# A Comprehensive Approach to Horizontal Class Fragmentation in a Distributed Object Based System*

C.I. Ezeife and Ken Barker

Advanced Database Systems Laboratory
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2

Tel.: (204) 474-8832

FAX: (204) 269-9178

{christie,barker}@cs.umanitoba.ca

**Abstract**

Optimal application performance on a Distributed Object Based System (DOBS) requires class fragmentation and the development of allocation schemes to place fragments at distributed sites so data transfer is minimized. Fragmentation enhances application performance by reducing the amount of irrelevant data accessed and the amount of data transferred unnecessarily between distributed sites. Algorithms for effecting horizontal and vertical fragmentation of *relations* exist, but fragmentation techniques for class objects in a distributed object based system are yet to appear in the literature. This paper first reviews a taxonomy of the fragmentation problem in a distributed object base. The paper then contributes by presenting a comprehensive set of algorithms for horizontally fragmenting the four realizable class models on the taxonomy. The fundamental approach is top–down, where the entity of fragmentation is the class object. Our approach consists of first generating primary horizontal fragments of a class based on only applications accessing this class, and secondly generating derived horizontal fragments of the class arising from primary fragments of its subclasses, its complex attributes (contained classes), and/or its complex methods classes. Finally, we combine the sets of primary and derived fragments of each class to produce the best possible fragmentation scheme. Thus, these algorithms account for inheritance and class composition hierarchies as well as method nesting among objects, and are shown to be polynomial time.

*This research was partially supported by the Natural Science and Engineering Research Council (NSERC) of Canada under an operating grant (OGP-0105566) and a grant from Manitoba Hydro.

# List of Figures

1 The Object Design Taxonomy . . . . . 5
2 The Linkgraph Generator . . . . . 8
3 Simple Predicate Generator . . . . . 9
4 Horizontal and Derived Fragments Integrator . . . . . 10
5 Class Horizontal Fragments Generator . . . . . 11
6 Horizontal Fragmentation – Complex Attribute and Simple Method . . . . . 14
7 Capturing the Complex Method dependency information Class Fragments . . . . . 17
8 Horizontal Fragmentation For Simple attributes and Complex methods . . . . . 18
9 Horizontal Fragmentation For Complex Attribute and Complex Methods . . . . . 19
10 The Sample Object Database Schema . . . . . 20
11 Complex Class Lattice . . . . . 21
12 Class Composition Hierarchy . . . . . 21
13 Complex Class Lattice's Link Graph . . . . . 24
14 Class Composition Hierarchy Linkgraph . . . . . 24
15 Generation of Derived Fragments of Classes based on Complex Methods . . . . . 26
16 The Structure Chart For Model with Complex Attributes and Complex Methods . . . . . 27

# 1 Introduction

Many researchers have demonstrated the importance of entity fragmentation in distributed relational database design. Optimal application performance on a Distributed Object Based System (DOBS) requires class fragmentation and the development of allocation schemes to place fragments at distributed sites so data transfer is minimized [9]. A DOBS supports an object oriented data model including features of *encapsulation* and *inheritance*.

The problem of distributed database design comprises first, the fragmentation of database entities and secondly, the allocation of these fragments to distributed sites. Two approaches are possible in a distributed database design – top-down and bottom-up. With the top-down approach, the input to the design process is the global conceptual schema (GCS) and the access pattern information, while the output from the design process is a set of local conceptual schemas (LCSs) [17]. The input to the design process is obtained from an *a priori* static and system requirements analysis which defines the system environment and collects an approximation of both the data and processing needs of all potential database users [20]. The expectations of the system with respect to performance, reliability and availability, economics and flexibility are also specified from the analysis. The view and conceptual design activities produced from the requirements study defines interfaces for end users, entity types and their relationships. Statistical information collected from these design activities include the frequency of user applications and reference patterns of applications. The output of this static and system analysis are the GCS and the access pattern information which constitute inputs to the distribution design process described here.

This paper uses this top-down design approach and designs the LCSs by distributing the entities over distributed sites. In the relational database environment, the entity of distribution is a relation while in a distributed object based system (DOBS), our entity of distribution is a class. This work aims at dividing classes into class fragments, which are later distributed.

There are many existing object based systems that support some form of distribution including ITASCA [8], ENCORE [7], GOBLIN [10], THOR [12], and EOS [16]. Some of these systems have efficient techniques for grouping and clustering objects of a class with objects of their most needed parent on the same disk unit (container) [16]. These techniques, however, depart from obtaining partitions of, and the distribution, of each database entity (like a class). Secondly, though they provide techniques for handling object migration, minimizing replication and migration of objects upfront is not emphasized. Moreover, with these approaches, logical design and organization of data is not clearly independent of the physical organization of data. The work of Bertino and Kim [1] on index selection is also complementary to our work, but different in the sense that they aim at efficient design at the physical level (local internal schema), while our design aims at efficient design at the logical level (local conceptual schema). Distributed relational databases [17] benefit greatly from fragmentation and these benefits should be realized in a distributed object environment [9]. A partial list of benefits include:

- Different applications access or update only portions of classes so fragmentation will reduce the amount of irrelevant data accessed by applications.
- Fragmentation allows greater concurrency because the "lock granularity" can accurately reflect the applications using the object base. Fragmentation allows parallel execution of a single query by dividing it into a set of subqueries that operate on fragments of a class.
- Fragmentation reduces the amount of data transferred when migration is required.
- Fragment replication is more efficient than replicating the entire class because it reduces the update problem and saves storage.

Although algorithms currently exist for fragmenting relations, fragmentation and allocation of objects is still a relatively untouched field of study. This work contributes and is unique in the following ways: each object-oriented database entity (database class) is fragmented and can be distributed, and partitioning of each entity aims at minimizing the need for object replication, migration and redundancy.

The overhead and difficulty involved in implementing distributed design techniques include the generation of inputs from static analysis. Earlier work has argued that since 20% of user queries account for 80% of the total data accesses, this analysis is feasible [19]. Secondly, these distribution techniques work best for domains without frequent drastic changes in requirements. Accommodating major changes in a domain would entail a re-analysis of the system and re-running of the distributed design algorithms. Future research will investigate how these techniques can be incorporated into a dynamic system.

The balance of the paper is organized as follows. We complete this section by briefly reviewing previous work on distributed database design. In Section 2 our DOBS model and a taxonomy for various class models is presented. This paper contributes by presenting in Section 3, the fragmentation algorithms for an object model with simple attributes and simple methods outline of which are also in [5, 6]. It further extends this earlier work to the other models, including: a model where attributes are composed in a part-of hierarchy and accessed using simple methods (Section 4); a model consisting of simple attributes and complex methods (Section 5); and a model consisting of complex attributes and complex methods (Section 6). Section 7 discusses other important features of the fragmentation schemes including their time complexities. Finally, Section 8 concludes and suggests future research directions.

## 1.1 Related Work

Three fragment types are defined on a database entity. Horizontal Fragmentation is the breaking up of a relation/class into a set of horizontal fragments with only subsets of its tuples/instance objects. Vertical Fragmentation is the breaking up of a class into a set of vertical fragments with only subsets of its attributes and methods. Hybrid Fragmentation is the breaking up of a class into a set of hybrid fragments with both subsets of its tuples/instance objects as well as subsets of their attributes and methods.

This section reviews previous work on horizontal class fragmentation in distributed database systems. Previous work on relational horizontal fragmentation is reviewed followed by work on fragmentation in DOBS.

**Horizontal Fragmentation (relational):** Several researchers have worked on fragmentation and allocation in the relational data model including Ceri, Negri and Pelagatti [2], Özsu and Valduriez [17], Navathe *et al.* [13] , Navathe and Ra [15], and Shin and Irani [3].

Ceri, Negri and Pelagatti [2] show that the main optimization parameter needed for horizontal fragmentation is the number of accesses performed by the application programs to different portions of data (file of records). They define applications in terms of boolean *predicates* and use access pattern information to achieve the design. Predicates are collected into sets of *minterms* which form the horizontal fragments. Navathe, Karlapalem and Ra [14] define a scheme for simultaneously applying the horizontal and vertical fragmentation algorithms on a relation to produce a grid. A technique similar to the vertical fragmentation schemes discussed in Navathe *et al.* [13, 15] is used to produce horizontal fragments. Özsu and Valduriez [17] define the database information needed for horizontal fragmentation of the universal relation and show how the database relations are reconstructible using joins. Ceri *et al.* [4] model this relationship explicitly using directed links drawn between relations via equijoin operations. The relation at the tail of a link is called the *owner* of the link and the relation at the head is the *member* [4]. Primary horizontal fragmentation is performed on all owner relations, while derived horizontal fragmentation is performed on all member relations of links. Shin and Irani [18] partition relations horizontally based on estimated user reference clusters (URCs). URCs are estimated from user queries [2, 17] but are refined using semantic knowledge of the relations.

**Horizontal Fragmentation (objects):** Karlapalem, Navathe and Morsi [9] identify some of the fragmentation issues in object bases including: How are subclasses of a fragment of a class handled? Which objects and attributes of the objects are being accessed by the methods? What type of methods are considered: simple methods that access a set of attribute values of an object or complex methods that access a set of objects and instance variables?[1] Further, they argue that a precise definition of the processing semantics of the applications is necessary. They do not present solutions for horizontally fragmenting class objects but argue that techniques used by Navathe *et al.* [14] for fragmenting relations could be applied.

## 2 The DOBS model

A distributed object based system is a collection of local object bases distributed at different sites, interconnected by a network. We assume the database management system (DBMS) is distributed and the objects making up the object base are placed around the network. The general architecture of a

[1] They take complex methods as being synonymous with an application.

distributed DBMS is available in Özsu and Valduriez [17].

