Semantic Multi-granular Lock Model in Object Oriented Database Systems

Venkatasubramanian Geetha[1] and Niladuri Sreenath[2]

[1]*Department of Information Technology*
[2]*Department of Computer Science & Engineering*
Pondicherry Engineering College, Puducherry-605104, India
vgeetha@pec.edu

Abstract

In this paper, a semantic based multi-granular lock model for object-oriented database systems is proposed. It addresses the concurrency control issues related to all types of transactions to an object: run time transactions and design time transactions. In the case of run time transactions, it ensures consistency while providing fine granularity. In the case of design time transactions, the proposed work has the following features: First, it provides separate lock modes for all types of design time operations. Second, it provides fine granularity for design time transactions. Third, it reduces deadlocks due to lock escalation. The proposed work shows how concurrency can be maximized while ensuring consistency for all types of transactions. A simulation model is constructed to evaluate the performance of the proposed work. This model is used to compare the proposed work with two existing techniques. The performance results show that proposed scheme is better than existing works.

Keywords: *object oriented databases, concurrency control, multi granular lock model, class relationships, design time transactions, run time transactions*

1. Introduction

Object Oriented Database System (OODBMS) is widely used in 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 (run time transactions) or schema (design time transactions). A transaction in OODBMS is defined as partially ordered set of method invocations on objects [1]. A typical run time 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 run time 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 [25]. 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. Since it is 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. This class diagram is called as class lattice. Class lattice is a group of

classes related by all types of relationships namely inheritance, aggregation and association. They are graph structured and may have cycles. 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.

Concurrency control mechanisms are applied to synchronize the transactions accessing the database to maintain its consistency. Complex concurrency control mechanisms are needed for OODBMS because of its complex nature. However, the concurrency control mechanisms should not affect the performance of OODBMS. 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. Among the concurrency control mechanisms such as locking, time stamp ordering and optimistic concurrency control, locking is widely used because of its easy implementation.

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, they can execute. If they 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. Usually transactions in CAD, CAM applications using OODBMS are long duration transactions. Preemption of such long transactions due to incompatibility wastes system resources. At the same time, letting a transaction to hold resources for longer duration may delay other transactions and reduce the throughput.

Multi- Granular Lock Model (MGLM) is a common technique for implementing concurrency control on transactions using the OODBMS. The main advantages of MGLM are high concurrency and minimal deadlocks. Using MGLM, transactions can request the same database in different granule sizes varying from coarse granules to fine granules. The hierarchy of granules in OODBMS is as follows: 1. Class lattice, 2. Subclass lattice/ class hierarchy, 3. class, 4. object, 5. attribute and methods. Subclass lattice is a subset of class lattice and share the same properties as that of class lattice. Class hierarchy is a group of classes related by all relationships excluding relationships like multiple inheritance, shared aggregation etc. that form cycles.

Though there are several semantic based MGLM [11, 16, 17, 18, 19, 21], none of them has fully exploited the semantics of attributes, methods and class relationships to maximize the concurrency of design time transactions. They have also not proposed fine granularity lock modes for all types of design time transactions. This paper aims in proposing a MGLM, which will improve the degree of concurrency for design time transactions and run time transactions by fully utilizing the semantics of object oriented features.

In this paper, a multi-granular lock model is proposed to increase concurrency among transactions. The proposed scheme has these characteristics: First, it proposes fine granularity for design time transactions. Second, it provides separate lock modes covering all types of operations possible by design time transactions based on [4]. Third, it shows how fine granularity of run time transactions can be provided without affecting consistency. Fourth, it proposes enhanced compatibility matrix between all design time transactions and run time transactions. The proposed scheme provides more parallelism between design time transactions and run time transactions.

The paper is organized as follows. In the next section, related works are analysed for their merits and demerits. In Section 3, the proposed scheme is described in detail. In Section 4, the performance of proposed scheme is compared with existing works. Section 5 concludes the paper.

2. Related Works

The various concurrency control schemes can be assessed based on the level of concurrency they provide for parallel execution of design time and run time transactions without compromising on consistency. These two types of transactions induce three different types of conflicts among transactions to a class lattice: conflicts among run time transactions, conflicts among design time transactions and conflicts between run time and design time transactions. In this chapter, literature related to these types of access conflicts is discussed.

