CEO"* to the end of the sequence produced by the Subtract operation. We then apply this expression to the set h_1:

$$h_2 := \lambda_{exp(x)}(h_1)$$

The result is what we desired. The following two message-sendings let, respectively, *Mike* and each scientist in *MyGroup* to set up a project by using a strategy from h_2 that includes an invoker with value *"Group scheduling"*:

$$\delta_{\text{"Group scheduling"} \in_\times x}(h_2) \rightsquigarrow Mike$$

$$\delta_{\text{"Group scheduling"} \in_\times x}(h_2) \overset{1}{\rightsquigarrow} MyGroup$$

For the following example, let *NewStrategy* denote a sequence representing a new project-setup strategy created by some end-user.

Query: Let *Mike* follow only those steps in *NewStrategy* that also appear in one or more strategies in *SetupProject*.

$$\cap_\times(NewStrategy, \varpi(SetupProject)) \rightsquigarrow Mike$$

Here, the Collapse operation $\varpi(SetupProject)$ produces a set object containing all behaviors used in the sequences in set *SetupProject*. (Recall that our Collapse operator can "collapse" a set of sequences!) The Intersect operator $\cap_\times$ drops those behaviors from *NewStrategy* that are also equal to an object in the set produced by the Collapse operation. Note that the ordering of sequence *NewStrategy* is still preserved.

8.4 Queries in an Interactive Application

In this section, we demonstrate the queries in a database application for interactive animation designs. We believe that such applications can really take advantage of the behavior-building features provided by the KBO algebra. In such environments, it is often required that any generated complex behaviors have to be executed immediately (i.e., sent to the receiver). In other words, we can use the KBO algebra to do some "real-time programming", but in an associative manner. Note that in such environments there is no time for us to pause to check what is exactly being generated as the invoker. Such an environment would be experimental in nature — you try many different things on the screen which helps you to visually decide which animation you like the best.

In our example application, there is a class called SMURF which has three instances (denoted by $Smurf_1, Smurf_2, Smurf_3$) and seven behaviors shown in the following interface table (i.e., $\dot{\Sigma}$(SMURF)):

oid	class	value	vigor		
b_1	TEXT	*"Go"*	SMURF	() ⟶ SELF	kh_1
b_2	TEXT	*"Bounce"*	SMURF	() ⟶ SELF	kh_2
b_3	TEXT	*"Waddle"*	SMURF	() ⟶ SELF	kh_3
b_4	TEXT	*"Hop"*	SMURF	() ⟶ SELF	kh_4
b_5	TEXT	*"Somersault"*	SMURF	() ⟶ SELF	kh_5
b_6	TEXT	*"Crawl"*	SMURF	() ⟶ SELF	kh_6
b_7	TEXT	*"Sneak"*	SMURF	() ⟶ SELF	kh_7
b_8	TEXT	*"Tired out?"*	SMURF	() ⟶ BOOL	kh_8

Each of these behaviors has a knowhow that knows how to produce a "smurf move" on the screen (or recorded in the database for future display). Suppose, having produced many episodes, the end-users (cartoon designers) have gathered a large number of walk-manners for Smurfs. They are stored in a set object denoted by *WalkManners*, as shown in the following. Let *WalkManners* be an instance of class WALK-MANNERS where

$$\dot{\Xi}(\text{WALK-MANNERS}) = \{\text{TEXT-SEQUENCE}\}$$
$$\dot{\Xi}(\text{TEXT-SEQUENCE}) = <\text{TEXT}>$$

And the set object *WalkManners* has the following value:
$WalkManners =$

{ < *"Go"*, *"Crawl"*, *"Hop"*, *"Somersault"*, *"Hop"*, *"Hop"* >,
< *"Go"*, *"Hop"*, *"Waddle"*, *"Waddle"*, *"Sneak"*, *"Bounce"*, *"Crawl"*, *"Bounce"* >,
< *"Go"*, *"Somersault"*, *"Sneak"*, *"Crawl"*, *"Hop"*, *"Somersault"*, *"Hop"* >,
⋮
⋮
< *"Go"*, *"Waddle"*, *"Hop"*, *"Waddle"*, *"Hop"*, *"Waddle"*, *"Sneak"*, *"Crawl"* > }

Now let us consider the following queries in an interactive design session:

1. **Query 1:** Show me a walk-manner of $Smurf_1$ that does a somersault.

   $$\delta_{\text{"Somersault"} \in_x x}(WalkManners) \rightsquigarrow Smurf_1$$

   where the Pickup operation picks up a single member x (if any) from set $WalkManners$ which contains a somersault move.

2. **Query 2:** Show me all the walk-manners in which $Smurf_2$ waddles and crawls.

   $$\sigma_{\text{"Waddle"} \in_x x \wedge \text{"Crawl"} \in_x x}(WalkManners) \rightsquigarrow Smurf_2$$

   where the sequence objects contained in the set produced by the Select operation will be sent to the receiver $Smurf_2$ in an arbitrary order (i.e., the case when the invoker is a set object).

3. **Query 3:** Create a new walk-manner that makes $Smurf_3$ hop and bounce.

   $$\cup(\delta_{\text{"Hop"} \in_x x}(WalkManners), \delta_{\text{"Bounce"} \in_x x}(WalkManners)) \rightsquigarrow Smurf_3$$

   where the Union operation concatenates two sequence objects picked up from set $WalkManners$.

4. **Query 4:** Show me how differently these Smurfs do their somersaults in a walk-manner.

   $$\delta_{\text{"Somersault"} \in_x x}(WalkManners) \overset{1}{\rightsquigarrow} Smurfs$$

   where $Smurfs = \{Smurf_1, Smurf_2, Smurf_3\}$. Notice that shallow-send is used here; alternatively, deep-send may also be used here to do the same thing.

5. For this query, assume class SMURF is an AGG class and has an attribute Color.

Query 5: Show me how a blue Smurf waddles in all walk-manners containing a waddle move.

$$\sigma_{\text{"Waddle"} \in_X x}(WalkManners) \rightsquigarrow \delta_{x.Color = \text{"blue"}}(Smurfs)$$

Notice that the invoker above is a set object containing selected sequence objects.

6. For the following query, we assume that class TEXT-SEQUENCE has two behaviors with values *"Odd part"* and *"Length"*. The former yields a new sequence consisting of all the elements at the odd positions in a sequence, and the latter yields the number of elements in a sequence. For example, if object w has value

   < *"Go"*, *"Somersault"*, *"Waddle"*, *"Sneak"*, *"Bounce"*, *"Crawl"*, *"Somersault"* >

   then *"Odd part"* $\rightsquigarrow w$ will return a new sequence object with value < *"Go"*, *"Waddle"*, *"Bounce"*, *"Somersault"* >, and *"Length"* $\rightsquigarrow w$ will return the integer 7.
   Query 6: Show me how $Smurf_1$ walks in the odd parts of all the walk-manners whose odd part has more than 3 moves.

$$\sigma_{(\text{"Length"} \rightsquigarrow x) > 3}(\rho(\text{"Odd part"}, WalkManners)) \rightsquigarrow Smurf_1$$

   In this query, the Apply operation is evaluated first, which produces a set object (say w_1) consisting of the new walk-manners — the odd parts of all the walk-manners in $WalkManners$. Then the Select operation $\sigma_{(\text{"Length"} \rightsquigarrow x) > 3}(w_1)$ is evaluated, which produces another set object (say w_2) consisting of the new walk-manners in w_1 that contain more than three moves. Here, the variable x in the selection predicate (*"Length"* $\rightsquigarrow x) > 3$ ranges over set w_1. Finally, $w_2 \rightsquigarrow Smurf_1$ is evaluated, which causes $Smurf_1$ to walk in each new walk-manner contained in w_2, in an arbitrary order.

With the notion of object identity, objects can share sub-objects. This makes it possible to have *cyclically referencing objects*. In our model, this basic feature turns out to be a natural mechanism to represent some simple iterative structures in the form of complex invokers. As an example, let h_1 be one of those objects in $WalkManners$:

$h_1 :=$ < *"Go"*, *"Crawl"*, *"Hop"*, *"Somersault"*, *"Hop"*, *"Hop"* >

Let h_2 denote the following object:

$h_2 :=$ < h_1, *"Tired out?"* >

And let *KeepGoing* denote the following bi-object:

$KeepGoing := \{ h_2, \textbf{same}, KeepGoing \}$

Notice that bi-object *KeepGoing* is constructed from existing objects h_2 (and thus h_1) and itself. Now, if we use *KeepGoing* as the invoker in the message-sending

$$KeepGoing \rightsquigarrow Smurf_1$$

then we will see $Smurf_1$ moving repeatedly in the manner described in h_1 until $Smurf_1$ is tired out. (Assume that each move made by a smurf will tire him a bit and at a certain threshold the smurf will respond to the message "Tired out?" with **True**.) More precisely, this is what happens when $KeepGoing \rightsquigarrow Smurf_1$ takes place: (1) $h_2 \rightsquigarrow Smurf_1$ occurs first, which in turn will be evaluated as *"Tired out?"* $\rightsquigarrow (h_1 \rightsquigarrow Smurf_1)$; (2) $h_1 \rightsquigarrow Smurf_1$ will cause $Smurf_1$ to do a sequence of "smurf moves", and then, after finishing these moves, $Smurf_1$ will receive the message *"Tired out?"*; at this point, (3) if $Smurf_1$ returns **True**, that is, $Smurf_1$ is in deed "very tired", then $\textbf{same} \rightsquigarrow Smurf_1$ occurs, which of course returns $Smurf_1$ itself; however, (4) if $Smurf_1$ returns **False**, that is, $Smurf_1$ is not really tired yet, then the first message-sending $KeepGoing \rightsquigarrow Smurf_1$ occurs again and the above process is then repeated, until $Smurf_1$ is finally tired out.

Now, suppose we have many such persistent bi-objects (including *KeepGoing*) stored in a set object denoted by *NewWalkManners*. Then, the following query will "pick up" a recursive walk manner from *NewWalkManners* which may stop when the moving target (a smurf) is "tired out":

$$\delta_{x \equiv x.else \wedge \text{"Tired out?"} \in_{\times} x.if}(NewWalkManners) \rightsquigarrow Smurf_1$$

where variable x in the Pickup predicate ranges over set. Obviously the bi-object *KeepGoing* satisfies the Pickup predicate because its else-part is identical to itself and its if-part has a member object with value "Tired out?". Therefore, it is possible for object *KeepGoing* to be picked up from *NewWalkManners* and sent to $Smurf_1$. This example also illustrates the advantage of treating a complex behavior with a branching nature as a regular data object.

The above examples have illustrated some advantages of the KBO model in which complex objects can be sent to receivers, in an associative manner, to cause a composite behavior. Here, an important point must be stressed. One might say that these examples are trivial to do in some persistent programming languages (like persistent Smalltalk or CLOS). However, *we are not trying to extend the power of any programming languages.* Instead, our ultimate goal here is to design an "end-user

oriented" query language (something like SQL for relational databases) that provides a simple querying facility for both data and behaviors in OODBs. In other words, our intention is to extend the power of current OODB query models so that end-users can also manipulate (and combine/invoke) complex behaviors in an ad hoc and associative manner. The need for such a way of extracting information from OODBs *without having to write a program* has recently been recognized in many papers, such as [B*89, C*89, F*87, Osb88, SZ89, RS87]. To the best of our knowledge, no existing query model for OODBs supports manipulation (and associative composition/invocation) of data-like complex behaviors. Therefore, we believe that our work is, although a limited one, a promising first step towards that end.

CHAPTER 9

PROPERTIES OF THE KBO ALGEBRA

9.1 Introduction

In Chapter 8 we presented an object algebra, namely the KBO Algebra, which is capable of manipulating KBO objects. Since KBO objects also have dynamic semantics when used as invokers, a significant result of having such an object algebra is that end-users are now provided with the associative power to select, construct, invoke and maintain complex arbitrary behaviors. In this chapter, we discuss some special properties of the KBO algebra.

The property of *object-type completeness* for an object algebra is desired mainly because of the immutability of object-identities [Atk89] in OODBs — during the life time of an object, its object-identity remains the same while any other intrinsic parts of the object (like value, class, etc.) are subject to change and evolution.

The property of *closure* guarantees that each operation on an object (objects) produces a new object which has exactly the same status as the original one (ones). All the operations of the object algebra are potentially applicable to it. By having this property, the result of a query can be used as the input for other queries or can be stored as a user's view. Unfortunately, most existing query models for OODBs do not preserve closure [ASH89].

The property of *relational consistency* enables our object algebra to simulate the power of relational algebra. In other words, we show that the KBO algebra is at least as expressive as the relational algebra.

We also provide some provably correct *algebraic transformations* between KBO algebra expressions. They correspond to rewrite rules which can be used by a query optimizer. It should be pointed out that development of a viable query optimizer (such as the work by Straube and Ozsu [SO90, Str91]) for the KBO algebra is a very interesting topic but beyond the scope of this thesis. In other words, our focus here is to study some general algebraic rewriting properties of the KBO algebra.

At the end of this chapter, we give an evaluation of the KBO algebra using the evaluation framework proposed in [YO91a]. The evaluation shows that the most distinguished features of the KBO algebra are the support for behavior constructors and behaviors-as-objects. Unlike conventional objects in existing OODBs, complex be-

haviors (or complex invokers) in KBO databases are selectable, constructable, maintainable (and even invocable) through a database query facility, namely the KBO object algebra.

9.2 Object-Type Completeness

Definition 75: *Object-Type Completeness.* An object algebra is considered to be *object-type complete* if all of its operators can be applied to operands (objects) of any classes supported by the underlying data model. ■

Like the relational algebra whose operators can be applied to any relations, an "object" algebra should be applicable to all objects describable in the underlying data model. If an object algebra is object-type complete, then its users will not find that some object classes are apparently arbitrarily precluded from appearing in the algebra operations. For example, given that Select or Project can operate on SET objects, the user can confidently expect to be able to use them to operate on SEQ, BIO, AGG or even CAPSULE objects which are also describable in the KBO data model. The concept of having one object algebra defined on multiple types of objects is first introduced in Osborn's algebra [Osb88, Osb89a, Osb89b], where all the algebra operators can be applied to objects of all three types (i.e., Atomic, Aggregate and Set) supported by the data model. Note that although some of our algebra operators (mainly, Union, Intersect and Subtract) return ***null*** when applied to CAPSULE, AGG or BIO objects, they are not reporting a failure — the semantics of ***null*** in our message-sendings is well-defined.

