SEMANTIC CONCURRENCY CONTROL AND DEADLOCK HANDLING TECHNIQUES FOR DISTRIBUTED OBJECT ENVIRONMENTS

A THESIS

submitted to Pondicherry University in partial fulfilment of the requirements for the award of the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE AND ENGINEERING

by

V.GEETHA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
PONDICHERRY ENGINEERING COLLEGE
PUDUCHERRY – 605 014
INDIA

OCTOBER 2012

Dr. N.SREENATH, M.Tech., Ph.D (IITM)
Professor of CSE
Pondicherry Engineering College
Puducherry -605014.

CERTIFICATE

Certified that this thesis entitled "**SEMANTIC CONCURRENCY CONTROL AND DEADLOCK HANDLING TECHNIQUES FOR DISTRIBUTED OBJECT ENVIRONMENTS**" submitted for the award of the degree of **DOCTOR OF PHILOSOPHY** in **COMPUTER SCIENCE AND ENGINEERING** of the Pondicherry University, Puducherry is a record of original research work done by **Mrs. V.GEETHA** during the period of study under my supervision and that the thesis has not previously formed the basis for the award to the candidate to any Degree, Diploma, Associateship, Fellowship or other similar titles. This thesis represents independent work on the part of the candidate.

(Dr. N.SREENATH)
Supervisor

Date :
Place : Puducherry

ACKNOWLEDGEMENT

Many people have contributed to the escalation of this thesis with either their personal or technical support, for which I wish to express my humble gratefulness.

First I would like to record my gratitude to my Supervisor **Dr.N.Sreenath,** Professor of Computer Science & Engineering, for his supervision, advice and guidance from the very early stage of this research as well as encouraging and supporting me through out the work.

My sincere thanks to **Dr. V. Prithivi Raj,** Principal, Pondicherry Engineering College for his encouragement and permission to carry out the research work.

I am grateful to my Doctoral Committee members **Dr.R.Panneerselvam**, Professor, School of Management, Pondicherry University and **Dr.P.Dananjayan**, Professor of Electronics and Communication & Engineering Pondicherry Engineering College for providing valuable comments and constructive suggestions towards improving the quality of this research.

I wish to express my sincere thanks to **Dr. N. Sreenath**, Professor and Head, Department of Computer Science and Engineering and **Dr. M. Ezhilarasan**, Associate Professor and Head, Department of Information Technology for providing a peaceful working environment. I would also like to thank **Dr. K. Vivekanandan** and **Dr. S. Kanmani**, for their support in this research.

I wish to express my thanks to **Teaching** and **Supporting Staff** of both the Department of Computer Science and Engineering and Department of Information Technology for their kind cooperation.

I am indebted to my parents who sacrificed their entire life to educate me.

My gratitude is shown to my beloved husband **Mr. G.Venkatasubramanian**, who helped me for the fine execution of this thesis. He is always my inspiration and without his help, I could not have completed this thesis.

I am much grateful to my son **Mr. V. Rajeshwar Balaji** for his unlimited love and support shown during the entire research period.

Finally I am indebted to all my friends and relatives for their inspiring encouragement in carrying out this research.

V. Geetha

TABLE OF CONTENTS

Chapter	Title	Page

ABSTRACT

Nowadays distributed applications are implemented for various domains like business, engineering etc. for wide access and immediate response time. In distributed system, server tier is implemented using procedures and data store is implemented using RDBMS. This results in “Impedance Mismatch” that requires conversion of data format between procedures and databases. They do not support complex data modeling. The distributed applications for emerging domains are migrating to the distributed object environment to exploit the features of object oriented paradigm for modeling their complex domain data and their relationships and to avoid the “Impedance Mismatch” Problem. In practice, distributed object environment exist as object oriented databases and object oriented distributed systems. Here, objects are viewed as reusable data resources. Objects encapsulate attributes and methods or member functions. The data from the database tier is mapped on the attributes. The methods operate on them to serve the clients.

They support two types of transactions namely runtime transactions for data access and design time transactions to modify the schema. Due to the complexity of data and their complicated relationships, sophisticated concurrency control techniques are required to ensure the consistency of the objects. This research work aims in providing better concurrency control and deadlock handling techniques for distributed object environments.

The existing semantic multi-granular lock models for object oriented databases perform better than the conventional concurrency control techniques by exploiting the features of object oriented paradigm. However they provide coarse granular lock modes and hence concurrency is limited. In some of the existing models, concurrency is provided at the cost of consistency. In all the existing models, lock modes for all types of design time operations are not provided. To overcome these limitations, a Consistency Ensured Semantic Multi-Granular Lock model (CESGML) is proposed that ensures consistency and provides lock modes of fine granularity for data access and schema access.

The applications that are implemented using the object oriented databases tend to evolve over time to provide better service to the clients and to expand the scope of the domain. This requires frequent modifications of schema to reflect the improvements made on the domain structure. It implies that more number of design time transactions will arrive in parallel along with the runtime transactions. In order to promote concurrency, the existing semantic multi granular lock models make use of the access vectors for providing fine granularity of data access and schema access. The access vectors maintain the lock status of attribute, methods, classes and their relationships. The access vectors are to be searched and updated to support concurrent runtime transactions and design time transactions. The overhead increases linearly as the number of design time transactions increase.

Two models namely semantic multi-granular lock model using access control lists and semantic multi-granular lock model using lock rippling are proposed to eliminate the search and maintenance overhead of access vectors while providing high degree of concurrency. The semantic multi-granular lock model using lock rippling defines commutative matrix based on operations. It does not require any access vector for its execution and hence avoids the delay due to search and maintenance of access vectors. Thus it reduces the transaction response time and promotes concurrency.

The semantic multi-granular lock model using access control lists also does not require any access vectors. It splits the lock table to store the data items separately based on read or write operations. This reduces the search time and improves response time. The object semantics is used to identify the conflicting operations.

The concurrency control mechanisms of object oriented databases cannot be extended as they are to object oriented distributed systems. This is because query languages are used to access object oriented databases. But in object oriented distributed systems, programming languages like C++ and Java are used to make the client transactions. Then the lock types and the granularities of resources are to be ascertained from the client code using document tools like docC++ and Javadoc. After identifying the lock modes for all the objects used in the client code, the compatibility matrix defined in object oriented databases can be extended to object oriented distributed systems. A methodology has been proposed to map the types and properties of methods in objects into suitable lock modes and granularities of lock. Then the compatibility matrix based on object relationships from OODBMS has been modified and extended to object oriented distributed systems.

Application of concurrency control may need to deadlocks. Deadlock affects throughput of the system and increases the transaction response time. Deadlocks can be handled by prevention or detection and resolution. Object oriented distributed system basically supports AND model requests. Detection of deadlocks for AND model requests in distributed system is tedious. It is already proven that deadlock prevention algorithms perform better than deadlock detection algorithms in distributed environments. A deadlock prevention algorithm for AND model requests ensures getting all the resources before execution. It is expected to break circular wait and avoid starvation. A deadlock prevention algorithm based on resource ordering technique is proposed by exploiting the semantics of object oriented paradigm. It also proposes access ordering that eliminates starvation in poverty and starvation in wealth.

Probe based deadlock detection algorithm is a very popular algorithm for detecting deadlocks in distributed environments. In this algorithm, the initiator sends probes to detect circular wait. If the probe comes back to the initiator, it infers deadlock, otherwise it is assumed to be live lock. However it is not fault tolerant. In a faulty environment, if the probe does not come back to the initator, it does not know whether it is due to live lock or site failure. A colored probe based distributed deadlock detection algorithm is proposed that can inform the initiator the status of the probe in all possible scenarios. The status could be live lock, deadlock or system isolation due to hardware, software and network failures.

Deadlock resolution phase chooses a victim transaction after detecting a deadlock. The victim is aborted to break the circular wait and make the system active again. Extensive survey of existing resolution algorithm is done. From the survey, the impact of transaction attributes on the system parameters is inferred. A weight based victim selection algorithm is proposed to exploit the inference in choosing victim transaction based on the desirable parameters of the system. The existing probe based distributed deadlock detection and resolution algorithm requires a separate deadlock resolution phase. The probe is modified to include the victim dynamically selected using the proposed weight based victim selection algorithm. When the probe reaches the initiator of the deadlock, the victim ID will be available that may be aborted to break the circular wait.

The outcome of this research can be used for providing better concurrency and eliminating deadlocks in distributed object environments.

LIST OF FIGURES

Fig. No.	Title	Page

LIST OF TABLES

LIST OF ACRONYMS AND ABBREVIATIONS

AA	Add Attribute
AAV	Attribute Access Vector
ACID	Atomicity, Consistency, Isolation and Durability
AE	Add Entity
AI	Add Instance
AM	Add Method
BG	Bi-directed Graph
CDV	Class Dependency Vector
CESMGL	Consistency Ensured Semantic Multi-Granular Lock Model
COM	Component Object Model
DA	Delete Attribute
DAG	Directed Acyclic Graph
DDDR	Distributed Deadlock Detection and Resolution
DE	Delete Entity
DI	Delete Instance
DM	Delete Method
DPA	Deadlock Prevention Algorithm
FCFS	First Come First Served
FIFO	First In First Out
GWFG	Global Wait for Graph
IA	Instance Access
IS	Intension Shared
ISCS	Intension Shared Class Shared

ISO	Intension Shared Object
ISOS	Intension Shared Object Shared
IX	Intension Exclusive
IXCS	Intension Exclusive Class Shared
IXO	Intension Exclusive Object
IXOS	Intension Exclusive Object Shared
MA	Modify Attribute
MAV	Method Access Vector
MCD	Modify Class Definition
MCL	Modify Class Lattice
MCL	Move Class Lattice
MCR	Modify Class Relationship
MDD	Modify Domain Definition
MGML	Multi-Granular Lock Model
MMI	Modify Method Implementation
MMS	Modify Method Signature
MSCL	Modify Sub-Class Lattice
OCC	Optimistic Concurrency control
OODB	Object Oriented Databases
OODBMS	Object Oriented Database Management Systems
OODS	Object Oriented Distributed Systems
RA	Runtime Access
RA	Read Attribute
RAA	Read Adapted Attributes
RAMI	Read Adapted Method Implementation

RAMS	Read Adapted Method Signature
RCD	Read Class Definition
RCL	Read Class Lattice
RCR	Read Class Relationship
RD	Read Definition
RDBMS	Relational Database Management System
RDD	Read Domain Definition
RM	Resource Manager
RMA	Runtime Modify Access
RMI	Read Method Implementation
RMS	Read Method Signature
RRA	Runtime Read Access
RS	Read Schema
RSCL	Read Sub-Class Lattice
S	Shared lock mode
SCC	Semantic Concurrency Control
SCC	Semantic Concurrency Control
SIX	Shared Intension Exclusive
SIXCS	Shared Intension Exclusive Class Shared
SIXO	Shared Intension Exclusive Object
SIXOS	Shared Intension Exclusive Object Shared
TM	Transaction Manager
WD	Write Definition
WS	Write Schema
X	Exclusive lock mode

CHAPTER 1

INTRODUCTION

1.1 Distributed Environment

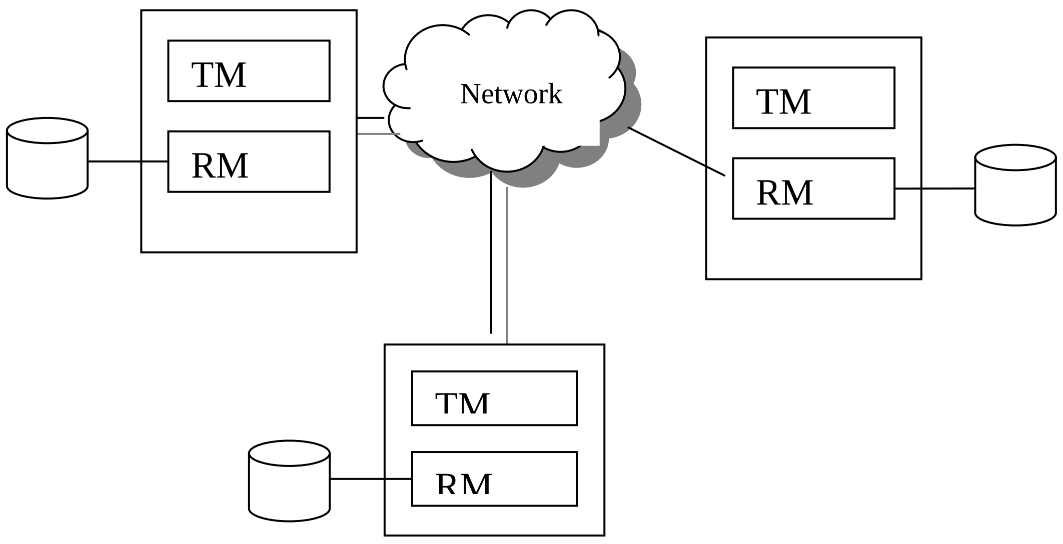
Distributed environment is a collection of loosely coupled processors interconnected by a communication network that work together collaboratively. The location of the processor is referred as a site. So typically, a distributed system is a collection of sites.

There are four major reasons for building distributed systems namely resource sharing, computation speedup, reliability and communication. In a distributed environment, resources at one site can be accessed by the local as well as remote users. The resource can be a physical resource like high speed processor, printer etc., or a logical resource like files, databases etc. Computation speedup can be achieved by shifting the jobs from overloaded site to a slack site. This idea of load sharing among the sites improves the performance of the system. Apart from the performance, reliability is also increased by providing fault tolerance. When one of the sites fails due to hardware failure or software bug, its function is automatically taken over by another site. Though this requires redundancy of data, it improves the reliability of the system. The failure of one site does not affect the functioning of other sites in the system.

Distributed system can be centrally managed or truly distributed. In a centrally managed distributed system, one of the sites is chosen as the controller. The controller site is responsible for managing and coordinating the functions of all the other sites. This poses a bottleneck threat. I.e., everything relies on this controller. If the controller site fails, the entire system fails. So this setup is not favored much.

In the truly distributed system, each site allows autonomous management. Each site is responsible for managing itself. Therefore each site is unaware of the activities happening at other sites. They communicate only through the messages. The structure of a truly distributed system is given in figure 1.1.

In the figure, each site has a local Transaction Manager (TM) whose function is to serve the clients arriving at the site. It manages the execution of the transactions that access data stored in this site. The resources needed for the transaction can be accessed by requesting the Resource Manager (RM) local to the site. It has to provide



TM- Transaction Manager RM- Resource Manager

Figure 1.1 Architecture of a truly distributed system

appropriate concurrency control techniques for the local as well as the remote transactions arriving at this site to access its resource. It is also responsible for getting access to remote resources on behalf of its local transactions.

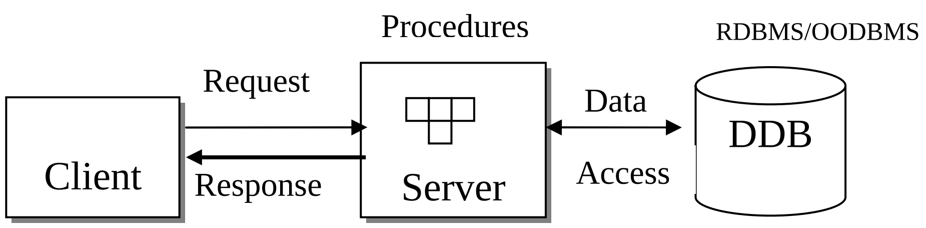


Figure 1.2 Client/server model in distributed systems

In a distributed system, the clients' transactions are served by executing procedures stored in the server. The server accesses the data resources in the form of databases like relational databases and object oriented databases etc. It is the responsibility of the data resources to ensure their data consistency. So only the full fledged database management systems can be connected at the back end. The legacy file formats cannot be supported. Figure 1.2 shows the client/server model practiced in the distributed system.

1.2 Distributed Object Environment

The earliest and most relevant applications of distributed system were in business and administration. Recently several applications have cropped up that require support for the management of complex data types. There are several application areas that are limited by the restrictions of the relational data model and procedural approach. Hence the object oriented approach is brought into the design of distributed systems. It has the features of encapsulation, abstraction, security and reusability.

The object model has structure and behavior. It encapsulates a set of attributes that contain the data and a set of methods that have a body of code to implement a service. The interface between an object and the rest of the system is defined by a set of allowed messages. The objects are interrelated by inheritance, aggregation and association relationships.

Typically objects exist in distributed systems in two forms namely Object Oriented Databases (OODBMS) and Object Oriented Distributed Systems (OODS).

1.2.1. Object oriented databases

The first choice for the applications that require support for complex data and long duration transactions is obviously object oriented databases. OODBMS is a collection of objects. The objects are classified into classes and instances. A class is a collection of instances. Objects represent complex data. Their complex relationships are defined by combinations of object relationships such as inheritance, composition (aggregation) and association. The support of complex objects imposes several requirements on both the object data model and object management. They are

- The object model should support structural modeling and interrelationships in a natural way.
- It must also support modeling of object behaviors and dynamic constraints.
- In addition in the intended application environments, the object structures, behavior and interrelationships may evolve over time [Bertino1991].

From the object management perspective, the following features are required.

- Object versioning mechanisms to support evolution of objects.
- Transactions can extend in time and involve large amounts of data. This requires the complex recovery and concurrency control mechanisms.
- Due to the evolutionary nature of applications, extension of schema modification operations should be supported without requiring system slowdown or shutdown along with the execution of data requests.
- Should provide security mechanisms.

Several advanced database products like Avance, Encore, Gemstone, Iris, O_2, Orion and Vbase have provided support for the aforementioned features.

1.2.2. Object oriented distributed systems

Object oriented distributed system is the result of merging object oriented techniques with distributed systems technology. This approach makes objects as the unit of computation and distribution. It has the ability to encapsulate both data and operations in a single computational unit. It has the following benefits:

- Systems with diversified behavior for the same service request can be provided using polymorphism.
- It can support heterogeneous environments by its ability to separate their interface from their implementation.
- It can support the continuous evolution of business domains to suit their new requirements.
- Encapsulation can hide the implementation and therefore easily change the object behavior without affecting the users much.
- There is a uniform invocation irrespective of whether it is local or remote call using message passing.

In OODS, the objects are viewed as reusable data resources. Figure 1.3 shows the client/server model of the object oriented distributed systems.

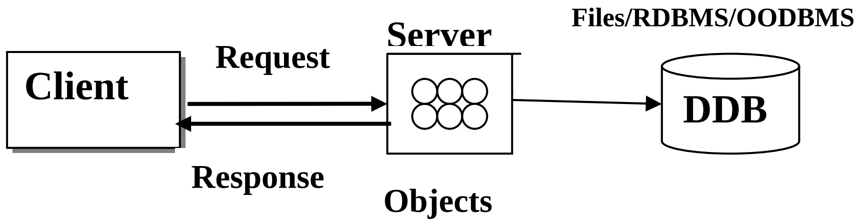


Figure 1.3 Client/server model in object oriented distributed system

In this figure, the client's request is satisfied by invoking the methods in the objects residing at server tier. So unlike distributed system which is procedure oriented, OODS is action oriented. The objects encapsulate the attributes and associated methods or member functions. The objects are related by inheritance, aggregation and association. Inheritance defines a 'parent-child relationship'. Aggregation defines 'has a' relationship. The association is used to define all other relationships such as associated-with, using etc. The domain objects and their relationships are represented as a class diagram..

1.3 Concurrency Control and Deadlock Handling Techniques

In a distributed system, typically several transactions may arrive at a server site at the same time. These parallel transactions should not affect the consistency of the data. They should not lead to dirty reads or dirty writes. The ACID property of the database has to be preserved. Ideally parallel reads are allowed. Mixed read and write operations are not allowed. Parallel write operations are also not allowed.

The business domains are continuously evolving in nature. They might want to improve their services to the clients. This requires modification of schema to reflect the changes in the business domain. Transactions would arrive to modify the schema. Then all the consistency requirements defined for data have to be extended for schema also.

Concurrency control mechanisms are applied to synchronize the transactions accessing the database to maintain data and consistency. Complex concurrency control mechanisms are needed for the object oriented system because of its complex nature. However, the concurrency control mechanisms should not affect the performance of the system. A good concurrency control mechanism should improve the throughput of the system by improving concurrency so that maximum number of transactions can run in parallel.

Timestamp Ordering, Optimistic Concurrency Control (OCC) and Locking are the commonly used concurrency control techniques. The timestamp ordering is based on the time stamps assigned to the transactions as they arrive at the system. The transactions are served in the system in First Come First Served (FCFS) order. But it is very difficult to implement time stamps in distributed systems.

OCC is a validation based protocol. Each transaction executes in two or three different phases in its lifetime, depending on whether it is a read or update operation. The phases are

1. Read phase- Reads all data items and write operations are performed on temporary local variables.
2. Validation phase- Checks the validity of updating the database with temporary variable values without any conflicts.
3. Write phase- If the validation phase is successful, the database is updated. Otherwise the transaction is rolled back.

This is a better scheme than timestamp ordering, but there is a possibility of starvation of long duration transactions.

Locking is the most common method of concurrency control. Among the various concurrency control mechanisms, locking is widely used because of its ease in implementation. It enforces the requirement of allowing a transaction to access a data item only if it holds a lock on it. In lock based concurrency control scheme, a transaction has to acquire locks before accessing the database and release them after use. Locking technique uses compatibility or commutativity matrix to decide whether a new transaction can concurrently execute with those that are already executing without affecting consistency. If the lock modes are compatible, the transactions can execute. If the lock modes are conflicting, then the transaction that is requesting the lock will be blocked. It has to wait until the transaction currently holding the resource has released the lock or preempted. There are two possible lock modes.

1. Shared (S) – To read a data item. Write is not allowed. Several transactions can use this lock mode to share a data item.
2. Exclusive (X) - Both read and write operations can be performed on the data item. But it cannot be shared.

The compatibility matrix for S and X lock modes is given in the table 1.1.

Table 1.1 Compatibility matrix for S and X lock modes

	S	X
S	Y	N
X	N	N

In the table, ‘Y’ indicates locks are compatible. ‘N’ indicates locks are incompatible. The locking technique requires a lock table along with the compatibility matrix. The lock table defines the current lock status of all the data items in the domain. The lock table has to be updated on every lock request and lock release message.

The performance of locking technique is further improved by Multi-Granular Lock Model (MGLM). It is a common technique for implementing concurrency control on transactions using the databases. Using MGLM, transactions can request the same database in different granule sizes varying from coarse granules to fine granules. This maximizes the concurrency while minimizing the number of locks. The main advantages of MGLM are high concurrency and minimal deadlocks. There are several multi-granular lock models proposed in the literature. Gray1978 has defined semantic MGLM for relational databases. In MGLM, intention locks are used to infer the presence of locked resources at a smaller granule level. The lock modes defined in Gray1978 are Intension locks –IS(Intension Shared) and IX (intension Exclusive) to lock fine granules, S (Shared - Read), X (eXclusive – Write) and SIX (Shared Intension eXclusive – locks all in S mode but a few of them to be updated alone in X mode). The compatibility matrix proposed in Gray1978 is given in table 1.2.

Table 1.2 Compatibility matrix of Gray1978's multi-granular lock model

	IS	IX	S	X	SIX
IS	Y	Y	Y	N	Y
IX	Y	Y	N	N	N
S	Y	N	Y	N	N
X	N	N	N	N	N
SIX	Y	N	N	N	N

This multi-granular lock model is extended to object oriented environments also with suitable modification by applying the semantics of the object oriented paradigm.

Though concurrency control techniques ensure consistency of data and schema, they have the negative effect of introducing deadlocks. A system is in a deadlock state if there is a set of transactions such that every transaction is waiting for another transaction in the set in a circular way. None of the transactions can make progress in such a situation. This results in poor throughput and high transaction response time which are undesirable.

In distributed systems, the dependency among transactions can be described as a directed graph called as Global Wait-For Graph (GWFG). The graph consists of a pair G = (V, E) where V is a set of vertices representing active transactions, E is a set of edges representing their dependency. When there is an edge Ti -> Tj, it indicates that Ti is waiting for Tj to release a data item that it needs. The presence of a deadlock in the system can be inferred from the presence of cycle in the GWFG.

There are two principal methods to deal deadlocks. We can use a deadlock prevention algorithm to ensure that the system will never enter a deadlock state. It is a pessimistic and proactive approach. Alternatively, we can allow the system to enter a deadlock state and then try to recover by using deadlock detection and recovery scheme. This is an optimistic and reactive approach.

There are two approaches to deadlock prevention. One approach ensures that no cyclic waits can occur by ordering the transactions or holding all the resources before execution. In resource ordering technique, an ordering is imposed on all the data items. It insists on the transactions to lock the data items in a sequence consistent with the ordering rules. This scheme is easy to implement as long as the set of data items accessed by a transaction is known, when the transaction starts execution. The other approach is to roll back, whenever the wait could potentially result in a deadlock.

Several deadlock detection and resolution algorithms are available in the literature. One of them is the popular probe based deadlock detection algorithm presented in Chandy1983**.** It is very popular because of its easy implementation. In this algorithm, when a transaction suspects deadlock, it sends a probe along the edges of GWFG. If the probe comes back to the initiator, it indicates the presence of deadlock.

When the detection algorithm detects a deadlock existence, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the cycle. Selection of a victim should be such that there is no starvation or degradation of desirable parameters such as throughput, response time and resource utilization of the system.

1.4 Motivation

Nowadays, business domains are implemented as distributed applications for reaching wider range of clients. In distributed systems, the business data and methods are scattered across various sites. Distributed databases are used to store the business data. Business methods are implemented as procedures. In order to avoid the inconsistency of the business data due to concurrent access by several clients, concurrency control techniques are applied on the databases in the database tier. The legacy data files cannot be used in the distributed systems because they are primitive and do not provide concurrency control. Further, distributed systems require separate concurrency control technique for each of the database models in which the business data are stored.

Hence, the possibility of shifting the concurrency control mechanisms from the database tier to the server tier is explored. In distributed systems, it is not possible to shift the concurrency control to server tier as it is implemented using procedures. This is possible in OODS because the client transactions to the database tier can be accessed only through the objects in server tier. This will help the OODS to support legacy file formats also. Moreover, it is sufficient to provide a common concurrency control mechanism independent of the nature of persistent storage. While exploring the requirements of concurrency control mechanisms for OODS, the concurrency control mechanisms defined for OODBMS is a good place to look for as it is the only other distributed object environment.

The three common concurrency control mechanisms such as locking, timestamp ordering and optimistic concurrency control have been dealt in the literature for adoption in OODBMS. Locking is more widely used than the other two techniques, because of its ease in implementation. MGLM- one of the types of locking, is a common technique for implementing concurrency control on transactions using the OODBMS. The other reason for the wide usage of multi granular locking is that it allows application of the semantics of object oriented paradigm to improve the performance.

Even though the concurrency control mechanisms in OODBMS can be considered, they cannot be adopted as they are in OODS. This is because query languages are used to request data from the OODBMS. But in OODS, object oriented programming languages like C++ and Java are used to make the client transactions. Then the lock types and the granularities of resources are to be ascertained from the client code using document tools. Document tool is a tool that parses the declarations and documentation comments in a set of source files and produces a set of HTML pages describing the classes, interfaces, constructors, methods, and fields.

The doc tools like docC++ and Javadoc can be used for identifying the method type and properties. After identifying the lock modes for all the classes used in the client code, the commutativity matrix defined in OODBMS can be extended in OODS.

All the business domains might eventually want to upgrade their services to clients. This evolution of business domain introduces the need to change the structure of the business domain. The domain is represented using the class diagram in OODS. Then the evolution of the business domain might require changes in the definitions of attributes, methods, classes and their relationships.

Usually, the object oriented distributed systems receive runtime transactions requesting the access of business data. Due to the continuous evolution of business domains, the systems may also receive transactions to modify the structure. These transactions are called as design time transactions. Both types of transactions can be either read or write transactions. This requires the application of concurrency control to protect the consistency of the objects. These two types of transactions may induce three different types of conflicts among the transactions to a class diagram: conflicts among run time transactions, conflicts among design time transactions and conflicts between run time and design time transactions.

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.

Then proposing a concurrency control mechanism for OODS involves two steps:

1. Define lock types and granularity for all types of methods defined in the classes that are related by relationships namely inheritance, aggregation and association and represent the business domain.
2. Propose a compatibility matrix based on class relationships that will address the conflicts mentioned above.

The existing semantic based MGLM for OODBMS can be classified as MGLM based on relationships and MGLM based on operations. Models based on relationships define lock modes for each class relationship separately. However, lock modes for the combinations of class relationships are not defined. Hence, they are not widely used.

The models based on operations have maximized the concurrency for runtime transactions by using access vectors. They have the following limitations:

1. The concurrency of runtime transactions is maximized at the cost of inconsistency of business data.
2. None of the lock models have addressed the conflicts among design time transactions and conflicts between runtime and design time transactions
3. The existing lock models have not fully exploited the semantics of attributes, methods and class relationships to maximize the concurrency of design time transactions.
4. They have also not proposed fine granularity lock modes for all types of design time operations defined in Bannerjee1987.
5. Design time transactions are executed in coarse granularity which reduces the concurrency and thus reduces the throughput also.
6. Run time transactions and design time transactions that are arriving simultaneously are blocked, even if they access different part of the class diagram. This affects the performance of the system.

So a MGLM has to be proposed which will improve the degree of concurrency for design time transactions and run time transactions by fully utilizing the semantics of object oriented features. Separate lock modes covering all types of design time operations based on Bannerjee1987 have to be proposed. It should eliminate the inconsistencies in the existing models. An enhanced compatibility matrix has to be defined to provide high parallelism between design time transactions and run time transactions.

Multi granular lock models using access vectors provide high concurrency for domains which do not alter its services frequently (stable domains). Here the design time transactions are rare. These models require access vectors in addition to the compatibility matrix and lock table. But in the case of continuously evolving domains, the schema of the domain needs to be changed frequently to match the new changes in the domain. Then, the access vectors as well as lock table should be altered every time a schema change is made. Because of this, the maintenance overhead is more than the conventional locking technique. This introduced the need for a new concurrency control scheme to support continuously evolving systems with less overhead. This algorithm should provide same or higher concurrency than the existing models with nil or less overhead.

Concurrency control mechanisms provide consistency at the cost of introducing deadlocks. OODS transactions support AND model [Hac1989]. This means that a transaction can execute only when it gets all the resources. If there exist no alternatives for any of the resources and if the resource request model is AND model, then Holt1972 says that the necessary and sufficient condition for deadlock is the presence of a cycle in GWFG. Shaw1974 and Coffman1971 have shown that by resource ordering, deadlocks can be prevented in the single resource model. Then a novel resource ordering technique is to be proposed as a common resource ordering policy cannot be adopted for all the systems. A resource ordering policy exploiting the semantics of class relationships is more suitable. To eliminate starvation, an access ordering policy based on semantics of object oriented features is also to be proposed.

Though the existing distributed deadlock detection algorithms perform well, they are not fault tolerant. They enforce a constraint that the implementation environment is free from hardware, software and network failures. So there is a need for improved fault tolerant distributed deadlock detection algorithm.

Once a deadlock is detected, a victim transaction has to be selected to break the cycle. Selection of an optimal victim that incurs minimum cost dynamically is a NP-

Complete problem [Gary1979]. So a cost based victim selection algorithm has to be proposed which lets the selection of factors based on which the victim can be chosen.

Thus the object of high concurrency, low overhead concurrency control and deadlock handling techniques are to be proposed by exploiting the object oriented semantics of distributed object environments.

1.5 Objectives of the Research Work

- To enhance the existing semantic MGLM for OODBMS by proposing fine granularity for design time operations in stable domains.
- To propose semantic MGLM for OODBMS supporting transactions from continuously evolving domains with high concurrency and low maintenance overhead.
- To propose lock types, lock granularity and compatibility matrix for all class relationships namely inheritance, composition and association for distributed object oriented system.
- To propose a deadlock prevention technique for distributed object oriented systems.
- To propose a fault-informant algorithm to send colored probes to the initiator of deadlock detection about the status of sites.
- To propose a weight based victim selection algorithm for deadlock resolution.

1.6 Organization of Chapters in the Thesis

The thesis is organized as follows.

Chapter 1 introduces the research problem by briefly describing the concepts of distributed object environment along with the challenges of the problem domain. The motivation and the objectives of the thesis are also formulated and explained.

Chapter 2 is dedicated to review the related existing works for their merits and demerits. It gives a brief classification of semantic concurrency control techniques and deadlock handling techniques in distributed object environment. The representative algorithms are identified and reported. The limitations in the existing algorithms are summarized. At the end of this chapter, the proposed research work is defined and described.

Chapter 3 describes an enhanced semantic MGLM for OODBMS supporting stable domains. The performance comparison of the proposed algorithm and the similar existing algorithms are reported in this chapter.

Chapter 4 proposes two semantic concurrency control mechanisms in OODBMS for continuously evolving domains namely semantic MGLM using lock rippling and semantic MGLM using access control lists. They are compared with the existing semantic MGLM.

