CHAPTER 2

LITERATURE REVIEW

2.1 Preamble

Object Oriented Databases (OODB) are widely used for many advanced applications like CAD, CAM etc. because of its modeling support to represent complex data and their complex relationships. Complex data are represented as objects. Complex relationships are defined by combinations of object relationships such as inheritance, composition (aggregation) and association. This modeling power makes OODBMS to have high potential for many of the future applications.

OODBMS is a collection of objects. The objects are classified into classes and instances. A class is a collection of instances. There are two types of accesses to OODBMS. Users may access the OODBMS for data (runtime transactions) or schema (design time transactions). A transaction in OODBMS is defined as partially ordered set of method invocations on objects [Agrawal1992]. A typical runtime transaction involves execution of associated methods (also called member functions) to read or alter the value of attributes in an instance. The values of the attributes map on to the data in the underlying database. The runtime access to the database can be at class level (involving all the instances in a class) or at instance level (involving any one instance in a class) based on the property of the methods [Riehle2000b]. The design time transaction involves reading and modifying the structure of the domain. The domain structure is represented by schema in databases. Hence, design time transactions are used to alter the schema. Since it is OODBMS, the structure is defined by a set of related classes participating in the domain. The collection of related classes is called as class lattice. It is represented using a class diagram. Class lattice is a group of classes related by inheritance, aggregation and association relationships. The access to the database can be a read or write operation. The read operations can be executed in shared lock mode and write operations should be executed in exclusive lock mode to avoid dirty reads and dirty writes.

Existing concurrency control schemes cannot be adopted for object oriented environments because of the following reasons:

- The inherent complex nature of objects is not exploited in promoting concurrency.
• The rich structural and behavioral semantics of the objects provide better performance.
• The transactions in object oriented environments are long duration in nature. Existing concurrency control schemes are not equipped to support them. Preemption of such long transactions due to incompatibility under utilizes system resources. At the same time, letting a transaction to hold resources for longer duration may delay other transactions and reduce the throughput.

Object Oriented Distributed Systems (OODS) blend the benefits of distributed systems and object oriented programming. While distributed systems promote resource sharing, object oriented programming helps to simplify the design of complex systems by its bottom up approach. In OODS, the reusable data resources are modeled as objects. The objects in server tier, encapsulate the state and methods that implements the business logic of the domain. The set of values of attributes define the state of the object. The methods implement the business logic and operate on the attributes to serve the clients. The objects residing in the server tier access the persistent store in the database tier to update their states.

In distributed systems, the database tier could be modeled as relational database or object oriented database. The concurrency control mechanisms are usually applied on the databases in the data store tier as in figure 2.1. This eliminates the possibility of using legacy data sources like files that are simple in nature and lack in concurrency control mechanisms.

In OODS, the concurrency control mechanisms can be shifted to application server tier as in figure 2.2. Then concurrency control mechanisms can be applied on the objects in the application server tier. The shift of concurrency control mechanism from database tier to application server tier aids in supporting all types of persistent store of data. The other advantage is rather than defining concurrency control mechanisms.

Figure 2.1 Concurrency control at database tier
mechanisms for each type of data store, a common concurrency control mechanism can be proposed for objects.

![Concurrency control at server tier](image)

**Figure 2.2** Concurrency control at server tier

The possibility of shifting concurrency control mechanisms from databases in the database tier to objects in the server tier may be explored so that any form of data persistence can be supported. This is because the client requests for data in the database tier can be accessed only through the objects in server tier. So this will help OODS to support legacy file formats also. Then obviously concurrency control mechanisms defined for OODBMS is a good place to look for defining optimal concurrency control algorithms in OODS.

Though concurrency control mechanisms in OODBMS can be considered, they cannot be adopted as they are in OODS. This is because query language is used to request databases. But in OODS, object oriented programming languages like C++, Java are used to make client requests. Then lock types and granularity of resources are to be ascertained from the client code. The doc tools like docC++, Javadoc can be used to identify the method type and properties [Riehle2000a, Riehle2000b]. After this, the compatibility matrix used in OODBMS can be considered for adoption in OODS.

The business domain implemented in OODS may eventually be upgraded to provide better services to clients. Then it introduces the need for changing the structure of the domain. The domain is represented using class diagram in OODS. The evolution of business domain might require changes in definitions of attributes, methods, classes and their relationships. Then OODS may receive data requests as well as schema change requests for which concurrency control is to be imposed.

Then OODS will receive two types of requests: Runtime data requests and Design time requests to modify the structure of domain. Both of them can be either read or write requests. Then concurrency control has to be applied to protect the consistency of the objects. These two types of requests induce three different types of
conflicts among requests to a class diagram: conflicts among runtime requests, conflicts among design time requests and conflicts between runtime and design time requests. In OODBMS, both types of transactions can be executed in parallel as transactions are written in query languages. In OODS, the transactions are implemented in programming languages as mentioned earlier. The commutativity matrix proposed should resolve all the above conflicts.

The application of concurrency control techniques have the negative effect of resulting in deadlocks. Deadlock prevention is one of the proactive techniques to handle deadlocks. Prevention of deadlocks has the benefit of low runtime cost and better response time. Coffman1971 states that prevention algorithms work by preventing any one or more necessary conditions of deadlock namely mutual exclusion, non pre-emption, hold and wait and circular wait. It also states that prevention of deadlock, by eliminating mutual exclusion and non-preemption conditions is generally influenced by the nature of resources. Hold and wait, when it leads to circular wait results in deadlock.

Shaw1974 says that deadlock prevention can be implemented by using any of the following techniques namely collective requests, maximal claims and ordered resource allocation. Collective requests can be used for batch processing systems and are not suitable for real time systems. Maximal claims method can be used when all the resources are not required initially and resources can be requested as the execution of transaction progresses.

