A QUERY MODEL AND AN OBJECT ALGEBRA FOR OBJECT-ORIENTED DATABASES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Reda ALHAJJ
February, 1993

A QUERY MODEL AND AN OBJECT ALGEBRA FOR OBJECT-ORIENTED DATABASES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF
COMPUTER ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BİLKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Reda ALHAJJ
February 1993

Reda Alhajj
tarafından [illegible]

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. M.Erol ARKUN (Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. Asuman DOĞAÇ

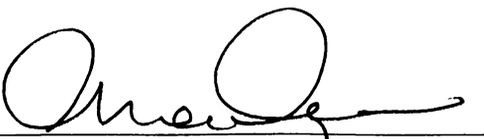
I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. Akif EYLER

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Assist. Prof. Dr. Kemal OFLAZER

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.



Dr. Aral EGE

Approved for the Institute of Engineering and Science:

Mehmet BARAY, Ph. D.
Director of Institute of Engineering and Science

Abstract

A QUERY MODEL AND AN OBJECT ALGEBRA FOR OBJECT-ORIENTED DATABASES

Reda ALHAJJ

Ph. D. in Computer Engineering and Information Science

Supervisor: M.Erol ARKUN, Ph. D.

February 1993

A query model is an important component of any database system. In this sense, the relational model has a well defined underlying query model. On the other hand, a well defined query model for object-oriented databases has not been accepted yet. This is one of the common complaints against object-oriented databases. So defining a formal object algebra is one of the most challenging steps in developing a theory for object-oriented databases. In object-oriented data models, although messages serve to manipulate the database, a query model is still required to effectively deal with more complex situations and to facilitate associative access. In this thesis, a query model for object-oriented databases is described, where both the structure and the behavior of objects are handled. Not only the manipulation of existing objects, but also the creation of new objects and the introduction of new relationships are supported in the model. Equivalents to the five basic operations of the relational model as well as other additional operations such as one level project, nest and aggregate function application are defined. Hence, the proposed object algebra subsumes the relational algebra. Linear recursion is also supported without requiring any additional operator to serve the purpose. Both the operands as well as the results of these operations are characterized as having a pair of sets -a set of objects and a set of message expressions (sequences of messages) applicable to them. The closure property is shown to be preserved in a natural way by the results of operations possessing the same characteristics as the operands in a

query. It is shown that every class possesses the properties of an operand by defining a set of objects and deriving a set of message expressions for it. Furthermore, it is shown that the output of a query has the characteristics of a class. Thus, it is also shown how the super/subclass relationships of the result of a query with its operands can be established and how the result can be placed persistently in the lattice (schema) as a class. Such a class is naturally and properly placed in the lattice by maximizing reusability due to inheritance. Also equivalent object algebra expressions are presented and the associativity of the cross-product operation which is an important property in query optimization is proved. Lastly, as it was recognized that schema evolution is an important requirement to be satisfied by object-oriented databases, hence the handling of schema evolution functions through the proposed object algebra operations is also developed as another contribution of the thesis.

Keywords: *database system, object-oriented data model, object-oriented database management system, object-oriented query model, object-oriented query language, object algebra, reusability, recursive queries, transitive closure, query optimization, schema evolution, schema modification.*

Özet

NESNESEL VERİ TABANLARI İÇİN
SORGULAMA MODELİ VE NESNESEL CEBİR

Reda ALHAJJ

Bilgisayar ve Enformatik Mühendisliği Doktora

Tez Yöneticisi: Prof. Dr. M.Erol ARKUN

Subat 1993

Sorgulama modeli, herhangi bir veri tabanının en önemli kısmıdır. Bu bağlamda, ilişkisel model, çok iyi tanımlanmış bir sorgulama modeline sahiptir. Buna karşılık nesnesel veri tabanları için iyi tanımlanmış bir sorgulama modeli henüz kabul edilmemiştir. Bu, nesnesel veri tabanlarına karşı getirilen en önemli eleştiridir. Böylece, formal bir nesnesel cebir tanımlanması, genel nesnesel veri tabanı teorisinin geliştirilmesinde en önemli basamaklardan biridir. Nesnesel veri tabanlarında, mesajlar veri tabanını kullanmaya olanak tanımalarına ragmen, hala karmaşık işlemlerin kolayca yapılabilmesi ve içerikle erişimin gerçekleştirilmesi için bir sorgulama modeline ihtiyaç vardır. Bu tez çalışmasında, nesnesel veri tabanları için nesnelerin davranışlarına ek olarak, yapılarının da gözönüne alındığı bir sorgulama modeli tanımlanmaktadır. Bu modelde, sadece hali hazırdaki nesnelerin işlenmesi değil, aynı zamanda yeni nesne ve bağıntıların yaratılması desteklenmiştir. İlişkisel modelin beş temel işlemine eşdeğer işlemlerin yanı sıra, ek olarak tek düzey izdüşüm, yuvalama ve bütünleme fonksiyonları tanımlanmıştır. Böylece, önerilen nesnesel cebir, ilişkisel cebiri kapsamaktadır. Aynı zamanda, doğrusal özyineleme, hiç bir ek işlev gerektirmeksizin tanımlanmıştır. Hem işleçler hem de buna ek olarak işlevlerin sonuçları kümeler çifti –nesneler kümesi ve bunlara uygulanabilen mesaj terimleri kümesi (mesaj dizileri) olarak karakterize edilirler. Kapalı olma özelliğinin, işlemlerin sonuçlarının bir sorgudaki işleçler gibi aynı karakteristiğe sahip olması nedeni ile doğal olarak korunduğu gösterilmektedir.

Her sınıfın, bir nesne kümesi tanımlaması ve bunun için bir mesaj terim kümesi türetilmesiyle, bir işlecin özelliklelerine sahip olduğu gösterilmektedir. Bununla birlikte, bir sorgunun çıktısının bir sınıfın niteliklerine sahip olduğu gösterilmektedir. Ayrıca, bir sorgu çıktısı ile icsleçlerinin arasındaki alt/üst sınıf ilişkisinin nasıl oluştuğu ve sonucun şema yapısında kalıcı bir sınıf olarak nasıl yerleştirileceği gösterilmektedir. Böyle bir sınıf, tekrar kullanılabilirliği kalıtım vasıtası ile maksimuma ulaştıracak şekilde doğal ve uygun olarak şemada saklanabilmektedir. Ayrıca nesnesel cebir işlevlerinin eşdeğerleri tanımlanarak sorgu optimizasyonunda önemli bir özellik olan Cartesian-Çarpım işleminin birleşme özelliğinin doğruluğu kanıtlanmaktadır. Son olarak, şema evriminin, nesnesel veri tabanlarınca sağlanması gereken bir özellik olduğu anımsanırsa, önerilen nesnesel cebir işlemleri vasıtası ile şema evriminin sağlanması tezin bir başka katkısı olarak geliştirilmiştir.

Anahtar sözcükler: veri tabanı sistemi, nesnesel veri modeli, nesnesel veri tabanı yönetimi sistemi, nesnesel sorgu modeli, nesnesel sorgu dili, nesne cebiri, tekrar kullanılabilirlik, yuvalanmış sorgu, şema evrimi, şema uyarlaması.

Acknowledgement

I have spent two exciting years at *Bilkent University* developing the research ideas and results which are presented in this thesis. During that time, many people directly or indirectly contributed to the successful completion of this thesis.

It is a pleasure to acknowledge the help and support I have received mainly from Prof.Dr. M.Erol ARKUN in the efforts that have culminated in this thesis. I have honored to be supervised by him. The presentation of my ideas was always improved by his helpful comments. His personality influenced my attitude as a researcher.

I have special debts to acknowledge. Prof. Asuman DOĞAÇ always stimulated me by giving the impression that I'm doing original work. Her support motivated me to be creative and contribute to the subject. Prof. Gültekin OZSOYOĞLU led my research in new directions that sound more relevant. Thanks are also extended to the referees of our papers accepted to various *journals* and *conferences* for investing their time carefully reading our papers providing comments that improved our research.

No work of this magnitude can see the light of day without a source of encouragement and motivation. In my case, that source has been and is my family and in particular my mother. She used to guide and push me towards this target. This thesis is dictated to the fond memory of her.

Contents

List of Figures

Chapter 1

Introduction

1.1 The Motivation: An Overview

Database systems in their conventional sense proved to be non-appropriate for and fell short in meeting the requirements coming mainly from engineering and information based applications including AI, CAD/CAM and OIS. Consequently, it was recognized that the relational model which could efficiently handle conventional business applications should undergo certain improvements to be adapted to the new applications. In other words, although the relational data model is suitable to handle conventional business applications, the first normal form restriction led to extensions to satisfy new application areas. Early extensions relaxed the first normal form by allowing set-valued attributes. Still a more advanced extension is based on complex objects where sets and tuples are arbitrarily nested [2, 44, 46, 57, 81]. To satisfy object sharing within complex objects, object identity was introduced. A more advanced step towards satisfying recent application requirements was the development of object-oriented systems. Object-oriented systems evolved to satisfy the demand for a more appropriate representation and modeling of real world entities. Such a demand comes mainly from data intensive applications including CAD/CAM, OIS and AI. To satisfy the requirements of such applications, it was recognized that an integration of object-oriented concepts [54, 93] with the database technology [46] leads to more appropriate representation methods and many object-oriented data models have thus been developed [28, 36, 40, 48, 52, 56, 72]. But, still there is no agreement on standardization within the realm of object-orientation. Neither the boundaries

for the query model have been set up nor an object-oriented query language has yet been formally defined. This is one of the common complaints against object-oriented databases [76, 79]. However, it is agreed that object-oriented databases are more powerful than conventional databases at the modeling phase. An object-oriented model is more powerful than the relational model at both the modeling and the manipulation phases. It is more powerful at the modeling phase due to the features of inheritance, encapsulation, identity and complex objects. It is more powerful at the manipulation phase due to messages that handle both stored and derived values which result in full computational power. We argue that this superiority should be maintained as far as the query model is concerned. This thesis is focused in this direction where the shortcomings of the proposed object-oriented query models have been identified in order to try to overcome them.

One basic object-oriented concept is that every entity of the real world be represented by an object that captures both the state and behavior of the entity. An **object** has an identity to distinguish it from other objects in the database. Objects that have the same behavior and state structure are grouped in a **class**. A **class** is allowed to reuse (**inherit**) **state structure** and **behavior** defined for other classes. The inheritance mechanism leads to a **hierarchy** or a **lattice**.

Comparing the relational model and an object-oriented model shows that the latter is more powerful at the modeling stage, but yet does not support a standard formal query model. While the non-atomic domain concept is supported by the nested relational model [2, 44, 57, 81], we see inheritance, identity and encapsulation among the features that the relational model lacks. Identity provides for object sharing and object independence [58]. Inheritance provides for structure and behavior sharing. Encapsulation provides for abstraction. Furthermore, in the nested relational model, if a relation r has an attribute with domain relation r_1, then r_1 can not have any attribute with domain r, however, such domain restrictions have been relaxed in object-oriented models. As a result, an object-oriented query model should benefit from such features and hence should be at least as powerful as the relational query model.

Query algebras are of interest for two main reasons. First, they provide an abstract language in which to reason about the meanings and the expressiveness of queries coded in user query languages. Second, query algebras have great practical utility in query optimization- a user query once translated into an algebraic expression can in many

cases be transformed through algebraic identities into an equivalent expression which can be evaluated much more rapidly. Algebraic query optimization is an established technique in the implementation of relational databases [71] and should be adapted for object-oriented databases.

A general powerful characteristic of object-oriented query languages is that messages substitute most queries in conventional databases. For instance, the message **name()** when sent to an instance in the *student* class, the name of the particular student is returned. While a single message is sufficient for such an operation in the object-oriented context, a selection and a projection are necessary to get the same result in the relational model. An additional join should precede when name is not an attribute of the student relation. Another example can be seen in sending the message **courses()** to a student and the message **grade()** to the result of the first message. Although it is handled due to the implicit join [61] present in object-oriented models, this corresponds to an explicit join in the relational model. The two messages **courses()** and **grade()** form a message expression. In general, a message expression is defined to be a sequence of messages $m_1...m_n$, with $n \geq 1$. However, message expressions do not thoroughly substitute the query language requirement. Rather it is widely accepted that a query language must be a part of any database system. In other words, while simple message expressions give superiority to object-oriented systems over the relational model, an *ad hoc* object-oriented query language is still needed for more complex situations and to support associative access. In other words, although the modeling power of an object-oriented database supports implicit joins [61] by allowing instances in a class to form the domain for an instance variable in another class, an explicit join is necessary in introducing new relationships into the model; otherwise the manipulative power of the model will be restricted. Allowing an explicit join raises the problem of maintaining the closure property. Therefore, it is necessary to have an object algebra that facilitates the introduction of new relationships while maintaining the closure property; otherwise the relational model will be more powerful. The requirement for deriving new objects in terms of existing others, and for the introduction of new relationships does exist; either for output purposes or for further processing as in knowledge-base systems where the knowledge is acquired by forming new relationships among existing facts.

Concerning the closure property, we say that the set of natural numbers N is closed with respect to addition and multiplication but not with respect to subtraction or

division. That is, $\forall x, y \in N$, $(x + y) \in N$ and $(x \times y) \in N$, but it is not guaranteed that the difference of any two elements of N to be an element of N, i.e., $\forall x, y \in N$, $x < y \Rightarrow (x - y) < 0$, not an element of N. When applying the same concept to the relational model, elements of the relational model are relations and the allowed operations are those of the relational algebra [16]. The relational model satisfies the closure property with respect to the relational algebra operations and the result of any operation is a relation. Concerning object-oriented models, for the closure property to be satisfied, it should be possible to use the result of a query operation allowed in any such model as an operand.

Our approach is to maintain the closure property without violating object-oriented properties. An object-oriented model should be more powerful than the relational model at both the modeling and the manipulation phases. It is more powerful at the modeling phase due to the features of inheritance, encapsulation, identity and complex objects. It is more powerful at the manipulation phase due to the handling of both stored and derived values which result in a full computational power without any need to have an embedded query language leading to impedance mismatch [27]. The impedance mismatch problem occurs when a set oriented query language and a programming language with different computational paradigms are used together.

1.2 Scope and Contributions

In this thesis, we describe a query model for object-oriented data models. Different parts of this thesis has already been published in various journals and conference proceedings [12, 13, 14, 15, 16, 22, 20, 21]. Our object algebra is a superset of the relational algebra, but with different semantics and operands. Also, as the schema of an object-oriented data model may contain cycles (by having the domain of an instance variable being objects in a class) and due to the growing interest in recursive queries, we handle recursive queries as they are of great interest to the application areas of object-oriented databases, e.g., CAD/CAM and Software Engineering applications, which are modeled in terms of recursive definitions. Therefore, even if recursive queries are not special to object-oriented query languages only, query languages supporting advanced applications must include some form of recursion. Although only linear recursion is considered in our work, this includes an important set of recursive queries since it was

recognized that recursive queries encountered in real cases are linear in nature. Linear recursive queries are the ones in which only one appearance of the recursive predicate is allowed on the right hand side of a Horn clause [4]. Furthermore, efficient processing strategies have been defined to handle linear recursion [25, 70].

The main idea in our work is that an operator should equally handle structure as well as behavior of objects. So, an operand in our object algebra, as well as the output of any of the operations, has a pair of sets; a set of objects and a set of message expressions. The set of objects includes all objects that qualify to be in a class and in all of its direct and indirect subclasses; hence the set of objects is in general heterogeneous. The set of message expressions includes message expressions applicable to objects in the other set of the pair. By using such pairs as operands and in the output, the closure property is maintained in a natural and consistent way. Furthermore, a message expression leads to the invocation of behavior and acts as a behavior constructor because it leads to the execution of methods underlying the constituting messages in sequence as if they all form a single method invoked by the message expression.

The operators of our object algebra are the five basic operators of the relational algebra in addition to nest, one level project, aggregate function application and the support of recursion. The nest operator serves to establish additional relationships that are not a priori defined in the model. It is an explicit join that serves the place of a missing implicit join. It is equivalent to the cross-product operation under certain conditions. One level project outputs the result of the evaluation of a set of message expressions against objects of the operand. The aim of this operator is to reduce the depth of nesting. So we have two different projection operators, the relational like projection operator does not evaluate any message expression but just serves to eliminate some parts of the structure. A recursive query is coded by allowing an object variable bound to a resulting object in the evaluation of a query to also appear in a predicate in that query. Illustrative examples on recursive queries are given in section 4.5 of chapter 4. By using the operators of the algebra described in this thesis, we will be able to manipulate existing objects and establish new relationships and hence new objects.

We define a set of total instances for a class c, denoted as $T_{instances}(c)$, as the union of its instances with all the instances of its subclasses. Also a set of message expressions for a class can be derived starting from the set of messages used to invoke its methods.

Therefore, a class having a set of objects and a set of message expressions, can be an operand in a query. Furthermore, it is possible to derive the characteristics of a class from any pair of a set of objects and a set of message expressions [12, 13].

Using the object algebra operators, we build object algebra expressions and show that every object algebra expression has the characteristics of a class. Moreover, we derive the inheritance (sub/superclass) relationship between the result of an object algebra expression and the operand(s). Therefore, the result of any object algebra expression can be persistently and properly placed in the lattice in a natural way.

To sum up, the contributions of our work described in this thesis can be enumerated as follows:

- Operands and the result of a query are defined in a way not to violate object-oriented constructs and to maintain the closure property.
- Behavior is also uniformly handled like the structure of objects; creation of methods as well as objects in terms of other existing ones is facilitated.
- The addition of new classes is facilitated where we specify the characteristics of a class derived in terms of existing ones and handle its proper placement in the lattice.
- Aggregation functions are supported in a consistent way so that the result could be used as an operand.
- Recursive queries are handled without any need to have a PROLOG-like query language.
- Computational completeness is maintained without any need to have an embedded query language- as embedded query language leads to the impedance mismatch problem.

All of these are satisfied without loss of generality and formality in the description.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows.

The related work is discussed in chapter 2. It is observed that two kinds of query models could be identified. These are object preserving query models and query models allowing the introduction of new objects. The latter are considered more expressive and powerful compared to the former. Query languages from these two trends are identified with their characteristics and drawbacks pointed out.

In chapter 3, the data model is described where the basic terminology used in the formalization is introduced. We give the characteristics of a class and later in chapter 4 we derive the same characteristics for the result of an object algebra expression. We define a set of objects and a set of message expressions as the constituents of pairs forming operand(s) and the output of a query. The inheritance relationship is defined to be a partial ordering among classes and used later in chapter 4 to derive the relationship between operand(s) and the result of an object algebra expression. Total instances and message expressions are emphasized for being the basic constructs in the query model. Different aspects of the data model are clarified via examples.

The query model is described in chapter 4. An informal description of the object algebra operators leads the formal definition of object algebra expressions (query expressions). Then the characteristics of an object algebra expression are determined to be the same as those of a class. The proper placement of such a class in the lattice is also considered. After that, we emphasize on maximizing reusability due to inheritance before giving illustrative examples and elaborate on linear recursion. Finally, equivalent object algebra expressions are identified where the associativity of the cross-product is proved forming a basis for any future work concerning query optimization.

As object-oriented databases have proved suitable for applications where the information about the domain is incomplete or become incrementally available, schema modification is considered important within the realm of object-oriented systems. In chapter 5 we describe how different schema evolution functions could be handled using the object algebra operators. Also, the invariants and the rules judging the correctness of schema evolution functions are enumerated.

Chapter 6 is the conclusions. After identifying the major conclusions drawn from the thesis, the basic contributions due to the described work are enumerated. Finally, possible research areas based on this thesis are summarized.

Chapter 2

Related Work

Several query models are described in the literature aiming at providing the accessing facilities for particular object-oriented database systems. In this chapter a description of such a query model adapted from the literature is included. In section 2.1, an overview of the query models is presented. We differentiate between object preserving and object creating query models. A critique of those query languages is given in section 2.2 where the major characteristics of such languages are emphasized and their pros and cons are identified; thus, justifying the motivation for the development of the query model described in this thesis.

2.1 An Overview

A query language must be a component of any database system [95]. Consequently, W.Kim identifies a query language among the requirements of object-oriented systems despite the use of messages to manipulate the database [62]. Several query languages such as those of GemStone [37, 72], O_2 [26, 42, 49], EXODUS [41, 101], IRIS [52], ORION [29, 61, 64], OSAM* [5, 6], Postgres [82, 96], PDM [47, 74], ENCORE [86, 105] and the formal calculi and algebra developed by Straube and T. Özsu [97, 98] in addition to others [7, 24, 34, 35, 53, 59, 65, 66, 77, 78, 83, 84, 89, 104] have been proposed. These languages have been developed based on different paradigms. Some query language are based on the functional paradigm [47, 74], while others [29, 61] are based on the message-passing paradigm. Other languages are based on extensions to the relational paradigm: such as extensions to QUEL [41, 82] and extensions to SQL [42]. The query

language of IRIS [52] is based on both the functional and the relational paradigms where functions are used in Object-oriented SQL (OSQL) constructs. OSQL is embedded inside Common LISP via macro extensions, hence it does not overcome the impedance mismatch problem.

These languages are classified as either object-preserving [5, 29, 41, 72, 97, 98] or Object-creating [35, 42, 47, 61, 74, 78, 86, 105]. Such a distinction is due to the disagreement on whether it is possible to have all required relationships defined at the modeling phase. We and others, e.g., [78, 86], argue that the definition of new relationships and the creation of new objects, should be supported by a query model. A new relationship may have either a stored or a derived value and handled as the introduction of an instance variable or a method to the definition of a class, respectively. For the instance variable case, objects in the class to which the relationship is added are extended to include values for the new instance variable. For the method case, on the other hand, the behavior is extended without any stored value being added because the value of the relationship is determined by invoking the corresponding method as it is needed. A new object may be formed by collecting values from either objects in different classes or from the constituents of an object nested to an arbitrary level. However, it is necessary to resolve problems that arise due to the creation of objects; otherwise there will be inconsistencies. Among such problems is the maintenance of the closure property [5]. In other words, the output of a query should be allowed as an operand in the model.

2.2 Characteristics and Drawbacks

A major drawback of the languages already described in the literature [29, 72, 97, 98] is that they do not maintain the closure property. Others introduce non-object-oriented constructs for maintaining the closure property. Although operands in such languages have object-oriented properties, the outputs are relations which do not have the same structural and behavioral properties as the original objects. Consequently, the result of a query cannot be further processed by the same set of language operators. For instance in O_2 [42, 49] the value concept was introduced. In O_2, there are two distinct notions, class and type. While a class has objects with identities and encapsulate data and behavior, on the other hand, a type has only values. To every class there is an

associated type describing the structure of its instances. O_2 has an object algebra which handles values as well as objects and this leads to a kind of mismatch in having some operands violating encapsulation while others not. O_2 allows users to violate encapsulation when doing *ad hoc* queries. The query language of O_2 does not consider computational completeness as it is embedded inside CO_2, the programming language of O_2. The algebra of O_2 supports select, cross-product, set operations and a reduction operation similar to that of Kuper and Vardi [66] which reduces one field tuple structure to the elements of a set. The query languages of [24, 41, 65, 82] use nested relations as their logical view of object-oriented databases. A nested relation is allowed as an operand in addition to other operands with object-oriented features. For instance, the algebra described in [65] is based on supporting nested relations where a nested relational algebra is proposed to provide greater expressive power to deal with the hierarchical structure of data. Although operators in these languages operate on and produce nested relations, we argue that nested relations do not form a proper logical representation of object associations. In order to use nested relations to represent objects, a large amount of data has to be replicated in the representation.

The query language of Gemstone is a calculus sublanguage embedded inside OPAL, the object-oriented programming language of Gemstone. Furthermore, queries in Gemstone violate encapsulation because they are formed over the instance variables of an object. A similar query language is that of the ObjectStore database management system [67, 77]. The query language of [77], in addition to being categorized as object preserving, it is based on making $C++$ persistent. The Postgres data model is a successor of the INGRES [94] relational database system. It is an extended relational data model which includes abstract data types, data of type procedure (Postgres stores QUEL and C procedures as attribute values) and attribute and procedure inheritance. It also allows attributes which are arrays of conventional types. Its query language POSTQUEL is an extension of QUEL to satisfy the new constructs. Postgres is not considered object-oriented and although it supports abstract data types and inheritance, it utilizes relational query processing techniques. Such extended relational models with abstract and procedural data types are still considered value-based and record-oriented models. They aim at adding extensibility and object management capabilities to the relational model. Another extension of QUEL is GEM [104], which is a general purpose query language for the DSIS data model [68], which is a semantic data

model of the entity relationship type. GEM provides support for the notions of entities with surrogates, generalization, set valued attributes and reference attributes which have as value an entity occurrence. Galileo [7] is a complete programming language based on the functional programming language ML [75]. Gelileo is strongly typed and incorporates a model of data which is much like that of a semantic data model. It embeds such a model in the type system of a programming language with many of the features of modern object-oriented languages such as abstraction and hierarchy.

The algebra of Vandenberg [101] has an expressive power equivalent to the EXCESS query language of the EXTRA data model described in [41]; it assumes a data model in which several general type constructors are provided, and data structures are built through free composition of those constructors. However, since the EXCESS query language is based on QUEL, the underlying query processor of EXCESS is relational. In EXCESS, new types created during query processing do not participate in inheritance in any way -they do not inherit any attribute or method except those explicitly specified from the types from which they were created, nor they become part of any inheritance hierarchy. A major drawback of the algebra described in [101] is that values are the output from any query. We argue that the actual value of any object is relevant solely for output purposes. Thus, it is an overhead to have values as the output from a query. Instead, a predefined method could serve the purpose as the actual values are of output concern. It is an implementation issue not to be considered while formally dealing with an object algebra. For instance, a method *display(i)* could be provided with i being an argument to specify the depth of nesting up to which the values of instance variables with object values are to be resolved before the value of a given object is presented to the output device. Thus, *display(0)* only makes atomic values of an object available to the output device by ignoring values from domains which are objects in some classes. while *display(*)* resolves all values drawn from non-atomic domains and hence presents to the output device all the atomic values inside an object regardless of how deep they are nested. Based on the algebra described in [101] is that developed for Relevation system [45] and described in [100]. Relevation is a project on query processing in object-oriented databases.

The Daplex functional data model [88] illustrates an integration of functions, relations and object-oriented features. Its basic constructs are entities and functions which are intended to model conceptual objects and their properties. The Daplex

query language has a set of iterators that apply a predicate to a set of values. The two basic iterators are *for each* and *for some*. The former returns values satisfying a given predicate and the latter returns true when at least one value satisfies a given predicate. The algebra of PDM [47, 74] is based on an extension of the Daplex functional data model [88]. While Daplex supports only functions whose values are stored in the database, PDM has been extended to include functions whose values are derived from other values or computed by arbitrary procedures. PDM modifies the relational algebra to handle functions, i.e., the operators and the result are functions. The *apply and append* operator of the PDM algebra is equivalent to the relational join where a function is applied on argument tuples and the result is appended to them. A major restriction is that object identity is not supported and only union compatible items are allowed as operands to set-based operators.

The algebra of ENCORE [86, 105], is based on a data model [56] that has all types as abstract data types whose implementations are hidden from the algebra. It comprises a set of built-in functions to collection objects. These functions include, predicate-based selection of objects, collection manipulation and creation of new types and their instances. While the *image* operation projects on a single property of the operand, the *project* operation on the other hand does one or more evaluations of the *image* operation and collects the results in a tuple. The *ojoin* operation produces unnested collections similar to a flat relation where the *nest* operation adapted from the nested relational model is applied when nesting is required. The output of a query is of the Tuple type which is essentially the nested relational representation, since it allows the nesting of tuples. To insure type consistency of the result, in ENCORE union compatibility has been imposed on the algebra operators. Union compatibility states that members of the sets being operated on must be objects of types which are in a subtype relationship with one another. ENCORE views everything as an object with an identity. An opposing viewpoint [49] is that there is a distinction between objects, which possess identity and values which do not. A drawback of the latter algebra is that two identical queries don't give the same response. This is so because every resulting collection is a newly identified object in the database. For this reason, operators which eliminate duplicates are defined in this algebra. Such operators were not necessary for the case of not having the result from a query to be of the Tuple type, but having the characteristics of a class which is properly and naturally placed in

the lattice. Another algebra which is similar to that of ENCORE is the one described in [43]. However, this algebra returns values rather than objects as the output from a query and consequently does not generate object identifiers. Also, it does not use methods.