Chapter 5 proposes semantic concurrency control for OODS by defining the lock types, granularity for all types of object relationships. A compatibility matrix combining all the relationships is also proposed.

Chapter 6 proposes a deadlock prevention technique based on resource ordering for OODS. The existing probe based distributed deadlock detection algorithm is enhanced to work in faulty environments also. A weight based victim selection algorithm is also proposed.

Chapter 7 concludes the thesis by highlighting the findings that facilitated to accomplish the objectives. The limitations of the research work have been identified to carryout the possible future research to make further improvement in this direction.

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

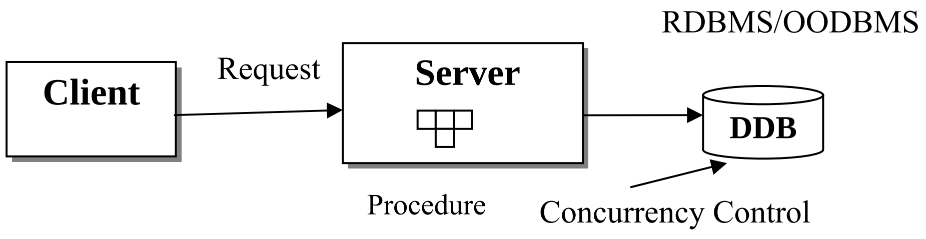


Figure 2.1 Concurrency control at database tier

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 for each type of data store, a common concurrency control mechanism can be proposed for objects.

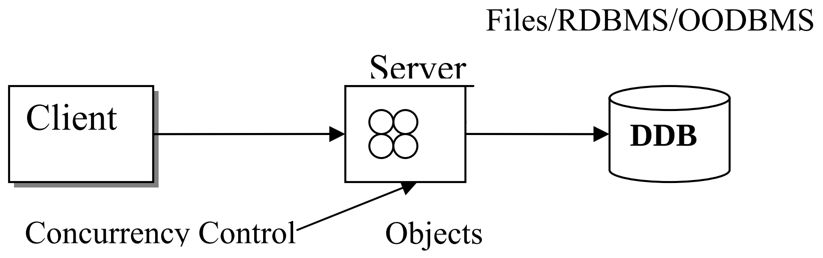


Figure 2.2 Concurrency control at server tier

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.

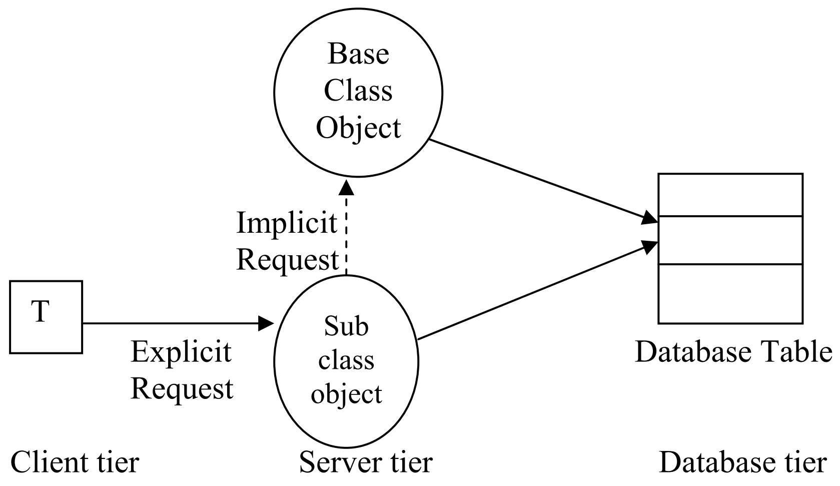


Figure 2.3 Locking the sub class object with its base class object to maintain consistency

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.

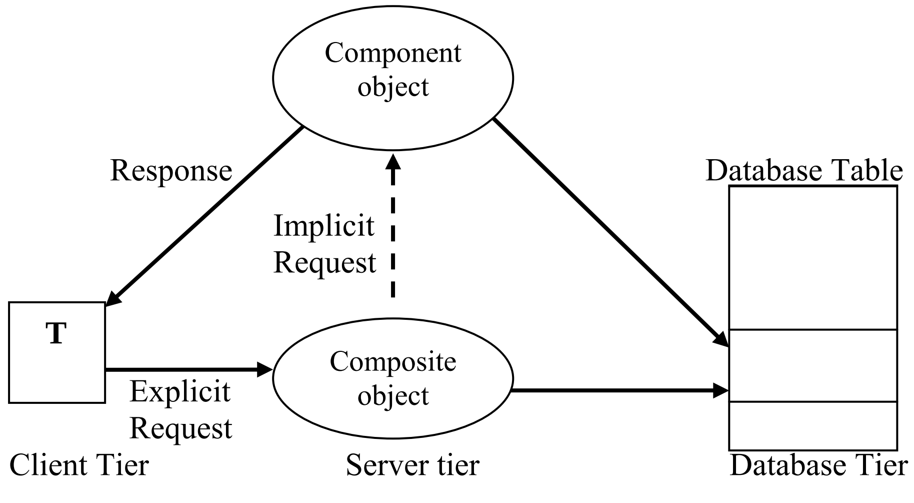


Figure 2.4 Locking the composite object with its component object to maintain consistency

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.

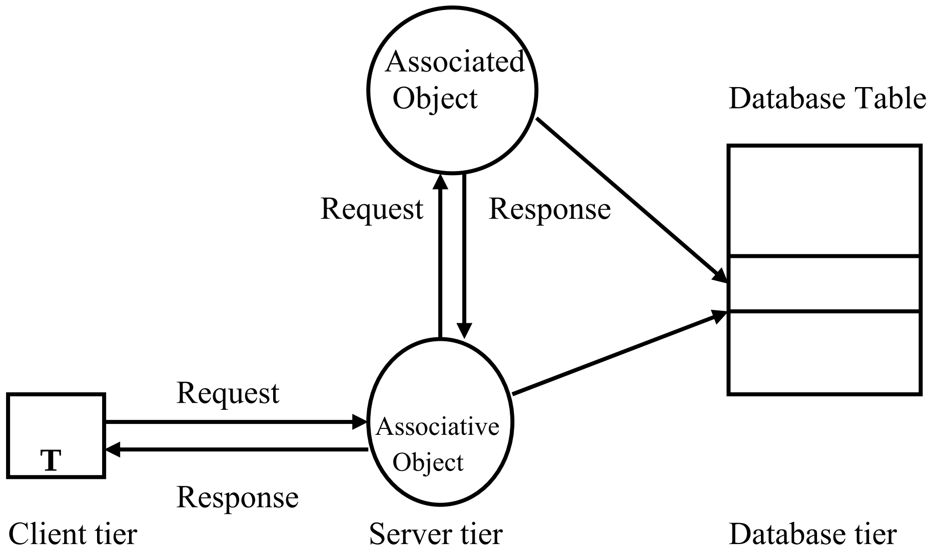


Figure 2.5 Locking the associative object with its associated object to maintain consistency

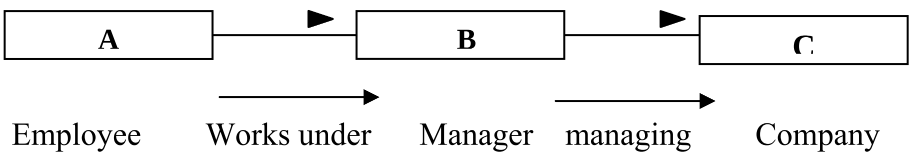
Association differs from Inheritance and Aggregation relationships in the following ways:

- 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.

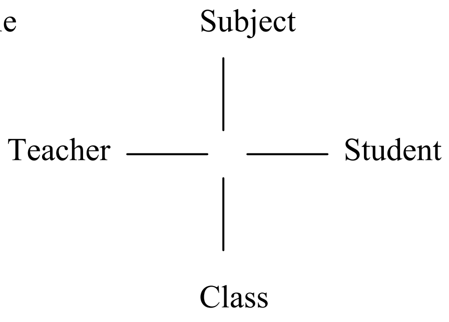


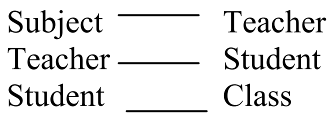
In the above example, the association between A, B and B, C are direct. But the association between A and C is indirect. This implies that if class A is requested, then B is also to be requested. This is because B is directly associated with A and A might need the services of B. But B is associated with 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.

for example





N-ary Association Binary Association

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.

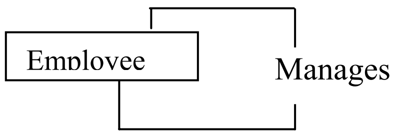
5. Static vs. Dynamic Association:

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.

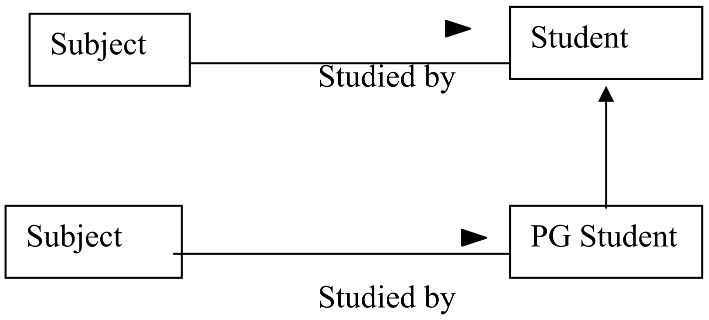
Example.



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

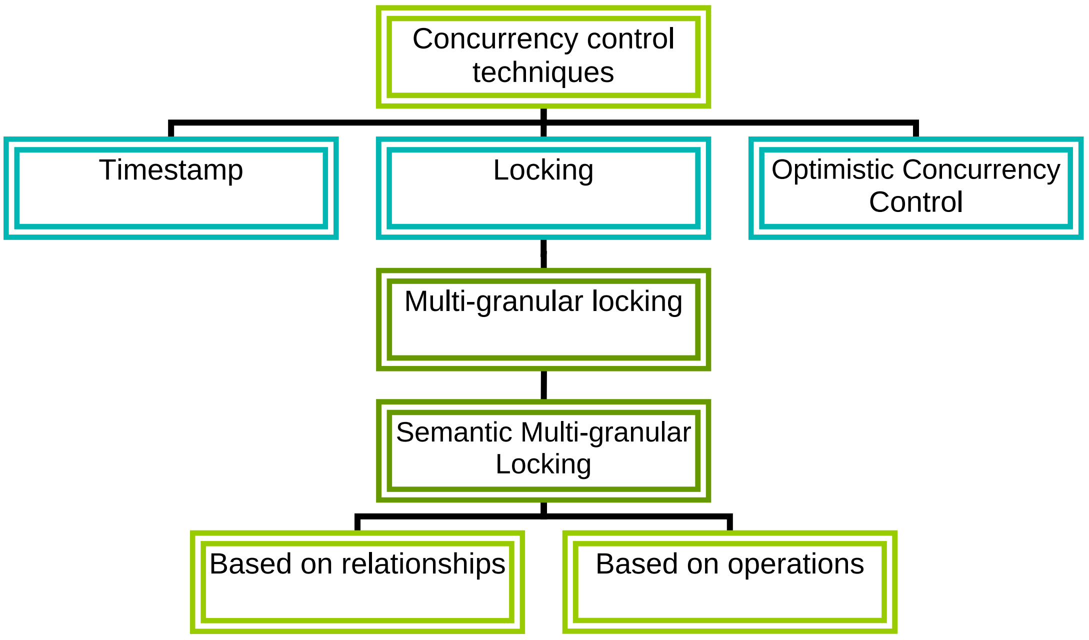

Concurrency control techniques
Timestamp
Locking
Optimistic Concurrency Control
Multi-granular locking
Semantic Multi-granular Locking
Based on relationships
Based on operations

Figure 2.6 Classification of concurrency control techniques

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 parallelly. 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 *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 *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.

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 $r_1 \cap r_2 \cap \ldots . r_n$ where $r_i \in R$. This means that a transaction can execute only when it gets all the resources.

Let T1, T2….Tn ∈ 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

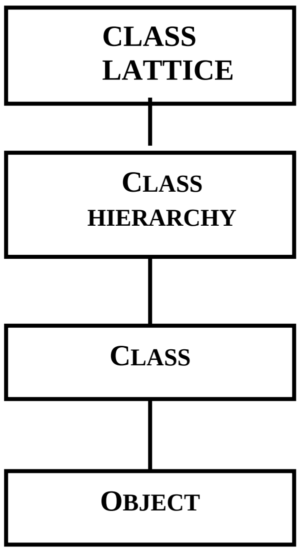


Figure 2.7 Hierarchy of lock granules in OODS

when the previous resource is granted to it. All the resources that it needs are maintained in the $REQUEST_{Ti}$ list. All the resources granted to it are maintained in $ALLOCATED_{Ti}$ list. Any request that is granted will be removed from $REQUEST_{Ti}$ and added to $ALLOCATED_{Ti}$. Once the execution of Ti is over, $ALLOCATED_{Ti}$ 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 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.

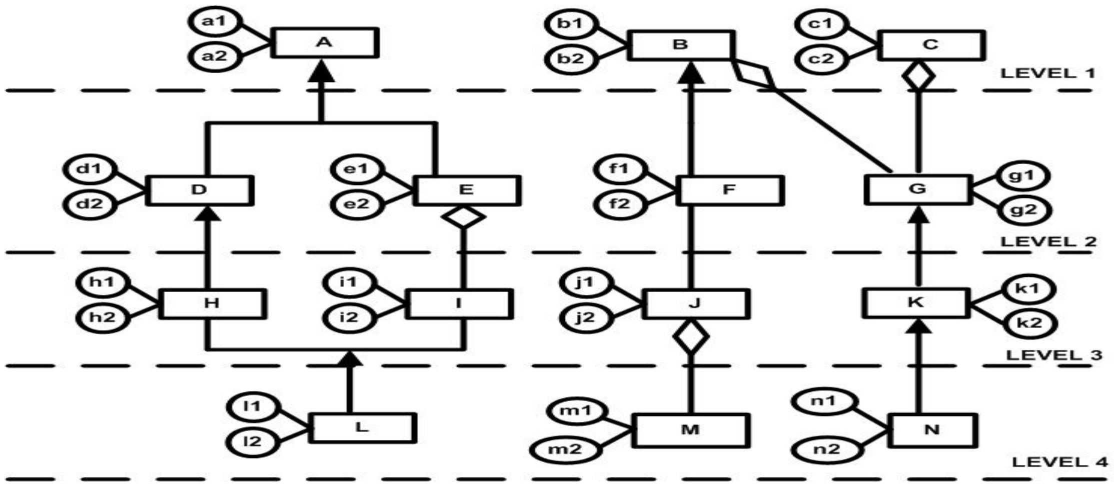


Figure 2.8 Sample class diagram with horizontal partitioning

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:

1. Selection Criteria: Youngest [Agarwal1987].

   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.

8. Selection Criteria: Blocker [Agarwal1987]

   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.

9. Selection Criteria: Minimum work done [Agarwal1987]

   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.

13. Selection Criteria: low priority + least resource priority + min. work done [Lin1996]

    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.

14. Selection Criteria: Minimum Abortion cost [Lin1994]

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 β 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.

CHAPTER 3

CONSISTENCY ENSURED SEMANTIC MULTI-GRANULAR LOCK MODEL (CESMGL) FOR OBJECT ORIENTED DATABASES IMPLEMENTING STABLE DOMAINS

3.1 Preamble

OODBMS is a collection of objects. Users may access the OODBMS for data (runtime transactions) or schema (design time transactions). 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 instance level (involving any one instance in a class) based on the property of methods [Riehle2000b]. A 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. In OODBMS, the structure is defined by a set of related classes participating in the domain. The classes may be related by inheritance, aggregation and association relationships. Design time transactions are classified into transactions requesting at node content level, transactions requesting at node level and transactions requesting at edge level [Bannerjee1989]. The design time transactions are sparse in stable business domains.

Multi- Granular Lock Model (MGLM) is a common technique for implementing concurrency control on the transactions using the OODBMS. The main advantages of MGLM are providing high concurrency and minimal deadlocks.

Though there are several semantic based MGLM [Garza1988, Kim1989, Malta1993, Lee1996, Saha2009, Jun2000], they have the following lacuna:

- Maximum concurrency is achieved for runtime transactions in Jun2000's scheme at the cost of consistency. In the other schemes, the granularity for runtime transactions is coarse.

- None of them has fully exploited the semantics of attributes, methods and class relationships to maximize the concurrency of design time transactions. In Jun2000's scheme, medium size granularity is provided for design time transactions requesting at node content level. The node level and link level design time transactions are executed in serialized manner because of the coarse granularity support.
- Fine granularity lock modes for the above mentioned three types of design time transactions are not proposed.

This chapter aims in proposing a semantic MGLM, which will improve the degree of concurrency for design time transactions and runtime transactions by fully utilizing the semantics of object oriented features. The proposed CESMGL scheme has the following characteristics: First, it proposes fine granularity for all types of design time transactions. Second, it provides separate lock modes covering all types of operations possible by design time transactions mentioned in Bannerjee1987. Third, it demonstrates that fine granularity of runtime transactions can be provided without affecting consistency. Fourth, it proposes enhanced commutativity matrix between all design time transactions and runtime transactions. The CESMGL scheme provides more parallelism between design time transactions and runtime transactions. A simulation model is constructed to evaluate the performance of the proposed work. This model is used to compare the proposed work with the latest existing techniques. The performance results show that the CESMGL scheme is better than existing works.

3.2 The CESMGL Scheme

The CESMGL scheme aims in providing the following objectives. It ensures consistency among runtime transactions and provides fine granule locking for design time transactions. It will not only improve concurrency among design time transactions, but will also improve concurrency between runtime transactions and design time transactions. The principles based on which the concurrency is improved among design time transactions and between runtime transactions and design time transactions are discussed in the following sections. In the next section, modifications are made to Jun2000's scheme to preserve the consistency among runtime transactions. In section 3.2.2, commutativity matrix of fine grained design time operations is defined. Another commutativity matrix is defined to maintain the consistency in accessing the attributes and methods from adapted classes. In section 3.2.3, semantics involved in parallel access of data and schema is discussed.

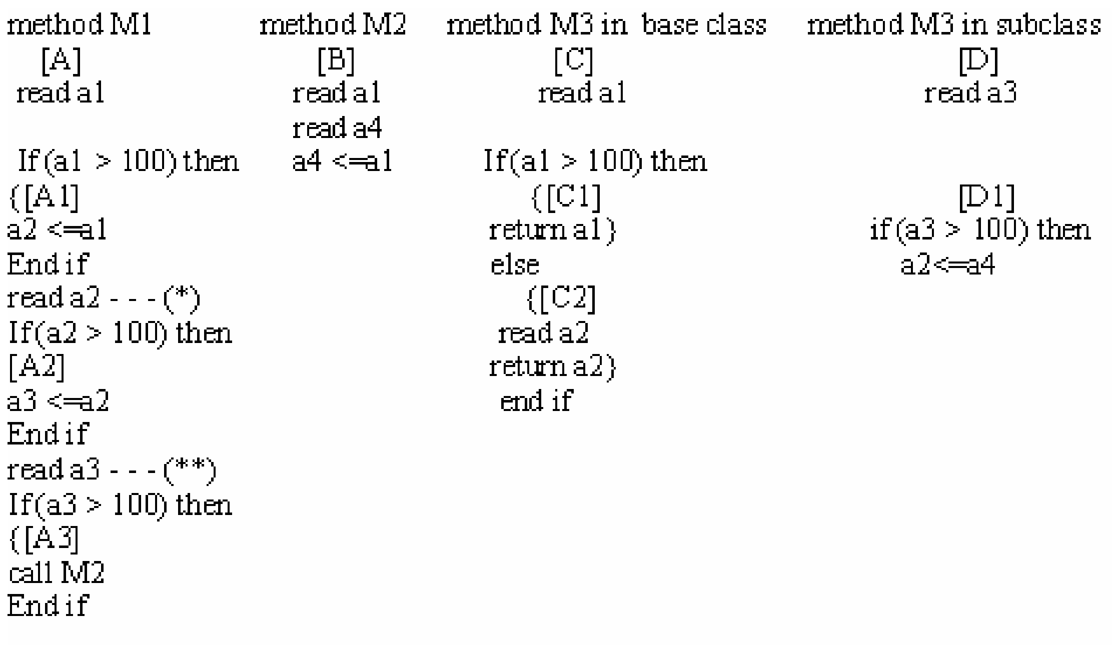
3.2.1 Consistency among runtime transactions

The models proposed before Jun2000 offer coarse granularity for runtime transactions. Jun2000 provides the finest granularity of access for runtime transactions. It offers the granularity of attributes with breakpoints which is smaller than the granularity of attributes. This means that the attribute can be shared by runtime transactions between break points. Jun2000 also provides better concurrency for design time transactions than the models proposed before his scheme.

In Jun2000, a pre-analysis is done to define commutativity among all methods in a class. It involves two steps: (1) construction of DAV for every method and (2) construction of a commutativity table of methods. In each method, a programmer or a compiler inserts a breakpoint when a conditional statement is encountered. Every method has a special breakpoint called first breakpoint before the first statement in the method. There are three types of DAVs in each method: (1) a final DAV of the first breakpoint, which is a DAV of the entire method as in Malta1993, (2) an initial DAV of the first breakpoint, which is a union of access modes of each attribute used by statements between the first breakpoint and the next breakpoint and access modes of each attribute used by statements from the first statement to the last statement that are executed regardless of execution paths. A union operation “+" is equivalent to max, e.g., R+W =W take more restrictive mode among two operations. Union operation is necessary to build worst-case access mode of each attribute, and (3) an initial DAV of every other breakpoint, which contains access modes of all attributes used by statements between this breakpoint and the next breakpoint. This is done up to the end of the method.

In Jun2000, the dependency of a class with other classes is not considered while constructing the DAV of a class. The DAV proposed in this paper is correct and performs well for implementation inheritance, aggregation and association. However, it will not work for interface inheritance. In implementation inheritance, both the interface and implementation of a base class method (template method) are inherited into the subclass. In interface inheritance, only the interface of the method is inherited into the subclass. The subclass is allowed to define its own implementation for this method. The base class methods that are reimplemented in the subclasses are called hook methods [Riehle2000a]. The reimplementation of hook methods in subclasses is not only applicable for single inheritance but also for multi level inheritance and hierarchical inheritance. In multi level inheritance, the hook method is reimplemented at every level of inheritance. In hierarchical inheritance, all the subclasses inherited from the common base class will have separate implementations. Then while defining DAV for a class, it is necessary to consider the implementations of these hook methods in all the subclasses. This is required to preserve the consistency of the attribute. It can be explained with an example. The example used in Jun2000 is used here to show the changes to be done to preserve the consistency. Here single inheritance is assumed for simplicity.

Assume that there are four attributes a1, a2, a3, a4 and three methods M1, M2 and M3 in the base class. Object O1 is an instance of base class. Let M1, M2 be template methods and they are inherited as it is in subclasses. Let M3 be a hook method and it has separate implementations in base class and subclass. Let A, A1, A2, and A3 are breakpoints of M1, B is a breakpoint of M2, and C, C1, and C2 are breakpoints of M3. Let D and D1 be break points of M3 in subclass implementation. Object O2 is an instance of subclass. Note that the operator '+' stands for union. The contents and DAVs of each method are given below:



```
method M1           method M2    method M3 in base class    method M3 in subclass
  [A]                  [B]             [C]                          [D]
 read a1             read a1          read a1                      read a3
                     read a4
 If(a1 > 100) then   a4 <=a1        If(a1 > 100) then
{[A1]                                 {[C1]                         [D1]
a2 <=a1                              return a1}                 if(a3 > 100) then
End if                               else                          a2<=a4
read a2 - - - (*)                      {[C2]
If(a2 > 100) then                     read a2
[A2]                                 return a2}
a3 <=a2                               end if
End if
read a3 - - - (**)
If(a3 > 100) then
{[A3]
call M2
End if
```

The contents and DAVs of each method are given below.

The DAVs constructed for method M1 are

initial DAV of [A] = DAV of [A] + DAV of[*] + DAV of [**]

= [R,N,N,N] + [N,R,N,N]+. [N,N,R,N]

= [R, R, R, N]

initial DAV of [A1] = [R,W,N,N]

initial DAV of [A2] = [N,R,W,N]

initial DAV of [A3] = final DAV of M2 = [R,N, N,W]

final DAV of [A] = initial DAV of [A]+ initial DAV of [A1]+ initial DAV of [A2]+ initial DAV of [A3]

= [R,R,R,N]+[R,W,N,N]+[N,R,W,N]+[R,N,N,W]+[R,W,W,W]

= [R, W, W, W]

Similarly, the DAVs for M2 are

final DAV of [B] = [R,N,N,W]

initial DAV of [B] = [R,N,N,W]

DAV for base class method M3 are

initial DAV of [C] = [R,N,N,N]

initial DAV of [C1] = [R,N,N,N]

initial DAV of [C2] = [N,R,N,N]

final DAV of [C] = [R,N,N,N] + [R,N,N,N] + [N,R,N,N]

= [R, R, N, N]

DAV for subclass method M3 are

Final DAV of [D] = initial DAV [D] + initial DAV [D1]

= [N, N, R, N] + [N, W, R, R]

= [N, W, R, R]

While in the scheme proposed in Malta1993, the DAVs for the methods would be

DAV of [M1] = [R,W,W,W]

DAV of M2 = [R,N,N,W]

DAV of base class method [M3]=[R,R,N,N]

DAV of subclass method [M3] = [N, W, R, R].

After constructing the DAVs for all the breakpoints in all methods, a commutativity table of methods is constructed. In a commutativity table, a lock requester's entry contains names of the final DAVs of the first breakpoints in all methods (represented as NF where N is the name of the first breakpoint in each method). For example, AF represents a final DAV of the first breakpoint A in method M1, which is [R, W, W, W]. A lock holder's entries contain names of the final DAV of the first breakpoint (in the form of N_F), name of the initial DAV of the first breakpoint (in the form of N_I) and names of the initial DAVs of other breakpoints (represented as Ni where i is ranging from 1 to number of breakpoints-1) in each method. For example, in method M1, A_F, A_I, A1, A2 and A3 represent the following DAVs [R,W,W,W], [R,R,R,N], [R,W,N,N], [N,R,W,N], [R,N,N,W] respectively. Since, the worst-case access mode is assumed for each attribute before execution to avoid problems of lock conversion, lock requesters always have the most restrictive access modes (i.e., final DAVs of the first breakpoints).However, after the execution of a method, a lock holder may have a less restrictive access mode. Two breakpoints commute if their corresponding DAVs commute. Two DAVs commute if, for every attribute, its access mode in the two DAVs commutes.

Table3.1 Commutativity matrix for the example comparing CESGML scheme and Jun2000 scheme

	A_F	A_I	A1	A2	A3	B_F	C_F	C_I	C1	C2	D_F	D_I	D1
A_F	N	N	N	N	N	N	Y	Y	Y	Y	N	N	N
B_F	N	Y	Y	Y	N	N	Y	Y	Y	Y	N	Y	N
C_F	N	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	N	N
D_F	N	N	N	N	N	N	Y	Y	Y	Y	N	Y	N

Table3.2 Commutativity matrix for the example comparing CESGML scheme and Malta1993 scheme

	M1	M2	BASE CLASS M3	SUBCLASS M3
M1	N	N	N	N
M2	N	N	Y	N
BASECLASS M3	N	Y	Y	N
SUBCLASS M3	N	N	N	N

Table 3.1 gives the commutativity table constructed in the CESMGL scheme to that of Jun2000's scheme. The shaded portion shows Jun2000's scheme. The CESMGL scheme shows where consistency is to be preserved. Table 3.2 gives the commutativity table for the CESMGL scheme and the shaded portion in the scheme proposed in Malta1993. Note that Y and N denote "compatible" and "incompatible" status of transactions respectively. In Jun2000's scheme, method M2 commutes with base class implementation of M3. However M2 does not commute with subclass implementation of M3.This consistency requirement is ignored in Jun2000's scheme. The CESMGL scheme appends the consistency requirements to preserve the

consistency of attributes. It is also interesting to note in the example that the base class and subclass implementations of M3 themselves do not commute. From this it can be inferred that when both objects O1 as well as O2 calls M3, they should not be allowed to execute in parallel as they do not commute. The CESMGL concurrency control scheme is based on two-phase locking. Therefore the correctness of proposed commutative of operations need not be proved [Eswaran1976].

It should be noted that whenever the implementation of a method is changed, the corresponding DAV should be changed accordingly. This means that commutativity relationships should also be redefined.

3.2.2 Concurrency among design time transactions

The CESMGL scheme defines the lock modes for design time operations with the following objectives:

1. Exploit the features of object oriented concepts to identify mutually exclusive operations in the system.
2. Maximize concurrency by providing rich set of lock modes.
3. Provide commutativity matrix independent of domain or specific instances, so that it does not require any apriori analysis.
4. Impose concurrency control wherever consistency is affected due to semantics of object oriented concepts.

The CESMGL scheme aims to cover all the operations that could be done on the schema as defined in [Kim1990, Bannerjee1987]. The operations can be seen at three levels: at node (class) contents level, at node level and at link level. As specified in Bannerjee1987, the node contents are instances, attributes and methods. In the following paragraphs, the mutually exclusive operations and dependent operations are identified to propose the lock modes for design time operations.

Based on the semantics of inheritance, subclasses can read as well as modify their attributes, but they can only read base class attributes. Only the base classes can modify the base class attributes. This theory can be extended to aggregation and association also. The definitions of attributes defined in a component class can be modified only in that component class where as composite objects can only read them. Similarly the attributes in an associated class can be modified only in that associated class. The associative classes cannot alter them. In simple words, modification is possible only in the class in which the attributes are defined.

Therefore, the attributes from the base classes, component classes and associated classes can be viewed as adapted attributes in the subclass or composite class or associative classes. In Jun2000, adapted attributes and attributes defined in the class cannot be accessed in parallel even though they are mutually exclusive, without using AAV. It also does not address the need for controlling the parallel execution of modifying an attribute definition in the base class (component class/associated class) while reading the same attribute definition in the subclass (composite class/associative class). Allowing this will lead to dirty reads.

By object oriented semantics, modification of methods typically includes modifying the interface of the method, modifying its implementation and modifying its location i.e., moving a method from one class to another class in the class hierarchy. Modifying the interface means modifying the name of the method, adding or deleting the input parameters, changing their order and changing the returning type. In Jun2000, all the operations related to modification of methods are done in the same lock mode.

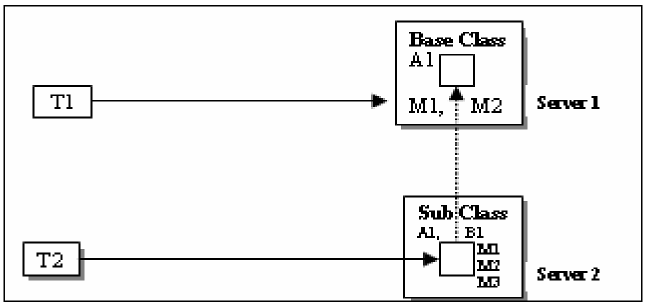


Figure 3.1 Locking in Inheritance

For example, consider figure 3.1. A1 is an attribute of base class and it is inherited in subclass. B1 is subclass attribute. M1 and M2 are methods of base class. In this M1 is inherited as it is in subclass (called as template method) [Riehle2000a], whereas M2 is overridden in subclass (called as hook method) [Riehle2000a]. M3 is a method defined in subclass. By Jun2000, all types of attributes i.e., adapted attribute A1 and defined attribute B1 are locked by a single lock mode when the sub class object is involved. All the methods i.e., template method M1, hook method M2 and local method M3 are locked by a single lock mode in the subclass. The semantics of each of these attributes and methods are not utilized to maximize the concurrency.

There are certain aspects that can be inferred from figure 3.1. Attribute A1 can be read in both base class as well as in subclass. However, modifying A1 is possible only in base class. It is worth noting that while base class is modifying the definition of A1, no transaction should be allowed for the subclass to read the definition of A1 to maintain consistency. In subclass, attribute B1 can be read or modified. So the attributes in any class can be categorized into two categories: Attributes adapted from other classes and Attributes defined in the same class. Hence, the attributes are classified into Adapted Attributes (AA) and Attributes (A). Then separate lock modes can be defined for reading adapted attributes(RAA) and reading and modifying defined attributes (RA,MA). Since adapted attributes cannot be modified in this class, lock mode for modifying the adapted attributes is not available in the sub class. So, all the subclasses will have adapted attributes and attributes defined in that class. However as the base classes do not have any parents; their adapted attributes list will be empty.