But Hac1989 states that all the resources are to be granted for a transaction to proceed in distributed systems. This is because the resources are scattered across various sites and none of the sites know the status of other sites. Hence checking for circular wait condition is tedious in distributed systems than in centralized systems. Then techniques like maximal claims and collective requests cannot be applied in distributed systems. Moreover, rollback of transactions on detection of deadlocks will cause more overhead. Therefore, the transactions should get all the resources they need, before they proceed. Since the transactions get all the resources before execution, their execution can continue without any wait time. However, the resource requests are to be known apriori to prevent deadlocks. In order to know the requests apriori, the resource requirements of all transactions needs to be known. Though this technique is sub optimal in utilization of resources, rollback overhead is sub optimal to this and causes increase in response time and lesser throughput. Hence, it is a trade
off between utilization of resources against throughput and response time. Further Hac1989 have shown that deadlock prevention algorithms are better than deadlock detection algorithms with better performance and response time in distributed systems.

A good Deadlock Prevention Algorithm (DPA) should avoid starvation. In DPA, access ordering defines how the simultaneous transactions should be ordered to access the resources. Poor access ordering policy leads to starvation. It involves abortion of the same transaction repeatedly. It is categorized into starvation in poverty [Holt1972] and starvation in wealth [Parnas1972]. It is an outcome of exercising concurrency control to the simultaneous transactions. They can be defined as follows:

*Starvation in poverty [Holt1972]*: A resource request made by a transaction is never satisfied there after; alternately the requested resource is assigned to other transactions repeatedly.

*Starvation in wealth [Parnas1972]*: A resource requested by a transaction is never satisfied though it is permanently satisfiable from a particular time instant.

In DPA, access ordering is usually on FIFO basis. This generally ensures fairness in the system. However, strict adherence of FIFO strategy may introduce starvation in wealth, which states that latter transaction which could have been satisfied, is kept waiting, since earlier transaction is waiting. Alternately, assigning static priority for transactions introduces starvation in poverty, which is a consequence of expedient scheduling strategy. This makes shortage of resources and makes lower priority transactions permanently blocked.

Deadlock detection is a reactive strategy for handling deadlocks. It is the best mechanism for systems with lower and moderate number of deadlocks. Deadlock can be usually detected by checking for presence of cycle in Wait-For Graph (WFG). Detection of deadlock is more difficult in distributed systems than in centralized systems. This is because the resources are distributed in different sites and transactions access them from any of these sites. They communicate only through messages. Hence in order to know the wait-for status of the transactions, a Global WFG (GWFG) has to be constructed. Selection of a victim using this GWFG is complex.

The most popular algorithm for distributed deadlock detection and resolution is the probe based algorithm by Chandy1983. In this algorithm, the transaction that suspects deadlock sends probe messages along the wait for edges of the GWFG. If the
probe returns back to the initiator, it indicates the presence of deadlock. Simultaneous initiation of probe messages by many transactions for the same deadlock may lead to phantom deadlocks. Hence priority based algorithms [Chowdary1989; Mitchell1984; Sinha1985] have been proposed. These algorithms ensure that only one probe is sent per deadlock cycle. The initiator is decided based on priority. Later several DDDR (Distributed Deadlock Detection and Resolution) algorithms have been proposed for various request models which optimize on message complexities.

All these algorithms expect the underlying system model to be fault free. In Ozsu1999, it is stated that the failures in distributed systems could be categorized as:

1. Transaction failure- bug in code
2. Site failure- processor failure
3. Link failure- communication link failure

So there is a need for fault tolerant DDDR that can handle the above mentioned failures. If faults cannot be handled, atleast the initiator should be informed about the status to avoid infinite wait.

Once a deadlock is detected, one of the transactions should be chosen as victim. Aborting it will break the cycle and thus eliminate the deadlock. The victim thus selected needs to rollback and restart later. Hence the negative outcome of the deadlock resolution is the possibility of penalization of the same transaction again and again i.e., starvation.

In section 2.2, object oriented concepts related to the research work are explained. In section 2.3, existing semantic multi-granular models for object oriented environments are explored and their lacuna are identified. In section 2.4, the adaptability of DPA to OODS is explored and the existing algorithms are analysed. The popular probe based DDDR algorithm by Chandy1983 is not fault tolerant. Survey is done to see whether there are any other fault tolerant DDDR algorithms existing. Several existing victim selection algorithms for deadlock resolution are analysed for optimality of performance. In section 2.5, the limitations of the literature survey are summarized and objectives of the research work are finalized.

### 2.2 Object Oriented Concepts

This section revisits the object oriented concepts related to the research work. The types and properties of object methods are explained first. Then the semantics of class types, attribute types and class relationships with respect to locking is discussed.
The client requests are satisfied by executing the methods defined in the object. These methods need to operate on the data to satisfy the request. The methods not only have types but also properties. Depending on the type of methods, the read or write operations can be ascertained. Then concurrency control mechanisms can be defined whenever there are R-W and W-W conflicts. Riehle2000a has classified the object methods into three types:

1. **Query method**: returns some information about the object being queried. It does not change the object’s state. There are four main query method types: *Get method, Boolean query method, Comparison method and Conversion method.*

2. **Mutation method**: changes the object’s state (mutates it). Typically, it does not return a value to the client. There are three main mutation method types: *Set method, Initialization method and Command method.*

3. **Helper methods**: performs some support task for the calling object. There are two types of helper methods: *Factory method and Assertion method.*

Apart from types, a method also has properties [Riehle2000b]. Example of method properties are whether the method is primitive or composed, whether it is available for overriding through subclasses (hook method), or whether it is a mere wrapper around a more complicated method (Template method). A method has exactly one method type, but it can have several properties. Method types and properties are orthogonal and can be composed arbitrarily.

Two types of classes are defined in object oriented systems namely *Abstract* and *Concrete classes*. Abstract classes are usually used to define the class template. Instances are not created from this type of classes. Usually they act as base classes from which one or more concrete classes are derived. Concrete classes are classes defined mainly to create instances. They support all types of methods to create, query, mutate and delete objects. The locks on concrete classes depend on the type of member method which is invoked. Both read (S) and write (X) locks must be available for them at both design time as well as runtime. So lock types for both abstract and concrete classes are to be ascertained.