Straube and Özsu developed a set-based object-oriented query algebra and a corresponding calculus, but their algebra does not satisfy the closure property. Also, they studied the problem of type unions in some detail. Their Map operator is similar to *Apply to all* operator in functional data models and to the *image* operator of ENCORE. However, although it has a formal basis, their algebra is less expressive compared to others described in the literature. Their object calculus is similar to the tuple relational calculus definition provided in [99].

Osborn's object algebra [78] was developed for a general object-oriented data model defined on three generic classes of atomic, aggregate and set objects. She extends the relational algebra by adding an *Apply* operator to apply operations on objects and *Deepcopy* to create a complete copy of an object without sharing any sub-objects with the old one; her *combine* operator is equivalent to the relational join. A major drawback of Osborn's algebra is that it does not support encapsulation and the closure property is not thoroughly maintained; set operations do not accept atoms and aggregate objects produced by other operations.

The first version of the query language developed for ORION [29] preserves objects in the database; it is only based on the selection operation assuming the other operations to be implicitly present at the modeling phase. A major drawback of this language is that a query on a class returns either the value of a single attribute or some objects of the class; for the former case, objects from the domain of the attribute are returned. Also, when more than one class are involved in a query, those classes have to be nested with respect to each other. However, the second version of this query language considers the addition of new objects by explicitly providing operations which were assumed implicit. In the second version of the query model of ORION [61], although the result of a query operation is a class, the improper placement of resulting classes in the lattice leads to duplication of class contents; hence ORION violates the reusability feature of object-oriented systems. However, we argue that it is an overhead to have a class as the output of a temporary query, as ORION does. In this thesis, we describe the output of a query by the minimum requirements of an operand and

from such characteristics we show how to derive the characteristics of a class when it is required to have the result persistent [12, 13]. In $OSAM^*$ operands in a query are the database itself and all subdatabases derived from the original database by query operations; the result of a query is a subdatabase.

The work described in [35] focuses on class hierarchies, behavioral aspects and the closure property; however, neither a data model nor a query language has been proposed. Rather, the impact of the features of object-oriented data models on the design of a query language is discussed. The goal behind the work described in [35] is to define a unifying framework that constitutes a common basis for the presentation and discussion of the features of object-oriented query languages. For this purpose, an object-oriented predicate calculus (OOPC) based on the relational predicate calculus and its extensions for nested relations, is defined. The underlying data model assumes only simple inheritance. In their attempt to maintain the closure property, they follow the approach proposed by W. Kim in [61] in having the result of a query as a class which is made a direct subclass of the root, TOP-CLASS in their model. Furthermore, the result of a query does not have user defined methods; it much resembles a set of tuples in a relation and consequently encapsulation is not respected.

In [89] Siegelmann and Badrinath describe an algebra where query results are presented as implicit answers (expressions), in which a class name replaces an explicit enumeration of all its instances in a step towards allowing information exchange at higher levels of abstraction which is a useful capability in decision support systems. A subset of instances from a class are explicitly enumerated only in case that there is no class that includes all of them and no other instances. However, the data model on which their algebra is based supports only simple inheritance and atomic domains, i.e., no complex objects. Also, they do not describe any method for making an implicit answer explicit. The same concept of implicit answers is also discussed in [87].

The aim of the study described in [69] is to develop a deductive-augmented database system that could handle recursive queries within the realm of an object-oriented database system via a query language which is based on PROLOG. However, they treat the unification process without maintaining the object-oriented features. In other words, their introduction of *meta-variables* only serves the unification process while the resulting data model does not satisfy multiple inheritance and schema evolution functions could never be handled within such a model; although both are considered

among the basic features of an object-oriented database system.

In [83] it is claimed that class creation by set operations has been ignored in the literature. Consequently, the research described in [83] focuses on presenting a framework for executing set theoretical operations on the class construct. Thus, class definition abstractions including *specialization, selection* and *cartesian-product* are not addressed. Although the authors described the *union, difference, intersection* and *symmetric difference* operations, we argue that the *union* and *difference* operations would be sufficient as the others are representable in terms of them. For instance, given two sets A and B, $A \cap B = A - (A - B)$ and $A \Delta B = (A \cup B) - (A \cap B)$. The authors do not handle heterogeneous sets. They simply mix objects from two classes into one class. When the two participating classes are not type compatible, objects added from any of the two classes into the result are extended to include the value *undefined* for the additional properties due to the other class. By this, they get homogeneous set of objects in any class, although having a heterogeneous set is more flexible in having any object in a subclass being considered in all its superclasses. Instead of having the value *undefined* for some of the properties of an object, we argue that the same thing could be achieved by dynamically allowing to extract from an object any of the properties defined for a subclass of its class and returning the value *undefined* for the case of the object not having the required property. This should be facilitated benefiting from defining objects in a class to also include objects of its subclasses. Hence, it is not necessary to have objects in a class having only the properties facilitated in that class, but every such object must at least possess such properties; other properties may be possessed due to considering that object in a subclass of the former class. For instance, if it is required to find the salary of all persons, the value *undefined* is returned for every object in the person class not coming from any of its subclasses and the actual salary is returned for employed persons. However, always the semantics of the operation applied to an object should be considered for not to allow semantically incorrect situations to arise. For the latter case, instead of the value *undefined*, an indication of semantically incorrect operation should be considered. In other words, a distinction between semantically incorrect and *undefined* should be considered. For instance, having an object o from the class of *apples* makes *o children*() semantically incorrect. Furthermore, the authors argue that having objects in class A being subset from objects in class B does not lead to the fact that A is a subclass of B. This is

so because they assumed a set to be untyped according to the set theory concepts. However, it is important to have elements of a set having some common properties which are utilized as the set is expressed in comprehension. Such common properties are due to the common type applicable to all elements of the set. Also, it is possible to have subsets from a given set A with elements constituting each such subset having more common properties than those common to other elements in the set A. This is analog to the type and subtype terminology in object-oriented data models. So, their differentiation between a type and a set does not sound reasonable. We argue that for every set of objects there exists at least one common type c and any subset from that set has its corresponding type being either the type c itself or a subtype of it. For the worst case, c is considered to be the root OBJECT class. Another research concerned with set operations could be found in [55]. However, in there the type of the result is specified without any reasoning.

Chapter 3

The Data Model

In this chapter we describe the features necessary to be present in a data model* as they relate to the object algebra. In section 3.1, an informal description of such features is presented. The basic notations related to the data model and utilized in the rest of this thesis are enumerated in section 3.2. In section 3.3, we elaborate on message expressions as one of the basic constructs in the formalization developed and proposed in this thesis.

3.1 Informal Description

The data model is required to support objects, classes and methods. An object has a state and behavior where the state is reachable via the behavior. To maintain the object-oriented features, it is important for the operators of the object algebra to equally handle both the state and the behavior of objects. Furthermore, an object has an identity and a value. Identity distinguishes one object in the database from other existing objects and provides for object sharing [58]. A value may be either a single value or a set of values drawn from a particular domain. A domain is either atomic or non-atomic; an atomic domain may be any of the conventional domains including integers, characters, etc. On the other hand, a non-atomic domain includes the set of objects of a class represented by their identities. The following are objects where o_i represents identity:

*A detailed description of the data model could be found in [8].

o_1 <"*Jack*", 21, "*M*", ϕ >

o_2 <"*Mary*", 42, "*F*", {o_1, o_7} >

o_3 <"*Michel*", 5, "*M*", ϕ >

o_4 <"*John*", 65, "*M*", {o_6, o_8} >

o_5 <"*Susan*", 25, "*F*", ϕ, 5, {o_{11}, o_{16}}, o_{10} >

o_6 <"*Smith*", 45, "*M*", {o_1, o_7}, 50*K*, o_{10} >

o_7 <"*Tom*", 18, "*M*", ϕ, 3, {o_{13}, o_{14}}, o_{10} >

o_8 <"*Adams*", 40, "*M*", ϕ, 60*K*, o_{10} >

o_9 <"*George*", 22, "*M*", ϕ, 5, {o_{11}, o_{16}}, o_{10}, 15*K*, o_{10} >

o_{10} <"*Computer Science*", o_8 >

o_{11} <"*CS565*", "*Database Theory*", 3, ϕ >

o_{12} <"*CS101*", "*Introduction to Programming*", 3, ϕ >

o_{13} <"*CS211*", "*Design of Programming Languages*", 3, {o_{12}} >

o_{14} <"*CS330*", "*Data Structures*", 3, ϕ >

o_{15} <"*CS450*", "*Database Design*", 3, {o_{13}, o_{14}} >

o_{16} <"*CS578*", "*Parallel Machines*", 4, ϕ >

Objects that have the same state structure and behavior definition are collected in one class. For instance, looking at the previous objects, it seems that o_1 and o_2 should be in the same class. A class definition includes a set of instance variables that reflects properties of its objects, a set of methods (operations) applicable to its objects to support encapsulation and information hiding and a set of superclasses to provide reusability through inheritance. Inheritance is supported to overcome duplication of definition and to allow for reusability. Inheritance covers state structure and behavior definition. Next are the state structures of the classes related to the previous objects:

person < $\emptyset$*, *name*:*string*, *age*:*integer*, *sex*:["*M*", "*F*"], *children*:{*person*} >

student < {*person*}, *year*:*integer*, *courses*:{*course*}, *student-in:department* >

staff < {*person*}, *salary*:*integer*, *works-in:department* >

research-assistant < {*student*, *staff*} >

course < $\emptyset$, *code*:*string*, *name*:*string*, *credit*:*integer*, *prerequisites*:{*course*} >

department < $\emptyset$, *name*:*string*, *head*:*staff* >

where any pair *iv:d* represents an instance variable defined such that *iv* is the instance

*The empty set indicates that the root OBJECT class is the direct superclass of the class *person*

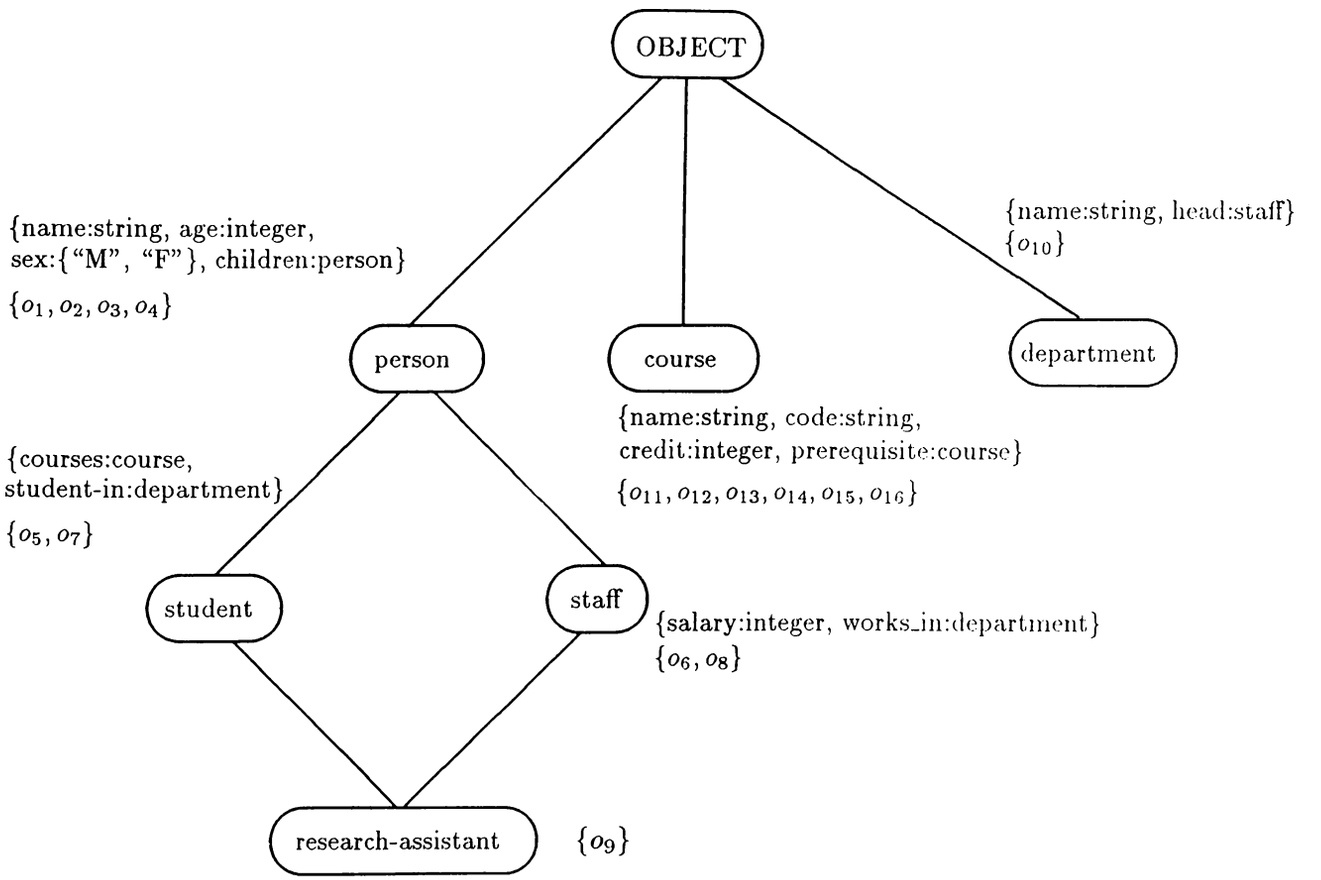


Figure 3.1: A graphical representation of the example classes

variable name and d is the corresponding underlying domain. For instance, the domain of the *age* instance variable is the set of integers. A domain specified between braces indicates that always a set is expected as the value of that instance variable, even a single element is represented by a singleton set. For instance, *children*:{*person*} specifies a set of objects from the *person* class as the children of a person.

The first argument in a class definition is a set with elements being classes from which inheritance is achieved. We say that *person* is a superclass of *student* and *staff*, while each of *student* and *staff* is a subclass of *person*. Any instance in *student* or *staff* is actually an instance in *person* but the reverse is not true. A subclass may include additional instance variables and behavior definition. As inheritance is

concerned, classes are arranged in a lattice† with the general class OBJECT at the root, i.e., direct or indirect superclass of all other classes. The root OBJECT class includes the definition common to all the classes found in the schema. An empty set of superclasses for a class c implicitly indicates that the class c is a direct subclass of the root OBJECT class. Conversely, a non-empty set of superclasses could not include the OBJECT class which otherwise automatically becomes an indirect superclass for being a direct or indirect superclass of those classes constituting the set of superclasses of the class c. Furthermore, given any two classes c_1 and c_2 in the set of superclasses of the class c, it is required that neither of them, i.e., c_1 nor c_2 be a direct or indirect superclass of the other. This is illustrated in Figure 3.1 where the example classes are shown with their corresponding instance variables and objects. The super/subclass relationship is also indicated in Figure 3.1.

3.2 Basic Notations

Related to a class c we use the following notation:

- messages(c) is the set of messages used to invoke any of the methods defined in or inherited by class c. In other words, every method H is invoked via a corresponding message and implements a predefined function

  $$f : d_1 \times d_2 \times \ldots \times d_n \longrightarrow d_r,$$

  where d_1 is the domain of the receiver, $d_2, d_3, \ldots, d_n$ are the domains of the arguments of f and d_r is the domain of the result of the application of f on objects of d_1, i.e., d_r is the range of f. Given objects $o_i \in d_i$, where $i = 1$ *to* n *and* r,

  $$f(o_1, o_2, \ldots, o_n) = o_r.$$

  The message that invokes the method H should have $(n-1)$ arguments drawn from the domains d_2 to d_n, respectively. For instance, the method invoked by the message *name*() implements the function

  $$f_1 : T_{instances}(person) \longrightarrow string.$$

  Function f_1 does not expect any argument because corresponding domains are not specified. The message *increase-salary(i)* invokes the method implementing the function

†The term lattice is used to refer to a Directed Acyclic Graph, which is in fact a semi-lattice

$$f_2 : T_{instances}(staff) \times integer \longrightarrow integer,$$

where given $o \in T_{instances}(staff)$, $f_2(o, i) = (o\ salary()) + i$.

The domain of the receiver of f_2 is $T_{instances}(staff)$ and f_2 expects a single argument from the domain that is the set of integers. Also, the result of f_2 is from the set of integers, i.e., range of f_2 is the set of integers. For instance,

$$f_2(o_9, 2K) = o_9\ salary() + 2K = 15K + 2K = 17K.$$

Therefore, methods are used not only to deal with properties of objects but also to manipulate either stored values or in deriving new values in terms of properties and existing values of objects. Related to the previous classes, the following sets of messages are assumed:

$$messages(person) = \{name(), age(), sex(), children()\}$$

$$messages(staff) = \{name(), age(), sex(), children(), salary(), works\text{-}in(), net\text{-}salary(t), increase\text{-}salary(t)\}$$
$$= messages(person) \bigcup \{salary(), works\text{-}in(), net\text{-}salary(t), increase\text{-}salary(t)\}$$

$$messages(student) = messages(person) \bigcup \{year(), courses(), student\text{-}in()\}$$

$$messages(research\text{-}assistant) = messages(student) \bigcup messages(staff)$$

$$messages(department) = \{name(), head()\}$$

$$messages(courses) = \{code(), name(), credit(), prerequisites()\}$$

Thus a class defines the behavior of its objects by providing a set of methods operating on them. A message should have as many arguments as the corresponding function expects. For instance, no arguments are specified for the message *name*(), while one argument is specified for the message *net-salary(t)* and drawn from the domain which is the set of reals as indicated in the corresponding function,

$$f_3 : T_{instances}(staff) \times real \longrightarrow integer.$$

Although the argument of f_3 is from the set of reals to specify the percentage deduction in salary, the result is rounded to be in the set of integers.

Given a class c, let $m \in messages(c)$ and $o \in T_{instances}(c)$. Applying the message m to the object o, i.e., $o\ m$, returns a value from the range of the function which corresponds to the message m. Let V be the set of all such values. The returned value may be either a single value or a set of values as the latter is allowed as a value in definition 3.2, given next. Furthermore, given $O \subseteq T_{instances}(c)$, applying

m to objects in O, i.e., $O\ m$, returns a set of subsets from the set V. An element in the returned set may be either a single value or by itself a set. This is because the message m is actually applied to the individual objects constituting the set O and the obtained results form a set.

- $I_{variables}(c)$ is the set of all instance variables defined in or inherited by class c. For any instance variable iv, domain(iv) and value(iv) denote the domain and the value of instance variable iv. The value of an instance variable is drawn from the corresponding domain. For instance, having the domain of the *children* instance variable in the *person* class being specified as *person* between braces, forces its value to be solely any subset of the objects in the *person* class and nothing else. Next are the formal definitions of possible *domains* and *values*.

Definition 3.1 (Domain) *The set of domains D is defined to include:*

- *$d \in D$ and $2^d \in D^*$, where d is any of the atomic domains such as the set of integers, the set of reals, the set of characters, etc,*
- *for any class c_i $T_{instances}(c_i) \in D$ and $2^{T_{instances}(c_i)} \in D$.* □

Definition 3.2 (Value) *The set of values V is defined to include:*

- *$\forall\ d_i \in D$, we have $d_i \subseteq V$*
  Since ϕ is a subset of any set, then $\phi \in V$ due to the use of the powerset in specifying some domains in D. So, ϕ is used to indicate the value nil[†]. □

Given a class c, let $iv \in I_{variables}(c)$, then $value(iv) \in domain(iv)$. The instance variable iv expects either a set value or a single value depending on whether the corresponding domain is specified between braces or not, respectively. For instance, the instance variable *children* in the *person* class expects a set value for its domain being specified between braces, while the instance variable *head* in the *department* class expects a single value in accordance with its specified domain. A single value is drawn from the set d for an atomic domain and from $T_{instances}(c_i)$

*2^d indicates the powerset or the set of all subsets of the set d

†sending any message to *nil* returns *nil*, i.e., $\phi\ m = \phi$, for any message m.

for a non-atomic domain, while a set value including singleton sets is drawn from 2^d for an atomic domain and from $2^{T_{instances}(c_i)}$ for a non-atomic domain.

The name of an instance variable, actually a message, when sent to an object, returns the value of the instance variable in the receiving object. For instance,

o_1 age() returns 21 as the age of object o_1, while

o_5 courses() returns $\{o_{11}, o_{16}\}$ as the set of courses o_5 is attending, and

o_5 courses() name() returns {*"Database Theory", "Parallel Machines"*} as the set of names of the courses o_5 is attending.

Such methods return existing stored values while other methods that do not correspond to any instance variable return derived values, computed starting with existing stored values as will be shown in example 4.6.

- *instances*(c) is the set of objects in class c but not in any of its subclasses. Any object in *instances*(c) must have some value for any instance variable in $I_{variables}(c)$ and nothing more. It also must understand all the messages in *messages*(c). For instance,

  instances(*person*) = $\{o_1, o_2, o_3, o_4\}$
  instances(*student*) = $\{o_5, o_7\}$
  instances(*staff*) = $\{o_6, o_8\}$
  instances(*research-assistant*) = $\{o_9\}$
  instances(*course*) = $\{o_{11}, o_{12}, o_{13}, o_{14}, o_{15}, o_{16}\}$
  instances(*department*) = $\{o_{10}\}$

  An object has an identity, a value and belongs to a certain class. Related to an object o we use *value(o)* and *identity(o)* to denote the value (the value of an object is a set of values of the instance variables defined in its class; simple values or identities of nested objects) and the identity of object o, respectively. For instance, $<"\text{Jack}", 21, "M", \phi>$ is the value of the object whose identity is o_1.

  Given an object $o \in instances(c)$ for some class c,

  $value(o) \in X_{i=1}^{card(I_{variables}(c))} (domain(I_{variables}(c)_i)$[‡].

[‡]given two sets A and B, $A \times B = \{(x, y) \mid x \in A \wedge y \in B\}$. In general $A \times B \neq B \times A$, however here we assume that equality always holds,i.e., order is not important inside the resulting tuples because those values are handled via messages

Based on the notion of *value* and *identity* we define equality of objects:

Definition 3.3 (Equality of objects) *Two objects o_1 and o_2 are:*

- *identical ($o_1 = o_2$) iff identity(o_1)=identity(o_2)*
- *shallow-equal ($o_1 \doteq o_2$) iff value(o_1)=value(o_2)*
- *deep-equal ($o_1 \cong o_2$) iff by recursively replacing every object o_i in value(o_1) or value(o_2) by value(o_i), equal values are obtained.* □

$$(o_1 = o_2) \Rightarrow (o_1 \doteq o_2) \Rightarrow (o_1 \cong o_2)$$
$$\text{identical} \Rightarrow \text{shallow-equal} \Rightarrow \text{deep-equal}$$

and these correspond to identity, shallow-equality and deep-equality of *Smalltalk-80* [54].

- $T_{instances}(c)$=instances(c)$\bigcup_{i=1}^{card(S)} T_{instances}(S_i)$
  where $S = \{S_1, S_2, ..., S_{card(S)}\}$ is the set of direct subclasses of class c, i.e., $c \in supers(S_i)$. Notice that for an object to be in $T_{instances}(c)$, it must have some value for every instance variable in $I_{variables}(c)$ and must understand messages in $messages(c)$. This is so because for any class S_i being a subclass of class c, class S_i has $I_{variables}(S_i) \supseteq I_{variables}(c)$ and $messages(S_i) \supseteq messages(c)$. In other words, any object in $T_{instances}(S_i)$ has at least some value for every instance variable in $I_{variables}(S_i)$ and hence in its subset $I_{variables}(c)$ and understands messages in $messages(S_i)$ and hence in its subset $messages(c)$. In fact, an object in $T_{instances}(c) \bigcap T_{instances}(S_i)$ has values for more instance variables than those required for such an object to be considered in $Tinstances_c$. The additional values are due to the instance variables and the messages in $(I_{variables}(S_i) - I_{variables}(c))$ and $(messages(S_i) - messages(c))$, respectively. For instance:

$$T_{instances}(course) = \{o_{11}, o_{12}, o_{13}, o_{14}, o_{15}, o_{16}\}$$
$$= instances(course)$$
$$T_{instances}(department) = \{o_{10}\}$$
$$= instances(department)$$
$$T_{instances}(research\text{-}assistant) = \{o_9\}$$
$$= instances(research\text{-}assistant)$$

$$T_{instances}(staff) = \{o_6, o_8, o_9\}$$
$$= T_{instances}(research\text{-}assistant) \bigcup \{o_6, o_8\}$$
$$= T_{instances}(research\text{-}assistant) \bigcup instances(staff)$$
$$T_{instances}(student) = \{o_5, o_7, o_9\}$$
$$= T_{instances}(research\text{-}assistant) \bigcup \{o_5, o_7\}$$
$$= T_{instances}(research\text{-}assistant) \bigcup instances(student)$$
$$T_{instances}(person) = \{o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8, o_9\}$$
$$= T_{instances}(student) \bigcup T_{instances}(staff) \bigcup \{o_1, o_2, o_3, o_4\}$$
$$= T_{instances}(student) \bigcup T_{instances}(staff) \bigcup instances(person)$$

Notice that, for any class c which has no subclasses, $T_{instances}(c) = instances(c)$. Also, as already elaborated, having an object o in $instances(c)$ for some class c, leads to the fact that o is in $T_{instances}(c_i)$ for any class c_i which is a superclass of the class c. This is true because by definition, $I_{variables}(c)$ includes $I_{variables}(c_i)$, i.e., $I_{variables}(c_i) \subseteq I_{variables}(c)$. Hence any object in the class c possesses the properties expected for objects to be considered in $T_{instances}(c)$ and could respond to any of the messages understandable by objects in $T_{instances}(c_i)$. In fact, the object o could have more properties than those expected for objects in any of the classes c_i. This is so because of the possible additional properties possessed by the object o for being in $instances(c)$. For instance, $o_5 \in instances(student)$ and $o_5 \in T_{instances}(person)$ because the *person* class is a superclass of the *student* class. The object o_5 has all of the properties expected for an object to be in $T_{instances}(person)$, i.e., the properties defined in the *person* class. In addition, the object o_5 has the additional properties defined in the *student* class due to being considered in $instances(student)$.

- $supers(c)$ is the set of direct superclasses of class c. For instance,

  $supers(person) = \phi$
  $supers(student) = \{person\}$
  $supers(staff) = \{person\}$
  $supers(research\text{-}assistant) = \{student, staff\}$
  $supers(course) = \phi$
  $supers(department) = \phi$

$\forall\, c_i \in supers(c)$, *we have* $c_i \neq OBJECT$, the object class is implicitly a direct superclass of the class c for $supers(c) = \phi$; otherwise, it is an indirect superclass for being a direct or indirect superclass of a class in $supers(c)$.

When class c_2 is a superclass of class c_1, we say c_1 is a subclass of c_2. Instances of c_1 have at least the properties defined for instances of c_2 and operations defined in c_1 are at least those of c_2, i.e., c_1 inherits the properties and operations defined in c_2. So, any instance in class c_1 is an instance in class c_2, but the reverse is not true. Related with the subclass/superclass relationship between classes, we define a partial ordering ($\leq_c$) among classes.

Definition 3.4 (Partial ordering ($\leq_c$) among classes)
Given two classes c_1 *and* c_2, *we say that* $c_1 \leq_c c_2$ *iff:*

- $I_{variables}(c_2) \subseteq I_{variables}(c_1)$
  That is, $\forall iv_2 \in I_{variables}(c_2)\ \exists iv_1 \in I_{variables}(c_1)$ *such that,*
  $iv_2 = iv_1 \wedge (domain(iv_1) \leq_c domain(iv_2) \vee domain(iv_2) = domain(iv_1))$
- $methods(c_2) \subseteq methods(c_1)$ □

The second property in definition 3.4 reflects a Cardelli-like semantics of subtyping [39].

$c_1 \leq_c c_2 \Leftrightarrow c_1$ is a subclass of c_2 and c_2 is a superclass of c_1
$\Leftrightarrow instances(c_1) \subseteq T_{instances}(c_1) \subseteq T_{instances}(c_2)$

3.3 Message Expressions

Elements of messages(c) are used only to invoke methods of the class c. When the result is an object o_i, messages in the class of object o_i are used to invoke methods applicable to it. So, combining a message in the class c which returns an object o_i as its result with any of the messages in the class of object o_i will form a message pair applicable to objects in class c to access possible values in related objects in the class of object o_i. Also when any of such pairs returns an object as its result, messages in the class of the latter object could be combined with that pair forming triples applicable to objects in class c. In the same way, quadruples, quintuples and so on, could be formed.