In figure 3.1, M1 is a template method whose interface and implementation can be modified only in the base class. It can only be read in the subclasses. M2 is a hook method. Therefore, its interface is modifiable only in the base class and implementation is modifiable in both base class and subclass. This means hook methods can be overridden and can have different implementations in base class and subclass. M3 is a method defined in subclass and can be read and modified only in subclass, as it is not visible in the base class. Similarly, interface and implementation of methods defined in the component class cannot be modified in composite class. They are available only for reading in the composite class. Hence, they can be treated similar to template methods of inheritance. This can be extended to association also. In Jun2000, MM is the only lock mode to handle all these method types.

Szyperski2002 says that the interface i.e. method definition is independent of method implementation. This concept is called as separation of concerns. In object oriented environment, the implementation of a method can be modified any number of times. As long as its interface definition does not change, the clients need not be informed about the change in the implementation. The implementation of the methods is usually modified to provide better service to clients. However, when the interface is changed, the clients need to be informed, as they are going to avail this service only by calling in this interface format. In fact, the interface is viewed as a contract between client and server. So lock mode for accessing a method is split into Method Signature (MS) and Method Implementation (MI) in the CESMGL scheme. The interfaces or definitions of the methods in base classes, component classes and associated classes are separately maintained in the class. Hence separate lock modes can be defined for reading the interfaces of adapted class methods (RAMS) and for reading and modifying subclass methods (RMS, MMS). Since interfaces of the methods in adapted class cannot be modified in the class where they are used, lock mode for modifying the interface of these adapted methods is not available in this class.

Modification of template methods in base classes, component classes and associated classes is possible only in respective classes and is bound to the related subclass lattice. However, Hook methods are overridden in subclasses. Hence, their implementation in base class and subclasses can be independently modified without affecting the other implementations.

Hence, MI is further split into Adapted Method Implementation (AMI) and Method Implementation (MI) in the CESMGL scheme. The implementation can be read or modified by the lock modes (RAMI, MAMI, RMI and MMI). There is only read mode available for adapted method implementations for all the methods except hook methods. MAMI represents modification of the hook methods and MMI is for modifying methods defined in this class. When the transactions request to modify adapted attributes in the respective classes where they are defined and try to read them in the class where they are adapted, such parallel accesses should not be allowed to maintain consistency. This is applicable to the modification of interfaces and implementation of methods also to maintain consistency. Hence, the compatibility of lock modes on these attributes, interfaces and implementations of these methods at both the classes where they are defined and where they are read, need to be checked to ensure consistency. New lock modes such as AI (Add Instance), DI (Delete Instance) AA (Add Attribute), DA (Delete Attribute), AM (Add Method) and DM (Delete Method) are defined to insert and delete instances, attributes and methods for a class. Thus we have defined lock modes for all the design time operations accessing at the node content level as defined in Bannerjee1987. Now the node level operations are explored to provide fine granularity and promote concurrency.

Modifying a class involves adding, deleting a class or moving its location in the class lattice. In Jun2000, class definition includes name of the class, all attributes and methods defined in the class, set of super classes and subclasses of the class. This combines the class level and link or edge level operations in the schema. Based on Bannerjee1987 classification of modification operations of schema, it can observed that class lattice level operations represented by MCR (Modify Class Relationship) lock mode in Jun2000 blocks all the class level as well as other class lattice level operations along with runtime transactions.

In Jun2000, MCR lock mode is defined for modifying class definitions. It is used to add or drop a class and to modify super class/subclass relationship. RCR lock mode is defined to read the definition of a class. Class definition includes name of the class, set of all attributes defined or inherited into the class, sets of super classes and subclasses of the class and set of methods defined or inherited into the class. If the operations defined in Jun2000 are compared with Bannerjee1987, it can be inferred that the operations defined in Jun2000 is only a subset of that of Bannerjee1987. This results in two consequences. First, there are no sufficient lock modes to cover all the schema design operations. Second, for a large number of operations, the granularity is coarse. Then to promote concurrency, the operations in class level and edge level are to be categorized into two groups:

1. Operations affecting at subclass lattice level
2. Operations affecting at class lattice level.

Lock modes like AA, MA, DA, RA, AM, MMS, MMI, DM, AI, DI affect not only the class for which they are requested but also its adapted classes. This collection of related classes is called as subclass lattice. There can be more than one subclass lattice in a class lattice. Apart from the operations mentioned above, changing class name, changing the order of relationship between two classes, adding a class, dropping a class, making a class as parent class (component class/associated class) to a subclass (composite class/associative class), removing a class from the list of parents (component classes/associated classes) of a class and changing the order of parent classes of a class are also operations affecting at subclass lattice level. These operations do not affect other subclass lattices.

The operations like move an attribute/method from one class to another class and move a class from one position to another position in the class lattice affect the whole lattice as they may be moved from one subclass lattice to another subclass lattice. Then it can be inferred that move operation affects the entire class lattice. This operation should not be allowed to execute with any of the other transactions. So the lock modes for the above operations can be grouped as Modify Subclass Lattice (MSCL) and Modify Class Lattice (MCL).

Table 3.3 Proposed commutativity matrix for design time transactions

	AA	RAA	RA	MA	DA	AM	RAMS	RMS	MMS	RAMI	MAMI	RMI	MMI	DM	AI	DI	RSCL	MSCL	RCL	MCL
AA	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N
RAA	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N
RA	Y	Y	Y	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N
MA	Y	Y	N	N	N	N	Y	Y	Y	Y	N	Y	Y	N	N	N	N	N	N	N
DA	Y	Y	N	N	N	N	Y	N	N	Y	N	N	N	Y	N	N	N	N	N	N
AM	Y	Y	Y	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N
RAMS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	N	Y	N
RMS	Y	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	N	N	Y	Y	Y	N	Y	N
MMS	Y	Y	Y	Y	N	Y	Y	N	N	Y	Y	N	N	N	N	N	N	N	N	N
RAMI	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	N	Y	N
MAMI	Y	Y	Y	N	N	Y	N	Y	Y	N	N	Y	Y	Y	N	N	N	N	N	N
RMI	Y	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	N	N	Y	Y	Y	N	Y	N
MMI	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N
DM	Y	Y	Y	N	Y	Y	Y	N	N	Y	Y	N	N	Y	N	N	N	N	N	N
AI	N	Y	Y	N	N	N	Y	Y	N	Y	N	Y	N	N	N	N	N	N	Y	N
DI	N	Y	Y	N	N	N	Y	Y	N	Y	N	Y	N	N	N	N	N	N	Y	N
RSCL	Y	Y	Y	Y	N	N	Y	Y	N	N	N	Y	N	N	Y	Y	Y	N	Y	N
MSCL	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
RCL	N	Y	Y	N	N	N	Y	Y	N	Y	N	Y	N	N	Y	Y	Y	N	Y	N
MCL	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

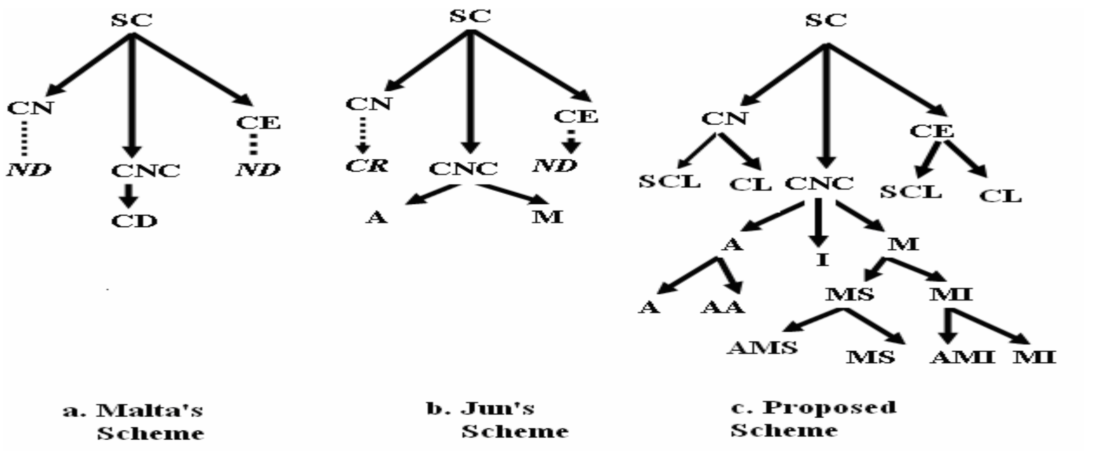
Table 3.4 Commutativity matrix for design time transactions accessing defined class and adapted class

		Adapted Classes			
		RAA	RAMS	RAMI	MAMI
Defined Class	AA	N	Y	Y	N
	RA	Y	Y	Y	Y
	MA	N	Y	Y	N
	DA	N	N	N	N
	AM	Y	N	N	N
	RMS	Y	Y	Y	Y
	MMS	Y	N	N	N
	MMI	Y	Y	N	Y
	RMI	Y	Y	Y	Y
	DM	Y	N	N	N

Table 3.3 defines the commutativity matrix defined for class lattice. Table 3.4 defines the commutativity matrix for classes where the attributes and methods (interface and implementation) are defined against where they are adapted. In the

tables, Y means two methods always commute and N means they never commute. The shaded portion in the table indicates all the possible operations on a class content namely attributes, methods and instances. As they are frequently accessed operations at sub class lattice level, separate lock modes are defined for all of them. The remaining operations are allowed using MSCL lock mode.

Using the commutativity of lock modes in Table 3.3, a finer granularity lock can be obtained. The lock granularity in the proposed work is one of the lock modes such as MA, MMS, MMI, MAMI, AA, DA, AM, DM, AI, DI, MSCL, MCL and RAA, RA, RAMS, RMS, RAMI, RMI, RSCL, RCL. Whenever a design time transaction arrives, Table 3.3 is checked for compatibility. If the transaction accesses subclass, composite class or associative class and lock mode is one of RAA, RAMS, RAMI and MAMI or if it is base class, component class or associated class and the lock mode is one of AA, RA, MA, DA, AM, MMS, MMI, DM, Table 3.4 is checked for compatibility. The hierarchy of granules in design time transactions in Malta1993, Jun2000 and CESMGL scheme can be summarized as given in figure 3.2.

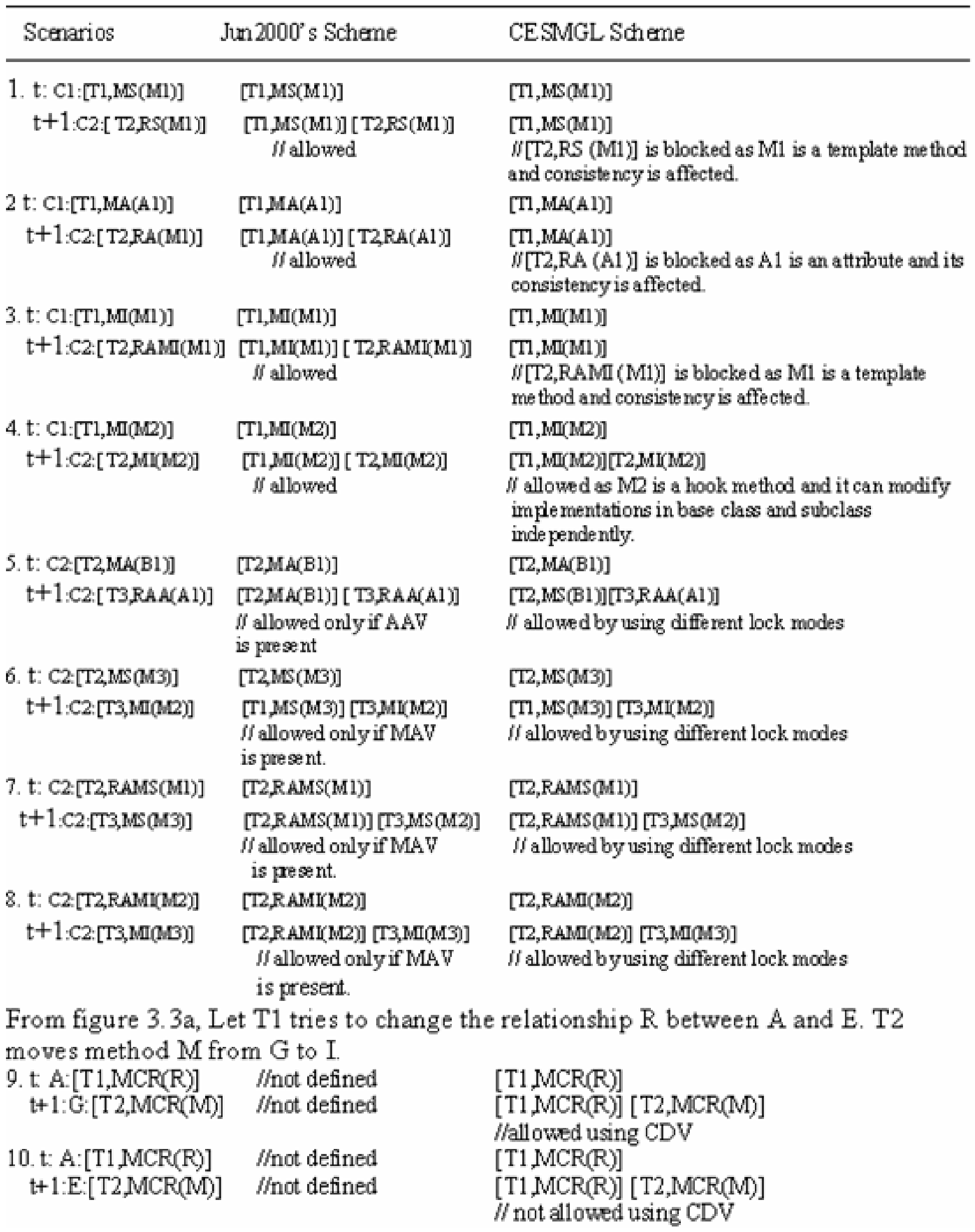


ND – Not Defined	MS – Method Signature	M – Methods
SC – Schema Changes	CN – Changes to Node	I – Instances
MI – Method Implementation	A – Attributes	AA – Adapted Attributes
CE – Changes to Edges	CL – Class Lattice level	SCL – Subclass Lattice level

CNC – Changes to Node Content
AMS – Adapted Method Signature AMI – Adapted Method Implementation

Figure 3.2 Comparison of hierarchy of granules of design time transactions

Let us consider the various scenarios to show how Jun2000's scheme and CESMGL scheme works: Let T1 be a transaction arriving at t. Let T2 and T3 be transactions arriving at t+1. Let us assume that each transaction takes at least 1 second to complete. Let class name: [tran-name, lock type (item name)] be the format for design time transaction. Item name refers to the name of the attribute or method which is accessed. Let us consider figure 3.1. Let the base class be C1.Let its subclass be C2. T1 requests C1. T2 and T3 requests C2. The scenarios will show how the scheme ensures consistency wherever necessary and improves concurrency wherever possible.



Scenarios	Jun2000's Scheme	CESMGL Scheme
1. t: C1:[T1,MS(M1)]	[T1,MS(M1)]	[T1,MS(M1)]
t+1:C2:[T2,RS(M1)]	[T1,MS(M1)] [T2,RS(M1)] // allowed	[T1,MS(M1)] //[T2,RS (M1)] is blocked as M1 is a template method and consistency is affected.
2 t: C1:[T1,MA(A1)]	[T1,MA(A1)]	[T1,MA(A1)]
t+1:C2:[T2,RA(M1)]	[T1,MA(A1)] [T2,RA(A1)] // allowed	[T1,MA(A1)] //[T2,RA (A1)] is blocked as A1 is an attribute and its consistency is affected.
3. t: C1:[T1,MI(M1)]	[T1,MI(M1)]	[T1,MI(M1)]
t+1:C2:[T2,RAMI(M1)]	[T1,MI(M1)] [T2,RAMI(M1)] // allowed	[T1,MI(M1)] //[T2,RAMI (M1)] is blocked as M1 is a template method and consistency is affected.
4. t: C1:[T1,MI(M2)]	[T1,MI(M2)]	[T1,MI(M2)]
t+1:C2:[T2,MI(M2)]	[T1,MI(M2)] [T2,MI(M2)] // allowed	[T1,MI(M2)][T2,MI(M2)] // allowed as M2 is a hook method and it can modify implementations in base class and subclass independently.
5. t: C2:[T2,MA(B1)]	[T2,MA(B1)]	[T2,MA(B1)]
t+1:C2:[T3,RAA(A1)]	[T2,MA(B1)] [T3,RAA(A1)] // allowed only if AAV is present	[T2,MS(B1)][T3,RAA(A1)] // allowed by using different lock modes
6. t: C2:[T2,MS(M3)]	[T2,MS(M3)]	[T2,MS(M3)]
t+1:C2:[T3,MI(M2)]	[T1,MS(M3)] [T3,MI(M2)] // allowed only if MAV is present.	[T1,MS(M3)] [T3,MI(M2)] // allowed by using different lock modes
7. t: C2:[T2,RAMS(M1)]	[T2,RAMS(M1)]	[T2,RAMS(M1)]
t+1:C2:[T3,MS(M3)]	[T2,RAMS(M1)] [T3,MS(M2)] // allowed only if MAV is present.	[T2,RAMS(M1)] [T3,MS(M2)] // allowed by using different lock modes
8. t: C2:[T2,RAMI(M2)]	[T2,RAMI(M2)]	[T2,RAMI(M2)]
t+1:C2:[T3,MI(M3)]	[T2,RAMI(M2)] [T3,MI(M3)] // allowed only if MAV is present.	[T2,RAMI(M2)] [T3,MI(M3)] // allowed by using different lock modes

From figure 3.3a, Let T1 tries to change the relationship R between A and E. T2 moves method M from G to I.

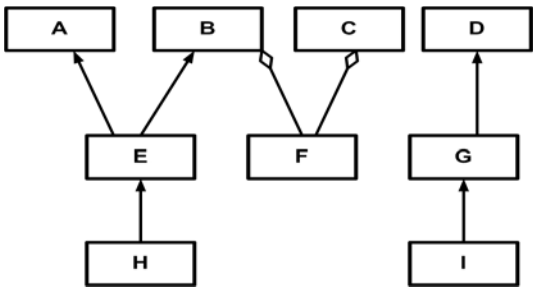
9. t: A:[T1,MCR(R)]	//not defined	[T1,MCR(R)]
t+1:G:[T2,MCR(M)]	//not defined	[T1,MCR(R)] [T2,MCR(M)] //allowed using CDV
10. t: A:[T1,MCR(R)]	//not defined	[T1,MCR(R)]
t+1:E:[T2,MCR(M)]	//not defined	[T1,MCR(R)] [T2,MCR(M)] // not allowed using CDV

3.2.3 Concurrency between runtime transactions and design time transactions

The runtime transactions and design time transactions can have fine granularity access by using lock modes as well as access vectors. In Jun2000's scheme, AAV and MAV are defined to concurrently access more than one attribute or method in the same class. However for all the other operations the granularity is still coarse. Changes to class lattice at both node level as well as link level are represented by

MCR lock mode. Therefore, overall concurrency is still restrained. This can be explained with an example. Consider the sample class diagram as in figure 3.3a. The CDV (Class Dependency Vector) is used to identify the subclass lattices in the class diagram. There are 3 subclass lattices in the sample class diagram namely (A,B,E,H), (B,C,F) and (D,G,I). So, if any class in a subclass lattice is requested, all the classes in the same subclass lattice are also locked. This provides concurrency at subclass lattice level for class lattice level design operations with runtime operations.

In general in a class lattice, there is an upward dependency for runtime transactions. i.e., whenever a runtime transaction is made on subclass, composite class or associative class, its corresponding base class, component class or associative class also has to be locked. On the other hand, there is a downward dependency for design time transactions. Whenever there is any structural modification in base class, component class or associative class, it is passed on to the relative subclass, composite class or associative class. Then concurrency can be further improved by locking the defined classes for runtime transactions and adapted classes for design time transactions. For example in figure 3.3a, if a runtime request is made to E, its defined classes A and B are to be locked. If a design time request is made to E, its adapted class H is to be locked.



Defined Classes	Class	Adapted classes
----	A	E,H
------	B	E,H,F
-------	C	F
-----	D	G,I
A,B	E	H
B,C	F	---
D	G	I
A,B,E	H	---
D,G	I	-----

(a) (b)

Figure 3.3(a) Sample class diagram **(b)** CDV for the sample class diagram in table format

This can be defined using Class Dependency Vector (CDV) in table format as in figure 3.3b. From figure 3.3b, it can be observed that classes at leaf level like F, H and I do not influence other classes. Then any changes done at the node level in these classes like changing the class name, deleting this class, adding/ deleting attributes and adding / deleting methods can be done at class level. Similarly classes closer to the root level like A, B, C and D are not affected by other classes. Then the commutativity matrix for transactions between defined class and adapted classes need not be checked for these classes. Thus it can be observed that for all subclass lattice operations of the transactions, maximum concurrency can be achieved. The AAV and MAV vectors facilitates intra class parallelism, while CDV facilitates inter class parallelism.

Table 3.5 gives the commutativity matrix for design time and runtime transactions. Table 3.6 is used to ensure consistency among transactions that are accessing in parallel a defined class and its adapted classes. In the tables, Y means two methods always commute, N means they never commute and Δ means that the two methods commute only when they access disjoint portions of a class.

Table 3.5 Proposed commutativity matrix for design time transactions and runtime transactions using access vectors

	AA	RAA	RA	MA	DA	AM	RAMS	RMS	MMS	RAMI	MAMI	RMI	MMI	DM	AI	DI	RSCL	MSCL	RCL	MCL	IA
AA	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Δ	Δ	Δ	Δ	Δ	N	Δ
RAA	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Δ	Y	N	Y
RA	Y	Y	Y	Δ	Δ	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Δ	Y	N	Y
MA	Y	Y	Δ	Δ	Δ	Δ	Y	Y	Y	Y	Δ	Y	Y	Δ	Δ	Δ	Δ	Δ	Δ	N	Δ
DA	Y	Y	Δ	Δ	Δ	Δ	Y	Δ	Δ	Y	Δ	Δ	Δ	Y	Δ	Δ	Δ	Δ	Δ	N	Δ
AM	Y	Y	Y	Δ	Δ	Y	Y	Y	Y	Y	Y	Y	Y	Y	Δ	Δ	Δ	Δ	Δ	N	Δ
RAMS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Δ	Y	Y	Y	Y	Y	Y	Δ	Y	N	Y
RMS	Y	Y	Y	Y	Δ	Y	Y	Y	Δ	Y	Y	Y	Δ	Δ	Y	Y	Y	Δ	Y	N	Y
MMS	Y	Y	Y	Y	Δ	Y	Y	Δ	Δ	Y	Y	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	N	Δ
RAMI	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Δ	Y	Y	Y	Y	Y	Y	Δ	Y	N	Y
MAMI	Y	Y	Y	Δ	Δ	Y	Δ	Y	Y	Δ	Δ	Y	Y	Y	Δ	Δ	N	Δ	Δ	N	Δ
RMI	Y	Y	Y	Y	Δ	Y	Y	Y	Δ	Y	Y	Y	Δ	Δ	Y	Y	Y	Δ	Y	N	Y
MMI	Y	Y	Y	Y	Δ	Y	Y	Y	Y	Y	Y	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	N	Δ
DM	Y	Y	Y	Δ	Y	Y	Y	Δ	Δ	Y	Y	Δ	Δ	Y	Δ	Δ	Δ	Δ	Δ	N	Δ
AI	Δ	Y	Y	Δ	Δ	Δ	Y	Y	Δ	Y	Δ	Y	Δ	Δ	Δ	Δ	Δ	Δ	Y	N	Δ
DI	Δ	Y	Y	Δ	Δ	Δ	Y	Y	Δ	Y	Δ	Y	Δ	Δ	Δ	Δ	Δ	Δ	Y	N	Δ
RSCL	Y	Y	Y	Y	Δ	Δ	Y	Y	Δ	Δ	Δ	Y	Δ	Δ	Y	Y	Y	Δ	Y	N	Y
MSCL	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	N	Δ
RCL	Δ	Y	Y	Δ	Δ	Δ	Y	Y	Δ	Y	Δ	Y	Δ	Δ	Y	Y	Y	Δ	Y	N	Y
MCL	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
IA	Δ	Y	Y	Δ	Δ	Δ	Y	Y	Δ	Y	Δ	Y	Δ	Δ	Δ	Δ	Y	Δ	Y	N	Δ

Table 3.6 Commutativity matrix for transactions accessing defined class and adapted classes using access vectors

Adapted Classes					
Defined Class		RAA	RAMS	RAMI	MAMI
	AA	Δ	Y	Y	Δ
	RA	Y	Y	Y	Y
	MA	Δ	Y	Y	Δ
	DA	Δ	Δ	Δ	Δ
	AM	Y	Δ	Δ	Δ
	RMS	Y	Y	Y	Y
	MMS	Y	Δ	Δ	Δ
	MMI	Y	Y	Δ	Y
	RMI	Y	Y	Y	Y
	DM	Y	Δ	Δ	Δ

For example, consider the following operations by transactions T1, T2 and T3 for the sample diagram in figure 3a. At time t, T1 is doing a runtime access on instance I1 of class B. At time t+1, T2 is modifying implementation of the hook method M1 of class H. At time t+2, T3 is modifying the name of the class G. At time t+3, T1 is modifying the definition of attribute a1 in class F. At time t+4, T2 is doing runtime transaction on instance I2 of class D.

Time	T1	T2	T3
t	B: IA(I1)		
t+1		H: MAMI on M1	
t+2			G:MSCL(Change class name)
t+3	F: MA (a1)		
t+4		D: IA (I2)	

The following paragraph shows how locks are changed on the class diagram for the above scenario.

t: CDV:[A: N, B: IA (I1), E: N, H:N]

Initially the lock status of subclass lattice (A, B, E, H) is null. At time t, the lock status of B is updated to a runtime lock. At time t+1, the lock status of H is updated.

t+1: CDV: [A: N, B:T1, IA (I1), E: N, H: T2,MAMI (M1)]

At time t+2, lock is requested to modify the name of class G. As any structural change in G will not affect class D, its lock status is not updated. However, I is inherited from G. So any change in G is also inherited to G. So I is also locked to maintain the consistency.

t+2: CDV: [D: N, G: T3, MSCL (class name), I: T3, MSCL]

At time t+3, T1 modifies the attribute definition of a1 in class F. F does not have any adapted class. Any structural change in class F, does not affect the other classes in the subclass lattice.

t+3: CDV: [A: N, B: N, F: T3, MA(a1)]

At time t+4, there is an instance access to I2 of class D. D is a base class. There are no defined classes for D. so it is enough to lock D alone.

t+4: CDV: [D: IA (I2), G: N, I: N]

Apart from CDV, AAV and MAV are maintained for every class to support parallel access of attributes and methods in the same class.

3.3 Experimental Results and Discussion

The CESMGL scheme is tested using a simulation model as in [Kim1991]. In this section, the simulation model, experimental details and analysis of the results are presented.

The simulation model used in the existing works is adopted for testing the CESMGL scheme. The existing simulation model [Kim1991] is used to facilitate easy comparison. The simulation model is implemented using Java. Fig 3.4 shows the architecture diagram of the simulation model. The various components in simulation model are transaction generator, transaction manager, CPU scheduler, lock manager, deadlock manager and buffer manager.

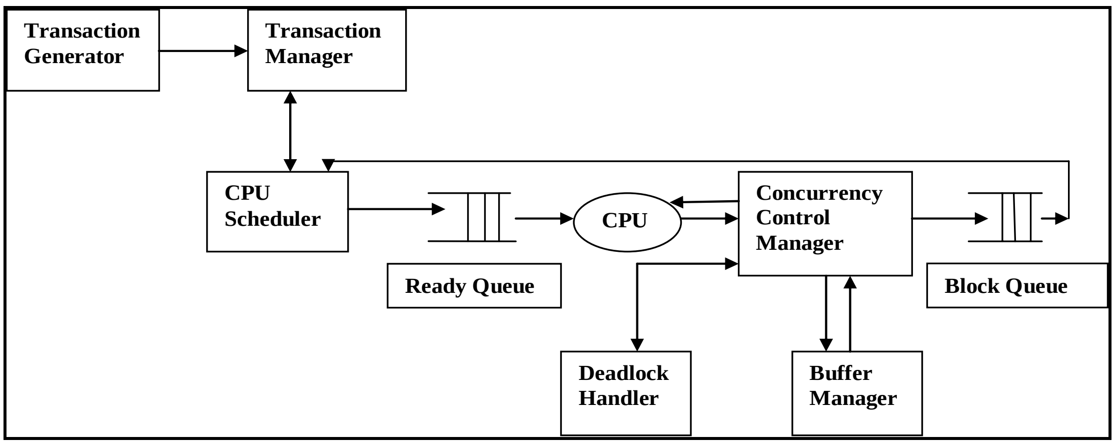


Figure 3.4 Simulation model [Kim1991]

The transaction generator creates each transaction with its transaction type, unique transaction identifier and creation time. The transaction format is (transaction type, resource type, resource-id). The transaction type can be runtime or design time transaction. The resource type can be one of attribute, method, instance, class, subclass lattice and class lattice. Transaction manager is responsible for scheduling the execution of all transactions. It sends lock requests to the lock manager and sends release messages on transaction completion. Deadlock handler detects presence of deadlock and aborts a transaction. The transaction manager eventually starts the aborted transaction. The aborted transaction still maintains the original creation time to preserve its seniority. The CPU scheduler is responsible for scheduling in-coming transactions. The transactions are served in FIFO order. The transaction holding CPU will not be preempted for other transactions to avoid wastage of work and resources.

The lock manager orders the resource accesses based on the proposed concurrency control scheme. A transaction request is served if its lock mode is compatible with the existing transactions. It is blocked if the lock mode is incompatible. The data is accessed from main memory. Buffer manager augments main memory access as disk access is time consuming and will not give correct picture on the performance of the system.

007 Benchmark by [Carey1993, Carey1994] is well known for testing the performance of OODBMS. It is used in Jun2000 for showing the performance of his proposal. But 007 benchmark defines the benchmark only for runtime transactions. It does not define any testing cases for design time transactions. So it cannot be fully adopted for the CESMGL scheme. However in the CESMGL scheme, the database model and testing cases of runtime transactions of 007 bench mark are adopted for performance evaluation. 007 benchmark classifies databases into small, medium and large, based on their size. Here, small size is chosen for simplicity. The design time transactions are formulated to cover all the three types of schema changes defined in Bannerjee1987. Table 3.7 gives the simulation parameters.

Table 3.7 Simulation parameters

Parameters	Default value(range)
Time to process one operation	0.00000625ms
Mean time to set lock by runtime transaction	0.3301 ms
Mean time to set lock by design time transaction	0.3422ms
Mean time to release lock	0. 0015ms
Multiprogramming level	8 (5-15)
Prob. of Traversal	0.25 (0-1)
Prob. of Query	0.25 (0-1)
Prob. of Schema change	0.5 (0-1)
Prob. of Changes to nodes	0.15 (0-1)
Prob. of Changes to edges	0.20(0-1)
Prob.of changes to node contents	0.15(0-1)
Transaction inter-arrival time	500(100-1000)
Database model [Carey1993]	Small (small, medium, large)

Three testing cases are chosen to measure the performance of the CESMGL scheme for all types of transactions: Varying arrival rate of the transactions, varying read-to-write ratio of design time transactions and varying runtime transaction to design time transaction ratio. The existing schemes chosen to compare against the

CESMGL scheme are Orion scheme [Garza1988] and Jun2000 scheme. Orion scheme is chosen as it is the best scheme based on relationships. Jun2000's scheme is chosen as it is the latest scheme based on access vectors. As the CESMGL scheme is also based on access vectors, it is compared against these two schemes.

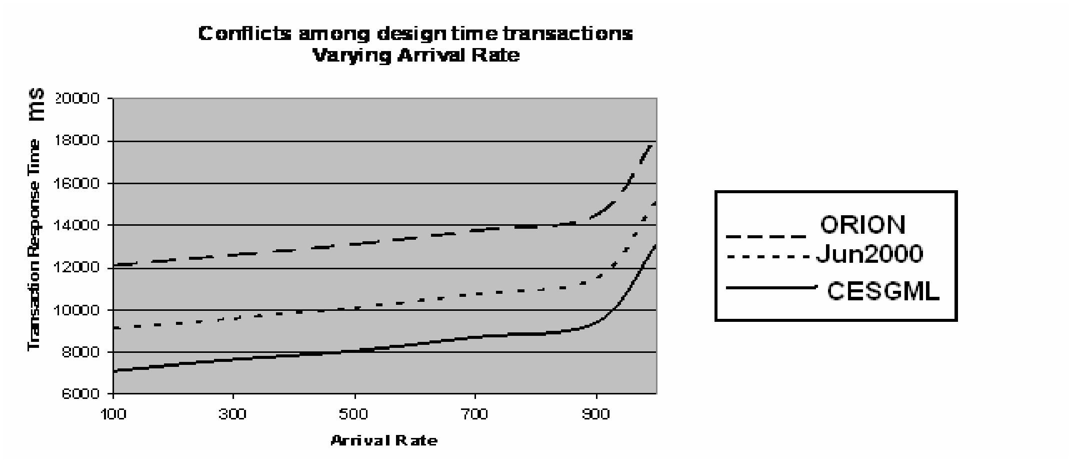


Figure 3.5 Performance by varying arrival rate

Figure 3.5 shows the test case of varying arrival rate. This test is done to evaluate how the schemes work under various system loads. Orion takes the maximum response time. Jun2000's scheme is better than Orion. The CESMGL scheme works better for all the loads. The average lock waiting time of Orion, Jun2000's scheme and the CESMGL scheme are 46.12ms, 35.43ms and 28.12ms respectively. Thus the CESGML scheme takes the least waiting time. It is because of the coarse to fine granularity they provide. Orion locks the entire instance object for runtime transactions and locks the entire class object for the class content level design time operations. It locks the entire class lattice for node level and edge level design time operations. In Jun2000's scheme fine granularity is achieved for runtime transactions at the level of attributes with break points at the cost of consistency. Medium granularity is achieved for class content design operations for accessing attributes and methods. For all other operations the entire class lattice is locked. In the CESMGL scheme, further concurrency is enhanced by introducing new lock modes. On the whole, CESMGL scheme is better than Orion by 46.88% and Jun2000's scheme by 18.5%. Jun2000's scheme is better than Orion by 34.8%.