In OODBMS, only instance level attributes are referred. The scope of values of these attributes is restricted to the state of the object in which they are present. They are mutually independent and directly inaccessible by other objects of the same class. In OODS, instance level attributes as well as class level attributes are present. The class level attributes are shared by all instances of a class. They are also called as
static attributes of a class. For e.g., `nextregno` can be defined as a static member in the
student class to generate the next register number for a new student object. Hence the
smallest granule size for instance level attributes could be object or individual
attributes, whereas the granule size of class level attribute can be as small as a class.

As mentioned earlier, the classes are related by inheritance, aggregation and
association relationships. The inheritance relationship also called as “IS A”
relationship is sub divided into single inheritance, multi level inheritance, multiple
inheritance, hierarchical inheritance and hybrid inheritance. The inheritance
relationship except multiple inheritance can be represented using tree structure and is
called class hierarchy. The inclusion of multiple inheritance will lead to network
structure and is called class lattice.

The aggregation also called as “HAS A” relationship defines the containment of
component objects in a composite object. The composite object uses the services of
component objects to provide its service. There are two types of aggregation namely
strong and weak aggregation. The weak aggregation is a subtype of association and
hence the rules used for association can also be extended to this. The strong
aggregation is also called as composition and defines “PART OF” relationship. The
composition [Kim1989] can be classified into dependent or independent based on the
dependence of creation and deletion of component objects on composite objects. The
composition is also classified into shared or exclusive based on the possibility of
sharing component objects by more than one composite object.

The association relationship defines the USING relationship, where one or more
objects use the service of an object. Since it is an object relationship, a binary
association can be treated as shared composition with single component and N-ary
association can be treated as shared composition with multiple component objects.
The rules defined for composition may be extended to association.

Garza1988 and Kim1989 have explored the types and properties of inheritance
and aggregation. However it is worth noting certain points regarding these
relationships:

1. Transactions can request a single object or all the objects of a class based on the
   member function present in it. The property of the member function may be
   instance level or class level [Riehle2000b]. Garza1988 states that when class level
   methods are called, instead of setting individual locks on all objects, a single lock
   on its class may be set to minimize the lock escalation.
2. When a transaction requests a sub class object (figure 2.3), the sub class object and its corresponding base class object mapping to the same record in a database table must also be locked to maintain consistency. Hence base class object is an implicit resource needed for a transaction, when a transaction makes explicit resource request to sub class object. However when base class objects are requested, sub class objects need not be locked.

![Figure 2.3 Locking the sub class object with its base class object to maintain consistency](image)

3. When a transaction requests a composite object, its component objects also need to be locked. In aggregation, component objects constitute composite object. Hence component objects are implicit resources to composite object (explicit resource). The composite object gets the request and forwards it to component object, if the service is implemented in component object. The component object provides the service to the transaction as in figure 2.4.

![Figure 2.4 Locking the composite object with its component object to maintain consistency](image)
4. In association, when a transaction calls an associative object, it may access associated object to provide the service. Then associated object needs to be locked along with the requested object to maintain consistency as in figure 2.5.

![Diagram of locking the associative object with its associated object to maintain consistency](image)

**Figure 2.5** Locking the associative object with its associated object to maintain consistency

Association differs from Inheritance and Aggregation relationships in the following ways:

- Association requires several qualifying attributes to completely define itself, unlike “IS-A” and “HAS-A” relationships that are complete and semantically strong.
- In Inheritance and Aggregation, the cardinality of the relationship is usually 1. But in association; the cardinality can range from 0 to many. Hence a policy must be decided to fix the granule size.
- Reflexive association is present only in association, in which one object may associate with 0 or more objects of the same class. This leads to self looping.
- Usually inheritance and aggregation are static. These relationships are decided at design time. But association can be static or dynamic.

Henderson1997 has classified the association in the following categories:

1. Direct vs. Indirect Association:

   In direct association, the two classes are directly linked. This will be usually binary association.
In the above example, the association between A, B and B, C are direct. But the association between A and C is indirect. This implies that if class A is requested, then B is also to be requested. This is because B is directly associated with A and A might need the services of B. But B is associated with C. This implies that B might use the services of C to serve A. Hence A is indirectly associated with C. When B is locked along with A, C also needs to be locked. This association type decides the extent of locking.

2. Binary Vs N-ary Association:

Binary association is association between two classes. If more than two classes are associated, then it is called N-ary association. N-ary association is difficult to implement as it is. Hence it is implemented as a collection of binary associations.

<table>
<thead>
<tr>
<th>N-ary Association</th>
<th>Binary Association</th>
</tr>
</thead>
</table>

for example

Subject

Teacher ———— Student

Subject ———— Teacher

Teacher ———— Student

Student ———— Class

3. Referential Vs Dependent Association:

In referential or independent association, the association is logical. The associated classes are called as target and source classes. Target class is connected to source class which provides service. This typically defines “USING” relationship. When source classes are removed, the target classes are not removed. They are independent of each other.

Alternately, dependent association is physical. Here the classes are called producer and client. If producer is removed, the client also ceases to exist. In other words, client depends on server for its existence. This imposes constraints on creation and deletion of client on producer.

4. Shared vs. Exclusive Association:

In this type, the association is either dedicated to one class or shared with many classes.
5. Static vs. Dynamic Association:

Stevens2002 states that association can have static or permanent links (long term association) or dynamic links (short term association). Static links are defined at design time. But Dynamic links are transient, contextual and initiated only on request. Hence request for dynamically associated classes are deferred till runtime.

6. Reflexive Association:

This is a rarity in association itself. An object can be a client of other objects in the class.

Example.

A supervisor, who is also an instance of employee, manages other employees. This is called as self looping.

7. Inherited Association:

Example.

The association between subject and student is inherited to the derived class PG Student also. This lets redefinition of the association between student and subject.

Any association is expected to define the following attributes to be semantically complete.

1. Role name: Two classes may have more than one association. This helps to select a specific association at a time between the two associated classes. This helps to deduce what attributes are going to be accessed for a particular association. Then concurrency may be increased.

2. Interface specifier: Along with role name, this also helps to identify attributes required, the services (methods) provided in a specific association.

3. Visibility: Specifies the access rights to other attributes and methods in the class. A transaction in OODS is typically constituted of interfaces. An interface may contain one or more methods or member functions of the implementing class. Then it is
required that these methods are declared as ‘public’. Otherwise they are hidden from the client and their request will not be satisfied.