For instance, o_9 is an object in the *student* class; the message *courses*() in the *student* class invokes the method implemented to return the set of courses registered by a given student and so,

o_9 *courses*() returns $\{o_{11}, o_{16}\}$ from the *course* class.

Any of the messages in the *course* class, e.g. *code*(), could be applied to any object in the *course* class and

$\{o_{11}, o_{16}\}$ *code*() returns $\{"CS565","CS578"\}$

At this point one could say that the combination *courses*() *code*() could be applied to any object in the *student* class and

o_9 *courses*() *code*() returns $\{"CS565","CS578"\}$.

It is recognized that both *courses*() and *courses*() *code*() are elements of a superset of *messages*(*student*) as the latter does not include the element *courses*() *code*(). We call such a superset *the set of message expressions* of the class *student* and every element of this set is called a *message expression.* The set of message expressions of a class c is denoted by $M_e(c)$ and every element of $M_e(c)$ returns either a stored value or a derived (i.e., computed) value. As formally stated in the following definition, elements of $M_e(c)$ are recursively defined in terms of messages, starting with messages(c).

Definition 3.5 (Message expressions) *Given a class c, the set $M_e(c)$ is defined by:*

- *messages(c)*$\subseteq M_e(c)$
- *if* $x \in M_e(c)$ *and x returns a value from* $T_{instances}(c_1)$ *then*
  $(x$ *messages*$(c_1))^* \subseteq M_e(c)$

Therefore, starting from messages(c) we can determine whether a given message expression is an element of $M_e(c)$. □

We use $len(x)$ to denote the length of message expression x, i.e., the number of messages constituting x. For instance, $len($*department*() *head*() *salary*$()) = 3$, i.e., the message expression *department*() *head*() *salary*(), which returns the salary of the chairperson of the department attended by a given student s a sequence of three messages. Related to the previous classes, the following sets of message expressions are derived:

---

*x is concatenated with every element of the set of messages of class c_1. For example, $(x\ \{m_1, m_2\})=\{x_{11}, x_{21}\}$ where $x_{11} = (x\ m_1)$ and $x_{21} = (x\ m_2)$

$M_e(person) = messages(person) \bigcup children()^+ messages(person)$ †
$= children()^* messages(person)$
$M_e(student) = M_e(person) \bigcup \{year(), courses(), student\text{-}in()\}$
$\bigcup student\text{-}in()\ M_e(department)$
$M_e(staff) = M_e(person) \bigcup \{salary(), works\text{-}in(), net\text{-}salary(), increase\text{-}salary()\}$
$\bigcup works\text{-}in()\ M_e(department)$
$M_e(research\text{-}assistant) = M_e(student) \bigcup M_e(staff)$
$M_e(department) = messages(department) \bigcup head()\ M_e(staff)$
$M_e(course) = prerequisites()^*\ messages(course)$

We differentiate between implicit and explicit representations of $T_{instances}(c)$, $messages(c)$ and $M_e(c)$ for a given class c. While in an implicit representation a subset of the elements of the represented set is substituted by a single set name, an explicit representation enumerates all elements of a set. For instance,

$$\{name(), age(), sex(), children(), year(), courses(), student\text{-}in()\}$$

is an explicit enumeration of the set $messages(student)$, while

$$messages(person) \bigcup \{year(), courses(), student\text{-}in()\}$$

is an implicit representation of the same set. As illustrated in the example classes and by definition, for any class c, $T_{instance}(c)$ could be implicitly represented in terms of $instances(c)$ and $T_{instances}(c_i)$, where c_i is a subclass of c; $messages(c)$ could be implicitly represented in terms of $messages(c_j)$, where $c_j \in supers(c)$; $M_e(c)$ could be implicitly represented in terms of $M_e(c_k)$, where $c_k \in supers(c)$ or $c_k = domain(iv)$ for some instance variable $iv \in I_{variables}(c)$. Sometimes, it is not possible to explicitly enumerate elements of $M_e(c)$ for some class c, especially in case of a direct or indirect cycle in specifying the domain of instance variables. In such cases, an implicit representation becomes necessary and easier to follow and understand. For instance, because of the cycle caused by having the *person* class itself as the domain of the *children* instance variable in the *person* class, there is no explicit representation of $M_e(person)$, however an implicit representation is possible and given above in the examples on message expressions.

After introducing message expressions, it is necessary to decide on the relationship between the sets of message expressions and the sets of messages of two classes. Such

† notice that a^* is used to indicate zero or more concatenations of a with itself, i.e., e, a, aa, ..., while a^+ indicates one or more concetanations of a with itself, i.e., a, aa, aaa,...

a relationship is important as a class is derived from the result of a query later in chapter 4.

Lemma 3.1 *Given two classes c_1 and c_2*

$M_e(c_1) \subseteq M_e(c_2) \iff$ *messages*$(c_1) \subseteq$ *messages*(c_2),

i.e., $\forall x \in M_e(c_1)$ *such that len*$(x)=1$ *we have* $x \in M_e(c_2)$.

Proof:

(if part) $x \in$ *messages*$(c_1) \Rightarrow x \in M_e(c_1)$ *(by definition 3.5)*

$\Rightarrow x \in M_e(c_2)$ *(because* $M_e(c_1) \subseteq M_e(c_2)$*, given)*

$\Rightarrow x \in$ *messages*(c_2) *(because len*$(x)=1$ *and the only elements of* $M_e(c_2)$ *of length one are elements of messages*(c_2)*, from definition 3.5)*

$\Rightarrow$ *messages*$(c_1) \subseteq$ *messages*(c_2),

(only if part) $x \in M_e(c_1) \Rightarrow x = x_1 \ldots x_n$, *with* $n \geq 1$ *such that* $x_1 \in$ *messages*(c_1) *(by definition 3.5)*

$\Rightarrow x_1 \in$ *messages*(c_2) *(because messages*$(c_1) \subseteq$ *messages*(c_2)*, given)*

$\Rightarrow x \in M_e(c_2)$ *(by definition 3.5)*

$\Rightarrow M_e(c_1) \subseteq M_e(c_2)$ □

We will utilize lemma 3.1 while constructing object algebra expressions in definition 4.2 and while deciding on the inheritance relationship between classes that correspond to object algebra expressions in theorem 4.2. Informally, the proof of lemma 3.1 follows from definition 3.5 where starting from message expressions of length one, i.e., messages of a class, it is possible to derive all other possible message expressions of that class.

A message expression when received by an object, returns a value from a particular domain. This particular domain is the range of the last message in the message expression. A returned value is either a stored or a derived value, a property that gives a full computational power to the user without having an embedded query language leading to impedance mismatch.

Due to message expressions, it is not necessary to have within the realm of object-oriented databases all required relationships with stored values. Deriving the value of a relationship in terms of existing ones is facilitated by the use of message expressions that handle derived values. For instance, it is possible to have the relationships *wife-of, husband-of, brother-of* and *sister-of* as derived values using the stored-valued

children relationship between persons with the *sex* value being recorded for persons. Each of *wife-of, husband-of, brother-of* and *sister-of* is handled as a message with an underlying method implementing the desired relationship. In general, a derived value is determined after executing a sequence of one or more methods underlying the message(s) constituting a corresponding message expression. Such a facility saves both space and time needed in storing and maintaining related values in a consistent state.

Chapter 4

The Query Model

In this chapter, we describe an object algebra where the closure property is maintained in a natural way without violating the object-oriented features. Although many of the existing query languages are restricted to the manipulation of existing objects without creating new ones, we and others [78, 85, 86, 105] argue the need for a more powerful query language that handles new objects in addition to the manipulation of existing ones. This adds the flexibility of introducing new relationships into the model making the manipulation more powerful. Our object algebra is a superset of the relational algebra and hence it is at least as powerful as the relational algebra. An operand e in our object algebra should have a pair of sets, a set of objects and a set of message expressions, denoted by $<T_{instances}(e), M_e(e)>$; using elements of $M_e(e)$ one can access elements of $T_{instances}(e)$. Since a class has a defined set of objects and a derived set of message expressions, a class can be an operand. The output of an operation as well should have a pair of sets derived in terms of the pair(s) of operand(s). So, an operand in a query could be replaced by another query whose output is the actual operand. We call any operand, whether an actual pair or an unevaluated query, an *object algebra expression.* Therefore, our object algebra acts on and produces items that have defined pairs. Hence our object algebra maintains the closure property in a natural way and remains within the realm of object-oriented concepts.

This chapter is organized as follows. Section 4.1 is an informal description of the different operators of the object algebra. The corresponding formal definitions are given in section 4.2, where object algebra expressions are recursively constructed. As the only known characteristics of the output from a query are the pair of sets, section 4.3 includes

a series of lemmata to derive other characteristics leading to a class. In section 4.4, the inheritance relationship between the classes due to the output from a query and the operand(s) is determined in a step towards maximizing reusability to have that class properly and naturally placed in the lattice and consequently the assignment of new identities is minimized. New identities are necessarily assigned only for cases where the output class can not be in inheritance relationship with any other class, i.e., it is a direct subclass of the root OBJECT class and with no subclasses. Illustrative examples are given in section 4.5 and linear recursion is treated in section 4.6. Equivalent object algebra expressions are included in section 4.8 where the associativity of the cross-product operation is proved; an important property as query optimization is concerned. Finally, we elaborate in section 4.7 on the superiority of the query model described in this thesis over the relational query model.

4.1 Informal Description

Concerning the operators, our object algebra includes the five basic operators of the relational algebra in addition to nest, one level project and aggregate function applications. The selection operation presents a restriction on objects of the operand. Although Straube claims that his multiple operand selection is more powerful than a selection with single operand [97, 98], we still insist on supporting a single operand selection. Because Straube does not maintain the closure property in his algebra, he has the cross-product operation embedded into the selection. We argue that on comparing two algebras, the power of the algebra as a whole must be considered, not merely the individual operations. A language that supports the creation of new objects is considered necessary and more powerful than any other language devoted to the manipulation of existing objects only. In our object algebra, the *selection* has a single operand and produces an output consisting of a pair, where the included objects are those satisfying a stated predicate expression, defined next. The set of message expressions of the resulting pair is the same as that of the operand. For instance, the result of selecting courses with no prerequisites from the *course* class is the pair:

$< \{o_{12}, o_{14}, o16\}, M_e(course) >$.

Depending on message expressions (definition 3.5) we define predicate expressions next.

Definition 4.1 (Predicate expressions) *The following are predicate expressions:*

P_1 : *T and F are predicate expressions representing true and false.*

P_2 : *Given two values y_1 and y_2 having the same underlying domain such that at least y_1 or y_2 is of the form $(o\ x)$, where o is an object variable bound to objects of an operand in a query and x is a message expression applicable to objects substituting o.*

$P_{2.1}$: $y_1\ op\ y_2$ *is a predicate expression where,*

$$op \in \begin{cases} \{=, \neq, \leq, \geq, >, <\} & \text{if both } y_1 \text{ and } y_2 \text{ are single values from an atomic domain} \\ \{\in, \notin\} & \text{if } y_1 \text{ is a single value and } y_2 \text{ is a set of values} \\ \{\subseteq, \not\subseteq, =, \neq\} & \text{if both } y_1 \text{ and } y_2 \text{ are sets of values, } y_2 \text{ may be } T_{instances}(e) \text{ where } e \text{ is an object algebra expression} \\ \{\equiv, \doteq, \cong\}, & \text{if both } y_1 \text{ and } y_2 \text{ are single values from a non-atomic domain, i.e., } T_{instances}(c) \text{ for some class } c. \end{cases}$$

$P_{2.2}$: $\forall | \exists z \in y_1 \wedge z\ op\ y_2$ *is a predicate expression where, y_1 is a set of values and*

$$op \in \begin{cases} \{=, \neq, \leq, \geq, >, <\} & \text{if } y_2 \text{ is a single value from an atomic domain} \\ \{\in, \notin\} & \text{if } y_2 \text{ is a set of values, } y_2 \text{ may be } T_{instances}(e) \text{ where } e \text{ is an object algebra expression} \\ \{\equiv, \doteq, \cong\} & \text{if } y_2 \text{ is a single value from a non-atomic domain} \end{cases}$$

$P_{2.3}$: $\exists z \subseteq y_1 \wedge z\ op\ y_2$ *is a predicate expression where, y_1 is a set of values and*

$$op \in \begin{cases} \{\subseteq, \not\subseteq, =, \neq\} & \text{if } y_2 \text{ is a set of values, } y_2 \text{ may be } T_{instances}(e) \text{ where } e \text{ is a query expression} \\ \{\ni, \not\ni\} & \text{if } y_2 \text{ is a single value} \end{cases}$$

P_3 : *if p and q are predicate expressions then (p), $\neg p$, $p \wedge q$ and $p \vee q$ are predicate expressions.*□

Let s_1 and s_2 be object variables ranging over instances of the *student* class:

- *"CS565"* $\in$ s_1 *courses() code()* is an example of $P_{2.1}$ to check students attending the course *"CS565"*;

- $\exists c \in s_1$ *courses*()$\wedge$ $c \in s_2$ *courses*()$\wedge$ $s_1 \neq s_2$ is an example of P_3, where an example of $P_{2.2}$, $\exists c \in s_1$ *courses*() $\wedge$ $c \in s_2$ *courses*(), is combined with an example of $P_{2.1}$, $s_1 \neq s_2$, to check whether two given students have at least one course in common;
- $\forall c \in s_1$ *courses*() $\wedge$ $c \notin s_2$ *courses*() is another example of $P_{2.2}$ to check whether two given students do not have any course in common;
- $\exists c \subseteq s_1$ *courses*() $\wedge$ $c \subseteq s_2$ *courses*() is an example of $P_{2.3}$ to check whether two given students have some courses in common.

So predicates within an object-oriented context are more powerful than in the relational model where only atomic values are compared. Furthermore, extending predicate expressions to allow quantifiers to propose the creation of objects does affect the query power. For example, $\exists O \wedge O \subseteq T_{instances}(c)$ for some class c, binds O to a subset of $T_{instances}(c)$; the subset objects to which O is bound could be built by this query. Such an object creation facility gives the algebra the power to do recursive queries by giving the ability to form a powerset [2].

Although the set of objects of an operand is in general heterogeneous, the only values accessible in each object are those specified by the set of message expressions of the pair. So, dropping some message expressions by the *project* operation hides some values from the accessible objects. For instance, by projecting the pair:

$< T_{instances}(person), M_e(person) >$ on $\{name(), age(), sex()\}$, the pair:

$< T_{instances}(person), \{name(), age(), sex()\} >$

is obtained. On the other hand, the *inverse of the project* operation is to extend the set of message expressions in a pair to include more message expressions applicable to objects of the pair, i.e., give more facilities to the user; this operation is defined in terms of others as shown later in definition 4.2 of section 4.2.

To facilitate the evaluation of a subset M_1 of the message expressions of a given pair against objects of the pair, the *one level project* operation is defined which forms new objects out of the obtained values and a corresponding set of message expressions is also determined to facilitate accessing the values encapsulated within the derived objects. For instance, having $\{name(), student\text{-}in()\, name()\}$ as a subset of $M_e(student)$, the *one level projection* of the pair: $< T_{instances}(student), M_e(student) >$ on this subset results in the derivation of the following pair:

$<\{<"Susan", "Computer\ Science">, <"Tom", "Computer\ Science">,$
$<"George", "Computer\ Science">\}, \{name(), name1()\}>.$

Notice the usage of *name1()* for the retrieval of the *department-name* value of the derived objects; it is done this way to resolve the conflict in naming. Here, it is suggested to tag a number (in sequence) to every duplicate of a message already added to the result.

In general, the set of message expressions in the result of the *one level project* operation is defined depending on message expressions in M_1 to include:

> all message expressions prefixed by any message m appearing as the last message in any of the message expressions x_1 in the subset M_1 such that x_1 returns a stored value. In other words, find a message expression x_3 in message expressions of the operand such that it is prefixed by x_1, i.e., $x_3 = x_2\, m\, x_4$ and $x_1 = x_2\, m$. Thus, among message expressions applicable to objects in the result of the one level project operation are all message expressions x such that $x = m\, x_4$.
>
> A new message is added to this resulting set of message expressions for every message expression x_2 present in the subset M_1 such that x_2 returns a derived value. The underlying method of each such new message solely returns the corresponding value in the derived new objects. To illustrate this, consider $\{name(), head()\ net\text{-}salary()\}$ as a subset of $M_e(department)$; the *one level projection* of the pair: $<T_{instances}(department), M_e(department)>$ on this subset returns the pair:
>
> $<\{<"Computer\ Science", 54K>\}, \{name(), m()\}>,$
>
> where $m()$ is a message which returns the stored value 54K when sent to the single object in the output. The message $m()$ is included in the set of message expressions of the output pair due to the message expression *head() net-salary()* which returns the derived net salary of the chairperson after deducting taxes at the rate of 0.1.

The purpose is to collect together in a new class all objects constructed by collecting the values reachable by the message expressions in M_1 applied to objects in the operand.

Despite the fact that many relationships between objects are represented by the objects themselves, an explicit operation is required to handle cases when a relationship is not defined in the model. Both the *cross-product* and the *nest* operations are defined

to introduce such relationships. While the cross-product operation is defined to be associative, the nest operation is not. However, the two operations are equivalent under certain conditions [14].* Associativity of the cross-product operation is useful in query optimization [12, 14], although not discussed in this thesis, being left for future research. The cross-product operation creates new objects, out of objects in the operands, and a set of message expressions is derived to handle the new objects. Also, the nest operation introduces missing relationships. We define such operators because it is not possible to have all the desired relationships predefined at the modeling phase. While the nest operation extends the value of each object in the first operand to include a reference to object(s) in the second operand†, the result of the cross-product operation depends on the nature of the domains of the messages of the operands as explicitly stated in definition 4.2 given next in section 4.2. For instance, nesting the *student* class and the *staff* class to assign to every student the chairperson of his department as a supervisor‡ results in the pair:

$$<\{<"Susan", 25, "F", \phi, 5, \{o_{11}, o_{16}\}, o_{10}, o_8>,$$
$$<"Tom", 18, "M", \phi, 3, \{o_{13}, o_{14}\}, o_{10}, o_8>,$$
$$<"George", 22, "M", \phi, 5, \{o_{11}, o_{16}\}, o_{10}, 15K, o_{10}, o_8>\},$$
$$M_e(student) \bigcup (m()\ M_e(staff))>.$$

Here, $m()$ is a new message understandable by the objects in the resulting pair to return the supervisor, o_8 in this example. Since the supervisor is himself a staff member, any of the message expressions in $M_e(staff)$ could be applied to the result of the message $m()$.

The cross-product of the the *student* class and the pair $<\{o_8\}, M_e(staff)>$ results in the pair: $<\{<o_5, o_8>, <o_7, o_8>, <o_9, o_8>\}, \{m_1(), m_2()\}>$,
where the result of the message $m_1()$ is drawn from the set $\{o_5, o_7, o_9\}$, i,e, the identity of a student object; while the message $m_2()$ has its result being the identity of a supervisor. However, the cross-product operation could have given the same result as the nest operation if all the message expressions of length one in $M_e(student)$, i.e.,

*the equivalence of some object algebra expressions are also presented in section 4.8

†Later in this chapter, it is shown that the result of the nest operation is a subclass of the first operand. Accordingly, objects of the first operand migrate into the result with their structure being extended to carry the new property due to the nest operation

‡Actually, there is a selection on the *staff* class before nesting

messages(student) were to return non-atomic values (object identities) and at least one of the message expressions of length one from $M_e(staff)$, i.e., *messages(staff)* were to return an atomic value. To illustrate a third case of the cross-product operation, assume that all the message expressions of length one from $M_e(staff)$, i.e., *messages(staff)* return non-atomic values (object identities) and at least one of the message expressions of length one from $M_e(student)$, i.e., *messages(student)* returns an atomic value. In this case, the resulting pair is

$$< \{< o_5, \text{"}Adams\text{"}, 40, \text{"}M\text{"}, \phi, 60K, o_{10} >, < o_7, \text{"}Adams\text{"}, 40, \text{"}M\text{"}, \phi, 60K, o_{10}, >,$$
$$< o_9, \text{"}Adams\text{"}, 40, \text{"}M\text{"}, \phi, 60K, o_{10} >\}, (m()\, M_e(student)) \bigcup M_e(staff) >,$$

where the message $m_1()$ returns the identity of a student object supervised by the receiving staff object. To illustrate the fourth case of the cross-product operation, assume that all the message expressions of length one from $M_e(staff)$ and $M_e(student)$, i.e., *messages(staff)* and *messages(student)*, respectively were to return non-atomic values (object identities). The cross-product operation under this condition would result in the pair:

$$< \{<\text{"}Susan\text{"}, 25, \text{"}F\text{"}, \phi, 5, \{o_{11}, o_{16}\}, o_{10}, \text{"}Adams\text{"}, 40, \text{"}M\text{"}, \phi, 60K, o_{10} >,$$
$$<\text{"}Tom\text{"}, 18, \text{"}M\text{"}, \phi, 3, \{o_{13}, o_{14}\}, o_{10}, \text{"}Adams\text{"}, 40, \text{"}M\text{"}, \phi, 60K, o_{10} >,$$
$$<\text{"}George\text{"}, 22, \text{"}M\text{"}, \phi, 5, \{o_{11}, o_{16}\}, o_{10}, 15K, o_{10}, \text{"}Adams\text{"}, 40, \text{"}M\text{"}, \phi,$$
$$60K, o_{10} >\}, \; M_e(student) \bigcup M_e(staff) >.$$

As mentioned before, the object algebra described in this work handles and produces a pair of sets, a set of objects and a set of message expressions to handle objects in the first set. So as it deals with sets, two basic set operations, *union* and *difference*, are supported by the object algebra; *intersection* is defined in terms of the difference operation, while the *symmetric difference* operation is defined in terms of the union, the difference and the intersection operations. The union operation returns a pair where the set of objects is in general heterogeneous and the set of message expressions is calculated as the intersection of the sets of message expressions of the operands. The heterogeneous set of objects is the union of the sets of objects of the operands. For instance, the union of the pairs:

$< T_{instances}(student),\, M_e(student) >$ and $< T_{instances}(staff),\, M_e(staff) >$

(corresponding to the *student* and *staff* classes, respectively) is the pair:

$< T_{instances}(staff) \bigcup T_{instances}(student),\, M_e(staff) \bigcap M_e(student) >$

which also could be represented as:

$< T_{instances}(person) - instances(person),\ M_e(person) >$.

The difference operation is handled in one of two ways depending on the relationship between the sets of message expressions of the operands. If the set of message expressions of the first operand is subset from that of the second operand, the difference operation returns objects from the first operand which are not in the second operand; message expressions of the result are those of the first operand. Otherwise, it is handled as a projection of objects in the first operand on values that have no corresponding message expression in the second operand. To illustrate the first case, the difference of the pairs:

$< T_{instances}(person),\ M_e(person) >$ and $< T_{instances}(student),\ M_e(student) >$

(corresponding to the *person* and *student* classes, respectively) is the pair:

$< instances(person),\ M_e(person) >$.

On the other hand, the second case is illustrated by considering the difference between the pairs: $< T_{instances}(research\text{-}assistant),\ M_e(research\text{-}assistant) >$ and $< T_{instances}(student),\ M_e(student) >$

(corresponding to the *research-assistant* and *student* classes, respectively) is the pair:

$< T_{instances}(research\text{-}assistant),\ M_e(staff) >$.

4.2 Object Algebra Expressions

After the informal description of the object algebra in section 4.1, we move into the formal definition. To start with, any item that has a pair of sets, a set of objects and a set of message expressions, is referred to as an *object algebra expression.* Since a class is defined to have a set of objects and a set of message expressions can be derived for a class by definition 3.5, a class corresponds to an object algebra expression. Next we formally define object algebra expressions. When speaking about len(x) in any of the constraints (if-statements) given in the rest of this thesis, we consider only message expressions x such that x returns a stored value with the underlying domain being an atomic domain d or 2^d for d being an atomic domain.

Definition 4.2 (Object Algebra Expressions)

Let E be the set of object algebra expressions.

Being an object algebra expression, every element of the set E must have a pair of sets -a set of objects and a set of message expressions. Thus, formally speaking,

$\forall e \in E$, $M_e(e)$ *is defined and* $T_{instances}(e)$ *is defined.*

Given $e_1 \in E$ *and* $e_2 \in E$;

let $M_e(e_1)=X_1$, $M_e(e_2)=X_2$, $T_{instances}(e_1)=T_1$, *and* $T_{instances}(e_2)=T_2$

Elements of E are enumerated as follows:

- Given a class c_i, by definition $M_e(c_i)$ and $T_{instances}(c_i)$ are both defined, *then* $c_i \in$ E

- *Selection*: Given a predicate expression p, $e_1[\mathrm{p}] \in$ E with

  $M_e(e_1[\mathrm{p}])=M_e(e_1)=X_1$

  $T_{instances}(e_1[\mathrm{p}])=\{\mathrm{o} \mid \mathrm{o} \in T_1 \wedge \mathrm{p(o)}$ *$\}$

- *Projection*: Given $\mathrm{X} \subseteq X_1$, $e_1[\mathrm{X}] \in$ E with

  $M_e(e_1[\mathrm{X}])=\mathrm{X}$

  $T_{instances}(e_1[\mathrm{X}])=T_{instances}(e_1)$

  The project operation serves to drop the message expressions that are in $X_1 - X$ while preserving all the instances.

- *Cross-Product*: $(e_1 \times e_2) \in$ E with,

$$M_e(e_1 \times e_2) = \begin{cases} (m_1 X_1) \cup (m_2 X_2) & \text{if } \exists x_i \in X_1, \mathrm{len}(x_i)=1 \wedge \exists x_j \in X_2, \mathrm{len}(x_j)=1 \\ X_1 \cup (m_2 X_2) & \text{if } \forall x_i \in X_1, \mathrm{len}(x_i)>1 \wedge \exists x_j \in X_2, \mathrm{len}(x_j)=1 \\ (m_1 X_1) \cup X_2 & \text{if } \exists x_i \in X_1, \mathrm{len}(x_i)=1 \wedge \forall x_j \in X_2, \mathrm{len}(x_j)>1 \\ X_1 \cup X_2 & \text{if } \forall x_i \in X_1, \mathrm{len}(x_i)>1 \wedge \forall x_j \in X_2, \mathrm{len}(x_j)>1 \end{cases}$$

where m_1 is message applicable to objects in the result of the first and third cases of the definition. Sending m_1 to any such object returns the identity of the related object from T_1. On the other hand, m_2 is a message applicable to objects in the result of the first and the second cases of the definition. On sending m_2 to any such object the identity of the related object from T_2 is returned. In other

*Given an object o, we use $p(o)$ to denote the evaluation of predicate expression p by o substituting for an object variable in p.

words, m_1 and m_2 are two messages returning values (object identities) drawn from the domains T_1 and T_2, respectively.

$$T_{instances}(e_1 \times e_2) = \begin{cases} \{o \mid \exists o_1 \in T_1\ \exists o_2 \in T_2 \wedge value(o) = identity(o_1).identity(o_2)\} \\ \qquad \text{if } \exists x_i \in X_1, \text{len}(x_i)=1 \wedge \exists x_j \in X_2, \text{len}(x_j)=1 \\ \{o \mid \exists o_1 \in T_1\ \exists o_2 \in T_2 \wedge value(o) = value(o_1).identity(o_2)\} \\ \qquad \text{if } \forall x_i \in X_1, \text{len}(x_i)>1 \wedge \exists x_j \in X_2, \text{len}(x_j)=1 \\ \{o \mid \exists o_1 \in T_1\ \exists o_2 \in T_2 \wedge value(o) = identity(o_1).value(o_2)\} \\ \qquad \text{if } \exists x_i \in X_1, \text{len}(x_i)=1 \wedge \forall x_j \in X_2, \text{len}(x_j)>1 \\ \{o \mid \exists o_1 \in T_1\ \exists o_2 \in T_2 \wedge value(o) = value(o_1).value(o_2)\} \\ \qquad \text{if } \forall x_i \in X_1, \text{len}(x_i)>1 \wedge \forall x_j \in X_2, \text{len}(x_j)>1 \end{cases}$$

where . is being used to indicate a concatenation of the two arguments; it is commutative because the resulting value is actually a set of values constructed out of the values constituting the two arguments. It is also necessary to indicate that the four cases of the cross-product operation are defined to have the operation associative and to prevent the migration of objects including atomic values. This is important because while non-atomic values are drawn solely from object identities and all have the same length, an atomic value may be drawn from any of the atomic domains and hence the length may be long enough to make it inefficient to move an object with such an atomic value; instead it is referenced using its object identity.