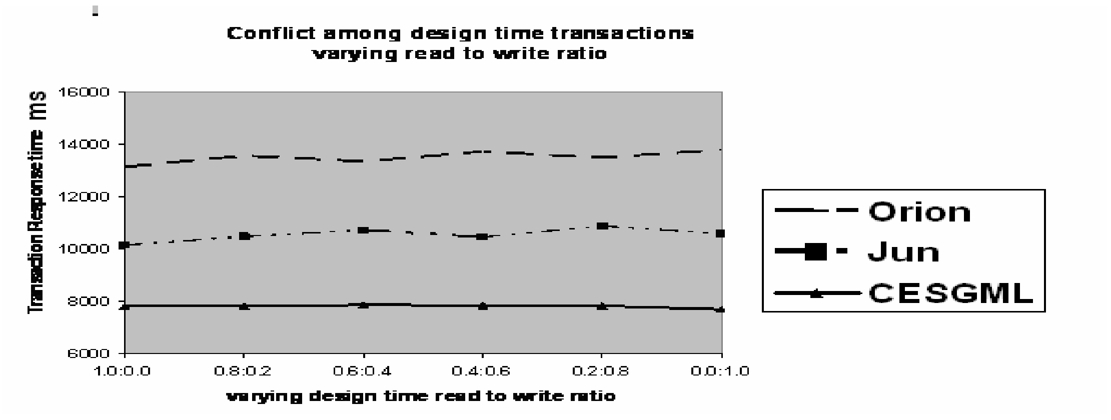


Figure 3.6 Performance by varying design time read to write ratio

Figure 3.6 shows the test case of varying design time read-to-write ratio. In this also, Orion's performance is poor. It is because Orion takes entire class object as the lock granularity for class definition access and there is no concurrency between read and write operations on class definition access. Further it serializes all class lattice and subclass lattice operations. Due to this the performance of Orion is poor. In Jun2000's scheme, the granularity is refined for class content modification. But other design operations are still at coarse level. In the CESMGL scheme, the finest granularity is achieved for class content level operations. The other design operations are also divided into sub class lattice and class lattice level operations. The average lock waiting time of Orion, Jun2000's scheme and CESMGL scheme are 49.12ms, 35.43ms and 28.12ms respectively. On the whole, CESMGL scheme is better than Orion by 68% and Jun2000's scheme by 32.1%. Jun2000's scheme is better than Orion by 28.04%.

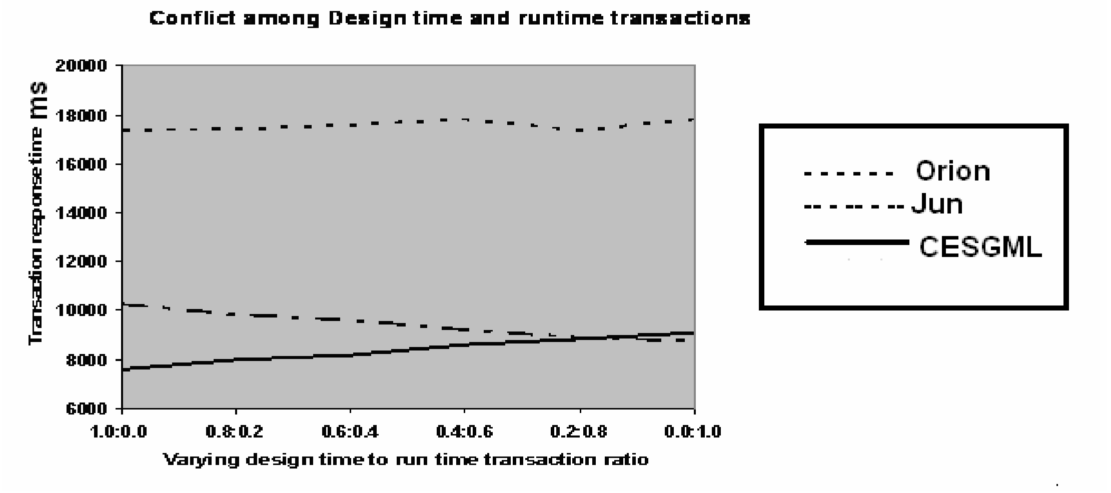


Figure 3.7 Performance by varying design time to runtime transaction ratio

Figure 3.7 shows the test case of varying design time to runtime ratio. Orion takes highest response time. As it takes coarse granularity for both types of transactions, its response time is very high. Jun2000's scheme is better than Orion because it provides fine granularity for runtime transactions and medium granularity for design time transactions. However CESMGL scheme performs the best. This is because of fine granularity of subclass lattice level operations with the help of CDV. In the graph, it can be noted that Jun2000's scheme response time is lower than CESMGL scheme for higher ratio of runtime transactions. This is because CESMGL scheme blocks inconsistent runtime transactions while it is not checked in Jun2000's scheme.

ANOVA (Analysis of Variants) is used to compare the performance of the three techniques as given below. Here it is assumed that the response variable 'Response Time' is affected by three factors namely, 'number of transactions' (represented as *numtran*), 'type of transaction' (*Dtorratio*) and 'technique' (*techniqname*). The response times are noted for 100, 400 and 700 transactions to study the behavior of the techniques for small to fairly large number of transactions. The transactions could be runtime transactions or design time transactions. There are three subtypes of design time transactions namely changes to node contents, nodes and links as mentioned in Bannerjee1987. Out of the total number of transactions five ratios of design time to runtime transactions are taken to study the behavior of the techniques namely 100:0, 75:25, 50:50, 25:75 and 0:100. In each of the ratios, the sub types of design time operations namely node content level operations, node level operations and link level operations are considered. The response times are taken for each subtype of design time transactions for all ratios. The techniques considered are Orion, Jun and Proposed called as CESGML. For each experimental combination of the three factors, two replications have been carried out.

The significance of the differences between the treatments of each of the components of the model is tested using 15 null hypotheses by taking combinations of the above mentioned four factors. As the objective is to show that CESGML performs better than the other two existing techniques, only one hypothesis is considered here.

H_{01}: There is no significant difference in response time by using different techniques.

As the significant ratio is less than 0.05, the null hypotheses are rejected with 95% confidence level. This means that there is a significant difference in response time due to change in the number of transactions, design time to runtime ratio and due to different techniques.

The null hypothesis H_{01} for the techniques is rejected. Since there is a significant difference between different algorithms, second level of test is conducted called as Duncan multiple range test to discriminate the differences among the techniques.

Table 3.8 ANOVA results

Tests of Between-Subjects Effects

Dependent Variable:responsetime

Source	Type III Sum of Squares	Degrees of Freedom	Mean Square	F	**Significant Ratio**
Model	1.167E6	108	10802.483	2012.568	**.000**
Numtran	264314.827	2	132157.413	24621.722	**.000**
Techniqname	117371.494	2	58685.747	10933.508	**.000**
Dtorratio	17596.878	3	5865.626	1092.801	**.000**
Dtrtype	3833.689	2	1916.845	357.120	**.000**
numtran * techniqname	38143.284	4	9535.821	1776.581	**.000**
numtran * dtorratio	11855.364	6	1975.894	368.121	**.000**
numtran * dtrtype	1764.414	4	441.104	82.180	**.000**
techniqname * dtorratio	1088.747	6	181.458	33.807	**.000**
techniqname * dtrtype	6346.830	4	1586.707	295.613	**.000**
dtorratio * dtrtype	2733.085	6	455.514	84.865	**.000**
numtran * techniqname * dtorratio	431.161	12	35.930	6.694	**.000**
numtran * techniqname * dtrtype	2026.393	8	253.299	47.191	**.000**
numtran * dtorratio * dtrtype	4111.168	12	342.597	63.828	**.000**
techniqname * dtorratio * dtrtype	793.496	12	66.125	12.319	**.000**
numtran * techniqname * dtorratio * dtrtype	1319.259	24	54.969	10.241	**.000**
Error	579.691	108	5.368		
Total	1167247.845	216			

a. R Squared = 1.000 (Adjusted R Squared = .999)

Table 3.9 Duncan Range test results

Multiple Comparisons - Dependent Variable response time

						95% Confidence Interval	
	(I) technique name	(J) technique name	Mean Difference (I-J)	Std. Error	Significant Ratio	Lower Bound	Upper Bound
Duncan	CESGML	Jun	8.3555*	.38613	**.000**	7.4165	9.2945
		Orion	53.0948*	.38613	**.000**	52.1558	54.0338
	Jun	CESGML	-8.3555*	.38613	**.000**	-9.2945	-7.4165
		Orion	44.7393*	.38613	**.000**	43.8003	45.6783
	Orion	CESGML	-53.0948*	.38613	**.000**	-54.0338	-52.1558
		Orion	-44.7393*	.38613	**.000**	-45.6783	-43.8003

Based on observed means. The error term is Mean Square(Error) = 5.368.
* The mean difference is significant at the .05 level

The results of Duncan Range test is given in table 3.9. In table 3.9, it can be observed that there is a significant improvement in the response time by each of the techniques as the significant ratio is still in 0.000 which is less than 0.05.

The tests are conducted using SPSS17.0 and the estimated means of response time with respect to each of the technique is plotted as in figure 3.8. It is found that technique 3 (CESGML) has the lowest response time when compared to 1(ORION) and 2 (JUN).

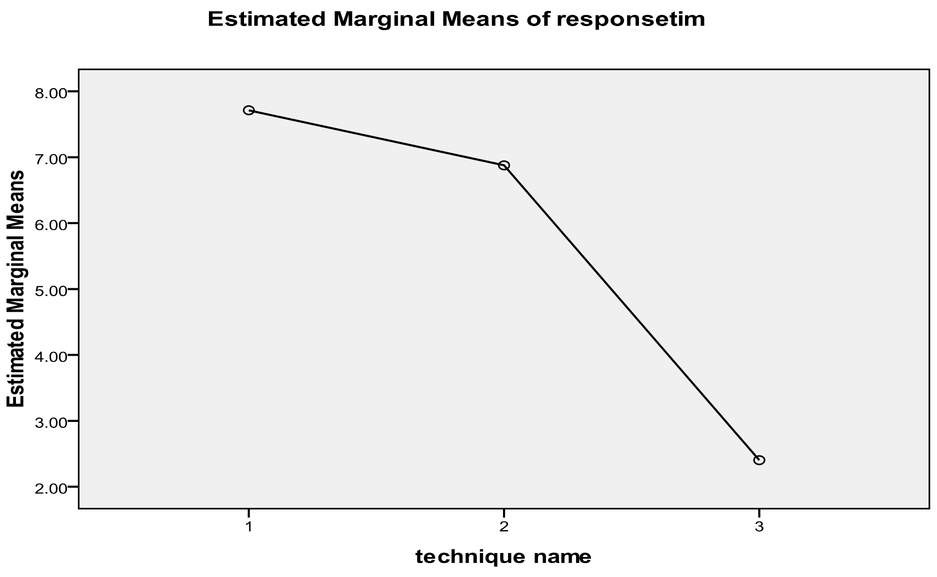


Figure 3.8 Plot of technique name versus response time

3.4 Summary

In this chapter, a concurrency control scheme is proposed based on multiple granularity lock model to provide fine granularity among design time transactions and runtime transactions. The CESMGL scheme imposes concurrency control on the write-to-read conflicts and write-to-write conflicts between classes related by inheritance and aggregation for both design time as well as runtime accesses. It minimizes the need for AAV and MAV used in Jun2000's scheme by proposing rich set of lock modes based on the semantics of object oriented concepts. Fine granularity of lock modes modifying the class relationships is proposed by defining CDV and splitting the lock modes separately for class level changes and class lattice level changes. The objective of this model is to provide finest granularity on all lock modes to provide highest concurrency, however with the overhead of maintaining AAV, MAV and CDV.

CHAPTER 4

SEMANTIC MULTI-GRANULAR LOCK MODELS FOR OBJECT ORIENTED DATABASES IMPLEMENTING CONTINUOUSLY EVOLVING DOMAINS

4.1 Preamble

Semantic MGLM techniques for OODBMS can be categorized into semantic MGLM based on compatibility of relationships and semantic MGLM based on commutativity of operations. The limitations of semantic MGLM based on compatibility of relationships are as follows:

- Coarse granularity
- Does not support complex relationships combining inheritance, aggregation and association.
- Same lock mode for all types of operations.

The limitations in semantic MGLM based on commutativity of operations are as follows:

- Inconsistency due to runtime transactions
- Coarse granularity of design time transactions
- Checks only intra-class dependencies, does not explore inter-class dependencies.

Among the two types of semantic MGLM, MGLM based on commutativity of operations showed better performance. The limitations of semantic MGLM based on commutativity of operations are eliminated in CESMGL. It ensures semantic consistency between classes where attributes and method or member functions are defined and where they are used. This was overlooked in Jun2000. Concurrency is further enhanced by defining CDV which is used to lock only the related sub class lattices instead of locking the entire class lattice for operations involving node changes. CDV maintains a list of parent classes and child classes for every class. Parent classes are the classes from which the class is derived. Child classes are the classes which are derived from class. Thus CDV improved the performance further. In CESMGL, separate lock modes have been defined to handle changes to nodes and edges. It provides fine granularity with the help of commutativity matrix and access vectors. Access vectors can be omitted at the cost of limited concurrency. Hence there is a trade-off between limited concurrency and access vectors maintenance overhead.

CESMGL perform better for the stable domains. In stable domains, the users make frequent runtime transactions to access the data. The design time transactions are few and are far in-between. However, in the case of continuously evolving domains, the class lattice has to be changed frequently. Then, more number of design time transactions will arrive along with runtime transactions. The design time operations may need to lock the data items in different granule sizes.

Let us take the following sample scenarios to highlight the complexity involved in the existing scheme. If a design time transaction needs to modify the signature or implementation of a method in a class, it needs to block other design time transactions that are trying to modify the definitions of attributes used in that method. The compatibility matrix provides this semantic concurrency control. MAV locks the method in 'X' mode so that it is not shared. AAV is used to lock the associated attributes used in the method. The existing scheme needs access to both MAV and AAV to provide better concurrency. This involves access overhead, search overhead and updation overhead of two access vector lists.

If a design time transaction modifies the class definition, it needs to lock the class definition in 'X' mode. Locking the class also involves locking its attributes and methods using AAV and MAV. In order to preserve consistency, all the classes that are related to this class inferred from CDV, also need to be locked. Then the lock status of all the attributes and methods of all the relevant classes are also to be updated in the AAV and MAV.

If the design time transaction wishes to change the relationship between two classes, then CDV is also to be accessed along with AAV and MAV.

In all the above mentioned cases, the DAV of all the classes involved in design time changes are to be modified and runtime transactions are to be blocked until the DAV are updated. In order to handle a single request, all the access vector lists are accessed. Because of this, the maintenance overhead is increased.

Though the use of DAV, AAV, MAV and CDV provide higher concurrency, they have the following limitations:

1. Prior knowledge of the structure of the class is required.

2.. The access vectors are to be updated every time the schema is changed due to a design time transaction which involves maintenance overhead for continuously evolving domains.
3. The search overhead is also involved in searching the lock status of the data item requested by the transactions as it needs to search the entire list to read or update the lock status of a data item.

This introduced the need for a new concurrency control scheme to support continuously evolving systems with nil or less overhead.

Two solutions are proposed to handle the overheads namely Semantic MGLM using access control lists and Semantic MGLM using lock rippling. The proposed schemes have these advantages: It is based on multi-granularity locking. As the schemes do not use any access vectors, they do not have any overhead of updating them every time the schema is changed. They do not need prior knowledge of the structure of objects. Further the proposed scheme provides same parallelism between design time transactions and runtime transactions as in existing schemes, without any extra cost.

Semantic MGLM using access control lists is explained in the next section and Semantic MGLM using lock rippling is explained in section 4.3.

4.2 Semantic MGLM using Access Control Lists

The proposed scheme provides the same level of concurrency as in table 3.5 and 3.6 without the limitations of access vectors, by splitting the lock table into three lists namely *Available, Shared and Exclusive* lists. This eliminates the need for maintaining vector tables along with lock table. The maintenance overhead is minimized as access vectors are not needed. The search time is minimized as the lock table is split into three lists. Any transaction requires search to only one of these lists instead of all of them. This reduces the search overhead to roughly about one third.

In *Available list,* the attributes and methods of each class that are currently available are included. In *Shared list,* the attributes and methods of each class that are currently in shared (read) lock mode are included. In *Exclusive list,* the attributes and methods of each class that are currently in exclusive (write) lock mode are included.

The format of *Available, Shared and Exclusive* lists are given in figures 4.1 and 4.2. During 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 attribute has two facets namely attribute definition and attribute value and is chosen depending on the type of transaction.

Runtime transactions lock the methods in read mode as their contents are not modified by execution. Design time transactions read or modify the method definitions. When any attribute or method definition is modified, runtime transactions accessing them should be blocked. *RType* (resource type) can be attribute definition, attribute value or method definition. *RType* - attribute definition is used by design time transactions. *RType* - attribute value is accessed by runtime transactions. So, two entries are maintained for every attribute. The objective for maintaining two entries is to allow concurrent reading of attribute definition while attribute value is read or modified. *Resource ID* holds the name of the attribute or method. *Class ID* is used to distinguish attributes or methods whose names are used in more than one class. Shared list maintains one more field called *Refcount* to maintain the number of transactions sharing the resource.

Initially, the attributes and methods of all the classes are included in the *Available* list. As the transactions arrive, they are checked for compatibility of locks in the compatibility matrix. If the locks are compatible, the requested resources are added to either *Shared* or *Exclusive* list, depending on the lock mode. Though all the resources are in the *Available* list in the beginning, eventually they will be distributed to other lists depending on the lock type requested by the arriving transactions.

In order to save search time, list search policies given in figure 4.3 are used. Exclusive lock mode is allowed for a requested resource only if the resource is currently present in *Available* list. Shared lock mode is allowed, only if the resource is not in *Exclusive* list. It is allowed when the resource is in *Available* list or *Shared* list. Searching the resource in both lists is time consuming. So if it is not available in *Exclusive* list, grant message can be sent and resource can be accordingly updated from *Available* list or *Shared* list in the background. Several transactions can share a resource in read mode. *Refcount* field in *Shared* list is used to count the number of transactions that are currently sharing the resource. First grant message adds the resource to the *Shared* list and sets the *Refcount* to 1. Every grant message after that, increments the count. Every release message decrements it. When the *Refcount* is 0, the resource is removed and added to the *Available* list.

Class ID	RType	Resource ID	Ref count

Figure 4.1 Format of *Shared* list

Class ID	RType	Resource ID

Figure 4.2 Format of *Available* and *Exclusive* lists

Requested Lock mode	List to be Searched
Exclusive (X)	Available list
Shared (S)	Exclusive list

Figure 4.3 List search policies for S and X lock modes

Let us now see the working principle of the proposed scheme for runtime and design time transactions.

The runtime transactions request resources by giving the *class ID* and *method ID*. The attributes used along with their lock mode in every method can be documented using document tools like JavaDoc or DocC++. From this, the attributes to be locked can be deduced by preprocessing. As mentioned earlier, the methods are locked in read mode for runtime transactions. So the requested *method ID* is removed from the *Available* list and added to *Shared* list. Several runtime transactions executing the same method can share the implementation. The value entries for all the *attribute ID*s used in the method are added to *Shared* list if they are input parameters, and to *Exclusive* list if they are not. In runtime transaction, only the *attribute value* is read or modified. So, entry with *RType-attribute value* is removed from *Available list* and added to *Shared or Exclusive* list. The granularity of runtime transaction is attribute level which is the fine granularity provided in the existing schemes.

The design time transactions usually read or modify the schema. As mentioned in Bannerjee1987, the design time transactions can be one of these three types: Changes to node (class), Changes to class contents (attributes and methods) and Changes to edges (relationships and position). The design time transactions are handled as below.

The smallest granularity supported for design time transactions is the attribute. Attribute definitions can be read or modified. Runtime transactions on an attribute (IA) can be executed in parallel with read attribute definition (RA/RAA) design time transactions. RA lock mode uses *RType – attribute definition* entry. When RA and

RAA lock modes are requested, corresponding attribute definition entry is added to the *Shared* list based on the policy mentioned above.

When MA lock mode is requested, parallel runtime transactions involving this attribute should not be allowed to maintain consistency. So, both definition entry and value entry of the attribute should be included in *Exclusive list*. If only definition entry is available, it implies that the attribute is currently being used in runtime transaction. Then, it has to wait until both the entries are available in *Available list.*

When lock modes RAMS, RMS, MMS, RAMI, RMI, MMI, MAMI are requested on a method, the search policies as mentioned above are followed to update access control lists. When new attributes or methods are added in a class using lock modes AA and AM, they are added to the *Available list*. The arriving transactions can request for this attribute or method only after this insertion. When existing attributes or methods are deleted using DA and DM lock modes, they can be removed only when they are in the *Available list* to preserve consistency. This is ensured in the commutativity matrices in table 3.3, 3.4, 3.5 and 3.6. Changes involving nodes and edges involve removing all the attributes and methods of the related classes from *Available list* and adding them to *Exclusive list*. They should be serialized to maintain the semantic consistency of the database.

4.3 Semantic MGLM using Lock Rippling

The schema of OODBMS is represented as a class diagram. The class diagram is a collection of classes related by inheritance, aggregation and association relationships. Group of classes related by inheritance (excluding *multiple inheritance*) is called as *class hierarchy*. It is represented as DAG. Group of classes related by a combination of all types of relationships mentioned above is called *class lattice*. Then the class diagram can be viewed as a class lattice and represented as Directed Graph (DG). In DG, cycles are possible due to multiple inheritance, shared aggregation and association. The classes are viewed as nodes and the relationship links connecting classes are viewed as edges. The design time transactions can do changes to schema in three ways as specified in Kim1989 and Bannerjee1987. The schema operations are categorized into changes to the contents of a node, changes to nodes and changes to edges.

From the above group of operations, certain semantic aspects can be inferred. During runtime transactions, the values of attributes are read or modified by executing the associated method 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, attribute has two facets and they are 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 by a design time transaction, runtime transactions accessing them should not be allowed.

A runtime transaction can have attribute, instance or class level of granularity. It is based on the property of the method as to whether the method is *primitive* or *composed* and *instance* or *class* level as defined by Reihle2000a.

Further, it is pointed out in CESMGL that when runtime transaction requests a base class object, it is enough to lock only the base class object. However, when a runtime transaction requests a sub class object, it is required to lock the associated base class object that access the same record also to preserve the database consistency. Thus, there is an upward dependency from the sub classes to the base class. This is applicable to aggregation and association also. In aggregation, the component objects are locked along with composite objects. In association, associative objects are dependent on associated objects and thus needs to be locked.

The operations allowed during schema changes as mentioned in Bannerjee1987 can be grouped into five categories. They are:

1. Add a new attribute/ method/ instance/ class/ edge (relationship).
2. Delete an existing attribute/ method/ instance/ class/ edge (relationship).
3. Modify the definitions, values or implementation of attribute/ method/ instance/ class/ edge (relationship).
4. Read the definitions, values or implementation of attribute/ method/ instance/ class/ edge (relationship).
5. Move an existing attribute/ method/ instance/ class/ edge (relationship) from one location to another location.

From the above categories of operations, their dependency on creation, deletion and modification can be inferred. The attributes and methods can be added only to an existing class. Similarly instances can be created only after defining a new class and adding its attributes and methods. New relationships can be established only among existing classes. When a new class is added, until it is related to the existing classes by a relationship edge, it can be in parallel done with any other schema change without affecting consistency. Once the new class is included in the already existing class lattice as a base class or component class or associative class by adding an edge, then its attributes and methods have to be included in its subclasses, composite classes or associated classes respectively. Then it can be inferred that the possible granularities for the addition operation is class level or sub class lattice level (group of related classes).

A class can be deleted only after deleting its attributes, methods and instances. If the class is related to other classes by inheritance or aggregation or association, its attributes and methods are to be deleted from them. Any other schema change in the related sub class lattice should not be allowed when deletion of a class is done. If the class is a base class or component class or associative class, then its attributes and methods have to be deleted in its subclasses, composite classes or associated classes respectively. For example in figure 3.3a, the attributes in A inherited into E and H are to be deleted, while deleting A.

When a transaction reads the definitions of attribute/ method/ instance/ class/ relationship (edge), it can be done in parallel with all other read operations. It can also be noted that when an attribute definition is read, its value can be modified by a runtime transaction. When a design time transaction tries to move an attribute/ method/ instance/ class/ relationship (edge), then the whole class diagram has to be locked in exclusive mode. This is because the move operation may involve the entire class lattice as the destination is unpredictable.

The proposed scheme is based on the semantics mentioned above. The class diagram is represented as Bi-directed Graph (BG). In Directed Graph (DG), the edges flow from independent classes or parent classes (base classes, component classes and associated classes) to dependent classes or child classes (sub classes, composite classes and associative classes). It is unidirectional. The children can be reached from parent, but the reverse is not possible. In BG, the edges link both ways. This is needed for the reduction of search time and for lock rippling.

Table 4.1 shows the proposed compatibility matrix for runtime and design time transactions. The lock modes have been defined based on the inferences mentioned above. The semantics of the lock modes are as given below.

Table 4.1 Compatibility matrix for runtime and design time transactions

	RRA	**RMA**	**MDD**	**RDD**	**AE**	**DE**	**MCL**
RRA	Y	N	N	Y	Y	N	N
RMA	N	N	N	Y	Y	N	N
MD	N	N	N	N	Y	N	N
RD	Y	Y	N	Y	Y	N	N
AE	Y	Y	Y	Y	Y	N	N
DE	N	N	N	N	N	N	N
MCL	N	N	N	N	N	N	N

1. **RRA** – Runtime Read Access- Read the values of attributes as a runtime transaction.
2. **RMA**- Runtime Modify Access - Modify values of the attributes as a runtime transaction.
3. **RDD-** Read Domain Definition – Read the domain and name of attribute/ instance/ class/ relationship (edge or link) and method. Modifying a method includes modifying interface- name of the method, input arguments and output arguments and implementation.
4. **MDD-** Modify Domain Definition – Read the domain and name of attribute/ instance/ class/ relationship. Read the interface and implementation of a method in a class.
5. **AE –** Add Entity – Add a new attribute/ method/ instance/ class/ edge (relationship).In the case of adding edges, a new relationship is defined between two existing classes.
6. **DE –** Delete Entity – Delete an existing attribute/ method/ instance/ class/ edge (relationship).
7**. MCL** – Move Class Lattice – This lock mode is used to move the entities namely attribute/ method/ instance/ class/ edge (relationship) from one position to another position in the lattice.

The proposed scheme allows locking from the root node to the leaf node for design time transactions and from the leaf node to the root node locking for runtime transactions. The procedure for locking can be summarized as follows:

- Check for lock compatibility.

- If the lock modes are compatible, set the lock on the resource at the requested lock mode.
- Ripple the locks from root to leaf, if it is a design time transaction.
- Ripple the locks from leaf to root, if it is a runtime transaction.
- Release the locks in the reverse order after release request.

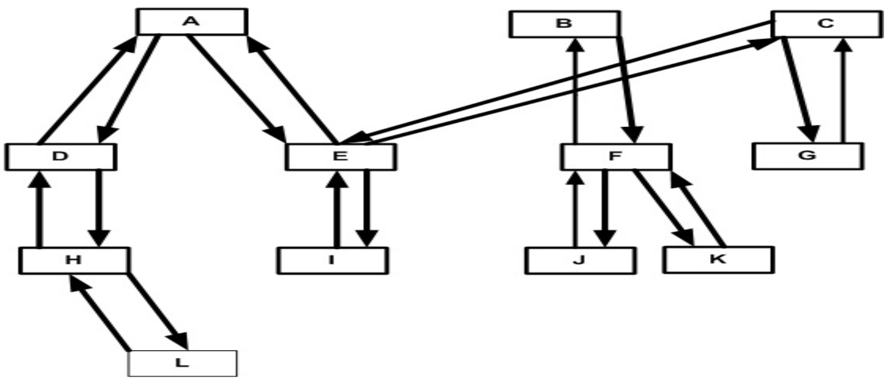


Figure 4.4 Sample class diagram 3.3a represented as Bi-directed graph

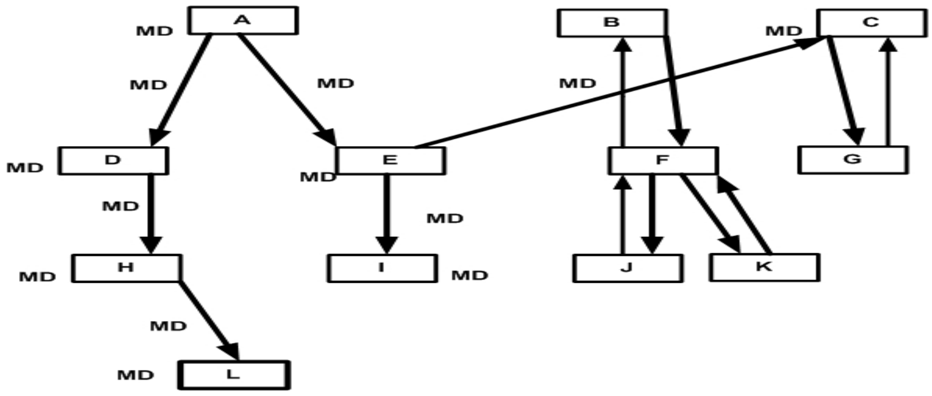


Figure 4.5 Lock rippling to lock associated children classes on a Modify Definition (MD) request to class A

Figure 4.4 shows the sample class diagram (schema) in figure 3.3a represented as BG. In the proposed scheme, when a design time transaction is made for a base class, the requested lock mode is set on the class. Then the lock is rippled to all its children including the edges. When a change is made on the definition of an attribute/ method/ instance/ relationship or the class itself, all the classes related to this class (called sub class lattice), should also be locked in the same lock mode to maintain the consistency as mentioned in section 2.2. Figure 4.5 shows the rippling of lock, when a Modify

Definition (MD) request is made to class A. In the case of design time transactions, the locks are rippled downwards from parent to children.

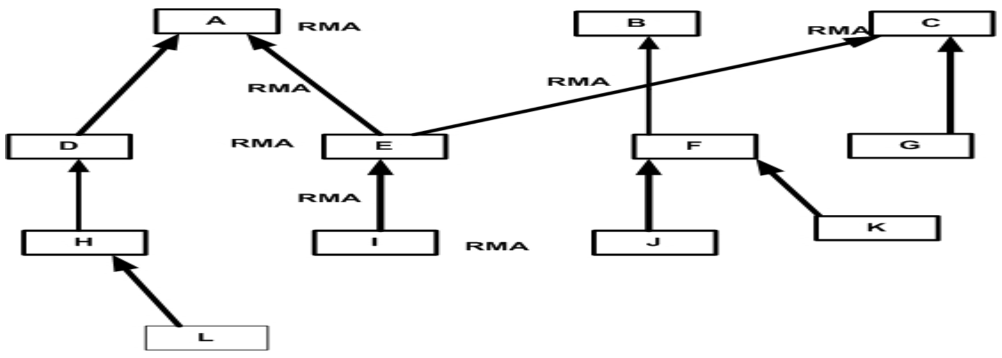


Figure 4.6 Lock rippling to lock associated parent classes on Runtime Modify Access (RMA) request to class I

Similarly when a runtime transaction transactions to modify the attribute values (RMA) of an object in a subclass, its associated objects in parent classes are also locked by lock rippling. In the case of runtime transactions, the locks are rippled upwards from child to all its parents.