4. Cardinality/ Multiplicity: Cardinality specifies the correspondence between the associated classes. This can be used to deduce granule size.

The above mentioned factors can be utilized while defining lock model for objects related by association. So far, the association relationship is not considered because of its inability to completely define the relationship semantically.

2.2.1 Classification of design time operations

In OODBMS, schema or the class diagram is viewed as directed acyclic graph. The classes are viewed as nodes and the relationship links connecting classes are viewed as edges. In [Kim1990; Bannerjee1987], the design time transactions altering the schema are classified into changes to class definition and changes to the class hierarchy structure. The changes to class definition can be

1. Modifying the definition of attributes defined in the class such as changing its name and domain,
2. Adding/ deleting attribute
3. Adding/ deleting method
4. Modifying interface (signature) or implementation of a method
5. Creating/ deleting instance
6. Moving an attribute from one class to another class
7. Moving a method from one class to another class

In Bannerjee1987, the changes to class hierarchy are classified into changes to the nodes and changes to the links. Changes to the node involve

1. Adding a new class
2. Dropping an existing class
3. Changing the name of a class
4. Moving a class from one position in class hierarchy to another position

Changing an edge or link means changing the relationships between any two classes in the class diagram. This includes

1. Making a class as parent class to a subclass
2. Removing a class from the list of parents of a class and
3. Changing the order of parent classes of a class.
Hence changes to the link actually involve changing the relationship between classes and changing the position of a class in the class diagram. Then it is obvious that changes to class level requires locking at class level and changes to class hierarchy structure requires locking at class hierarchy level. Though in Bannerjee1987, links refer to class hierarchy level, it can be rephrased as class lattice level as links do not refer to inheritance alone but also other relationships like aggregation and association. Any concurrency control scheme is expected to provide support for all the design time operations mentioned above.

2.3 Existing Semantic Multi-Granular Lock Models

Figure 2.6 shows the classification of existing concurrency control techniques. Concurrency control techniques are broadly classified as Timestamp ordering, Locking and Optimistic Concurrency Control techniques. Locking is widely favored for ease in implementation. Multi-Granular Lock Model (MGLM) is a popular model of Locking. In the literature, it has been proven that the application of object oriented concepts in determining granularity improves the performance. These existing semantic multi-granular lock models for OODBMS provide commutativity among transactions in two ways: based on relationships and based on commutativity.

One group of works proposes concurrency control based on the compatibility of relationships namely inheritance, aggregation and association between the objects. Separate lock modes are defined for each of the above relationships. In the second
group of concurrency control schemes, compatibility is defined based on the commutativity of operations. They require application programmers to perform semantic analysis on the source code of methods (member functions) of the class. These semantic multi-granular lock models can be assessed based on the level of concurrency they provide for parallel execution of design time and runtime transactions without compromising on consistency. These two types of transactions induce three different types of conflicts among transactions to a class lattice: conflicts among runtime transactions, conflicts among design time transactions and conflicts between runtime and design time transactions. Algorithms addressing these types of access conflicts are discussed below.

2.3.1 Conflicts among runtime transactions

Gray1976 has first introduced MGLM for relational databases. Intension locks are used to infer the presence of locked resources at smaller granule level. The lock modes defined here are S (Shared - Read), X (eXclusive – Write) and SIX (Shared Intension eXclusive – locks all in S mode but a few of them in X mode). IS and IX intension lock modes are to be set at coarse level before locking resources using S, X and SIX lock modes at fine granule level.

MGLM was first extended to object oriented databases by Garza1988 for ORION. In this paper, MGLM is defined for objects related by inheritance and exclusive aggregation only. They have applied the lock modes defined by Gray1976 for OODBMS. The locks defined in Garza1988 are of granularities of classes (collection of instances) and instances. Later Kim1989 has extended it to all types of aggregation (namely shared and exclusive aggregation, dependent and independent aggregation). In this paper, apart from the lock modes in Garza1988, new lock modes like ISOS, IXOS, SIXOS are added to support shared aggregation. In Jun1998, concurrency control for runtime transactions on classes related by inheritance is proposed. The smallest granule in all these schemes for runtime transactions is only up to instance level and all of them have proposed MGLM based on relationships only. In Saha2009, a self adjusting MGLM is defined to let the transactions to dynamically choose their granularity from coarse to finer size on a particular resource based on the increasing degree of resource contention.

As mentioned earlier, the schema in OODBMS is represented using class diagrams. In class diagrams, the class relationships namely inheritance, aggregation
and association exist in different combinations. These concurrency control schemes define lock modes for each relationship separately. They have not defined lock modes for objects which have combination of relationships. Hence they are not suitable for representing complex data models. Further, their granularity is restricted to instance level.

In the lock models based on commutativity of operations, Badrinath1988 initiated this by defining commutativity based on the operations defined in class methods with the objective of defining the granularity less than object level. In Badrinath1988, attribute is the smallest granularity supported. They state that any two methods in a class can be parallely executed if they do not share any attribute. This provides a granularity smaller than object. But it requires knowledge of the structure of all methods in a class. In Badrinath1992, the idea of recoverability is defined. i.e., the methods can be executed in any order. But the commit order is fixed. This also requires apriori knowledge of all possible outcomes of all methods. In Agrawal1992, the idea of Right Backward (RB) commutativity is introduced. It states that “an operation o1 is said to have RB commutativity with another operation o2 on an object if for every state in which executing o2 followed by o1 has the same state and result as executing o1 followed by o2”. This is less restrictive than commutativity relationship, as it is included in commutativity. However application programmers need to know all possible results of each method.