## 2.1 The Data Model

The data in a DOBS consists of a set of encapsulated objects. The data values (attributes) are bundled with the methods (procedures) for manipulating them to form an *encapsulated* object. Objects with common attributes and methods belong to the same class, and every class has a unique identifier. Inheritance allows reuse and incremental redefinition of new classes in terms of existing ones. Parent classes are called *superclasses* while classes that inherit attributes and methods from them are called *subclasses*. The database contains a root class called *Root* which is an ancestor of every other class in the database. The overall inheritance hierarchy of the database is captured in a class lattice. A class is an ordered relation $\mathcal{C} = (\mathbf{K},\mathcal{A},\mathcal{M},\mathcal{I})$ where $\mathbf{K}$ is the class identifier, $\mathcal{A}$ the set of attributes, $\mathcal{M}$ the set of methods and $\mathcal{I}$ is the set of objects defined using $\mathcal{A}$ and $\mathcal{M}$[2]. Each horizontal fragment ($\mathcal{C}_h$) of a class contains all attributes and methods of the class but only some instance objects ($\mathcal{I}' \subseteq \mathcal{I}$) of the class. Thus, $\mathcal{C}_h = (\mathrm{K},\mathcal{A},\mathcal{M},\mathcal{I}')$. Each vertical fragment ($\mathcal{C}^v$) of a class contains its class identifier, and all of its instance objects for only some of its methods ($\mathcal{M}' \subseteq \mathcal{M}$) and some of its attributes ($\mathcal{A}' \subseteq \mathcal{A}$). Thus, $\mathcal{C}^v = (\mathrm{K},\mathcal{A}',\mathcal{M}',\mathcal{I})$. Each hybrid fragment ($\mathcal{C}_h^v$) of a class contains its class identifier, some of its instance objects ($\mathcal{I}' \subseteq \mathcal{I}$) for only some of its methods ($\mathcal{M}' \subseteq \mathcal{M}$), and some of its attributes ($\mathcal{A}' \subseteq \mathcal{A}$). Thus, $\mathcal{C}_h^v = (\mathrm{K},\mathcal{A}',\mathcal{M}',\mathcal{I}')$.

Two types of attributes in a class are possible (simple and complex). Simple attributes have only primitive attribute types that do not contain other classes as part of them. Complex attributes have the domain of an attribute as another class. The complex attribute relationship between a class and other classes in the database is usually defined using a class composition or aggregation hierarchy. Two possible method structures in a distributed object based system are simple and complex methods. Simple methods are those that do not invoke other methods of other classes. Complex methods are those that can invoke methods of other classes. The classes making up the DOBS are classified based on the nature of the attributes and methods they contain. Although two basic method types exist, a simple method of a contained (part-of) class is referred to as a contained simple method because it is a simple method of a class that is contained in another class. Thus, the variety of class models that could be defined in a DOBS are: Class models consisting of simple attributes and simple methods, Class models consisting of complex attributes and contained simple methods, Class models consisting of simple attributes and complex methods, and Class models consisting of complex attributes and complex methods. This classification (Figure 1) enables us accommodate all the necessary features of object orientation and provide solutions for object bases that are structured in various ways. The taxonomy (Figure 1) has three dimensions. The axes are the *type of fragmentation*, the *type of attributes*, and the *type of method invocations* in the object model. Distributed object based design enhances performance by

[2]We adopt the notation of using calligraphic letters to represent sets and roman fonts for non-set values.

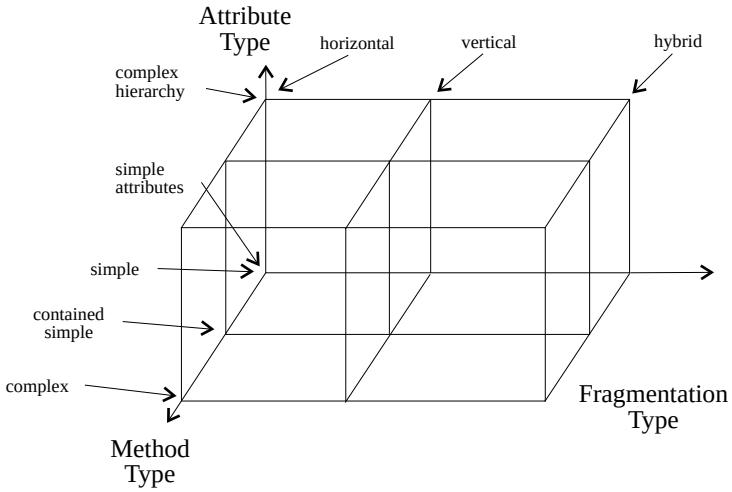


Figure 1: The Object Design Taxonomy

organizing database entities in fragments such that the amount of irrelevant data accessed by applications is reduced while reducing the amount of data that needs to be transferred between sites.

# 3 Simple Attributes and Methods

This section presents horizontal fragmentation algorithms for classes consisting of objects that have simple attributes using simple methods as discussed in [5, 6]). We proceed by explicitly stating assumptions, definitions required, and then summarize the algorithm. The explicit assumptions are given next. Firstly, objects of a subclass physically contain only pointers to objects of its superclasses that are logically part of them. In other words, an object of a class is made from the aggregation of all those objects of its superclasses that are logically part of this object. Secondly, there are no cycles in the dependency graphs discussed later. The first assumption makes for a more efficient storage structure because inherited attributes and methods are not replicated at subclasses. The structure is more realistic and reflects the actual object and specialization and many object oriented database systems use this storage structure, e.g., ENCORE [7]. The alternative storage structure also has its advantages and is fragmented as well by our approach without modification. The significance of this assumption lies in the fact that providing a fragmentation scheme for the alternative storage structure would call for an extensive modification of the fragmentation algorithm in order to accommodate the storage structure in the first assumption. Some applications may require an object structure where provision of a cycle is necessary in the aggregation or class composition graph (e.g., when two classes are part of each other). Our approach is to define a uni-directional dependency graph from the aggregation hierarchy, and in this case, we select which of the two bi-directional links in the aggregation graph has higher application access frequency. The effect of the uni-directional dependency graph is providing a deterministic direction for

propagation of primary fragments to derived fragments between two classes that depend on each other.

*Primary Horizontal Fragmentation* is the partitioning of a class based on only applications accessing the class. For example, an application running on a class **Student**, with an attribute **dept**, could have predicates {$P_1$:dept="Math",$P_2$:dept="Computer Sc",$P_3$:dept="Stats"}. *Derived horizontal fragmentation* occurs in three ways. (1) Partitioning of a class arising from the fragmentation of its subclasses, (2) fragmentation of a class arising from the fragmentation of some complex attribute (part–of) of the class, and (3) fragmentation of a class arising from the fragmentation of some complex method classes invoking methods of this class. *The cardinality of a class* is the number of instance objects in the class (denoted **card**($C_i$)). Several more definitions are required:

**Definition 3.1** *A user query* accessing database objects is a sequence of method invocations on an object or set of objects of classes. The invocation of method $j$ on class $i$ is denoted by $M_j^i$ and a user query $Q_k$ is represented by $\{M_{j1}^{i1}, M_{j2}^{i2}, \ldots, M_{jn}^{in}\}$ where each $M$ in a user query refers to an invocation of a method of a class object. ■

**Definition 3.2** *Access frequency of a query* is the number of accesses a user application makes to data. Data in this context can be a class, a fragment of a class, an instance object of a class, an attribute or method of a class. If Q = $\{q_1, q_2, \ldots, q_q\}$ is a set of user queries, acc($q_i$,$d_j$) indicates the access frequency of query $q_i$ on data item $d_j$. We represent the access frequency of a minterm $m_i$ as acc($m_i$,$d_j$) since minterms formalize queries. ■

**Definition 3.3** *Object Pointer join (↑)* between a fragment $F_o^p$ of an owner class $C_o$ and a member class $C_m$ returns the set of objects in the member class which are pointed to by objects in the fragment of the owner class $F_o^p$. ■

**Definition 3.4** *Instance Object join (⊙)* between a pointer to an instance object of a superclass and an instance object ($I_j$) of a class $C_i$ returns the aggregate of instance objects of the two classes that represent the actual instance object of the class. ■

To illustrate the aggregate returned by the object join function, we consider the following example. In a database with *Student* a superclass of *Grad*, an instance object $I_3$ of *Grad* is represented as (Student pointer5) ⊙ {Grad3,Mary Smith}. This means that the actual $I_3$ of *Grad* is the quantity representing the instance object $I_5$ of the superclass *Student* added to the quantity {Grad3,Mary Smith}. *A method invocation* $M_j^i$ on the objects of a class $C_i$ is represented as a set of simple predicates that describe which objects to access. These simple predicates are represented as $\{Pr_1^i, Pr_2^i, \ldots, Pr_n^i\}$ for class $C_i$. *Minterm selectivity* is the number of instance objects of the class accessed by a user query specified according to a given minterm predicate.

The proposed algorithms are guided by the intuition that an optimal fragmentation keeps those instance objects accessed frequently together while preserving the inheritance and class composition hierarchies. Secondly, the fragments defined are guaranteed correct by ensuring they satisfy the following

correctness rules of completeness, disjointness and reconstructibility. Completeness requires that every attribute or method belongs to a class fragment, while disjointness means every attribute or method belongs to only one class fragment. Finally, reconstructibility requires that the union of all class fragments should reproduce the original class.

## 3.1 The Algorithm (HorizontalFrag)

We assume that the database and application information required for distributed design are pre–defined and application information definitions are given as required in the following narrative. Input to the fragmentation process consists of the set of applications or user queries, the set of database classes, and the inheritance hierarchy information. The output expected from this fragmentation process is a set of horizontal fragments for all classes in the database. The four-step algorithm required for horizontally fragmenting this first class model consisting of simple attributes and simple methods is called *HorizontalFrag*. Discussions of the four steps follow.

**Step One – Define the Link Graph For all Classes in the Database**

The object base information required is the global conceptual schema. In the relational model, this shows how database relations are implicitly connected to one another through joins. Generally, the object base information needed by the fragmentation procedure are of two types: **The class lattice** showing the superclass–subclass relationship between all classes, and **the dependency graph**[3] which captures the method link or attribute link between any two classes in the database. This means that, if an attribute of a class $C_i$ is another class $C_j$, then there is an attribute link between classes $C_i$ and $C_j$ and is represented by an arc in the dependency graph ($C_j \rightarrow C_i$). We capture both the class lattice information and dependence information of the database schema using a link graph. Since only simple attributes and simple methods are used in this first model, the link graph is restricted to class lattice information and is generated as follows. Every subclass $C_j$ depends on its superclass $C_i$ in the link graph and has an arc ($C_j \rightarrow C_i$) inserted in the link graph. Starting from the leaf classes of the class lattice, we define a link from each class to its superclasses. This is a direct consequence of the inheritance hierarchy which implicitly partitions every superclass into subclasses. By starting fragmentation with the subclass and propagating intermediate results up the class hierarchy we preserve the implicit partitioning of the inheritance hierarchy and produce results that approach our intuitive notion of optimality. Directed links among database classes are used to show these relationships and if class $C_i$ depends on class $C_j$, an arc is inserted ($C_j \rightarrow C_i$). Algorithm *Linkgraph* of Figure 2 gives a formal presentation of this step. This algorithm starts from the leaf classes of the class lattice and defines a link from each class to its superclasses (lines 3-10).