2.1. Conflicts among Run Time Transactions

MGLM was introduced [15] for relational databases. Intension locks are used to infer the presence of locked resources at smaller granule level. The lock modes defined in this paper 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). The existing MGLM based concurrency control schemes for OODBMS provide compatibility among transactions in two ways: based on relationships and based on commutativity.

One group of works proposes concurrency control based on the relationships namely inheritance, aggregation and association between the objects. Separate lock modes are defined for each of the above relationships. MGLM was first extended to object oriented databases, *i.e.*, ORION [11]. In this paper, MGLM is defined for objects related by inheritance and exclusive aggregation only. They have applied the lock modes defined by[15] for OODBMS. The locks defined in [11] are of granularities of classes (collection of instances) and instances. Later [18] 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 [11], new lock modes like ISOS, IXOS, SIXOS are added to support shared aggregation. In [12], these shared intension locks are extended to shared inheritance (multiple inheritance) also. In [16], concurrency control for run time transactions on classes related by inheritance is proposed. The smallest granule in all these papers for run time transactions is only up to instance level and all of them have proposed MGLM based on relationships only. In [26], 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. Further, their granularity is restricted to object level.

In the second group of concurrency control schemes, compatibility is defined based on commutativity. They require application programmers to perform semantic analysis on the source code of methods (member functions) of the class. In [1], attribute is the smallest granularity supported. They state that any two methods in a class can be

executed in parallel, 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 [3,] 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 a-priori knowledge of all possible outcomes of all methods. In [1], 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.

Commutativity of methods [22] was proposed to resolve lock conflicts between run time transactions. In this, 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 run time, by determining it apriori using direct access vectors (DAV) [2]. 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 [17], fine granularity of run time 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. In [13], it is pointed out 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 ad 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 [24]. . 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 while trying to provide fine granularity for run time transactions. In ZODB[9], concurrency control of runtime requests are based on timestamps.

2.2. Conflicts among Design Time Transactions

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 [19, 4], 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

- Modifying the definition of attributes defined in the class such as changing the name and domain;
- Adding/ deleting attribute;
- Adding/ deleting method;
- Modifying interface (signature) or implementation of methods;
- Creating/ deleting instances;
- Move an attribute from one class to another class;
- Move a method from one class to another class.

In [4], the changes to class hierarchy are classified into changes to the nodes and changes to the links. Changes to the node involve

- Adding a new class;
- Dropping an existing class;
- Changing the name of a class;
- Move a class from one position in class lattice to another position.

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

- 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.

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 [4], 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.

In [21], all the above operations are done using only one lock mode by locking the entire schema with Read Schema (RS) and Write Schema (WS) lock modes. In [22], lock mode for changing the class definition (class contents) is provided by RD (Read Definition) and MD (Modify Definition) lock modes. They have omitted the other types of schema changes. In [1], they 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 [21,22], transactions modifying class relationships are serialized and no other run time transactions or design time transactions are allowed to execute in parallel, *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 [4].

In [17], 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 this, 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 [17], 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 [17] at any time, only one change in class relationship is allowed at a time in every class. It does not take into account the structural modifications between classes, *i.e.*, it considers intra class relationships only and excludes inter class relationships. Then it can be observed that all class level and class lattice level operations represented by MCR (Modify Class Relationship) lock mode blocks all the other design time transactions along with run time transactions. So in all these existing works, the granularity of design time transactions is still coarse. The transactions that modify class relationships are serialized. In ZODB [9, 10], the design time transactions is handled in 4 ways. Change is accommodated by the ZODB in a number of ways. Changes in object methods are easily accommodated because classes are not stored in the object database. Changes to class implementation are reflected in 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 [28], 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.

2.3. Conflicts between Run Time Transactions and Design Time Transactions

In run time 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 and is chosen depending on the type of transaction. During run

time 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, run time transactions accessing them should not be allowed.