Theorem 19: *The KBO object algebra is object-type complete.*

Proof: The proof is straightforward from the definitions of the KBO algebra operators. In other words, the theorem holds because all objects describable in the KBO model are instances of one of the five kernel classes: CAPSULE, AGG, BIO, SET, and SEQ, and all operators in the KBO algebra are formally defined on these five types of operands. ■

It is desirable to make an object algebra object-type complete in identity-based OODBs, not only because users do not have to learn a different query facility for a different kind of objects (as they are forced to in the many-sorted algebra proposed in [VD91] for multi-sets, arrays, tuples and object identifiers), but because the operands of our algebra operations are immutable object-identities which may have mutable values and types. For example, if we have an algebra expression (the Select operation)

$$\sigma_{p(x)}(MyReport)$$

where *MyReport* is an object-identity with a set value (e.g., a set of CHAPTER objects), then we do not have to change this query even if the operand *MyReport* is now evolved to have a sequence value (e.g., a sequence of chapter objects, which represents a finished technical report). Note here, that a general assumption is made for objects in our model: while an object' identity cannot be changed during its life time, any thing else within the object can be changed (e.g., class, value, knowhow, etc.).

9.3 Closure Property

Theorem 20: *The set of operations defined in Chapter 8 forms a closed algebra on the object universe $\mathcal{O}$. Thus, the result of each operation is again describable as an object in $\mathcal{O}$.*

Proof: The proof is straightforward from the definitions of the KBO algebra operators. In other words, the theorem holds because each of our operator is defined to take any kinds of objects in $\mathcal{O}$ and produce an object in $\mathcal{O}$. ■

The most important outcome of this theorem is that *the KBO data model is closed under its manipulation operations.* Thus, we are able to concatenate several KBO algebra operators to build higher-level and more complex operations. As a result, queries of arbitrary complexity can be built up in one algebra expression. The end-user can even give each intermediate result a name (handle) to remember it for future queries.

9.4 Relational Consistency

A major criticism of the existing object-oriented database systems is that they may take us back to the days of CODASYL database systems in which data is accessed by using pointers to navigate through the database [AG89]. We believe that it is important for an object algebra to reflect the spirit of relational algebra. For an object algebra to be at least as powerful as relational algebra, it should provide, as a minimum, an object-oriented counterpart for each of the five operations that serve to define relational algebra: *Selection, Projection, Cartesian product, Union,* and *Difference* [Ull88]. We say that an object algebra is *relationally consistent* if these five relational algebra operators are expressible in that object algebra on the objects representing normalized or nested relations. It should be noted that in the KBO model, normalized relations can be represented as a set of AGG objects whose

attributes are CAPSULE objects, and nested relations can be represented as a set of AGG objects whose attributes can be another AGG object.

Theorem 21: *The KBO algebra is relationally consistent.*

Proof: For any normalized (or nested) relation r we can define a direct subclass c_r of AGG such that (1) the tuples contained in the relation r are the instances of c_r, and (2) if class c_r has n attributes in its representation

$$\dot{\Xi}(c_r) = [A_1\colon c_1, \ldots, A_n\colon c_n]$$

then it has n behaviors in its interface

$$\dot{\Sigma}(c_r) = \{b_1, ..., b_n\}$$

where $value(b_i)$ = "*get* A_i" and $sig(b_i) = () \longrightarrow c_i$, for $1 \leq i \leq n$. In other words, $b_1, \ldots, b_n$ are explicit attribute accessors. That is, when an invoker with TEXT value "*get* A_i" is sent to an instance of c_r, behavior b_i will be invoked to return the object at the attribute A_i within the receiver. Now, let r be a direct instance of SET.

1. The Selection operation in the relational algebra, i.e., $\sigma_{q(A'_1,...,A'_k)}(r)$ where $q(A'_1, ..., A'_k)$ is the selection criterion involving attributes $A'_1, ..., A'_k$, can be expressed by our Select operator on one operand:

$$\sigma_{p(x)}(r)$$

   where x ranges over set r and the predicate $p(x) = q(x.A'_1, ..., x.A'_k)$. Here $\{A'_1, ..., A'_k\} \subseteq \{A_1, ..., A_n\}$. Note that expression $x.A'_i$ can also be replaced by message-sending ("*get* A'_i" $\leadsto x$).

2. The Projection operation in the relational algebra, i.e., $\pi_{A'_1,...,A'_k}(r)$, can be expressed by our Project operator as follow:

$$\pi(\text{"}get\ A'_1\text{"}, \ldots, \text{"}get\ A'_k\text{"}, r)$$

   where $1 \leq k \leq n$ and each A'_i is an attribute label in $\{A_1, ..., A_n\}$.

3. Let r_1 and r_2 be two relations created in the similar way we create r. Then, the Cartesian product operation in the relational algebra, i.e., $r_1 \times r_2$, can be expressed by the combination of our Combine, Collapse and Expression Apply operators as follow:

$$\lambda_{\varpi(x)}(\chi_{L_1,L_2}(r_1, r_2))$$

where L_1, L_2 are two arbitrary attribute labels. The inner expression $\chi_{L_1,L_2}(r_1, r_2)$ produces a SET object (say s) with the following set value

$$\{[L_1: x, L_2: y] |\ x \in r_1 \wedge y \in r_2\}$$

and the expression $\lambda_{\varpi(x)}(s)$ produces a SET object with the following set value

$$\{[A_1: x_1, ..., A_n: x_n, B_1: y_1, ..., B_m: y_m] \mid \\ \exists x \in s: x.L_1 = [A_1: x_1, ..., A_n: x_n] \wedge x.L_2 = [B_1: y_1, ..., B_m: y_m]\}$$

4. The Union operation in the relational algebra, i.e., $r_1 \cup r_2$, can be expressed by our extended Union operator as follow:

$$\cup_{\times_1}(r_1, r_2)$$

where any two members in the result are not identical.

5. The Difference operation in the relational algebra, i.e., $r_1 - r_2$, can be expressed by our extended Subtract operator as follow:

$$-_{\times_1}(r_1, r_2)$$

where any two members in the result are not identical.

Therefore, the KBO algebra is relationally consistent. ■

9.5 Formal Algebraic Transformations

In this section, we study a set of general algebraic transformations and their applicability conditions for rewriting the KBO algebra expressions. Our formal analysis will lead to the equivalence theorems that can be expressed as rewrite rules, and input to an optimizer generator (which is an interesting future research topic) based on such rules. Our transformation rules will be written as

$$exp_1\ \theta\ exp_2$$

where θ is $\equiv$, $\asymp$ or $\asymp_n$ $(n > 0)$, which specifies that an algebra expression exp_1 is identical, equal or n-equal, respectively, to another algebra expression exp_2. Like most rules presented in [Osb88], most of our rules only guarantee 1-equality (called "shallow identity" in [Osb88]). It has been pointed out by Osborn [Osb88] that, in set/sequence

oriented environments, users generally do not care about the object-identity of the set/sequence containing the objects they have obtained from the database, but would be content to know that the same objects are in the set/sequence (and the same ordering is preserved in the sequence). In some cases where the operands are not SET or SEQ objects, we can derive identity instead of 1-equality.

Since our data model supports not only set objects but also sequence objects, we often need to prove that two sequence objects are 1-equal. To do so, we usually first prove that they contain identical members and then prove that they have the same ordering. To assist such proofs, we shall introduce the notion of order preservation.

Definition 76: *Order Preservation.* Let x, y be two sequence objects. We say that *x preserves the ordering* of y if any two members o_1, o_2 in x are positioned such that if o_1 precedes o_2 in y then o_1 also precedes o_2 in x. ■

For example, sequence $< o_1, o_3, o_5 >$ preserves the ordering of $< o_1, o_2, o_3, o_4, o_5 >$. In other words, if we drop some members from a sequence without changing the relative order of what is left, then the new sequence preserves the ordering of the original sequence.

In the following we present our equivalence theorems in a framework similar to that used in [Osb88, Osb89a]. That is, we divide them into four categories: *idempotence properties, commutativity properties, associativity properties*, and *distributivity properties*.

9.5.1 Idempotence Properties

Theorem 22: *Idempotence of Select.* Let $1 \leq k < n$.

1. If r ⓘ SET or SEQ, then

$$\sigma_{p_1(x_1,\ldots,x_k,x) \wedge p_2(x_{k+1},\ldots,x_n,x)}(s_1, \ldots, s_n, r)$$
$$\asymp_1$$
$$\sigma_{p_1(x_1,\ldots,x_k,x)}(s_1, \ldots, s_k, \sigma_{p_2(x_{k+1},\ldots,x_n,x)}(s_{k+1}, \ldots, s_n, r))$$

2. If r ⓘ CAPSULE, AGG or BIO, then

$$\sigma_{p_1(x_1,\ldots,x_k,x) \wedge p_2(x_{k+1},\ldots,x_n,x)}(s_1, \ldots, s_n, r)$$
$$\equiv$$
$$\sigma_{p_1(x_1,\ldots,x_k,x)}(s_1, \ldots, s_k, \sigma_{p_2(x_{k+1},\ldots,x_n,x)}(s_{k+1}, \ldots, s_n, r))$$

Proof:

1. Let r ⓘ SET. Without loss of generality, let $s_1, ..., s_n$ all be SET or SEQ objects (note that if s_i is not a SET or SEQ object, its use in the selection predicate is equivalent to single member set object $\{s_i\}$ which is ranged over by variable x_i). The LHS expression of the first equation produces a new object-identity with a set value which can be expressed as:

$$\{o \in r \mid \exists o_1 \in s_1, ..., \exists o_n \in s_n : (p_1(o_1, ..., o_k, o) \wedge p_2(o_{k+1}, ..., o_n, o))\}$$

Since predicates p_1 and p_2 do not share object variables, the existential quantifiers can be distributed across the conjunction as follows:
$\{o \in r \mid \exists o_1 \in s_1, ..., \exists o_k \in s_k : p_1(o_1, ..., o_k, o)$
$\wedge \exists o_{k+1} \in s_{k+1}, ..., \exists o_n \in s_n : p_2(o_{k+1}, ..., o_n, o)\}$
This expression can be rewritten as

$$\{o \in r' \mid \exists o_1 \in s_1, ..., \exists o_k \in s_k : p_1(o_1, ..., o_k, o)\}$$

where r' is the new range for o:

$$r' = \{x \in r \mid \exists o_{k+1} \in s_{k+1}, ..., \exists o_n \in s_n : p_2(o_{k+1}, ..., o_n, x)\}$$

By definition, the above new set value is the one produced by the RHS expression of the first equation in the theorem. That is, the LHS expression and the RHS expression contain the same members. Since Select operations always generate a new object-identity when the operand r (call it the "target operand") is a SET (or a SEQ) object, it follows immediately that

$$\sigma_{p_1(x_1,...,x_k,x) \wedge p_2(x_{k+1},...,x_n,x)}(s_1, ..., s_n, r)$$
$$\asymp_1$$
$$\sigma_{p_1(x_1,...,x_k,x)}(s_1, ..., s_k, \sigma_{p_2(x_{k+1},...,x_n,x)}(s_{k+1}, ..., s_n, r))$$

In a similar fashion, we can prove that the LHS expression and the RHS expression contain the same number of members when the target operand r is a SEQ object. Note that if r contains two identical members which satisfy the selection predicate, then both of them will be left in the result of the Select operation due to the difference of their positions in the sequence. Now, we only have to examine whether the two expressions have the same ordering. By definition, Select operations only drop unqualified members from a sequence target operand and do not change the ordering of the remaining members. Therefore, the two expressions have the same ordering — both of them preserve the ordering of the target operand r.

2. Let r ⓘ CAPSULE, AGG or BIO. The LHS expression of the second equation produces the target operand r itself if the quantified predicate

$$\exists o_1 \in s_1, ..., \exists o_n \in s_n : (p_1(o_1, ..., o_k, r) \wedge p_2(o_{k+1}, ..., o_n, r))$$

is satisfied. The RHS expression of the equation produces the target operand r if the quantified predicate

$$\exists o_1 \in s_1, ..., \exists o_k \in s_k : p_1(o_1, ..., o_k, r) \wedge \exists o_{k+1} \in s_{k+1}, ..., \exists o_n \in s_n : p_2(o_{k+1}, ..., o_n, r)$$

is satisfied. Since predicates p_1 and p_2 do not share object variables (except constant r), the above two quantified predicates are equivalent in meaning. Hence, the expressions on the two sides of the second equation produce identical result, which is either r or ***null***.

Example 64: For query "find all employees in *MyGroup* that are younger than an employee in *XGroup* but older than an employee in *YGroup*", we can formulate the following object algebra expression:

$$\sigma_{x.Age<x_1.Age \ \wedge \ x.Age>x_2.Age}(XGroup, YGroup, MyGroup)$$

where x_1 ranges over *XGroug*, x_2 over *YGroup*, and x over *MyGroup*. According to Theorem 21, we can transform this expression into the following one:

$$\sigma_{x.Age<x_1.Age}(XGroup, \sigma_{x.Age>x_2.Age}(YGroup, MyGroup))$$

which produces a result that is 1-equal to the one produced by the original expression.

The next theorem allows us to perform a Select operation on one of the non-target operands before we perform the Select operation on the target operand.

Theorem 23: *Idempotence of Select.* Let $1 \leq k \leq n$.

1. If r ⓘ SET or SEQ, then

$$\sigma_{p_1(x_k) \wedge p_2(x_1,...,x_k,...,x_n,x)}(s_1, ..., s_n, r)$$
$$\asymp_1$$
$$\sigma_{p_2(x_1,...,x_n,x)}(s_1, ..., \sigma_{p_1(y)}(s_k), ..., s_n, r)$$

2. If r ⓘ CAPSULE, AGG or BIO, then

$$\sigma_{p_1(x_k)\wedge p_2(x_1,\ldots,x_k,\ldots,x_n,x)}(s_1,\ldots,s_n,r)$$
$$\equiv$$
$$\sigma_{p_2(x_1,\ldots,x_n,x)}(s_1,\ldots,\sigma_{p_1(y)}(s_k),\ldots,s_n,r)$$

Proof:

1. Let r ⓘ SET. Without loss of generality, let $s_1,\ldots,s_n$ all be SET or SEQ objects. Then the LHS expression of the first equation produces a new object-identity with a set value which can be expressed as:

$$\{o\in r \mid \exists o_1\in s_1,\ldots,\exists o_k\in s_k,\ldots,\exists o_n\in s_n: (p_1(o_k)\wedge p_2(o_1,\ldots,o_k,\ldots,o_n,o))\}$$

This expression can be rewritten as

$$\{o\in r \mid \exists o_k\in s_k: (p_1(o_k)\wedge \exists o_1\in s_1,\ldots,\exists o_{k-1}\in s_{k-1},\exists o_{k+1}\in s_{k+1},\ldots,\exists o_n\in s_n: p_2(o_1,\ldots,o_k,\ldots,o_n,o))\}$$

which is equivalent in meaning to

$$\{o\in r \mid \exists o_1\in s_1,\ldots,\exists o_k\in s'_k,\ldots,\exists o_n\in s_n: p_2(o_1,\ldots,o_k,\ldots,o_n,o)\}$$

where s'_k is the new range for o_k:

$$s'_k=\{x\in s_k \mid p_1(x)\}$$

which is, by definition, the set value produced by $\sigma_{p_1(y)}(s_k)$. Immediately, we know that the new set value is the set produced by the RHS expression:

$$\sigma_{p_2(x_1,\ldots,x_n,x)}(s_1,\ldots,s_{k-1},\sigma_{p_1(y)}(s_k),s_{k+1},\ldots,s_n,r)$$

In a similar fashion, we can prove that the LHS and RHS expressions contain the same members when the target operand r is a SEQ object. Furthermore, the expressions on the two sides of the equation preserve the same ordering of the target operand r. Hence, the theorem also holds when r is a SEQ object.