**Step Two - Define Primary Horizontal Fragments of Classes**

The quantitative database information required is the cardinality of each class. Both qualitative and

[3] An example of a graph depicting dependency information is the class composition hierarchy shown in Kim [11].

**Algorithm 3.1** *(Linkgraph - captures the inheritance hierarchy)*

**Algorithm Linkgraph**
**input:** $\mathcal{C}_d$**:** set of classes in the database; $\mathcal{C}_l$**:** set of leaf classes and $\mathcal{C}_l \subseteq \mathcal{C}_d$.
LT: Class lattice of the database showing subclass/superclass relationships.
**output:** $LG$: The Link graph for the database schema.
LG = ($\Gamma$,$\lambda$) where $\Gamma$ is a set of nodes; $\lambda$ is a set of arcs connecting nodes in $\Gamma$.
**begin**
$LG \leftarrow$ initialized with a node $\forall C_i \in \mathcal{C}_d$; (1)
s.t. $\Gamma \leftarrow \{C_k | C_k \in \mathcal{C}_d\}$ and $\lambda = \emptyset$;
$C_i \leftarrow$ leaf class in $\mathcal{C}_l$
$C_i = \{C_k | C_k \in \mathcal{C}_l\}$ (2)
**while** $C_i \neq$ Root **do** (3)
**for each** $C_i \in \mathcal{C}_d$ (4)
**for each** $C_j \in superclass(C_i)$ (5)
$\lambda = \lambda \cup (C_i \rightarrow C_j)$ (6)
$C_l = C_l \cup C_j$ (7)
**end**; {for $C_j$}
**end**; {for $C_i$}
$C_l = C_l - C_i$; (8)
$C_i =$ a leaf node in $\mathcal{C}_d$ (9)
**end**; {while $C_i$}
**return** ($LG$); (10)
**end**;

Figure 2: The Linkgraph Generator

quantitative application information is required. The fundamental qualitative information consists of the predicates used in external methods (messages or procedure calls) for user queries. A primary horizontal fragmentation is defined by the effects of user queries on objects of the owner classes. A primary horizontal fragment of a class is a set of objects of this class accessed together by only applications running on this class. In the object oriented case, owner classes constitute all classes in the object base whose objects are accessed by user queries. Recall that each user query is a sequence of method invocations on objects of a class and each method of a class is represented as a set of predicates defined on values of attributes of that class. Therefore, to get the set of simple predicates on any class $C_i$, we form the union of all predicates of all methods from all the user queries, operating on attributes of the class $C_i$ that are defined over some domain $D_i$.

A simple predicate is defined as follows. Given a class $C_i$ with the attributes ($A_1$,$A_2$,...,$A_n$) where $A_i$ is an attribute defined over domain $D_i$, a simple predicate $p_j$ defined on the class $C_i$ has the form: $p_j$: $A_i$ $\theta$ *value* where $\theta \in \{=, <, \neq, \leq, >, \geq\}$ and *value* is chosen from $D_i$. We use $\mathcal{P}_i$ to denote the set of all simple predicates defined on a class $C_i$. The members of $\mathcal{P}_i$ are denoted by $P_{ij}$ in the object base. Following Özsu and Valduriez [17], given a class with the set $\mathcal{P}_i = \{P_{i1}, P_{i2}, \ldots, P_{im}\}$ of simple predicates we generated for the class from applications, we generate the set of minterm predicates $MM_i = \{m_{i1}, m_{i2}, \ldots, m_{iz}\}$. $MM_i$ is formed using $\mathcal{P}_i$ and $MM_i = \{m_{ij} \mid m_{ij} = \bigwedge_{p_{ik} \in p_i} P^*_{ik}\}$, $1 \leq k \leq$

**Algorithm 3.2** *(SimplePredicates - Generate simple predicates from user queries)*

**Algorithm SimplePredicates**
**input:** $\mathcal{Q}$: set of all user queries (ie a set of sequences of methods)
$\mathcal{C}_d$: set of all database classes
**output: $\mathcal{P}$: set of set of simple predicates for all classes in the database.**
where the set of simple predicates for class $C_i$ is $\mathcal{P}_i$.
**begin**
**for each** class $C_i \in C_d$ **do** (1)
$\mathcal{P}_i = \emptyset$ (2)
**for each** $q_k \in \mathcal{Q}$ **do** (3)
**for each** $M_i^j \in q_k$ **do** (4)
**for each** $P_{i1}^{j1} \in M_i^j$ (5)
$\mathcal{P}_i = \mathcal{P}_i \cup P_{i1}^{j1}$ (6)
**end**; {for $M_i^j$}
**end**; {for $q_k$}
$\mathcal{P} = \bigcup \mathcal{P}_i$ (7)
**end;**

Figure 3: Simple Predicate Generator

$n, 1 \leq j \leq z$, where $P_{ik}^* = P_{ik}$ or $P_{ik}^* = \neg P_{ik}$.[4] We now use the semantics of the class to eliminate meaningless minterm predicates. Algorithms *SimplePredicates* of Figure 3 and *COM-MIN* of [17] are used in this step. COM-MIN is an algorithm that generates a complete and minimal set of predicates from a set of simple predicates. The Algorithm *SimplePredicates* of Figure 3 converts all user queries into a set of simple predicates for all classes in the database (lines 3-7).

**Step Three – Derived Horizontal Fragmentation on Member Classes**

A derived horizontal fragmentation is defined on the member classes of links according to its owner. We partition a member class according to the fragmentation of its owner class and define the resulting fragment on the attributes and methods of the member class only. Therefore, given a link L where owner(L) = S and member(L) = R, the derived horizontal fragments of R are defined as: $R_i = R \uparrow S_i$, where $1 \leq i \leq w$ and $w$ is the number of fragments defined on R based on the owner. The link between owner and member classes is a pointer reference ($\uparrow$) which generates those instance objects of a member class that are pointed to by instance objects in an owner fragment. For example, an instance object of a subclass points to an instance object of its superclass which is logically part of this subclass instance object. A derived fragment of a member class based on a primary fragment of an owner class is a set of member class objects accessed together by applications running on the owner class. The derived horizontal fragments of member classes of links are generated by lines 5 through 7 of *HorizontalFrag* of Figure 5. On Line 7, $F_j^{d_k}$ is the kth derived fragment of class $C_j$ and $F_o^{p_k}$ is the kth primary fragment of class $C_o$. The symbol $\uparrow$ denotes the object pointer join and $F_j^{d_k}$ is the set of objects in the member class that are pointed to by objects in a fragment of the owner class.

[4] In other words, each simple predicate can occur in minterm predicates either in its natural form or negated form.

**Algorithm 3.3** *(HorizontalMember - Primary and Derived Fragments Integrator)*

**Algorithm HorizontalMember**
**input**: $\mathcal{F}_i^p$: set of primary horizontal fragments of class $C_i$
$\mathcal{F}_i^d$: set of derived horizontal fragments of class $C_i$
**output**: $\mathcal{F}_i$: set of horizontal fragments of class $C_i$
**begin**
**for each** $F_i^d \in \mathcal{F}_i^d$ (1)
select $F_i^p$ according to Affinity Rule 3.1 (2)
$\mathcal{F}_i^p = \mathcal{F}_i^p \cup F_i^d$ (3)
Ensure disjointness
**for each** overlapping $I_i \in F^p$ **do** (4)
select $F_{im}^p$ according to Affinity Rule 3.2 (5)
**for each** $F_{in}^p, n \neq m$ **do** (6)
$F_{in}^p = F_{in}^p - I_i$; (7)
**end**; {for $F_i^p$}
**end**; {for $I_i$}
**end**; {for $F_i^d$}
**if card**$(\mathcal{F}_i^p) = 1$ **then** (8)
**begin**
**for each** $F_i^d \in \mathcal{F}_i^d$ **do** (9)
$F_i^p = F_i^d$ (10)
**end;** Ensure Completeness
$\mathcal{F}_{ik}^p = \mathcal{F}_i^p - \bigcup_{k \neq j} \mathcal{F}_{ij}^p$ (11)
**end**
$F_i = \mathcal{F}_i^p$ (12)
**end**;

Figure 4: Horizontal and Derived Fragments Integrator

**Step Four – Combining Primary and Derived Fragments**

The final horizontal fragments of a class is composed of objects accessed together by both applications running only on this class and those running on its owner classes. Therefore, we must determine the most appropriate primary fragment to merge with each derived fragment of every member class. Several simple heuristics could be used such as selecting the smallest or largest primary fragment or the primary fragment that overlaps the most with the derived fragment. Although these heuristics are simple and intuitive they do not capture any quantitative information about the distributed object base. Therefore, a more precise approach is described that captures the environment and attempts to use it in merging derived fragments with primary fragments. Algorithm *HorizontalMember* of Figure 4 gives a formal presentation of this step. *HorizontalMember* selects a primary fragment of a member class that is most appropriate to combine with a derived fragment of the same class. The primary fragment of choice is the one that has highest affinity with the current derived fragment by applying affinity Rule 3.1 (Lines 1-3). If the class has no primary fragments, the derived fragments are made the final horizontal fragments and instance objects not yet contained in a derived fragment are placed in one of the fragments (Lines 8-11).