In [11], 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 access to the same class. Moreover, the same S and X lock modes are used for run time transactions also. This scheme provides limited concurrency since a class definition read does not commute with any run time transaction. Actually, a class definition read commutes with an instance write as described in [7]. In [7], 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 run time lock modes, concurrency between class definition accesses (class definition read and class definition write) and run time 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 [22], MD blocks any other instance access as well as RD and MD, since MD lock does not commute with any other lock modes. In [27], 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, [21] 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 [1] 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 [23], 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 [17], 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 every attribute in all classes to maintain its 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. The MCR lock mode in [12] is split into MCD and MCR to improve concurrency [13]. The lock mode MCD can be used to add a new class, delete an existing class and to change a class name. The lock mode MCR is for changing the relationships between classes and changing the position of a class in class lattice. However as pointed in Section 2.2, this operation set is not complete.

3. Proposed Scheme

The proposed scheme aims in providing the following objectives. It ensures consistency among run time transactions and provides fine granule locking for design time transactions. This will not only improve concurrency between design time transactions, but will improve concurrency between run time transactions and design time transactions also. The principles based on which concurrency is improved among design time transactions and between run time transactions and design time transactions are discussed in the following sections. In the next section, we will see how consistency is preserved between run time transactions.

3.1. Consistency among Run Time Transactions

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			

In [17]' 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 [22], (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, that is, 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 [17], the dependency of a class with other classes is not considered while constructing the DAV of this class. The DAV proposed in this paper is correct and works fine 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[24]. 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 (Jun, 2000) 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 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 [22], 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 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 NF), name of the initial DAV of the first breakpoint (in the form of NI) 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, AF, AI, 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 a method execution, 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.

Table 1 gives the commutativity tables constructed in the proposed scheme to that of [17]. The shaded portion shows the Jun's scheme. The proposed scheme shows where consistency is to be preserved. Table 2 gives the commutativity table for proposed scheme and the shaded portion in the scheme proposed in [22]. Note that Y and N denote ``compatible'' and ``incompatible'' status of transactions respectively. In Jun'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 Jun's scheme. The proposed 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.

Table 1. Example of the Commutativity Matrix to Show the Consistency Requirements on Interface Inheritance for Proposed Scheme and Jun's Scheme

	Lock granted													
LOCK REQ		AF	AI	A1	A2	A3	BF	CF	CI	C1	C2	DF	DI	D1
	AF	N	N	N	N	N	N	Y	Y	Y	Y	N	N	N
	BF	N	Y	Y	Y	N	N	Y	Y	Y	Y	N	Y	N
	CF	N	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	N	N
	DF	N	N	N	N	N	N	Y	Y	Y	Y	N	Y	N

Table 2. Example of the Commutativity Matrix to Show the Consistency Requirements on Interface Inheritance for Proposed Scheme and Malta Scheme

	Lock Granted				
Lock Requested		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

The proposed concurrency control is based on two-phase locking[8]. 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. Thus, the proposed scheme works well provided the OODB system is stable.

3.3. Concurrency among Design Time Transactions

The proposed scheme defines the lock modes with 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 locking 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 proposed scheme aims to cover all the operations that could be done on schema as defined in [19, 4]. It can be seen at three levels: at node (class) contents level, at node level and at link level. The node contents are instances, attributes and methods [4].

By 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 component class can be modified only in component classes where as composite objects can only read them. Similarly the attributes in associated class can be modified only in associated classes. The associative classes cannot alter them. In simple words, modification is possible only in the class in which the attributes are defined.