Figure 4.6 shows the rippling of runtime transaction lock to parent class using the upward links. Let a runtime transaction request for class I. Note that A, E and C are the parents of class I. It can also be noted that the edges are also locked to block any request to change the relationship between these classes. This eliminates the problem of setting intension locks for multiple inheritance in ORION scheme [Garza1988]. Intension locks are always set in ORION scheme from root to leaf. ORION scheme can lock the classes A, E and I along the path. However, it will not lock C, which has to be locked to preserve consistency.

It is also worth noting that if runtime transactions request for base classes, component classes and associative classes, the locking will be only on the object of that class, as it will not have any upward edges. For example, in the sample class diagram, classes A, B and C are parent classes that do not have any parents. Hence, these classes alone are locked. Similarly, if child classes are requested for design time transactions they alone are locked, as they will not have any downward edges. For example consider the classes L, I, J and K in the sample class diagram.

It is already discussed that the request for moving an entity (ME) requires locking the entire class lattice. The other operations can be parallelized in different sub class lattices. Hence, concurrency is improved while ensuring consistency wherever necessary.

4.4. Experimental Results and Discussion

In order to evaluate the proposed schemes in general environment, the simulation model by Kim1991 in section 3.3 is adapted here and experiments are conducted. The same simulation model is adapted here so that the proposed schemes can be compared with CESGML under the same environment in which CESGML was tested. 007 benchmark by Carey1993 is extended here also and the same simulation parameters are used here.

Figure 4.7 shows the performance of the proposed schemes against CESMGL. CESMGL provides best performance for stable domains. So test scenarios are chosen to see their performance, when there is an increasing degree of design time transactions. The performance is tested by varying the design time transaction to runtime transaction ratio. CESMGL performs the worst. This is because fine concurrency is possible only with access vectors. Access vectors involve maintenance overhead for continuously evolving domains.

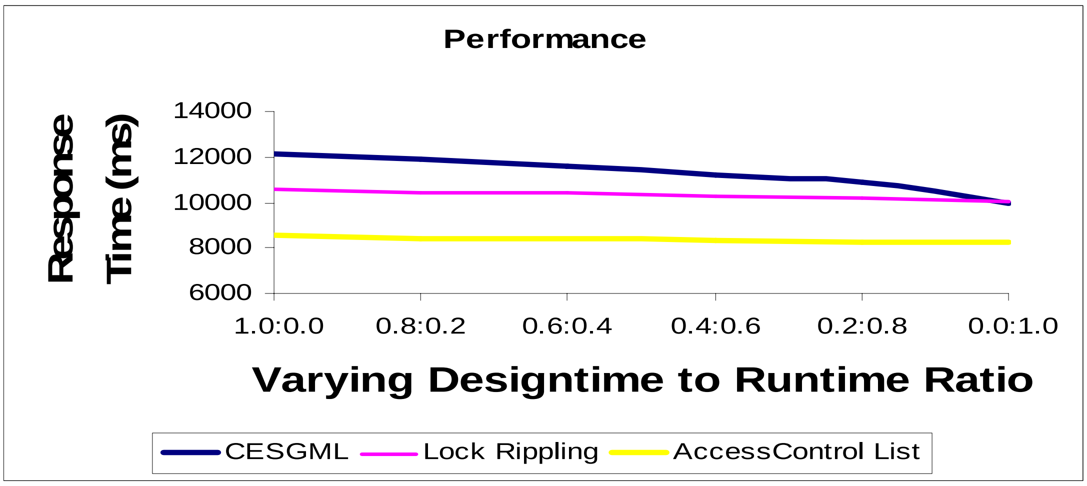


Figure 4.7 Performance varying design time transaction to runtime transaction ratio

The results show that the performances of all the schemes are approximately same for stable domains where roughly 100% of the transactions are runtime transactions. But as the ratio of design time transactions increase (implies evolving domains), the performance of CESMGL deteriorates. The reason is that, as the ratio of design time transaction increases, the time taken to update the access vectors increase.

As a result, response time increases for the CESGML scheme. In the proposed schemes, it can be observed that the response time is relatively constant because there is no need for updating access control lists. The proposed schemes give almost the same response time for all combinations of design time transactions and runtime transactions. Semantic MGLM using lock rippling is better than CESGML by 14%. Semantic MGLM using access control list is better than CESGML by 21%. The response time for lock rippling is more because of the coarse granularity of lock modes for runtime transactions. However they do not need any apriori knowledge of object structure. Thus, the proposed schemes perform better for continuously evolving domains.

4.5. Summary

In this chapter, two multi-granular lock models namely MGLM using lock rippling and MGLM using access control lists are proposed for OODBMS implementing continuously evolving domains. MGLM using lock rippling uses object semantics to define lock modes by combining intension locks and commutativity of operations. MGLM using access control lists reduces the search overhead and maintenance overhead by defining search policies and reducing the number and size of tables used. They both eliminate the overhead for access vectors used by the existing semantic MGLM to provide high concurrency. They also eliminate the requirement of apriori knowledge about the domain structure of all the classes used to implement the domain.

CHAPTER 5

SEMANTIC MULTI-GRANULAR LOCK MODEL FOR OBJECT ORIENTED DISTRIBUTED SYSTEMS

5.1 Preamble

OODBMS provides better complex data modeling support for the newly emerging distributed applications than relational databases. OODBMS is used as persistent data store for distributed systems. It resides in the database tier. However in distributed systems, the server tier is implemented as procedures as in the figure 2.1. This requires a conversion between procedural paradigm to object oriented paradigm and vice versa for all the communications between server tier and data base tier. Further each of the database models in distributed systems has its own concurrency control mechanisms that cannot be adapted to any of the other models. The concurrency control policies are defined only for the database tier and the server tier has no control over the concurrency control. This introduces a restriction of using only the refined persistent data store for the domain data like database management systems. Primitive data stores like files are not supported in distributed systems and hence they cannot be reused.

The above limitations can be over come in OODS. In OODS, the server tier is also implemented as objects as in the figure 2.2. So the conversion of data format between server tier and data base tier is not necessary, if OODBMS provides persistent data storage. However conversion of data format is still required, if other database models are used. Hence the possibility of providing a common concurrency control mechanism that is independent of the persistent data store type is explored by shifting the concurrency control from database tier to server tier.

In OODS, objects are the reusable data sources. The clients can access the data from the data store in database tier only through the objects in the server tier. Hence a common concurrency control mechanism can be defined for the objects in the server tier. The other advantage of this shift in the concurrency control to the server tier allows usage of legacy data stores.

Already semantic concurrency control mechanisms have been proposed for object oriented data bases by exploiting the object oriented paradigm features. They out-perform the conventional concurrency control mechanisms for OODBMS. Hence the feasibility of extending the same mechanisms to OODS may be analyzed.

OODS support continously evolving domains in which the services are frequently enhanced to provide better client support. Hence both runtime and design time transactions are to be supported.

Though concurrency control mechanisms in OODBMS can be considered, they cannot be extended as they are in OODS. This is because query languages are used to access databases. But in OODS, object oriented programming languages like C++, Java are used to make client requests. Then lock types and granularities of resource are to be ascertained from the client code. The doc tools like docC++, Javadoc can be used for identifying the method type and properties. Following this, the compatibility matrix used in OODBMS can be considered for adoption in OODS.

In OODBMS, lock modes are defined only for concrete classes. The lock modes for abstract classes are not ascertained. The compatibility matrices for inheritance and aggregation are defined for OODBMS. To use those matrices, lock types and granule sizes are to be determined for inheritance and aggregation (composition) using the classification of class method types and properties proposed by [Riehle2000a;Riehle2000b]. Association is one the important relationships frequently used to relate objects participating in a domain. Wu1998 have proposed directed graph based association algebra for query processing and optimization of objects in object oriented databases. But so far, the types, properties and attributes of association have not been explored for their probable impact on concurrency or concurrency control. The lock modes, granule sizes and lock compatibility for association have not been explored so far.

In the following section, a semantic concurrency control technique is proposed for object oriented distributed systems. It is done in two steps namely (1) defining lock types and granularity for all types of classes and their relationships (2) extending the compatibility matrix defined for OODBMS to OODS. Section 5.3 concludes the chapter.

5.2 Defining Lock Types and Granularity for Classes and their Relationships

5.2.1 Defining lock types and granularity for attributes and classes

In OODBMS, only instance level attributes are referred. The scope of the 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 the 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 ‘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 only a class.Table 5.2a gives their granularity.

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. So at runtime, they should be locked only in S (read) mode. This is because the concrete classes that are inherited from this abstract class would be reading them. As they do not create instances (objects) and thereby do not affect the state of the system. However at design time, modifications may be done to the abstract class by inserting new methods or attributes, modifying the signature of the existing methods or modifying the data types of the attributes or deleting one or more attributes and/or methods. Hence the design time clients must be allowed to lock the abstract class by both S (read) and X (write) locks. It is also worth noting that the smallest accessible granule of abstract class is a class.

Table 5.1 Lock types for types of classes

Class type	Lock type (Design time)	Lock type (Runtime)
Abstract class	S/X	S
Concrete class	S/X	S/X

Concrete classes are classes defined mainly to create instances. They have all types of methods to create, query, mutate and delete objects. So the locks on concrete classes depend on the type of method which is invoked. So both S and X locks must be available for them at both design time as well as runtime. [Gray1976;Garza1987; Kim1990a; Lee1996;Jun2000] address only concrete classes. Their granularity can be as small as attribute [Jun2000]. Table 5.1 summarizes the lock types allowed for the types of classes at design time and run time and table 5.2b summarizes their granularity.

Table 5.2a Granularity of attributes

Type of Attributes	Granularity
Instance level	Instance/Attribute
Class level	Class

Table 5.2 b Granularity of classes

Type of class	Granularity
Abstract class	Class hierarchy/Class
Concrete class	Class/ Instance/Attribute

5.2.2 Types and granularity of locks based on method types for Inheritance

Based on the definitions of the method types and its properties in section 2.2, the locks can be determined for inheritance relationship as given below. Gray1978 has defined the following types of lock modes for coarse and fine granules for relational databases as in table 1.2. It is extended to object oriented databases as follows.

Instance objects can have only S and X locks. The class objects can be locked in S, X, IS, IX and SIX modes. The semantics of these modes are defined below:

- An IS (Intention Share) lock on a class means that instances of the class are to be explicitly locked in S or X mode as necessary.
- An IX (Intention Exclusive) lock on a class means instances of the class will be explicitly locked in S or X mode as necessary.
- An S (Shared) lock on a class means that the class definition is locked in S mode, and all instances of the class are implicitly locked in S mode and thus are protected from any attempt to update them.
- An SIX (Shared Intention Exclusive) lock on a class implies that the class definition is locked in S mode, and all instances of the class are implicitly locked in S mode and instances to be updated (by the transaction holding the SIX lock)will be explicitly locked in X mode.
- An X (Exclusive) lock on a class means that the class definition and all instances of the class may be read or updated.

Table 5.3 defines the locks based on the types of object methods. The types of locks are also based on the class level / instance level/ attribute level of access. For class level methods, the class hierarchy is locked by intension locks and classes are locked by S or X locks. If it is instance method, then class is set by intension locks and its instances are locked by S or X locks. The objects are accessible only after their creation. Their accessibility ceases after destruction.

Table 5.3 Lock type based on method types of Inhertiance

METHOD TYPES		Class/Instance/ Attribute	Class hierarchy / Class/ Instance
Query method		S	IS
Mutation Method	Set/Initialization method	X	IX
	Command method	S& X	SIX
Helper Method	Factory method	X	IX
	Assertion method	S	IS

Table 5.4. Lock granularity based on method properties in Inheritance

	Instance method				Class method			
	Primitive method	Composed method	Template method	Hook Method	Primitive method	Composed method	Template method	Hook method
Query method	Attribute	Object	Class hierarchy	Class	Class	Class	Class hierarchy	Class
Mutation method	Attribute	Object	Class hierarchy	Class	Class	Class	Class hierarchy	Class
Helper method (factory method)	-	Object	Class hierarchy	Object	-	Class	Class hierarchy	Class

Table 5.4 defines the lockable granules for various methods based on their properties as below. By combining the types and properties of the methods, the lock type and lockable granule size can be deduced.

5.2.3 Types and granularity of lock based on method types for Aggregation

Aggregation is an object relationship. In order to maintain consistency, when a client requests a composite object, intension lock must be set on its class. Along with that, the component objects that constitute the composed object must also be set on intention object lock. These intention locks, while locking the particular object that constitute the composite objects, let other objects of the same class to be used by other clients. This improves concurrency. Aggregation may have exclusive or shared reference. Exclusive reference does not allow the component objects to be shared by other composite objects where as shared reference allows it. Further aggregation may be dependent or independent on component objects for creation and deletion i.e. in the case of dependent aggregation, the composite object can be created only after creating the component objects and it is destroyed when all its composite objects are destroyed.

Table 5.5 Lock type based on method types for Aggregation

METHOD TYPES		Aggregation Root Object/ Attribute	Aggregation Root Class/ Object	Exclusive Component Class/Object	Shared Component Class/Object
Query method		S	IS/ISO	ISO/ISA	ISOS/ISAS
Mutation method	Set method/ Initialization method	X	IX/IXO	IXO/IXA	IXOS/SIXAS
	Command method	S/X	SIX/SIXO	SIXO/SIXA	SIXOS/SIXAS
Helper method	Factory method	As per creation and deletion rule basd on dependent / independent aggregation			
	Assertion method	S	IS/ISO	ISO/ISA	ISOS/ISAS

In the case of independent aggregation, the life cycle of composite object is independent of its composite objects. Table 5.5 gives the types of locks based on method types for aggregation. It is followed by granularity of locks as in Table 5.6.

Table 5.6 Lock granularity for Aggregation or Composition

Class Type	Granularity of locks	
	Primitive method	Composed method
Primitive class	Component attribute	Component object
Non Primitive class	Composite object hierarchy	

5.2.4 Types and granularity of lock based on method types for Association

Table 5.7 Type of locks based on method types for Association

METHOD TYPES		Association Root Object/ Attribute	Association Root Class/ Object	Exclusive Associated Class/Object	Shared Associated Class/Object
Query method		S	IS/ISO	ISO/ISA	ISOS/ISAS
Mutation method	Set method / Initialization method	X	IX/IXO	IXO/IXA	IXOS/IXAS
	Command method	S/X	SIX/SIXO	SIXO/SIXA	SIXOS/SIXAS
Helper method	Factory method	As per creation and deletion rule basd on dependent / independent association			
	Assertion method	S	IS/ISO	ISO/ISA	ISOS/ISAS

Association is also an object relationship. In order to maintain consistency, when a client requests a associative object, intension lock must be set on its class . Further, the associated objects that constitute the associative object must also be set on intention object lock. These intention objects, while locking the particular object that constitute the associative objects, lets other objects of the same class to be used by other clients. This improves concurrency.

Association may have exclusive or shared reference. Exclusive reference does not allow the associated objects to be shared by other associative objects where as shared reference allows it; further association may be dependent or independent on associated objects for creation and deletion. i.e. in the case of dependent association, the associative object can be created only after creating associated objects and it is destroyed when all its associated objects are destroyed. In the case of independent association, the life cycle of associative object is independent of its associated objects. Association relationship also possess association hierarchy like aggregation hierarchy. Table 5.7 below gives the types of lock based on method types for association. It is followed by granularity of locks as in Table 5.8.

Table 5.8 Lock granularity for Association

Class ype	Granularity of locks	
	Primitive method	Composed method
Primitive class	Associated attribute	Associated object
NonPrimitive class	Associative object hierarchy	

5.2.5 Compatibility matrix for runtime transactions based on class relationships

The compatibility matrix specified in Garza1987 for inheritance is given in table 5.9. The inheritance can be classified as exclusive inheritance or shared inheritance. The inheritance types single inheritance, multilevel inheritance, multiple inheritance allow exclusive inheritance of a parent class to one or more child classes. But in hierarchical inheritance, several sub classes are inherited from the same parent class or th parent is shared by many siblings.

Table 5.9 Compatibility matrix for Inheritance [Garza1987]

	IS	IX	S	SIX	X
IS	Y	Y	Y	Y	N
IX	Y	Y	N	N	N
S	Y	N	Y	N	N
SIX	Y	N	N	N	N
X	N	N	N	N	N

If the compatibility matrix specified in Garza1987 is extended for this shared inheritance, then concurrency will be restricted. At any time, only one sub class is allowed to lock the parent class. Hence separate intension lock modes must be defined to increase concurrency. In the compatibility matrix below, separate lock modes need to be defined in shared and exclusive inheritance. Three more lock modes ISCS (Intension Shared Class Shared), IXCS (Intension Shared Exclusive Shared) and SIXCS (Shared Intension Exclusive Class Shared) can be defined to support shared inheritance. These new lock modes can be appended to the compatibility matrix in Garza1987 as in table 5.10. The figures 5.1a, 5.1b, 5.1c show the different types of shared and exclusive inheritance and the locking policy in each type of inheritance. Table 5.10 gives the revised compatibility matrix.

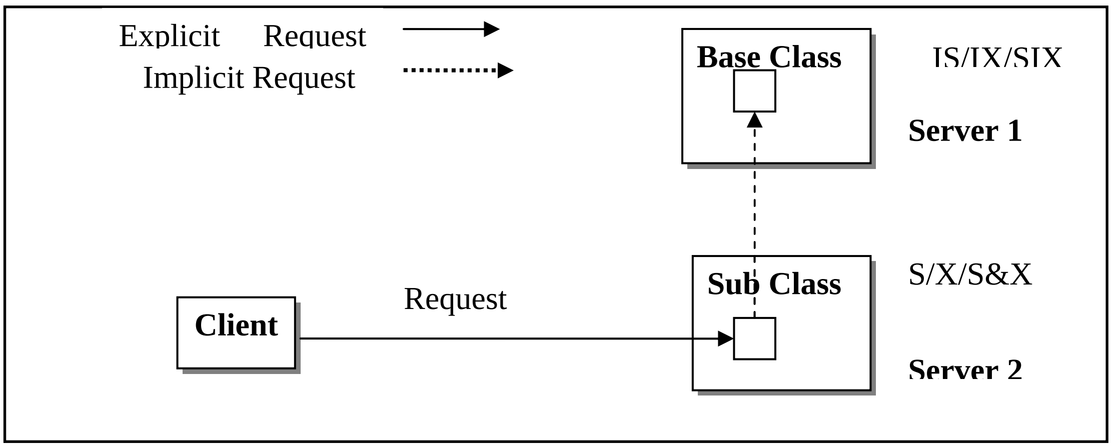


Figure. 5.1a. Locking in Single Inheritance

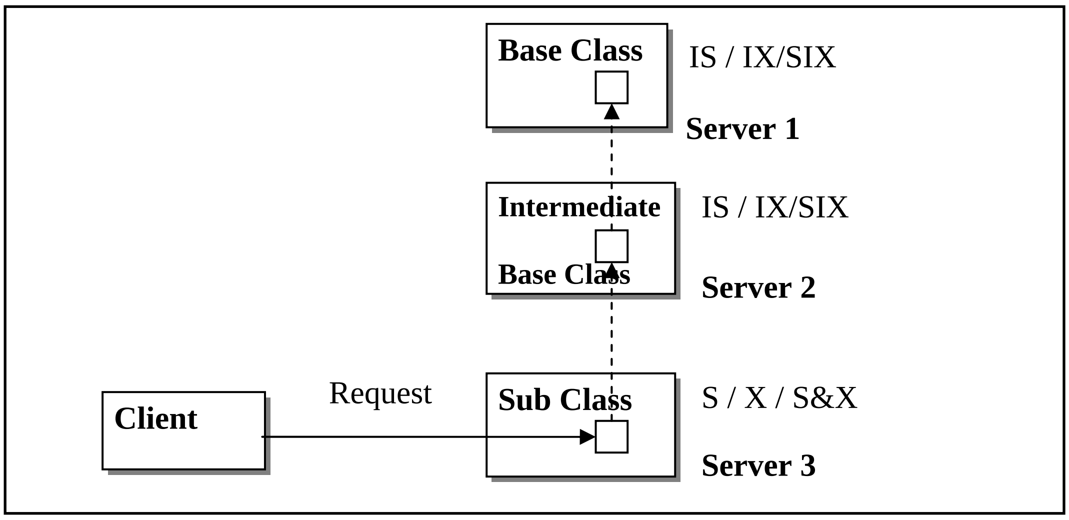


Figure. 5.1b. Locking in Multilevel Inheritance

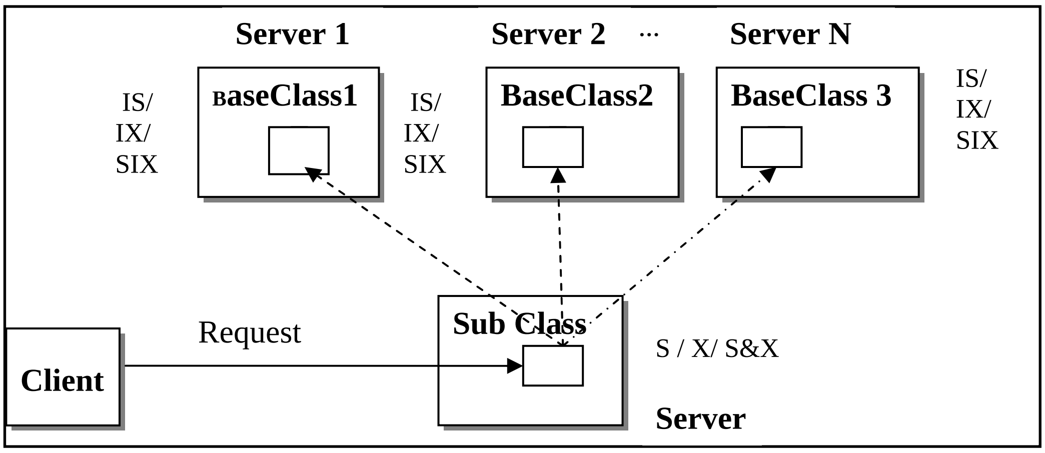


Figure 5.1c. Locking in Multiple Inheritance

Table 5.10 Revised compatibility matrix for Inheritance

	IS	ISCS	IX	IXCS	S	SIX	SIXCS	X
IS	Y	Y	Y	Y	Y	Y	Y	N
ISCS	Y	Y	Y	Y	Y	Y	Y	N
IX	Y	Y	Y	Y	N	N	N	N
IXCS	Y	Y	Y	Y	N	N	N	N
S	Y	Y	N	N	Y	N	N	N
SIX	Y	Y	N	N	N	N	N	N
SIXCS	Y	Y	N	N	N	N	N	N
X	N	N	N	N	N	N	N	N

Table 5.11 gives the compatibility matrix for aggregation extended from Kim1989. Kim1989 has defined compatibility matrix for aggregation by extending the lock modes defined for inheritance to aggregation. But its granularity size is restricted to object level. It is further extended to attribute level in the proposed scheme to improve the concurrency.

Table 5.11 Revised compatibility matrix for Aggregation and Association

	ISA	ISAS	IXA	S	IXAS	SIXA	SIXAS	X	ISO	IXO	SIXO	ISOS	IXOS	SIXOS
ISA	Y	Y	Y	Y	N	Y	Y	N	Y	N	N	Y	N	N
ISAS	Y	Y	Y	Y	N	Y	Y	N	Y	N	N	Y	N	N
IXA	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N
IXAS	N	N	N	N	N	N	N	N	Y	Y	N	N	N	N
S	Y	Y	N	Y	N	N	N	N	Y	N	N	Y	N	N
SIXA	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N
SIXAS	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N
X	N	N	N	N	N	N	N	N	N	N	N	N	N	N
ISO	Y	Y	N	Y	Y	N	N	N	Y	Y	Y	Y	Y	Y
IXO	N	N	N	N	Y	N	N	N	Y	Y	N	Y	Y	N
SIXO	N	N	N	N	N	N	N	N	Y	N	N	Y	N	N
ISOS	Y	Y	N	Y	N	N	N	N	Y	Y	Y	Y	N	N
IXOS	N	N	N	N	N	N	N	N	Y	Y	N	N	N	N
SIXOS	N	N	N	N	N	N	N	N	Y	N	N	N	N	N

Table 5.12 Compatibility matrix for runtime transactions

	IS	ISCS	IX	IXCS	S	SIX	SIXCS	X	ISO	IXO	SIXO	ISOS	IXOS	SIXOS	ISA	IXA	SIXA	ISAS	IXAS	SIXAS
IS	Y	Y	Y	Y	Y	Y	Y	N	Y	N	N	Y	N	N	Y	N	N	Y	N	N
ISCS	Y	Y	Y	Y	Y	Y	Y	N	Y	N	N	Y	N	N	Y	N	N	Y	N	N
IX	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
ISCS	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
S	Y	Y	N	N	Y	N	N	N	Y	N	N	Y	N	N	Y	N	N	Y	N	N
SIX	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
SIXCS	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
X	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
ISO	Y	Y	N	N	Y	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
IXO	N	N	N	N	N	N	N	N	Y	Y	N	Y	Y	N	Y	Y	N	Y	Y	N
SIXO	N	N	N	N	N	N	N	N	Y	N	N	Y	N	N	Y	N	N	Y	N	N
ISOS	Y	Y	N	N	Y	N	N	N	Y	Y	Y	Y	N	N	Y	Y	Y	Y	N	N
IXOS	N	N	N	N	N	N	N	N	Y	Y	N	N	N	N	Y	Y	N	N	N	N
SIXOS	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	Y	N	N	N	N	N
ISA	Y	Y	N	N	Y	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
IXA	N	N	N	N	N	N	N	N	Y	Y	N	Y	Y	N	Y	Y	N	Y	Y	N
SIXA	N	N	N	N	N	N	N	N	Y	N	N	Y	N	N	Y	N	N	Y	N	N
ISAS	Y	Y	N	N	Y	N	N	N	Y	Y	Y	Y	N	N	Y	Y	Y	Y	N	N
IXAS	N	N	N	N	N	N	N	N	Y	Y	N	N	N	N	Y	Y	N	N	N	N
SIXAS	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	Y	N	N	N	N	N

The compatibility matrix for association has to give separate lock modes for attribute level association and object level association. Association is also an object relationship like aggregation. As lock modes for object level locking and attribute level locking has already been defined for aggregation, it can also be extended to association. Hence it is same as the compatibility matrix for aggregation as given in table 5.11. The compatibility matrix of table 5.12 completely defines the semantics of all the lock modes for run time transactions. It combines the compatibility matrix defined for each relationship seperately as given in table 5.10 and 5.11.

5.2.6 Compatibility matrix for runtime and design time transactions

In OODBMS, fine level lock modes are also defined for design time operations. It is not possible to extend the same to OODS, because it has no schema and query language support. When any design time operations are performed, the code implementing the domain has to be changed. As it is very difficult to predict which part of the code is getting modified in OODS, coarse level locking is offered for design time operations in OODS.

Table 5.13 Revised compatibility matrix for design time transactions and runtime transactions

	RD	WD	RA
RD	Y	N	Y
WD	N	N	N
RA	Y	N	Depends on table 5.12

The schema locking defined in Lee1996 may be taken into account for design time transactions. Just as schemas are changed periodically, OODS can also provide improved services. This requires the updation of behavior defined by object methods. The lock modes can be called as RD(Read Definition), WD(Write Definition) and RA (Runtime Access). Then analogous to the schema locks RS and WS, compatibility matrix can be defined. The compatibility matrix for design time transactions and runtime transactions are defined in table 5.13.

5.3 Summary

The compatibility matrix mentioned in this chapter needs to be implemented. In OODBMS, the compatibility matrix is implemented as part of the DBMS. In OODS the domain is implemented using object oriented languages like java, c++ etc. Then it has to be implemented as operating system services or language constructs in programming languages say as an extended library of the language or as part of the

component itself. Providing concurrency control at operating system level is too complex. Providing it at language level is possible. Already Java,eiffel etc., offers such extended libraries for various services. Among all the solutions, implementing it as part of the component is much more feasible. Already, COM has set a precedence of managing the clients in a primitive way using reference counts. Active component approach called JADEX has been proposed in Braubach2011 in which the concurrency module is built as part of the component. They have provided primitive concurrency control mechanism to avoid dirty reads and writes. They have not exploited the semantics of object oriented paradigm. If the proposed compatibility matrix can be incorporated in such components the performance will improve.

This chapter proposes a semantic based concurrency control mechanism for object oriented distributed systems. It is based on multi granular lock model. The Compatibility matrix defines lock modes for objects based on the semantics of object oriented paradigm. It provides fine granularity for runtime data requests. But design time requests are still in coase level. Hence the future work will be to explore the possibility of providing fine granularity also for design time requests.

CHAPTER 6

DEADLOCK HANDLING TECHNIQUES FOR OBJECT ORIENTED DISTRIBUTED SYSTEMS

6.1 Preamble

In OODS, the reusable data sources i.e. databases are mapped onto the objects. Hence, objects are viewed as resources here. To preserve the consistency of the objects, concurrency control is applied on objects and as a consequence live locks and deadlocks might occur.

The structure of the object oriented system is defined using static structural diagram namely class diagram or class lattice. From the class diagram, objects participating in the system, their attributes, methods and relationships with other objects in the system can be inferred. The transactions call the methods of these objects to satisfy their request. Hence, the dependency among the transactions can be inferred from the class diagram. Deadlocks can be handled by prevention, avoidance and detection and resolution. Deadlock prevention is easier to implement than deadlock detection in distributed systems. Deadlock prevention using resource ordering is generic than other types of Deadlock Prevention Algorithm (DPA) as it does not impose any constraints on the nature of resources.

Hence, our objectives are namely: - proposing resource-ordering policy for objects and proposing a deadlock prevention algorithm using the proposed resource ordering policy for resources i.e. objects in OODS. As mentioned earlier, resource ordering technique is not new. The novelty in the proposed algorithm lies in defining the resource ordering policy by exploiting the dependency among objects participating in the domain. The dependency of the objects with other objects can be inferred by their relationship with other objects. The possible relationships between objects can be inheritance, association and aggregation. Section 2.2 shows the dependency existing among objects in inheritance, association and aggregation.

The proposed algorithm also alleviates the problem of starvation in poverty and starvation in wealth by framing a policy for access ordering of transactions rather than using only timestamp ordering. Hence, by combining resource ordering with transaction access ordering, the proposed scheme prevents deadlock as well as starvation.

Deadlock detection and resolution algorithms are also available for distributed systems. Probe based distributed deadlock detection and resolution algorithm is one of the popular DDDR algorithms widely used because of its simple mechanism. The initiator, usually one of the transactions involved in the circular wait, send probe messages along the dependency edges of the Global Wait-For-Graph. When the probe comes back to the initiator, it indicates the presence of deadlock. Then resolution phase is initiated to choose a victim transaction and abort it.

The Probe based DDDR algorithm has two limitations namely: It cannot work in faulty environment and it requires a separate resolution phase to identify a victim and abort it. If the probe does not come back to the initiator, it could be because of live lock or faulty environment. If it is due to live lock, then the initiator can wait for finite time before sending the request message. If it is due to faulty environment, the probe might have been lost and initiator will not be aware of it. It will assume that the delay is due to live lock and wait. Providing fault tolerance is not a function of DDDR. But the initiator must always know the status of the probe in all cases.

The victim transaction can be identified while sending the probe instead of spending time on resolution algorithm to resolve the victim. If the victim can be dynamically selected at every site and included as part of the probe, then the probe will contain the final victim when it reaches the initiator. Then a message can be sent to kill the victim.

In section 6.2, a deadlock prevention algorithm based on resource ordering is proposed using the object semantics. In section 6.3, a modified fault informant probe based algorithm is proposed using colored probes. In section 6.4, the existing deadlock resolution algorithms are explored to identify the system parameters favored by the transaction attributes. A weight based victim selection algorithm is proposed to select the victim dynamically based on the choice of the desirable system parameters like throughput, resource utilization, fairness etc.

6.2 Deadlock prevention Algorithm for OODS

OODS follows AND request model as transactions need all the resources before execution. The resource ordering policy decides how the resources are ordered. The transactions are expected to request for the resources from smaller IDs to larger IDs. Then it is ideal to assign smaller resource IDs to independent resources and higher resource IDs to resources dependent on these independent resources. The proposed resource ordering technique is based on the dependency among the objects participating in the system. In OODS, the dependency among objects is based on their relationship with other objects as seen in section 2.2. Then dependency can be categorized based on

1. Relationship between object and class.
2. Objects related by inheritance.
3. Objects related by aggregation or composition.
4. Objects related by association.

The objective for resource ordering is to promote concurrency and it is based on granularity and degree of dependency. When the granularity size varies from coarse to fine, the concurrency increases. So lower resource IDs are given for fine granules and higher resource IDs are given for coarse granules. As mentioned earlier, the dependency is based on the relationships. As the dependency becomes more as given in section 2.2, number of objects needed is more. This reduces the concurrency. Based on these observations and constraints mentioned in section 2.2, partial ordering is done on each of the case mentioned above, and then total ordering of all the objects in a system is done.