- *Union*: $(e_1 \cup e_2) \in \text{E}$ with

  $M_e(e_1 \cup e_2) = X_1 \cap X_2$

  $T_{instances}(e_1 \cup e_2) = T_1 \cup T_2$

  Only message expressions in $X_1 \cap X_2$ return some values, including *nil* from objects in $T_1 \bigcup T_2$. However, message expressions in the symmetric difference of X_1 and X_2, i.e., $X_1 \Delta\ X_2$ which is interpreted as $(X_1 \bigcup\ X_2) - (X_1 \bigcap\ X_2)$, should return the value *"undefined"* for a message expression not in the set of message expressions forming the same pair with the set of objects from which the

receiving object is chosen. For instance, objects in T_1 answer by *"undefined"* to message expressions in $(X_1 \Delta\ X_2) - X_1$, and objects in T_2 answer by *"undefined"* to message expressions in $(X_1 \Delta\ X_2) - X_2$.

- *Difference*: $(e_1 - e_2) \in$ E with

$$M_e(e_1 - e_2) = \begin{cases} X_1 & \text{if } X_1 \subseteq X_2 \text{ (by lemma 3.1)} \\ X_1 - X_2 & \text{otherwise} \end{cases}$$

$$T_{instances}(e_1 - e_2) = \begin{cases} T_1 - T_2 & \text{if } X_1 \subseteq X_2 \\ T_1 & \text{otherwise} \end{cases}$$

- *Nest*: $(e_1 >> e_2) \in$ E with
  $M_e(e_1 >> e_2) = X_1 \cup (m_2\ X_2)$
  where m_2 is a message applicable to objects in the result to return the object identity of the related object in T_2. In other words, the value returned by the message m_2 from any object in the result is drawn from the domain T_2.

  $T_{instances}(e_1 >> e_2) = \{o \mid \exists o_1 \in T_1 \wedge \text{value(o)} = \text{value}(o_1).o_2$, where $o_2 = (o\ m_2) \wedge o_2 \in T_2\}$

- *One level projection*: Given $X \subseteq X_1$, $e_1![X] \in$ E with

  $M_e(e_1![X]) = \{x \mid \exists x_1 \in X$ with x_1 returning a stored value, $x_1 = (x_2\ m) \wedge len(x_1) = len(x_2) + 1 \wedge \exists x_3 \in X_1 \wedge x_3 = (x_2\ x) \wedge x = (m\ x_4)\}$
  $\bigcup \{x \mid \exists x_1 \in X$ with x_1 returning a derived value, len(x)=1 $\wedge$ $\forall o_1 \in T_{instances}(e_1) \exists o \in T_{instances}(e_1![X])$ such that $o_1\ x_1 = o\ x\}$

  $T_{instances}(e_1![X]) = \{o \mid \exists o_1 \in T_1 \wedge value(o) = (o_1\ X)^{\dagger}\}$

  The depth of nesting decreases as the length of the longest message expression in X increases. In other words, the depth of nesting is inversely proportional to the length of message expressions in X.

$^{\dagger}(o_1\ X)$ returns the set of the results of the application of elements of X to o_1.

- *Aggregation*: Given $X \subseteq X_1$ and $x_i \in X_1$, e_1<X,f,x_i>$\in$E with $M_e(e_1<X,f,x_i>)=(m_1\ X_1)\cup\{m_3\}$, where m_1 is a message which when sent to any object in the result returns the object identity of the related object in T_1; m_3 is a message which returns from any object in the result the value obtained from the application of the function f to objects in T_1 which are giving the same result for the message expressions in X. In other words, the value returned by the message m_3 from any object in the result has its domain being the range of the function f.

  $$T_{instances}(e_1<X,f,x_i>)=\{o \mid (o\ m_1)\subseteq T_1 \wedge (o\ m_3)=f(\{(o_1\ x_i) \mid o_1\in T_1 \wedge \forall o_2\in(o\ m_1), (o_2\ X)=(o_1\ X)\})\}$$

  The aggregation function is applied on e_1 by evaluating the function f on the result of the message expression x_i for all objects that return the same values for elements of the set of message expressions X. In other words, objects in T_1 are partitioned into equivalence classes[‡] based on the result of the evaluation of message expressions in X against those objects. Then, the aggregate function f is applied to objects in each of the equivalence classes by considering the value returned by the message expression x_i applied to each such object.

- *Unnest*: defined in terms of projection as,

  $$(e_1<<e_2)=e_1[X_1-X \mid X=(m_2\ X_2) \wedge \forall o_1\in T_1, (o_1\ m_2)\in T_2]$$

  We project on all message expressions of e_1 except those leading to e_2.

- *Intersection*: defined in terms of the difference operation as,

  $$(e_1\cap e_2)=e_1-(e_1-e_2)$$

- *Inverse project*: to add a subset X of $M_e(e_2)$ to $M_e(e_1)$, first e_1 and e_2 are nested then a one level projection is done to have all $M_e(e_2)$ and $M_e(e_1)$ together forming one set; after that projection of the result on $M_e(e_1)\bigcup X$ is done to get the target set of message expressions in the resulting pair.

  $$e_1]e_2 : X[=(e_1 >> e_2)\ ![messages(e_1)\bigcup(m_2\ messages(e_2))]\ [M_e(e_1)\bigcup X]$$

  where $X \subseteq M_e(e_2)$ is the set of message expressions to be added to $M_e(e_1)$, and m_2 is a message in the result of $e_1 >> e_2$ with its domain being $T_{instances}(e_2)$.

[‡]An equivalence class is a set of objects having common characteristics such that every two equivalence classes are disjoint, i.e., given any two equivalence classes A and B, $A\bigcap B=\phi$.

Notice that the one level project operation results in a pair which contains $M_e(e_1) \bigcup M_e(e_2)$. So, the project operation is used to get the required message expressions in the result.

The already described formulation of the inverse of the project operation is valid for the case of adding some existing methods to a class. However, for the case of having X consisting of new methods $X = \{m_1 : f_1, m_2 : f_2, ..., m_n : f_n\}$, (where $m_i : f_i$ specifies the message m_i to be used to invoke the method that implements the function f_i) the definition is changed to: $e_1]X[\in E$ with,

$M_e(e_1]X[) = M_e(e_1) \bigcup \{m_1, m_2, ..., m_n\}$

$T_{instances}(e_1]X[) = T_{instances}(e_1)$

- *Join*: defined in terms of cross-product or nest combined with selection,

  $e_1 <p> e_2 = e_1 \times e_2 [p] = e_1 >> e_2 [p]$. □

Using operations of the query language, objects may be constructed out of existing ones and new relationships may be introduced into the model. A new relationship is an extension to either the state of objects or their behavior. In other words, a new relationship has either a stored or a derived value. A stored value is due to the nest operation which takes two operands and extends each object in the first to include a value referencing object(s) in the second operand, while a derived value is due to the inverse of the project operation which extends the behavior of objects in the operand without their states being affected. On the other hand, the one-level-project operation constructs new objects out of existing objects by collecting values found at different levels of nesting. Also the fourth case in the definition of the cross-product operation results in new objects, while other cases introduce new relationships.

4.3 From an Object Algebra Expression to a Class

After the formal definition of object algebra expressions, we claim that every object algebra expression has the characteristics of a class and this follows from the lemmata given next in this section. However, before going into the details of the lemmata, it is important to recall that, as stated in chapter 3, by definition a class has a set of superclasses, a set of instance variables, a set of methods and a set of objects. According to definition 4.2, an object algebra expression has a set of objects and a

set of message expressions. In addition, given a class c, methods(c) and $I_{variables}(c)$ are defined to include methods and instance variables of superclasses of class c. Therefore, finding methods and instance variables of a class implicitly leads to the set of its superclasses. Furthermore, a method implements a specific function and it is invoked via a corresponding message. Consequently, for every method there exists a corresponding message; so, finding a set of messages for an object algebra expression is equivalent to finding a set of methods; the corresponding method of a given message m is the one which is invoked by the message m when used from the current class according to the priority rules given in chapter 5. As a result, for any object algebra expression to have the characteristics of a class, it is enough to find for that object algebra expression a set of instance variables and a set of messages; a set of objects is already defined. The set of superclasses is explicitly determined in section 4.4.

Let e_1 and e_2 be two object algebra expressions such that $M_e(e_1) = X_1$ and $M_e(e_2) = X_2$. According to definition 4.2, a class is an object algebra expression. In other words, some object algebra expressions are classes. Thus, assume that $I_{variables}(e_1)$, $I_{variables}(e_2)$, messages(e_1) and messages(e_2) are all defined. Based on this assumption, we have the following lemmata, 4.1 to 4.9, leading to the sets of messages and instance variables of other object algebra expressions and this leads to the fact that every object algebra expression corresponds to a class. Lemmata 4.1 to 4.9 have similar proofs. So only the proof of lemma 4.1 is given, others can be proved by the same way.

Lemma 4.1 *Messages and Instance variables of $e_1[p]$: where p is a predicate expression*

$M_e(e_1[p])=X_1 \Longrightarrow$

- *messages($e_1[p]$)=messages(e_1)*
- $I_{variables}(e_1[p]) = I_{variables}(e_1)$

Proof:

$m \in$ *messages*$(e_1[p]) \Longleftrightarrow \exists\, x \in M_e(e_1[p])$, *such that* $x = m\ x_i$ *(by definition 3.5)*

(but $M_e(e_1[p]) = M_e(e_1)$*, by definition 4.2)*

$\Longleftrightarrow \exists\, x \in M_e(e_1)$, *such that* $x = m\ x_i$

$\Longleftrightarrow m \in$ *messages*(e_1) *(by definition 3.5)*

Therefore, $m \in$ *messages*$(e_1[p]) \Longleftrightarrow m \in$ *messages*(e_1)

Hence, *messages*$(e_1[p])$ = *messages*(e_1)

and

$iv \in I_{variables}(e_1[p]) \iff \exists\, m \in messages(e_1[p])$ *such that,*
given $o \in T_{instances}(e_1[p])$, o *satisfies* $(o\ m) = value(iv)$
(but $messages(e_1[p]) = messages(e_1)$, *already proved*
and $T_{instances}(e_1[p]) \subseteq T_{instances}(e_1)$ *by definition 4.2)*
$\iff iv \in I_{variables}(e_1)$

Therefore, $iv \in I_{variables}(e_1[p]) \iff iv \in I_{variables}(e_1)$

Hence, $I_{variables}(e_1[p]) = I_{variables}(e_1)$ □

Before going into the details of the lemma that leads to the messages and instance variables of $e_1[X]$, consider the following algorithm that derives the instance variables of $e_1[X]$.

Algorithm 4.1 *Instance variables of* $e_1[X]$*:*

0. for every $m_i \in messages(e_1)$
1. Let $X_i \subseteq M_E{}^*$ *such that* $(m_i\ X_i) \subseteq X$
2. if $X_i \neq \phi$ *then*
/ the instance variable that corresponds to m has non-atomic domain*
3. if $\exists iv_i \in I_{variables}(e_1)$ *such that* $X_i = M_e(OAE(domain(iv_i)))^\dagger$ *then*
4. $iv_i \in I_{variables}(e_1[X])$
5. elseif $\exists iv_i \in I_{variables}(e_1)$ *such that* $X_i \subseteq M_e(OAE(domain(iv_i)))$ *then*
6. $iv_i \in I_{variables}(e_1[X])$ *and* $domain(iv_i)$ *in* $e_1[X]$ *is:*
7. $domain(iv_i) := T_{instances}(< domain(iv_i), M_e(OAE(domain(iv_i))) > [X_i])$
8. endif
9. elseif $\exists iv_i \in I_{variables}(e_1)$ *such that given* $o \in T_{instances}(e_1)$, $value(iv_i, o)^\ddagger = (o\ m_i)$ *then*
/ the message* m_i *corresponds to the instance variable* iv_i *which has an atomic domain*
10. $iv_i \in I_{variables}(e_1[X])$
11. endif
12. endfor

*Set of all message expressions, i.e., for any class c, $M_e(c) \subseteq M_E$

†Evaluating an object algebra expression e leads to the pair $< T_{instances}(e), M_e(e) >$ and OAE($T_{instances}(e)$) denotes the object algebra expression e.

‡returns the value of the instance variable iv_i in object o

Lemma 4.2 *Algorithm 4.1 returns instance variables of* $e_1[X]$

Proof:

The *for-loop* in step 0 of the algorithm 4.1 iterates over all the messages in $messages(e_1)$ to determine those corresponding to instance variables in $I_{variables}(e_1)$. When such a message m_i is found, the corresponding instance variable iv_i is added to $I_{variables}(e_1[X])$ with its domain specified depending on the results of the *if-statements* where:
In step 1, from X a subset that has all message expressions starting with m_i, i.e., $m_i\ X_i$ is determined. The tag X_i is considered in step 2 for being non-empty, accordingly a non-atomic domain is determined for the instance variable iv_i added to $I_{variables}(e_1[X])$. In step 4, iv_i in $I_{variables}(e_1[X])$ has its domain the same as iv_i in $I_{variables}(e_1)$, because X_i happened to be $M_e(OAE(domain(iv_i)))$ [§]. For the other case, i.e., $X_i \subseteq M_e(OAE(domain(iv_i)))$, in steps 6 and 7, the domain of iv_i is specified as the result of the projection of $< domain(iv_i), M_e(OAE(domain(iv_i))) >$ on X_i, a subset of its message expressions. Since the class which corresponds to the result of the projection is proved, later in lemma 4.11 of section 4.4, to be a superclass of that which corresponds to the operand, the domain of iv_i in $I_{variables}(e_1[X])$ is a superclass of the domain of iv_i in $I_{variables}(e_1)$ which satisfies definition 3.4. In step 10, the domain of iv_i in $I_{variables}(e_1[X])$ is determined to be the same as the domain of iv_i in $I_{variables}(e_1)$ for the case of $domain(iv_i)$ being atomic. □

Theorem 4.1 *Algorithm 4.1 determines* $I_{variables}(e_1[X])$ *in time* $O(n + \sum_{l=1}^{p} k_l)$ *where n is* $card(messages(e_1))$, *p is the number of executions of step 7 of the algorithm and* k_l *is* $card(messages(e_j))$ *with* e_j *being the object algebra expression used as the operand in step 7 of the algorithm.*

Proof:

The for-loop of step 0 is performed n times and there are three cases to consider concerning the statements in the body of that for-loop,

1. In case that the conditions of steps 2 and 3 hold true, only step 4 could be performed.

2. In case that the condition of step 9 holds true, only step 10 is performed.

[§]In the expression $M_e(OAE(domain(iv_i)))$, iv_i in $I_{variables}(e_1)$ is considered

3. In case that the conditions of steps 2 and 5 hold true, steps 6 and 7 are performed.

The first two cases do not have any side effect and therefore without having the condition of step 5 holding true in any of the n executions of the for-loop, the algorithm executes in time $O(n)$.

However, if the condition of step 5 holds true for one of the n executions of the for-loop, the algorithm executes in time $O(n+k_i)$, where k_i is the number of messages in the set of messages of the object algebra expression used as the operand in the projection of step 7. By the same way, having the condition of step 5 holding true p times during any of the executions of the for-loop (including subsequent executions) leads to have the algorithm performing in $O(n+k_1+k_2+...+k_p)$, i.e., $O(n+\sum_{l=1}^{p} k_l)$ □

Lemma 4.3 *Messages and Instance variables of* $e_1[X]$*: Given* $X \subseteq X_1$,

. $messages(e_1[X])=\{m \mid m \in messages(e_1) \wedge \exists x \in X \text{ with } x=m\ x_i\}$

. $I_{variables}(e_1[X])$ *is derived by algorithm 4.1 as interpreted in lemma 4.2.* □

Lemma 4.4 *Messages and Instance variables of* $e_1 \times e_2$:

case 1: - if $\exists x_1 \in X_1$, $len(x_1)=1 \wedge \exists x_2 \in X_2$, $len(x_2)=1$ *then*

$M_e(e_1 \times e_2)=(m_1\ X_1) \bigcup (m_2\ X_2) \Longrightarrow$

. $messages(e_1 \times e_2)=\{m_1, m_2\}$

. $I_{variables}(e_1 \times e_2)=\{iv_1, iv_2\}$,

where $domain(iv_1)=2^{T_{instances}(e_1)}$ *and* $domain(iv_2)=2^{T_{instances}(e_2)}$

case 2: - if $\forall x_1 \in X_1$, $len(x_1)>1 \wedge \exists x_2 \in X_2$, $len(x_2)=1$ *then*

$M_e(e_1 \times e_2)=X_1 \bigcup (m_2\ X_2) \Longrightarrow$

. $messages(e_1 \times e_2)=messages(e_1) \bigcup \{m_2\}$

. $I_{variables}(e_1 \times e_2)=I_{variables}(e_1) \bigcup \{iv_2\}$,

where $domain(iv_2)=2^{T_{instances}(e_2)}$.

case 3: - if $\exists x_1 \in X_1$, $len(x_1)=1 \wedge \forall x_2 \in X_2$, $len(x_2)>1$ *then*

$M_e(e_1 \times e_2)=(m_1\ X_1) \bigcup X_2 \Longrightarrow$

. $messages(e_1 \times e_2)=\{m_1\} \bigcup messages(e_2)$

. $I_{variables}(e_1 \times e_2)=\{iv_1\} \bigcup I_{variables}(e_2)$,

where $domain(iv_1)=2^{T_{instances}(e_1)}$.

case 4: - if $\forall x_1 \in X_1$, $len(x_1) > 1 \wedge \forall x_2 \in X_2$, $len(x_2) > 1$ *then*

$M_e(e_1 \times e_2) = X_1 \bigcup X_2 \Longrightarrow$

. $messages(e_1 \times e_2) = messages(e_1) \bigcup messages(e_2)$

. $I_{variables}(e_1 \times e_2) = I_{variables}(e_1) \bigcup I_{variables}(e_2)$ □

Lemma 4.5 *Messages and Instance variables of* $e_1 \bigcup e_2$*:*

$M_e(e_1 \bigcup e_2) = X_1 \bigcap X_2 \Longrightarrow$

. $messages(e_1 \bigcup e_2) = messages(e_1) \bigcap messages(e_2)$

. $I_{variables}(e_1 \bigcup e_2) = I_{variables}(e_1) \bigcap I_{variables}(e_2)$ □

Lemma 4.6 *Messages and Instance variables of* $e_1 - e_2$*:*

case 1: - if $X_1 \subseteq X_2$ *then*

$M_e(e_1 - e_2) = X_1 \Longrightarrow$

. $messages(e_1 - e_2) = messages(e_1)$

. $I_{variables}(e_1 - e_2) = I_{variables}(e_1)$

case 2: - if $X_1 \not\subseteq X_2$ *then*

$M_e(e_1 - e_2) = X_1 - X_2 \Longrightarrow$

. $messages(e_1 - e_2) = messages(e_1) - messages(e_2)$

. $I_{variables}(e_1 - e_2) = I_{variables}(e_1) - I_{variables}(e_2)$ □

Lemma 4.7 *Messages and Instance variables of* $e_1 >> e_2$*:*

$M_e(e_1 >> e_2) = X_1 \bigcup (m_2\ X_2) \Rightarrow$

. $messages(e_1 >> e_2) = messages(e_1) \bigcup \{m_2\}$

. $I_{variables}(e_1 >> e_2) = I_{variables}(e_1) \bigcup \{iv_1\}$, *where* $domain(iv_1) = 2^{T_{instances}(e_2)}$.

□

Lemma 4.8 *Messages and Instance variables of* $e_1![X]$*: given* $X \subseteq X_1$,

$M_e(e_1![X])$ *given in definition 4.2* $\Longrightarrow$

. $messages(e_1![X]) = \{m \mid \exists x \in M_e(e_1![X])$ *with* $x = m\ x_j\}$

. $I_{variables}(e_1![X]) = \{iv \mid domain(iv) = d \wedge \forall o \in T_{instances}(e_1![X])$

$\exists m \in messages(e_1![X])$ *with* $(o\ m) \in d\}$ □

Lemma 4.9 *Messages and Instance variables of* $e_1 < X, f, x_i >$*: given* $X \subseteq X_1$ *and* $x_i \in X_1$,

$M_e(e_1 < X, f, x_i >)$ *given in definition 4.2* $\Longrightarrow$

- $messages(e_1 < X, f, x_i >) = \{m_1, m_3\}$
- $I_{variables}(e_1 < X, f, x_i >) = \{iv_1, iv_2\}$

where $domain(iv_1) = T_{instances}(e_1)$ *and* $domain(iv_2)$*= the domain of the result of* f □

The proofs of lemmata 4.3 to 4.9 are omitted as they are similar to the proof of lemma 4.1. Informally, since every object algebra expression has a set of message expressions, then by considering message expressions of length one, the set of messages is derived. Furthermore, the fact that by definition every instance variable has a corresponding message leads to the derivation of the set of instance variables of an object algebra expression. This is done by collecting from the operand those instance variables with a corresponding message in the already derived set of messages.

Combining definition 4.2 and lemmata 4.1 to 4.9, every object algebra expression has a set of objects, a set of messages and a set of instance variables; the set of superclasses of the corresponding class is determined by lemmata 4.10 to 4.17 given next in section 4.4. The set of messages leads to the set of methods because every message has a corresponding method. Therefore, an object algebra expression has the characteristics of a class leading to the following corollary.

Corollary 4.1 $\forall e \in E$, *e corresponds to a class c.*

Proof:

$T_{instances}(e)$ are given by definition 4.2;

$I_{variables}(e)$ and $messages(e)$ are given by lemmata 4.1 to 4.9;

$supers(e)$ are determined by lemmata 4.10 to 4.17 and algorithm 4.2.

Therefore, having the characteristics of a class, e is in fact a class c. □

4.4 Maximizing Reusability

One of the distinguishing features of object-oriented systems is inheritance. Inheritance leads to reusability where a class c_1 uses the facilities of its superclass c_2 as if those facilities were defined within the class c_1.

In general, object-oriented database systems support multiple inheritance. However, still there exist systems supporting single inheritance, e.g. [69]. Multiple inheritance is considered advantageous over and covers simple inheritance. In multiple inheritance, reusability is achieved to a greater degree than in simple inheritance. So, supporting multiple inheritance in a data model helps in increasing reusability by having the flexibility of increasing the number of superclasses if required in a way to increase the facilities that a class c inherits and hence decrease the facilities defined inside the class c without being inherited. This is one of the reasons behind supporting multiple inheritance in the data model described in chapter 3. Later in this section, an algorithm which adjusts the set of superclasses of a given class aiming at maximizing reusability is introduced.

Let I and L be the inherited and non-inherited (locally defined) facilities of a class, respectively. Facilities of a class include the structure (instance variables) and behavior (messages) defined for its objects. The total facilities of a class are $I \bigcup L$. Our aim is to maximize I and hence minimize L, as the sizes of the two sets are inversely proportional, i.e., as one increases the other decreases and vice versa. For that purpose, we change the classes found in the set of superclasses of a class to include classes that maximizes I. To do that, the inheritance relationship between *object algebra expressions* is defined first. Based on that relationship, we prove that an object algebra expression e inheriting from other object algebra expressions $e_1, e_2, ..., e_n$, leads to have the corresponding class c to inherit from classes $c_1, c_2, ..., c_n$. This completes the proof of corollary 4.1 where $supers(c)$ are determined due to lemmata 4.10 to 4.17 and theorem 4.2 given next in this section.

Definition 4.3 (Partial Ordering ($\leq_e$) among object algebra expressions) *Given two object algebra expressions e_1 and e_2, we say that e_1 inherits from e_2, i.e., $e_1 \leq_e e_2$ iff:*

1. $M_e(e_2) \subseteq M_e(e_1)$
2. $T_{instances}(e_1) \subseteq T_{instances}(e_2)$ □

Notice that definition 4.3 considers only message expressions and total instances which are the characteristics known for an object algebra expression by definition 4.2. Other characteristics leading to a class and given in lemmata 4.1 to 4.9 are not considered.

After having every object algebra expression to be corresponding to a class

by corollary 4.1, it is necessary to decide on the inheritance relationship between the class that corresponds to a given object algebra expression and other existing classes. It is enough to decide on the inheritance relationship between object algebra expressions, because theorem 4.2 leads to the inheritance relationship between the corresponding classes. Based on definition 4.3, the following lemmata 4.10 to 4.17 lead to the inheritance relationship between object algebra expressions. Based on such relationship, the inheritance relationship between the corresponding classes is determined.

Given two object algebra expressions e_1 and e_2; let $M_e(e_1)=X_1$ and $M_e(e_2)=X_2$. Lemmata 4.10 to 4.17 give the inheritance relationship between object algebra expressions. This way is followed because the proofs of lemmata 4.10 to 4.17 are straightforward following from definitions 4.2 and 4.3, as illustrated by sketching the proofs of lemmata 4.10 and 4.11. From lemmata 4.10 to 4.17 and based on theorem 4.2, supers(c) for the class c which corresponds to a given object algebra expression e could be determined and later adjusted by algorithms 4.2 where reusability is maximized.

Lemma 4.10 *Inheritance relationship of* $e_1[p]$ *with* e_1, *where* p *is a predicate expression,*

$$e_1[p] \leq_e e_1$$

Proof: *(By definition)*

$M_e(e_1[p]) = M_e(e_1)$ *and* $T_{instances}(e_1[p]) \subseteq T_{instances}(e_1)$ *(by definition 4.2)*

$\Longleftrightarrow e_1[p] \leq_e e_1$ *(by definition 4.3)* □

Lemma 4.11 *Inheritance relationship of* $e_1[X]$ *with* e_1, *where* $X \subseteq X_1$,

$$e_1 \leq_e e_1[X].$$

Proof: *(By definition)*

$M_e(e_1[X]) \subseteq M_e(e_1)$ *and* $T_{instances}(e_1[X]) = T_{instances}(e_1)$ *(by definition 4.2)*

$\Longleftrightarrow e_1 \leq_e e_1[X]$ *(by definition 4.3)* □

Lemma 4.12 *Inheritance relationship of* $e_1 \times e_2$ *with* e_1 *and* c_2*:*

case 1: - if $\exists x_1 \in X_1$, $len(x_1)=1 \wedge \exists x_2 \in X_2$, $len(x_2)=1$ *then*

$$(e_1 \times e_2) \not\leq_e e_1 \text{ and } (e_1 \times e_2) \not\leq_e e_2$$

case 2: - if $\forall x_1 \in X_1$, $len(x_1)>1 \wedge \exists x_2 \in X_2$, $len(x_2)=1$ *then*

$$(e_1 \times e_2) \leq_e e_1$$

case 3: - if $\exists x_1 \in X_1$, $len(x_1)=1 \wedge \forall x_2 \in X_2$, $len(x_2)>1$ *then*

$$(e_1 \times e_2) \leq_e e_2$$

case 4: - if $\forall x_1 \in X_1$, $len(x_1)>1 \wedge \forall x_2 \in X_2$, $len(x_2)>1$ *then*

$$(e_1 \times e_2) \leq_e e_1 \text{ and } (e_1 \times e_2) \leq_e e_2$$ □

Lemma 4.13 *Inheritance relationship of* $e_1 \bigcup e_2$ *with* e_1 *and* e_2*:*

$$e_1 \leq_e (e_1 \bigcup e_2) \text{ and } e_2 \leq_e (e_1 \bigcup e_2).$$ □

Lemma 4.14 *Inheritance relationship of* $e_1 - e_2$ *with* e_1 *and* e_2*:*

case 1: - if $X_1 \subseteq X_2$ *then*

$$(e_1 - e_2) \leq_e e_1$$

case 2: - if $X_1 \not\subseteq X_2$ *then*

$$e_1 \leq_e (e_1 - e_2)$$ □

Lemma 4.15 *Inheritance relationship of* $(e_1 >> e_2)$ *with* e_1*,*

$$(e_1 >> e_2) \leq_e e_1$$ □

Lemma 4.16 *Inheritance relationship of* $e_1![X]$ *with* e_1*, where* $X \subseteq X_1$*,*

$$e_1![X] \not\leq_e e_1 \text{ and } e_1 \not\leq_e e_1![X].$$ □

Lemma 4.17 *Inheritance relationship of* $e_1 < X, f, x_i >$ *with* e_1*, where* $X \subseteq X_1$ *and* $x_i \in X_1$

$$e_1 < X, f, x_i > \not\leq_e e_1 \text{ and } e_1 \not\leq_e e_1 < X, f, x_i >$$ □

Although omitted, the proofs of lemmata 4.12 to 4.17 follow from definitions 4.2 and 4.3 by the same way as the proofs of lemmata 4.10 to 4.11 are done. After deciding on the inheritance relationship between object algebra expressions, theorem 4.2 leads to the inheritance relationship among the corresponding classes. First the OBJECT class is assumed to be the direct superclass of any class c ($supers(c) = \phi$) that corresponds to an object algebra expression resulting from a query. Then other user defined classes are included in $supers(c)$ by applying the appropriate lemma out of lemmata 4.10 to 4.17 followed by the application of theorem 4.2. In other words, given an object algebra expression, at the start I of the corresponding class is assumed empty. However, the following theorem is considered for a first step towards maximizing I and

hence minimizing L by deriving the inheritance relationship between classes based on the inheritance relationship between the corresponding object algebra expressions, according to lemmata 4.10 to 4.17.