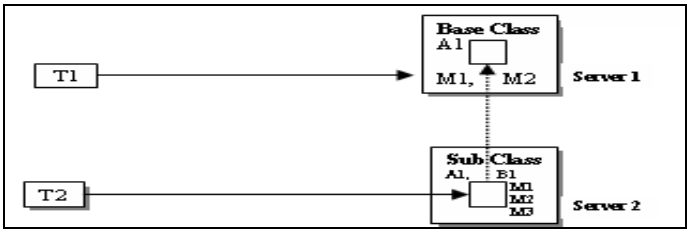


Figure 1. Locking in Inheritance

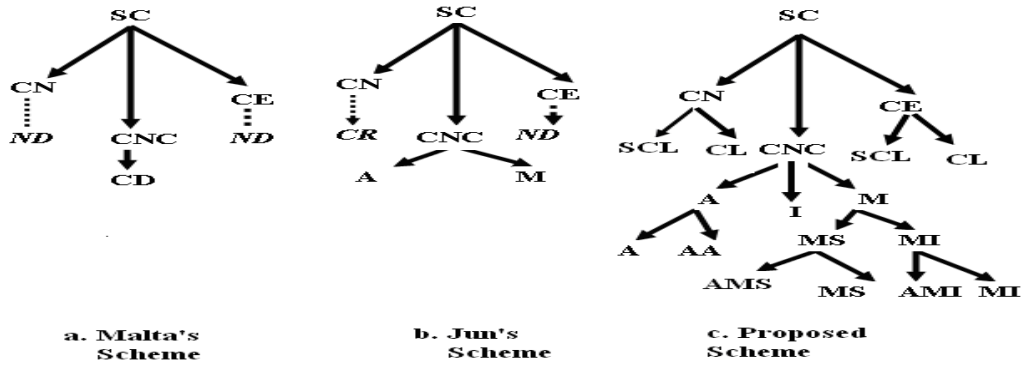
Therefore, the attributes from the base classes, component classes and associated classes can be viewed as adapted attributes in the subclass or composite class or associative class. In [17], adapted attributes and attributes defined in this class cannot be accessed in parallel even though they are mutually exclusive, without using AAV. It also does not address the need for controlling parallel execution of modifying an attribute definition in base class (component class/associated class) while reading the same attribute definition in subclass (composite class/associative class). Allowing this will lead to dirty reads.

By object oriented semantics, modifying 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 [17], all the operations related to modification of methods are done in the same lock mode.

For example, consider Figure 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 [24], whereas M2 is overridden in subclass [called as hook method [24]. M3 is a method defined in subclass. In [17], in the subclass, all types of attributes, *i.e.*, A1 and B1 are locked by a single lock mode and all methods, *i.e.*, M1, M2 and M3 are locked by a single lock mode. 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 1. Attribute A1 can be read in both base class as well as subclass. However, modifying A1 is possible only in base class. It is worth noting that while base class is modifying definition of A1, no transaction should be allowed in 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 namely, 1) Attributes adapted from other classes and 2) 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 this class. So, all the subclasses will have adapted attributes and attributes defined in that class. However as base classes do not have any parents, their adapted attributes list will be empty.

In Figure 1, M1 is a template method whose interface and implementation can be modified only in base class. It can only be read in subclasses. M2 is a hook method. Therefore, its interface is modifiable only in 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 base class. In [17], MM is the only mode to handle all these method types. Similarly, interface and implementation of methods defined in 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.



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

Figure 2. Hierarchy of Granules of Design Time Transactions

The interface, *i.e.*, method definition is independent of method implementation [28]. 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 implementation. The implementation of 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 format. In fact, the interface is viewed as a contract between client and server. So lock mode for method is split into Method Signature (MS) and Method Implementation (MI) in the proposed scheme. The interfaces or definitions of methods in base classes, component classes and associated classes are separately maintained in the class. Hence separate lock modes can be defined for reading interfaces of adapted class methods (RAMS) and for reading and modifying subclass methods (RMS, MMS). Since interfaces of methods in adapted class cannot be modified in this class, lock mode for modifying the interface of these adapted methods is not available in this class.

Hence, MI is further split into Adapted Method Implementation (AMI) and Method Implementation (MI) in the proposed scheme. Therefore, implementation can be read or modified by lock modes (RAMI, MAMI, RMI and MMI). There is only read mode available for adapted method implementations for all methods except hook methods. MAMI represents modification of hook methods and MMI is for modifying methods defined in this class. However, when 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 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. Similarly new lock modes AI (Add Instance), DI (Delete Instance) are defined to insert and delete instances for a class.

Modifying class relationship involves adding, deleting a class or moving its location in the class lattice. In [17], 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 is not complete definition of class relationships. Modification operations of schema are categorized [4] as in Section 2.2. Then it can observed that class lattice level operations represented by MCR (Modify Class Relationship) lock mode in [17] blocks all the class level as well as other class lattice level operations along with run time transactions.

In [17], 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 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 sets of methods defined or inherited into the class. If the operations defined in [17] are compared with [4] as given in Section 2.2, it can be inferred that operations defined in Jun is only a subset of [4]. 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 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 lattice.