2. Let r ⓘ CAPSULE, AGG or BIO. The LHS expression of the equation produces the target operand r itself if r satisfies the following quantified predicate:

$$\exists o_1\in s_1,\ldots,\exists o_k\in s_k,\ldots,\exists o_n\in s_n: (p_1(o_k)\wedge p_2(o_1,\ldots,o_k,\ldots,o_n,r))$$

which is equivalent in meaning to

$$\exists o_1 \in s_1, ..., \exists o_k \in \{y \in s_k | \, p_1(y)\}, ..., \exists o_n \in s_n : p_2(o_1, ..., o_k, ..., o_n, r)$$

which is the quantified predicate satisfied by the RHS expression when it produces the target operand r. Hence, the two expressions are identical.

■

Note that the above theorem can be obviously extended to the case where more than one non-target operand has the property like x_k.

Example 65: For query "find all employees in *MyGroup* that are younger than an employee in *XGroup* who is married", we can formulate the following object algebra expression:

$$\sigma_{x.Age<x_1.Age \wedge \neg(x_1.Spouse \equiv \mathbf{null})}(XGroup, MyGroup)$$

According to Theorem 22, we can transform this expression into the following one:

$$\sigma_{x.Age<x_1.Age}(\sigma_{\neg(y.Spouse \equiv \mathbf{null})}(XGroup), MyGroup))$$

where y ranges over $XGroup$, x_1 ranges over the result of the inner Select operation, and x ranges over $MyGroup$. ■

Similar to the above two theorems, we have the following two theorems for the Pickup operation.

Theorem 24: *Idempotence of Pickup.* If r ⓘ CAPSULE, AGG or BIO, then

$$\delta_{p_1(x_1,...,x_k,x) \wedge p_2(x_{k+1},...,x_n,x)}(s_1, ..., s_n, r)$$
$$\equiv$$
$$\delta_{p_1(x_1,...,x_k,x)}(s_1, ..., s_k, \delta_{p_2(x_{k+1},...,x_n,x)}(s_{k+1}, ..., s_n, r))$$

Proof: When r is a CAPSULE, AGG or BIO object, the Pickup operator by definition has the same semantics as the Select operator. Therefore, from Theorem 21, this theorem holds too. ■

Theorem 25: *Idempotence of Pickup.* If r ⓘ CAPSULE, AGG, BIO or SEQ, then

$$\delta_{p_1(x_k) \wedge p_2(x_1,...,x_k,...,x_n,x)}(s_1, ..., s_n, r)$$
$$\equiv$$
$$\delta_{p_2(x_1,...,x_n,x)}(s_1, ..., \delta_{p_1(y)}(s_k), ..., s_n, r)$$

Proof: When r is a CAPSULE, AGG or BIO object, the Pickup operator by definition has the same semantics as the Select operator. Therefore, from Theorem 22, this theorem holds too. When r is a SEQ object, the LHS expression and the RHS expression both pick up the first member in r (if any) that satisfies p_1 and p_2. Hence, the LHS and RHS expressions are also identical when r is a SEQ object. ■

It should be noted that the equations in the above two theorems do not hold when r is a SET object since two Pickup operations on the same operand may pick up two different members that satisfy the predicates. However, such equations may be used as semantic transformations when the target operand r is a SET object, since users may not care about which member is "picked up" so long as it satisfies the desired predicates. Semantic transformations of KBO algebra expressions will be an interesting future research topic.

In the following, without loss of generality, we can omit the blackboard which may be specified for the Apply operator. The generalization of the following theorem to include the blackboard (which supplies labeled arguments) is straightforward.

Theorem 26: *Idempotence of Apply.* Let s, r be any kind of KBO objects.

$$\rho_{p_1(x_s) \wedge p_2(x_r)}(s, r) \asymp_1 \rho(\sigma_{p_1(x_s)}(s), \sigma_{p_2(x_r)}(r))$$

Proof: Let us first consider the case where s and r are both SET objects. The LHS expression produces a new object-identity with a set value defined as

$$\{\ \{o_s \rightsquigarrow o_r |\ o_s \in s \wedge (p_1(o_s) \wedge p_2(o_r))\}\ |\ o_r \in r\}$$

Since p_1 and p_2 do not share object variables, this expression can be rewritten as

$$\{\ \{o_s \rightsquigarrow o_r |\ o_s \in s \wedge p_1(o_s)\}\ |\ o_r \in r \wedge p_2(o_r)\}$$

We change the ranges of elements o_s and o_r as follows

$$\{\ \{o_s \rightsquigarrow o_r |\ o_s \in s'\}\ |\ o_r \in r'\}$$

where $s' = \{x \in s|\ p_1(x)\}$ and $r' = \{x \in r|\ p_2(x)\}$. Immediately, this expression is the set produced by the RHS expression of the equation. When s and r are both SEQ objects or one of them is a SEQ object, the expressions on the two sides of the equation also have the same ordering because the two Select operations on the RHS of the equation preserves the ordering of s and r. In a similar fashion, we can show that theorem holds trivially in other cases. ■

Theorem 27: *Idempotence of Assimilate.* Let $l \leq k$.

1. If r $\textcircled{i}$ CAPSULE, AGG or BIO, then

$$\alpha_k(r) \equiv \alpha_l(\alpha_k(r))$$

2. If r $\textcircled{i}$ SET, then

$$\alpha_k(r) \asymp_{k+1} \alpha_l(\alpha_k(r))$$

3. If r $\textcircled{i}$ SEQ, then

$$\alpha_k(r) \asymp_1 \alpha_l(\alpha_k(r))$$

Proof: When r $\textcircled{i}$ CAPSULE, AGG or BIO, the first equation trivially holds because both sides returns the operand r itself.

When r $\textcircled{i}$ SET, the LHS expression $\alpha_k(r)$ returns a new object-identity (say o_1) with a set value in which members are pair-wise not k-equal. This implies that some members in r may have been dropped from the result o_1. On the RHS of the equation, $\alpha_k(r)$ returns a new object-identity (say o_2) with a set (or sequence) value in which members are pair-wise not k-equal. This again implies that some members in r may have been dropped from the result o_2. Since $l \leq k$, this also implies that members in o_2 are also pair-wise not l-equal. Therefore, the result of $\alpha_l(\alpha_k(r))$ contains exactly the same members as those in o_2. Now, if a member x in r does not appear in o_1 (o_2) but appears in o_2 (o_1), then x must be k-equal to a member in o_1 (o_2) because this is the reason x was dropped from o_1 (o_2). This implies that the LHS and RHS expressions will have members which are pairwise k-equal. Since the two set objects produced by the two expressions have a new object-identity (i.e., they are not identical), the two set objects are $k+1$ equal.

Now we consider the case where r $\textcircled{i}$ SEQ. By definition, if there are two or more k-equal members in the sequence object r, the operation $\alpha_k(r)$ always keeps the first such member in the sequence and drops the rest. Therefore, the two sequences produced by the LHS and RHS expressions contain same members and have the same ordering. This means that two sequences are 1-equal. ■

9.5.2 Commutativity Properties

Theorem 28: *Commutativity of Select.*

1. If r $\textcircled{i}$ SET or SEQ, then

$$\sigma_{p(x_1,\ldots,x_n,x)}(s_1,\ldots,s_n,\sigma_{q(y_1,\ldots,y_m,y)}(s'_1,\ldots,s'_m,r))$$
$$\asymp_1$$

$$\sigma_{q(y_1,\ldots,y_m,y)}(s'_1,\ldots,s'_m,\sigma_{p(x_1,\ldots,x_n,x)}(s_1,\ldots,s_n,r))$$

2. If r ⓘ CAPSULE, AGG or BIO, then

$$\sigma_{p(x_1,\ldots,x_n,x)}(s_1,\ldots,s_n,\sigma_{q(y_1,\ldots,y_m,y)}(s'_1,\ldots,s'_m,r))$$
$$\equiv$$
$$\sigma_{q(y_1,\ldots,y_m,y)}(s'_1,\ldots,s'_m,\sigma_{p(x_1,\ldots,x_n,x)}(s_1,\ldots,s_n,r))$$

Proof:

1. Let r be a SET object. By definition, the LHS expression produces a new object-identity with a set value in which each member o is first selected from set r because it satisfies $q(y_1,\ldots,y_m,o)$, and then selected from the result of the inner Select operation because it satisfies $p(x_1,\ldots,x_n,o)$. The RHS expression produces a new object-identity with a set value in which each member o is first selected from set r because it satisfies $p(x_1,\ldots,x_n,o)$, and then selected from the result of the inner Select operation because it satisfies $q(y_1,\ldots,y_m,o)$. Therefore, the results of the LHS expression and RHS expression contain exactly the same members selected from r, independent of the order in which they are evaluated against the two predicates p and q.

   Similarly, we can prove that the theorem holds when r is a SEQ object. In particular, for any o_1, o_2 in sequence r, if o_1 is positioned before o_2 and if they are both selected in the result from either the LHS expression or RHS expression, they will still be positioned in the same order because Select operations do not change the ordering of those selected members from the target operand r.

2. If r is a CAPSULE, AGG or BIO object, and if r satisfies both $p(x_1,\ldots,x_n,r)$ and $q(y_1,\ldots,y_m,r)$, both the LHS expression and the RHS expression produce r itself because the inner Select operation on both sides produces r itself. Therefore, the LHS expression and the RHS expression produce the identical result r (or ***null***).

■

Theorem 29: *Commutativity of Pickup.* If r ⓘ CAPSULE, AGG or BIO, then

$$\delta_{p(x_1,\ldots,x_n,x)}(s_1,\ldots,s_n,\delta_{q(y_1,\ldots,y_m,y)}(s'_1,\ldots,s'_m,r))$$

$$\equiv$$

$$\delta_{q(y_1,\ldots,y_m,y)}(s'_1,\ldots,s'_m,\delta_{p(x_1,\ldots,x_n,x)}(s_1,\ldots,s_n,r))$$

Proof: On the LHS of the equation, r is picked up if r satisfies first $q(y_1,\ldots,y_m,r)$ and then $p(x_1,\ldots,x_n,r)$. On the RHS of the equation, r is picked up if r satisfies first $p(x_1,\ldots,x_n,r)$ and then $q(y_1,\ldots,y_m,r)$. Since r is the same object in both predicates, the result r (or ***null***) will be returned regardless of which predicate is evaluated first. ■

Note that this theorem does not hold for sequences because the two inner Pickup operations on both sides may pick up different members since they have different predicates.

Theorem 30: *Commutativity of Union and Intersect.* If r, s ⓘ SET, then

$$\cup(s,r) \asymp_1 \cup(r,s)$$

$$\cap(s,r) \asymp_1 \cap(r,s)$$

Proof: By definition, these two equations are obviously valid. That is, the two sides contain the same members but not identical, because Union/Intersect operations always generate a new object-identity. Hence, they can only be 1-equal. ■

Note that the above theorem does not hold when r, s are SEQ objects (or one of them is SEQ object) because the two expressions can not have the same ordering.

9.5.3 Associativity Properties

Theorem 31: *Associativity of Union and Intersect.* If all of r, s, t ⓘ SET, or all of r, s, t ⓘ SEQ, then

$$\cup(\cup(r,s),t) \asymp_1 \cup(r,\cup(s,t))$$

$$\cap(\cap(r,s),t) \asymp_1 \cap(r,\cap(s,t))$$

Proof: When r, s, t are all SET objects, these two equations are obviously valid from the definition of $\cup$ and $\cap$ operators. When r, s, t are all SEQ objects, by definition both sides of the first equation produce a sequence that is the concatenation of r, s, t, and both sides of the second equation produce a sequence that preserves the ordering of r among those members of r which are also contained in sequences s and t. Therefore, the results from both sides not only contain the same members but also have the same ordering. ■

9.5.4 Distributivity Properties

Theorem 32: *Distributivity of Select over Combine.* If r, r' ⓘ SET or SEQ, then

$$
\begin{aligned}
&\sigma_{p_1(x_1,\ldots,x_k,x.A)\wedge p_2(x_{k+1},\ldots,x_n,x.B)}(s_1, \ldots, s_n, (\chi_{A,B}(r, r'))) \\
&\asymp_1 \\
&\chi_{A,B}(\sigma_{p_1(x_1,\ldots,x_k,y)}(s_1, \ldots, s_k, r), \sigma_{p_2(x_{k+1},\ldots,x_n,z)}(s_{k+1}, \ldots, s_n, r'))
\end{aligned}
$$

Proof: Let both r and r' be SET objects. Then, on the LHS of the equation, the Combine operation $\chi_{A,B}(r, r')$ produces a new object-identity (say w) with the set value defined as

$$\{[A\colon o, B\colon o'] \mid o \in r \wedge o' \in r'\}$$

Again, without loss in generality, we let $s_1, \ldots, s_n$ be SET or SEQ objects. Then, the Select operation

$$\sigma_{p_1(x_1,\ldots,x_k,x.A)\wedge p_2(x_{k+1},\ldots,x_n,x.B)}(s_1, \ldots, s_n, w)$$

returns a new object-identity with set value

$$\{o \in w \mid \exists o_1 \in s_1, \ldots, \exists o_n \in s_n : (p_1(o_1, \ldots, o_k, o.A) \wedge p_2(o_{k+1}, \ldots, o_n, o.B))\}$$

Since p_1 and p_2 do not share object variables (except o), the above set value is equivalent in meaning to

$$
\begin{aligned}
&\{o \in w \mid \exists o_1 \in s_1, \ldots, \exists o_k \in s_k : p_1(o_1, \ldots, o_k, o.A) \\
&\wedge \exists o_{k+1} \in s_{k+1}, \ldots, \exists o_n \in s_n : p_2(o_{k+1}, \ldots, o_n, o.B))\}
\end{aligned}
$$

Based on the value of w, this expression can be rewritten as

$$
\begin{aligned}
&\{\ [A\colon o, B\colon o'] \in w \mid \\
&\quad \exists o_1 \in s_1, \ldots, \exists o_k \in s_k : p_1(o_1, \ldots, o_k, o) \wedge \exists o_{k+1} \in s_{k+1}, \ldots, \exists o_n \in s_n : p_2(o_{k+1}, \ldots, o_n, o'))\}
\end{aligned}
$$

which is equivalent in meaning to

$$\{[A\colon o, B\colon o'] \mid o \in t \wedge o' \in t'\} \tag{9.1}$$

where

$$t = \{x \in r \mid \exists o_1 \in s_1, \ldots, \exists o_k \in s_k : p_1(o_1, \ldots, o_k, x)\}$$

and

$$t' = \{x \in r' \mid \exists o_{k+1} \in s_{k+1}, \ldots, \exists o_n \in s_n : p_2(o_{k+1}, \ldots, o_n, x)\}$$

Immediately, the set expression (9.1) is the set value produced by the RHS expression of the equation.