Theorem 4.2 *Let e_1 and e_2 be two object algebra expressions with c_1 and c_2 being their corresponding classes by corollary 4.1, respectively.*

$$e_1 \leq_e e_2 \iff c_1 \leq_c c_2$$

Proof:

First of all lemmata 4.1 to 4.9 give $messages(c_1)$ and $messages(c_2)$.

$$e_1 \leq_e e_2 \iff M_e(e_2) \subseteq M_e(e_1) \text{ and } T_{instances}(e_1) \subseteq T_{instances}(e_2)$$
(by definition 4.3)
$$\iff messages(c_2) \subseteq messages(c_1) \text{ (by lemma 3.1)}$$

and

$$iv \in I_{variables}(c_2) \iff \exists\, m \in messages(c_2) \text{ such that given } o \in T_{instances}(c_2),$$
$$o\ satisfies\ o\ m = value(iv)$$
(but $messages(c_2) \subseteq messages(c_1)$, already proved)
$$\implies iv \in I_{variables}(c_1)$$
$$\implies I_{variables}(c_2) \subseteq I_{variables}(c_1)$$

Therefore, $e_1 \leq_e e_2 \iff messages(c_2) \subseteq messages(c_1) \wedge I_{variables}(c_2) \subseteq I_{variables}(c_1)$

Hence, $c_1 \leq_c c_2$ □

Theorem 4.2 determines if any classes should constitute $supers(c)$ for a given class c which corresponds to an object algebra expression e. This is considered as a first step towards the maximization of I. The maximization of I and hence the minimization of L continues by considering the classes which are either direct or indirect superclasses of the class c which corresponds to an operand in the object algebra expression e due to corollary 4.1, as candidates for inclusion in $supers(c_1)$, where c_1 is the class corresponding to e. For that purpose, the *Inheritance List* of the class c, denoted $IL(c)$, is defined and used to maximize I by algorithm 4.2 given next in this section. The maximization of I makes sense for the case of having $e_1 \leq e_2$, where e_1 and e_2 are an operand in and the result of an object algebra expression, respectively. It is intended to have the $IL(c)$ ordered from the OBJECT class towards the class c, i.e., direct subclasses of the OBJECT class (on a path from the class c to the OBJECT class)

should be near the head of the list and direct superclasses of the class c ($supers(c)$) should be near the tail of the list.

Definition 4.4 (Inheritance List) *Given a class c, $IL(c)$ is defined to include:*

1. $supers(c) \subseteq IL(c)$

2. $\forall\, c_i \in IL(c)$, *we have* $supers(c_i) \subseteq IL(c)$,

*$supers(c_i)$ are placed at the head of $IL(c)$** □

The motivation behind having $IL(c)$ being ordered becomes clear after algorithm 4.2, where it is important to start with classes near the OBJECT class while checking classes for inclusion in the set of superclasses of class c_1 (the result of an operation on c). This is so because the reusability increases as one moves from the OBJECT class down in the lattice, i.e., as the number of classes on a path from the OBJECT class to a given class increases. This is proved in theorem 4.3, given next.

Theorem 4.3 *Reusability increases as the length of a path from the OBJECT class to a given class increases.*

Proof:

Consider two classes c_1 and c_2 such that $I_{c_1} = I_{c_2}$ and $L_{c_2} \subseteq L_{c_1}$;

c_1 and c_2 are direct subclasses of the same classes, i.e., $supers(c_1) = supers(c_2)$. Since $supers(c_1) = supers(c_2)$, the number of paths from c_1 to the OBJECT class is equal to the number of paths from c_2 to the OBJECT class because every such path passes by a superclass. Also, for every path from c_1 to the OBJECT class, there is a path with the same length from c_2 to the object class. Let c_i be a common superclass with l being the length of a path from c_i to the OBJECT class. Then $l + 1$ is the length of the paths from c_1 and c_2 to the OBJECT class passing via c_i.

Since $L_{c_2} \subseteq L_{c_1}$, the following can be done to have $c_1 \leq_c c_2$:

I_{c_1} is maximized by including c_2 in $supers(c_1)$ and L_{c_1} is set to $L_{c_1} - L_{c_2}$.

By including c_2, $supers(c_2)$ are eliminated from $supers(c_1)$ as they become indirect superclasses of c_1. So, an increase in reusability has been achieved by having the length of the path from c_1 to the OBJECT class being increased after including c_2. □

*order of classes in $IL(c)$ is important, since in algorithm 4.2 classes near to the OBJECT class are going to be considered first.

Algorithm 4.2 (Maximize I)

/ Maximize I of e_2 for the case of having $e_1 \leq e_2$, where e_1 is an operand and e_2 is the result of an object algebra expression*

1. *Let c_1 and c_2 be the classes that correspond to an operand in an object algebra expression e_1 and the expression e_2 itself, respectively.*

2. *For any class c_i, let e_i be the corresponding object algebra expression.*

3. *Find $IL(c_1)$ by definition 4.4.*

4. *For j:=1 to $card(IL(c_1))$*
   / adjust the schema*
   if $e_2 \leq_e e_j$ then
   $supers(c_2) := supers(c_2) - supers(c_j)$
   $supers(c_2) := supers(c_2) \bigcup \{c_j\}$
   $I_{variables}(c_2) := I_{variables}(c_2) - I_{variables}(c_j)$
   $messages(c_2) := messages(c_2) - messages(c_j)$
   endif
   endfor

5. *For every $c_k \in supers(c_1)$*
   / discard from $supers(c_1)$ any class c_k which has been added to $supers(c_2)$ in step 4; c_k becomes an indirect superclass of c_1*
   if $c_k \in supers(c_2)$ then
   $supers(c_1) = supers(c_1) - \{c_k\}$
   endif
   endfor

Lemma 4.18 *Algorithm 4.2 maximizes reusability when $e_1 \leq e_2$, where e_1 and c_2 are the operand and the result of an object algebra expression.*

Proof:

In step 3 of algorithm 4.2, the $IL(c_1)$ is determined. In step 4, elements of $IL(c_1)$ are checked in order, starting with those which are near the root OBJECT class until the direct superclasses of c_2 are reached. This order is followed for being important in increasing reusability according to theorem 4.3 as we go from the OBJECT class

down in the lattice. The possibility of the inclusion of the checked class in $supers(c_2)$ is always considered in a step to maximize I. The optimum is achieved in the case that any of the classes in $supers(c_1)$ moves into $supers(c_2)$. In step 5, classes found in $supers(c_1) \bigcap supers(c_2)$ are discarded from $supers(c_1)$ to maintain the relationship $supers(c_1) \bigcap supers(c_2) = \phi$. □

Theorem 4.4 *Algorithm 4.2 executes in time $O(n + k)$, where n is the number of classes in $IL(c_1)$ and k is the number of classes in $supers(c_1)$.*

Proof:
The if-statement in the body of the for-loop of step 4 is performed once for each of the classes in $IL(c_1)$ and therefore it is performed $card(IL(c_1)) = n$ times. On the other hand, the if-statement in the body of the for-loop of step 5 is performed once for each of the classes in $supers(c_1)$ and therefore it is performed $card(supers(c_1)) = k$ times. Hence, the whole algorithm is performed in time $O(n + k)$. □

Algorithm 4.2 maximizes reusability by increasing the number of classes along a path from the OBJECT class to a class which corresponds to an object algebra expression. It is meaningless to apply algorithm 4.2 to classes which are already subclasses of the class that corresponds to the operand(s). In those cases, I has already been maximized by c_1 itself being included in $supers(c_2)$. Rather, algorithm 4.2 is useful in cases where the class that corresponds to the result is a superclass of the class that corresponds to the operand(s) and mainly for the results of the projection operation, the union operation and the second case of the difference operation.

It has been shown that the class c_i corresponding to the output from a query could also be a subclass of the operand(s), e.g., the result of the nest operation is a subclass of the first operand. Consequently, it is necessary to decide on the subclasses of such a class, if any. Already, it has been shown that via algorithm 4.2 it is possible to decide on the superclasses of the class c_2 when such a class is a superclass of the operand(s). In other words, superclasses and subclasses of the result c_2 has been decided on for the case of having the result c_2 as a superclass of the operand(s). On the other hand, when the result c_2 is a subclass, its superclasses are known to be the operand. However, it is important to decide on the subclasses of c_2 for placing the result naturally in the lattice. For this purpose, there are two applicable approaches. In the first, all subclasses of the operand are made subclasses of the class c_2 (c_2 becomes the only

direct subclass of the operand). In the second, every subclass c of the operand results in an artificial subclass c' of itself. Objects in $instances(c)$ migrate into $instances(c')$ (the corresponding artificial subclass) and hence are considered in $T_{instances}(c)$.

Given any two classes c_i and c_j which result in the artificial subclasses c'_i and c'_j, respectively, having c_i as a subclass of c_j leads to have c'_i as a subclass of c'_j. Formally, $c_i \leq c_j \Leftrightarrow c'_i \leq c'_j$. The artificial classes corresponding to the direct subclasses of the operand are made direct subclasses of the class c_2 (the result of the object algebra expression) which is the only subclass with no corresponding artificial class. It is an implementation issue to make artificial classes invisible to and inaccessible by the user.

In the first approach, the definition of all subclasses of the operand has to be extended to capture the additional property due to the new class c_2. It may be semantically undesirable to make reachable the additional properties acquired by objects in those classes except when such objects are accessed for being in $T_{instances}(c_2)$. From such a point of view, the second approach seems more reasonable where the new properties are made accessible solely from class c_2, which is the exact target. The second approach is formalized in the following algorithm.

Algorithm 4.3 (Subclasses of the result for the subclass case)
/ due to this algorithm a sublattice of artificial classes equivalent to that rooted at the operand is constructed with its root being the output from the query with the operand as a superclass.*

1. *Let c_1 and c_2 be the classes corresponding to the operand in and the output from a query such that c_2 should be a subclass of c_1 ($c_2 \leq c_1$, but c_2 has not been added as a subclass of c_1 yet).*
2. *Let L be a list of classes which is initially empty.*
3. *Let $c_m := c_1$ and $c_n := c_2$*
4. *For every direct subclass c_j of c_m*
   - *create an artificial subclass c'_j*
   - *move objects in $instances(c_j)$ into $instances(c'_j)$*
   - *make c'_j a subclass of c_n, i.e., $supers(c'_j) := \{c_n\}$*
   - *add c_j to the list L*

5. *Until L becomes empty*

   - $c_m := head(L)$
   - $c_n := c'_m$ $\quad /* \textit{the artificial class corresponding to } c_m$
   - *delete* c_m *from* L
   - *perform step 4*

6. *make* c_2 *a subclass of* c_1, *i.e.,* $supers(c_2) := \{c_1\}$

Lemma 4.19 *Algorithm 4.3 determines subclasses of the result for operations resulting in a subclass of the operand(s).*

Proof:
The first execution of step 4 creates an artificial subclass for each of the direct subclasses of the operand c_1 and all these artificial classes are made direct subclasses of the output c_2 from the query. In step 5, every one of the subclasses of c_1 for which some artificial subclass has been created is considered. Steps 4 and 5 are iteratively executed until all the direct and indirect subclasses of the operand c_1 are considered. Always objects migrate from a class into its corresponding artificial subclass. The overall result of this iterative execution is the construction of a sublattice of artificial classes rooted at the resulting class c_2 and symmetric to the sublattice rooted at the operand c_1. This way, the new values added due to the evaluated operation are accessible solely from within the class c_2 corresponding to the output of the query. □

Theorem 4.5 *Algorithm 4.3 executes in time* $O(n)$, *where* n *is the number of direct and indirect subclasses of the operand class* c_1.

Proof:
Step 4 is first performed for all the direct subclasses of the operand class. Step 5 causes the execution of step 4 for the direct subclasses of every subclass considered previously in step 4. Therefore, steps 4 and 5 are performed interchangeably until all of the direct and indirect subclasses of the operand class are considered. Hence, the algorithm executes in time $O(n)$. □

Any of the artificial classes created in step 4 has only inherited properties; inherited from the class it was created from and the new property due to the evaluated query

inherited from the class c_2. The choice of which approach to follow depends on the desirable semantics and whether it is feasible to deal with such artificial classes due to the second approach.

4.5 Illustrative Examples

In this section, several examples are included to illustrate the distinguishing aspects of the query model presented in this chapter. The examples given next in this section will assume the classes presented in chapter 3.

Example 4.1 *Select students attending the course "CS565"*

$$S_1 = student\sharp s\ ["CS565" \in s\ courses()\ code()]$$

where $\sharp$ indicates that the variable s is bound to and ranges over the objects of the operand, here the *student* class. More than one variable may range over objects of an operand. For example, $student\sharp s_1\sharp s_2$ indicates that s_1 and s_2 range over objects of the *student* class. In the predicate expression, "$CS565" \in$ s courses() code(), the right hand side is of the form $(o\ x)$; hence satisfies definition 4.1. The use of $=$, calls for an evaluation of this query on a temporary basis. Thus, the resulting pair S_1 consists of the sets $T_{instances}(S_1) = \{o_5, o_9\}$ and $M_e(S_1) = M_e(student)$. Notice that the student with object identity o_7 is not included in $T_{instances}(S_1)$ because of not attending the course "$CS565$".

We differentiate between temporary and persistent evaluations of a query, where an assignment free query is always evaluated on a temporary basis while we use $=$ and $:=$ to differentiate between temporary and persistent based evaluations, respectively. While a temporary based evaluation of a query ends by finding the pair of sets in the result, a persistent based evaluation continues with the finding of class characteristics of the determined pair by using lemmata 4.1 to 4.17, theorem 4.2 and algorithm 4.2 is applied to have the result properly and naturally placed in the lattice.

Example 4.2 *Find brothers of 'Adams'.*

$$person\sharp p_1[p_1\ sex() = "M" \wedge \exists\ p_2 \in T_{instances}(person) \wedge\ p_2\ name() = "Adams" \wedge \exists\ p_3 \in T_{instances}(person) \wedge\ \{p_1, p_2\} \subseteq\ p_3\ children()]$$

In this example, the operand is the pair $< T_{instances}(person), M_e(person) >$ and the output is the pair $< \{o_6\}, M_e(person) >$. The predicate expression of this example

guarantees that the objects in the result are solely those who are children of the same person as *"Adams"*.

Example 4.3 *Find the wife of "Smith" assuming that they have children.*

$$person\sharp p\ [\exists p_1 \in T_{instances}(person) \wedge\ p_1\ name() =" Smith" \wedge\ p\ sex() =" F" \\ \wedge\ p_1\ children() = p\ children()]$$

The predicate expression of this example filters objects in the operand $< T_{instances}(person), M_e(person) >$ *to return as a result the pair* $< \{o_2\}, M_e(person) >$ *where the included object is that having the same children with "Smith".*

Example 4.4 *Assume that the student class were not present in the lattice and the research-assistant class is defined as:*

$$research\text{-}assistant < \{staff\},\ year : integer,\ courses : course >$$

To derive the student class as a persistent class and assuming that a student attends the department he works in, the research-assistant class is projected with respect to a set of messages:

$$student := research\text{-}assistant\ [\{name(),\ age(),\ sex(),\ children(),\ year(),\ courses(), \\ works\text{-}in() \rightarrow student\text{-}in()\}]$$

where works-in()→student-in(), indicates message renaming. The subset $\{name(), age(), sex(), children()\}$ *of the projection set could be replaced by messages(person) because the latter is the implicit representation of the former. So, the query could have been coded by implicitly specifying the projection set as:*

$$student := research\text{-}assistant[\{year(),\ courses(),\ works\text{-}in() \rightarrow student\text{-}in()\} \\ \bigcup messages(person)]$$

According to lemma 4.11 and theorem 4.2, the derived student class will be a direct superclass of the research-assistant class with $supers(student) = \phi$ and

$I_{variables}(student) = \{name : string,\ age : integer,\ sex : ["M",\ "F"],\ children : person,\ year : integer,\ courses : course,\ student\text{-}in : department\}$

$messages(student) = \{name(),\ age(),\ sex(),\ children(),\ year(),\ courses(),\ student\text{-}in()\}.$

$T_{instances}(student) = T_{instances}(research\text{-}assistant) = \{o_9\}$ and $instances(student) = \phi$ (not $\{o_5,\ o_7\}$ because of the assumption that the student class were not in the lattice and thus it is impossible to speak about instances of a class which does not exist).

In general a class c derived by an object algebra expression has $instances(c) = \phi$ just after its derivation. Later the set $instances(c)$ may be extended to include new instances as they exist.

Due to algorithm 4.2 which aims to maximize reusability, the derived student class will be recognized as a subclass of the person class and naturally placed in the lattice to have $supers(student) = \{person\}$ with $I_{variables}(c)$ and $messages(c)$ adjusted accordingly.

Example 4.5 *Find the names and courses of students attending at least one course*

student‡s [s courses() ≠ φ] ![{name(), courses() code()}]

Notice the use of the message expression, *courses() code()*, which is a concatenation of two messages, one from each of *student* and *course* classes, respectively. First students attending some courses are selected and the pair $<\{o_5, o_7, o_9\}, M_e(student)>$, i.e., all objects in $T_{instances}(student)$ are selected in the result. Second, the one level project operation is performed with the pair obtained by the selection operation as the operand to get the pair

$<\{<"Susan", \{"CS565", "CS578"\}>, <"Tom", \{"CS211", "CS330"\}>,$
$<"George", \{"CS565", "CS578"\}>\}, \{name(), code()\}>$

as the result.

Example 4.6 *Let net-salary(t) be a method defined in the staff class to return the net salary of a staff member after deducting taxes at the rate of t. To get the names and net salaries of staff members, assuming t=0.1, write:*

staff ![{name(), net-salary(0.1)}]

In this example, *net-salary(t)* returns a value derived in terms of the stored value *salary*. The operand is the pair $<T_{instances}(staff), M_e(staff)>$, while the output is the pair

$<\{<"Smith", 45K>, <"Adams", 54K>, <"George", 13.5K>\}, \{name(), m()\}>$,

where $m()$ is a message added to the result to handle the values derived by the *net-salary(0.1)* method and included in each of the objects in the output pair.

Example 4.7 *Assume that both the student and the staff classes have an instance variable 'field' specifying the field of interest. To assign every student the set of staff members that he can consult, assuming that a student can consult staff members sharing his field of interest, we write:*

$$(student\sharp s_1 >> (staff - research\text{-}assistant)\sharp s_2)\ [s_1\ field() = s_2\ field()]$$

where s_1 is an object variable bound to objects of the student class and s_2 is bound to objects in the result of the difference operation. The student class is nested with the difference of the staff and research-assistant classes, by assigning to every student the set of staff members satisfying the given predicate expression; which is actually a join operation as follows:

$$student\sharp s_1 < s_1\ field() = s_2\ field() > (staff - research\text{-}assistant)\sharp s_2$$

Example 4.8 *Find couples having at least one child.*

$$person\sharp p_1 >> person\sharp p_2\ [p_1\ sex() = "M" \wedge\ p_2\ sex() = "F" \wedge\ p_1\ children() \neq \phi \wedge p_1\ children() = p_2\ children()]$$

The pair $<T_{instances}(person),\ M_e(person)>$ is nested with itself by assigning to every male person the female sharing the same children. The result is the pair $<\{<"Smith", 45, "M", \{o_1, o_7\}, 50K, o_{10}, o_2>\},\ M_e(person) \bigcup (m()\ M_e(person))>$, where $m()$ is a message added to the result to return from any object in the output pair the object identity of his wife.

In this example, using the cross-product operation instead of the nest operation returns the pair: $<\{<o_6, o_2>\},\ (m()\ M_e(person)) \bigcup (m_1()\ M_e(person))>$, where $m()$ and $m_1()$ are two messages added to the result to return from any object in the output pair the object identity of the male and the female, respectively.

Example 4.9 *Find the department whose head is "Adams".*

$$department\sharp d[d\ head()\ name() = "Adams"]$$

The output pair of this query is the pair:

$<\{<"Computer\ Science", o_8>,\ M_e(department)>$.

Example 4.10 *Find students attending the department whose head is "Adams".*

$$student\sharp s[s\ student\text{-}in()\ head()\ name() = "Adams"]$$

The same query can also be coded as:

$$(student\sharp s_1 >> staff\sharp s_2[s_1\ student\text{-}in()\ head() = s_2])\sharp s_3\ [s_3\ m()\ name() = "Adams"][messages(student)]$$

While the output pair due to the first form of the query is:

$<T_{instances}(student),\ M_e(student)>$,

the output pair of the second form of the query is the result of the project operation on the pair returned by the selection operation applied on the result of the nest operation. The nest operation assigns to every student object the identity of the chairperson of his department and returns the pair:

$$<\{<"Susan", 25, "F", \phi, 5, \{o_{11}, o_{16}\}, o_{10}, o_8>,$$
$$<"Tom", 18, "M", \phi, 3, \{o_{13}, o_{14}\}, o_{10}, o_8>,$$
$$<"George", 22, "M", \phi, 5, \{o_{11}, o_{16}\}, o_{10}, 15K, o_{10}, o_8>\},$$
$$M_e(student) \bigcup (m()\ M_e(staff))>.$$

Here, $m()$ is a new message understandable by the objects in the resulting pair to return the chairperson of the attended department. Since the chairperson is himself a staff member, any of the message expressions in $M_e(staff)$ could be applied to the result of the message $m()$. Consequently, the selection operation filters objects in the result of the nest operation to return those attending the department chaired by *"Adams"*. In this example, no object has been eliminated because all the objects in the result of the nest operation are attending the department chaired by *"Adams"* and hence the pair resulting from the selection operation is the same pair obtained as the result of the nest operation. Finally, the project operation produces a pair which includes the same set of objects returned by the nest operation but the set of message expressions is $M_e(student)$.

Example 4.11 *Find students attending the department in which "Adams" is working.*

$$(student \sharp s_1 >> staff \sharp s_2[s_1\ student\text{-}in() = s_2\ works\text{-}in() \wedge s_2\ name() = "Adams"]) \sharp s_3[s_3\ m() \neq \phi][messages(student)]$$

This query could be interpreted by the same way as the second form of example 4.10.

Example 4.12 *Assume that the person class were not present in the lattice (thus, objects in instances(student) are ignored in this example) with the student and the staff classes defined as follows:*

student $<$ $\emptyset$, *name:string, age:integer, sex:["M", "F"], children:person, year:integer, courses:course, student-in:department* $>$

staff $<$ $\emptyset$, *name:string,age:integer,sex:["M", "F"], children:person, salary:integer, works-in:department*$>$

The person class is derived as:

$$person := student \bigcup staff$$

According to lemma 4.13 and theorem 4.2, the *person* class is a superclass of both operands and includes the union of their objects, but the intersection of their message expressions as stated in lemma 4.5. Thus,

$T_{instances}(person) = T_{instances}(student) \bigcup T_{instances}(staff)$,

$M_e(person) = M_e(student) \bigcap M_e(staff)$,

$I_{variables}(person) = \{name{:}string, age{:}integer, sex{:}["M", "F"], children{:}person\}$

$messages(student) = \{name(), age(), sex(), children()\}$

Actually, the derived *person* class due to this query is the same *person* class defined in chapter 3.

Example 4.13 *Find students who are not research assistants*

$$student - research\text{-}assistant$$

Since $M_e(student) - M_e(research\text{-}assistant) = \phi$, *because* $M_e(student) \subseteq M_e(research\text{-}assistant)$, *in the output pair* $M_e(student)$ *is returned according to definition 4.2. Remembering that* $T_{instances}(research\text{-}assistant) \subseteq T_{instances}(student)$, *the same query can be coded using the select operation as follows:*

$$student\sharp s\ [s \notin T_{instances}(research\text{-}assistant)]$$

Here s is bound to objects in the *student* class.

Regardless of which one of the two forms is considered, the output pair of this query is $< instances(student),\ M_e(student) >$.

Example 4.14 *In example 4.6 it was assumed that t=0.1 is fixed for all staff members. In this example, we assume t=0.1 for research-assistants and t=0.15 for other staff members. To find the names and net salaries of staff members, we write:*

$$(staff - research\text{-}assistant)\ ![\{name(), net\text{-}salary(0.15)\}]$$
$$\bigcup\ research\text{-}assistant\ ![\{name(), net\text{-}salary(0.1)\}]$$

First the difference operation is used to find staff members who are not research assistants; then the one level project operation is applied on the result with t=0.15 and on *research-assistants* with t=0.1; the union of both results is considered to be the output from this query. However, this formulation of the query assumes a previous knowledge of all the subclasses of the *staff* class. Also, this formulation seems reasonable because of having only a single subclass of the *staff* class. Otherwise, it is cumbersome to handle such a query by considering the given formulation. Rather, a general formulation

regardless of the number of subclasses without the need to know all of them is possible as follows:

$$OAE(instances(staff))\ ![\{name(),\ net\text{-}salary(0.15)\}]$$
$$\bigcup OAE(T_{instances}(staff) - instances(staff))\ ![\{name(),\ net\text{-}salary(0.1)\}]$$

In this formulation, we benefit from having $instances(staff)$ being defined to include objects in the *staff* class not common with any of its subclasses.

Example 4.15 *Find pairs of students attending the same courses*

$$(student\sharp s_1 \times student\sharp s_2)\ [s_1\ courses() = s_2\ courses() \wedge s_1\ name() < s_2\ name()]$$

The result of this query is the pair:

$$<\{<o_5,\ o_9>\},\ (m_1()\ M_e(student))\bigcup(m_2()\ M_e(student))>$$

Where $m_1()$ and $m_2()$ are new two messages added to return the first and second constituting values (object identity) of a receiving object from the resulting pair, respectively.

Remember from definition 4.2 that, when combined with a selection operation, both of the cross-product and the nest operations result in a join operation. While the join due to a nest is an outer-join, the join due to a cross-product is an inner-join. Notice that the result of the query of *example 4.15* will be a direct subclass of the root because the *student* class has some instance variables with atomic domains. However, using nest instead of cross-product forces the result to be a subclass of the *student* class. The difference is due to the fact that while the nest operation appends to every student a set of identities of related students, the cross-product operation on the other hand forms, according to the definition of cross-product operation in definition 4.2, new values each consisting of the identity of a student together with the identity of a related student as indicated.

Example 4.16 *Find staff members earning more than the average salary in their department*

$$staff\sharp s_1 >> staff < \{works\text{-}in()\},\ average, salary() > \sharp s_2$$
$$[s_1 \in s_2\ m() \wedge s_1\ salary() > s_2\ avsalary()]\ [\{name()\}]$$

where $m()$ and *avsalary()* are the two messages in the result of the aggregate function application operation to return the set of object identities of staff members working in the same department and the calculated average salary, respectively; the message

avsalary() is a concatenation of the first two letters of the applied function, *average*, with the last message in the used message expression, here *salary()*. The *staff* class is nested with the result of the application of the aggregate function *average* on *staff* members grouped by *works-in()*. In other words, first the set $T_{instances}(staff)$ is partitioned into equivalence classes based on the result of the message expressions in the set $\{works\text{-}in()\}$ by collecting in the same equivalence class staff members working for the same department. The second step is the application of the message expression *salary()* to every object to get the corresponding salary and the aggregate function *average* is applied to get the average salary for objects in every equivalence class, separately to get the pair: $< \{o_{17} < \{o_6, o_8, o_9\}, 41.667K >\}, \{m(), avsalary()\} >$. The staff class is nested with this pair resulting from the aggregate function application operation to get the pair:

$< \{ <"Smith", 45, "M", \{o_1, o_7\}, 50K, o_{10}, o_{17} >,$
$<"Adams", 40, "M", \phi, 60K, o_{10}, o_{17} >,$
$<"George", 22, "M", \phi, 15K, o_{10}, o_{17} >, \},$
$M_e(staff) \bigcup \{m_1()\, m(), m_1()\, avsalary()\} >,$

where m_1 is a message added to the result of the nest operation to facilitate reaching the related objects in the pair resulting from the aggregate function application operation. Then those staff members satisfying the given predicate expression are selected and finally projection on $\{name()\}$ is performed. The overall result of this query is the pair

$< \{ <"Smith", 45, "M", \{o_1, o_7\}, 50K, o_{10}, o_{17} >,$
$<"Adams", 40, "M", \phi, 60K, o_{10}, o_{17} >\}, \{name()\} >$

Example 4.17 *Find students taking the same number of credits*

$student1 = student < \{name()\}, sum, courses()\ credit() >$
$(student1 \sharp s_1 >> student1 \sharp s_2)\ [s_1\ sucredit() = s_2\ sucredit() \wedge$
$s_1\ m()\ name() < s_2\ m()\ name()]$

student1 is evaluated on a temporary basis due to the usage of =. The aggregate function *sum* is evaluated against the result of the message expression *courses() credit()* after grouping objects in the student class by collecting together students giving for whom the same result is returned by message expressions in the set $\{name()\}$, i.e., every student is handled separately assuming that name is unique.