Table 3. Proposed Compatibility Matrix for Design Time Transactions

	AA	RAA	RA	MA	DA	AM	RAMS	RMS	MMS	RAMI	MAMI	RMI	MMI	DM	AI	DI	R S C L	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

The operations like move an attribute from one class to another class, move a method from one class to another class and move a class from one position to another position in 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 4. Commutativity Matrix for Transactions Accessing Defined Class and Adapted Classes

		Adapted Classes			
D		RAA	RAMS	RAMI	MAMI
e	AA	N	Y	Y	N
f	RA	Y	Y	Y	Y
i	MA	N	Y	Y	N
n	DA	N	N	N	N
e	AM	Y	N	N	N
d	RMS	Y	Y	Y	Y
C	MMS	Y	N	N	N
l	MMI	Y	Y	N	Y
a	RMI	Y	Y	Y	Y
s	DM	Y	N	N	N

Table 3 defines the commutativity matrix defined for class lattice. Table 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. The hierarchy of granules in design time transactions in [22, 17] and proposed scheme can be summarized as given in Figure 2.

Using the commutativity of lock modes in Table 3, a finer granularity lock can be obtained. The lock granularity in the proposed work is one of 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 request is made, Table 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 lock mode is one of AA,RA,MA, DA,AM,MMS, MMI, DM, Table 4 is checked for compatibility.

Let us consider the various scenarios to show how Jun's scheme and proposed 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 atleast 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 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 proposed scheme ensures consistency wherever necessary and improves concurrency wherever possible.

	Jun's Scheme	Proposed 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.

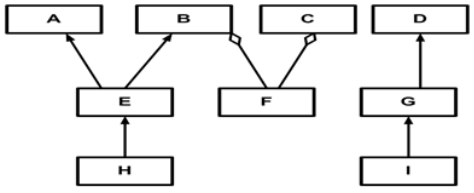
From Figure 2,

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 RV
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 RV

3.3. Concurrency among Design Time Transactions and Run Time Transactions

The run time transactions and design time transactions can have fine granularity by using lock modes as well as access vectors. In Jun's scheme, AAV and MAV are defined to concurrently access more than one attribute or method in the same class. However for other operations the granularity is still coarse. In [13], 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(a). The RAV (Relationship Access 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 in [14], 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 run time operations.



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	-----

Figure 3. (a) Sample Class Diagram; (b) Relationship Access Vector (RAV) for the Sample Class Diagram in Table Format

In general in a class lattice, there is an upward dependency for run time transactions. i.e., whenever a run time transaction is made on subclass, composite class or associative class, its corresponding base class, component class or associative class also has to be locked as given in Section 3.1. On the other hand, there is 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 run time transactions and adapted classes for design time transactions. For example in Figure 3(a), if a run time transaction is made to E, its defined classes A and B are to be locked. If a design time transaction is made to E, its adapted class H is to be locked. This can be defined using Relationship Access Vector (RAV) in table format as in Figure 3(b). From Figure 3(b), it can be observed that classes at leaf level like F, H and I do not influence other classes.

Table 5. Proposed Compatibility Matrix for Design Time Transactions and Run Time Transactions

	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 6. Commutativity Matrix for Transactions Accessing Defined Class and Adapted Classes

Adapted Classes					
D		RAA	RAMS	RAMI	MAMI
e	AA	Δ	Y	Y	Δ
f	RA	Y	Y	Y	Y
d	MA	Δ	Y	Y	Δ
	DA	Δ	Δ	Δ	Δ
C	AM	Y	Δ	Δ	Δ
l	RMS	Y	Y	Y	Y
a	MMS	Y	Δ	Δ	Δ
s	MMI	Y	Y	Δ	Y
s	RMI	Y	Y	Y	Y
	DM	Y	Δ	Δ	Δ

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 5 gives the commutativity matrix for design time and run time transactions. Table 6 is used to ensure consistency among design time transactions that are accessing a defined class and its adapted classes in parallel. 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 an object.

For example, consider the following operations by transactions T1, T2 and T3 for the sample diagram in Figure 3(a). At time t, T1 is doing a run time access on instance I1 of class B. At time t+1, T2 is modifying implementation of 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 run time 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 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 run time 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 this AAV and MAV are maintained for every class to support parallel access of attributes and methods in the same class.