In section 6.2.1 the resource ordering policy is framed and its formal model is given using predicate calculus in section 6.2.2. In section 6.2.3, system model is described. In sections 6.2.4 and 6.2.5, DPA modules and the algorithm are explained. In section 6.2.6, the access ordering rules proposed to alleviate the problem of starvation are listed. In sections 6.2.7 and 6.2.8, informal and formal proofs for the proposed DPA are given. Section 6.2.9 gives the summary.

6.2.1 Resource ordering technique

Partial ordering on a class and its objects:

The transactions can request for resources in two granularities namely single object in a class or all the objects in a class. If all the objects are requested, the lock is applied on the class rather than on the object to minimize the number of locks. Simultaneous request to both the cases is not allowed to maintain consistency. In this situation, two possibilities exist:

Case 1: Objects are assigned lower resource IDs than their class.
Case 2: Classes are assigned lower resource IDs than its objects.

In case 2, transactions executing class methods and requesting all the objects will have higher priority than transactions executing instance (object) method and requesting single object.

Example 1:

Let A be a class. Let a1, a2 and a3 be its objects.

Let T1 requests A. i.e., T1 requests {a1, a2, a3}.

(i.e. all objects in the class. Hence, the class itself is locked as per point (1) in 2.2)

Let T2 requests {a1}.

Let T3 requests {a3}.

Then concurrent execution of these requests proceeds as follows, if case 1 is considered. Both T2 and T3 are executed first and T1 is executed afterwards.

Case 1

Transaction T1	Transaction T2	Transaction T3
Wait for a1,a3 : : Get a1,a2,a3 Execute Release a1,a2,a3	Get a1 Execute Release a1	Get a3 Execute Release a3

Case 2

Transaction T1	Transactio n T2	Transactio n T3
Get a1,a2,a3 Execute Release a1,a2,a3	Wait for a1 : : Get a1 Execute Release a1	Wait for a3 : : Get a3 Execute Release a3

If case 2 is considered for the same scenario, then execution will be as follows. Both T2 and T3 are blocked, while T1 is executed first. Here case 1 improves concurrency and hence improves the throughput of the system. Hence, case 1 is better than case 2. It can be justified as follows. Hence case 1 is better than case 2.Then the rule can be defined as follows.

Rule 1: *For all objects O belonging to class C, resource IDs of objects O should be less than resource ID of their class C.*

(Note: Transactions will request either only one or all objects, ordering among the objects of a class is not necessary)

Partial ordering on base class and its inherited sub classes:

Here also resource IDs can be assigned in two ways:

Case 1: Base Classes are assigned lower resource IDs than their subclasses.
Case 2: Sub classes are assigned lower resource IDs than their base classes.

By definition of inheritance, attributes of base class are included as attributes of subclass. The definition and implementation of template member functions and the definitions of hook methods of base class are also included in the subclass. Hook methods are allowed for overriding in subclass.

Example 2:

Let A and B be base classes. Let a1, a2 be instances of A and b1, b2 be instances of B. Let C be a subclass inherited from A and B. Let c1, c2 be instances of C. Let a1 and b1 be associated base class objects for sub class object c1. Similarly, a2 and b2 be associated base class objects for c2.

Let T1 requests {A}; T2 requests {B}; T3 requests {C}.

Then the resource set for T3	= {A, B, C}.	(By point 2 in 2.2)
The resource set of T1	= {a1, a2}.	(By point 1 in 2.2)
Resource set of T2	= {b1, b2}.	(By point 1 in 2.2)
Resource set of T3	= {a1, a2, b1, b2, c1, c2}	(by point 1 in 2.2)

Case 1

Transaction T1	Transaction T2	Transaction T3
Get A Execute Release A	Get B Execute Release B	Wait for A, B : : Get A, B, C Execute Release A, B, C

Case 2

Transaction T1	Transaction T2	Transaction T3
Wait for A : : Get A Execute Release A	Wait for B : : Get B Execute Release B	Get A, B, C Execute Release A ,B ,C

If case 1 is considered, then T1 and T2 get more priority than T3. Then concurrency is improved in case 1 than case 2. Therefore, resource IDs for base classes (BC) should be less than resource IDs of sub classes (SC) and represented as follows.

Rule 2: *For all base classes BC, if a sub class SC is inherited from BC, then resource ID of base class BC should be less than the resource ID of subclass SC.*

(Note: Since a transaction will request only one of the sub classes at a time, the ordering among sub classes of a parent class is not necessary.)

Partial ordering on component objects and their composite object:

Here also resources can be ordered in two ways:

Case 1: Component objects are assigned lower resource IDs than their composite object.

Case 2: Composite object is assigned lower resource ID than its component objects.

Aggregation is an object relationship. By definition of aggregation, component objects are part of composite object. Composite object avails the service of component objects, to satisfy the transaction request.

Example 3:

Let A and B be component classes. Let a be an instance of A and b be an instance of B. C is a composite class constituted from A and B. c is an instance of C. Let a and b be associated component class objects for composite class object c.

Let T1 requests {a}; Let T2 requests {b}; Let T3 requests {c}.

Then the resource set for T3 = {a, b, c}   (By point 3 in 2.2)

If case 1 is considered, T1 and T2 get priority over T3. As their resource sets are mutually exclusive, they can be concurrently executed. If case 2 is considered, T3 gets priority over T1 and T2.Case 2 gives higher priority to dependent classes over independent classes, where as case 1 improves concurrency.

Case 1

Transaction T1	Transaction T2	Transaction T3
Get a Execute Release a	Get b Execute Release b	Wait for a, b : : Get a, b, c Execute Release a, b, c

Case 2

Transaction T1	Transaction T2	Transaction T3
Wait for a : : Get a Execute Release a	Wait for b : : Get b Execute Release b	Get a, b, c Execute Release a ,b ,c

Therefore, the partial ordering in the case of composition relationship among component class (CC) and composite class (CM) will be,

Rule 3: *For all composite classes CM, if a component class CC is a part of CM, then resource ID of component class CC should be less than the resource ID of composite class CM.*

(Note: Again, the ordering among component objects is immaterial, since they will be accessed mutually exclusively).

Partial ordering of associative objects and their associated objects: Association relationship is also an object relationship. Aggregation relationship is a subset of association. By extending rule 3 to association, the partial ordering among associative objects (TO) and their associated objects (AO) will be as follows:

Rule 4*: For all associative class (TC), if an associative class (TC) is associated with associated class (AC), then resource ID of associative class TC should be less than associated class AC.*

Total resource ordering:

The partial ordering of resources defined above is applicable when objects are related by only one relationship. However, in a business domain, objects may have complex relationships by having combination of above relationships. Hence, total ordering of resources that can have a combination of above relationships has to be defined. By combining the partial ordering rules proposed earlier, the total ordering of resources can be defined as follows:

Case 1: When a class has inheritance relationship

Here the ordering needs to be done on base class, its objects, sub class and its objects. This can be done by combining rule 1 and 2 in the previous section. There are two options here

Option 1: Resource ID of Base Class Objects < Resource ID of Base Class < Resource ID of Sub Class Objects < Resource ID of Sub Class.

Option 2: Resource ID of Base Class Objects < Resource ID of Sub Class Objects < Resource ID of Base Class < Resource ID of Sub Class

Here all the objects are given lower IDs and then their classes are given higher IDs. From examples 1 and 2, it is obvious that option 2 has higher concurrency than option 1. Hence, the base class objects and sub class objects are given lower IDs and their classes are given higher IDs.

Rule 5*: For all base class objects BCO belonging to BC and all sub class objects SCO belonging to SC, if a sub class SC is inherited from base class BC, then Resource ID (BCO) < Resource ID (SCO) <Resource ID (BC)< Resource ID (SC).*

Case 2: When a class has aggregation relationship

There are two options here

Option 1: Resource ID of Component Class Objects < Resource ID of Component Class < Resource ID of Composite Class Objects < Resource ID of Composite Class

Option 2: Resource ID of Component Class Objects < Resource ID of Composite Class objects < Resource ID of Component Class < Resource ID of Composite Class

Here by examples 1 and 3, it is clear that option 2 is better than option 1.Hence component objects and composite objects are given lower IDs and their classes are given higher IDs.

Rule 6*: For all component class objects CCO belonging to CC and all composite class objects CMO belonging to CM, if a component class CC is a part of composite class CM, then Resource ID (CCO) < Resource ID (CMO) < Resource ID (CM)< Resource ID (CM).*

Case 3: When a class has association relationship

There are two options here

Option 1: Resource ID of Associative Class Objects < Resource ID of Associative Class < Resource ID of Associated Class Objects < Resource ID of Associated Class

Option 2: Resource ID of Associated Class Objects < Resource ID of Associative Class objects < Resource ID of Associated Class < Resource ID of Associative Class

It is clear that option 2 is better than option 1 by extending examples 1 and 3 to association. Hence, associative objects and associated objects are given lower IDs and their classes are given higher IDs.

***Rule 7**: For all associative class objects TCO belonging to TC and all associated class objects ACO belonging to associated class AC, if an associative class TC is associated with associated class AC, then Resource ID (ACO) < Resource ID (TCO) < Resource ID (AC)< Resource ID (TC).*

Case 4: When a class has inheritance, association and aggregation relationships:

A class can have all the relationships in any order. For example, an inherited sub class can be component object of another class. Similarly, a class can be inherited from composite object class. It implies that the class relationships can be in any order. Hence ordering cannot be based on relationship alone. The class diagram is partitioned horizontally by various levels. In a class diagram, as the level increases, dependency increases and concurrency decreases. Concurrency decreases, because more no of resources are required for a transaction requesting high-level object or class to maintain consistency. Hence, lower resource IDs can be assigned for lower level, and higher resource IDs are assigned for higher level.

***Rule 8**: For all objects OA belonging to A, for all objects OB belonging to B, for all classes A in level i and for all classes B in level j of the class diagram where i< j, then Resource ID (OA) < Resource ID (OB) < Resource ID (A) < Resource ID (B).*

The above rules can be formally defined using predicate calculus.

6.2.2 Formal model for resource ordering using predicate calculus

Let the class diagram representing the domain be the Universal set U. U is a collection of classes C representing the domain. Let O be the collection of objects instantiated from the classes C. Then U can be represented as U(C (O)). The classes are related to each other by inheritance, aggregation and association relationships.

Let BC be base class and SC be sub class related by inheritance. Let BCO be base class objects and SCO be subclass objects. Let CC be component class and CM be composite class related by aggregation. Let CCO be component class objects and CMO be composite class objects. Let TC be associative class and AC be associated class related by association. Let TCO be associative class objects and ACO be associated class objects.

Let Rid (X) be a function that returns resource id for a resource X. Let Inherit-from (SC, BC) be a predicate that means SC is inherited from BC. Let Part-of (CC, CM) be a predicate that means CC is a part of CM. Let Associated-with (TC, AC) be a predicate that means TC is associated with AC. Let Level (Yi) be a predicate that means class Y is in level i. Let LL(i, j) be a predicate that means level i is less than level j.

Then rules 1- 8 can be written in predicate calculus as follows:

Rule 1: (O) (C) (O $\in$ C $\rightleftarrows$ Rid (O) $<$ Rid (C)).

Rule 2: (BC) (SC) (Inherit-from (SC, BC) $\rightleftarrows$ Rid (BC) $<$ Rid (SC)).

Rule 3: (CC) (CM) (Part-of (CC, CM) $\rightleftarrows$ Rid (CC) $<$ Rid (CM)).

Rule 4: (TC) (AC) (Associated-with (TC, AC) $\rightleftarrows$ Rid (AC) $<$ Rid (TC)).

Rule 5: (BCO) (BC) (SCO) (SC) ((BCO$\in$ BC) $\wedge$ (SCO$\in$SC) $\wedge$ Inherit- from (SC, BC) $\rightleftarrows$ Rid (BCO) $<$ Rid (SCO) $<$ Rid (BC) $<$ Rid (SC)).

Rule 6: (CCO) (CC) (CMO) (CM) ((CCO$\in$ CC) $\wedge$ (CMO$\in$CM) $\wedge$ Part-of (CC, CM) $\rightleftarrows$ Rid (CCO) $<$ Rid (CMO) $<$ Rid (CC) $<$ Rid (CM)).

Rule 7: (TCO) (TC) (ACO) (AC) ((TCO$\in$ TC) $\wedge$ (ACO$\in$AC) $\wedge$ Associated -with (TC, AC) $\rightleftarrows$ Rid (ACO) $<$ Rid (TCO) $<$ Rid (AC) $<$ Rid (TC)).

Rule 8: (i) (j) [(LL (i, j)$\wedge$ Level (Ci) $\wedge$ Level (Cj)) $\rightarrow$ {(Oi $\in$ Ci) (Oj $\in$ Cj) $\rightarrow$ Rid (Oi) $<$ Rid (Oj) $<$ Rid (Ci) $<$ Rid(Cj) }].

6.2.3 System Model

Fig 6.1 shows the architecture of the proposed system. It is assumed that the issues relating to partitioning of objects [Huang1990] in OODS are resolved. The Message Handler receives the client request (transaction) that is usually a call to an object method and derives method type and properties [Riehle2000a, Riehle2000b]. The transaction manager identifies the type of lock. Further, the granularity of the lock is decided by the type and properties of methods as in previous section. Since the resources are objects in OODS, resource manager in each of the sites is called as object manager. The transaction manager needs to get the resources from object manager to execute the transaction. The lock manager is responsible for maintaining the lock status of the resources in the site.

The resources granted to a transaction have to be locked before usage. The lock modes are shared (S) and exclusive(X). This is done to enforce synchronization and concurrency control through serialization. Once the execution is over, the lock is released and the resource can be utilized by the next waiting transaction. The object subsystem is responsible for the lifecycle of objects. It is also responsible for providing object persistence and garbage collection and other related activities. The deadlock preventor module checks for the presence of circular wait condition. If circular wait condition is not present, transactions can get the resources and execute.

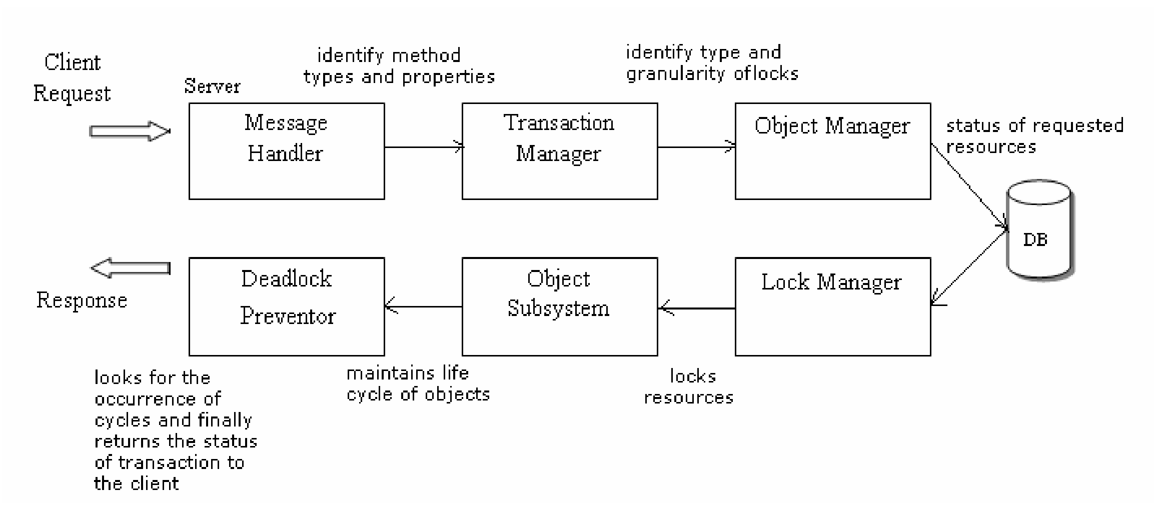

Client Request
Server
identify method types and properties
identify type and granularity oflocks
Message Handler
Transaction Manager
Object Manager
status of requested resources
DB
Response
Deadlock Preventor
Object Subsystem
Lock Manager
looks for the occurrence of cycles and finally returns the status of transaction to the client
maintains life cycle of objects
locks resources

Figure 6.1 Architecture diagram of the proposed system

6.2.4 Deadlock prevention modules description

Preparing the system

The algorithm is based on prevention of circular wait condition. Prevention of circular wait condition is achieved using resource ordering and access ordering. Hence, DPA requires some preprocessing to be done before accepting the client transactions. The preprocessing steps are:

1. Global ordering of the partitioned resources in the system.
2. Deployment of resources to various sites.

Global ordering

As the proposed algorithm is based on resource ordering, it is necessary to order the resources globally to avoid access conflicts. Since objects are the resources and their global relationship is known from the class diagram, the ordering is done on the class diagram. The class diagram is partitioned into levels based on the rules of horizontal partitioning. Then resource ordering is done by applying the total ordering rules specified in section 6.2.1.

Resource ordering method

1. Let < be the initial total order of all the objects. This ordering starts from level 1 up to level n.
2. For every set of objects O1 belonging to classes C1 in level 1, add the objects arbitrarily.
3. For objects in level 2 to level n, order the objects in increasing dependency such that resource id of objects in level i is less than resource id of objects in level j, where i< j. Define the function max_object_Rid(O), to return the maximum of resource ID of all objects O participating in the system which will be the resource id of last object in level n.
4. For all classes in level 1, i.e. independent classes(classes that need not lock any other classes to preserve consistency, like base classes and component classes), add the classes arbitrarily to create the total order such that max_object_Rid (O) is always less than resource IDs of independent classes i.e. classes in level 1.
5. For classes in level 2 to n, order the classes in increasing dependency, such that resource ID of classes Ci in level i is less than resource IDs of classes Cj, where i<j.
6. Let be the smallest transitively – closed order that is compatible with steps 1-5.

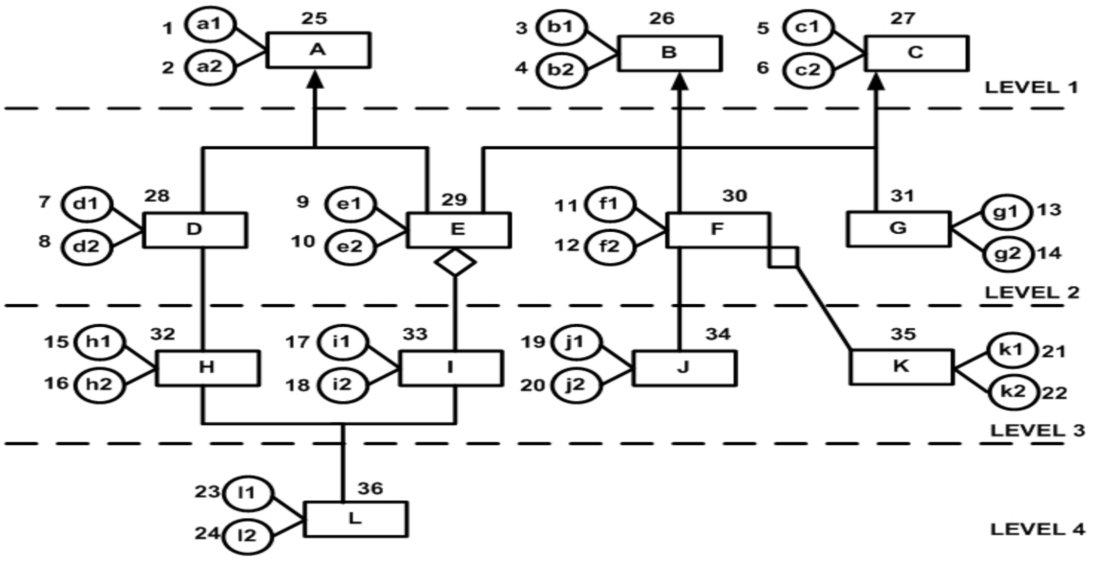


Figure 6.2 Sample class diagram after resource ordering

Deployment

Figure 6.2 shows the resource ordering of the sample class diagram. In this class diagram, there are four levels. The objects and classes are ordered from level 1 to level 4 using the resource ordering technique proposed earlier. The maximum resource id for objects is 24. Figure 6.3 shows the class diagram after deployment of classes, objects and associated database fragments to various servers. The transactions may come to any of these servers. The class diagram is horizontally partitioned and assigned to four sites S1 to S4.

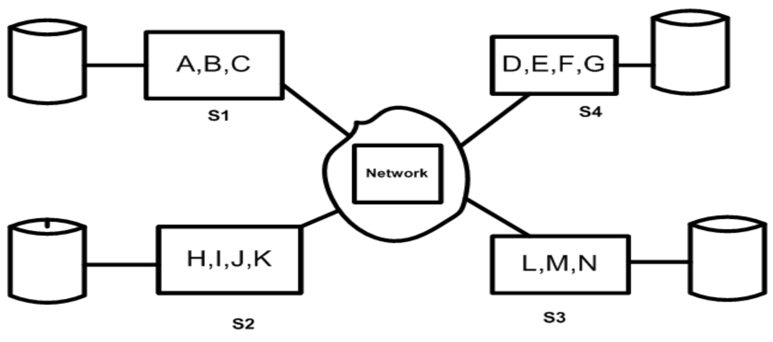


Figure 6.3 Deployment of sample class diagram

6.2.5 Deadlock prevention algorithm

The deadlock prevention algorithm imposes the following constraints:

- All transactions are time stamped.
- A transaction can request only one resource at a time. It can request for the next resource only when previous resource is granted.
- A transaction cannot request for resource of higher ID without getting the resource of lower ID.

It involves the following modules.

1. *Determine the explicit resources needed*

When a transaction enters the system, it request for resource. The resource that is determined from the transaction directly is called explicit resource. The resource type and granularity should be determined first. If the transaction maps onto instance method, the resource will be an object and its ID is known from the system. If it is a class method, it means all objects of the class needs to be locked. Here instead of locking objects, class has to be locked. Then the resource ID will be the ID of the class whose method is called in the transaction.

2. *Determine the implicit resources needed*

This step is needed mainly to maintain consistency of the system. As already mentioned, both base class and sub classes should be locked simultaneously, when sub class is requested by a client. Since base classes and their sub classes are mapped on to the same database, simultaneous access may lead to inconsistency. Similarly, a composite object and its component objects are not allowed simultaneous access. This is due to the 'part of 'relationship shared between component objects and composite objects. This is also extended to association.

In this module, the resource ID of the explicit (dependent) resource is checked for its relationship with other classes. This information can be documented using C++doc or Javadoc. If it is a subclass, all its parent classes need to be locked. Similarly, in the case of composite objects, all its component objects need to be locked. Then parent classes and/or component classes are implicit (independent) resources for the request.

Finally, this module will enumerate all the resources needed for satisfying the transaction request.

3. *Request Ordering*

First, the list of resources for every transaction, obtained from the previous module is sorted in increasing order of resource IDs. The objective of this request ordering is to enforce the rule that resources of higher IDs can be requested only after obtaining resources of lower IDs. The location of the servers for these distributed requests need to be identified, as the resources are partitioned to several sites.

4. *Access Ordering*

Any good DPA should prevent deadlocks and starvation. Starvation is abortion of same transaction repeatedly. If the transactions are allowed to access the resources based on their arrival time by FIFO basis, starvation in poverty will happen. If transactions are allowed to access the resources based on priority, starvation in wealth will happen. Hence, the proposed scheme provides a combination of FIFO ordering and priority ordering as given below. Here the priority is based on whether the transaction requests for dependent or independent resources. As the resource ID requested by a transaction increases, its priority decreases. This priority policy is chosen to improve concurrency.

Since horizontal portioning is followed, all the transactions will be accessing the sites in same order. First, all the transactions will request the resource of smaller IDs.

Due to horizontal partitioning, this will localize the transactions to the same site. The following rules are applied to grant a resource.

RULE 1: Transactions Ti and Tj are granted resources independently of their arrival time, if they satisfy 1 and 2 or 1 and 3 of the following conditions:

1. *The resources requested by Ti and Tj are different*
   i.e. Rid (ReqTi) $\cap$ Rid (ReqTj) = Φ.

2. *Both Ti and Tj have their resource requests i.e. Rid (ReqTi) and Rid (ReqTj) < = max_object_Rid (O).*

3. *Both Ti and Tj have requests i.e. Rid (ReqTi) and Rid (ReqTj)> max_object_Rid (O).*

RULE 2: Transactions Ti and Tj are granted resources on FIFO basis when the resource IDs are conflicting i.e. Rid(ReqTi) $\cap$ Rid (ReqTj) $\neq$ Φ

RULE 3: Transaction Ti has higher priority over Tj when

1. *The resources requested by Ti and Tj are different*
   i.e. Rid(ReqTi) $\cap$ Rid (ReqTj)= Φ
2. *Ti has its Rid(ReqTi) < = max_object_Rid (objects) and Tj has its Rid (ReqTj)> max_object_Rid (objects).*

6.2.6 Informal Proof

The following scenarios are considered to show the working of proposed DPA by taking typical sample scenarios. The transactions can request the resources only in two granularities namely classes and objects. So it is enough if the access ordering is done for transactions requesting objects, classes and objects and classes.

Scenario 1: This is a scenario where rule 1 is applied. (*Maximum Concurrency*)

Rule1 ensures maximum concurrency**.** The transactions are allowed to execute in parallel as long as their resource requests are mutually exclusive.

For example with reference to the above sample class diagram Let Ti requests {d1} and Tj requests {f1}.

Resource requests for Ti = {a1, d1}.

(Where a1 is the associated base class object of d1)

Requested resource IDs for Ti = {1, 7}.

Similarly resource requests for Tj = {b1, f1}.

(Where b1 is the associated base class object of f1)

Then resource request IDs for Tj = {3, 11}

They satisfy Rule 1, since {1, 7}$\cap$ {3, 11) = Φ

And both resource request sets {1,7} and {3,11}are < 24 (max_object_Rid (objects))

Then Ti and Tj can execute in parallel.

Scenario 2: This is a scenario where rule 2 is applied. (*Starvation in Poverty*)

This eliminates starvation in poverty. When the resource requests are conflicting, they are served on FIFO basis, as both transactions want some or all the resources that the other transaction also needs. Then younger transaction will have to wound and wait. Wound and wait is better than wait and die, because of rollback and restart costs. For example

Ti requests {h1};Tj requests {d1}.

Then resource request for Ti = {a1, d1, h1}. Resource IDs for Ti are {1, 7, 15}.

Tj request list is {a1, d1}.And request resource IDs are {1, 7}

Rid (ReqTi) $\cap$Rid (ReqTj) = {1, 7, 15} $\cap$ {1, 7} ≠ Φ

Then Ti and Tj are serialized based on their arrival time.

Scenario 3: This is a scenario where rule 3 is applied. (*Starvation in Wealth*)

This eliminates starvation in wealth. When two transactions implicitly conflict, then priority is given to transaction that needs lesser number of resources as an expedient scheduling measure. This will improve concurrency.

For example

Ti requests {d1};.Tj requests {D} (class locking implies it needs all its objects)

Resource requests of Ti = {a1, d1} ;And requested resource IDs for Ti = {1, 7}

(where a1 is the associated base class object of d1).

Resource requests of Tj = {a1, a2, d1, d2}

(where a1, a2 are the associated base class objects of d1 and d2.)

Then resource request ID for Tj = {32}

They satisfy Rule 1, since {1, 7}$\cap$ {32} = Φ

But implicitly Tj needs resources {1, 2, 7, 8}. This can be inferred by the second condition, Ti's resource request sets {1, 7} <24 and Tj's request ID {32} > 24 (max_object_Rid (O)). Then Ti gets higher priority over Tj.

6.2.7 Formal Proof

Theorem 1: The resource ordering < done using the proposed resource ordering method is total and meets rules 1-8.

$<$ **orders each pair of resources r_i and r_j :** If the resources r_i and r_j are not already ordered, by step 2 and 3 all objects are ordered. Then at least one of them should be a class.

Case 1: r_i is an object and r_j is its parent class (Rule 1)

Steps 2 and 3 orders all objects by their dependency. Without loss of generality, by step 3 and 4, max_object_Rid (r_i) < Rid (r_j). By transitivity of **<,** it is proved that $r_i < r_j$.

Case 2: r_i is a base class and r_j is its sub class (Rule 2)

If r_i is in level i and r_j is in level j, and levels $i < j$, then in general by steps 4 and 5, Rid (r_i) <Rid (r_j). By definition of inheritance, base classes will be always in lower level than their subclasses. Hence, it is proved.

Case 3: r_i is a component class and r_j is its composite class (Rule 3)

If r_i is in level i and r_j is in level j, and levels $i < j$, then in general by steps 4 and 5,Rid (r_i) < Rid (r_j). By definition of composition, component classes will be always in lower level than their composite classes. Hence, it is proved.

Case 4: r_i is an associative class and r_j is associated class (Rule 4)

If r_i is in level i and r_j is in level j, and levels $i < j$, then in general by steps 4 and 5, Rid (r_i) < Rid (r_j). By definition of association, associated classes will be always in lower level than their associated classes. Hence, it is proved.

Case 5: Proof of constraint Rule 5

From cases 1 & 2 by transitivity, rule 5 holds good for all resources r_i and r_j.

Case 6: Proof of constraint Rule 6

From cases 1 & 3, it is inferred that rule 6 holds good for all resources r_i and r_j.

Case 7: Proof of constraint Rule 7

From cases 1 & 4, it is inferred that rule 7 holds good for all resources r_i and r_j.

Case 8: Proof of constraint Rule 8

r_i is in level i , r_j is in level j and $i<j$. By steps 1- 6, from cases 5, 6 and 7, it is proved that they are totally ordered.

Theorem 2: Ordering < is well defined, if r_i is less than r_j, then $r_j < r_i$ does not hold.

Let $\zeta_<$ be defined as {R, E}, where resources R represent nodes and E is defined as set of edges linking those resources that are related by the ordering $r_i < r_j$, where $r_i , r_j \in R$ and < defines E, then it is enough to show that $\zeta_<$ contains no cycles.

The resources can be either objects or classes. All objects r_i and r_j are ordered by step 1 and 2 and edges $r_i \rightarrow r_j$ are added for all objects at these steps. If r_i and r_j are classes related by inheritance and/or composition and/or association, then by steps 4 and 5 they are ordered and $r_i \rightarrow r_j$ are added in this step. So it is clear that so far there are no back edges $r_j \rightarrow r_i$ added, as the objects and classes are ordered individually. In step 3 objects and classes are ordered by defining max_object_ Rid (object) to be less than the resource IDs for all classes. So, it is clear by transitivity, that all objects and classes are ordered. The edge $r_i \rightarrow r_j$ is added where r_i is an object with maximum resource ID and r_j is a class. Since edges are added for each type of resource exclusively, there will be no cycles.

6.2.8 Summary

Any DPA is expected to handle 3 conditions namely (1) Deadlock (2) Starvation on Poverty (3) Starvation on Wealth.

1 is solved in our algorithm by access ordering. Transactions can access the resources only in specific order, i.e. resources from lower IDs to higher IDs. Since FIFO ordering is followed, younger transactions wait on older transactions. Hence, circular wait is broken. This ensures breaking of cycles and hence deadlock is prevented.

2 Transactions are satisfied on FIFO basis. This eliminates starvation in poverty. Starvation in poverty generally occurs when larger requests are kept pending permanently. This is because smaller transactions are favored over bigger transactions to increase throughput. However, in our algorithm, the transactions are served in FIFO basis. Hence, starvation in poverty is eliminated.

3 Though, our algorithm favors FIFO ordering, when two transactions arrive at the same time, it favors the transaction requesting least number of resources. Hence, an expedient strategy is followed to speed up the computation, improve the resource utilization and alleviate starvation of wealth.

Thus our algorithm has shown that deadlock prevention algorithm is possible for distributed object oriented systems.

6.3 Fault Informant Probe based Distributed Deadlock Detection Algorithm

Probe based DDDR algorithm is widely used in distributed systems because of its simplicity. It detects the deadlock by sending a probe through the Wait-for-Edges in GWFG. If the probe returns back to the initiator, deadlock is deduced. This algorithm expects the network to be fault free. Practically networks are fault prone, hardware, network and software failures are bound to occur.

Though there are several fault tolerance algorithms for distributed systems, generally it is not expected by DDDR to provide fault tolerance. Hence much research work is not done in this area. However the DDDR algorithm should facilitate for the initiator of deadlock detection to infer whether it is due to live lock or site failure (where deadlock may be present) that the probe does not come back to it. If it is due to live lock, then the transactions can wait for finite time and then start sending resources' requests again. If the probe does not come back due to site failure, then the system needs to be reconfigured to continue. But if it is actually due to deadlock, and probe does not return to initiator due to site failure, then it is a serious problem.

Since there is no ideal fault tolerance algorithm and site failures are bound to happen, the initiator needs to get the probe back always despite whether it is live lock or dead lock. Hence this chapter aims to propose a fault informant algorithm that sends colored probes to initiator indicating the sites' status. It detects at most two site failures per deadlock cycle. The proposed algorithm uses the following colors in probe messages to indicate the status.

RED: Indicates deadlock and there is no site failure.

ORANGE: Indicates site failure. In this deadlock/ live lock status is unknown due to site failure.