4.6 Recursive Queries

Recursive query evaluation is a capability of deductive database systems that conventional database systems do not support well if at all. In this section, we describe how to handle linear recursive queries which are the ones in which only one appearance of the recursive predicate is allowed on the right hand side of a Horn clause [4]. Horn clauses are used in logic based languages as illustrated in the following example which returns the ancestors of a person:

ancestor(x,y):- parent(x,y).

ancestor(x,y):- ancestor(x,z) $\wedge$ parent(z,y).

A recursive query assumes some base elements in the resulting set, and keeps on adding elements satisfying a given criteria specified via a predicate expression, until no more elements could be added. The evaluation of a recursive query is equivalent to determining the transitive closure of the resulting set. It is in general, a selection with the predicate expression consisting of the disjunction of two or more predicate expressions, which implies that the truth value of one of them being true is enough to decide on the truth value of the whole predicate expression to be true. Only the selection operation is considered in recursive queries because the join and the projection operations which are explicitly utilized while handling recursive queries in the relational model are considered implicit in an object-oriented data model. One of the predicate expressions is responsible for including in the result of the query some starting objects based on which a recursive evaluation of the other predicate expressions is handled until no more objects could be added to the result. The predicate expression which is responsible for including the base objects in the result of the query which at the start is empty, is not considered any more except for the first step. This is so because a predicate expression evaluates to true only against objects of the operand, if possible. The set of total instances of the operand is not extendible to include new objects while evaluating the query and hence a single evaluation of that predicate expression is enough to include the required related objects from the operand in the result of the query. On the other hand, any object added to the result at any time, could cause the addition of other objects by evaluating the rest of the predicate expressions. Always objects which were latest included in the result are considered while evaluating the rest of the predicate expressions for the possibility of appending new objects to the result

of the query.

Formally speaking, to determine objects in the result of a recursive query, three sets R, R_1 and O are assumed to include objects in the result, objects added to the result at an intermediate step and objects in the operand, respectively. As a recursive query is equivalent to a selection, let the corresponding predicate expression be $p \vee q$, where p does not include any object variable bound to objects in the result while q does. To get objects constituting the result R, the following steps are done, where R_1^i indicates the set of objects calculated at the intermediate step $i \geq 0$.

1. $OAE(R_1^0) = OAE(O)[p]$
2. $OAE(R_1^i) = OAE(O)[q]$, $(i \geq 1)$ with q being adjusted to have the object variable in q to be bound to objects in the result R being bound to objects in the intermediate result R_1^{i-1}. Next in example 4.18, it is shown how q is adjusted to serve the purpose of this step.
3. $R = \bigcup(R_1^i)$ $(i \geq 0)$

Step 2 is iteratively performed until for some $i \geq 0$, R_1^i is empty. Evaluating any recursive query results in the following pair:

$M_e(OAE(R)) = M_e(OAE(O))$

$T_{instances}(OAE(R)) = R$

In the following examples we are going to illustrate recursive queries.

Example 4.18 *Find all descendants of "Smith".*

$$D\sharp d = person\sharp p[(\exists p_1 \in T_{instances}(person) \wedge\ p_1\ name() = "Smith" \wedge\ p \in p_1\ children()) \vee\ (p \in d\ children())]$$

In the query of example 4.18, the first part of the predicate expression,

$(\exists p_1 \in T_{instances}(person) \wedge p_1\ name() = "Smith" \wedge\ p \in p_1\ children())$,

is responsible for adding the children of *"Smith"* to the result D, which is prior to that empty. After that, the first part is ignored and only the second part of the predicate expression, $(p \in d\ children())$, is recursively considered to add to the result children of a person already added, until no more persons could be added any more. By this way, the obtained result D includes from the person class those who are the descendants of *"Smith"*.

Explicitly speaking, in example 4.18, R_1^0 and R_1^i $(i \geq 1)$ are determined by:

$$OAE(R_1^0) = person\sharp p[(\exists p_1 \in T_{instances}(person) \wedge\ p_1\ name() = "Smith" \wedge p \in p_1\ children())]$$

$$OAE(R_1^i) = person\sharp p[\exists\ d \in R_1^{i-1} \wedge\ (p \in d\ children())]$$

Thus, at any intermediate step, for $(i \geq 1)$, R_1^i is determined depending on R_1^{i-1} which is determined in the previous step. Notice how the second predicate expression has been adjusted to have the object variable d which was bound to objects in the result D in the original query, is being bound to objects in R_1^{i-1} by adding $\exists\ d \in R_1^{i-1}$ to the predicate expression present at any intermediate step. At the end, the final result D has:

$$M_e(D) = M_e(person)$$

$$T_{instances}(D) = \bigcup\ R_1^i$$

Example 4.19 *Find all ancestors of "Smith".*

$$A\sharp d = person\sharp p[(\exists p_1 \in T_{instances}(person) \wedge\ p_1\ name() = "Smith" \wedge p_1 \in p\ children()) \vee\ (d \in p\ children())]$$

The first part in the predicate expression of this query,

$$(\exists p_1 \in T_{instances}(person) \wedge\ p_1\ name() = "Smith" \wedge\ p_1 \in p\ children()),$$

adds the parents of *"Smith"* to the result A. The second part of the predicate expression, $(d \in p\ children())$, keeps on adding the parents of any person added to the result, until no more persons could be added.

Example 4.20 *Find all prerequisites of the course "CS450".*

$$P\sharp p = course\sharp c[(\exists c_1 \in T_{instances}(course) \wedge\ c_1\ code() = "CS450" \wedge c \in c_1\ prerequisites()) \vee\ (c \in p\ prerequisites())]$$

The first part of the predicate expression of the query of this example,

$$(\exists c_1 \in T_{instances}(course) \wedge\ c_1\ code() = "CS450" \wedge\ c \in c_1\ prerequisites()),$$

adds prerequisites of the course *"CS450"* to the result P. Then the second part of the predicate expression, $(c \in p\ prerequisites())]$, adds the prerequisites of any course added to the result P until no more courses could be added.

Example 4.21 *Find all relatives of "Tom"*

$$R\sharp r = person\sharp p[(p\ name() = "Tom") \vee (p \in r\ children()) \vee (r \in p\ children()) \vee (\exists\, p_1 \in T_{instances}(person) \wedge \{p, r\} \subseteq p_1\ children())]$$

The first predicate expression of example 4.21, $(p\ name() = "Tom")$, adds *"Tom"* himself to the result R which is initially empty. The second predicate expression, $(p \in r\ children())$, extends R by adding *children* of a person in R. The third predicate expression, $(r \in p\ children())$, extends R by adding the parents of any person in R. The fourth predicate expression, $(\exists\, p_1 \in T_{instances}(person) \wedge \{p, r\} \subseteq p_1\ children())$, extends R by adding siblings of any person in R. The result R keeps on extending due to the second, third and fourth predicates being interchangeably evaluated against the persons which were added to R and not considered yet, until no more persons can be added.

Example 4.22 *Assume the existence of a class called employee, with each employee having a manager. It is required to find all employees working under the supervision of "Adams".*

$$S\sharp s = employee\sharp e[(e\ name() = "Adams") \vee (e\ manager() \in s)]$$

In this query the first part of the predicate expression, $e\ name() = "Adams"$, guarantees the addition of *"Adams"* to the result. Then the second part of the predicate expression, $e\ manager() \in s$, keeps on extending the result by adding new employees from those managed by *"Adams"* or managed by any of the employees managed directly or indirectly by *"Adams"*, until the result is no more extendible.

Example 4.23 *Find descendants common to "Jack" and "Mary".*

$$\sharp j\ person\sharp p[(\exists p_1 \in T_{instances}(person) \wedge p_1\ name() = "Jack" \wedge p \in p_1\ children()) \vee (p \in j\ children())] \bigcap \sharp m\ person\sharp p[(\exists p_1 \in T_{instances}(person) \wedge p_1\ name() = "Mary" \wedge p \in p_1\ children()) \vee (p \in m\ children())]$$

The output from this query is the result of the intersection operation with the two operands being obtained as results of recursive queries similar to that of example 4.18. The first of these operands is a recursive query to find descendants of *"Jack"* with the

object variable j being bound to objects of the result while it is evaluated. The second operand is also a recursive query which determines the descendants of *"Mary"* with m being the object variable bound to objects of the result while it is evaluated.

Example 4.24 *Find descendants of "Jack" who are not descendants of "Mary".*

$$\sharp j\ person \sharp p[(\exists p_1 \in T_{instances}(person) \wedge\ p_1\ name() = "Jack" \wedge\ p \in p_1\ children()) \vee\ (p \in j\ children())]$$
$$- \sharp m\ person \sharp p[(\exists p_1 \in T_{instances}(person) \wedge\ p_1\ name() = "Mary" \wedge\ p \in p_1\ children()) \vee\ (p \in m\ children())]$$

The query of this example is similar to that of example 4.23, but with the intersection operation being replaced by the difference operation.

4.7 Superiority of the Object Algebra over the Relational Algebra

It is important to emphasize that, since we have the five basic operators of the relational algebra, the object algebra has at least the power of the relational algebra. In fact, our object algebra is more powerful because the relational algebra handles only atomic domains and only stored values can be retrieved which is nothing more than a restriction that leads to an embedded query language and hence impedance mismatch in contrast to the object algebra which handles stored as well as derived values and hence satisfies computational completeness. Furthermore, the proposed model allows for set based predicates and quantifiers are allowed in predicates in contrast to atomic predicates in relational algebra. Also we support encapsulation, object identity and inheritance. Generally speaking, the expressiveness of the constructors of an object-oriented data model do affect the expressiveness of the corresponding query language over the relational algebra. In other words, an object-oriented data model allows the definition of data through abstraction, supports derived data in addition to multivalued properties, complex objects, identity and inheritance. As a result the same real world situation can be expressed by a simpler object-oriented schema than the relational schema and hence all queries that are coded using the relational algebra could be expressed using an object-oriented query language; however, the reverse is not true. In addition to that, recursion is not supported by the relational algebra, however linear

recursion is handled by the object algebra described in this work. Hence, an object-oriented query language is more expressive than the relational algebra for capturing the distinguishing properties of an object-oriented data model.

Concerning the nested relational algebra, although it handles non-atomic domains, it imposes the restriction of manipulating only stored values which is equivalent to having only message expressions that return stored values and excluding those that return derived values and hence does not overcome the impedance mismatch problem. Also, it does not support inheritance, neither identity nor encapsulation. In other words, the nested relational model aims to represent complex objects by nesting relations, but still they are value-based and record-oriented models. Next are the object algebra equivalents for the Nest and Unnest operations of the nested relational algebra. We assume that e_1 has a set of attributes N_1 and consider every element of N_1 as a message that returns the corresponding stored value in a receiving object (a tuple in a nested relation). Furthermore, we assume $T_{instances}(e_1)$ the same as the set of tuples in an equivalent nested relation and $M_e(e_1)$ has an equivalent calculation starting with attributes of e_1 and combining with nested attributes. Now given $N \subseteq N_1$:

$$\begin{aligned}\text{Nest}(e_1,\text{N})= \quad & (e_1[M_e(e_1)-\{x \mid x \in M_e(e_1) \wedge \exists m \in N \ \wedge \ x = (m\ x_j)\}] \\ >> \ & e_1[\{x \mid x \in M_e(e_1) \ \wedge \ \exists m \in N \ \wedge \ x = (m\ x_j)\}])\%s \\ & [\exists s_1 \in T_{instances}(e_1) \ \wedge \ s\ m_1()\ N = s_1\ N \ \wedge \\ & \quad s\ (M_e(e_1) - N) = s_1\ (M_e(e_1) - N)]\end{aligned}$$

where $m_1()$ is a message added to the result of the nest operation (>>) to facilitate the access of objects in the second operand.

Unnest(e_1,N)= e_1 ![$messages(e_1)-\{m_2\}\bigcup(m_2\ messages(e_2))$]

where $messages(e_2)$ corresponds to the set of attributes N and $m_2 \in messages(e_1)$ with underlying domain $T_{instances}(e_2)$.

So, the one level project operation corresponds to a sequence of Unnest followed by a projection in the nested relational model [2, 44, 57]. Also, the one level project operation does the function of *project* and *image* operations described in [74, 105], the *apply* operation of [78] and the *map* operation described in [97, 98], but we maintain the closure property without additional constructs.

As a result, the object algebra has the power of the nested algebra. It is more powerful due to the manipulation of stored and derived values, in addition to supporting the object-oriented features.

4.8 Equivalence of Object Algebra Expressions

In this section we are going to present some equivalences between object algebra expressions which are important in query optimization. In particular, associativity of the cross-product operation is proved for being important in query optimization.

Let e_1, e_2 and e_3 be object algebra expressions, such that:
$M_e(e_1)=X_1$, $M_e(e_2)=X_2$ and $M_e(e_3)=X_3$.

- Given two predicate expressions p_1 and p_2,
  - $e_1[p_1][p_2] = e_1[p_1 \wedge p_2] = e_1[p_2 \wedge p_1] = e_1[p_2][p_1]$
- Given $X \subseteq X_1$ and a predicate expression p_1,
  - $e_1[p_1][X] = e_1[X][p_1]$ iff $\forall x$ appearing in p_1, we have $x \in X$
- $e_1 \times e_2 = e_2 \times e_1$
- $e_1 \times (e_2 \times e_3) = (e_1 \times e_2) \times e_3$
- $e_1 \bigcup e_2 = e_2 \bigcup e_1$
- $e_1 \bigcup (e_2 \bigcup e_3) = (e_2 \bigcup e_1) \bigcup e_3$
- Given $X_4 \subseteq X_1$ and $X_5 \subseteq X_1$,
  - $e_1[X_4][X_5] = e_1[X_5]$ iff $X_5 \subseteq X_4$
- $e_1 \times (e_2 \bigcup e_3) = (e_1 \times e_2) \bigcup (e_1 \times e_3)$ iff all instance variables in e_2 and e_3 have non-atomic underlying domains regardless of the domains of the instance variables in e_1.
- $e_1 \bigcup (e_2 \times e_3) = (e_1 \bigcup e_2) \times (e_1 \bigcup e_3)$ iff all instance variables in e_1, e_2 and e_3 have non-atomic underlying domains.
- $(e_1 \times e_2)[p] = e_1[p] \times e_2[p]$ where p is a predicate expression.
- $(e_1 \bigcup e_2)[p] = e_1[p] \bigcup e_2[p]$ where p is a predicate expression.
- $(e_1[p_1] \bigcup e_2[p_2]) = e_1[p_1 \vee p_2]$ where p_1 and p_2 are predicate expressions.
- $(e_1[p_1] - e_2[p_2]) = e_1[p_1 \wedge \neg p_2]$ where p_1 and p_2 are predicate expressions.

- $(e_1 \times e_2)[X] = e_1[X] \times e_2[X]$ where $X \subseteq (X_1 \bigcup X_2)$.
- $(e_1 \bigcup e_2)[X] = e_1[X] \bigcup e_2[X]$ where $X \subseteq (X_1 \bigcap X_2)$.
- $(e_1 \bigcup e_2)![X] = e_1![X] \bigcup e_2![X]$ where $X \subseteq (X_1 \bigcap X_2)$.

The one-level-project and the project operations are equivalent when $M_1 \subseteq messages(e_1)$ and $X \subseteq M_e(e_1)$ such that elements of X return only stored values:

$$e_1\,[X] = e_1\,![M_1]$$

where $M_1 = \{m_1, m_2, ..., m_n\}$ and X$=\{x_1, x_2, ..., x_n\}$ with $x_i = (m_i\ x_{p_i})$ for $1 \leq i \leq n$ and arbitrary x_{p_i}.

The cross-product operation is equivalent to a combination of the nest, project and one-level-project operations as follows:

1. If all the stored values in $T_{instances}(e_1)$ and $T_{instances}(e_2)$ have non-atomic underlying domains:

   $$e_1 \times e_2 = (e_1 >> e_2)![messages(e_1) \bigcup (m\ messages(e_2))]$$

   where $T_{instances}(e_2)$ is the range (domain of the result) of the message m in the result of $e_1 >> e_2$

2. If only the stored values in $T_{instances}(e_2)$ have non-atomic underlying domains:

   $$e_1 \times e_2 = e_2 >> e_1$$

3. If only the stored values in $T_{instances}(e_1)$ have non-atomic underlying domains:

   $$e_1 \times e_2 = e_1 >> e_2$$

4. If at least one of the stored values in each of $T_{instances}(e_1)$ and $T_{instances}(e_2)$ has atomic underlying domain:

   $$e_1 \times e_2 = (e_1 >> e_2)[m] >> e_1$$

   where $T_{instances}(e_2)$ is the domain of the result of the message m in the result of $e_1 >> e_2$

Under this same condition, i.e. condition 4, we have:

$$e_1 >> e_2 = (e_1 \times e_2)[(m_1\ messages(e_1)) \bigcup \{m_2\}]$$

where m_1 and m_2 are two messages in the result of the $e_1 \times e_2$ with their domains being $T_{instances}(e_1)$ and $T_{instances}(e_2)$,respectively.

Finally, the following properties of the object algebra are useful; given an object algebra expression e,

$e[M_e(e)]=e$,

$e![messages(e)]=e$.

The proofs of the given equivalences will be left out as they all follow from definition 4.2. As an example, next we sketch the steps that lead to the proof of the associativity of the cross-product. For that purpose, we give a lemma on the length of message expressions in $(e_1 \times e_2)$.

Lemma 4.20 *Let e_1 and e_2 be elements of E.*

$\forall x \in M_e(e_1 \times e_2)$, such that the range of x is atomic, $len(x)>1$

Proof:

Let X_1 and X_2 be the message expressions of e_1 and e_2, respectively.

It follows from the definition of $(e_1 \times e_2)$ that there are four cases to consider:

1) $\exists x_1 \in X_1,\ len(x_1)=1 \wedge \exists x_2 \in X_2\ len(x_2)=1$

$M_e(e_1 \times e_2)=\{x | x \in ((m_1\ X_1) \cup (m_2\ X_2))\}$ *(By definition)*

$\Rightarrow \forall x \in M_e(e_1 \times e_2)\ \exists x_1 \in X_1,\ len(x)=len(x_1)+1 \vee \exists x_2 \in X_2,\ len(x)=len(x_2)+1$

$\Rightarrow len(x)>1$

2) $\forall x_1 \in X_1,\ len(x_1)>1 \wedge \exists x_2 \in X_2,\ len(x_2)=1$

$M_e(e_1 \times e_2)=\{x | x \in (X_1 \cup (m_2\ X_2))\}$ *(By definition)*

$\Rightarrow \forall x \in M_e(e_1 \times e_2)\ \exists x_1 \in X_1,\ len(x)=len(x_1) \vee \exists x_2 \in X_2,\ len(x)=len(x_2)+1$

$\Rightarrow len(x)>1$

3) $\exists x_1 \in X_1\ len(x_1)=1 \wedge \forall x_2 \in X_2,\ len(x_2)>1$

$M_e(e_1 \times e_2)=\{x | x \in ((m_1\ X_1) \cup X_2)\}$ *(By definition)*

$\Rightarrow \forall x \in M_e(e_1 \times e_2)\ \exists x_1 \in X_1,\ len(x)=len(x_1)+1 \vee \exists x_2 \in X_2,\ len(x)=len(x_2)$

$\Rightarrow len(x)>1$

4) $\forall x_1 \in X_1,\ len(x_1)>1 \wedge \forall x_2 \in X_2,\ len(x_2)>1$

$M_e(e_1 \times e_2)=\{x | x \in (X_1 \cup X_2)\}$ *(By definition)*

$\Rightarrow \forall x \in M_e(e_1 \times e_2)\ \exists x_1 \in X_1,\ len(x)=len(x_1) \vee \exists x_2 \in X_2,\ len(x)=len(x_2)$

$\Rightarrow len(x)>1$

Hence, $len(x)>1$.□

Lemma 4.20 shows that $e_1 \times e_2$ has all message expressions satisfying the condition that the length of a message expression is greater than one when the underlying domain

of the returned value is atomic. In other words, all messages of $e_1 \times e_2$, i,e, message expressions of length one, return values from non-atomic domains. This characteristic is utilized in the following theorem to justify the associativity of the cross-product operation.

Theorem 4.6 (The Cross-product operation is associative.)
$\forall e_1,e_2,e_3 \in E$, *we have:*

$$(e_1 \times e_2) \times e_3 = e_1 \times (e_2 \times e_3)$$

Proof:

Follows from definition 4.2 and lemma 4.20 as:
Let X_1, X_2 *and* X_3 *be* $M_e(e_1)$, $M_e(e_2)$ *and* $M_e(e_3)$, *respectively, and*
Let T_1, T_2 *and* T_3 *be* $T_{instances}(e_1)$, $T_{instances}(e_2)$ *and* $T_{instances}(e_3)$, *respectively.*
By the definition of the Cross-product, there are eight cases to consider:

1) $\exists x_1 \in X_1$, $len(x_1)=1 \wedge \exists x_2 \in X_2$, $len(x_2)=1 \wedge \exists x_3 \in X_3$, $len(x_3)=1$

$M_e(e_1 \times e_2)=\{x | x \in ((m_1\ X_1) \cup (m_2\ X_2))\}$ *(By definition)*

$M_e(e_2 \times e_3)=\{x | x \in ((m_2\ X_2) \cup (m_3\ X_3))\}$ *(By definition)*

$M_e((e_1 \times e_2) \times e_3)= \{x | x \in (M_e(e_1 \times e_2) \cup (m_3\ X_3))\}$ *(By definition and Lemma 4.20)*

$=\{x | x \in ((m_1\ X_1) \cup (m_2\ X_2) \cup (m_3\ X_3))\}$

$=\{x | x \in ((m_1\ X_1) \cup M_e(e_2 \times e_3))\} = M_e(e_1 \times (e_2 \times e_3))$

$T_{instances}(e_1 \times e_2)= \{o | \exists o_1 \in T_1 \exists o_2 \in T_2 \wedge value(o)=identity(o_1).identity(o_2)\}$
(By definition)

$T_{instances}(e_2 \times e_3)= \{o | \exists o_2 \in T_2 \exists o_3 \in T_3 \wedge value(o)=identity(o_2).identity(o_3)\}$
(By definition)

$T_{instances}((e_1 \times e_2) \times e_3)= \{o | \exists o_{11} \in T_{instances}(e_1 \times e_2)\ \exists o_3 \in T_3 \wedge$
$value(o)=value(o_{11}).identity(o_3)\}$
(By definition and Lemma 4.20)

$=\{o | \exists o_1 \in T_1 \exists o_2 \in T_2\ \exists o_3 \in T_3 \wedge value(o_{11})=identity(o_1).identity(o_2)$
$\Rightarrow value(o)=identity(o_1).identity(o_2).identity(o_3)\}$

$=\{o | \exists o_1 \in T_1 \exists o_{13} \in T_{instances}(e_2 \times e_3) \wedge value(o_{13})= identity(o_2).identity(o_3)$
$\Rightarrow value(o)=identity(o_1).value(o_{13})\}$

$=T_{instances}(e_1 \times (e_2 \times e_3))$

2) $\forall x_1$, $len(x_1)>1 \wedge \in X_1 \exists x_2 \in X_2$, $len(x_2)=1 \wedge \exists x_3 \in X_3$, $len(x_3)=1$

$M_e(e_1 \times e_2)=\{x | x \in (X_1 \cup (m_2\ X_2))\}$

$M_e(e_2 \times e_3) = \{x | x \in ((m_2\ X_2) \cup (m_3\ X_3))\}$

$M_e((e_1 \times e_2) \times e_3) = \{x | x \in (M_e(e_1 \times e_2) \cup (m_3\ X_3))\}$

$= \{x | x \in (X_1 \cup (m_2\ X_2) \cup (m_3\ X_3))\}$

$= \{x | x \in (X_1 \cup M_e(e_2 \times e_3))\} = M_e(e_1 \times (e_2 \times e_3))$

$T_{instances}(e_1 \times e_2) = \{o | \exists o_1 \in T_1\ \exists o_2 \in T_2 \wedge value(o) = value(o_1).identity(o_2)\}$

$T_{instances}(e_2 \times e_3) = \{o | \exists o_2 \in T_2\ \exists o_3 \in T_3 \wedge value(o) = identity(o_2)\ identity(o_3)\}$

$T_{instances}((e_1 \times e_2) \times e_3) = \{o | \exists\ o_{11} \in T_{instances}(e_1 \times e_2) \exists o_3 \in T_3 \wedge$

$value(o) = value(o_{11}).identity(o_3)\}$

$= \{o | \exists o_1 \in T_1 \exists o_2 \in T_2\ \exists o_3 \in T_3 \wedge value(o_{11}) = value(o_1).\ identity(o_2)$

$\Rightarrow value(o) = value(o_1).identity(o_2).\ identity(o_3)\}$

$= \{o | \exists o_1 \in T_1 \exists o_{13} \in T_{instances}(e_2 \times e_3) \wedge value(o_{13}) = identity(o_2).identity(o_3)$

$\Rightarrow value(o) = value(o_1).value(o_{13})\}$

$= T_{instances}(e_1 \times (e_2 \times e_3))$

3) $\exists x_1 \in X_1,\ len(x_1) = 1 \wedge \forall x_2 \in X_2,\ len(x_2) > 1 \wedge \exists x_3 \in X_3,\ len(x_3) = 1$

$M_e(e_1 \times e_2) = \{x | x \in ((m_1\ X_1) \cup X_2)\}$

$M_e(e_2 \times e_3) = \{x | x \in (X_2 \cup (m_3\ X_3))\}$

$M_e((e_1 \times e_2) \times e_3) = \{x | x \in (M_e(e_1 \times e_2) \cup (m_3\ X_3))\}$

$= \{x | x \in ((m_1\ X_1) \cup X_2 \cup (m_3\ X_3))\}$

$= \{x | x \in ((m_1\ X_1) \cup M_e(e_2 \times e_3))\} = M_e(e_1 \times (e_2 \times e_3))$

$T_{instances}(e_1 \times e_2) = \{o | \exists o_1 \in T_1 \exists o_2 \in T_2 \wedge value(o) = identity(o_1).value(o_2)\}$

$T_{instances}(e_2 \times e_3) = \{o | \exists o_2 \in T_2 \exists o_3 \in T_3 \wedge value(o) = value(o_2).identity(o_3)\}$

$T_{instances}((e_1 \times e_2) \times e_3) = \{o | \exists o_{11} \in T_{instances}(e_1 \times e_2) \exists o_3 \in T_3 \wedge$

$value(o) = value(o_{11}).identity(o_3)\}$

$= \{o | \exists o_1 \in T_1 \exists o_2 \in T_2 \exists o_3 \in T_3 \wedge value(o_{11}) = identity(o_1).value(o_2)$

$\Rightarrow value(o) = identity(o_1).value(o_2).identity(o_3)\}$

$= \{o | \exists o_1 \in T_1 \exists o_{13} \in T_{instances}(e_2 \times e_3) \wedge value(o_{13}) = value(o_2).identity(o_3)$

$\Rightarrow value(o) = identity(o_1).value(o_{13})\}$

$= T_{instances}(e_1 \times (e_2 \times e_3))$

4) $\exists x_1 \in X_1,\ len(x_1) = 1 \wedge \exists x_2 \in X_2,\ len(x_2) = 1 \wedge \forall x_3 \in X_3\ len(x_3) > 1$