Capturing the quantitative information requires a few additional definitions. First, we define the access

**Algorithm 3.4** *(HorizontalFrag – Horizontal Fragments Generator)*

**Algorithm HorizontalFrag**
**input**: $\mathcal{Q}_i$: set of user queries; $\mathcal{C}_d$ : set of database classes
**L(C)** : the class lattice
**output**: $\mathcal{F}_{c_i}$: set of horizontal fragments of classes in the database.
**var**
Linkgraph : tree
$\mathcal{P}_i$ : set of simple predicates for class $C_i$.
$\mathcal{F}_i^p$ : set of primary horizontal fragments for class $C_i$
$\mathcal{F}_i^d$ : set of derived horizontal fragments for class $C_i$
$\mathcal{M}_c^L$ : set of member classes for the link graph L(C)
**begin**
**linkgraph = LinkGraph($\mathcal{C}_i$,L(C));** (1)

$\mathcal{P}_i$ **= SimplePredicates($\mathcal{Q}_i$,$\mathcal{C}_i$)** (2)
**for each class** $C_i \in$ **owner(L(C))** (3)
$\mathcal{F}_i^p$ **= minterms of [COM-MIN($\mathcal{P}_i$)]** (4)

**for each** member class $C_j \in$ L(C) **do** (5)
**for every** primary fragment in an owner class of $C_j$, $C_o$ (6)
$F_j^{d_k} = C_j \uparrow F_o^{p_k}$ (7)
**end; {for k}**
**end; {for** $C_j$**}**

**for each** $C_j \in \mathcal{M}_c^L$ **do** (member class) (8)
$\mathcal{F}_{c_j}$ **= HorizontalMember($\mathcal{F}_j^p$,$\mathcal{F}_j^d$)** (9)
**for each** $C_i \in (\mathcal{C}_d - \mathcal{M}_c^L)$ **do** (nonmember class) (10)
$\mathcal{F}_{c_i} = \mathcal{F}_i^p$ (11)
**end;**

Figure 5: Class Horizontal Fragments Generator

frequencies of an object as:

**Definition 3.5** *Access frequency acco(I)* of an object ($I$) is the sum of the access frequencies of all the applications $\mathcal{Q}_i$ accessing the object. Thus, acco(I) = $\sum_i^q acc(q_i, I)$ for all q applications accessing the object. ■

This definition can be used to define the number of relevant and irrelevant accesses to an object with respect to another fragment.

**Definition 3.6** *Relevant accesses($F_i^d$,$F_i^p$)* are those made to local objects of both fragments $F_i^d$ and $F_i^p$ and are defined as all of the access frequencies of objects belonging to both the derived and primary fragments. Thus, Relevant accesses($F_i^d$,$F_i^p$) = $\sum_{I_i}$ acco($I_i$), for all $I_i \in (F_i^d \cap F_i^p)$. ■

**Definition 3.7** *Irrelevant access($F_i^d$,$F_i^p$)* are those made to instance objects which are in the primary fragment, $F_i^p$, or the derived fragment, $F_i^d$, but not in both. Irrelevant access($F_i^d$,$F_i^p$) = $\sum_{I_j}$ acco($I_j$), for all $I_j \in ((F_i^p \cup F_i^d)$ - $(F_i^p \cap F_i^d))$. ■

We now define the affinity between a derived fragment ($F_i^d$) and a candidate primary fragment ($F_i^p$).

**Definition 3.8** *Affinity between a derived Fragments $F_i^d$ and a primary fragment $F_i^p$ of the same class, aff($F_i^d$,$F_i^p$)* is a measure of how frequently objects of these two fragments are needed together by applications. Thus, aff($F_i^d$,$F_i^p$) = Relevant access($F_i^d$,$F_i^p$) − Irrelevant access($F_i^d$,$F_i^p$). ■

**Definition 3.9** *The Object Affinity* $aff o(I_i, I_j)$, the affinity between two objects $I_i$ and $I_j$ is the sum of each object's accesses (affinity) for all queries that access both objects. Thus, $aff o(I_i, I_j) = \sum_{\{q_k \| q_k \in \mathcal{Q} \wedge q_k \text{access} I_i \wedge I_j\}} (acc(q_k, I_i) + acc(q_k, I_j))$ ■

**Definition 3.10** *Affinity between an object $I_i$ and a fragment F, faff($I_i$,F)* is the sum of object affinities between this object $I_i$ and all objects of the fragment F. Thus, faff($I_i$,F) = $\sum_{I_j \in F}$ affo($I_i$,$I_j$), $i \neq j$.

■

We are now in a position to determine the primary horizontal fragment in the member class that is most suitable to merge with a particular derived fragment in the member. This is summarized in the following rule.

**Affinity Rule 3.1** Select the primary fragment that maximizes the affinity measure aff($F^d$, $F_i^p$) where $F^d$ is the derived fragment and $F_i^p$ is a primary fragment in the class ranging over all candidate fragments. This is the primary fragment that has the highest affinity with this derived fragment. ■

This rule selects the most suitable primary fragment to merge with the derived fragment. The merging process results in some overlap of objects in a set of primary fragments. Disjointness requires that an instance object appearing in more than one fragment is not permitted so a technique is required to

determine the object's best location. One approach would be to eliminate the instance from all primary fragments other than the one selected using Affinity Rule 3.1. This is likely to be suboptimal. Our algorithm uses an object's affinity to its fragments to determine the best final placement for the object. This is accomplished with the following rule.

**Affinity Rule 3.2** The primary fragment $F_j^p$ that maximizes the function $\text{aff}(I_i, F_j^p)$ is where $I_i$ is placed. ■

The final step of this process converts derived fragments to primary for a class that has no primary fragments due to application access pattern.

# 4 Complex Attributes and Simple Methods

This section presents an algorithm for horizontally fragmenting classes consisting of complex attributes that support a class composition hierarchy using simple method invocations. The algorithm first uses the attribute link information from the class composition hierarchy [11] to fragment classes according to the fragmentation of their contained classes. The class composition hierarchy is used to define the link graph showing the dependencies between any two classes. If class $C_i$ has class $C_j$ as part–of it there is a link from class $C_j$ to class $C_i$ in the link graph. The first iteration invokes *HorizontalFrag*, using the link graph from the class composition hierarchy, to produce an initial fragmentation. Generating derived fragments of a member class originating from class composition hierarchy entails finding the predicates of its owner class and defining fragments of the owner class based on these predicates. The second iteration of the algorithm executes *HorizontalFrag* with the link graph defined using the inheritance hierarchy to produce fragments based on applications that preserve inheritance hierarchy too. The first iteration involves:

1. Form a class link graph from the class composition hierarchy. If class $C_i$ is composed using class $C_j$ (i.e. $C_i$ contains $C_j$), form a link from class $C_j$ to $C_i$.
2. Define *primary horizontal fragments* on owner classes of links which compose the contained classes.
3. Define *derived horizontal fragments* on member classes (containing classes). These are fragments of the containing class that point to only some instance objects of the contained classes.
4. Form the *union* of primary and derived fragments for member classes (containing classes) ensuring all fragments remain disjoint.

This iteration produces the primary horizontal fragments of the contained classes in the aggregation graph, which are propagated to the containing classes to produce their initial horizontal fragments (line (1), Figure 6). The final horizontal fragments of the containing classes are produced using the class

**Algorithm 4.1** *(Complex Attribute and Simple Method)*

**Algorithm Hor_CA_SM**
**input**: $\mathcal{Q}_i$: set of user queries
$\mathcal{C}_d$ : set of database classes including classes with complex attribute and simple methods
**L(C)** : the class lattice
**A(C)** : class composition hierarchy showing attribute link
**output**: $\mathcal{F}_{c_i}$: set of horizontal fragments of the set of classes in the database
**begin**
$\mathcal{F}_{c_i}$ **= HorizontalFrag(**$\mathcal{Q}_i$**,** $\mathcal{C}_d$**, A(C))** (1)
$\mathcal{F}_{c_i}$ **= HorizontalFrag(**$\mathcal{Q}_i$**,** $\mathcal{F}_{c_i}$**, L(C))** (2)
**end; {Hor_CA_SM}**

Figure 6: Horizontal Fragmentation – Complex Attribute and Simple Method

inheritance lattice to obtain a "new" link graph. The *HorizontalFrag* algorithm uses this link graph and the fragments of containing class from the first iteration as its primary fragments (line (2), Figure 6). It is possible that the second application of the algorithm results in the original class. This means the application requirements make fragmentation unnecessary.

# 5 Simple Attributes and Complex Methods

Horizontal fragmentation of classes consisting of objects with simple attributes using complex methods requires that we know *à priori* those objects accessed by a method invocation and that encapsulation is not violated. The former requirement is accomplished with static analysis and the latter is inherent in the object model. Achieving optimal fragmentation requires that objects that invoke methods on a set of other related objects should be contained within the same fragment. Obviously, if this can be achieved it will be possible to allocate highly related objects together. This goal is reflected in the following definitions because we can develop an algorithm that maximizes local relevant access and minimizes local irrelevant access. Thus, we present definitions that enable us define groups of objects of each class being fragmented that are needed together based on nested method invocations from other classes in the database.

**Definition 5.1** *Object reference set of method* $M_j^i$ denoted oref($M_j^i$) contains all objects referenced by method $M_j^i$ of some object in class $C_i$. ■

We can enumerate the objects in oref($M_j^i$) by $\{I_1^k, I_2^l, \ldots, I_n^m\}$ where $I_n^m$ is the $n^{th}$ object needed by method $M_j^i$ and is also an object of class $C_m$.

**Definition 5.2** *Class fragment object reference set coref($F_i^h$,$C_m$)* are objects of class $C_m$ referenced by methods of class fragment $F_i^h$ of class $C_i$. This is derived from oref($M_j^i$) as :
coref($F_i^h$,$C_m$) = $\bigcup_{I_m \in oref(M_j^i)} I_m$ for all $M_j^i \in F_i^h$. ■

If fragments generated with coref statistics contain overlapping objects, we use method object affinity value between each overlapping object and each of the fragments to decide which one fragment it is most beneficial to keep this attribute in, taking into consideration all application and nested method needs. This measure is also important in calculating the global affinity between two fragments (say a derived and primary fragment) and used during merging of such fragments. Thus, the method relevant and irrelevant access statistics in this model have been modified to include nested method accesses to these objects.