WHITE: Indicates live lock and there is no site failure

In the proposed algorithm, transaction uses forward and backward probe messages to detect the reason for not getting the resource. Initially the color of the probe message is RED. The messages travel along the wait for edges and on the opposite sides by traversing node by node. A node that is receiving the probe message should send an acknowledgement to its sender. This is used to inform the active status of receiver to the sender.

If sender does not receive acknowledgement message before timeout, it infers that the receiver site has failed. Then sender will change the color of forward/backward probe messages into ORANGE after updating the bit in fault vector corresponding to the faulty site. The sender will send a return probe which is addressed always to the initiator with the color of forward/ backward probe along with updated fault vector and fault site ID. The initiator will broadcast the faulty state of the site to all the sites. This faulty status is modified only when the faulty site broadcasts awake message.

If the faulty site ID in both forward and backward probe messages is same, it can be inferred that it is one site failure. If they are different, then it is “two site failure” situation.

If there is no site failure, both the forward and backward probe messages will reach the initiator and by the RED color of the probe, the initiator will infer the presence of deadlock and will start deadlock resolution phase.

If there is no failure, but the wait for graph terminates at some node which does not have wait for edge, then receiver will not be able to send message any further. It will send acknowledgement message to sender indicating that it is active. Then receiver will change the color of the forward/backward probe into WHITE, and return probe with forward/backward probe color is sent to initiator. When the initiator realizes it is live lock, it will wait for some more time.

In the next section, the data structures and message formats used in the algorithm are defined. In section 6.3.2, the system model is described. In section 6.3.3, the proposed fault informant probe based DDDR algorithm is explained with pseudo code. In section 6.3.4, formal proof for the algorithm is given and section 6.3.5 summarizes the paper.

6.3.1 Definitions

Definition 1: Wait for Graph (WFG (N, E)) is a directed graph where nodes N represent transactions currently participating in the system and E is a finite set of edges representing the transaction dependency on resources. Ti-→Tj є E where Ti is waiting on Tj for the resource held by Tj. So Tj is successor of Ti and Ti is predecessor of Tj.

Definition 2: A Deadlock is identified by a directed cycle in the WFG.

Definition3: Forward probe (Forward_Probe (Initiator, Sender, Receiver, Forward_Probe_Color)) is a traversal of dependency edges in WFG from initiator and propagates until it reaches back initiator or terminates when there is no dependency edge for a transaction in the path i.e. TI → T1→T2…Tn, where {Tn = TI or Tn has no dependency edge | TI ,T1,T2….Tn є N}. The probe color is RED if there is a deadlock and WHITE if there is live lock in fault free environment. This probe will not reach the initiator in faulty environment.

Definition 4: Backward probe (Backward_Probe (Initiator, Sender, Receiver, Backward_Probe_Color)) is a traversal from initiator and propagates backwards along the dependency edges in WFG, i.e. TI $\leftarrow$ T1$\leftarrow$T2…Tn, where {Tn = TI or Tn has no dependency edge | TI $\rightarrow$ T1$\rightarrow$T2…Tn are directed edges ϵ E and T1,T2….Tn ϵ N}. The objective of using both probes is to identify at most 2 site failures in a deadlock cycle than 1 site failure as in[l]. The probe color is RED if there is a deadlock and WHITE if there is live lock in fault free environment. This probe will not reach the initiator in faulty environment.

Definition 5: A Fault Vector (FaultVector) V = S1S2…Sn, where S1, S2 …Sn denotes the N sites participating in the system domain. Si = 1, if site i is faulty; Si = 0, if site i is non faulty. Instead of PMC diagnosis model, the site fault is identified by message response from the neighboring sites.

Definition 6: Return probe (Return_Probe (Initiator, Sender, Forward/ Backward_Probe_Color, FaultVector, FaultSiteID,)) is the probe forwarded by the site Si holding transaction Ti to the initiator about its successor faulty site Sj holding transaction Tj, where Ti$\rightarrow$Tj ϵ E. This probe updates the status of site Sj in fault vector and sends it to the initiator. The initiator updates the status of Sj and broadcasts to all the other nodes for future requests. It stays unchanged until the awake message is received from the faulty site Sj. This is done during forward probe. In backward probe, if predecessor Tj is faulty, then this return probe is forwarded by the successor. The return probe color is WHITE if there is live lock in fault free environment. The return probe color is ORANGE if there is site failure. The return probe will have FaultVector and FaultSiteID only under faulty environment.

Definition 7: Acknowledgement message (Ack_msg (Receiver, Sender)):- Every site on receiving the probe message from its sender should send an acknowledgement message to its sender. If this message is not received by the sender, by time out period, it assumes that receiver is faulty. Then sender sends return probe to initiator updating fault vector about this faulty site.

Definition 8: Clean message (Clean_message) is to broadcast all the sites to clean the probes sent by victim which is in faulty site.

Definition 9: Victim is the lowest priority transaction which will be aborted to break the cycle. Here initiator is the victim.

Definition 10: Awake message (awake_mesg (SiteID)) is a message sent by all sites on startup or after fault recovery. This message is needed to update its status in fault vector and include it for further transaction requests.

6.3.2 System Model

The system is assumed to be free of congestion for timely delivery and messages are received in the order in which they are delivered. Further priority based DDDR algorithm [Mitchell1984] exists to ensure the least priority transaction in the cycle to become initiator of probe messages. This helps avoiding phantom deadlocks due to simultaneous initiation of probe messages for the same cycle. In each site it is assumed that only one transaction is running at a time. A transaction failure is also assumed as site failure. Each site is running one transaction for simplicity sake. The site index and transaction index are assumed as same.

6.3.3 Fault-Informant Probe Based DDDR Algorithm

In this proposed scheme, initiator will send forward as well as backward probes. Hence the algorithm in Li1993 uses backward probe alone and can detect only one site failure per deadlock cycle. Intuitively if we use both forward and backward probes in our algorithm, at most two failures can be detected. To improve the reliability of the system, we use both probes in our mechanism. The procedures for deadlock detection and resolution are given below.

```
Procedure Site_Initialization
If (fault_recovery or start_up)
Broadcast awake_mesg(SiteID)
End procedure.
```

This procedure will be called whenever a site is started or recovered from failure. On receiving this message, all the other non faulty sites will update the status of this site in their fault vector.

```
Procedure Transaction_ Initialization
Probe = null;
Fault_ Vector = Get_faultvector();
End Procedure
```

Any transaction that comes to the system will initially have the probe queue empty. It will get the current status of sites from the neighboring sites. Any transaction after making request for a resource will wait till time out or grant message which ever comes early. After time out, it will start sending probe messages. Any transaction Ti that receives forward or backward probe messages will execute the following procedure.

```
Transaction Ti::
Do
If Receive Forward_Probe (Initiator I, Sender Ti-1,
Receiver = Ti, Forward_Probe_Color = RED)
{
   Send Ack_msg to Sender Ti-1
   If there is no dependency edge from Ti
      {
         Send Return-Probe (Initiator, Sender,
            Forward_Probe_Color =WHITE);
         Exit;
      }
      else
      {
         Send Forward_Probe (Initiator I, Sender=Ti,
            Receiver = Ti+1,Forward_Probe_Color = RED)
         Until timeout
               {
                Wait for Ack_msg from Ti+1
                If Receive Ack_msg from Ti+1   break;
                }
         Update FaultVector[Si+1] = 1;
         Update FaultSiteID = S i+1;
         Send Return- Probe (Initiator I, Sender Ti,
            Forward_Probe_Color = ORANGE, FaultVector,
            FaultSiteID);
}

If Receive Backward_Probe (Initiator I, Sender Ti+1,
   Receiver = Ti, Backward_Probe_Color = RED)
{
      Send Ack_msg to Sender Ti+1
      If there is no dependency edge from Ti
         {
            Send Return-Probe (Initiator I,Sender Ti,
               Backward_Probe_Color = WHITE);
            Exit;
         }
      else
         {
            Send Backward_Probe Initiator I,Sender=Ti,
               Receiver = Ti-1,Backward_Probe_Color=RED)
            Until timeout
            {
               Wait for Ack_msg from Ti-1
               If Receive Ack_msg from Ti-1 break;
            }
            Update Faultvector[Si-1] = 1;
            Update FaultSiteID = S i-1;
            Send Return- Probe (Initiator =I,
            Sender=Ti,Backward_Probe_Color=ORANGE,
                  FaultVector, FaultSiteID);
         }
   }
Od.
```

The existing DDDR algorithm [Chowdary1989, Roseler1988, Sinha1985] determines the lowest priority transaction in a deadlock cycle and nominates it as initiator. Initiator will send forward probe along dependency edges and backward probe along the opposite direction of dependency edges.

```
Initiator:
Switch on case
{
Case 1: //Livelock/Deadlock-INITIATOR'S NEIGHBORING
            SITE(S) FAILURE
   {
      Send Forward_Probe (Initiator I,Sender=I,
         Receiver=Ti, Forward_Probe_Color=RED)
      {
         Until timeout
            {
               Wait for Ack_msg from Receiver Ti;
               If Receive Ack_msg from Receiver Ti break;
            }
         Update Faultvector[Si] = 1;
         Update FaultSiteID = Si;
         Forward_Probe_Color = ORANGE;
         // Declare Live lock or Deadlock due to site
         failure Si
         Broadcast Clean_message to roll back transaction
            Ti in the faulty site Si;
      }
         Send Backward_Probe (Initiator I, Sender I,
            Receiver = Tj, Backward_Probe_Color = RED)
            {
         Until timeout
            {
               Wait for Ack_msg from Tj
               If Receive Ack_msg from Tj break;
            }
         Update Faultvector[Sj] = 1;
         Update FaultsiteID = Sj;
         Backward_Probe_Color = ORANGE;
// Declare Live lock or  Deadlock due to site failure Sj;
         Broadcast Clean message to roll back transaction
            Tj in the faulty site Sj;
  }
}
Case 2: //Deadlock–NO SITE FAILURE;Probe comes back to
the initiator

If Receive Forward_Probe (Initiator I, Sender Ti,
receiver = Initiator, Forward_Probe_Color = RED) AND
Receive Backward_Probe (Initiator I, Sender Tj, receiver
= Initiator, Backward_Probe_Color = RED)
  {
   Call Deadlock Resolution Algorithm;
     // Declare Deadlock;
   }

Case 3:        // Livelock – NO SITE FAILURE
If Receive Return_Probe (Initiator I, Sender Ti, Receiver
= Initiator, Forward_Probe_Color = WHITE) AND Receive
Return_Probe (Initiator I, Sender Tj, Receiver =
Initiator, Backward_Probe_Color = WHITE)
   {
       Wait until timeout;       // Declare Livelock;
   }

Case 4:     // Deadlock/ Livelock in 1 / 2 SITE FAILURES

If Receive Return_Probe (Initiator I, Sender Ti, Receiver
= Initiator, Forward_Probe_Color = ORANGE, FaultVector,
FaultSiteID) AND Receive Return_Probe (Initiator, Sender
Tj, Receiver = Initiator, Backward_Probe_Color = ORANGE,
FaultVector, FaultSiteID)

If FaultSiteID in Forward probe== FaultSiteID in Backward
probe
   {
//Declare Live lock or Deadlock due to 1 site failure
   Broadcast Clean message to roll back faulty
      transaction Ta in the faulty site Sa;
   }

If FaultSiteID in Forward probe <> FaultSiteID in
Backward probe
   {
 // Declare Live lock or Deadlock due to 2 site failures.
 Broadcast Clean message to roll back transactions
   Ta and Tb in the faulty sites Sa and Sb;
   }
}

EndCase.
```

6.3.4 Formal Proof

The algorithm is proved correct under the following assumptions:

1. Transactions use single request model for requesting the resource.
2. No transaction in deadlock aborts unless it is victimized in resolution phase.
3. There are at most 2 site failures in a cycle.

Theorem 1: *The algorithm detects deadlock only if there is a deadlock.*

Proof: This algorithm detects a deadlock only when it receives both forward and backward probes and their colors are red. In that case, no site failure was there, when the probe was traversing. If a site had not sent acknowledgement message to its sender, the initiator would have received return probe messages. See figure 6.4.a for the traversal of probes in deadlock situation. See fig 2b for the probe traversal in faulty environment.

Scenarios

These scenarios are considered to give informal proof to the algorithm. They show how the algorithm works under both fault free and faulty environment. They also show how deadlock and live lock status are detected. In fault diagnosis model, it is shown that 2t+1 processors are needed to detect t failures. In our proposed algorithm, it is shown that 2t-1 processors are enough to detect t failures through messaging.

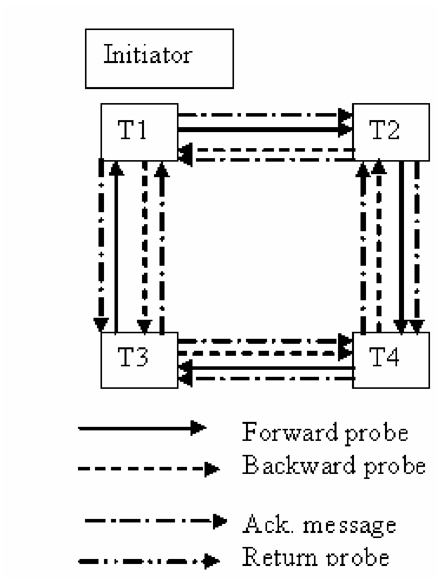


Figure 6.4a Deadlock in fault free environment

Since the proposed algorithm also can detect at most 2 site failures, minimum number of nodes are taken to show the working of the proposed algorithm.

Figure 6.4 shows various possible scenarios in a distributed system during deadlock detection phase.

Scenario 1: Figure 6.4a depicts the scenario when deadlock occurs in fault free environment. Initiator T1 sends red colored forward and backward probes along dependency edge and opposite direction of dependency edge. When the initiator receives back both probes, it infers the presence of deadlock and that there is no site failure along the path. Then initiator (lowest priority) is victim. The color of the probe is RED.

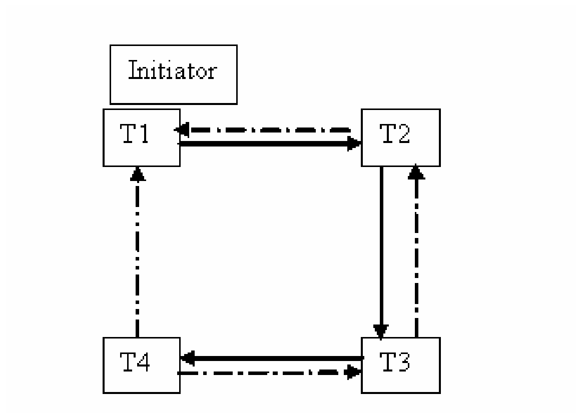


Figure 6.4b: Live lock in fault free environment- case 1

Scenario 2: In figure 6.4b, let us assume that T1-the initiator does not have predecessor. Then backward probe will not be sent back to the initiator. As T4 does not have any dependency edge, forward probe terminates at T4. T4 changes the forward probe to WHITE indicating live lock status and T4 is active. This is also a scenario in fault free environment.

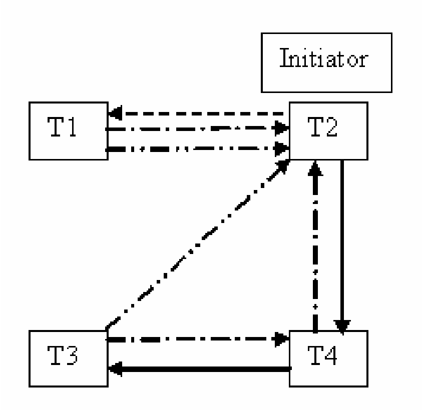


Figure 6.4c Live lock in fault free environment – case 2

Scenario 3: Figure 6.4c also depicts live lock status in fault free environment. Here T2 is assumed to be having least priority. Hence it becomes the initiator.T3 and T1 send forward and backward probes indicating live lock. So they change the color of the forward and backward probes WHITE and send return probe back to initiator.

***Scenario 4**:* In figure 6.5a, let us assume T4 is faulty.T3 waits until time out for acknowledgement message from T4. If there is no acknowledgement message from T4, then it updates the fault vector for site 4, changes the color of probe message to ORANGE indicating site failure and sends return probe to initiator.T1 also sends backward probe. T4 sends no acknowledgement even after timeout. T1 concludes T4 faulty. It is confirmed by ORANGE forward probe from T3. Since the fault site id in return probe from T3 matches with the faulty site deduced by initiator, it concludes one site failure and it may be a live lock or deadlock situation.

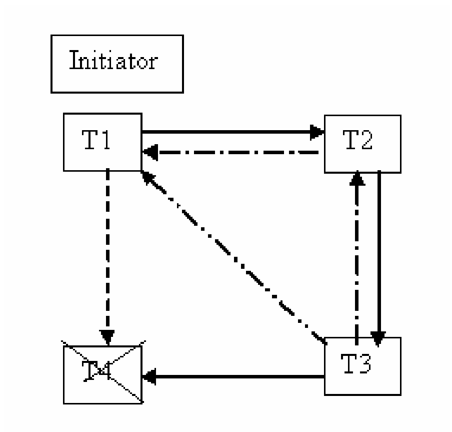


Figure 6.5a Live lock/Deadlock in faulty environment (1 site failure)- case1

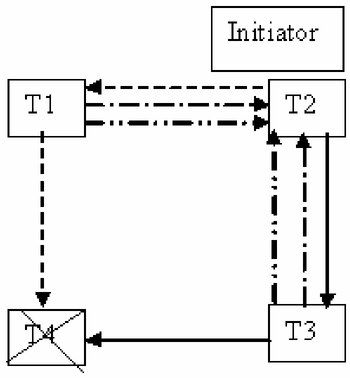


Figure 6.5b: Live lock / Deadlock in faulty environment (1 site failure)- case 2

***Scenario 5**:* In figure 6.5b, T2 is initiator. It sends forward probe to T3 and backward probe to T1. T4 is faulty. So T1 and T3 will not receive acknowledgement messages. They will change the probe color to ORANGE and send back to initiator after updating fault vector on T4. Since both T1 and T3 will have their fault site ID same, it is concluded that it is live lock/ deadlock due to single site failure.

Scenario 6: In figure 6.5c, Assume T3 and T4 are both faulty. T2 will send forward probe to T3. T3 will not send acknowledgement message. So T3 is updated as faulty on time out. T2 sends T1 backward probe. T1 forwards backward probe to T4. As T4 is also faulty, it will not send acknowledgement to T1. So T1changes return probe to ORANGE and updates T4 status. Since the fault side IDs updated by initiator and T1 are different, T2 will understand both T3 and T4 are both faulty and infer that it is two site failures scenario.

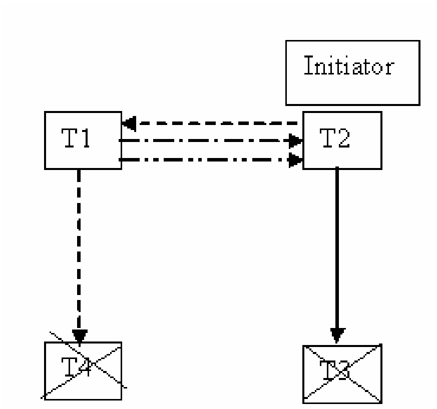


Figure 6.5c Live lock /Deadlock in faulty environment (2 site failure)- case1

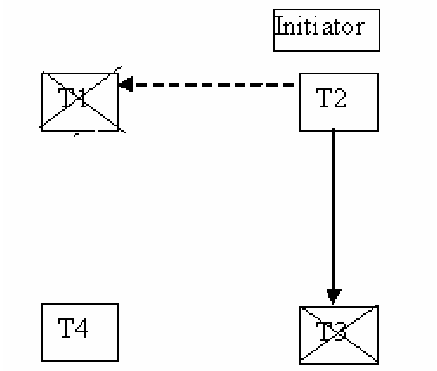


Figure 6.5d Live lock/Deadlock in faulty environment (2 site failure) – case 2

Scenario 7: This is the worst case for 2 site failures with 4 sites in picture. Let T1 and T3 are faulty. Initiator sends forward and back ward probes to them. On time out, since it does not receive acknowledgement messages from neither T1 nor T3, it deduces that T1 and T3 are both faulty. However the status of T4 is unknown, as it is unreachable by both T1 as well as T3. Since this algorithm can only detect at most 2 site failures, it cannot be inferred. However in Preparata1967, it is stated that at least 2t+1 processors are needed to detect t failures using PMC diagnosis model. However in our case it can be inferred that t failures can be detected with 2t – 1 processors.

6.3.5 Summary

A new fault tolerant algorithm for Distributed Deadlock Detection and Resolution is proposed with the following improvements:

- Initiator always knows the status of probe whether deadlock or live lock or site failure.
- In Li1993 every non-faulty site tests other sites periodically for site failures. In the proposed algorithm the site failure is decided by acknowledgement messages. This improves the throughput of non-faulty sites.
- Checking whether faulty sites are rectified is known by awake message. This situation is not handled separately in Li1993.
- Only one site failure is handled in Li1993. This paper however handles at most 2 site failure which improves fault tolerance
- The color of the probe is used to indicate the status of the system. Red indicates deadlock with no site failure. Orange indicates live lock or deadlock due to site failure. White indicates live lock due to a transaction having no dependency edge.
- The worst case message complexity is 4n where n is the number of transactions. This occurs when there is no site failure and deadlock occurs. The four messages are the forward probe message and backward probe message to next nodes and acknowledgement messages for both forward and backward messages to senders (see figure 6.4a).
- Further fault identification is better than fault diagnosis model, which needs 2t+1 processors to identify t failures. In the messaging mechanism, 2t-1 processors are enough to identify t failures.

6.4. Weight Based Victim Selection Algorithm

Deadlock resolution phase follows deadlock detection phase. In this a transaction is chosen (called as victim) for abortion to break the circular wait that is causing the deadlock. Several victim selection algorithms have been analyzed in chapter 2.4.3.

6.4.1 Performance of existing victim selection algorithm

Based on the definition of the victim selection algorithms, their characteristics along with their time complexity can be summarized as in table 6.1.The time complexity helps to determine the deadlock resolution latency and defined in terms of ‘n’- the number of transactions. From the table, it is worth noting certain points.

Table 6.1: Comparison of various victim selection policies

Victim Selection Policy	Optimal in	Time Complexity
Youngest [Agarwal1987]	Fairness	O(n)
Min. History [Agarwal1987]	Fairness	O(n)
Least Static Priority [Newton1979]	Response time	O(n)
Maximum Size[Chow1991]	Throughput	O(n)
Min. no. of locks[Agarwal1987]	Resource Utilization	O(n)
Max. no of cycles[Singhal1989]	Throughput	O(n2 x k + 2) k– max. cycle size
Minimum abortion cost[Srivastava2007]	Resource Utilization	O (n3)
Max. Edges [Singhal1989]	Resource Utilization	O (n4)
Blocker [Agarwal1987]	Resolution latency	O(1)
Initiator	Resolution latency	O(1)
Max. release set[Moon1997]	Resource utilization, throughput	O (n3m) m- no of resources
Min. work done so far[Holt1972]	Resource Utilization	O (n3)
Priority + resource priority + min. work done[Garey1979]	Resource utilization, throughput	O(n4.m) m- no of resources

Youngest is fair in giving priority to the transactions based on their arrival time. So this will eliminate starvation in poverty [Holt1972]. But this might introduce starvation in wealth [Parnas1972]. Static priority lets the user to configure the priorities of the transactions participating in the system. This eliminates starvation in wealth. This policy is ideal for real time systems. The transactions can be prioritized based on the need of immediate or delayed response time. But resolution using history eliminates both starvation in wealth and starvation in poverty. History based resolution is also fair in the sense that it does not penalize any transaction again and again. Resolution based on transaction size will increase the number of transactions completed per unit time i.e. throughput.

Resolution policies like minimum number of locks, minimum work done and minimum abortion cost focus on lower rollback cost of a transaction, while algorithms like initiator, blocker, maximum edges and maximum release set choose a victim based on overall better performance of the system than on concerned individual transactions. The victims chosen using these algorithms might have already acquired all the required resources, completed maximum amount of execution or had been aborted again and again in the past. So they are not fair on individual transactions.

Blocker and initiator can be lower priority transactions and their only benefit is better deadlock resolution latency, especially in distributed systems. While all the above mentioned algorithms abort one transaction per cycle, resolution based on maximum number of cycles tries to reduce this. So number of transactions executed per unit time i.e. throughput increases in this case.

A simulation experiment has been made to study their characteristics with respect to other attributes. To study these desirable characteristics, attributes of 500 transactions are randomly generated and tested. From the given victim selection algorithms, blocker and initiator algorithms are not considered because, they are optimal only in deadlock resolution latency and poor in other aspects. Victim selection algorithm by Srivastava2007 takes maximum deadlock resolution latency, hence it is also not considered.

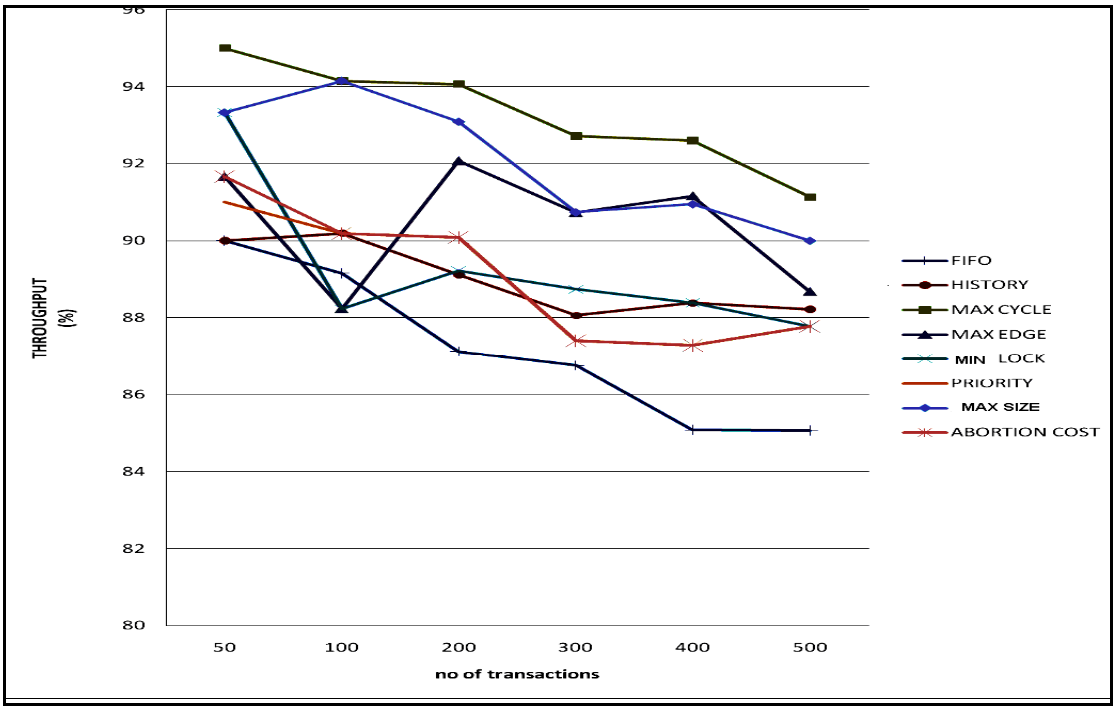


Figure 6.6 Performance for number of transactions versus throughput

In figure 6.6, it can be noticed that algorithm choosing victim based on maximum number of cycles provide maximum throughput i.e. more than 96%.This is because it aborts at most ‘n’ transactions, when there are ‘n’ cycles, whereas other algorithms abort atleast ‘n’ transactions. Then selection on maximum size provides better throughput i.e. 94%.This is because smaller transactions finish in time when the transactions are relatively smaller in size. In max edge algorithm, by aborting one transaction many transactions can proceed. Therefore the throughput is more in this case also. The performance of other resolution algorithms also depend on attributes of participating transactions and are bound to vary.

In figure 6.7, resource utilization is compared by varying number of transactions. While throughput is measured in terms of number of transactions, resource utilization is measured in terms of resources. Maximum resource utilization happens when there is minimal rollback. Resource utilization is maximized in resolution algorithms of minimum number of locks and maximum size. It is also noticeable that minimum abortion cost based on arrival time and minimum number of operations has made the algorithm suboptimal in both aspects .But it is better than algorithms considering single transactional attribute.

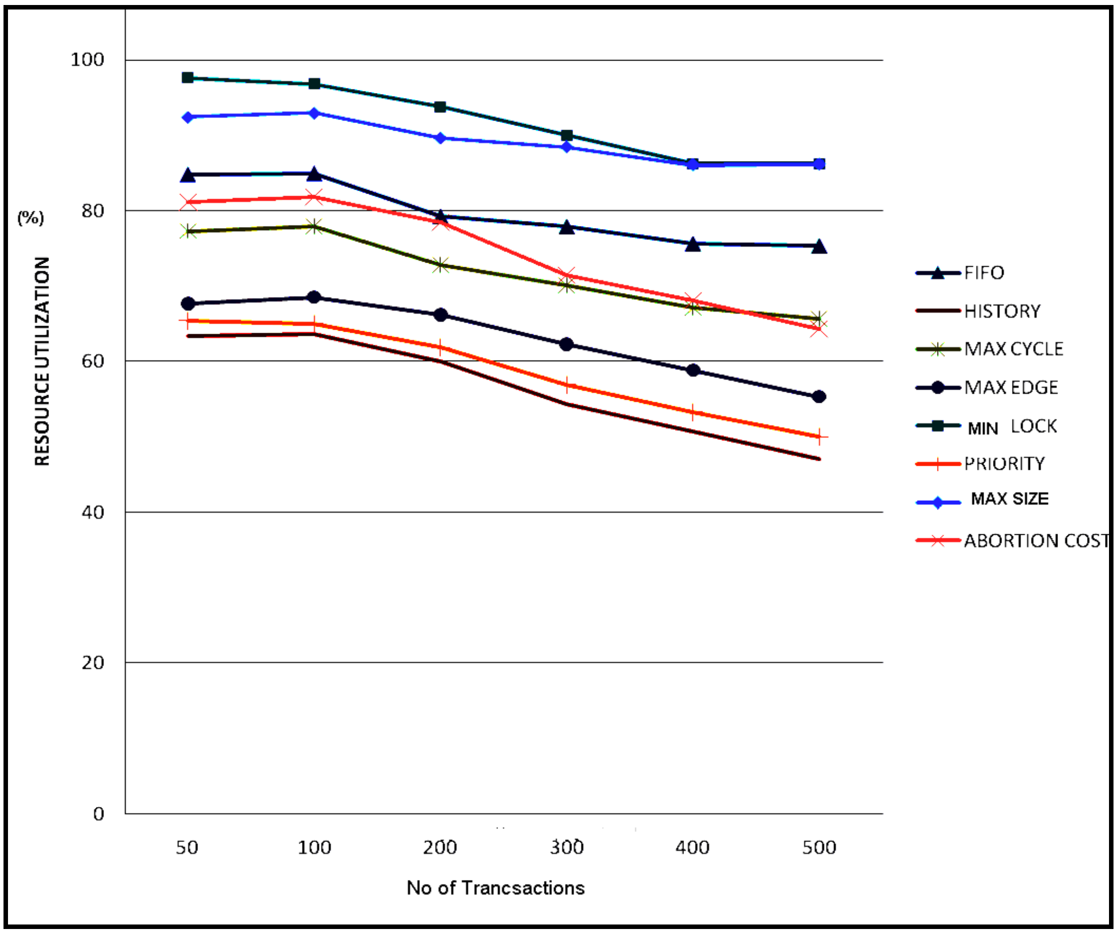


Figure 6.7 Performance for number of transactions versus resource utilization

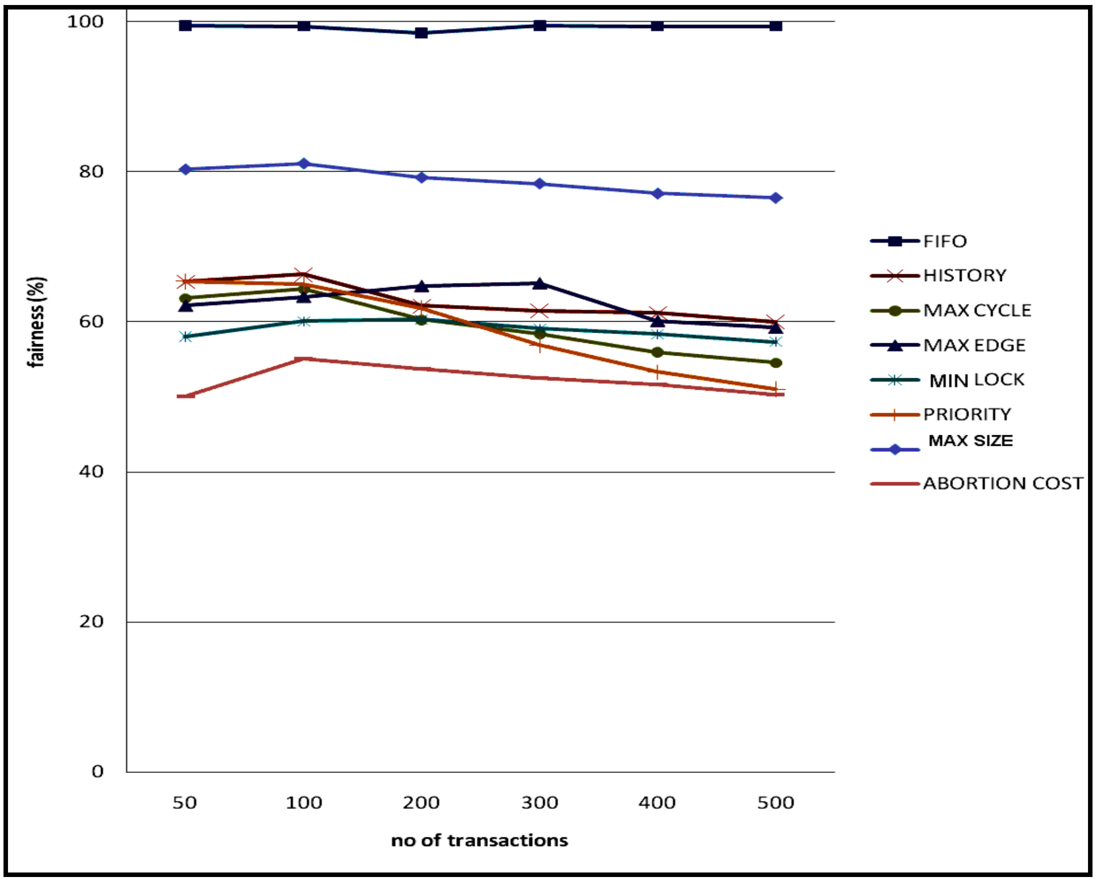


Figure 6.8 Performance for number of transactions versus fairness

In figure 6.8, fairness is compared with number of transactions. Fairness can be viewed in two aspects: based on age and starvation. The fairness considered in figure 6.8 is based on arrival time. While throughput, deadlock resolution latency and resource utilization are desirable attributes of the system, non- starvation and fairness are desirable attributes of individual transactions.