4. Performance Evaluation

The proposed scheme is tested using a simulation model as in [20]. In this section, the simulation model, experimental details and analysis of the results are presented.

4.1. Simulation Model

The simulation model used in the existing works is adopted for testing the proposed scheme. The existing simulation model [20] is used to facilitate easy comparison. The simulation model is implemented using Java. Figure 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.

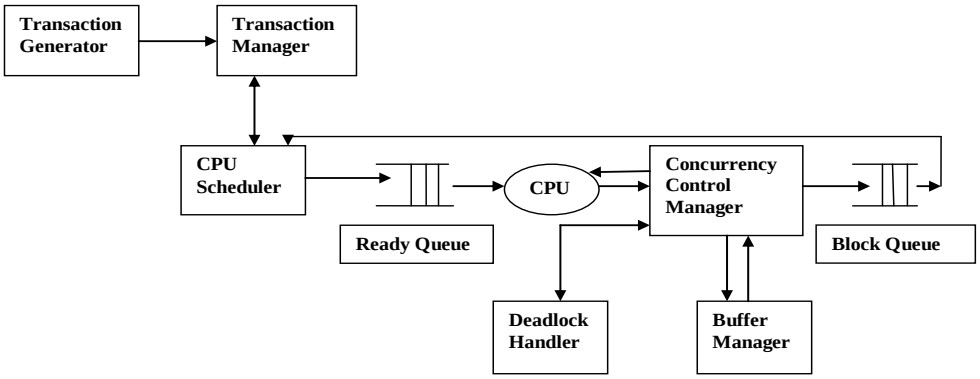


Figure 4. Simulation Model [20]

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 run time 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 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 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.

Table 7. Simulation Parameters

Parameters	Default value(range)
Time to process one operation	0.00000625ms
Mean time to set lock by run time 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 [5]	Small (small, medium, large)

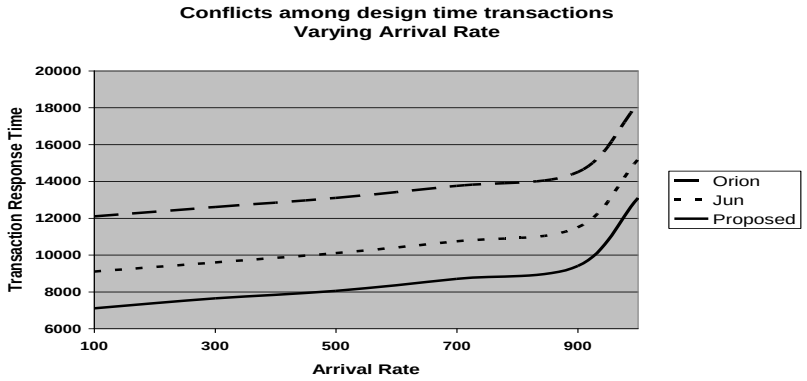


Figure 5. Varying Arrival Rate

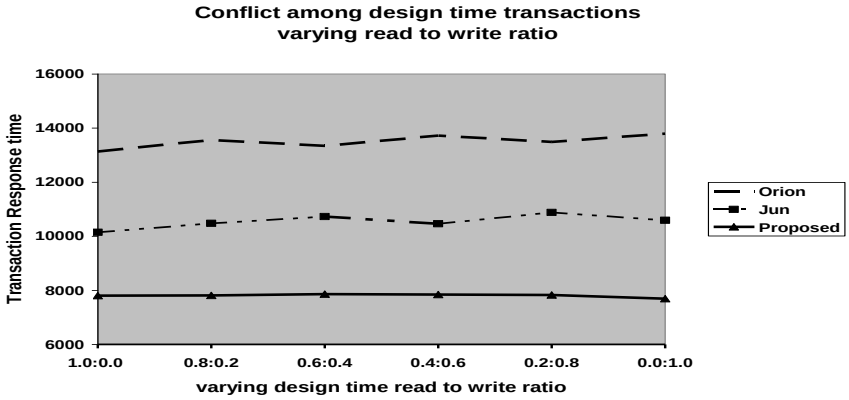


Figure 6. Varying Design Time Read to Write Ratio

4.2 Analysis

007 Benchmark [5, 6] is well known for testing performance of OODBMS. It is used in [17 for showing the performance of his proposal. But 007 benchmark defines the benchmark only for run time transactions. It does not define any testing cases for design time transactions. So it cannot be fully adopted for the proposed scheme. However in the proposed scheme, some of the aspects of 007 bench mark are adopted for performance evaluation. The database model and testing cases of run time transactions are adopted. 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 framed to cover all three types of schema changes. Table 7 gives the simulation parameters.