**Definition 5.3** *Number of references* of method $M_j^i$ to an object $I_m$ of class $C_m$ is denoted numref($M_j^i$,$I_m$) which is the number of times method $M_i^j$ accesses object $I_m$ during one invocation of $M_j^i$. ■

**Definition 5.4** *Class number of reference cnumref($F_i^h$,$I_m$)* of class fragment $F_i^h$ on object $I_m$ of class $C_m$, is the sum of all accesses made to this object by all methods of this class fragment. Thus, cnumref($F_i^h$,$I_m$) = $\sum_{M_j^i \in F_i^h}$ numref($M_j^i$,$I_m$). ■

**Definition 5.5** *The Method Object Affinity maffo($I_i$,$I_j$)* is the sum of accesses made by any methods of a class fragment or applications to both objects $I_i$ and $I_j$. maffo($I_i$,$I_j$)= (application accesses to both objects) + (method accesses to both objects)

maffo($I_i$,$I_j$) = $\sum_{\{q_k \| q_k \in \mathcal{Q} \wedge q_k \text{access} I_i \wedge I_j\}} (acc(q_k, I_i) + acc(q_k, I_j))$ +

$\sum_{\{M_j^{c_k} \| M_j^{c_k} \in C_k \wedge M_j^{c_k} \text{access} I_i \wedge I_j\}} (numref(M_j^{c_k}, I_i) + numref(M_j^{c_k}, I_j))$ ■

**Definition 5.6** *Method Access frequency macco($I_m$)* of an object ($I_m$) is the sum of the access frequencies of all the applications $\mathcal{Q}_i$ accessing the object plus the sum of all method references to the object. Thus, $macco(I_m) = \sum_i^q acc(q_i, I_m) + \sum_{\forall M_j^i | I_m \in oref(M_j^i)} numref(M_j^i, I_m)$. ■

**Definition 5.7** *Method relevant access MRA($F_i^d$,$F_i^p$)* between a derived fragment $F_i^d$ and primary fragment $F_i^p$ of a class $C_i$, measures the number of times objects of both fragments are accessed by methods of other objects. Thus, MRA($F_i^d$,$F_i^p$) = $\sum_{I_m \in \{F_i^d \cap F_i^p\}}$ numref($M_j^i$,$I_m$) for any $M_j^i$. ■

**Definition 5.8** *Method Irrelevant access MIA($F_i^d$,$F_i^p$)* between a derived fragment $F_i^d$ and a primary fragment $F_i^p$, measures the number of times objects members of either of the fragments but not both are accessed by methods of other objects.

MIA($F_i^d$,$F_i^p$) = $\sum_{I_m \| I_m \in \{((F_i^p \cup F_i^d) - (F_i^d \cap F_i^p))\}} acc(q_p, I_m) + \sum_{I_m \| I_m \in \{((F_i^p \cup F_i^d) - (F_i^d \cap F_i^p))\}} numref(M_j^i, I_m)$ for all applications $q_p$ and methods $M_j^i$ accessing this object. ■

**Definition 5.9** *Object Fragment Affinity* between an object $I_m$ and a horizontal fragment $F_i^h$, denoted ofaff($I_m$,$F_i^h$), is the amount of accesses made to fragment $F_i^h$ *vis a vis* the object $I_m$.

ofaff($I_m$,$F_i^h$) = $\sum_{I_j \in F_i^h}$ maffo($I_m$,$I_j$), m ≠ j. ■

**Definition 5.10** *Method Affinity between Fragments maff($F^d$,$F^p$)* is the difference between relevant and irrelevant access: maff($F_i^d$,$F_i^p$)= MRA($F_i^d$,$F_i^p$) - MIA($F_i^d$,$F_i^p$). ■

**Affinity Rule 5.1** Select the primary fragment that maximizes the method affinity measure maff$(F_i^d, F_i^p)$ where $F_i^d$ is the derived fragment generated from method dependencies and $F_i^p$ is a primary fragment in the class ranging over all candidate fragments. ■

**Affinity Rule 5.2** The primary fragment $F_i^p$ that maximizes the function ofaff$(I_k, F_i^p)$ is where $I_m$ is placed. ■

Thus three dependence types between classes are captured in a complete design with the most complex object model (Section 6). The inheritance relationship between classes is captured in the fragmentation scheme by propagating the fragmentation of a subclass to a superclass so that derived fragments of a superclass, based on the fragments of its subclasses, is obtained. The second type of dependence is from the complex hierarchy. If a class $C_i$ contains another class $C_j$ as part of it, the fragmentation of the contained class $C_j$ is used to obtain derived fragments of the containing class $C_i$. The third is method dependence. If objects of a fragment of a class $C_i$ invoke methods in objects of another class $C_j$, we want to group all objects of the class $C_j$ invoked by all objects of the fragment of $C_i$ into a derived fragment of $C_j$, so that these two fragments are allocated together.

## 5.1 The Algorithm

This section presents an algorithm where objects consist of simple attributes and complex methods. In this model, only two types of dependencies exist between classes – inheritance hierarchy and method dependencies. Thus, algorithm *Hor_SA_CM* (Figure 8) first generates class fragments from the inheritance hierarchy information using *HorizontalFrag* (line (1)). The second iteration captures the method dependency information using object reference statistics. The objective of this iteration is to take each existing fragment of every class, say class $C_k$ in the object base and produce a derived fragment $F_i^d$ of the class being fragmented $C_i$, due to complex method dependencies of objects of this other class fragment $F_k^j$. It further selects a primary fragment $F_i^p$ of this class $C_i$ most appropriate to merge with this derived fragment $F_i^d$ and eventually produces a non-overlapping set of horizontal fragments that has incorporated method dependency information. This second iteration is formally defined as algorithm *Derived_From_CompM* of Figure 7. The first step of Figure 7 takes every class (line (1)) and creates a set of objects referenced by each class's fragments (line (3-5)), called the *derived fragment*. The algorithm then determines which primary fragment the derived fragment has the greatest affinity with (line (6-13)) and adds it to the appropriate primary fragment (line (14)). Finally, since it is possible to place an object in more than one primary fragment, Affinity Rule 5.2 is used to find the primary fragment with which it has the greatest affinity (line (15-22)) and it is removed from all other (line (23-25)).

**Algorithm 5.1** *(Generate Derived Fragment Based on Complex Methods)*

**Algorithm Derived_From_CompM**
**input**: $\mathcal{F}_{c_i}$: set of horizontal fragments of the classes
$oref(M_i^j)$: set of object references for all $M_i^j \in C_i$.
$\mathcal{C}_d$ : set of database classes
**output**: $\mathcal{F}_{c_i}$: set of horizontal fragments of the classes
**var**
$\mathcal{F}_i^d$ : set of derived horizontal fragments for class $C_i$
Prifragforderived, Prifragforobj : set of objects;
k : integer;
**begin**
This **algorithm** captures the method dependency information
**for every** fragment $F_{ik} \in$ class $\mathcal{F}_{c_i}$ (1)
$F_i^d = \emptyset$ (2)
**for each** class $C_m \in C_d$ **do** (3)
**if** $m \neq i$ **then** (4)
$F_i^d = F_i^d \cup$ coref($F_{ik}$,$C_m$) (5)
**end; {for** $C_m$**}**
**end; {for** $F_{ik}$**}**
**for each** primary fragment $F^p \in \mathcal{F}_{c_i}$ of class $C_i$, (6)
k = 1 (7)
Prifragforderived = $F_k^p \in \mathcal{F}_{c_i}$ (8)
**for every** fragment $F_i^p \in \mathcal{F}_{c_i}$ (9)
**if** maff($F_i^d$,$F_i^p$) > maff($F_i^d$,$F_k^p$) (10)
**then begin**
Prifragforderived = $F_i^p$ (11)
k = i (12)
**end; {begin}**
**end; {for** $F_i^p$**}**
$F_i^p$ = Prifragforderived (13)
$F_i^p = F_i^p \cup F^d$ (14)
**end; {for** $F_i^d$**}**
**for each** overlapping object $I_i$ of class fragments ($F_{i1}^p$,...,$F_{in}^p$) (15)
k = 1 (16)
Prifragforobject= $F_k^p \in C_i$ (17)
**for every** fragment $F_i^p \in$ class $C_i$ (18)
**if** ofaff($I_i$,$F_i^p$) > ofaff($I_i$,$F_k^p$), (19)
**then begin**
Prifragforobject = $F_i^p$ (20)
k = i (21)
**end; {begin}**
**end; {for** $F_i^p$**}**
$F_i^p$ = Prifragforobject (22)
**for every** fragment $F_k^p \in$ class $C_i$ (23)
**if** ($I_i \in F_k^p$) and ($F_k^p \neq F_i^p$) (24)
**then** $F_k^p = F_k^p - I_i$ (25)
**end; {for** $F_k^p$**}**
**end; {for** $I_i$**}**
**end; {of Derived_From_CompM}**

Figure 7: Capturing the Complex Method dependency information Class Fragments

**Algorithm 5.2** *(Simple Attribute and Complex Methods)*

**Algorithm Hor_SA_CM**
**input**: $\mathcal{Q}_i$: set of user queries
$\mathcal{C}_d$ : set of database classes
**L(C)** : the class lattice
$oref(M_i^j)$: set of object references for all $M_i^j \in C_i$.
**output**: $\mathcal{F}_{c_i}$: set of horizontal fragments of the classes
**var**
$\mathcal{P}_i$ : set of simple predicates for class $C_i$
$\mathcal{F}_i^p$ : set of primary horizontal fragments for class $C_i$
$\mathcal{F}_i^d$ : set of derived horizontal fragments for class $C_i$
$\mathcal{M}_c^L$ : set of member classes for the link graph L(C)
**begin**
$\mathcal{F}_{c_i}$ **= HorizontalFrag(**$\mathcal{Q}_i$, $\mathcal{C}_d$, **L(C))** (1)
**for each** class $C_i \in C_d$ **do** (2)
$\mathcal{F}_{c_i}$ = Derived_From_CompM($\mathcal{F}_{c_i}$,$C_i$,$oref(M_i^j)$,$\mathcal{C}_d$) (3)
**end; {for** $C_i$**}**
**end; {of Hor_SA_CM}**

Figure 8: Horizontal Fragmentation For Simple attributes and Complex methods

# 6 Complex Attributes and Complex Methods

This section presents an algorithm for horizontally fragmenting classes consisting of complex attributes and complex methods. With this model, the database information that needs to be captured include: the inheritance hierarchy, the attribute link to reflect the part-of hierarchy, and the method links to reflect the use of methods of other classes by fragments of a class. The algorithm is essentially the sum of the previous model's algorithms. The first and second iterations of this algorithm involve generating a set of horizontal fragments that capture the inheritance hierarchy and attribute link information using the same technique presented in Section 3. During the third iteration, the method link information is captured by generating a set of derived fragments using the same techniques as in the second iteration of the algorithm presented in Section 5 with algorithm *Derived_From_CompM* of Figure 7. These final derived fragments are merged with horizontal fragments from the second iterations to obtain the final fragments of the classes. The formal presentation of this algorithm *Hor_CA_CM* is given in Figure 9. We illustrate the application of the third iteration of this algorithm using an example given in the following section.