Malta1993 proposed commutativity of methods to resolve lock conflicts between runtime transactions. In Malta1993, the lock modes are defined independent of object relationships. This paper has claimed to eliminate the burden of determining commutativity exhaustively for every pair of methods at runtime, by determining it apriori using Direct Access Vectors (DAV). It is based on the idea of Badrinath1988. A DAV is a vector defined for every method, whose field corresponds to each attribute defined in the class on which the method operates. Each value composing this vector denotes the most restrictive access mode used by the method when accessing the corresponding field. The access mode of any attribute can be one of the three values, N(null), R(read), W(write) with N < R < W for their restrictiveness. The access vectors are defined for all methods based on their lock mode on every attribute defined in the class. Commutativity is based on access modes. If access modes are compatible, then DAVs of corresponding methods commute. If the DAVs commute, then the methods commute. This involves two steps. 1. DAV is constructed for each
method. 2. The commutativity table of methods is constructed. Then final DAV for all the methods specifies the most restrictive access of all the attributes in a class. This paper has claimed to reduce locking overhead, lock escalation and deadlocks. Since the most restrictive lock mode is decided in the beginning itself, lock overheads due to lock conversions are reduced, and hence deadlock is minimized. Moreover, this paper has extended concurrency up to attribute level.

In Jun2000, fine granularity of runtime transactions is provided to the attribute level using DAV. For every attribute, the methods that are using this attribute are considered. From the method implementation, it is inferred whether the method reads or writes the attribute value. The granularity is assessed to the level of break points. This provides finer granularity smaller than attribute level. It is to be noted that the attributes are not only used in the classes where they are defined but also in other classes that are related to the defined class by inheritance, aggregation and association. These related classes are called as adapted classes. Then while constructing DAV, the DAV of methods in adapted classes also should be considered along with the defined class methods. In aggregation and association, the method implementations in defined class are used as they are in adapted classes. Therefore, new DAV is not required for classes related by aggregation and association.

However in inheritance, the methods inherited from base class to subclasses are classified into two types namely template methods and hook methods as in Riehle2000a. Template methods are adapted as they are from the base class. I.e. both the interface and implementation are same in both base class and subclasses. This means that implementation inheritance is followed for template methods. In hook methods however only interface or signature is inherited. This supports method overriding. The base class and subclasses are allowed to have separate implementations for this interface. This is called as interface inheritance. Both template methods and hook methods of subclasses can access the attributes of the base class. Then commutativity table for the base class should include final DAV of hook methods in subclasses as they can also access attributes in base class but may be in different lock mode. Thus these mechanisms fail either in providing fine granularity or in consistency.

Now let us analyze the concurrency control strategies followed by some of the popular object oriented databases. In ZODB [Fulton1996, Fulton1999], concurrency control of runtime requests is based on timestamps. The optimistic time-stamp
protocol used by the ZODB is well suited to design environments and other environments where there are complex data structures and in which reads are far more common than writes.

Versant [Versant2008] by default uses a pessimistic locking strategy to ensure that objects in the database server are in sync with client access in an ACID way. This is done by using a combination of locks against both schema and instance objects. In brief, the database server process maintains lock request queues at the object level to control concurrency of access to the same object.

ObjectStore [Objectstore2011] uses locks to isolate transactions from the effects of other transactions acting on the same data. ObjectStore uses strict two-phase locking for controlling concurrent access. Lock contention occurs when a client attempts to access persistent data that is incompatibly locked by another client. For example, when a client attempts to read data that another client has already locked for writing. The effect of lock contention is that one client is blocked and must wait to acquire its lock until the other client releases its lock. Lock contention has no effect on the correctness of the operations involved in lock contention, nor does it in any way compromise data integrity. But it does impact the performance of the blocked clients.

### 2.3.2 Conflicts among design time transactions

In this section, conflicts among the transactions requesting the design time operations as in section 2.2.1 are addressed. In Lee1996 all the schema operations are supported by locking the entire schema with Read Schema (RS) and Write Schema (WS) lock modes. In Malta1993, lock mode for changing the class definition (class contents) is provided by RD (Read Definition) and MD (Modify Definition) lock modes. They have overlooked the other types of schema changes. Agrawal1992 provided finer granularity by defining separate lock modes for attributes and methods (which are class contents). They have not defined any separate lock mode for operations involving changes to nodes and edges. In Lee1996 and Malta1993, transactions modifying class relationships are serialized and no other runtime transactions or design time transactions are allowed to execute parallely. i.e., the entire class diagram is locked and indirectly the entire database is locked. Therefore, there is no separate lock mode defined in the literature to read or modify class relationships as defined in Bannerjee1987.
In Jun2000, class definition has been divided into three compartments namely
1. Reading and Modifying Attributes (RA, MA),
2. Reading and Modifying Methods (RM, MM) and
3. Reading and Modifying Class Relationships (RCR, MCR).

Lock mode for attributes involves changing the domain of the attribute, or deleting the attribute. In Jun2000, there is only one lock mode shared by all the attributes of a class. At any time, only one attribute can be modified in a class. This lock mode considers the access conflicts within the class only. It does not consider the conflicts arising due to the relationship of this class with other classes.

In Jun2000, there is only one lock mode for all the methods defined in a class. At any time, only one method can be modified in a class. This lock mode considers the access conflicts of methods within the class only. It does not consider the access conflicts arising due to the inheritance, association and aggregation relationships of this class with other classes.

In Jun2000, operations involving change in class relationship are serialized. It does not take into account the structural modifications between classes. I.e., it considers only intra class relationships and excludes inter class relationships. Then it can be observed that all the class level and class lattice level operations represented by MCR (Modify Class Relationship) lock mode blocks all the other design time transactions along with runtime transactions. So in all these existing works, the granularity of design time operations is still coarse. The transactions that modify class relationships are serialized.

In ZODB [Fu1ton1996], the design time transactions are handled in 4 ways. Changes in object methods are easily accommodated because classes are not stored in the object database. Changes to class implementation are reflected in its instances, the next time an application is executed. Adding attributes to instances is straightforward if a default value can be provided in a class definition. More complex data structure changes must be handled in __setstate__ methods. A __setstate__ method can check for old state structures and convert them to new structures when an object's state is loaded from the database.

In Versant [Versant2008], schema evolution is supported by lazy evaluation. It supports versioning of schema. In existing schema, it generates errors and allows the user to choose the version. In either case, the runtime transactions are temporarily suspended.
ObjectStore [Objectstore2011] by default uses \textit{batch mode} while installing schema in a user database. All schema data in the application schema database is added to the user database when the application first accesses the database typically, when the application creates the database. Thereafter, no schema needs to be added to the database unless the application has been changed to access persistent objects of a new type.