Three testing cases are chosen to measure the performance the proposed scheme for all types of transactions: Varying arrival rate of the transactions, varying read-to-write ratio of design time transactions and varying run time transaction to design time transaction ratio. The existing schemes chosen to compare against the proposed scheme are Orion scheme and Jun scheme. Orion scheme is chosen as it is the best scheme based on relationships. Jun's scheme is chosen as it is the latest scheme based on access vectors. As the proposed scheme is also based on access vectors, it is compared against these two schemes.

Figure 5 shows the test case of varying arrival rate. This test is done to evaluate how the schemes work under various system loads. Orion performs the worst. Jun's scheme is better than Orion. The proposed scheme works better for all the loads.

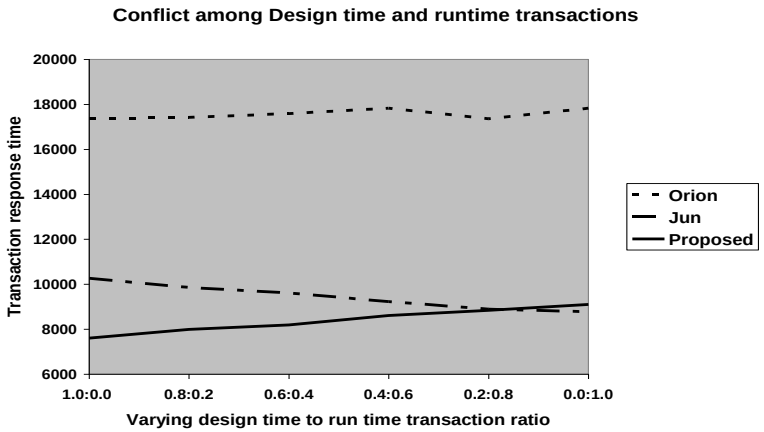


Figure 7. Varying Design Time to Run Time Transaction Ratio

The average lock waiting time of Orion, Jun's scheme and proposed scheme are 46.12, 35.43 and 28.12ms respectively. Thus the proposed scheme takes least waiting time. It is because of the coarse to fine granularity they provide. Orion locks the entire instance object for run time transaction 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 Jun's scheme fine granularity is achieved for run time transactions at the level of attributes with break points. Medium granularity is achieved for class content design operations by accessing attributes and methods. For other operations the entire class lattice is locked. In the proposed scheme, further concurrency is enhanced by introducing new lock modes. On the whole, proposed

scheme is better than Orion by 46.88% and Jun's scheme by 18.5%. Jun's scheme is better than Orion by 34.8%.

Figure 6 shows the test case of varying design time read-to-write ratio. In this also, Orion performs the worst. 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 Jun's scheme, the granularity is refined for class content modification. But other design operations are still at coarse level. In the proposed 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, Jun's scheme and proposed scheme are 49.12, 35.43 and 28.12ms respectively. On the whole, proposed scheme is better than Orion by 68% and Jun's scheme by 32.1%. Jun's scheme is better than Orion by 28.04%.

Figure 7 shows the test case of varying design time to run time ratio. Orion takes highest response time. As it takes coarse granularity for both types of transactions, its response time is very high. Jun's scheme is better than Orion because it provides fine granularity for run time transactions and medium granularity for design time transactions. However the proposed scheme performs the best. This is because of fine granularity of subclass lattice level operations with the help of class dependency vector. In the graph, it can be noted that Jun's scheme response time is lower than proposed scheme for higher ratio of run time transactions. This is because proposed scheme blocks inconsistent run time transactions while it is not checked in Jun's scheme.