## 6.1 An Example

This example incorporates class models consisting of complex attributes and complex methods. The extended complex class object base is given in Figure 10 [5].

[5]For readability, we have preceded each key attribute name of a class with $k$, each attribute name with an $a$ and each method name with an $m$.

**Algorithm 6.1** *(Complex Attribute and Complex Method)*

**Algorithm Hor_CA_CM**
**input**: $\mathcal{Q}_i$: set of user queries
$\mathcal{C}_d$ : set of database classes
**L(C)** : the class lattice
**A(C)** : the class composition hierarchy with attribute link between classes
$oref(M_i^j)$: set of object references for all $M_i^j \in C_i$.
**output**: $\mathcal{F}_{c_i}$: set of horizontal fragments of the classes
**var**
$\mathcal{P}_i$ : set of simple predicates for class $C_i$
$\mathcal{F}_i^p$ : set of primary horizontal fragments for class $C_i$
$\mathcal{F}_i^d$ : set of derived horizontal fragments for class $C_i$
$\mathcal{M}_c^L$ : set of member classes for the link graph L(C)
**begin**
$\mathcal{F}_{c_i}$ **= HorizontalFrag(**$\mathcal{Q}_i$, $\mathcal{C}_d$**, L(C))** (1)
$\mathcal{F}_{c_i}$ **= HorizontalFrag(**$\mathcal{Q}_i$**, (**$\mathcal{F}_{c_i}$**, A(C))** (2)
**for each** class $C_i \in C_d$ **do** (3)
$\mathcal{F}_{c_i}$ = Derived_From_CompM($\mathcal{F}_{c_i}$,$C_i$,$oref(M_i^j)$,$\mathcal{C}_d$) (4)
**end; {for** $C_i$**}**
**end; {Hor_CA_CM}**

Figure 9: Horizontal Fragmentation For Complex Attribute and Complex Methods

The database schema information that is one input to this design process consists of the class lattice of the object base which is as given in Figure 11, and the class composition hierarchy in Figure 12. The other input is the application information $Q_1$ to $Q_5$ given below.

$Q_1$**:** This application groups grads according to their area of specialization which is determined by the name of their supervisor. The methods used are defined on class **Grad** and the predicates are:

$$\{P_1\text{: supervisor = ``Prof John West''},P_2\text{: supervisor = ``Prof Mary Smith''}\}$$

$Q_2$**:** This application groups profs by their addresses. The methods used are defined on class **Prof** with the following predicates:

$$\{P_1\text{: address = ``Winnipeg''},P_2\text{: address = ``Vancouver''},P_2\text{: address = ``Toronto''}\}$$

$Q_3$**:** This application separates profs with salaries greater than or equal to \$60,000 from those with salaries less than \$60,000. The methods are in the class **Prof** with the following predicates:

$$\{P_4\text{:salary} \geq 60000,P_5\text{:salary} < 60000\}$$

$Q_4$**:** Groups students by their departments. The methods are from class **Student** and the predicates used are:

$$\{P_1\text{:dept=``Math''},P_2\text{:dept=``Computer Sc''},P_3\text{:dept=``Stats''}\}$$

Person = {k.ssno,{a.name,a.age,a.address},{m.whatlast,m.daysold,m.newaddr},
{ $I_1$ {Person1,John James,30,Winnipeg}
$I_2$ {Person2,Ted Man,16,Winnipeg}
$I_3$ {Person3,Mary Ross,21,Vancouver}
$I_4$ {Person4,Peter Eye,23,Toronto}
$I_5$ {Person5,Mary Smith,40,Toronto}
$I_6$ {Person6,John West,32,Vancouver}
$I_7$ {Person7,Jacky Brown,35,Winnipeg}
$I_8$ {Person8,Sean Dam,27,Toronto} } }
$I_9$ {Person9,Bill Jeans,43,Vancouver}
$I_{10}$ {Person10,Mandu Nom,30,Winnipeg}
Prof = Person pointer ⊙ {k.empno,{a.status,a.dept,a.salary,a.student},{m.coursetaught,m.whatsalary},
{ $I_1$ (person pointer5) ⊙ {Prof1,asst prof,Computer Sc.,45000,students pointers}
$I_2$ (person pointer6) ⊙ {Prof2,assoc prof,Math,60000,students pointers}
$I_3$ (person pointer9) ⊙ {Prof3,full prof,Math, 80000,students pointers}
$I_4$ (person pointer10) ⊙ {Prof4,full prof,Math,82000,students pointers} } }
Student = Person pointer ⊙ {k.stuno,a.dept,a.feespd,a.coursetaken} ,
{m.stuno-of,m.dept-of,m.owing},
{ $I_1$ (person pointer1) ⊙ {Student1,Math,Y}
$I_2$ (person pointer4) ⊙ {Student2,Computer Sc.,N}
$I_3$ (person pointer2) ⊙ {Student3,Stats,Y}
$I_4$ (person pointer3) ⊙ {Student4,Computer Sc.,N} }
$I_5$ (person pointer7) ⊙ {Student5,Math,Y}
$I_6$ (person pointer8) ⊙ {Student6,Stats,N} } } }
Grad = Student pointer ⊙ {k.gradstuno,{a.supervisor},{m.whatprog}
{ $I_1$ (Student pointer1) ⊙ {Grad1,John West}
$I_2$ (Student Pointer2) ⊙ {Grad2,Mary Smith} } }
$I_3$ (Student Pointer5) ⊙ {Grad3,Mary Smith } } }
UnderG = Student
$I_1$ (Student pointer3)
$I_2$ (Student pointer4)
$I_3$ (Student pointer6)
Dept = {k.code,{a.name,a.profs,a.students},{m.number-of-profs}},
{ $I_1$ {Dept1,{Computer science,prof pointers,student pointers}
$I_2$ {Dept2,{Math,prof pointers,student pointers}
$I_3$ {Dept3,{Actuary science,prof pointers,student pointers}
$I_4$ {Dept4,{Stats,prof pointers,student pointers} } }

Figure 10: The Sample Object Database Schema

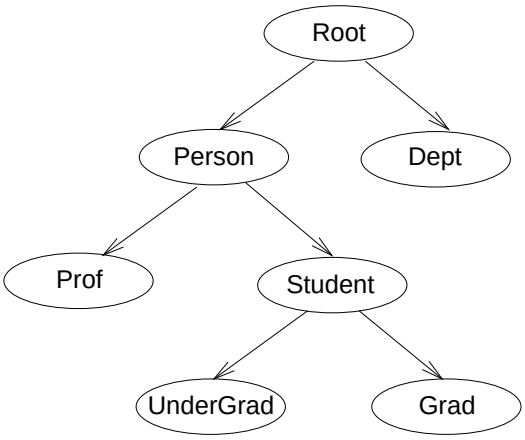


Figure 11: Complex Class Lattice

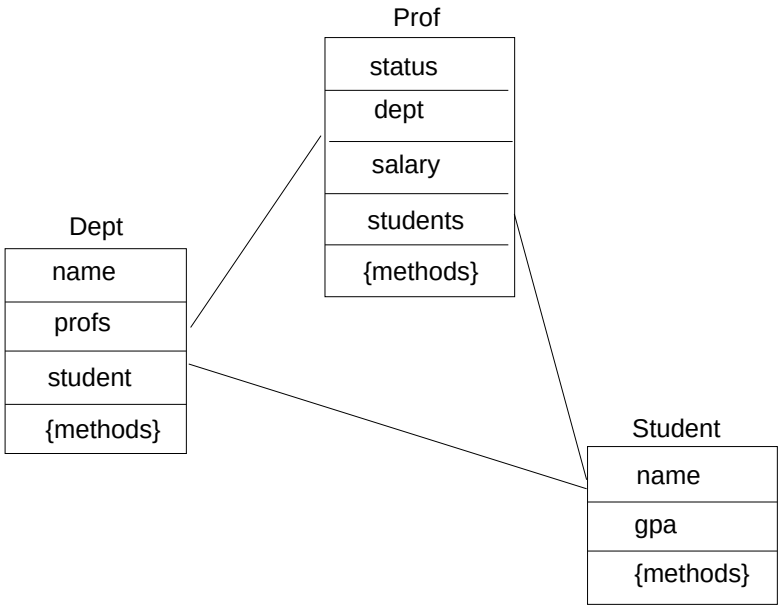


Figure 12: Class Composition Hierarchy

$Q_5$**:** This application groups departments by their general area. The methods are for the class **Dept** and the predicates used are:

$$\{P_1\text{:dept=“Math”},P_2\text{:dept=“Computer Sc”},P_3\text{:“Stats”}\}$$

**Fragmentation Process : Execution of Algorithm Hor_CA_CM**

Line 1 of algorithm Hor_CA_CM (Figure 9) generates the horizontal fragment using the complex class lattice as follows:

- Generates the complex class link graph shown in Figure 13 from Figure 11.
- Defines primary horizontal fragments for all owner classes on the link graph to obtain:

  **Class Grad**

  Fragments $F_1^p$ = instance object 1 $\{I_1\}$

  $F_2^p$ = instance objects 2 and 3 $\{I_2,I_3\}$

  **Class Student**

  Fragments $F_1^p = \{I_1,I_5\}$, $F_2^p = \{I_2,I_4\}$, and $F_3^p = \{I_3,I_6\}$.

  **Class Prof**

  Fragments $F_1^p = \{I_4\}$, $F_2^p = \emptyset$, $F_3^p = \{I_2,I_3\}$, $F_4^p = \emptyset$, $F_5^p = \emptyset$, and $F_6^p = \{I_1\}$

  **Class Dept**

  Fragments $F_1^p = \{I_2,I_3\}$, $F_2^p = \{I_1\}$, and $F_3^p = \{I_4\}$.