As an alternative to batch mode, you can specify \textit{incremental mode}. When ObjectStore uses this mode, it installs schema for a particular type only when an application first allocates persistent storage for an object of the type. Incremental mode has two advantages:

- It spreads the cost of schema installation over the lifetime of the database.
- It installs only schema for types that are allocated in the database, thus reducing the size of the schema data in the database.

Alternately, incremental mode can increase the chances of lock contention because it spreads schema installation across the lifetime of the application, rather than confining it to one time. When batch mode is in use, lock contention during schema installation can occur only when a process first accesses a database.

\textbf{2.3.3 Conflicts between runtime transactions and design time transactions}

In runtime transactions, the values of attributes are read or modified by executing the associated methods in a class. The attribute values are locked in read and write lock modes. In design time transactions, the attribute definitions are read or modified. Thus an attribute has two facets and is chosen depending on the type of transaction.

During runtime transactions, the methods are locked in read mode as their contents are not modified by execution. In design time transactions, the method definitions are read or modified. When any attribute or method definition is modified, runtime transactions accessing them should not be allowed.

In Garza1988, S (Shared) and X (eXclusive) lock modes are defined for reading and modifying class definition respectively. In this, an entire class object is taken for lock granularity. Since X mode is not compatible with all other lock modes, a class definition modification blocks all other accesses to the same class. Moreover, the same S and X lock modes are used for runtime transactions also. This scheme provides limited concurrency since a class definition read does not commute with any runtime transaction.
Actually, a class definition read commutes with an instance write as described in Cart1990. In Cart1990, only two lock modes are used for an entire class object: CR (Class Definition Read) and CW (Class Definition Write), respectively. Since CW conflicts with CR and any other runtime lock modes, concurrency between class definition accesses (class definition read and class definition write) and runtime accesses is limited. As discussed earlier, two lock modes on a class object limits concurrency between class definition write and instance access since higher concurrency is possible by taking finer locking granularity in both class objects and instance objects.

In Malta1993, MD blocks any other instance access as well as RD and MD, since MD lock does not commute with any other lock modes. In Servio1990, an exclusive lock is required for a modify class definition. It guarantees that other transactions cannot acquire any kind of lock on the object since an exclusive lock on a class does not commute with any other lock requesting transactions. This results in severe concurrency degradation. Similarly, Lee1996 offer two lock modes on a class object: Read Schema (RS) and Write Schema (WS). Since WS lock is not compatible with any other lock modes, concurrency between a class definition access and an instance access is limited.

A limited concurrency between class definition write and instance access is provided in Agrawal1992 as follows. Lock granularity as individual attributes and individual methods instead of an entire class object is adopted. That is, as long as two class definition access methods or instance access methods access disjoint portions of a class definition, they can run concurrently. These fine granularity locks are required each time an instance access method is invoked so that their scheme incurs large overhead.

In Olsen1995, an instance write method can run concurrently with a class definition write method on the same class. This concurrency is based on the following argument: "the instance update operation is given a copy of old class definition that is publicly available. Once a class definition is updated, it becomes publicly available and all new instances use it. After all instance update operations that used an old class definition have either aborted or completed, the new class definition is applied to all instances of that class". Although they allow concurrency between instance access and class definition access, their lock granularity is still too big because an entire instance object is taken.
In Jun2000 though the granularity is at attribute level, as it provides coarse granularity for operations handling class relationships, the granularity is not always fine. However, AAV (Attribute Access Vector) is defined for all the attributes in every class to maintain their lock status. Using this, simultaneous access to more than one attribute is facilitated. This paper offers a trade off between limited concurrency of accessing only one attribute at a time against maintenance overhead of AAV for concurrent access of all attributes of a class. Similarly, MAV (Method Access Vector) is defined for all methods in the domain to maintain their lock status. Using this, simultaneous access to all methods is facilitated. It offers a trade off between limited concurrency of accessing only one method at a time against maintenance overhead of MAV for all methods of every class. If separate lock modes can be defined for node changes and link changes, then concurrency can be enhanced.

2.4 Deadlock Handling Techniques in OODS

Application of concurrency control techniques may lead to deadlocks. It has been shown that application of semantics of object oriented concepts on concurrency control techniques improves concurrency. So it can be experimented to see whether it also works for deadlock prevention algorithms also. In the next section, the adaptability of deadlock prevention algorithms of distributed systems to OODS is explored. In section 2.4.2, the short comings of the popular distributed deadlock detection algorithm proposed by Chandy1983 are analyzed. In section 2.4.3, the existing victim selection algorithms are compared against their performance towards the desirable factors like throughput, response time, fairness, resource utilization.

2.4.1 Existing deadlock prevention algorithms

In Object-Oriented Distributed Systems (OODS), objects are the resources and they can be acquired using locks. The resources are multi–granular in nature and the hierarchy can be as given in figure 2.7. Class diagram is the structural diagram giving static view of the system. It gives details of all the objects participating in the domain, their attributes, member functions and their relationships with other objects in the system. It consists of a set of transactions T and a set of resources R. OODS requests support AND model [Hac1989]. A resource request in AND model system is of the form \( r_1 \cap r_2 \cap \ldots \cap r_n \) where \( r_i \in R \). This means that a transaction can execute only when it gets all the resources.
Let $T_1, T_2, \ldots, T_n \in T$. $T$ maintains a list of transactions that are currently executed in the system. Once a transaction has finished execution, it is removed from the list. In single request model, when a transaction enters into the system it requests for the resources one by one. A transaction can request for the next resource, only when the previous resource is granted to it. All the resources that it needs are maintained in the REQUEST$_{T_i}$ list. All the resources granted to it are maintained in ALLOCATED$_{T_i}$ list. Any request that is granted will be removed from REQUEST$_{T_i}$ and added to ALLOCATED$_{T_i}$. Once the execution of $T_i$ is over, ALLOCATED$_{T_i}$ is made empty. There is another list called FREE that holds all the resources that are available and not granted to any of the transactions. When any resource request is granted, it is removed from FREE list and attached to the ALLOCATED list of the transaction that requested the resource.