$M_e(e_1 \times e_2) = \{x | x \in ((m_1\ X_1) \cup (m_2\ X_2))\}$

$M_e(e_2 \times e_3) = \{x | x \in ((m_2\ X_2) \cup X_3)\}$

$M_e((e_1 \times e_2) \times e_3) = \{x | x \in (M_e(e_1 \times e_2) \cup X_3)\}$

$$=\{x|x\in((m_1\ X_1)\cup(m_2\ X_2)\cup X_3)\}$$
$$=\{x|x\in((m_1\ X_1)\cup M_e(e_2\times e_3))\}=M_e(e_1\times(e_2\times e_3))$$

$T_{instances}(e_1\times e_2)=\{o|\exists o_1\in T_1\exists o_2\in T_2\ \wedge\ value(o)=identity(o_1).identity(o_2)\}$

$T_{instances}(e_2\times e_3)=\{o|\exists o_2\in T_2\exists o_3\in T_3\ \wedge\ value(o)=identity(o_2).value(o_3)\}$

$T_{instances}((e_1\times e_2)\times e_3)=\{o|\exists o_{11}\in T_{instances}(e_1\times e_2)\exists o_3\in T_3\ \wedge$
$value(o)=value(o_{11}).value(o_3)\}$
$=\{o|\exists o_1\in T_1\exists o_2\in T_2\exists o_3\in T_3\ \wedge\ value(o_{11})=identity(o_1).identity(o_2)$
$\Rightarrow\ value(o)=identity(o_1).identity(o_2).value(o_3)\}$
$=\{o|\exists o_1\in T_1\exists o_{13}\in T_{instances}(e_2\times e_3)\ \wedge\ value(o_{13})=identity(o_2).value(o_3)$
$\Rightarrow\ value(o)=identity(o_1).value(o_{13})\}$
$=T_{instances}(e_1\times(e_2\times e_3))$

5) $\forall x_1\in X_1,\ len(x_1)>1\ \wedge\ \exists x_2\in X_2,\ len(x_2)=1\ \wedge\ \forall x_3\in X_3\ len(x_3)>1$

$M_e(e_1\times e_2)=\{x|x\in(X_1\cup(m_2\ X_2))\}$

$M_e(e_2\times e_3)=\{x|x\in((m_2\ X_2)\cup X_3)\}$

$M_e((e_1\times e_2)\times e_3)=\{x|x\in(M_e(e_1\times e_2)\cup X_3)\}$
$=\{x|x\in(X_1\cup(m_2\ X_2)\cup X_3)\}$
$=\{x|x\in(X_1\cup M_e(e_2\times e_3))\}=M_e(e_1\times(e_2\times e_3))$

$T_{instances}(e_1\times e_2)=\{o|\exists o_1\in T_1\exists o_2\in T_2\ \wedge\ value(o)=value(o_1).identity(o_2)\}$

$T_{instances}(e_2\times e_3)=\{o|\exists o_2\in T_2\exists o_3\in T_3\ \wedge\ value(o)=identity(o_2).value(o_3)\}$

$T_{instances}((e_1\times e_2)\times e_3)=\{o|\exists o_{11}\in T_{instances}(e_1\times e_2)\exists o_3\in T_3\ \wedge$
$value(o)=value(o_{11}).value(o_3)\}$
$=\{o|\exists o_1\in T_1\exists o_2\in T_2\exists o_3\in T_3\ \wedge\ value(o_{11})=value(o_1).identity(o_2)$
$\Rightarrow\ value(o)=value(o_1).identity(o_2).value(o_3)\}$
$=\{o|\exists o_1\in T_1\exists o_{13}\in T_{instances}(e_2\times e_3)\ \wedge\ value(o_{13})=identity(o_2).value(o_3)$
$\Rightarrow\ value(o)=value(o_1).value(o_{13})\}$
$=T_{instances}(e_1\times(e_2\times e_3))$

6) $\forall x_1\in X_1,\ len(x_1)>1\ \wedge\ \forall x_2\in X_2,\ len(x_2)>1\ \wedge\ \exists x_3\in X_3\ len(x_3)=1$

$M_e(e_1\times e_2)=\{x|x\in(X_1\cup X_2)\}$

$M_e(e_2\times e_3)=\{x|x\in(X_2\cup(m_3\ X_3))\}$

$M_e((e_1\times e_2)\times e_3)=\{x|x\in(M_e(e_1\times e_2)\cup(m_3\ X_3))\}$
$=\{x|x\in(X_1\cup X_2\cup(m_3\ X_3))\}$
$=\{x|x\in(X_1\cup M_e(e_2\times e_3))\}=M_e(e_1\times(e_2\times e_3))$

$T_{instances}(e_1 \times e_2) = \{o | \exists o_1 \in T_1 \exists o_2 \in T_2 \wedge value(o) = value(o_1).value(o_2)\}$

$T_{instances}(e_2 \times e_3) = \{o | \exists o_2 \in T_2 \exists o_3 \in T_3 \wedge value(o) = value(o_2).identity(o_3)\}$

$T_{instances}((e_1 \times e_2) \times e_3) = \{o | \exists o_{11} \in T_{instances}(e_1 \times e_2) \exists o_3 \in T_3 \wedge$
$value(o) = value(o_{11}).identity(o_3)\}$
$= \{o | \exists o_1 \in T_1 \exists o_2 \in T_2 \exists o_3 \in T_3 \wedge value(o_{11}) = value(o_1).value(o_2)$
$\Rightarrow value(o) = value(o_1).value(o_2).identity(o_3)\}$
$= \{o | \exists o_1 \in T_1 \exists o_{13} \in T_{instances}(e_2 \times e_3) \wedge value(o_{13}) = value(o_2).identity(o_3)$
$\Rightarrow value(o) = value(o_1).value(o_{13})\}$
$= T_{instances}(e_1 \times (e_2 \times e_3))$

7) $\exists x_1 \in X_1,\ len(x_1) = 1 \wedge \forall x_2 \in X_2,\ len(x_2) > 1 \wedge \forall x_3 \in X_3\ len(x_3) > 1$

$M_e(e_1 \times e_2) = \{x | x \in ((m_1\ X_1) \cup X_2)\}$

$M_e(e_2 \times e_3) = \{x | x \in (X_2 \cup X_3)\}$

$M_e((e_1 \times e_2) \times e_3) = \{x | x \in (M_e(e_1 \times e_2) \cup X_3)\}$
$= \{x | x \in ((m_1\ X_1) \cup X_2 \cup X_3)\}$
$= \{x | x \in ((m_1\ X_1) \cup M_e(e_2 \times e_3))\} = M_e(e_1 \times (e_2 \times e_3))$

$T_{instances}(e_1 \times e_2) = \{o | \exists o_1 \in T_1 \exists o_2 \in T_2 \wedge value(o) = identity(o_1).value(o_2)\}$

$T_{instances}(e_2 \times e_3) = \{o | \exists o_2 \in T_2 \exists o_3 \in T_3 \wedge value(o) = value(o_2).value(o_3)\}$

$T_{instances}((e_1 \times e_2) \times e_3) = \{o | \exists o_{11} \in T_{instances}(e_1 \times e_2) \exists o_3 \in T_3 \wedge$
$value(o) = value(o_{11}).value(o_3)\}$
$= \{o | \exists o_1 \in T_1 \exists o_2 \in T_2 \exists o_3 \in T_3 \wedge value(o_{11}) = identity(o_1).value(o_2)$
$\Rightarrow value(o) = identity(o_1).value(o_2).value(o_3)\}$
$= \{o | \exists o_1 \in T_1 \exists o_{13} \in T_{instances}(e_2 \times e_3) \wedge value(o_{13}) = value(o_2).value(o_3)$
$\Rightarrow value(o) = identity(o_1).value(o_{13})\}$
$= T_{instances}(e_1 \times (e_2 \times e_3))$

8) $\forall x_1 \in X_1,\ len(x_1) > 1 \wedge \forall x_2 \in X_2,\ len(x_2) > 1 \wedge \forall x_3 \in X_3\ len(x_3) > 1$

$M_e(e_1 \times e_2) = \{x | x \in (X_1 \cup X_2)\}$

$M_e(e_2 \times e_3) = \{x | x \in (X_2) \cup X_3)\}$

$M_e((e_1 \times e_2) \times e_3) = \{x | x \in (M_e(e_1 \times e_2) \cup X_3)\}$
$= \{x | x \in (X_1 \cup X_2 \cup X_3)\}$
$= \{x | x \in (X_1 \cup M_e(e_2 \times e_3))\} = M_e(e_1 \times (e_2 \times e_3))$

$T_{instances}(e_1 \times e_2) = \{o | \exists o_1 \in T_1 \exists o_2 \in T_2 \wedge value(o) = value(o_1).value(o_2)\}$

$T_{instances}(e_2 \times e_3) = \{o | \exists o_2 \in T_2 \exists o_3 \in T_3 \wedge value(o) = value(o_2).value(o_3)\}$

$$T_{instances}((e_1 \times e_2) \times e_3) = \{o | \exists o_{11} \in T_{instances}(e_1 \times e_2) \exists o_3 \in T_3 \wedge value(o) = value(o_{11}).value(o_3)\}$$
$$= \{o | \exists o_1 \in T_1 \exists o_2 \in T_2 \exists o_3 \in T_3 \wedge value(o_{11}) = value(o_1).value(o_2) \Rightarrow value(o) = value(o_1).value(o_2).value(o_3)\}$$
$$= \{o | \exists o_1 \in T_1 \exists o_{13} \in T_{instances}(e_2 \times e_3) \wedge value(o_{13}) = value(o_2).value(o_3) \Rightarrow value(o) = value(o_1).value(o_{13})\}$$
$$= T_{instances}(e_1 \times (e_2 \times e_3))$$

Hence, proved. □

Chapter 5

The Object Algebra and Schema Evolution

5.1 Introduction

Object-oriented systems evolved to satisfy the needs of application areas where information about the domain is incomplete or becomes available incrementally or even highly subject to change and requires flexibility in changing the database schema. This necessitates that schema changes should be one of the basic characteristics to be considered in judging the power of an object-oriented database system. W. Kim classifies in [62] schema changes among a number of architectural concepts developed for relational database systems that should be treated in object-oriented data models. Schema changes allowed in the relational model are considered primitive as compared to schema changes within the realm of object-oriented databases. Extensibility is considered in [23] as one of the characteristics that a system must possess in order to be termed an object-oriented database system. Clearly, it is desirable for an object-oriented database system to satisfy as many schema operations as possible; a property not considered in [69].

Existing data models allow a wide variety of schema changes. O_2 [48, 107] provides two modes for running an application. Schema changes are allowed in the development mode, but not in the execution mode where the schema is frozen and changes to it are forbidden. Schema changes in ORION [28, 30, 31] are based on multiple inheritance while in GemStone [80] simple inheritance based schema changes are treated. In [90, 91]

the approach used is based on keeping versions to maintain a consistent view of the type lattice after a schema change.

A basic consideration in schema evolution is how to bring existing objects in line with a modified definition of an existing class. Either all instances of a modified class are instantaneously changed or they are modified only when used, otherwise remain unchanged. ORION [31] and ObServer [91] follow the approach known as screening where the change is delayed and values are either filtered or corrected as they are used. Another approach, known as conversion, is used by GemStone [80] where all instances of a class are modified in accordance with the change. The first approach sounds more sensible as there is no need to do something that may not be used.

In this chapter, we show how different schema evolution functions could be handled using the object algebra of chapter 4. The invariants and the conflict resolving rules are specified in section 5.2. Such constraints and rules are specified aiming at detecting and preventing any inconsistency in the database due to a schema change. In section 5.3 schema evolution functions are identified as either basic or derivable. After specifying the set of basic schema evolution functions, it is shown how the rest are derivable in terms of those included in the basic set. Accordingly, the use of an object algebra in handling the basic schema changes is discussed in this chapter. The motivation behind that is the recognition that some algebra operations perform the desired schema changes. Consequently, it is proven that schema changes could be achieved without having a stand alone language developed solely to serve the purpose as is the case in other systems. However, in our approach described in this chapter we benefit from having the object algebra properly maintaining the closure property in having both the operands and the output from a query possessing the same properties leading to a class. Such a class is naturally placed in the lattice thus facilitating the handling of class related schema changes. To the best of our knowledge, none of the query languages developed for object-oriented database systems could handle schema changes. This is so because those languages are mainly based on the nested relational model. Our previous work on schema evolution is reported in [20, 21].

5.2 Constraints of Schema Evolution and Conflict Resolving Rules

After schema changes the class lattice should preserve certain properties in order not to leave the database in an inconsistent state. This section includes the properties of the class lattice that are preserved upon schema modifications. In addition to the constraints there are some rules that are used to resolve conflicts due to schema changes. All of the constraints and rules discussed next in this section have been formally adapted into the model. They are triggered as the result of any schema change to detect whether such a change is allowed or not. A schema change which leaves the database in an inconsistent state is ignored.

Class Lattice Invariant

The class lattice is defined to be a directed acyclic graph with a single root, the OBJECT class. It is connected, i.e., there is at least a path from each class to the root OBJECT class. Any schema change to the class lattice should maintain this property. This property is never violated in our model because as indicated in chapter 3, a class has a set of superclasses. In general, a set is either empty or else contains some elements; no third case is possible concerning a set. Thus, an empty set of superclasses implicitly indicates a direct subclass of the root, while a non-empty set of superclasses includes the superclasses themselves.

Non-empty Class Invariant

A class whose instance and class variables and methods are deleted is considered empty. An empty class only connects its subclasses to its superclasses without adding any property to the connection and hence its presence is meaningless. The class lattice should not contain any empty user-defined classes. If a schema change results in an empty class, this class is automatically dropped. Formally this is detected by the following rule:

if $instances(c) = \phi \wedge messages(c) = \phi$ then
  for any class c_i such that $c \in supers(c_i)$
    $supers(c_i) := supers(c_i) - \{c\} + supers(c)$
endif

DISTINCT NAMES INVARIANT

No two classes can have the same name. Further, no two methods defined within the same class can have the same name. In addition, the instance variables within the same class should have distinct names. Name conflicts resulting from the inheritance of objects and methods are resolved according to the *name conflict resolving rule* and *the inheritance priority rule*, given next in this section. Formally, distinct names requirement is enforced by the following three rules:

if $\exists m_i \in messages(c) \wedge m_i = new_{message}$ then ignore endif

if $\exists iv_i \in I_{variables}(c) \wedge iv_i = new_{instance\ variable}$ then ignore endif

if $\exists c_i \in schema \wedge c_i = new_{class}$ then ignore endif

NAME CONFLICT RESOLVING RULE

A name conflict occurs when a message (instance variable) in a superclass of a class c has the same name as a message (instance variable) in the class c or in any of the other superclasses of the class c. Class c is called the target class.

On name conflicts priority is given to the target class and the priority decreases while going away from the target class towards the OBJECT class. For classes that have the same priority according to this rule, i.e., classes found in the same superclass list of the target class, the conflict is resolved according to the order in the list. The class found at the head of the list is given the highest priority while that at the tail of the list is given the lowest priority. The OBJECT class has the least priority. Formally, name conflicts due to messages and instance variables are resolved according to the following two rules, respectively.

Let m_i and iv_i be the message and the instance variable in a class c related to which a name conflict is to be detected between the class c and its superclasses $c_1, c_2, ..., c_n$.

if $\exists c_i, c_j \in \{c_1, c_2, ..., c_n\} \wedge m_i \in messages(c_i) \wedge m_i \in messages(c_j)$ then
  if $length(path(c, c_i)) > length(path(c, c_j))$ then
    $m_{i_{c_j}}$ is inherited
  elseif $length(path(c, c_i)) < length(path(c, c_j))$ then
    $m_{i_{c_i}}$ is inherited
  elseif $length(path(c, c_i)) = length(path(c, c_j))$ then

if c_i *precedes* c_j then
$m_{i_{c_i}}$ is inherited
else
$m_{i_{c_j}}$ is inherited
endif
endif
endif

if $\exists c_i, c_j \in \{c_1, c_2, ..., c_n\} \wedge iv_i \in I_{variables}(c_i) \wedge iv_i \in Ivariables(c_j)$ then
if $length(path(c, c_i)) > length(path(c, c_j))$ then
$iv_{i_{c_j}}$ is inherited
elseif $length(path(c, c_i)) < length(path(c, c_j))$ then
$iv_{i_{c_i}}$ is inherited
elseif $length(path(c, c_i)) = length(path(c, c_j))$ then
if c_i *precedes* c_j then
$iv_{i_{c_i}}$ is inherited
else
$iv_{i_{c_j}}$ is inherited
endif
endif
endif

These two rules detect and resolve name conflicts between two classes; a class and any of its superclasses or two superclasses of a class.

Inheritance Priority Rule

The addition of an instance variable (method) to a class and the deletion of an instance variable (method) from a class may cause conflicts that were resolved according to the name conflict resolving rule. An instance variable (method) in name conflict with another one may be inherited on deleting the latter. Further, currently inherited instance variables (methods) may cease to be considered after the addition of an instance variable (method) with the same name to a class with a higher priority. To detect this, on adding an instance variable or a method to a class c, the corresponding name conflict resolving rule is executed accordingly.

Full Inheritance Invariant

A class inherits all the instance variables and methods from its superclass(es). No selection is done but name conflicts are resolved according to the name conflict resolving rule and the inheritance priority rule.

Homogeneous Domain Invariant

The domain for each instance variable should be bound to a specific class in the class lattice. When a class is specified as a domain, all its direct and indirect subclasses may be used in the same context because a class includes instances of its all direct and indirect subclasses. In other words, $T_{instances}(c)$ includes $T_{instances}(c_i)$ for any class c_i being a subclass of the class c. Hence, having a domain being specified as $T_{instances}(c)$ or $2^{T_{instances}(c)}$ lead to the fact that $T_{instances}(c_i)$ or $2^{T_{instances}(c_i)}$, respectively, substitutes that domain. Formally, this is detected and resolved by the following two rules:

if $domain(iv) = T_{instances}(c)$ then
    for every class c_i such that $c \in supers(c_i)$
        $domain(iv) := T_{instances}(c_i)$ is allowed.
endif

if $domain(iv) = 2^{T_{instances}(c)}$ then
    for every class c_i such that $c \in supers(c_i)$
        $domain(iv) := 2^{T_{instances}(c_i)}$ is allowed.
endif

Class Addition/Deletion Invariant

A new superclass c of an existing class c_i should be added as the last superclass in the order of superclasses of the class c_i. Moreover, on deleting c_i from the class lattice its supers should replace it as immediate superclasses of the immediate subclasses of class c_i. Superclasses of the class c_i are added at the end of the list because all such classes have lower priority than those classes which were in the same list with the class c_i.

Formally, this is detected and resolved according to the following two rules:

if c_i is added to $supers(c)$ then
$$supers(c) := supers(c) + \{c_i\}^*$$
endif

if c_i is deleted from $supers(c)$ then
for every class c_j such that $c_i \in supers(c_j)$
$$supers(c_j) := supers(c_j) + supers(c_i)$$
endif

Finally, the OBJECT class should be the superclass of a newly added class unless the supers of the added class are explicitly specified.

5.3 Handling Schema Evolution Functions Using the Object Algebra

A taxonomy of some schema evolution function is found in [20]. Schema evolution functions could be enumerated are follows:-

1. Changes to instance variables of a class
   - 1.1. Add an instance variable to a class
   - 1.2. Drop an instance variable from a class
   - 1.3. Change the name of an instance variable of a class
   - 1.4. Change the domain of an instance variable of a class
2. Changes to the methods of a class
   - 2.1. Add a new method
   - 2.2. Drop an existing method
   - 2.3. Change the name of an existing method
3. Changes to classes, i.e., changes to the structure of the class lattice

*Class c_i is appended at the end of $supers(c)$

3.1. Add a new class

3.2. Drop an existing class

3.3. Change the name of a class

3.4. Add a class to the superclass list of another class

3.5. Drop a class from the superclass list of another class

Of these schema evolution functions the following are considered basic in terms of which the other ones are derivable. Thus, the basic schema evolution functions are:

1.1. Add an instance variable to a class

1.2. Drop an instance variable from a class

2.1. Add a new method

2.2. Drop an existing method

3.1. Add a new class

3.2. Drop an existing class

3.3. Change the name of a class

3.4. Add a class to the superclass list of another class

3.5. Drop a class from the superclass list of another class

The rest of the schema evolution functions are derivable in terms of the basic ones as illustrated in what follows.

1.3 is derivable as 1.2 followed by 1.1

1.4 is derivable as 1.2 followed by 1.1

2.3 is derivable as 2.2 followed by 2.1

2.4 is derivable as 2.2 followed by 2.1

2.5 is derivable as 2.2 followed by 2.1

3.3 is derivable as 3.2 followed by 3.1

In the rest of this section, we show how the basic schema evolution functions can be handled using the operators of the object algebra described in chapter 4.

1.1. *Add an instance variable with domain c_1 to class c*

According to the distinct names invariant, the new instance variable should have a name distinct from all other instance variables in its class. Further, the addition of a new instance variable may affect the inheritance priority according to the name conflict resolving rule and the inheritance priority rule. Thus,

$$c := c >> c_1$$

Notice that the result of the nest operation is considered to be a subclass of the first operand c. However, the assignment is used to have this result replacing the class c itself. By this way, the instance variable iv_1 with $domain(iv_1)=T_{instances}(c_1)$ is added to $I_{variables}(c)$ (the instance variables of the class c). This is illustrated in the following examples.

Example 5.1 *Assign to every person his brothers.*

$$person := person\sharp p >> person\sharp p_1[p_1\ sex() = "M" \wedge \exists\, p_2 \in T_{instances}(person) \wedge \{p, p_1\} \subseteq p_2\ children()]$$

In this example, the value of the new instance variable in any object o of the person class is the identities of those persons who are children of the same person with o, if any; otherwise it is nil.

Example 5.2 *Assign to every person his parents*

To get the actual value of the instance variable showing the parents of every person being assigned as it is the case with the instance variable showing the brothers in example 5.1, the selection operation is used together with the nest operation as follows:

$$person := person\sharp p >> person\sharp p_1\sharp p_2[p_1\ children() = p_2\ children() \wedge p \in p_1\ children() \wedge p_1\ sex() \neq p_2\ sex()]$$

In this formulation, the value of the new instance variable is automatically set either to object identities of the couple having the same set of children including

the given person, or to *nil* in case of not having any such couple. Notice that $p_1\ sex() \neq p_2\ sex()$ is included in the predicate expression to avoid having p_1 and p_2 taking the same value.

1.2. *Drop from $I_{variables}(c)$ the instance variable whose domain is specified to be the class c_1 (the corresponding value in each object is handled via the message $m()$):*

Inheritance priority may change according to the name conflict resolving rule and the inheritance priority rule.

$$c := c[messages(c) - \{m()\}]$$

where $m() \in messages(c)$ and $m()$ handles the values of the instance variable $iv \in I_{variables}(c)$ with $domain(iv) = T_{instances}(c_1)$

Example 5.3 *Drop the instance variable whose value is handled via the message prerequisites() from the course class*

$$course := course[messages(course) - \{prerequisites()\}]$$

It is an implementation issue to decide on whether the value of the deleted instance variable is to be physically dropped from the objects in $T_{instances}(course)$ or not. In our model the only means which could be used to access the values constituting an object are the corresponding messages. Thus, after dropping the message that could access the prerequisites of a course, it will be impossible to access that value inside any of the objects in $T_{instances}(course)$, although it is present.

2.1. *Add to the methods of class c_1 one or more methods from class c_2 with their corresponding messages being $\{m_1, m_2, ..., m_i\}$:*

This could be performed because it is possible to add a method to the definition of an existing class. But distinct names invariant should be preserved by forcing the name of the new method to be distinct from those existing in the same class. Moreover, inheritance priority may change following both the inheritance priority rule and the name conflict resolving rule. Consequently,

$$c_1 := c_1]c_2 : \{m_1, m_2, ..., m_i\}[$$

However, for the case of having $m_1, m_2, ... m_i$ being new methods, the following formulation is valid:

c_1:=c_1]{$m_1 : f_1, m_2 : f_2, ..., m_i : f_i$}[

Example 5.4 *Add to the staff class the method net-salary(i) which deducts taxes at the rate of i from the salary.*

*staff := staff]{net-salary(i):f(o,i)=o salary() * (1-i)}[*

The message *net-salary(i)* with $0 \leq i \leq 1$, could be used to invoke the new method added to the staff class to implement the function $f(o, i)$ where o is an object variable bound to objects of the staff class, i.e., $o \in T_{instances}(staff)$ and indicates the receiver of the message. This method is automatically implemented; it is an implementation issue out of the scope of this thesis.

2.2. *Drop one or more methods from class c_1, their corresponding messages being $\{m_1, m_2, ..., m_i\}$:*

A method may be deleted after it becomes irrelevant due to some schema changes. A deleted method may be replaced by a new method added to the class definition or an existing method in another class according to both the inheritance priority rule and the name conflict resolving rule. This is done as:

c_1:=c_1[messages(c_1)−{$m_1, m_2, ..., m_i$}]

By this way, all message expressions which are prefixed by a message drawn from the set $\{m_1, m_2, ..., m_i\}$ are dropped from $M_e(c_1)$.

Finally, it is important to indicate that the instance variable deletion schema evolution function is recognized to be a special case of this schema evolution function.

3.1. *Add a class c to the lattice with the domains of its instance variables $iv_1, iv_2, ..., iv_n$ being $T_{instances}(c_1), T_{instances}(c_2), ..., T_{instances}(c_n)$, respectively.*

A new class may either have the OBJECT class as a direct superclass or else other existing classes in its superclass list. Furthermore, a new class may have zero,one or more subclasses. The non-empty class constraint and class lattice invariant should be maintained. According to the distinct names invariant, the name of the new class should not match with any of the existing classes. Thus,

$c := OBJECT >> c_1 >> ... >> c_n$

The OBJECT class is used to have the new class c as a direct subclass of the root. If the class c is desired to be a direct subclass of an existing class, say c_p, OBJECT is replaced by the class c_p in the above formulation. All the example classes given in chapter 4 could be defined by utilizing this schema evolution function. This is illustrated in the following example where it is shown how the *department* class is defined.

Example 5.5 *Add the class department to the lattice as a subclass of OBJECT with its instance variables having the domains string and staff, respectively.*

$$department := OBJECT >> string >> staff$$

Notice that the department class is a subclass of the root OBJECT class with:

$I_{variables}(department) = \{iv_1 : string, iv_2 : staff\}$
$messages(department) = \{m_1(), m_2()\}$, corresponding to the two instance variables iv_1 and iv_2, respectively.
$M_e(department) = messages(department) \bigcup (m_2()\, M_e(staff))$
$T_{instances}(department) = instances(department) = \phi$, because of a new class with no subclasses.

3.2. *Drop an existing class c from the lattice:*

Let $T_{instances}(c_1), ..., T_{instances}(c_n)$ be the domains of the instance variables defined in class c without being inherited, with their corresponding messages being $m_1(), m_2(), ..., m_n()$.
When all the definition and contents of the class c are dropped, then class c should be deleted as an empty class to preserve the non-empty class invariant. The immediate supers of class c replace it in the inheritance mechanism for not to leave its subclasses dangling and violate the class lattice invariant. After the deletion, the inheritance priority may change according to the inheritance priority rule and the name conflict resolving rule. Thus,

$$c := c[\{\}]$$

will automatically drop class c due to it is being empty to maintain the non-empty class invariant.

3.4. *Add a class c_1 to the superclass list of the class c; c_1 has instance variables $iv_1, iv_2, ..., iv_n$ with domains $T_{instances}(c_2), T_{instances}(c_3), ..., T_{instances}(c_n)$, respectively*

A class added to the superclass list of another existing class should not violate the class addition/deletion invariant. Thus, this is handled as:

$c := c >> c_2 >> ... >> c_n$

$c1 := c[\{m_1(), m_2(), ..., m_n()\}]$

where $\{m_1(), m_2(), ..., m_n()\}$ are the messages corresponding to the instance variables of the class c_1.

This is true for the case of c_1 being a new class. However, for c_1 being an existing class, the following is done:

$c := (c >> c_1)![messages(c) \bigcup \{m()\ messages(c_1)\}]$

where $m()$ is the message added to $c >> c_1$ and $m()$ corresponds to the instance variable iv_1 where $domain(iv_1) = T_{instances}(c_1)$.