- Generates derived fragments of member classes to obtain: **Class Student**

  Using the primary fragments of the owner class Grad, the derived fragments are:

  $F_1^d = \{I_1\}$, and $F_2^d = \{I_2,I_5\}$. **Derived fragments of the Class Person**

  The derived horizontal fragments of Person based on owner class (Student), are:

  $F_1^d = \{I_1,I_7\}$, $F_2^d = \{I_4,I_3\}$, and $F_3^d = \{I_2,I_8\}$.

  Derived fragments of Person based on the owner class Prof are:

  $F_4^d = \{I_{10}\}$, $F_5^d = \{I_6,I_9\}$, and $F_6^d = \{I_5\}$.

- Combines Primary and Derived fragments of member classes using the following access frequency information. The quantitative application information needed concern the access frequencies of applications to different groups of data in each class and we assume the following access frequency information[6]. For example, the first primary fragment of the class *Student* $F_1^p$ corresponds to the minterm predicate {dept= “Math” $\wedge dept \neq$ “$ComputerSc''$ $\wedge$ $dept \neq$ “$Stats''$} and this fragment contains the set of instance objects $\{I_1,I_5\}$. Since the access frequency to objects of this first fragment by application $Q_1$ is 10, and no other application accesses these objects, the access

[6] The fragments referred to in the access statistics are primary fragments generated from minterm predicates.

frequency of each of these objects is 10. In the case where minterms of a class are defined from predicates arising from more than one application, access frequency of an object in each fragment is obtained by adding relevant access frequencies of the relevant predicates of the two applications. For example, instance object $\{I_4\}$ of the $F_1^p$ of the class *Prof* has access frequency of (5 + 20) from the two applications $Q_2$ and $Q_3$ accessing this class respectively.

**Class Grad** : acc$(Q_1, F_1) = 20$, acc$(Q_1, F_2) = 10$.

**Class Prof** :

acc$(Q_2, F_1) = 5$, acc$(Q_2, F_2)$=10, acc$(Q_2, F_3)$=10, acc$(Q_3, F_1) = 20$, acc$(Q_3, F_2) = 5$.

**Class Student** : acc$(Q_4, F_1) = 10$, acc$(Q_4, F_2) = 5$, acc$(Q_4, F_3) = 0$.

**Member Class Student**

$F_1^d$ has the maximum affinity with $F_1^p$ of 0, and so we merge $F_1^d$ and $F_1^p$ to get :

$F_1^h$ =$\{I_1, I_5\}$

Details of this computation is given in the next three lines

$(aff(F_1^d, F_1^p) = 10 - 10 = 0$

$aff(F_1^d, F_2^p) = 0 - (10 + 5 + 5) = -20$

$aff(F_1^d, F_3^p) = 0 - (10 + 0 + 0) = -10)$

$F_2^d$ has the maximum affinity with $F_1^p$ of -5, and so we merge

$F_2^d$ and $F_1^p$ to obtain $F_1^h$ =$\{I_1, I_5, I_2\}$

Details of this computation is given in the next three lines

$(aff(F_2^d, F_1^p) = 10 - (10 + 5) = -5$

$aff(F_2^d, F_2^p) = 5 - (5 + 10) = -10$

$aff(F_2^d, F_3^p) = 0 - (5 + 0 + 10 + 0) = -15)$

However, since overlapping object $I_2$ has maximum faff($I_2$,$F_2^h$) of 5,

we keep $I_2$ in $F_2^h$ and delete it from $F_1^h$.

$F_2^h$ =$\{I_2, I_4\}$

The third primary fragment remains the same as

$F_3^h$ =$\{I_3, I_6\}$

**Class Person**

Since the class **Person** has no primary fragments so the derived fragments become primary.

This completes the first iteration. The second iteration is the execution of line 2 of algorithm *Hor_CA_CM* (Figure 9) using the fragments from the first iteration and the class composition hierarchy. The link graph generated from the class composition hierarchy is given in Figure 14.

We generate derived fragments of the member classes *Dept and Prof* based on applications on owner class *Student*. This is done by generating fragments of Prof that satisfy the dominating predicates of the class Student which are: $P_{student}$ : {dept="Math",dept="Computer Sc",dept="Stats"}. This

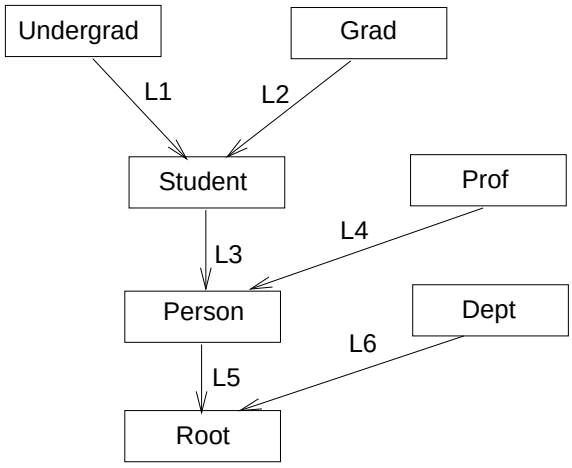


Figure 13: Complex Class Lattice's Link Graph

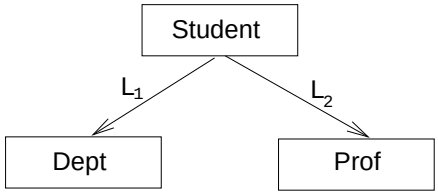


Figure 14: Class Composition Hierarchy Linkgraph

leads to the following three derived fragments of the class Prof.

**Class Prof**

$F_1^d = \{I_2, I_3, I_4\}$, $F_2^d = \{I_1\}$, and $F_3^d = (\emptyset)$

Derived Fragments for member class Dept

$F_1^d = \{I_2, I_3\}$, $F_2^d = \{I_1\}$, and $F_3^d = \{I_4\}$. Combining primary and derived fragments of member classes yields the following horizontal fragments:

**Class Prof**

$F_1^h = \{I_2, I_3, I_4\}$

$F_2^h = \{I_1\}$

This is because $F_1^d$ has highest affinity of 35 with $F_3^p$ of *Prof* and merges with it to generate $F_1^h$. Similarly, $F_2^d$ merges with $F_6^p$ to obtain $F_2^h$. Details of this computation is given in the next six lines.

$$(aff(F_1^d, F_1^p) = 25 - (30 + 30) = -35$$
$$aff(F_1^d, F_3^p) = (30 + 30) - 25) = 35$$
$$aff(F_1^d, F_6^p) = 0 - (15 + 30 + 30 + 25) = -100)$$

$$(aff(F_2^d, F_1^p) = 0 - (15 + 25) = -40$$

$aff(F_2^d, F_3^p) = 0 - (30 + 30) = -60$

$aff(F_2^d, F_6^p) = 25)$

**Class Dept**

$F_1^h = \{I_2, I_3\}$

$F_2^h = \{I_1\}$

$F_3^h = \{I_4\}$

Lines 3 - 5 of the algorithm captures method dependence information by defining fragments of classes that correspond to the *coref* of its fragments from the second iteration. Assume that objects of the class *Student* invoke methods of objects of classes *Prof* and *Dept*. Since *Student* is not a subclass or contained class of any of these two classes, then use of methods of these classes by class *Student* can not be treated under inheritance or class composition hierarchies, but as nested or complex method link. The pattern of method invocation of objects of a fragment of the class *Student* is shown in Figure 15. Using *oref* statistics, the derived fragments of classes based on complex methods of the first fragment of the class *Student* is as given next. Derived fragment of the Class *Dept* is $F_1^d = \{I_1, I_2, I_3\}$. The derived horizontal fragment of *Prof* based on methods of the first fragment of the class (Student), as shown in figure 15 is: $F_1^d = \{I_1, I_2, I_3\}$. For each new derived fragment of classes affected, we find a primary fragment it can combine with by applying method reference statistics to select the primary fragment with which it has highest maff. This computation requires more quantitative information about access pattern of methods. Assume the following quantitative statistics.

numref($M_3^{student}, I_1^{prof}$) = 15, numref($M_2^{student}, I_2^{prof}$) = 30, numref($M_3^{student}, I_3^{prof}$) = 20,

numref($M_1^{student}, I_1^{prof}$) = 40, numref($M_1^{student}, I_3^{prof}$) = 20.

**Class Prof**

We are now ready to select a current horizontal fragment of *Prof* generated from the second iteration that is suitable to merge with each derived fragment of *Prof* generated according to method dependencies of class fragments. For purposes of simplicity, we show one such derived fragment $F_1^d = \{I_1, I_2, I_3\}$ and the current horizontal fragments of *Prof* are $F_1^h = \{I_2, I_3, I_4\}$ and $F_2^h = \{I_1\}$. Since $F_1^h$ has highest maff($F_1^d, F_1^h$) with the derived fragment $F_1^d$ of 30, we merge $F_1^d$ with $F_1^h$ to obtain the fragment $F_1^h = \{I_1, I_2, I_3, I_4\}$

**Class Student**

Here, we assume that application information leads to the merge of $F_1^d$ and $F_2^h$ to generate $\{I_2, I_3, I_4, I_5\}$. This leads to $I_3$ and $I_5$ overlapping in both $F_2^h$ and $F_3^h$ as well as in $F_1^h$. Thus, we apply Affinity Rule 5.2 to the overlapping objects and the contending fragments to decide the best placement for them. The result is that $I_5$ gets removed from $F_2^h$ and $I_3$ gets removed from $F_3^h$ to produce the following final fragments for the class *Student*. $F_1^d = \{I_2, I_3, I_4, I_5\}$ and final fragments are: $F_1^h = \{I_1, I_5\}$, $F_2^h = \{I_2, I_4, I_3\}$ and $F_3^h = \{I_6\}$.