Let us assume that there exists only one unit of every resource and each resource type is unique. If there exist no alternatives for any of the resources and if resource request model is AND model, then Holt1972 says that the necessary and sufficient condition for deadlock is the presence of cycle in wait for graph. Shaw1974 and Coffman1971 have shown that by resource ordering, deadlocks can be prevented in single resource model.

In OODS, the resources specified in class diagram are distributed to various sites. Since it is a distributed system, objects and associated database fragments are to be partitioned and distributed to various sites. Several partitioning algorithms [Ozsu1999] like horizontal partitioning, vertical partitioning, path partitioning etc

![Hierarchy of lock granules in OODS](image-url)
have been proposed in the literature. Horizontal partitioning is simplest of all the algorithms. The other reason for choosing horizontal partitioning is to group closer resource IDs and isolate transactions with similar requests. The grouping of transactions will reduce deadlock-handling time. Figure 2.8 shows a sample class diagram after horizontal partitioning.

Here it can be observed that each level of classes is assigned to a different site.

![Sample class diagram with horizontal partitioning](image)

Several deadlock prevention algorithms have been defined for distributed systems and distributed object oriented systems.

Andrews1982 has proposed Deadlock Prevention Algorithm (DPA) for predicting hardware resource requirements and preventing deadlocks at runtime. Reddy1993 has proposed DPA for distributed database system, which is claimed to provide deadlock freedom at low message cost. It eliminates the deadlock by giving higher priority to active transactions. If all the conflicting transactions are active, the transaction having higher Transaction Identification Number (TIN - which is a triple field value (S, I, C) where S is the site ID, I is the unique transaction ID and C is the transaction arrival time in local clock) is given priority. Thus, it prevents cycles in wait- for-graph. However, it does not consider the case where conflicting transactions require multiple resources and latter transactions already have more resources than earlier transactions.

Davidson1993 have proposed AND-OR DPA for concurrent real time systems using resource ordering technique. Here the ordering is done for passive data resources and their associated active resources (like processors). In this DPA, the interdependency or relationship of data resources among themselves is not addressed.
Hence, this cannot be considered for objects in OODS that is related to other objects in many ways.

Cummins2001 recursively checks for presence of cycle of any size, whenever simultaneous transactions in distributed object system request for object resources. It does not exploit the structure of object oriented system defined using class diagrams and does not utilize the object relationships to infer the required resources apriori.

Lewis2008 has proposed DPA for multi threaded environment. A transaction i.e. thread in this case, should set the deadlock prevention mode indicator for every data resource as shared or exclusive. If the mode is exclusive, then access is serialized. Here also, deadlock is prevented by access ordering and effect of mutual dependency of resources is not addressed.

Anand2009 have proposed DPA for distributed environment. Deadlock prevention is achieved here by preempting threads (which are the resources) assigned to transactions. Here each transaction having complex nested calls request for multiple threads for their execution. They request the same thread to execute a method. Then conflicting transactions having one thread and requesting another may lead to deadlock. Then lower priority transaction is made to preempt.

*From the literature survey, it can be inferred that very few DPA have been proposed for OODS. Majority of the algorithms are generic. The algorithms that have been proposed for OODS does not exploit the semantics of object oriented paradigm. None of them propose any resource ordering technique using it.*

### 2.4.2 Fault tolerance in distributed deadlock detection algorithms

Deadlock detection is an optimistic approach for handling deadlocks. Probe based deadlock detection algorithm [Chandy1983] is one of the most popular algorithm because of its simplest approach to detect deadlocks. However it has the drawbacks of lack of fault tolerance and requirement of separate deadlock resolution phase.

In order to provide fault tolerant deadlock detection in distributed systems, Li1993 has proposed a totally distributed fault tolerant DDDR algorithm using fault diagnosis model. In this, the processors are categorized into faulty and non faulty. All non faulty processors will certify the other processors as faulty or non faulty. A fault vector is attached as part of the probe where each bit in the vector represents a processor in the system. 0 represents non faulty and 1 represents faulty. It has the following drawbacks: The processors are diagnosed periodically by the other non
faulty processors. If the period is very small, the non-faulty processors need to spend more time in diagnosing other processors than executing its transactions. This will reduce throughput of the system. On the other hand, if the periodicity is more, then reliability reduces. Hence the success of this algorithm lies in choosing ideal period of diagnosis. Fault diagnosis is not a function of deadlock detection. However fault information needs to be given. Message complexity is more in propagating updated processors’ status and clean messages. It can identify only one processor failure per deadlock cycle.

Apart from Li1993, very few works have been proposed on fault tolerant DDDR algorithms. Hansdah2002 discusses about link failure, where grant messages are lost or delayed. Brzezinski1995 offers solution for asynchronous messaging system, where the messages are not delivered in FIFO basis. It proposes a token based system to handle this. However this algorithm also assumes that there are no site failures.

2.4.3 Existing victim selection algorithms

Detection of deadlocks is followed by its resolution. In distributed systems, detection of deadlock is more difficult than in centralized systems. This is because the resources are distributed in different sites and transactions access them from any of these sites. They communicate through messages only. Hence in order to know the wait-for status of the transactions, a Global WFG has to be constructed. Selection of a victim using this GWFG is complex. Zobel1988, Newton1979 and Singhal1989 have done survey on various deadlock handling techniques, but they have not focused on victim selection algorithms for deadlock resolution. Moon1997 have compared the performance of deadlock handling techniques against the attribute of throughput alone.

The circular wait state can be broken by aborting one of the transactions participating in the cycle. The transaction chosen for abortion is called as victim. Several algorithms have been proposed in the literature for the selection of victim under different criteria as given below:

   Transaction Attribute: Arrival time or Age
   Transaction that has arrived latest or whose time stamp is greater than all the participating transactions is chosen as the victim. This assumes that the later transaction would not have done much progress and hence aborts the latest
transaction. It is highly fair and provides linear response time as it serves in FIFO basis.

2. **Selection Criteria: Minimum History [Agarwal1987]**
   
   **Transaction Attribute: History**
   
   The transaction that has been aborted least number of times so far (also called as history) will be chosen as the victim. This ensures elimination of starvation.