Now suppose r and r' are both SEQ objects. The ordering of the LHS expression preserves the ordering of $\chi_{A,B}(r,r')$ since the Select operation does not change the ordering of a sequence operand. By definition, the ordering of $\chi_{A,B}(r,r')$ is in turn decided by the ordering of r and the ordering of r'. The ordering of the RHS expression is decided by the ordering of the result of the first Select operation and the result of the second Select operation. However, since the results of Select operations preserves the ordering of their sequence operands (i.e., the target operands), the result from the Combine operation

$$\chi_{A,B}(\sigma_{p_1(x_1,\ldots,x_k,y)}(s_1,\ldots,s_k,r),\sigma_{p_2(x_{k+1},\ldots,x_n,z)}(s_{k+1},\ldots,s_n,r'))$$

has an ordering which is also decided by the orderings of r and r'. Hence, the LHS expression and the RHS expression not only have the same members but also the same ordering. It follows that they are 1-equal. Similarly, we can prove that theorem holds when r is a SET (or SEQ) object and r' is a SEQ (or SET) object. ■

Theorem 33: *Distributivity of Select over Union.* If r,r' ⓘ SET or r,r' ⓘ SEQ, then

$$\sigma_{p(x_1,\ldots,x_n,x)}(s_1,\ldots,s_n,\cup(r,r')) \asymp_1 \cup(\sigma_{p(x_1,\ldots,x_n,x)}(s_1,\ldots,s_n,r),\sigma_{p(x_1,\ldots,x_n,x)}(s_1,\ldots,s_n,r'))$$

Proof: Let LHS denote the set value produced by the LHS expression, and RHS denote the set value produced by the RHS expression. Let o be any object-identity contained in set RHS. Again, without loss of generality, we assume that $s_1,\ldots,s_n$ are all SET or SEQ objects. Let r and r' be two SET objects. We have the following logical equivalences:

$o \in RHS$

$\iff (o \in r \wedge \exists x_1 \in s_1 \ldots \exists x_n \in s_n : p(x_1,\ldots,x_n,o)) \vee$

$\quad (o \in r' \wedge \exists x'_1 \in s_1 \ldots \exists x'_n \in s_n : p(x'_1,\ldots,x'_n,o))$

We can use bound variables $x_1,\ldots,x_n$ to replace $x'_1,\ldots,x'_n$ because they have the same ranges and they quantify the same predicate:

$\iff \exists x_1 \in s_1 \ldots \exists x_n \in s_n : (o \in r \wedge p(x_1,\ldots,x_n,o)) \vee (o \in r' \wedge p(x_1,\ldots,x_n,o))$

$\iff \exists x_1 \in s_1 \ldots \exists x_n \in s_n : (o \in r \vee o \in r') \wedge p(x_1,\ldots,x_n,o)$

$\iff o \in LHS$

To show the theorem holds when both r and r' are both SEQ objects, we only need to show that the LHS expression and the RHS expression also have the same ordering. Since the Select operation does not change the ordering of the target operand, the

LHS expression preserves the ordering of the concatenation of r and r'. On the RHS of the equation, since the first Select operation preserves the ordering of r and the second Select operation preserves the ordering of r', the concatenation of them preserves the ordering of the concatenation of r and r'. (Note that the theorem does not hold if we switch the two Select operations on the RHS). Therefore, the LHS and RHS expressions have the same ordering. ■

In the following two theorems, we only show the case where the blackboard is not needed; the generalization of the theorem to include a blackboard is straightforward.

Theorem 34: *Distributivity of Apply over Union.* If r, r' ⓘ SET or r, r' ⓘ SEQ, and r and r' are disjoint (i.e., they do not have identical members), then

$$\rho_{p(x_s,x_r)}(s, \cup(r,r')) \asymp_1 \cup(\rho_{p(x_s,x_r)}(s,r), \rho_{p(x_s,x_r)}(s,r'))$$

Proof: Let us first consider the case where s ⓘ CAPSULE, AGG or BIO. Let r and r' be two SET objects. Then, the LHS expression produces a new object-identity with a set value defined as

$$\{s \leadsto x \mid (x \in r \vee x \in r') \wedge p(s,x)\}$$

which can be rewritten as

$$\{s \leadsto x \mid (x \in r \wedge p(s,x)) \vee (x \in r' \wedge p(s,x))\}$$

Since r and r' are SET objects, their union does not contain two identical members. Therefore, if there is x such that both $x \in r$ and $x \in r'$ are true, then message-sending $s \leadsto x$ in the above expression only happens once (i.e., there is only one of them in the set $\cup(r,r')$), not twice. Because r and r' are disjoint, the above expression is equivalent in meaning to the union of

$$\{s \leadsto x \mid x \in r \wedge p(s,x)\} \qquad \text{and} \qquad \{s \leadsto x \mid x \in r' \wedge p(s,x)\}$$

which is the set value of the result produced by the RHS expression.

When r and r' are both SEQ objects, the LHS expression produces a sequence containing the results of sending s to each qualified member in the concatenation of r and r', as follows:

$$< s \leadsto x_1, ..., s \leadsto x_n, s \leadsto y_1, ..., s \leadsto y_m >$$

where $x_i \in r$ and $p(s,x_i)$ is true, and $y_j \in r'$ and $p(s,y_j)$ is true. Since r and r' do not contain identical members, the first sub-sequence

$$< s \leadsto x_1, ..., s \leadsto x_n >$$

is exactly the sequence value produced by $\rho_{p(x_s,x_r)}(s,r)$, and the second sub-sequence

$$< s \rightsquigarrow y_1, ..., s \rightsquigarrow y_m >$$

is exactly the sequence value produced by $\rho_{p(x_s,x_r)}(s,r')$. Since the RHS expression produces the concatenation of the above two sequences, the LHS and RHS expressions produce the same sequence. When s is a SET or SEQ object, the theorem can be proved in a similar fashion. ■

Theorem 35: *Distributivity of Project over Union.* If r, r' Ⓘ SET or r, r' Ⓘ SEQ, and r and r' are disjoint (i.e., they do not have identical members), then

$$\pi_{A_1,...,A_n}(s_1, ..., s_n, \cup(r, r')) \asymp_2 \cup(\pi_{A_1,...,A_n}(s_1, ..., s_n, r), \pi_{A_1,...,A_n}(s_1, ..., s_n, r'))$$

Proof: Let r and r' be two SET objects. The LHS expression produces a new object-identity with a set value defined as

$$\{[A_1: s_1 \rightsquigarrow x, ..., A_n: s_n \rightsquigarrow x] \| \; x \in r \vee x \in r'\}$$

Because r and r' are SET objects, their union does not contain two identical members. Therefore, if there is x such that both $x \in r$ and $x \in r'$ are true, then message-sendings $s_1 \rightsquigarrow x, ..., s_n \rightsquigarrow x$ should only occur once, not twice, in the above expression. Since r and r' are disjoint, the above expression is equivalent in meaning to the union of

$$\{[A_1: s_1 \rightsquigarrow x, ..., A_n: s_n \rightsquigarrow x] \| \; x \in r\} \quad \text{and} \quad \{[A_1: s_1 \rightsquigarrow x, ..., A_n: s_n \rightsquigarrow x] \| \; x \in r'\}$$

which is the set value of the result produced by the RHS expression. However, the LHS and RHS expressions will generate a new object-identity for their result (a set object) and new object-identities for the aggregate objects contained in their result, and, therefore, the two expressions can only be 2-equal.

When r and r' are both SEQ objects, the LHS expression produces a sequence of results of sending s to each qualified member in the concatenation of r and r'. The first part of this sequence has the same aggregate values and the ordering as $\pi_{A_1,...,A_n}(s_1, ..., s_n, r)$ and the second part of this sequence has the same aggregate values and the ordering as $\pi_{A_1,...,A_n}(s_1, ..., s_n, r')$. Therefore, the two expressions are 2-equal. ■

Theorem 36: *Distributivity of Assimilate over Union.* If r, r' Ⓘ SET, then

$$\alpha_k(\cup(r, r')) \asymp_{k+1} \cup(\alpha_k(r), \alpha_k(r'))$$

Proof: The LHS expression produces a new object-identity with a set value in which members are either from r or r', and are pair-wise not k-equal. Let o_1 be an

arbitrary member in the result of the LHS expression. Suppose o_1 is originally from set r. Then o_1 is either in the result of the RHS expression or has been dropped by operation $\alpha_k(r)$. If the former is true, then o_1 is of course k-equal to itself. If the latter is true, then o_1 is k-equal to another member in $\alpha_k(r)$ which has been unioned into the result of RHS expression. Therefore, in either case, o_1 is k-equal to a member in the result of the RHS expression. By similar argument, we can show that if o_1 is from r' then o_1 is also k-equal to a member in the result of the RHS expression. Now let o_2 be an arbitrary member in the result of the RHS expression. Suppose o_2 is originally from the result of $\alpha_k(r)$. Then o_2 is either in the result of the LHS expression or has been dropped by the operation $\alpha_k(\cup(r,r'))$. If the former is true, then o_2 is of course k-equal to itself. If the latter is true, then o_2 must be k-equal to another member remaining in the result of LHS expression. Therefore, in either case o_2 is k-equal to a member in the result of the LHS expression. By similar argument, we can show that if o_2 is originally from $\alpha_k(r')$ then o_2 is still k-equal to a member in the result of the LHS expression. Since the LHS (RHS) result contains members that have a k-equal counterpart in RHS (LHS) result, the two results from the two sides are $k+1$ equal. ■

Note that the above theorem does not hold if r, r' are SEQ objects. This is because the RHS expression may produce a sequence containing two k-equal members while the LHS expression can only keep the first one.

Theorem 37: *Distributivity of Collapse over Union.* If r, r' ⓘ SET or r, r' ⓘ SEQ, then

$$\varpi(\cup(r,r')) \asymp_1 \cup(\varpi(r), \varpi(r'))$$

Proof: Let r and r' be two SET objects. Then, the LHS expression produces a new object identity with set value defined as

$$\{o | \exists x : (x \in r \vee x \in r') \wedge o \in x\}$$

Since a SET object does not contain two identical objects (i.e., set membership is based on object-identity), the above expression is equivalent to

$$\{o | \exists x \in r : o \in x \vee \exists y \in r' : o \in y\}$$

which is equivalent in meaning to the (identity-based) union of

$$\{o | \exists x \in r : o \in x\} \quad \text{and} \quad \{o | \exists y \in r' : o \in y\}$$

which is the set value produced by the RHS expression of the equation.

Suppose now that r and r' are two SEQ objects, and $r = < x_1, ..., x_n >$ and $r' = < y_1, ..., y_m >$. Then, the LHS expression produces a new object-identity with the sequence value

$$x_1 \star \cdots \star x_n \star y_1 \star \cdots \star y_m$$

which can be rewritten as

$$(x_1 \star \cdots \star x_n) \star (y_1 \star \cdots \star y_m)$$

which is equivalent in meaning to

$$\varpi(r) \star \varpi(r')$$

This expression is the result of the union of $\varpi(r)$ and $\varpi(r')$ — the RHS expression. Since the RHS expression also generates a new object-identity, the LHS and RHS expressions can only be 1-equal. ■

Theorem 38: *Distributivity of Select over Intersect.* If r, r' Ⓘ SET or r, r' Ⓘ SEQ, ·n

$$\sigma_{p(x_1,...,x_n,x)}(s_1, ..., s_n, \cap(r, r')) \asymp_1 \cap(\sigma_{p(x_1,...,x_n,x)}(s_1, ..., s_n, r), \sigma_{p(x_1,...,x_n,x)}(s_1, ..., s_n, r'))$$

Proof: Let r and r' be two SET objects. Let LHS denote the set value produced by the LHS expression, and RHS denote the set value produced by the RHS expression. Let o be any object-identity contained in set RHS. Again, without loss in generality, we assume that $s_1, ..., s_n$ are all SET or SEQ objects. Then, we have the following logical equivalences:

$o \in RHS$
$\Longleftrightarrow (o \in r \wedge \exists x_1 \in s_1 ... \exists x_n \in s_n : p(x_1, ..., x_n, o)) \wedge$
$\quad (o \in r' \wedge \exists x'_1 \in s_1 ... \exists x'_n \in s_n : p(x'_1, ..., x'_n, o))$

We can use bound variables $x_1, ..., x_n$ to replace $x'_1, ..., x'_n$ because they have the same ranges and they quantify the same predicate:

$\Longleftrightarrow \exists x_1 \in s_1 ... \exists x_n \in s_n : (o \in r \wedge p(x_1, ..., x_n, o)) \wedge (o \in r' \wedge p(x_1, ..., x_n, o))$
$\Longleftrightarrow \exists x_1 \in s_1 ... \exists x_n \in s_n : (o \in r \wedge o \in r') \wedge p(x_1, ..., x_n, o)$
$\Longleftrightarrow (o \in r \wedge o \in r') \wedge \exists x_1 \in s_1 ... \exists x_n \in s_n : p(x_1, ..., x_n, o)$
$\Longleftrightarrow o \in LHS$

To show the theorem holds when both r and r' are SEQ objects, we only need to show that the LHS expression and the RHS expression also have the same ordering, because from the above equivalences we know that they will contain the same members. Since the Select operation does not change the ordering of the target operand, the

LHS expression preserves the ordering of the intersection of r and r', which in turn preserves the ordering of r (the first operand in the Intersect operation). On the RHS of the equation, since the first Select operation preserves the ordering of r, the Intersect operation will also preserve the ordering of r. Therefore, the LHS and RHS expressions have the same ordering. Since both LHS and RHS expressions generate a new object-identity for their results, the LHS and RHS expressions can only be 1-equal. ■

****

Theorem 39: *Distributivity of Select over Subtract.* If r, r' Ⓘ SET or r, r' Ⓘ SEQ, then

$$\sigma_{p(x_1,...,x_n,x)}(s_1,...,s_n,-(r,r')) \asymp_1 -(\sigma_{p(x_1,...,x_n,x)}(s_1,...,s_n,r), \sigma_{p(x_1,...,x_n,x)}(s_1,...,s_n,r'))$$

Proof: Let r and r' be two SET objects. Let LHS denote the set value produced by the LHS expression, and RHS denote the set value produced by the RHS expression. Let o be any object-identity contained in set RHS. Again, without loss of generality, we assume that $s_1,...,s_n$ are all SET or SEQ objects. Then, we have the following logical equivalences:

$o \in RHS$

$\iff (o \in r \wedge \exists x_1 \in s_1 ... \exists x_n \in s_n : p(x_1,...,x_n,o)) \wedge$
$\quad \neg(o \in r' \wedge \exists x'_1 \in s_1 ... \exists x'_n \in s_n : p(x'_1,...,x'_n,o))$

We can use bound variables $x_1,...,x_n$ to replace $x'_1,...,x'_n$ because they have the same ranges and quantify the same predicate:

$\iff \exists x_1 \in s_1 ... \exists x_n \in s_n : (o \in r \wedge p(x_1,...,x_n,o)) \wedge \neg(o \in r' \wedge p(x_1,...,x_n,o))$

$\iff \exists x_1 \in s_1 ... \exists x_n \in s_n : (o \in r \wedge p(x_1,...,x_n,o)) \wedge (\neg(o \in r' \vee \neg p(x_1,...,x_n,o))$

$\iff \exists x_1 \in s_1 ... \exists x_n \in s_n : (o \in r \wedge p(x_1,...,x_n,o) \wedge \neg(o \in r')) \vee$
$\quad (o \in r \wedge p(x_1,...,x_n,o) \wedge \neg p(x_1,...,x_n,o))$

$\iff \exists x_1 \in s_1 ... \exists x_n \in s_n : (o \in r \wedge p(x_1,...,x_n,o) \wedge \neg(o \in r'))$

$\iff (o \in r \wedge \neg(o \in r')) \wedge \exists x_1 \in s_1 ... \exists x_n \in s_n : p(x_1,...,x_n,o)$

$\iff o \in LHS$

To show the theorem holds when r and r' are both SEQ objects, we only need to show that the LHS expression and the RHS expression also have the same ordering. Since the Select operation does not change the ordering of the target operand, the LHS expression preserves the ordering of $-(r,r')$, which, by definition, preserves the ordering of r (the first operand in the Subtract operation). On the RHS of the equation, since the first Select operation preserves the ordering of r, the Subtract operation will also preserve the ordering of r. Therefore, the LHS and RHS expressions have the same

ordering. Since both LHS and RHS expressions generate a new object-identity for their results, the LHS and RHS expressions can only be 1-equal. ■

Theorem 40: *Distributivity of Subtract over Union.* If r, s, t $\textcircled{i}$ SET or r, s, t $\textcircled{i}$ SEQ, then

$$-(r, \cup(s,t)) \asymp_1 \cap(-(r,s), -(r,t))$$

Proof: Let r, s, t be three SET objects. Let LHS denote the set value produced by the LHS expression, and RHS denote the set value produced by the RHS expression. Let o be any object-identity contained in set RHS. Then, we have the following logical equivalences:

$o \in RHS$
$\Longleftrightarrow (o \in r \wedge \neg(o \in s)) \wedge (o \in r \wedge \neg(o \in t))$
$\Longleftrightarrow o \in r \wedge (\neg(o \in s) \wedge \neg(o \in t))$
$\Longleftrightarrow o \in r \wedge \neg(o \in s \vee o \in t)$
$\Longleftrightarrow o \in LHS$

When r, s, t are SEQ objects, we only have to show that two expressions have the same ordering. By definition, the LHS expression will preserve the ordering of r, and the RHS expression will preserve the ordering of $-(r,s)$. Again by definition, $-(r,s)$ preserves the ordering of r. Therefore, the two expressions not only have the same members but also the same ordering. Since expressions on both sides generate a new object-identity, the two expressions can only be 1-equal. ■

Theorem 41: *Distributivity of Subtract over Intersect.* If r, s, t $\textcircled{i}$ SET or r, s, t $\textcircled{i}$ SEQ, then

$$-(\cap(r,s), t) \asymp_1 \cap(-(r,t), -(s,t))$$

Proof: Let r, s, t be three SET objects. Let LHS denote the set value produced by the LHS expression, and RHS denote the set value produced by the RHS expression. Let o be any object-identity contained in set RHS. Then, we have the following logical equivalences:

$o \in RHS$
$\Longleftrightarrow (o \in r \wedge \neg(o \in t)) \wedge (o \in s \wedge \neg(o \in t))$
$\Longleftrightarrow o \in r \wedge o \in s \wedge \neg(o \in t)$
$\Longleftrightarrow (o \in r \wedge o \in s) \wedge \neg(o \in t)$
$\Longleftrightarrow o \in LHS$

When r, s, t are SEQ objects, we only have to show that two expressions have the same ordering. By definition, the LHS expression will preserve the ordering of $\cap(r,s)$, which in turn preserves the ordering of r. The RHS expression will preserve the ordering of $-(r,t)$ which in turn preserves the ordering of r. Therefore, the two expressions not

only have the same members but also the same ordering. Since expressions on both sides generate a new object-identity, the two expressions can only be 1-equal. ■

9.6 An Evaluation of the KBO Algebra

In Chapter 2 we surveyed existing object algebras and gave a detailed evaluation of those object algebras using the "five category" evaluation framework proposed in [YO91a]. In this section, we will evaluate the KBO object algebra using the same evaluation framework. Such a comprehensive comparison with existing object algebras makes it much clearer to see what are the most unique features offered by the KBO algebra.

The following abbreviations will be used in our evaluation tables:

MD — Manola & Dayal's Object Algebra [MD86, OGM86, D*85, DS85]
Osb — Osborn's Object Algebra [Osb88, Osb89a, Osb89b]
SÖ — Straube & Özsu's Object Algebra [SO90, Str91]
SZ — Shaw & Zdonik's Object Algebra [SZ89, SZ90]
O2 — RELOOP Algebra for O2 [C*89]
OO — Dayal's OOAlgebra for OODAPLEX [Day89]
GSL — Guo, Su & Lam's Association Algebra for OQL [GSL91]
DD — Davis & Delcambre's Algebra [DD91]
VD — Vandenberg & DeWitt's Algebra [VD90, VD91]
KBO — The KBO Object Algebra

9.6.1 Object-Orientedness

- **Supports object identities and their manipulation.**
  The semantics of the KBO object algebra is defined over object-identities of all allowable objects in the KBO model. Furthermore, the KBO object algebra also supports arbitrary level identity-test and provides operations to manipulate identities, namely the Assimilate operator and the extended Union, Intersect and Subtract operators. Hence, the KBO object algebra satisfies this criterion.

- **Supports encapsulation.**
  The KBO object algebra supports encapsulation in the sense that no user's access to anything but the interface is allowed. It is only in the definition of the Collapse operator that object attributes seem to be directly manipulated by the operator when the operand is an AGG object. However, it should be noted that

the representation of an AGG class is really the specification of all the attribute access behaviors which are part of the interface of the class. It is invisible to the end-user whether they are implemented as stored data or computed data. (This also explains why when we simulate the relational algebra we have to require that each relation have a set of "get behaviors".) Therefore, the KBO algebra satisfies this criterion.

- **Supports inheritance hierarchy.**
  The KBO object algebra does not satisfy this criterion. The traditional "IS-A" semantics has to be realized by explicitly inserting objects of different types into a set or sequence object. However, an important advantage gained by doing so is a more "full-fledged" object algebra that operates on object-identities (rather than on classes) of any kinds of objects.

- **Supports polymorphism.**
  The KBO object algebra satisfies this criterion since all its operations take as input any kinds of KBO objects.

- **Distinguishes classes and collections.**
  The KBO object algebra satisfies this criterion by distinguishing between classes and set objects that have a class as the member type.

- **Supports heterogeneous sets/sequences.**
  The KBO object algebra supports heterogeneous sets in two different ways: (1) direct support — a direct instance of SET or SEQ may contain members of arbitrary classes, and (2) indirect support — any instance of a set or sequence class may contain as members the instances of any subclasses of that class. The latter can be realized by specifying the signature of an insertion behavior of the set/sequence class.

The above evaluations are summarized in the following table:

OBJECT-ORIENTEDNESS	MD	Osb	SO	SZ	O2	OO	GSL	DD	VD	KBO
Supports oids and their manipulation	◇	◇	◇	√	×	×	×	◇	◇	√
Supports encapsulation	√	×	√	√	×	×	×	×	×	√
Supports inheritance hierarchy	×	×	√	×	?	×	√	√	×	×
Supports polymorphism	×	√	×	×	×	√	×	×	×	√
Distinguishes classes and collections	×	√	×	√	√	√	×	×	√	√
Supports heterogeneous sets	?	√	√	√	√	?	√	√	?	√

√ — satisfies criterion ◇ — partial compliance

× — criterion not satisfied ? — not detailed in papers

9.6.2 Expressiveness

- **Extends relational algebra consistently.**
  We have proven (Theorem 20) that the KBO object algebra is at least as expressive as the relational algebra. Hence, the algebra complies with this criterion.

- **Includes object constructors.**
  In the KBO object algebra, object creation for user-defined classes is left to the behaviors of those classes (like the constructors in C++). The algebra can only create new (direct) objects of AGG, BIO, SET or SEQ classes from existing objects. For example, using our Project operator on an AGG object can create a new AGG object, and using Union operator on two SEQ objects can create a new SEQ object. However, the KBO object algebra does not explicitly include object construction operators like those in Osborn's algebra [Osb88, Osb89a, Osb89b]. Hence, the KBO algebra does not satisfy this criterion.

- **Supports sequence objects.**
  The KBO algebra supports the manipulation of sequence objects. In addition, many formal transformations for sequence operands are presented. Note that the only other object algebra that manipulates sequences is Vandenberg and DeWitt's algebra [VD90, VD91]. (Osborn's algebra is now being extended to operate on sequences) Hence, the KBO algebra satisfies this criterion.

- **Supports invocation of behaviors.**
  The KBO object algebra supports invocation of arbitrary behaviors in the following four different ways: (1) a message-sending can be used as the input of any KBO algebra operation; (2) any KBO algebra expression can be used either as the invoker or as the receiver of a message-sending; (3) message-sendings can be expressed in the predicates for Select, Pickup and Apply operations, and, in particular, the invoker of such message-sendings can be expressed as an object variable ranging over an operand in Select and Pickup operations; (4) the Apply and Project operators are defined as a group of structured message-sendings, and, in particular, the Apply operator can send a different invoker (or invokers) to different members in a set/sequence object. To the best of our knowledge, there is no other object algebra that supports the flexibility offered by (2), (3) and (4). Operators like Apply_Append in Manola & Dayal's algebra, Image in

Shaw & Zdonik's algebra, and Map in Straube & Özsu's algebra, can only apply a constant behavior (or behaviors in Map) to each member in a set. Hence, the KBO algebra complies with this criterion.

- **Includes behavior constructors.**
  The behavior constructors are actually data structures in the KBO object algebra, since any object in our model can be used as an invoker. One can always use the Apply operator to apply a CAPSULE object, an AGG object, a BIO object, a SET object, or a SEQ object, to a receiver (another CAPSULE, AGG or BIO object) or a group of receivers (another SET or SEQ object). Hence, the KBO object algebra satisfies this criterion.

- **Supports dynamic type creation.**
  Although the KBO algebra operations do not create new classes, it does create new objects without new classes. In most cases, the KBO algebra operations create such new objects as direct instances of kernel classes AGG, BIO, SET or SET. To organize such new objects, we could introduce an operator like ReClass in Osborn's algebra. However, we feel that such saving facilities can also be realized outside of the algebra (such as in Manola & Dayal's algebra) after the query session is over. The user can always pause to create a new class and save the previous query's result under that class, and then continue to the next query (if necessary). We could also explicitly introduce parameterized classes (like the types Set[T] and Tuple[$(A_1, T_1), ..., (A_n, T_n)$] in Shaw and Zdonik's algebra) to hold these new objects. However, it is not realistic to let the system automatically create a parameterized class to hold each intermediate result in a query expression. A complex query (nested many times) may have a lot of intermediate results which are not of interest to the user. Instead, we achieve the same flexibility by allowing our algebra to operate not only on AGG or SET objects but also direct instances of AGG and SET. (Note that the user can always save the new objects by manually creating a new class.) Therefore, we consider that the KBO object algebra partially satisfies this criterion.

- **Supports querying transitive closures.**
  The KBO object algebra does not support this feature.

- **Supports behaviors-as-objects.**
  This is the most unique feature supported by the KBO algebra. In Chapter 8 we have demonstrated that the KBO algebra can be used to (associatively)

select, generate, appl, and maintain behaviors-as-objects. We feel that this is the most important contribution of this algebra to OODB research, because it opens up a brand new area in the use of OODB query facilities.

The above evaluations are summarized in the following tables:

EXPRESSIVENESS	MD	Osb	SO	SZ	O2	OO	GSL	DD	VD	KBO
Extends relational algebra consistently	√	√	×	√	√	√	√	√	√	√
Includes object constructors	◇	√	×	×	√	√	×	√	√	×
Supports sequence objects	×	×	×	×	×	×	×	×	√	√
Supports invocation of behaviors	√	×	√	√	×	√	√	×	√	√
Includes behavior constructors	×	×	◇	×	×	×	×	×	×	√
Supports dynamic type creation	√	√	×	√	√	√	×	√	√	>
Supports querying transitive closures	√	×	×	×	×	×	√	×	×	×
Supports behaviors-as-objects	×	×	×	×	×	×	×	×	×	√

√ — satisfies criterion ◇ — partial compliance

× — criterion not satisfied ? — not detailed in papers

9.6.3 Formalness

- **Has a formal semantics.**

  The KBO object algebra satisfies this criterion since the semantics of all its operators is formally defined in Chapter 8. Based on our formal definitions, we also proved the correctness of some general algebraic transformations earlier in this chapter.

- **Is a closed algebra.**

  We have proven in Theorem 19 that the KBO object algebra is a closed object algebra.

The above evaluations are summarized in the following tables:

FORMALNESS	MD	Osb	SO	SZ	O2	OO	GSL	DD	VD	KBO
Has a formal semantics	×	×	√	◇	√	√	√	√	√	√
Is a closed algebra	×	×	√	√	√	√	√	√	√	√

√ — satisfies criterion ◇ — partial compliance

× — criterion not satisfied ? — not detailed in papers

9.6.4 Performance

- **Supports strong typing.**

  The KBO object algebra is not strongly typed because our operands are usually untyped handles denoting an object-identity. However, the real problem that prevents the KBO algebra from being strongly typed is the way we classify behaviors and complex behaviors — the class of a behavior has nothing to do with its signature, and it only reflects what value the behavior has (i.e., a text string, a set, a sequence, etc.). Therefore, unless all behaviors appearing in a query (such as Apply) are expressed as constants, we can only do the type-checking at run-time. In Chapter 7 we have obtained some formal results that can be used to implement the type-checking at the message-sending level.

  Again, we need to point out that the KBO model was originally intended for interactive database applications that desire a high experimentability on manipulating massive collections of behaviors, and we thus consider this disadvantage a reasonable trade-off for our research.

  It should be noticed that according to Albano's definition [A*89a]: "A language is strongly typed if all computations are checked for type errors. A strongly typed language is statically checked if all type errors are discovered at compile time, and dynamically checked if type errors are only discovered at run time", our algebra is actually "strongly typed", in the sense that our operators are defined to operate on any kind of KBO objects and that operators (Apply, Project, and predicates in Select, Pickup, Apply) that may cause message-sendings will be dynamically checked at the $\leadsto$ level by the $\vdash$ system developed in Chapter 7.

- **Provides algebraic optimization strategies.**

  We have provided provably correct algebraic transformations for the KBO algebra. Hence, this criterion is complied with.