$c1 := c[messages(c_1)]$

3.5. *Remove class c_1 from the superclass list of class c:*

$c := c[M_e(c) - M_e(c_1)]$

Schema evolution functions within the realm of object-oriented databases were discussed in this chapter. However, the approach followed is different from all the other approaches described in the literature. The approach described in this chapter is based on the object algebra. Having the object algebra properly maintaining the closure property and facilitating the placement of the output from a query as a class in the lattice, were the basic properties that led to the study described in this chapter. Thus, it is proved that a wide variety of schema changes could be handled using the object algebra without requiring a stand alone language to serve the purpose. While achieving that, schema evolution functions were identified to be either basic or derivable with functions in the latter being representable in terms of functions in the former. Thus, in this chapter only the basic schema evolution functions were considered after showing how to derive the others in terms of them. Different rules and constraints were also derived to detect and resolve conflicts and inconsistencies due to any schema change.

Chapter 6

Conclusions

In this thesis, we described a formal query model for object-oriented database systems. Our query model is not restricted to handle existing objects only, however, the introduction of new relationships as well as new objects is also facilitated. A new relationship could have a stored value by extending objects in the operand to include values for new instance variables. It is also possible for a new relationship to have a derived value in terms of existing values by extending the behavior of the operand to facilitate the derivation of the required relationship. Operands and the output of a query are defined to have a pair of sets, a set of objects and a set of message expressions. Thus having the characteristics of an operand, the output from a query could itself be an operand and hence the closure property is naturally maintained without having non-object-oriented constructs introduced into the model preventing the violation of object-oriented features.

A message expression results in the evaluation of the underlying methods and in the same sequence as if they all together form a single method invoked by that message expression. Furthermore, message expressions are used in the invocation of behavior as well as behavior constructors. Also, message expressions facilitate accessing of stored and derived values leading to computational completeness without having an embedded query language leading to impedance mismatch. Consequently, methods could be coded solely by utilizing the object algebra and hence simplify the optimization process. This is possible after being able to replace a message by the object algebra expression constituting the body of the method implementing the corresponding function. On the other hand, proposals that do not overcome the impedance mismatch problem are still

suffering from an inability to support full optimization. As a first step towards query optimization, equivalents of object algebra expressions are derived and the associativity of the cross-product operation is proved for being an essential requirement in improving the performance while optimizing queries.

The operators of our object algebra subsumes those of the relational and nested algebras and hence it is more powerful than either one. The equal handling of objects as well as the behavior defined on them is an important requirement of an object algebra; thus we satisfied it in the presented query model. This is due to the presence of data and behavior in an object-oriented data model in contrast to having only data in the relational data model. Behavior is handled via message expressions. We support aggregate functions whose outputs are also pairs of sets like any operand.

We started by defining a set of objects and a set of message expressions for a class. Having such a pair, a class is shown to be an operand. By this, some operands were defined to be existing classes. Other operands are defined to be the outputs of queries. As the only known characteristics of the output from a query are a pair of sets -a set of objects and a set of message expressions, we have proven that from such a pair other class characteristics can be derived. Having the characteristics of a class, the output from a query is in fact a class. Thus, we decided on the proper placement of such a class in the lattice. This is done by first deciding on the inheritance relationship between such a class and other existing classes through the operand(s) involved in the query. Also, we have shown how reusability due to inheritance could be maximized by minimizing the facilities defined inside the class that corresponds to the output of a query, when such facilities could be inherited. Otherwise, more effort will be required to maintain the database consistency in case of duplicate definitions in two or more classes.

Finally, we noted the possibility of supporting linear recursion without any need for a particular operator to serve the purpose. Later, this was recognized to significantly improve the power of the described object algebra for having real life recursive rules being linear in nature. A future extension of the described object algebra could be to provide support for recursive queries in general. For instance, such an extension should facilitate the treatment of recursion involved in the shortest path, the longest path and the critical path problems or the like.

At the end, the representation of different schema evolution functions using the

developed object algebra is shown. This is considered important for schema evolution being a basic characteristic distinguishing object-oriented databases from conventional databases.

6.1 Contributions and Enhancements

This thesis aimed at locating and overcoming the shortages and drawbacks encountered in the research related to the definition of a query model for object-oriented databases as reported in the current literature. The contributions and enhancements to the field due to our research presented in this work could be summarized as follows:

1. The closure property is naturally maintained. For this purpose, operands and the output from a query were defined to have pairs of sets -a set of objects and a set of message expressions- where objects could be accessed solely via the corresponding message expressions. By this, encapsulation is maintained.

2. Operators of the described algebra handle both of the constituent sets of the pair of an operand. Hence, not only objects are manipulated structurally, also the corresponding behavior reflected by the corresponding set of message expressions is handled. On deriving the output from a query, a set of message expressions as well as the set of objects are determined.

3. Although the output from a query is characterized by the pair of sets derived starting with those of the operand(s), it actually has the characteristics of a class. We have shown how to derive such class characteristics starting from the pair, the set of objects and the set of message expressions. Furthermore, we handled the proper and natural placement of such a class in the lattice by considering the inheritance relationship with the class(es) due to the operand(s). Moreover, we have shown how it is possible to maximize reusability when the class corresponding to the output from a query is a superclass of the operand(s). Reusability is maximized by considering the superclasses of the operand(s) to be the superclasses of the class which corresponds to the output of a query.

4. Method migration is also facilitated after defining the *inverse project* operation in terms of the *nest*, the *one-level-project* and the *project* operations. Such an

operation makes it possible to move a method from one class to another, e.g., from a class to any of its subclasses and vice versa. Thus, this operation helps to either extend or else shrink the behavior defined for objects of a given class.

5. Aggregate functions are supported with the result of such queries still having the characteristics of an operand. This, together with message expressions which handle both stored and derived values, leads to computational completeness without any need to have an embedded query language.

6. By handling the proper and natural placement of the class obtained as the result of a query, we have minimized the assignment of new identities to objects found in the result of a query. In other words, we have shown that a new class due to a query result could be placed in the class lattice obeying one of the following three cases; a subclass of the operand(s), a superclass of the operand(s) or a direct subclass of the root OBJECT class. New identities are assigned to the objects in the result of a query only for the case of having the corresponding class as a direct subclass of the root OBJECT class. On the other hand, when the result is a superclass of the operand(s), the same objects in the operand(s) are considered in the result. While for the case of having the result as a subclass of the operand, objects migrate from the operand pair to the result pair and still considered to be in the operand pair. This is so because a class includes objects in all of its subclasses in addition to the other objects added to the class itself.

7. As most recursive queries are linear and efficient processing strategies have been developed to deal with recursive queries, we have shown how problems involving linear recursion could be handled using the described object algebra without any need for additional operators. A recursive query is considered to be a selection with one of the object variables appearing in the corresponding predicate expression being bound to objects in the result.

8. Schema evolution has been accepted as an important feature of an object-oriented data model. Consequently, we have shown how the basic schema evolution functions could be handled using the described object algebra.

6.2 Further Research Directions

We believe that the research presented in this thesis forms the basis for further extensive research on the subject. Possible further research areas could be enumerated as follows:

- In this thesis, only linear recursion is considered. Although linear recursion covers most of the problems that involve recursion, an extension to general recursion may be interesting to handle many problems of *Graph Theory.*

- Query optimization based on equivalent object algebra expressions presented in section 4.8 could be done after representing an object algebra expression as a tree that has the following characteristics:

  An object algebra expression is the inorder traversal of a binary tree with the following characteristics:

  - a leaf node is a pair $<T_{instances}(e), M_e(e)>$ for some object algebra expression e.
  - a non-leaf node at level $i \geq 0$ is an object algebra operator applicable to its child node(s) at level $i+1$.
  - a single child of a node is a left child.□

  Such a binary tree helps in optimizing object algebra expressions.

- A user query language and its mapping into the object algebra could be another area of research. For instance, an object SQL could be developed in the same sense as that developed for the ORION system.

- An equivalent object calculus in the same sense as that done by Straube is another area of research. In the calculus of Straube, as it is the case with his equivalent algebra, the closure property is not considered. Developing an object calculus that maintains the closure property may be interesting.

- Implementing the described object algebra as a part of an existing object-oriented database system could be another area of research. Following this, aspects of query optimization at the physical level depending on the storage and indexing facilities of such a database system could be another possible area of research.

Bibliography

[1] S. Abiteboul and A. Bonner, "Objects and Views," *Proceedings of the ACM-SIGMOD International Conference on Management of Data,* pp.238-247, 1991.

[2] S. Abiteboul and C. Beeri, "On the Power of Languages for the Manipulation of Complex Objects," INRIA, Tech.Rep.No. 846, May 1988.

[3] S. Abiteboul and P.C. Kanellakis, "Object Identity as a Query Language Primitive," *Proceedings of ACM-SIGMOD International Conference on Management of Data,* pp.159-173, 1989.

[4] R. Agrawal, "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries," *IEEE Transactions on Software Engineering,* Vol. 14, July 1988.

[5] A. Alashqur, S.Y. Su and H. Lam, "OQL: A Query Language for Manipulating Object-Oriented Databases," *Proceedings of the* 15^{th} *International Conference on Very Large Databases,* Amsterdam, pp. 433-442, August 1989.

[6] A. Alashqur, S.Y. Su and H. Lam, "A Rule Based Language for Deductive Object-Oriented Databases," *Proceedings of the* 6^{th} *IEEE International Conference on Data Engineering,* Los Angeles, CA, pp. 58-67, February 1990.

[7] A. Albano, L. Cardelli and R. Orsini, "Gelileo: A Strongly-Typed Interactive Conceptual Language," *ACM Transactions on Database Systems,* Vol. 10, No. 2, pp. 230-260, 1985.

[8] R. Alhajj (Al-Hajj), "Storage Management and Indexing in Object-Oriented Database Management Systems," M.S. Thesis, Bilkent University, Turkey, June 1990.

[9] R. Alhajj (Al-Hajj) and M.E. Arkun "A Model for Storage Management in Object-Oriented Database Management Systems," *Proceedings of the* 5^{th} *International Symposium on Computers and Information Sciences,* Capadocia, October 1990.

[10] R. Alhajj (Al-Hajj) and M.E. Arkun "A Model for Indexing in Object-Oriented Database Management Systems," *Proceedings of the* 5^{th} *International Symposium on Computers and Information Sciences,* Capadocia, October 1990.

[11] R. Alhajj (Al-Hajj), A.U. Tansel and M.E. Arkun "Version Management in an Object-Oriented Database Management System," *Proceedings of the* 6^{th} *International Symposium on Computers and Information Sciences,* Antalya, October 1991.

[12] R. Alhajj (Al-Hajj), "A Query Model and a Query Language for Object-Oriented Database Systems," Technical Report, Bilkent University, Turkey, 1991.

[13] R. Alhajj (Al-Hajj) and M.E. Arkun, "A Data Model for Object-Oriented Databases," *Proceedings of the* 6^{th} *International Symposium on Computers and Information Sciences,* Antalya, October 1991.

[14] R. Alhajj (Al-Hajj) and M.E. Arkun, "A Formal Data Model and Object Algebra for Object-Oriented Databases," *Applied Mathematics and Computer Science,* Vol. 2, No. 1, pp. 49-63, 1992.

[15] R. Alhajj (Al-Hajj) and M.E. Arkun, "A Query Language for Object-Oriented Databases," *Proceedings of the* 7^{th} *International Symposium on Computers and Information Sciences,* Kemer-Antalya, November 1992.

[16] R. Alhajj (Al-Hajj) and M.E. Arkun, "Queries in Object-Oriented Database Systems," *Proceedings of the ISMM International Conference on Information and Knowledge Management,* Maryland, November 1992, *An extended version of this paper will appear in T. Finin, Y. Yesha and K. Nicholas (eds.) Lecture Notes in Computer Science, Berlin: Springer*

[17] R. Alhajj (Al-Hajj) and M.E. Arkun "An Object Algebra for Object-Oriented Database Systems," *(Accepted paper) The ACM SIGBIT Journal of Data Base.*

[18] R. Alhajj (Al-Hajj) and M.E. Arkun "A Formal Object-Oriented Query Model and an Object Algebra," *To be presented at the NATO-ASI Summer School on Object-Oriented Databases, IZMIR To be published as Lecture Notes in Computer Science, Berlin: Springer.*

[19] R. Alhajj (Al-Hajj) and M.E. Arkun, "A Query Model for Object-Oriented Database Systems," *Proceedings of the* 9^{th} *IEEE International Conference on Data Engineering,* Vienna, April 1993 (to appear).

[20] R. Alhajj (Al-Hajj) and M.E. Arkun, "A Schema Modification Methodology for an Object-Oriented Database System," *Proceedings of the XVIII Latin America Conference on Computers, PANEL'92,*

[21] R. Alhajj (Al-Hajj) and M.E. Arkun, "Schema Changes in an Object-Oriented Database System," *Proceedings of the ASME European Joint Conference on Systems Design and Analysis,* Istanbul, June 1992,

[22] R. Alhajj (Al-Hajj) and M.E. Arkun, "Object-Oriented Query Language," *(Accepted paper) Journal of Information and Software Technology.*

[23] M. Atkinson, et al., "The Object-Oriented Database System Manifesto," *Proceeding of the International Conference on Deductive Object-Oriented Databases,* Kyoto, Japan, December 1989.

[24] F. Bancilhon, et.al., "FAD: A Powerful and Simple Database Language," *Proceedings of the* 13^{th} *International Conference on Very Large Databases.* Brighton, pp. 97-105, 1987.

[25] F. Bancilhon and A. Ramakrishinan, "An Amateur's Introduction to Recursive Query Processing Strategies," *Proceedings of ACM-SIGMOD International Conference on Management of Data,* May 1986.

[26] F. Bancilhon, et.al.,"A Query Language for the O_2 Object-Oriented Database System," *Proceedings of the* 2^{nd} *International Workshop on Database Programming Languages,* June 1989.

[27] F. Bancilhon, "Object-Oriented Database Systems," *Proceedings of the 7^{th} ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems,* pp.152-162, August 1988.

[28] J. Banerjee, et al., "Data Model Issues for Object-Oriented Applications," *ACM Transactions on Office Information Systems,* Vol. 5, No. 1, pp. 3-26, 1987.

[29] J. Banerjee, W. Kim and K.C. Kim, "Queries in Object-Oriented Databases," *Proceedings of the 4^{th} IEEE International Conference on Data Engineering,* Los Algeles, CA, pp. 31-38, February 1988.

[30] J. Banerjee, et al., "Schema Evolution in Object-Oriented Persistent Databases," *Proceedings of the 6^{th} Advanced Database Symposium, Information Processing Society of Japan's Special Interest Group on Database Systems,* Tokyo, August 1986, pp.23-31.

[31] J. Banerjee, et al., "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," *Proceedings of ACM-SIGMOD International Conference on Management of Data,* San Francisco, CA, May 1987.

[32] D. Beech, "A Foundation for Evolution from Relational to Object Databases," *Proceedings of the International Conference on Extending Data Base Technology,* March 1988.

[33] C. Beeri, "Formal Model for Object-Oriented Databases," *Proceedings of the 1^{st} International Conference on Deductive and Object-Oriented Databases,* pp. 370-395, December 1989.

[34] C. Beeri and Y. Kornatzky, "Algebraic Optimization of Object-Oriented Query Languages," *Proceedings of the 3^{rd} International Conference on Database Theory,*, Paris, pp. 72-88, December 1990.

[35] E. Bentino, et.al., "Object-Oriented Query Languages: The Notion and the Issues," *IEEE Transactions on Knowledge and Data Engineering,* Vol. 4, No. 3, June 1992.

[36] J. Borgida, et al., "Classic: A Structural Data Model for Objects," *Proceedings of ACM-SIGMOD International Conference on Management of Data,* pp.58-67, 1989.

[37] P.Butterworth, A. Otis and J. Stein, "The GemStone Object-Oriented Database Management System," *Communications of ACM,* Vol. 34, No. 10, 1991.

[38] T. Bozkaya, et.al " An Object Memory for an Object-Oriented Database Management System," *Proceedings of the DECSYM,* Side, March 1992.

[39] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction and Polymorphism," *ACM Computing Surveys,* Vol. 17, No. 4, pp.471-522, December 1985.

[40] M.J. Carey and D.J. Dewitt, "The Architecture of the EXODUS Extensible DBMS," *Proceedings of the IEEE International Workshop on Object-Oriented Database Systems,* Pacific Grove, CA, pp. 52-65, September 1986.

[41] M.J. Carey, D.J. DeWitt and S.L. Vandenberg, "A Data Model and a Query Language for EXODUS," *Proceedings of ACM-SIGMOD International Conference on Management of Data,* Chicago, pp. 413-423, May 1988.

[42] S. Cluet, et. al., "Reloop, an Algebra Based Query Language for an Object-Oriented Database System," *Proceedings of the* 1^{st} *International Conference on Object-Oriented and Deductive Databases,* December 1989.

[43] S. Cluet and C. Delobel, "A General Framework for the Optimization of Object-Oriented Queries," *Proceedings of ACM-SIGMOD International Conference on Management of Data,* San Diego, CA, June 1992.

[44] L. Colby, "A Recursive Algebra and Query Optimization for Nested Relations," *Proceedings of ACM-SIGMOD International Conference on Management of Data,* pp. 273-283, 1989.

[45] S. Daniels, et.al., "Query Optimization in Relevation, an Overview," *IEEE Data Engineering Bulletin,* Vol. 14, No. 2, June 1991.

[46] C.J. Date, *An Introduction to Database Systems,* Fourth Edition, Vol. 1 and Vol. 2, Addison-Wesley, 1986.

[47] U. Dayal, "Queries and Views in an Object-Oriented Data Model," *Proceedings of the 2^{nd} International Workshop on Database Programming Languages,* pp. 80-102, June 1989.

[48] O. Deux, et al., "The Story of O_2," *IEEE Transactions on Knowledge and Data Engineering,* Vol.2, No.1, pp. 91-108, March 1990.

[49] O. Deux, et.al., "The O_2 System," *Communication of ACM,* Vol. 34, No. 10, 1991.

[50] K. Dittrich, "Object-Oriented Database Systems: The Notion and the Issues," *Proceedings of IEEE Workshop on Object-Oriented DBMS,* September 1986.

[51] G. Eray, et.al "Storage and Indexing Facilities of an Object-Oriented Database Management System," *Proceedings of the DECSYM,* Side, March 1992.

[52] D.H. Fishman, et al., "IRIS: An Object-Oriented Database Management System," *ACM Transactions on Office Information Systems,* Vol. 5, No. 1, pp. 48-69, 1987.

[53] G. Gardarin and P. Valduriez, "ESQL2: An Extended SQL2 with F-Logic Semantics," *Proceedings of the 10^{th} IEEE International Conference on Data Engineering,* Phoenix, Az, February 1992.

[54] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation,* Addison Wesley, 1983.

[55] M. Hammer and D.J. McLeod, "Database Description with SDM: A Semantic Data Model," *ACM Transactions on Database Systemsu,* Vo. 6, No. 3, pp.351-386, 1981.

[56] M.F. Hornick and S.B. Zdonik, "A Shared Segmented Memory System for an Object-Oriented Database," *ACM Transactions on Office Information Systems,* Vol. 5, No. 1, pp. 70-95, 1987.

[57] G. Jaeschke and H.J. Schek, "Remarks on the Algebra of Non-First Normal Form Relations," *Proceedings of the Symposium on Principles of Database Systems,* pp. 127-138, March 1982.

[58] S.N. Khoshafian and G.P. Copeland, "Object Identity," *Proceedings of the ACM International Conference on Object-Oriented Programming Systems, Languages and Applications,* Portland, OR, pp. 406-416, September 1986.

[59] M. Kifer, W. Kim and Y. Sagiv, "Querying Object-Oriented Databases," *Proceedings of ACM-SIGMOD International Conference on Management of Data,* San Diego, CA, June 1992.

[60] M. Kifer, Y. Sagiv and W. Kim, "A First-Order Query Language for Object-Oriented Databases," *Technical Report, UNISQL,* 1992.

[61] W. Kim, "A Model of Queries for Object-Oriented Databases," *Proceedings of the* 15^{th} *International Conference on Very Large Databases,* Amsterdam, pp. 423-432, 1989.

[62] W. Kim, "Object-Oriented Databases: Definition and Research Directions," *IEEE Transactions on Knowledge and Data Engineering,* Vol. 2, No. 3, pp. 327-341, 1990.

[63] W. Kim and F.H. Lochovsky, eds., *Object-Oriented Concepts, Databases and Applications,* Reading MA: Addison-Wesley, 1989.

[64] K-C. Kim, W. Kim and A. Dale, "Cyclic Query Processing in Object-Oriented Databases," *Proceedings of the* 5^{th} *IEEE conference on Data Engineering,* Los Angeles, CA, February 1989.

[65] H. Korth and X. Peltier, "Query Algebra for Object-Oriented Databases," *Proceedings of the Schlumberger Software Conference,* 1990.

[66] G. Kuper and M. Vardi, "A New Approach to Database Logic," *Proceedings of ACM Conference on Principles of Database Systems,* 1984.

[67] C. Lamb, et.al.,"The ObjectStore Database System," *Communications of ACM,* Vol. 34, No. 10, 1991.

[68] Y.E. Lien, J.E. Shopiro and S. Tsur, "DSIS- A Database System with International Semantics," *Proceedings of the* 7^{th} *International Conference on Very Large Databases,* pp. 465-477, 1981.

[69] Y. Lou and Z.M. Ozsoyoglu, "LLO, An Object-Oriented Deductive Language with Methods and Method Inheritance," *Proceedings of ACM-SIGMOD International Conference on Management of Data,* June 1991.

[70] H. Lu, K. Mikkilineni and J.P. Richardson, "Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation," *Proceedings of the* 3^{rd} *IEEE International Conference on Data Engineering,* Los Angeles, CA, February 1987.

[71] D. Maier, *The Theory of Relational Databases,* Computer Science Press, 1983.

[72] D. Maier and J. Stein, "Development and Implementation of an Object-Oriented DBMS," *Research Directions in Object-Oriented Programming,* Shriver B. and P. Wegner Eds, MIT Press, Cambridge, MA, 1987.

[73] D. Maier, A. Otis and A. Purdy, "Object-Oriented Database Development at Servio Logic," *IEEE Transactions on Database Engineering,* Vol.8, No.4, December 1985.

[74] F. Manola and U. Dayal, "PDM: an Object-Oriented Data Model," *Proceedings of the International Workshop on Object-Oriented Databases,* Pacific Grove, CA, pp. 18-25, 1986.

[75] R. Milner, "A Proposal for Standard ML," *Proceedings of the ACM Symposium on LISP and Functional Programming,* Austin, TX, pp.184-197, August 1984.

[76] E. Neuhold and M. Stonebraker, "Future Directions in DBMS Research," Technical Report 88-001, Intl. Computer Science Inst. Berkeley California, May 1988.

[77] J. Orenstein, et.al, "Query Processing in the ObjectStore Database System," *Proceedings of the ACM-SIGMOD International Conference on management of Data,* San Diego, CA, June 1992.

[78] S.L. Osborn, "Identity Equality and Query Optimization," *Proceedings of the* 2^{nd} *International Workshop on Object-Oriented Database Systems,* Ebernburg, pp. 346-351, September 1988.

[79] B. Paseman, "Object-Oriented Database Panel: Position Statement," *Proceedings of the* 5^{th} *IEEE International Conference on Data Engineering,* pp.419-421, 1989.

[80] D.J. Penney and J. Stein, "Class Modification in the GemStone Object-Oriented Database Management Systems," *Proceedings of the* 2^{nd} *ACM International Conference on Object-Oriented Programming Systems, Languages and Applications,* Orlando, FL, October 1987.

[81] M.A. Roth, H.F. Korth and A. Silberschatz, "Extending Algebra and Calculus for Nested Relational Databases", ACM Transactions on Database Systems, Vol.13, No.4, pp.389-417, December 1988.

[82] L.A. Rowe and M.R. Stonebraker, "The Postgres Data Model," *Proceedings of the* 13^{th} *International Conference on Very Large Databases,* Brighton, pp. 83-96, 1987.

[83] E.A. Rundensteiner and L. Bic, "Set Operations in Object-Based Data Models," *IEEE Transactions on Knowledge and Data Engineering,* Vo. 4, No. 3, pp.382-398, 1992.

[84] M.H. Scholl and H.J. Scheck, "A Relational Object Model," *Proceedings of the* 3^{rd} *International Conference on Database Theory,* Paris, pp. 89-105, December 1990.

[85] G. Shaw and S. Zdonik, "An Object-Oriented Query Algebra," *Quart. Bull. of the IEEE TC on Data Engineering,* Vol.12, No.3, pp.29-36, September 1989.

[86] G. Shaw and S. Zdonik, "A Query Algebra for Object-Oriented Databases," *Proceedings of the* 6^{th} *IEEE International Conference on Data Engineering,* Los Angeles, CA, pp. 154-162, 1990.

[87] C. Shum and R. Muntz, "Implicit Representation for Extensional Answers," *Proceedings of the* 2^{nd} *International Conference on Expert Database Systems,* Washington, pp.257-273, 1988.

[88] D. Shipman, "The Functional Data Model and the Data Language Daplex," *ACM Transactions on Database Systems,* Vol. 6, No. 1, March 1981.

[89] H.T. Siegelmann and B.R. Badrinath, "Integrating Implicit Answers with Object-Oriented Queries," *Proceedings of the* 17^{th} *International Conference on Very Large Databases,* Barcelona, pp.15-24, September 1991.

[90] A.H. Skarra and S.B. Zdonik, "The Management of Changing Types in an Object-Oriented Database," *Proceedings of ACM International Conference on Object-Oriented Programming Systems, Languages and Applications,* Portland, OR, pp.483-495, September 1986.

[91] A.H. Skarra and S.B. Zdonik, "Type Evolution in an Object-Oriented Database," *Research Directions in Object-Oriented Programming, ed. B. Shiver, and P. Wenger,* MIT Press Series in Computer Systems, 1987, pp.393-415.

[92] A.H. Skarra, et al., "An Object Server for an Object-Oriented Database," *Proceeding of ACM/IEEE International Workshop on Object-Oriented Database Systems,* 1986.

[93] M. Stefik, and D.G. Bobrow, "Object-Oriented Programming: Themes and Variations," *AI Magazine,* pp. 40-62, January 1986.

[94] M. Stonebraker, et. al., "The Design and Implementation of INGRES," *ACM Transactions on Database Systems,* Vol. 1, No. 3, pp.189-222, 1976.

[95] M. Stonebraker, et.al., "Third Generation on Database System Manifesto," *Proceedings of IFIP DS-4 Workshop on Object-Oriented Databases,* 1990.

[96] M. Stonebraker and G. Kemnitz, "The Postgres Next-Generation Database Management System," *Communications of ACM,* Vol. 34, No. 10, 1991.

[97] D.D. Straube, "Queries and Query Processing in Object-Oriented Database Systems," Ph.D. Thesis, Department of Computing Science, University of Alberta, Spring 1991.

[98] D.D. Straube and M.T. Özsu, "Queries and Query Processing in Object-Oriented Database Systems," *ACM Transactions on Information Systems,* Vol. 8, No. 4, pp. 387-430, 1990.

[99] J. Ullman, *Principles of Database and Knowledge-Base Systems,* 2 vols., Computer Science Press, Rockville, Maryland, 1989.

[100] B. Vance, "Towards an Object-Oriented Query Algebra," *Oregon Graduate Institute,* Tech.Rep.No. CS/E 91-008.

[101] S.L. Vandenberg and D.J. DeWitt, "Algebraic Support for Complex Objects with Arrays, Identity and Inheritance," *Proceedings of ACM-SIGMOD Conference on Management of Data,* June 1991.

[102] F. Yazar, et.al " An Object-Oriented Query Language for an Object-Oriented Database Management System," *Proceedings of the DECSYM,* Side, March 1992.

[103] F. Yazar, et.al "The Development of an Object-Oriented Database Management System," *Proceedings of the* 7^{th} *International Symposium on Computers and Information Sciences,* Kemer-Antalya, November 1992.

[104] C. Zaniolo, "The Database Language GEM," *Proceedings of ACM-SIGMOD International Conference on Management of Data,* San Jose, CA, pp. 207-218, May 1983.

[105] S.B. Zdonik, "Data Abstraction and Query Optimization," *Proceedings of the* 2^{nd} *Workshop on Object-Oriented Database Systems,* Ebernburg, pp. 368-373, September 1988.

[106] S.B. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems,* San Mateo, CA: Morgan-Kauffman, 1989.

[107] R. Zicari, "A Framework for Schema Updates in an Object-Oriented Database System," *Proceedings of the* 7^{th} *IEEE International Conference on Data Engineering,* Kobe, April 1991.