3. **Selection Criteria: Least Priority [Sinha1985]**
   
   **Transaction Attribute: Static Priority**
   
   The transaction having the least static priority will be aborted. This helps to decide the order of execution, given a collection of transactions. The priority of the transactions can be statically fixed by the users or the domain.

4. **Selection Criteria: Maximum Size[Weikum2005]**
   
   **Transaction Attribute: Size**
   
   The transaction, whose code size is largest among all the active transactions, will be aborted. As the transaction size increase, it is assumed to consume more resources and finish execution much later. Hence transaction with largest size is chosen as victim. This improves the throughput of the system as more number of smaller transactions is finished in the given time.

5. **Selection Criteria: Minimum number of locks [Agarwal1987]**
   
   **Transaction Attribute: In-degree in Wait for Graph**
   
   Transaction that has acquired least number of resources so far, inferred by the least number of grant messages and represented by in degree in WFG is chosen as victim. The transaction is chosen only based on its current resource holding status and hence may improve the throughput of the system. The resource utilization improves, because of the selection of a victim which has locked minimum number of resources in the system so far, and does not penalize a transaction on any other criteria.

6. **Selection Criteria: Maximum number of cycles [Chow1991]**
   
   **Transaction Attribute: Cycle participation**
   
   Transaction involved in maximum number of deadlock cycles will be aborted. Normally, it is expected to choose one victim per cycle. By using this algorithm, the number of victims may be reduced. Hence number of transactions rolled back is lesser and hence throughput increases.

7. **Selection Criteria: Maximum Edge cycle [Chow1991]**
   
   **Transaction Attribute: in-degree+ out -degree**
This resolution is based on maximum participation of a transaction in a number of cycles. The possibility could be that the transaction is already holding high priority resources and further requires more number of resources held by other transactions. Hence there is more number of edges in the GWFG. It is not only based on transaction attribute, but also based on resource attribute. It is the sum of request edges represented by out-degree and grant edges represented by in-degree in the GWFG.

   Transaction Attribute: Random blocker, current blocker
   The transaction that has caused the deadlock is aborted in this algorithm. The overhead of selecting a victim is nil in this case. It also reduces deadlock resolution latency. But other attributes are suboptimal.

   Transaction Attribute: Resource consumed
   Transaction that has consumed least amount of resources is chosen as the victim.

10. Selection Criteria: Initiator [Agarwal1987]
    Transaction Attribute: transaction that has initiated the deadlock detection
    The transaction, which had initiated the deadlock detection phase on time out, is chosen as victim. This minimizes the deadlock resolution latency in a distributed system, as initiator ID is always communicated to all sites.

11. Selection Criteria: Maximum release set [Terekov1999]
    Transaction Attribute: holding maximum number of resources
    Transaction holding more number of resources which, when aborted will benefit maximum number of transactions, is chosen as victim.

12. Selection Criteria: Minimum number of submitted operations [Holt1972]
    Transaction Attribute: Number of submitted operations
    The transaction which has done minimum work so far is chosen as the victim.

    Transaction Attribute: low priority + minimum work done
    This algorithm works in three phases. In the first phase a set of low priority victims are selected. In phase two, victims holding higher priority resources from the first phase list are chosen. In phase three, victim which has done least work done is aborted from phase two list.

Transaction Attribute: Age and work done

Abortion cost is a function of number of currently submitted operations and transaction age and given as, Abortion cost = $\propto N(T) + \beta t(T)$, where $\propto + \beta = 1$ and $N(t) =$ number of currently submitted operations, $t(T) =$ age of transaction. $\propto$ and $\beta$ are weights to choose between age and work done. Age improves fairness and work done improves throughput.

In all these victim selection algorithms, there is a trade off between desirable factors. One factor is achieved at the cost of another factor. Garey1979 has stated that identification of minimum number of victims at runtime is NP complete.

### 2.5 Extract of the Literature Survey

a. None of the existing semantic MGLM for OODBMS has exploited the semantics of object-oriented paradigm fully to ensure consistency of the database and maximize concurrency among the transactions. Fine granularity of access for the design time transactions is lacking. Lock modes for design time operations are implemented partially.

b. Existing semantic MGLM perform well for runtime transactions. They provide maximum concurrency for stable domains using access vectors. As the number of design time requests increase, their performance deteriorates. This is because of the increase in the search and maintenance overhead of access vectors that are used for maximizing concurrency for stable domains.

c. OODS differ from OODBMS. i.e., in OODS, client transactions are implemented in programming languages. In OODBMS, the queries are implemented in query languages. Therefore, semantic MGLM in OODBMS cannot be extended to OODS. So it is required to infer the lock modes and granularity from the code and map them to commutativity matrix defined for OODBMS.

d. Concurrency control in any AND model system invariably leads to deadlock. OODS supports AND model [Hac1989]. Therefore, deadlocks have to be handled. The OOP semantics can be exploited to provide a better deadlock handling technique.

e. Existing probe based deadlock detection algorithm [Chandy1983] is not fault tolerant. Further, it requires a separate resolution phase to select a victim transaction and abort it.
2.6 Summary

To overcome the limitations inferred from the literature survey, the research problem is defined with the following objectives:

a. Guaranteeing consistency of the data and the schema of OODBMS by defining a separate commutativity matrix for checking inter class dependencies between classes related by inheritance, aggregation and association. It is also proposed to provide fine granularity of all the design time operations defined by Bannerjee1987 using object oriented semantics.

b. Removing the overhead of maintaining and searching various access vectors in the existing SCC techniques of OODBMS to support continuously evolving domains.

c. Providing lock types and granularity for the methods in OODS from the code implementing the methods. It is also intended to extend the compatibility matrix from OODBMS for adoption in OODS.

d. Defining resource ordering and access ordering techniques for deadlock prevention in OODS by exploiting object oriented semantics to eliminate cycles, starvation in poverty and starvation in wealth.

e. Improving the existing probe based deadlock detection algorithm to be fault tolerant. It is also intended to include victim selection as part of the probe to reduce its time complexity in deadlock detection and resolution phases.