The above evaluations are summarized in the following tables:

PERFORMANCE	MD	Oob	SO	SZ	O2	OO	GSL	DD	VD	KBO
Supports strong typing	√	x	√	√	√	?	√	√	?	x
Provides algebraic optimisation strategies	?	√	√	√	?	?	√	√	√	√

√ — satisfies criterion
◇ — partial compliance
× — criterion not satisfied
? — not detailed in papers

9.6.5 Database Issues

- **Supports both persistent and transient objects.**

  Like Manola & Dayal's algebra, we choose to deal with persistence outside of the algebra, and we thus assume that all query results and intermediate results are transient objects (saving them is outside the algebra). Hence, the KBO object algebra does not satisfy this criterion.

- **Supports schema evolution.**

  The KBO object algebra does not support schema evolution. However, it should be noted that the polymorphic nature of the KBO algebra operators can support "object value evolution", such as the case where an object with a set value evolves to have a sequence value.

- **Has an equivalent object calculus.**

  The KBO object algebra does not have an equivalent object calculus; this will be a future research topic.

The above evaluations are summarized in the following tables:

DATABASE ISSUES	MD	Osb	SO	SZ	O2	OO	GSL	DD	VD	KBO
Supports persistent & transient objects	×	◇	×	?	?	?	×	×	×	×
Supports schema evolution	×	◇	×	×	×	×	×	×	×	×
Has an equivalent object calculus	×	×	◇	×	×	×	?	◇	?	×

√ — satisfies criterion

× — criterion not satisfied

◇ — partial compliance

? — not detailed in papers

CHAPTER 10
CONCLUSIONS

This chapter includes a summary of the results presented in the thesis. We discuss the major contributions and novelty of this research. Finally we outline the directions along which work can be pursued in the future to make KBO a viable system for building object-oriented database applications.

10.1 Summary of Research

The primary objective of this research was to develop a formal data model that unifies data, behaviors, and messages in OODBs into a uniform notion of objects, so that a *single* database approach can be adopted to manage and manipulat such objects. We have achieved this objective. The KBO model, whose design we have discussed in this thesis, provides the necessary primitives in six coherently related aspects to accomplish the task of representing, organizing and manipulating such uniformly modeled objects.

Chapter 1 motivated work by showing the need for a database approach to the management of arbitrary behaviors in OODBs. A detailed survey of the literature was then accomplished in Chapter 2. We discussed, in general, object-oriented programming systems, object-oriented data models, and object-oriented query algebras. Current OODB models were found to be insufficient in that they fail to offer the capability of managing arbitrary behaviors as intuitively meaningful database objects. The idea of a single database approach to the problem was briefly described in terms of classification, re-utilization, identification, invocation, composition, and manipulation of unified KBO objects. It was not our intention to develop a separate database approach only for the management of arbitrary behaviors.

Chapter 3 presented a formal description of the basic modeling constructs and abstractions in the KBO model. The notion of knowhow-bearing objects was introduced: data are objects bearing an empty knowhow, behaviors are objects bearing a non-empty knowhow, and messages are objects bearing either an empty or non-empty knowhow. A message-sending is thus viewed as sending an invoker object to a receiver object. Five abstraction constructs, namely the INSTANCE-OF, BEHAVIOR-OF, KIND-OF, REUSE-OF and PART-OF relationships, are defined, providing a conceptual level semantics for the KBO model. Every object is not only an INSTANCE-OF of a class or classes, but also a BEHAVIOR-OF of a class or classes. Classes can

be organized into two independent lattices based on the KIND-OF and REUSE-OF abstractions. Four kinds of aggregations (or associations) can be modeled by the PART-OF relationships: AGG objects, BIO objects, SET objects and SEQ objects. Unlike conventional object constructors, our four object constructors can be used not only for structural modeling but also for behavioral modeling.

Chapter 4 defined complete structures of KBO objects and KBO classes, based on the formal conceptual semantics given in Chapter 3. All KBO objects have an identity, a value, a class, and a vigor. Vigors allow objects to share their knowhows with different signatures. Objects can be compared in terms of multiple level equalities, as well as knowhow equality. A class is characterized by its direct superclasses, direct supplyclasses, representation, interface, and value-based interface. Techniques were presented for resolving possible conflicts in the derivation of a class's representation and value-based interface. A class's interface may contain equal behaviors but no identical behaviors, while its value-based interface contains no equal behaviors. The former provides support for monomorphic invocation while the latter provides support for polymorphic invocation.

Chapter 5 introduced three interpretation functions (static interpretation, dynamic interpretation, and reuse interpretation) for KBO classes based on the identities of objects contained in a database. With these interpretations of classes as object identities, a database level semantics was presented for KBO schemas in the form of k-compatibility and r-compatibility. Theorems were developed to provide a formal mechanism for the syntactic validation of KBO schemas. In other words, they provide the basis for the design of a syntax parser for validating user-specified class definitions. The definition of KBO databases was also given in this chapter.

Chapter 6 gave the complete definition of simple message-sending. The notion of complex message-sending was also presented and its operational semantics was defined. As a result, complex objects can serve as "sendable" complex objects because, when used as invokers, their data structures actually imply four common forms of behavior composition: AGG objects possess parallel invocability, BIO objects possess conditional invocability, SET objects possess random-order invocability, and SEQ objects possess sequential invocability. In other words, a complex object may imply a complex behavior to certain receivers. Since invokers and receivers are both objects which can be manipulated by end-users using a query facility, the notion of complex message-sending provides the possibility for end-users to associatively select, generate, apply and maintain complex behaviors. The message-sending operation $\rightsquigarrow$ was also extended to Shallow-Send, Deep-Send and N-Level-Send. Unlike the normal

message-sending operation $\rightsquigarrow$ (which treats the receiver as the only receiver), these extended message-sending operations can treat the receiver as a group of sub-receivers when the receiver itself cannot understand the message (the invoker).

Since any newly generated (either navigationally or associatively) complex object is immediately applicable ("sendable") when expressed as the invoker in a message-sending, we are naturally concerned about the safety and success of such complex behaviors. In other words, how can we be sure that such complex behaviors are meaningful complex behaviors? If they are not meaningful complex behaviors, then they can cause serious, unpredictable problems (such as undesirable side-effects) in the database. In order to eliminate such problems, Chapter 7 investigated the safety and success properties of complex message-sendings. The well-formedness of complex message-sendings was defined, and theorems were developed to ensure the well-formedness of complex message-sendings. These theorems provide a foundation for the design of a nontermination detector and a dynamic typing system.

Chapter 8 presented an object algebra, the KBO algebra, which defines a higher-level manipulative semantics of the KBO model, and also serves as a basis for development of other forms of higher-level query languages for KBO databases. The KBO algebra is a polymorphic algebra, in the sense that all its operators can take operands of any kind of KBO objects (not just set objects) and still produce meaningful objects. A unique feature of the KBO algebra is that it takes into account the "sendable" nature of KBO objects. For example, operands of some operators are used in their definitions as qualified invokers in message-sendings. In particular, the Apply operator can send different invokers to different receivers in a set or sequence, based on their relationship specified as the Apply predicate. Algebra expressions can also be freely intermixed with the message-sending operation (on the invoker side or on the receiver side). Therefore, the KBO algebra can be used to produce and maintain (immediately applicable) complex behaviors on a large scale without requiring hardcoding by programmers.

Chapter 9 formally analyzed the properties of the KBO algebra. We showed that the KBO algebra is closed and preserves the expressive power of relational algebra. Theorems were also developed to provide provably correct algebraic transformations of the KBO algebra expressions. They correspond to rewrite rules which can be used by a query optimizer for the algebra. A comprehensive evaluation of the KBO algebra was also conducted. The evaluation shows that the most distinguished features of the KBO algebra are the support for behavior constructors and behaviors-as-objects. In fact, in the KBO model, behavior constructors are exactly the same as data construc-

tors. Therefore, unlike conventional objects in existing OODBs, complex behaviors (or complex invokers) in KBO databases are selectable, constructable, maintainable and immediately invocable, through a database query facility, the KBO object algebra.

Finally, we should point out that the main disadvantage of the KBO model is that compile-time type-checking is difficult to perform on queries; we have to completely rely on dynamic type-checking. This is due to our decision not to use signatures as classes of behavior objects. Our gain is a semantic one because we can now classify behaviors in a semantically related manner, as opposed to syntactically related manner (based on signatures). Moreover, we have three other arguments:

- While type checking is undoubtedly a useful and essential form of data protection, there is no fundamental reason to enforce this checking during compile-time; it can be performed at run-time. In fact, dynamic binding, one major characteristic of object-oriented systems, has already prevented current OODBs from being able to do complete static type-checking.

- Static type-checking mechanisms actually restrict database manipulation to those constructs which are susceptible to static analysis.

- Behaviors themselves could be encoded in a programming language which is already type unsafe (i.e., run-time errors may be caused by a syntactically well-typed expression), such as C and LISP.

We nevertheless believe that, with a carefully built pre-processor, we should be able to mix static and delayed type-checking. The tradeoff here is the allowance for much more, generally less constrained, OODB applications to be undertaken. This is crucially important for those interactive design environments where the users often wish to try out many different ways of manipulation/invocation and explore their effects without permanently committing to them. With a continuous research effort, we expect the increased flexibility and seamless uniformity offered by the KBO model to offset the drawbacks associated with delayed type-checking.

10.2 Thesis Contributions

The primary contribution of this thesis is a formalized single database approach to the management of a single notion of objects that unifies data, arbitrary behaviors, and messages in object-oriented databases. As a result,

- the KBO model has amplified OODB's organization power in that classes can be organized in two non-interfering ways: the KIND-OF lattice based on instance set inclusion and the REUSE-OF lattice based on behavior set inclusion. Similar attempts have been made in some object-oriented programming systems [LTP86, Sny85, Sny86, RWW89], but our approach is new in the context of an object-oriented database system.

- the KBO model has amplified OODB's representation power in that not only hardcoded behaviors can be represented as semantically related objects (based on their annotations), but also complex behaviors can be represented explicitly as complex objects which can invoke the execution of a group of hardcoded behaviors. The significance is that certain simple forms of complex behavioral semantics become directly manipulatable as data by end-users, while still preserving encapsulation of hardcoded behaviors.

- the KBO model has amplified OODB's manipulation power in that behaviors can be directly manipulated by a query facility; in other words, arbitrary (simple or complex) behaviors can be selected, combined, (immediately) applied, and maintained through an object algebra, in an associative (or navigational) manner.

The KBO model is complete, in the sense that it provides not only the descriptive power (the object model) but also the corresponding manipulative power (the object algebra). Several other novel results developed in this thesis include:

1. The formal database semantics presented in Chapter 5 for two separate class lattices (based on k-compatibility and r-compatibility) is the only one known to us in object-oriented data models. This semantics is based on three interpretation functions that interpret KBO classes as three sets of object identities. Theorems were developed for syntactic validation of user-defined KBO schemas so that they conform to the KBO database semantics. Attempts have been made in some programming systems [LTP86, Sny85, Sny86, RWW89] (none of them treats behaviors as objects) to separate the inheritance hierarchy from the subclassing hierarchy, but their techniques are necessarily ad hoc and there is no precise semantics given to the two hierarchies [YO91b]. Consequently, none of them provides formal validation mechanisms for the two separated hierarchies.

2. The (simple) message-sending paradigm presented in Chapter 6 is again the only one known to us in OODBs which provides support for both polymorphism and

monomorphism. This new message-sending paradigm takes advantage of the fact that an object has an identity and a value, and identities have to be unique but values do not have to be unique. Consequently, conventional "overwritten" behaviors can still be invoked whenever polymorphism is not desired, without changing the value of an invoker.

3. The KBO model is the first object-oriented data model that equates its complex objects to complex behaviors. Theorems presented in Chapter 7 show that it is possible to design a safety checker that eliminates those unavoidable nonterminating complex behaviors constructed by the end-user or generated through a query facility.

4. The dynamic typing system presented in Chapter 7 was formally proven to be capable of eliminating those anomalous complex behaviors constructed by the end-user or generated through a query facility. The presence of the typing system should increase the confidence of the users of such complex behaviors.

5. A formal semantics is defined for the KBO algebra in Chapter 8. The novelty of this object algebra is that it takes into account the fact that behaviors or messages can also be operands for the algebra operations. For operators Select and Pickup, their qualification predicates can contain a message-sending where both the invoker and the receiver can be an object variable ranging over a set or sequence operand. For operator Apply, which can take a set/sequence of invokers (the first operand) and a set/sequence of receivers (the second operand), its predicate can impose a condition on the invoker variable and the receiver variable so that different invokers (from the first operand) may be applied to different receivers (from the second operand). To the best of our knowledge, such flexible features are not supported by other existing object algebras or query languages for OODBs.

6. Theorems for rewriting KBO algebra expressions were developed in Chapter 9. In particular, we showed that many of these theorems can be applied to expressions that contain sequence object variables. We believe this is a new result since we are not aware of other object algebras that provide a detailed list of algebraic transformations for expressions operating on sequences.

10.3 Future Research Directions

The research presented here, while it poses a solution to the problem of managing arbitrary behaviors as database objects, suggests many interesting areas for future work. Some of these areas are outlined below; the list is by no means complete.

1. Since the KBO model has been formally designed but not implemented, the implementation of the KBO system is obviously the next logical step in its development. Problems revealed during implementation may motivate the further extension, modification or simplification of the data model.

2. A proper schema definition language is required, which must be capable of managing the various class definitions and their associated behaviors (they h to be created as objects).

3. Algorithms need to be developed for validation of user-defined schemas, detection of nonterminating complex message-sendings, and type checking of complex message-sendings. They seem to be quite straightforward based on the theorems presented in this thesis.

4. The KBO algebra in its present form is quite complicated and not very user-friendly. It can be simplified by restricting its operators only to set objects or sequence objects (which may still contain CAPSULE, AGG, BIO, SET and SEQ objects). An extension to SQL could be designed based on such a simplified algebra.

5. Some useful operations (such as Nest and Unnest) in the nested relational algebras [AB88, Col89] may be modified and then included in the KBO algebra.

6. In the KBO model, type checking on a query expression is accomplished by a dynamic typing system (at run-time) at the message-sending level. There are two special situations where static type checking may be performed (at compile-time) on a query expression: (1) the query expression contains only constant invokers and (2) the query expression is constructed only from those KBO algebra operators whose semantics does not involve message-sendings (such as Union, Intersect, and Subtract). This is because in case (1) we can obtain the signatures of the behaviors to be invoked, and in case (2) we can derive the class (or a set of possible classes) of the result which will be produced by the query (such as in the work of [SO90, Str91]). Therefore, we hope to develop

some static type checking mechanisms for the KBO algebra expressions in these two special cases so that the cost of our dynamic type checking can be reduced by some partial compile-time checking.

7. An interesting area for future work is the development of a viable query optimizer for the KBO algebra. Much work needs to be done to study the semantic transformations of KBO algebra expressions. We can also explore the possibility of letting behaviors provide their own rules specified in a language which can be understood by the query optimizer (such as the optimizer generator in EXODUS [GD87]), so that these rules may be utilized by the query optimizer in its query processing.