6.4.2 Proposed weight based victim selection algorithm

The victim selection algorithms are based on transaction attributes and resource attributes. The common transaction attributes or characteristics are age, history, code size, priority, resource utilized so far and its attributes in WFG like in-degree, number of edges, out-degree and cycle participation. Apart from static priority, transactions are usually dynamically prioritized based on the attributes given above. Victim selection algorithm based on resource attributes could be based on resource priority, number of units of a particular resource, costly resource, resource most sought after etc.

The desirable attributes that could improve the system are higher throughput, better resource utilization and lesser deadlock latency. The desirable attributes in individual transaction execution is lower response time, fairness, no starvation and minimum roll back cost. It can be noticed that while each victim selection algorithm is optimal in one aspect, it is suboptimal in other aspects.

Hence the proposed algorithm is based on assigning weights which can be configured based on user requirement. For example in real time systems, response time is more important than other attributes. Similarly throughput is important in batch processing systems.

In the proposed algorithm, each transaction is expected to possess an attribute list to maintain its rank in various aspects. The attribute list of a transaction is as table 2a. The attribute list is created for every transaction arriving at the system. Attribute lists of all transactions whose execution are completed are deleted. Attribute lists of all live transactions whose execution are not completed, are also updated whenever a new transaction arrives or an old transaction leaves the system.

The rank of a transaction with respect to a particular attribute is based on its value relative to other transactions with respect to that factor. For example, if a transaction has arrived third among the active transactions in the system, then its rank with respect to age is fixed as 3. In general, the ranks are determined based on the seniority of the transaction.

Table 6.2a Transaction attribute list

Transaction ID	Rank
Age	
History	
No of resources requested	
No of resources granted	
Size	
Static priority	

The weights of desirable attributes in the system can be configured so that Σ (G, F, L, T, R) =100%.This is done based on the nature of the distributed system. The weights can be in the range 0 to 100%. The ranking of transactions in a centralized system is easy. However the deadlock detection and selection of a victim for resolution in distributed system is very tedious. The candidate victims are distributed in various sites. Selecting a victim transaction for abortion at each local site is suboptimal and affects the performance of the system. Hence global selection of victim needs propagation of transaction attributes to all sites. The sites in a distributed system communicate through messages. Hence probe based deadlock detection in Chandy1983 is one of the best distributed deadlock detection algorithms.

Table 6.2b Desirable performance attributes of the system

Desirable system attribute	Wt as %	Rank of Transaction attribute to be favored
Throughput	G	Size
Fairness	F	Age, history
Resolution latency	L	Initiator, random blocker
Response time	T	Static priority
Resource utilization	R	No of resources requested

In this algorithm, the transactions waiting for grant message will start sending probe message after time out. The probe is sent along the wait for edges of a Global Wait for Graph (GWFG). The probe message has the fields such as initiator (transaction initiating the probe), sender (transaction forwarding the probe), receiver (transaction receiving the probe).This algorithm is used to detect cycle which indicates the presence of deadlock. Two new fields' namely current victim's transaction ID and its attribute list can be added to the probe for propagation for global victim selection. At each site, the rank of the current victim is compared with the locally selected victim. If the rank of local victim is higher than the current victim, the current victim can be replaced before forwarding the probe. When the probe reaches the initiator, the Transaction ID which is to be aborted will be known. Then command can be sent to abort the victim to break the cycle and restart the system.

6.4.3 Summary

Resolution is an important phase in handling deadlocks. Selection of a victim influences the performance of the system strongly. The desirable performance parameters of the system like throughput, resource utilization are influenced by the victim selection. The desirable parameters of the transaction like fairness, resolution latency and response time are to be considered while selecting a victim. The weight based victim selection scheme balances the desirable parameters of transactions and system.

CHAPTER 7

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

This concluding chapter recounts the research contributions with a brief discussion on the merits of the proposed semantic multi-granular locking models for OODBMS and OODS and deadlock handling techniques for OODS. It also reveals a few open problems in the focused area of research.

7.1 Conclusions

Nowadays business domains and engineering applications are emerging as distributed applications. They require support of complex data modeling and long duration transactions. Application of object oriented paradigm provides support of complex data modeling. These domains are also continuously evolving in nature to provide better service to their clients. This requires provision of high concurrency and reliable consistency. Existing conventional concurrency control techniques are not equipped to meet these requirements of the recently emerging applications. Semantic concurrency control techniques exploiting the features of object oriented paradigm have shown better performance than the conventional concurrency control techniques.

In this research, semantic multi-granular lock models and deadlock handling techniques are proposed for distributed object environments. The developed models and algorithms were evaluated by conducting simulation experiments using the extended version of 007 benchmark. Semantic multi-granular lock models have been proposed for distributed object environments supporting stable as well as continuously evolving domains. A deadlock prevention algorithm has been proposed based on resource ordering and access ordering. The resource ordering and access ordering principles are based on the semantics of object oriented paradigm. Resource ordering policy breaks the circular wait. Access ordering policy eliminates the starvation in wealth and starvation in poverty. Existing probe based deadlock detection algorithm by Chandy1983 is very popular for its easy implementation. It uses probe messages to detect deadlock. It has the limitations of not being fault tolerant. It requires a separate resolution phase. These limitations are overcome by the proposed fault informant probe based algorithm and weight based victim selection algorithm. The fault informant probe based algorithm detects the status of the system as whether it is in deadlock or live lock state or having site failures. The weight based victim selection algorithm selects a victim based in the weights assigned to the desirable parameters of the system like throughput, fairness, resource utilization etc. This algorithm is used to dynamically select a victim by communicating the victim selected so far in the probe. Thus it eliminates the need for separate resolution phase. The features of the developed algorithms are narrated in the subsequent paragraphs.

(i) A Consistency Ensured Semantic Multi-Granular Lock Model (CESGML) is proposed for OODBMS implementing stable domains. In stable domains, the runtime transactions are more in number and design time transactions are rare. The model provides fine granularity of design time operations and ensures consistency of runtime transactions. It covers all the design time operations mentioned in the literature. It exploits the semantics of object oriented paradigm to provide rich set of lock modes by identifying the mutually exclusive operations. The need for access vectors is minimized by defining fine granular lock modes. Apart from the access vectors for attributes and methods, another access vector called Class Dependency Vector (CDV) is proposed for classes. This provides finer granularity for class definition and class relationship access. This improves parallelism for node level and link level design time operations.

Extended 007 benchmark for OODBMS is used to test its performance. The CESGML is compared with the latest existing techniques based on Semantic MGLM based on compatibility of relationships namely ORION and Semantic MGLM based on commutativity of operations namely Jun2000 scheme. CESMGL scheme is better than Orion by 68% and Jun2000's scheme by 32.1%. ANOVA is also performed by comparing the three techniques by varying the number of transactions, design time to runtime ratio and varying the types of design time operations. In ANOVA and Duncan range test, it is found that CESGML provides very less response time when compared to the other two schemes.

(ii) Two semantic multi-granular lock models namely semantic MGLM using lock rippling and MGLM using access control lists are proposed for OODBMS implementing continuously evolving domains. In continuously evolving domains, more number of design time transactions arrives along with runtime transactions. The existing optimal models for OODBMS require apriori knowledge of the structure of classes. They use access vectors along with commutativity matrix to provide high concurrency. They incur the search and maintenance overhead of access vectors. This overhead increases linearly with the number of design time transactions. The proposed models eliminate the need for access vectors and hence reduce the response time when compared to the existing models.

Semantic MGLM using lock rippling extends the ORION locking scheme. In ORION scheme, the locking is always from the parent to the leaf for both runtime as well as design time transactions and intension locks are used to provide multi-granular lock support. In object oriented paradigm, there is an upward dependency from children to parents for runtime transactions. There is a downward dependency from parents to children in the case of design time transactions. This principle is not utilized in ORION. Further, intension locks do not convey what semantic operation is done on the fine granules and only S and X lock modes are provided for all read and write operations. In lock rippling, this lacuna is remedied by rippling the lock mode to convey the semantic operation taking place. A commutativity matrix is specified to define the conflicting operations. Then more than one lock can be placed on the same class as long as they are not conflicting. This helps to provide more concurrency than ORION. It does not require any access vectors for its operation. So the access vector overhead is nil in lock rippling mechanism.

Semantic MGLM using access control lists provides the same level of concurrency as in CESGML scheme without the limitations of access vectors, by splitting the lock table into three lists namely *Available, Shared and Exclusive* lists. This eliminates the need for maintaining vector tables along with lock table. The maintenance overhead is minimized as access vectors are not needed. In *Available list*, the attributes and methods of each class that are currently available are included. In *Shared list*, the attributes and methods of each class that are currently in shared (read) lock mode are included. In *Exclusive list*, the attributes and methods of each class that are currently in exclusive (write) lock mode are included. The search time is minimized as the lock table is split into three lists. Any transaction requires searching only one of these lists instead of all of them. This reduces the search overhead to roughly about one third. In order to save search time further, list search policies are given. Exclusive lock mode is allowed for a requested resource only if the resource is currently present in *Available* list. Shared lock mode is allowed, only if the resource is not in *Exclusive* list.

Extended 007 benchmark is used to compare their performance with the existing techniques. Semantic MGLM using lock rippling is better than CESGML by 14%. Semantic MGLM using access control list is better than CESGML by 21%. The response time for lock rippling is more because of the coarse granularity of lock modes for runtime transactions. However they do not need any apriori knowledge of object structure. Thus, the proposed schemes perform better for continuously evolving domains.

(iii) A semantic MGLM based on compatibility of relationships is proposed for OODS. Lock modes and granularity of locks are proposed for attributes and classes. Lock modes and granularity of locks are proposed for methods based on their types and properties for each of the relationship namely inheritance, aggregation and association. The granularity of runtime transactions is extended to attribute level. The granularity of design time transactions are still coarse due to the limitations of programming languages using which the domain is implemented.

(iv) A deadlock prevention algorithm is proposed for OODS based on resource ordering. The resource ordering technique is based on the object semantics. A formal model of the resource ordering technique is proposed using predicate calculus. An expedient access ordering policy is also proposed to eliminate the starvation in poverty and starvation in wealth. Informal and formal proofs are given. Any DPA is expected to handle 3 conditions namely (1) Deadlock (2) Starvation on Poverty (3) Starvation on Wealth. Deadlock is prevented in our algorithm by access ordering. Transactions can access the resources only in specific order, i.e. resources from lower IDs to higher IDs. Since FIFO ordering is followed, younger transactions wait on older transactions. Hence, circular wait is broken. This ensures breaking of cycles and hence deadlock is prevented. Transactions are satisfied on FIFO basis. This eliminates starvation in poverty. Starvation in poverty generally occurs when larger requests are kept pending permanently. This is because smaller transactions are favored over bigger transactions to increase throughput. However, in our algorithm, the transactions are served in FIFO basis. Hence, starvation in poverty is eliminated. Though, our algorithm favors FIFO ordering, when two transactions arrive at the same time, it favors the transaction requesting least number of resources. Hence, an expedient strategy is followed to speed up the computation, improve the resource utilization and alleviate starvation of wealth. Thus our algorithm has shown that deadlock prevention algorithm is possible for distributed object oriented systems.

(v) A probe based distributed deadlock detection algorithm using colored probes is proposed to provide fault tolerance. In this, Initiator always knows the status of probe whether deadlock or live lock or site failure. In existing algorithm, every non-faulty site tests other sites periodically for site failures. In the proposed algorithm the site failure is decided by acknowledgement messages. This improves the throughput of non-faulty sites. Fault identification is better than the existing fault diagnosis model, which needs 2t+1 processors to identify t failures. In the messaging mechanism, 2t-1 processors are enough to identify t failures. This algorithm is capable of detecting at most two failures, whereas in the existing algorithm only one site failure can be detected. The worst case message complexity is 4n where n is the number of transactions. This occurs when there is no site failure and deadlock occurs.

(vi) A weight based victim selection algorithm is proposed to dynamically select a victim based on the system parameters chosen for deadlock resolution. The victim selection algorithms are based on transaction attributes and resource attributes. The common transaction attributes or characteristics are age, history, code size, priority etc. The desirable attributes that could improve the system are higher throughput, better resource utilization and lesser deadlock latency. The desirable attributes in individual transaction execution is lower response time, fairness, no starvation and minimum roll back cost. It can be noticed that while each victim selection algorithm is optimal in one aspect, it is suboptimal in other aspects. Hence the proposed algorithm is based on assigning weights which can be configured based on user requirement. For example in real time systems, response time is more important than other attributes In the proposed algorithm, each transaction is expected to possess an attribute list to maintain its rank in various aspects. Based on the effects of the various transaction attributes on the performance of the system and transactions, desirable system parameters are chosen and the transaction attributes influencing them are identified. Weights are assigned based on the choice of system parameters desired. The victim is thus computed in every site. When the probe circulates along the wait-for-edges in GWFG, the victim is updated by comparison. When the probe comes back to initiator, the victim ID is available for abortion.

7.2 Future Research Directions

In this research attempt is made to provide semantic concurrency control and deadlock handling for distributed object environments. The algorithms developed in this research can be further extended in the following aspects.

i. The semantic multi-granular lock models provide fine granularity using access vectors. They require prior knowledge of the structure of classes to provide high concurrency. There is a trade off between granularity and requirement of prior knowledge of class structure. Hence the models may be explored for providing fine granularity without the overhead of apriori knowledge of class structure.

ii. MGLM using lock rippling does not use any access vectors to provide concurrency control. However it provides coarse granularity as compared to MGLM using access control lists. Hence new lock modes may be explored to provide fine granularity of design time operations.

iii. Semantic MGLM proposed for OODS offers coarse granularity for design time transactions due to the limitations of programming languages. Hence possibilities may be explored to propose fine granule of design time operations.

iv. Semantic MGLM proposed for OODS is based on compatibility of relationships. A semantic MGLM based on commutativity of operations can be explored to compare their performance and identify the model that performs best.

v. Deadlock prevention is a proactive approach. Deadlock detection and resolution is reactive approach and hence favored. A deadlock detection algorithm can be explored by exploiting the semantics of objects. It may give better performance as it has done in semantic concurrency control and deadlock prevention.

REFERENCES

Agarwal1987 R.Agarwal, M.Carey and L.Mcvoy, “The performance of Alternative Strategies for dealing with Deadlocks in database Management Systems”, IEEE Transactions on Software Engineering, SE-13 (12), pp1348-1363, 1987.

Agrawal1992 D.Agrawal and A.Abbadi, “A non-restrictive concurrency control for object-oriented databases”, Proceedings of the Third International Conference on Extending Data Base Technology, Vienna, Austria, March, pp. 469-482,1992.

Anand2009 A.Anand and M.Sethi, “Prevention of Deadlock in a Distributed Computing Environment”, US Patents, Patent No. 0138886A1, 2009.

Andrews1982 G.R.Andrews and G.M.Levin, “On-the-fly Deadlock Prevention”, Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing, New York, USA, pp. 165- 172,1982.

Badrinath1988 B.Badrinath and K.Ramamritham, “Synchronizing transactions on Objects”, IEEE Transactions on Computers, volume 37(5), pp. 541-547, 1988.

Badrinath1992 B.Badrinath and K.Ramamritham, “Semantic-based concurrency control: beyond commutativity”, ACM Transactions of Database Systems, volume 17 (1), pp 163-199, 1992.

Bannerjee1987 J.Bannerjee, W.Kim, H.J.Kim and H.F.Korth, “Semantics and Implementation of Schema evolution in Object–Oriented Databases”, Proceedings of 1987 ACM SIGMOD international conference on Management of data, New York, USA, pp. 311-322, 1987.

Bertino1991 Elisa Bertino and Lorenzo Martino, “Object Oriented Database Management Systems: Concepts and Issues”, IEEE Transactions on Computers, volume 24(4), pp. 33-47, 1991.

Braubach2011 L. Braubach and A.Pokahr, “Intelligent Distributed Computing V”, Proceedings of the 5th International Symposium on Intelligent Distributed Computing (IDC 2011), Springer, pp141-151, 2011.

Brzezinski1995 J.Brzezinski, J.M. Helary, M.Raynal and M. Singhal, “Deadlock models and a General Algorithm for Distributed Deadlock Detection”, Journal of Parallel and Distributed Computing, volume 31, pp112-125, 1995

Carey1993. M.J.Carey, D.J.Dewitt and J.F.Naughton, “The 007 benchmark”, Proceedings of ACM SIGMOD Conference on Management of Data, Washington D.C, USA, pp. 12-21, 1993.

Carey1994 M.J.Carey, D.J.Dewitt, Chander Kant and J.F.Naughton , “A status report on the 007 OODBMS benchmarking effort”, Proceedings of OOPSLA, Portland, OR, USA, pp. 414-426, 1994.

Cart1990 M. Cart and J. Ferrie, “Integrating concurrency control into an object-oriented database system”, Proceedings of the Second International Conference on Extending Data Base Technology, Venice, Italy, March, pp. 363-377, 1990.

Chandy1983 K.Mani Chandy and Jayadev Mishra, “Distributed Deadlock Detection”, ACM Transactions on Computer Systems, volume 1(2), pp. 144-156, 1983.

Chow1991 Y.Chow, W.F.Klostermeyer and K.Luo, “Efficient techniques for Deadlock Resolution in Distributed Systems”, Proceedings of the Fifteenth IEEE Annual International Computer Software and Applications Conference, pp. 64-69, sep. 1991.

Chowdhary1989 A.N.Chowdhary, W.H.Kohler, J.A.Stankovic and D.Towsley, “A modified priority based probe algorithm for distributed deadlock detection and resolution”, IEEE Transactions on Software Engineering., volume SE-15, pp. 10-17, Jan. 1989.

Coffman1971 E.G.Coffman, M.J.Elphick and A.Shosani, “System Deadlocks”, ACM Computing Surveys, Volume 3(2), pp. 67-78, 1971.

Cummins2001 F.A.Cummins, “Distributed Object System with Deadlock Prevention”, US Patents, Patent No. US 6236995B1, 2001.

Davidson1993 Susan Davidson, Insup Lee and Victor Fay Wolfe, “Deadlock Prevention in Concurrent Real time Systems”, Kluwer Academic Publishers, Netherlands, Real time Systems, pp. 305 – 318, 1993.

Eswaran1976 K.P. Eswaran, J.N. Gray, R.A.Lorie and I.L. Traiger, “The notion of consistency and predicate locks in a database system”, Communication of ACM, volume 19(11), pp. 624-633, 1976.

Fulton1966 Fulton, L.James, “Extension Classes, Python Extension Types Become Classes”, http://www.digicool.com/releases/ Extension Class., 1996.

Fulton1999 Fulton, L.James, “Zope Object Database Version 3 UML model”, http://www.zope.org/Documentation/Models/ZODB, 1999.

Garey1979 M.Garey and D.Johnson, “Computers and Intractability: A Guide to the Theory of NP Completeness”, W.H. Freeman and Company, New York, 1979.

Garza1988 J.Garza and W.Kim, “Transaction management in an object-oriented database system”, Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, IL, USA, June, pp. 37-45, 1988.

Gray1976 J.N.Gray et al., “Granularity of locks and degrees of consistency in shared database, Modeling in Database management system”, G.M. Nijssen edition, Elsevier, North Holland, pp 393-491, 1976.

Gunther1981 Klaus D.Gunther, “Prevention of Deadlocks in Packet Switched Data Transport Systems,” IEEE Transactions on Communications, Volume 29(4), pp.512-524, 1981.

Hac1989 Anna Hac, Xiaowei Jin and Jo-Han Soo, “A Performance comparison of Deadlock Prevention and Detection Algorithms in a Distributed File System”, Eighth International Annual Conference on Computer and communications, 1989.

Hansdah2002 R.C.Hansdah, Nilanjan Gantait and Sandeep Dey, “A Fault Tolerant distributed Deadlock Detection Algorithm”, Lecture Notes in Computer Science, Springer link, Vol. 2571, pp. 78-87,2002**.**

Henderson1997 Brian Henderson-Sellers, “Towards the formalization of Relationships for Object Modeling”, Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems- TOOLS-25, Melbourne, Australia, pp. 267-283, 1997.

Holt1972 Richard Holt, “Some deadlock properties of Computer Systems”, ACM Computing Surveys, Volume 4(3), pp. 179-196, 1972.

Huang1990 Jong-His Huang and Feng-Jian Wang, “Some Design Issues of a Distributed Object oriented System”, Second IEEE workshop on future trends of distributed computing systems, 1990.

Jun1998 W.Jun and Le Gruenwald, “An Effective Class Hierarchy Concurrency Control Technique in Object – Oriented Database Systems”, Elsevier Journal of Information and Software Technology, pp 45-53, 1998.

Jun2000 Woochun Jun, “A multi-granularity locking-based concurrency control in object oriented database system”, Elsevier Journal of Systems and Software, pp 201-217, 2000.

Kim1989 W.Kim, E.Bertino and J.F.Garza, “Composite Objects revisited”, Proceedings of the ACM SIGMOD international conference on Management of data, New York, USA, pp. 337-347, 1989.

Kim1990 W.Kim, “Introduction to object-oriented databases”. MIT Press, Cambridge, MA, USA. 1990.

Kim1991 W.Kim, T.Chan and J.Srivastava, "Processor Scheduling and concurrency control in real-time main memory databases", IEEE symposium on Applied Computing, Kansas City, MO, USA, pp. 12-21, April 1991.

Lee1996 S.Y.Lee and R.Liou, "A multi-granularity locking model for concurrency control in object-oriented database systems", IEEE Transactions on Knowledge and Data Engineering, volume 8(1), pp. 144-156, 1996.

Lewis2008 Russell Lee Lewis, "Preventing Deadlocks", US Patents, Patent No. 0168448 A1, 2008.

Li1993 Pei-yu Li and Bruce McMillin, "Fault-tolerant Distributed Deadlock Detection/ Resolution", IEEE Transactions on parallel and distributed systems, pp. 224-230, 1993.

Lin1994 X.Lin, M.E.Orlowska and Y.Zhang, "An Optimal Victim Selection Algorithm for removing Global Deadlocks in Multidatabase Systems", 9th International Conference on Frontiers of Computer Technology, FCT'94, IEEE CS Press, pp.501-505, 1994.

Lin1996 Xuemin Lin and Jian Chen, "An Optimal Deadlock Resolution Algorithm in Multidatabase Systems", Proceedings of ICPADS, Nedlands, Western Australia, pp. 516-521, 1996.

Malta1993 C.Malta and J.Martinez, "Automating fine concurrency control in object-oriented databases", Proceedings of the Ninth IEEE Conference on Data Engineering, Vienna, Austria, pp. 253-260, April 1993.

Milicev2007 Dragan Milicev, "On the Semantics of Associations and Association Ends in UML", IEEE Transactions on Software Engineering, volume 33(4), pp 238-251, April 2007.

Mitchell1984 D.P.Mitchell and M.J.Merrit, "A distributed algorithm for deadlock detection and resolution", Proceedings of third ACM Symposium on Principles of Distributed Computing, Vancouver, Canada, pp. 282-284, Aug 1984.

Moon1997 A.Moon and H.Cho, "Performance Analysis of Global Concurrency Control Algorithms and Deadlock Resolution Strategies in Multi database Systems", IEEE Pacific Rim conference on communications, computers and signal processing, 1997.

Newton1979 G.Newton, "Deadlock prevention, detection and resolution: An annotated bibliography", ACM-SIGOPS OS Review, volume 13(4), pp. 33-44, 1979.

ObjectStore2011 **http://www.progress.com/en/objectstore/index.html**, retrieved in 2011.

Olsen1995 D.Olsen and S.Ram, "Towards a comprehensive concurrency control mechanism for object-oriented databases", Journal of Database Management, volume 6(4), pp. 24-35, 1995.

Ozsu1999 M.Tamer Ozsu and Patrick Valduriez, "Principles of Distributed Database Systems", Pearson Education, second edition, 1999.

Parnas1972 D.L.Parnas and A.N.Habermann, "Comment on deadlock prevention method", Communications of ACM, volume 15(9), pp. 840-841, 1972.

Pender2003 Tom Pender, "UML 2 Bible", Wiley Publishing Inc., First Edition, 2003.

Preparata1967 F.P.Preparata, G.Metze and R.T.Chien, "On the connection assignment problem of diagnostic systems", IEEE Transactions on Electronic Computers, Volume EC-16(6), 1967.

Reddy1993 P.Krishna Reddy and Subhash Bhalla, "Deadlock Prevention in Distributed Database Systems", ACM SIGMOD Record, volume 22(3), pp. 40 -46, 1993.

Roesler1988 M.Roesler, W.A.Burkhard and K.B.Cooper, "Efficient deadlock resolution for lock-based concurrency control schemes", IEEE 8th International conference on Distributed Computing Systems, pp. 224-233, 1988.

Riehle2000a Dirk Riehle and S.P.Berczuk, "Types of Member Functions in Java", Report, 2000.

Riehle2000b Dirk Riehle and S.P.Berczuk, "Properties of Member Functions in Java", Report, 2000.

Saha2009 D.Saha and J.Morrissey, "A Self–Adjusting Multi-Granularity Locking Protocol for Object–Oriented Databases", Second IEEE International Conference on the Applications of Digital Information and Web Technologies, pp. 832-834, 2009.

Servio1990 Servio, Servio Logic Corporation, "Transactions and Concurrency Control", Gemstone Product Overview, Alameda, CA, USA, 1990.

Shaw1974 A.C.Shaw, "The Logical Design of Operating Systems", Prentice-Hall, Englewood Cliffs, NJ, 1974.

Sinha1985 M.Sinha and M.Natarajan, "A Priority based Distributed Deadlock Detection Algorithm", IEEE transactions on Software Engineering, volume SE-11, pp. 67-80, 1985.

Singhal1989 Mukesh Singhal, "Deadlock Detection in Distributed Systems", IEEE survey and tutorial series, Vol. 22, pp. 37-48, 1989.

Srivastava2007 Alok Kumar Srivastava and Wilson Wai Shun, “Victim selection for deadlock detection”, United states patent, No. US7185339B2, 2007.

Stevens2002 P.Stevens, “On the Interpretation of Binary Associations in the Unified Modeling Language “, volume 1(1), pp. 68-79, 2002.

Szyperski2002 C. Szyperski, D. Gruntz and S. Murer, “Component Software – Beyond object -oriented programming”, Pearson Education, second edition, 2002.

Terekov1999 I.Terekov and T.Camp, “Time efficient deadlock resolution algorithms”, Information Processing Letters, Elsevier Science, pp. 149 -154, 1999.

Versant2008 Versant, “TechView Product Report”, Versant Object Database, www.odbms.org, 2008.

Weikum2005 G.Weikum and G.Vossen, “Transactional Information Systems”, ACM, 2005.

Wu1997 Shengli Wu and Nengbin Wang, “Directed Graph based Association Algebra for Object Oriented Database”, Proceedings of IEEE conference on Technology of Object oriented languages-TOOLS-24, Washington DC, USA, pp. 53-59, 1997.

Zobel1988 D.Zobel, C.Koch, “Resolution Techniques and complexity results with deadlocks: A classifying and annotated bibliography”, OS Review 22 (1), pp. 52-72, 1988.

LIST OF PUBLICATIONS

PUBLISHED PAPERS

1. V Geetha and N. Sreenath, ”Distributed Deadlock Detection using Fault Informing Probes.”, International Journal of Computer Applications Vol. 41 no. 8, pp6-11, March 2012, Published by Foundation of Computer Science, New York, USA. ISSN: 0975 – 8887.

2. V Geetha and N. Sreenath. “Augmenting the Performance of Existing OODBMS Benchmarks”, International Journal of Computer Applications vol. 40, no. 5, pp41-46, February 2012, Published by Foundation of Computer Science, New York, USA.ISSN:0975 – 8887.

3. V. Geetha and N. Sreenath, “Impact of Object Operations and Relationships on Concurrency Control in DOOS”, Tenth International Conference on Distributed Computing and Networking, Kolkata, India, Proceedings in LNCS, pp258-264, January 2010.

4. V. Geetha and N. Sreenath, ”A Multi-Granular Lock Model for Distributed Object Oriented Databases Using Semantics”, Seventh International Conference on Distributed Computing and Internet Technology, Bhubaneshwar, India, proceedings in LNCS, pp138-149, February 2011.

5. V. Geetha and N. Sreenath, " Fault-informant distributed deadlock detection using colored probes", Second International Conference on Advances in Communication, Network, and Computing, Bangalore, India, Proceedings in LNCS-CICS, March 2011.

6. V. Geetha and N. Sreenath, “Deadlock Prevention in Distributed Object Oriented Systems.”, International Conference on Advances in Computing C 2011, Cochin, India, proceedings in LNCS- CCIS, pp48-57, August 2011.

7. V. Geetha and N. Sreenath., ”Semantic Based Concurrency Control in OODBMS”, International Conference on Recent Trends in Information Technology, Chennai, India, Proceedings in IEEE Computer Society, June 3-5, 2011.

8. V. Geetha and N. Sreenath, “High Concurrency for Continuously Evolving OODBMS”, Eighth International Conference on Distributed Computing and Internet Technology, proceedings in LNCS, pp 94-105, 2012.

<u>COMMUNICATED PAPERS</u>

9. V.Geetha and N.Sreenath, “Preventing Deadlocks and Starvation in Distributed Object Oriented Systems”, submitted revised paper to Journal of Electrical Engineering and computer science, Elsevier.

10. V.Geetha and N. Sreenath, “Semantic multi-granular lock model in Object Oriented Database Systems”, submitted to Journal of Object Technology.

11. V.Geetha and N.Sreenath,” Performance Analysis of Victim Selection Algorithms in Distributed Systems and Proposal of Weight based Resolution Strategy”, submitted to International Journal of Artificial Intelligence and Information technology, MECS publishers.

12. V.Geetha and N.Sreenath,” Semantic Concurrency Control on Continuously Evolving OODBMS using Access Control Lists “, submitted to ninth International Conference on Distributed Computing and Internet Technology**,** 2013.

VITAE

V.GEETHA, the author of this thesis is currently working as Assistant Professor (Senior) in the Department of Information Technology at Pondicherry Engineering College, Puducherry, India. She was born in march1969 at Puducherry. She received her B.Tech. degree in Computer Science and Engineering from Pondicherry University, Puducherry, India in the year 1990 and M. Tech. degree in Computer Science and Engineering from Pondicherry University, Puducherry in 1999. She worked as Lecturer and Senior Lecturer in Computer Science in the Government Polytechnic Colleges, Puducherry for eleven years, since 1991. Later she joined as Lecturer in the Department of Computer Science & Engineering and Information Technology, Pondicherry Engineering College, Puducherry, in the year 2002. Subsequently she was promoted as Senior Lecturer in the Department of Information Technology in the year 2007. Her areas of interest include Object-oriented Design, Distributed Systems and Middleware Technologies. She has published research papers in International Journals and Conferences.