From Figure 15, the collection of all objects of $C_2$ (Prof) invoked by methods of objects in fragment

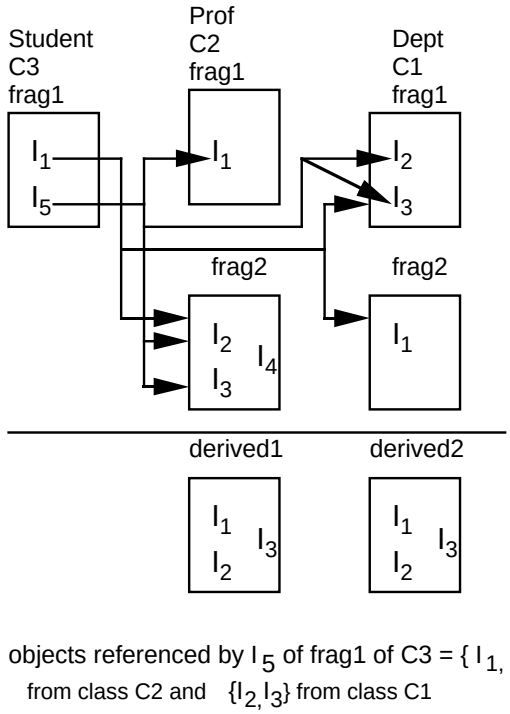


Figure 15: Generation of Derived Fragments of Classes based on Complex Methods

F1 of $C_3$ (Student) is $\{I_1,I_2,I_3\}$ and the collection of all objects of $C_1$ (Dept) invoked by methods of fragment F1 of $C_3$ (Student) is $\{I_1,I_2,I_3\}$. These collections form derived fragments of these classes based on complex methods of a fragment of *Student*. Next, quantitative information about number of accesses made by these methods to these objects are used in affinity rules to merge the set of primary and derived fragments of each class.

# 7 The Structure Chart and Complexities of Algorithms

This section discusses other important characteristics of the algorithms proposed in this paper. In the first part, a structure chart that concisely defines the fragmentation scheme for the most complex class model consisting of complex attributes and complex methods is presented; and secondly the complexities of the fragmentation schemes are discussed.

## 7.1 The Structure Chart

An interesting feature of this design is reusability of design for a simpler class model in a more complex class model. This is illustrated concisely using the structure chart for the fragmentation scheme of the most complex class model - consisting of complex attributes and complex methods, shown in Figure 16. The structure chart also gives a summary of all the algorithms that are used in the design.

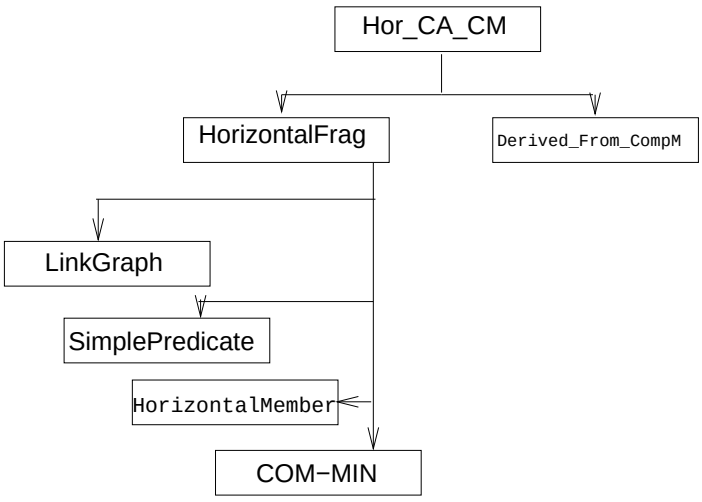


Figure 16: The Structure Chart For Model with Complex Attributes and Complex Methods

## 7.2 Complexities of Algorithms

The benefits of distributed database design generally outweigh the impending overheads involved in running the partitioning algorithms. This argument is supported by the time complexities of the fragmentation algorithms for the four class models presented in Sections 3 through 6 on the basis of the worst case analysis. Assume that $f$ represents the largest number of fragments possible, $o$ represents the largest number of instance objects in a class or a fragment, $q$ is the largest number of user queries, $c$ is the largest number of classes in the database, $m$ is the largest number of method invocations in a user query, while $p$ is the largest number of predicates in a method invocation and $p_c$ is the largest number of predicates in a class. The horizontal fragmentation algorithm for the class model consisting of simple attributes and simple methods *HorizontalFrag* of Figure 5 is of time complexity O($c^3$ + qmp + cf + $c^2$o + c$f^4$o + fc) which simplifies to O($c^3$ + qmp + $c^2$o + c$f^4$o). This is because line 1 of this algorithm *HorizontalFrag* which is the *Linkgraph* algorithm is of O($c^3$), and generation of simple predicates (line 2) is of O(qmp). Lines 5 through 7 of *HorizontalFrag* are of O($c^2$o) because the object join operation is of time O(o); while lines 8 through 10 have time complexity O(c$f^4$o) since the algorithm *HorizontalMember* (line 9) is of time O($f^4$o). The prevailing time complexity of the algorithm depends on number of classes in the object base and the number of queries and objects in a class. In a system with many more queries than classes, it is likely that the term *qmp* becomes the dominant term and that makes the complexity of *HorizontalFrag* O(qmp).

The time complexity of the algorithm *Hor_CA_SM* for the second model with complex attribute and simple method in Figure 6 is the same as that for HorizontalFrag and is O($c^3$ + qmp + $c^2$o + c$f^4$o). Both the third and fourth class models consisting of simple attributes and complex methods; and complex attributes and complex methods respectively, have time complexity which is O($c^3$ + qmp + $c^2$o + c$f^4$o + $c^2$fo) which simplifies to O($c^3$ + qmp + c$f^4$o + $c^2$fo). This is because the algorithm

*Derived_From_CompM* of Figure 7 is of time complexity O(cfo). Thus, these algorithms are polynomial in the sizes of these variable inputs.

# 8 Conclusions

This paper defines issues involved in the class fragmentation in a distributed object based system. The issues considered include the inheritance hierarchy, the nature of attributes of a class, and the nature of methods in the classes. The paper proposes algorithms for horizontal fragmentation of four class object models: (1) a model consisting of classes with simple attributes and methods, (2) a model consisting of classes with attributes that support a class composition hierarchy using simple methods, and (3) a model consisting of classes with complex attributes and simple methods, and (4) a model consisting of classes with complex attributes and complex methods. The algorithm necessary to support the first simple model is provided. A description and formal algorithms showing the iterative applications and extensions of this algorithm for fragmentation of each of the more complex models are also provided. The use and application of the algorithm for fragmenting the most complex model, consisting of complex attributes and methods are illustrated with an example. Finally, complexities of these algorithms are discussed and shown to be polynomial.

Current research efforts include developing performance measurements to demonstrate the utility of our algorithms. Such performance analysis is difficult because very few of these systems (or more specifically, applications on these systems) have been developed. We are currently analyzing object-based applications to determine the way they are being used with the goal of determining metrics for such a performance analysis. Ideally these techniques can be modified so they can be used in a dynamic environment where data is added and removed. Unfortunately, such an environment is very complicated because supporting it involves not only the accurate placement of fragments but also the need to transparently migrate object fragments while the system is being accessed by users. The initial step in this research requires that we determine a performance threshold below which dynamic redesign is required so the system will continue to meet its performance goals. The current research is attempting to determine if these techniques can be modified so that each iteration of the design process can be accomplished by only analyzing new data added to the system while updating those fragments that had been previously allocated. Furthermore, we are working on vertical and hybrid fragmentation schemes as well as discovering suitable theoretical models to allocate fragments to distributed sites. Finally, we are defining a distributed directory that supports distributed class fragments and implementing it on a heterogeneous network.

# References

[1] Elisa Bertino and Won kim. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2), 1989.

[2] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. SIGPLAN Notices, 1982.

[3] S. Ceri and S.B.Navathe. A comprehensive approach to fragmentation and allocation of data in distributed databases. In *Proceedings of the IEEE COMPCON Conference*, 1983.

[4] S. Ceri, S.Navathe, and G. Wiederhold. Distributed design of logical database schemas. *IEEE Transactions on Software Engeneering*, 9(4), 1983.

[5] C.I. Ezeife and K.E. Barker. Horizontal class fragmentation in a distributed object based system. Technical Report TR 93-04, Univ. of Manitoba Dept of Computer Science, October 1993.

[6] C.I. Ezeife and K.E. Barker. Horizontal class fragmentation in distributed object based systems. In *Proceedings of the Second Biennial European Joint Conference on Engineering Systems Design and Analysis*. ASME Publications, 1994.

[7] M.F. Hornick and S.B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office Information Systems*, 5(1), Jan. 1987.

[8] Itasca Systems Inc. Itasca distributed object database management system. Technical Report Technical Summary Release 2.0, Itasca Systems Inc., 1991.

[9] K. Karlapalem, S.B.Navathe, and M.M.A.Morsi. Issues in distribution design of object-oriented databases. In M. Tamer Ozsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*, pages 148–164. Morgan Kaufmann Publishers, 1994.

[10] M.L. Kersten, S. Plomp, and C.A Van Den Berg. Object storage management in goblin. In M. Tamer Ozsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.

[11] W. Kim. Object-oriented databases: Definition and research directions. *IEEE Transactions on knowledge and Data Engineering*, 2(3), Sept. 1990.

[12] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in thor. In M. Tamer Ozsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.

[13] S.B. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems*, 9(4), 1984.

[14] S.B. Navathe, K. Karlapalem, and M. Ra. A mixed fragmentation methodology for initial distributed database design. In *Technical Report*. CIS Dept, Univ. of Florida, Gainesville, FL, 1990. TR 90-17.

[15] S.B. Navathe and M. Ra. Vertical partitioning for database design: A graphical algorithm. In *Proceedings of the ACM SIGMOD*. SIGPLAN Notices, 1989.

[16] Gruber Oliver and Amsaleg Laurent. Object grouping in eos. In M. Tamer Ozsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.

[17] M.T. Ozsu and P.Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[18] D. Shin and K.B. Irani. Fragmenting relations horizontally using a knowledge-based approach. *IEEE Transactions on Software Engineering*, 17(9), Sept. 1991.

[19] G. Wiederhold. *Database Design*. McGraw-Hill, New York, 1982.

[20] S.B. Yao, S.B. Navathe, and J.L. Weldon. An integrated approach to database design. New York, 1982. Lecture Notes in Computer Science 132.