8. Completeness of object algebras can also be studied. However, a consensus needs to be reached on completeness of object algebras, against which we could measure our object algebra.

REFERENCES

[AB88] Abiteboul, S. and Beeri, C. On the Power of Languages for the Manipulation of Complex Objects. Technical Report, No. 846.

[AH87a] Abiteboul, S., and Hull, R. IFO: A Formal Semantic Database Model. *ACM Trans. Database Syst. 12*, 4 (Dec. 1987), 525-565.

[AK89] Abiteboul, S., and Kanellakis, P.C. Object Identity as a Query Language Primitive. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, Portland, Oregon, June 1989, 159-173.

[A*88] Albano, A., et al. The Type System of Galileo. In *Data Types and Persistence*, M.P. Atkinson, P. Buneman, and R. Morrison (Eds.), Springer-Verlag, 1988, 101-120.

[A*89] Albano, A., et al. Types for Databases: The Galileo Experience. In *Proc. of the Second International Workshop on Database Programming languages*, Richard Hull, Ron Morrison, and David Stemple (Eds.), 1989, 196-206.

[A*89a] Albano, A., et al. A Framework for Comparing Type Systems for Database Programming Languages. In *Proc. of the Second International Workshop on Database Programming languages*, Richard Hull, Ron Morrison, and David Stemple (Eds.), 1989, 196-206.

[AG89] Agrawal, R. and Gehani, N. H. Design of the Persistence and Query Processing Facilities in O++: The Rational. In *Data Engineering*, Quarterly Bulletin 12, 3 (Sep. 1989), 21-28.

[AH87] Andrews, T. and Harris C. Combining Languages and Database Advances in an Object-Oriented Development Environment. In *Proc. OOPSLA '87*, Oct. 1987, 430-440.

[ASH89] Alashqur, A.M., Su., S.Y.W., and Lam, H. OQL: A Query Language for Manipulating Object-Oriented Databases. In *Proc. Intl. Conf. on Very Large Data Bases*, 1989, 433-442.

[Atk89] Atkinson, M. et al. The Object-Oriented Database System Manifesto. *ALTAIR Technical Report No. 30-89*, GIP ALTAIR, LeChesnay, France, Sept. 1989 (also distributed as the position paper at ACM SIGMOD '90 conference on Management of Data, Atlantic City, N.J., May 1990).

[AL90] America, P., and Linden, F.V.D. A Parallel Object-oriented Language with Inheritance and Subtyping. In *Proc. OOPSLA '90*, Oct. 1990, 161-168.

[AM89] Afsarmanesh, H., and Mcleod, D. The 3DIS: An Extensible Object-Oriented Information Management Environment. *ACM Trans. on Info. Syst. 7*, 4 (Oct. 1989), 339-377.

[Ack82] Ackerman, W.B. Dataflow Languages. In *Computer*, Feb. 1982.

[Agr88] Agrawal, R. Alpha: An Extension of Relational Algebra to Expressa class of Recursive Queries. *IEEE Trans. on Software Eng. 14*, 7 (July 1988) 879-885.

[And90] Andrews, T. The Vbase Object Database Environment. In *Research Foundations in Object-Oriented and Semantic Database Systems*. A. F. Cardenas and D. McLeod (Eds.), Prentice Hall, 1990.

[B*89a] Bretl, R., el at. The GemStone Data Management System. In W. Kim and F.H. Lochovsky, eds., *Object-Oriented Concepts, Databases, and Appilcations* , 283-308.

[B*88] Banerjee, J. et al. Queries in Object-Oriented Databases. In *Proc. 4th Intl. Conf. on Data Engineering*, Los Angeles, Calif. Feb, 1988.

[BBO89] Breazu-Tannen, V., Buneman, P., and Ohori, A. Static Type-Checking in Object-Oriented Databases. In *Data Engineering*, Quarterly Bulletin 12, 3 (Sep. 1989), 5-12.

[BMW84] Borgida, A., Mylopoulos, J., and Wong, H.K.T. Generalization and Specialization as a Basis for Software Specification. In *On Conceptual Modelling*, M.L. Brodie, J. Mylopoulos, and J.W. Schmidt (Eds.), 1984, 87-118.

[BR84] Brodie, M.L., and Ridjannovic, D. On the design and specification of database transactions. In *On Conceptual Modelling*, M.L. Brodie, J. Mylopoulos, and J.W. Schmidt (Eds.), 1984, 277-312.

[BR88] Brodie, M.L., and Ridjannovic, D. Epilogue 1988 (for their paper *On the design and specification of database transactions*). In *Readings in Artificial Intelligence and Databases*, J. Mylopoulos and M. Brodie (eds), 203-204, 1988.

[BR85] Boral, H. and Redfield, S. Database Morphology. In *Proc. Intl. Conf. on VLDB*, Aug. 1985.

[Bee87] Beech, D. Groundwork for an Object Database Model. In Shriver, B. and Wegner, P. (eds.) *Research Directions in Object-Oriented Languages*. MIT Press, 1987, 317-354.

[Bee88] Beech, D. Intensional Concepts in an Object Database Model. In *OOPSLA '88 Proceedings*, Sept. 1988, 164-174.

[Blo87] Bloom, T. Issues in the Design of Object-Oriented Database Programming Languages. In *In Proc. OOPSLA '87*, Oct. 1987, 411-448.

[Bor88] Borgida, A. Class Hierarchy in Information Systems: Sets, Types, or Prototypes?. In *Data Types and Persistence*, M.P. Atkinson, P. Buneman, and R. Morrison (Eds.), Springer-Verlag, 1988, 137-154.

[B*87] Banerjee, J. et al. Data Model Issues for Object- Oriented Applications. *ACM Trans. on Office Info. Syst. 5*, 1 (Jan. 1987), 3-26.

[B*87b] Bancihon, F., et al. FAD, a Powerful and Simple Database Language. In *Proc. VLDB'87*, 97-105.

[B*89] Bancilhon, F., et al. A Query Language for the O_2 Object-Oriented Database System. *Tech Rep. Altair 35-89.*

[BI82] Borning, A.H., and Ingalls, D.H. Multiple Inheritance in Smalltalk-80. In *Proc. of the AAAI '82 Conference*, Pittsburgh, 1982.

[Bee89] Beeri, C. Formal Models for Object Oriented Databases. In *Proc. DOOD'89, Kyoto*, Dec. 1989.

[Bro84] Brodie, M.L. On the Develoment of Data Models. *On Conceptual Modelling*, Michael L. Brodie, John Mylopoulos, and Joachim W. Schimidt (Eds.), Springer-Verlag, 1984.

[C*89a] Cluet, S., Delobel, C., Lecluse, C., and Richard, P. Reloop, an Algebra Based Query Language for an Object-Oriented Database System. *Tech. Report 36-89*, GIP ALTAIR, 1989 (also appeared in DOOD89).

[CS87] Caruso, M. and Sciore, E. The VISION Object-Oriented Database System. In *Proc. Intl. Workshop on Database Programming Languages*, Roscoff France, 1987.

[CS88] Caruso, M. and Sciore, E. Meta-Functions and Contexts in an Object-Oriented Database Language. In *Proc. ACM SIGMOD Conf.*, June 1988, 56-65.

[Car84] Cardelli, L. A Semantics of Multiple Inheritance. In *Semantics of Data Types*, Lecture Notes in Computer Science 173, Springer-Verlag, 1984, 51-67.

[C*89] Canning, P.S., et al. Interfaces for Strong-Typed Object-Oriented Porgramming. In *Proc. OOSPLA '89* , Oct. 1989, 457-467.

[C*89b] Cardelli, L., et al. The Modula-3 Type System. In *ACM Symposium on Principles of Programming Languages*, Jan. 1989, 202-212.

[CDV88] Carey, M. J., DeWitt, D. J., and Vandenberg, S. L. A Data Model and Query Language for EXODUS. In *Proc. ACM SIGMOD 88' Conf.* Chicago, 1988, 413-423.

[CL89] Carey, M.J., and Livny, M. Parallelism and Concurrency Control Performance in Distributed Database Machines. In *Proc. 1989 ACM SIGMOD International Conference on the Management of Data*, Portland, Oregon, 122-133.

[CM88] Cardelli, L., and MacQueen, D. Persistence and Type Abstraction. In *Data Types and Persistence*, M.P. Atkinson, P. Buneman, and R. Morrison, Eds., 31-41, Springer-Verlag, 1988.

[Col89] Colby, L. A Recursive Algebra and Query Optimization for Nested Relations. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, Portland, Oregon, June 1989, 273-283.

[Cox86] Cox, B.J. Object-Oriented Programming: An Evolutionary Approach. Reading MA: Addison-Wesley Publishing Co., 1986.

[D*87] Dayal, U. et al. Simplifying Complex Objects: The PROBE Approach to Modelling and Querying Them. *Technical Report*, Computer Corporation of America, 1987.

[D*89] Dearle, A., et al. Napier88 - A Database Programming Language?. In *Proc. of the Second International Workshop on Database Programming languages*, Richard Hull, Ron Morrison, and David Stemple (Eds.), 1989, 179-195.

[DS85] Dayal, U., and Smith, J.M. PROBE: A Knowledge-Oriented Database Management System. *Proc. Islamorada Workshop Large Scale Knowledge Base and Reasoning Systems*, Islamorada, FL, (Feb. 1985) 103-137.

[Day89] Dayal, U. Queries and Views in an Object-Oriented Data Model. In *Proc. of the Second International Workshop on Database Programming languages*, Richard Hull, Ron Morrison, and David Stemple (Eds.), 1989, 80-102.

[Dat81] Date, C.J. An Introduction to Database Systems. Addison-Wesley, 1981.

[D*89] Dearle, A., et al. Napier88 - A Database Programming Language. In *Proc. of the Second International Workshop on Database Programming languages*, Richard Hull, Ron Morrison, and David Stemple (Eds.), 1989, 179-195.

[DD91] Davis, K.C., and Delcambre, L.M.L. A Denotationl Approach to Object-Oriented Query Language Definition. In *International Workshop on Specification of Database Systems*, Glasgon, Scotland.

[D*85] Dayal, U. et al. PROBE - A Research Project in Knowledge-Oriented Database Systems: Preliminary Analysis. *Technical Report*, Computer Corporation of America, CCA-85-03, July, 1985.

[Daw89] Dawson, J. A Family of Models — Can object-oriented databases be as successful as relational databases?. *BYTE*, Sept. 1989, 277-286.

[Day89] Dayal, U. Queries and Views in an Object-Oriented Data Model. In *Proc. of the Second International Workshop on Database Programming languages*, Richard Hull, Ron Morrison, and David Stemple (Eds.), 1989, 80-102.

[DS85] Dayal, U., and Smith, J.M. PROBE: A Knowledge-Oriented Database Management System. *Proc. Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, Islamorada, FL, (Feb. 1985) 103-137.

[EN89] Elmasri, R. and Navathe, S.B. Fundamentals of Database Systems. The Benjamin/Commings Publishing Company, Inc., 1989.

[Fre87] Freytag, J. A Rule-Based View of Query Optimization. In *Proc. ACM SIGMOD Int'l. Conf. on Managemant of Data*, 173-180, May 1987.

[F*87] Fishman, D.H., et al. IRIS: An Object-Oriented Database Management System. *ACM Trans. Office Info. Syst.* 5, 1 (Jan. 1987), 48-69.

[F*90] Fishman, D.H., et al. Overview of the IRIS DBMS. In *Research Foundations in Object-Oriented and Semantic Database Systems.* A. F. Cardenas and D. McLeod (Eds.), Prentice Hall, 1990.

[FSS89] Fegaras, L., Sheard, T., and Stemple, D. The ADABTPL Type System. In *Proc. of the Second International Workshop on Database Programming languages*, Richard Hull, Ron Morrison, and David Stemple (Eds.), 1989, 207-218.

[GR83] Goldberg, A. and Robson, D. Smalltalk-80: The Language and its Implementation. Addison-Wesley Publishing Company, 1983.

[GSL91] Guo, M., Su, S.Y.W., and Lam, H. An Association Algebra For Processing Object-Oriented Databases. In *Proc. 7th International Conference on Data Engineering*, 23-32, Feb. 1989.

[GD87] Graefe, G. and Dewitt, D. The EXODUS Optimizer Generator. In *Proc. ACM SIGMOD Int'l. Conf. on Managemant of Data*, 160-172, May 1987.

[GM88] Graefe, G. and Maier, D. Query Optimization in Object-Oriented Database Systems: A Prospectus. In K.R. Dittrich (ed), *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems*, 358-362, Springer-Verlag Lecture Notes in Computer Sceince 334, Sept. 1988.

[GZC89] Guting, R. H., Zicari, R., and Choy, D. M. An Algebra for Structured Office Documents. *ACM Trans. on Info. Syst.* 7, 2 (April 1989), 123-157.

[H*89] Hass, L.M. Extensible Query Processing in Starburst. In *Proc. ACM SIGMOD Int'l. Conf. on Managemant of Data*, 377-388, June 1989.

[HI88] Hekmatpour, S. and Ince, D. Software Prototyping, Formal Methods and VDM. *International Computer Science Series,* Addison-Wesley publishing company, 1988.

[HK86] Hudson, S., and King, R. CACTIS: A Database System for Specifying Functional-defined Data. In *Proc. of the Workshop on Object-Oriented Databases*, 1986, 26-37.

[HK89] Hudson, S., and King, R. CACTIS: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System. *ACM Trans. Database Syst. 14*, 3 (September 1989), 291-321.

[HK90] Hull, R. and King, R. A Tutorial on Semantic Database Modeling. In *Research Foundations in Object-Oriented and Semantic Databe e Systems.* A. F. Cardenas and D. McLeod (Eds.), Prentice Hall, 1990.

[Hoa89] Hoare, C.A.R. The Varieties of Programming Language. In G. Goos and J. Hartmanis (eds), *Proc. of Intl. Joint Conf. on Theory and Practice of Software Development*, 1-18, Springer-Verlag Lecture Notes in Computer Sceince 351, March, 1989.

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

[Hsi89] Hsieh, D. Generic Computer Aided Software Engineering (CASE) Databases Requirements. In *Proc. Fifth International Conference on Data Engineering*, 422-424, Feb. 1989.

[HZ87] Hornick, M.F., and Zdonik, S.B. A Shared, Segmented Memory System for an Object-oriented Database. *ACM Trans. Office Info. Syst. 5*, 1 (Jan. 1987), 71-95.

[JV85] Jarke, M. and Vassiliou, Y. A Framework for Choosing a Database Query Langauge. *Computing Surveys 17* , 3 (Sep. 1985), 313-340.

[Jac89] Jacobs, D. A Type System for Algebraic Database Programming Languages. In *Data Engineering*, Quarterly Bulletin 12, 3 (Sep. 1989), 13-20.

[Jon84] Jones, T.C. Reusability in Programming: a survey of the state of the art. *IEEE Trans. on Software Engineering*, Vol. 10, No. 5, Sep. 1984, 488-493.

[K*89] Kim, W., et al. Features of the ORION Object-Oriented Database System. Object-Oriented Concepts, Databases, and Appilcations. W. Kim and F.H. Lochovsky (Eds.), Addison-Wesley Publishing Company, 1989, 251-282.

[K*90] Kim, W., et al. Intefrating an Object-Oriented Porgramming System with a Database System. Research Foundations in Object-Oriented and Semantic Database Systems. A. F. Cardenas and D. McLeod (Eds.), Prentice Hall, 1990, 156-173.

[KG87] Kaiser, G.E., and Garlan, D. Melding Software Systems from Reusable Building Blocks. In *IEEE Software,* July 1987, 17-24.

[KM84] King, R., and McLeod, D. A Unified Model and Methodology for Conceptual Database Design. In *On Conceptual Modelling*, M.L. Brodie, J. Mylopoulos, and J.W. Schmidt (Eds.), 1984, 313-332.

[KM85] King, R., and McLeod, D. Semantic Database Models. In *Database Design*, S. B. Yao, Ed., Prentice Hall, Englewood Cliffs, N.J., 1985.

[Kim89] Kim, W. A Model of Queries For Object-Oriented Databases. In *Proc. Intl. Conf. on Very Large Data Bases*, 1989, 423-432.

[Kim90] Kim, W. Object-Oriented Databases: Definition and Research Directions. In *IEEE Trans. on Knowledge and Data Engineering*, Vol. 2, No. 3, Sept 1990, pp. 327-341.

[Kor88] Korth, H.F. Optimization of Object-Retrieval Queries. In K.R. Dittrich (ed), *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems*, 353-357, Springer-Verlag Lecture Notes in Computer Sceince 334, Sept. 1988.

[KC86] Khoshafian, S.N., and Copeland, G.P. Object Identity. In *Proc. OOPSLA '86*, Sept. 1986, 406-416.

[LTP86] LaLonde, W.R., Thomas, D.A., and Pugh, J.R. An Exemplar Based Smalltalk. In *Proc. of OOPSLA '86 Conference*, Sep. 1986, 322-230.

[LR88] Lecluse, C., and Richard, P. Modeling Inheritance and Genericity in Object-Oriented Databases. In *Proc. ICDT'88, 2nd Intl. Conf. on Database Theory*, Bruges, Belgium, August 1988, 223-238.

[LRV88] Lecluse, C., Richard, P., and Velez, F. O2, an Object-Oriented Data Model. In *Proc. 1988 ACM-SIGMOD Conference*, 424-433.

[LP89] Lecluse, C., and Richard, P. The O2 Database Programming Language. In *Proc. Intl. Conf. on Very Large Data Bases*, 1989, 411-422.

[LOC87] Lochovsky, F.H. Editorial: Introduction to the Special Issue on Object-Oriented Systems. *ACM Trans. Office Info. Syst.* 5, 1 (Jan. 1987), 1-2.

[Lie86] Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proc. of OOPSLA '86 Conference*, Sep. 1986, 214-223.

[MBW80] Mylopoulos, J., Bernstein, P.A., and Wong, H.K.T. A Language Facility for Designing Database-Intensive Applications. ACM Trans. on Database Syst. Vol. 5, No. ?, June 1980, 185-207.

[MMM90] Madsen, O.L., Magnusson, B, and Moller-Pederson, Birger. Strong Typing of Object-Oriented Languages Revisited. In *Proc. OOPSLA '90*, Oct. 1990, 140-149.

[MW88] Moss, J.E.B. and Wolf, A.L. Towards Principles of Inheritance and Subtyping in Programming Languages. *Tech. Rep. 88-95*, Dept. of Computer and Information Science, Univ. of Massachusetts, Nov. 1988.

[Mey86] Meyer, B. Genericity versus Inheritance. In *Proc. 1986 Conference on Object-Oriented Programming Systems, Languages, and Applications*, 391-405, Spet. 1986.

[Mit84] Mitchell, J. Type Inference and Type Containment. In G. Goos and J. Hartmanis (eds), *Proc. of Intl. Symposium on Semantics of Data Types*, 257-277, Springer-Verlag Lecture Notes in Computer Sceince 173, June, 1984.

[Mit89] Mitschang, B. Extending the Relational Algebra to Capture Complex Objects. In *Proc. Intl. Conf. on Very Large Data Bases*, 1989, 297-305.

[Moo86] Moon, D.A. Object-oriented Programming with Flavors. In *Proc. OOPSLA '86 Conf.*, Sep. 1986, 1-8.

[Moo89] Moon, D.A. The COMMON LISP Object-Oriented Programming Language Standard. In W. Kim and F.H. Lochovsky, eds., *Object-Oriented Concepts, Databases, and Appilcations* , 1989, 49-78.

[MD86] Manola, F. and Dayal, U. PDM: an object-oriented data model. In *Porc. 1986 International Workshop on Object-Oriented Database Systems*, 18-25, Sept 1986, 18-25.

[MP84] Maier, D. and Price, D. Data Model Requirements for Engineering Applications. In *IEEE 1st Intl. Workshop on Expert Database Systems*, 1984, 759-765.

[MS89] Matthes, F., and Schmidt, J.W. The Type System of DBPL. In *Proc. of the Second International Workshop on Database Programming languages*, Richard Hull, Ron Morrison, and David Stemple (Eds.), 1989, 219-225.

[Mai86] Maier, D., et al. Development of an object-oriented DBMS. In *Proc. 1986 Conference on Object-Oriented Programming Systems, Languages, and Applications,* 472-482, Spet. 1986.

[NR89] Nguyen, G.T. and Rieu, D. Schema Evolution in Object-Oriented Database Systems. In *Data & Knowledge Engineering*. 4 (1989), 43-67.

[NS88] Neuhold, E. and Stonebraker, M. Future Directions in DBMS Research. Technical Report 88-001, Intl. Computer Science Inst. Berkeley, California, May 1988.

[OBB89] Ohori, A., Buneman, P., and Breazu-Tannen. Database programming in Machiavelli – a Polymorphic Language with Static Type Inference. In *Proc. ACM SIGMOD conf.*, May-June 1989, 46-57.

[OH86] Osborn, S.L., and Heaven, T.E. The Design of a Relational Database System with Abstract Data Types for Domains. *ACM Trans. Database Syst. 11*, 3 (Sept. 1986), 357-373.

[OOM87] Ozsoyoglu, G., Ozsoyoglu, Z.M., and Matos, V. Extended Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions. *ACM Trans. Database Syst. 12*, 4 (Dec. 1987), 566-592.

[OSC89] Object-Science Corporation. Finally the true definition of the object-oriented database system is here ... . In *Journal of Object-Oriented Programming*, Vol. 2, No. 4, Dec. 1989, 7-7.

[Osb88] Osborn, S.L. Indentity, equality and query optimization. In K.R. Dittrich (ed), *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems*, 346-351, Springer-Verlag Lecture Notes in Computer Sceince 334, Sept. 1988.

[Osb89a] Osborn, S.L. Algebraic Query Optimization for an Object Algebra. Technical Report No. 251, University of Western Ontario, 198[illegible].

[Osb89b] Osborn, S.L. The Role of Polymorphism in Schema Evolution in an Object-Oriented Database. In *IEEE Trans. on Knowledge and Data Engineering*, Sept. 1989.

[OGM86] Orenstein, J.A., Goldhirsch, D., and Manola, F.A. The Architecture of the PROBE Database System. In *Probe Project Workshop*, 1986.

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

[PS87] Penney, D.J. and Stein, J. Class Modification in the GemStone object-oriented DBMS. In *Proc. OOPSLA '87 Conf.*, Sep. 1987.

[Pas89] Paseman, B. Object Oriented Database Panel Position Statement. In *Proc. Fifth International Conference on Data Engineering*, 419-421, Feb. 1989.

[RS87] Rowe, L. A. and Stonebraker, M. R. The POSTGRES Data Model In *Proc. 87' VLDB Conference*, 83-96, 1987.

[RS89] Rozen, S., and Shasha, D. Using A Relational System On Wall Street: The Good, The Bad, The Ugly, And The Ideal. *ACM Comm. Vol. 32,* No. 8, August 1989, 988-994.

[RWW89] Rosenblatt, W.R., Wileden, J.C., and Wolf, A.L. OROS: Towards a Type Model for Software Environments. In *Proc. OOPSLA '89*, Oct. 1989, 297-304.

[S*86] Schaffert, C., et al. An Introduction to Trellis/Owl. In *Proc. OOPSLA '86*, Sept. 1986, 9-16.

[SH85] Studer, R., and Horndasch, A. Modeling Static and Dynamic Aspects of Information Systems. In *Database Semantics*, E.T. Steel, Jr. and R. Meersman (Eds.), North-Holland, 1985, 13-26.

[SLU89] Stein, L.A., Lieberman, H., and Ungar, D. A Shared View of Sharing: The Treaty of Orlando. In W. Kim and F.H. Lochovsky, eds., *Object-Oriented Concepts, Databases, and Appilcations* , 1989, 31-48.

[SO90a] Straube, D.D., and Ozsu, M.T. Type Consistency of Queries in an Object-Oriented Database System. In *Proc. OOPSLA & ECOOP '90*, Oct. 1990, 224-233.

[SR86] Stonebrake, M.R., and Rowe, L.A. The Design of POSTGRES. In *Proc. of the ACM-SIGMOD Conf.*, Washington DC, 1986, 340-355.

[SS77a] Smith, J.M., and Smith, D.C.P. Database abstractions: aggregation and genelization. ACM Trans. on Database Syst. 2, 2 (June 1977), 105-133.

[SS77b] Smith, J.M., and Smith, D.C.P. Database abstractions: aggregation. Comm. ACM 20, 6 (June 1977), 405-413.

[SS78] Smith, J.M., and Smith, D.C.P. Principles of Database Conceptual Design. In Proc of the NYU Symposium on Database Design (May 1978), 35-49.

[Shi86] Shipman, D.W. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst.* 6, 1 (Mar. 1981), 140-173.

[Sny85] Snyder, A. Object-oriented Programming for Common Lisp. *Tech. Rep. ATC-85-1*, Hewlett-Pachard Company, July 1985.

[Sny86] Snyder, A. Encapsulation and Inheritance in Object-Oriented Programming Langauges. In *Proc. of the OOPSLA'86 Conference*, Sep. 1986, 38-45.

[St*90] Stein, J. (ServioLogic), Andrews, T. (Ontologic), Kent, B. (Hewlett-Packard), Rotzell, K. (Versant Object Technology), and Weinreb, D. (Object Design). PANEL: Issues in Object Database Management. In *Proc. of the OOPSLA & ECOOP '90 Conference*, Ottawa, Canada, Oct. 1990, 235-236.

[Sto90] Stonebraker, M., et al. Third-Generation Data Base System Manifesto. *Memorandum No. UCB/ERL M90/28*, the Committee for Advanced DBMS Function, College of Engineering, University of California, Berkeley, April 1990 (also distributed as the position paper at ACM SIGMOD '90 conference on Management of Data, Atlantic City, N.J., May 1990).

[Su83] Su, S.Y.W. SAM*: A Semantic Association Model for Corporate and Scientific Statistical Databases. *Inf. Sci.* 29, 1983, 151-199.

[SB86] Stefik, M. and Bobrow, D.G. Object-oriented Programming: themes and variations. *AI Magazine*, Vol. 6, No. 4, 1986, 40-62.

[SO90] Straube, D.D., and Ozsu, M.T. Queries and Query Processing in Object-Oriented Database Systems. *Trans. on Info. Syst. 8*, 4, October 1990.

[SZ89] Shaw, G.M., and Zdonik, S.B. An Object-Oriented Query Algebra. *Bulletin of IEEE technical committee on Data Engineering* 12, 3 (Sep. 89), 29-36.

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

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

[Str86] Stroustrp, B. The C++ Programming Langauge. Addison-Wesley Publishing Company. 1886.

[thi90] *THINK C User's Manual*, 1990.

[Ull87] Ullman, J.D. Database Theory - Past and Future. In *Proc. of the PODS Conference*, March 1987

[Ull88] Ullman, J.D. Principles of Database and Knowledge-Base Systems (Vol. 1). Computer Science Press, 1988.

[VD90] Vandenberg, S.L., and DeWitt, D.L. An Algebra for Complex Objects with Arrays and Identity. Tech. Report #918, Computer Science Dept., University of Wisconsin-Madison, March 1990.

[VD91] Vandenberg, S.L., and DeWitt, D.L. Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. In *Proc. ACM SIGMOD91*, 158-167.

[WW89] Wirfs-Brook, R. and Wilkerson, B. Object-Oriented Design: A Responsibility-Driven Approach. In *Proc. OOPSLA'89*, Oct. 1989, 71-75.

[Weg87] Wegner, P. Dimensions of Object-Based Language Design. In *Proc. of the OOPSLA '87 Conference*, Oct. 1987, 168-182.

[Wil90] Wilkerson, B. How to design an object-based application. In *Develop*, the Apple Technical Journal, Issue 2, April 1990, 178-203.

[WZ90] Wegner, P. and Zdonik, S. Models of Inheritance. In *Proc. 2nd Intl. Workshop on Database Programming Languages*, (eds. Hull, R., Morrison, R., and Stemple, D.), Morgan Kaufmann, San Mateo, CA, 248-255.

[Weg87] Wegner, P. Dimensions of Object-Based Language Design. In *Proc. of the OOPSLA '87 Conference*, Oct. 1987, 168-182.

[YO91a] Yu, L., and Osborn, S.L. An Evaluation Framework for Algebraic Object-Oriented Query Models. In *Proceedings of the 7th IEEE International Conference on Data Engineering*, Kobe, Japan. April 8-12, 1991, pp. 670-677.

[YO91b] Yu, L., and Osborn, S.L. Context Inheritance and Content Inheritance in an Object-Oriented Data Model. In F. Dehne, F. Fiala, and W.W. Koczkodaj (Eds.), *Advances in Computing and Information — ICCI'91 International Conference in Computing and Information*, Ottawa, Canada, May 1991, Springer-Verlag Lecture Notes in Computer Science 497, pp. 240-251.

[ZW88] Zdonik, S.B., and Wegner, P. Language and Methodology for Object-Oriented Database Environments. In *Data Types and Persistence*, M.P. Atkinson, P. Buneman, and R. Morrison (Eds.), Springer-Verlag, 1988, 155-171.

[Zdo88] Zdonik, S.B. Data Abstraction and Query Optimization. In K.R. Dittrich (ed), *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems*, 368-373, Springer-Verlag Lecture Notes in Computer Sceince 334, Sept. 1988.

[Zdo89] Zdonik, S.B. Query Optimization in Object-Oriented Databases. In *Proc. 22nd Annual Hawaii Intl. Conf. on System Sciences*, ed. Bruce Shriver, 1989, 19-25.

[ZM88] Zhu, J. and Maier, D. Abstract Object in an Object-Oriented Data Model In *Proc. 2nd Intl. Conf. on Expert Database Systems*, Larry Kerschberg (ed.), 1988, 73-105.