5. Conclusion

In this paper, a concurrency control scheme is proposed based on multiple granularities to provide fine granularity among design time transactions and run time transactions. The proposed 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 run time accesses. It minimizes the need for AAV and MAV used in Jun's scheme by proposing rich set of lock modes based on semantics of object oriented concepts. Fine granularity on modifying class relationships is proposed by defining relation vectors and splitting the lock modes separately for class level changes and class lattice level changes. The objective of this paper is to provide finest granularity on all lock modes to provide highest concurrency, however with the overhead of maintaining AAV, MAV and CDV. Future work is aimed at minimizing this overhead.

References

[1] 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, **(1992)** March, pp. 469-482.

[2] B. Badrinath and K. Ramamritham, "Synchronizing transactions on objects", IEEE Transactions on computers, vol. 37, no. 5, **(1988)**, pp. 541-547.

[3] B. Badrinath and K. Ramamritham, "Semantic-based concurrency control: beyond commutativity", ACM Transactions of Database Systems, vol. 17, no. 1, **(1992)**, pp. 163-199.

[4] J. Bannerjee, *et al.*, "Semantics and Implementation of Schema evolution in Object–Oriented Databases", proc. ACM SIGMOD conference, **(1987)**.
[5] M. Carey, *et al.*, "The 007 benchmark", In: Proceedings of the 1993 ACM SIGMOD Conference on Management of Data, Washington, DC, USA, **(1993)** May, pp. 12-21.
[6] M. Carey, *et al.*, "A status report on the 007 OODBMS benchmarking effort", Proceedings of OOPSLA, Portland, OR, USA, **(1994)**, pp. 414-426.
[7] 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, **(1990)** March, pp. 363-377.
[8] K. Eswaran, *et al.*, "The notion of consistency and predicate locks in a database system", Communication of ACM, vol. 19, no. 11, **(1976)**, pp. 624-633.
[9] J. L. Fulton, "Extension Classes, Python Extension Types Become Classes", **(1996)**, http://www.digicool.com/releases/ExtensionClass.
[10] J. L. Fulton, "Zope Object Database Version 3 UML model", **(1999)**, http://www.zope.org/ Documentation/Models/ZODB.
[11] 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, **(1988)** June, pp. 37-45.
[12] V. Geetha and N. Sreenath, "Impact of Object Operations and Relationships in Concurrency Control in DOOS", International Conference on Distributed Computing and Networking, Kolkata, Proceedings in LNCS, **(2010)**, pp. 258-264.
[13] V. Geetha and N. Sreenath, "A Multi-Granularity Lock Model for Object Oriented Databases using Semantics", International Conference on Distributed Computing and Internet Technologies, Bhubaneshwar, India, Proceedings in LNCS, **(2011)**, pp. 138-149.
[14] 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, **(2011)**, pp. 1313-1318.
[15] J. N. Gray, *et al.*, "Granularity of locks and degrees of consistency in shared database", Modeling in Database management system, G.M. Nijssen ed., Elsevier, North Holland, **(1976)**, pp. 393-491.
[16] W. Jun and Le Gruenwald, "An Effective Class Hierarchy Concurrency Control Technique in Object – Oriented Database Systems", Elsevier Journal of Information and Software Technology, **(1988)**, pp. 45-53.
[17] W. Jun, "A multi-granularity locking-based concurrency control in object oriented database system", Elsevier Journal of Systems and Software, **(2000)**, pp. 201-217.
[18] W. Kim, *et al.*, "Composite Objects revisited, Object oriented Programming", systems, Languages and Applications, **(1990)**, pp. 327-340.
[19] W. Kim, "Introduction to object-oriented databases", MIT Press, Cambridge, MA, USA **(1990)**.
[20] 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, **(1991)** April, pp. 12-21.
[21] S. Lee and R. Liou, "A multi-granularity locking model for concurrency control in object-oriented database systems", IEEE Transactions on Knowledge and Data Engineering, vol. 8, no. 1, **(1996)**, pp. 144-156.
[22] C. Malta and J. Martinez, "Automating concurrency control in object-oriented databases", In: Proceedings of the Ninth IEEE Conference on Data Engineering, Vienna, Austria, **(1993)** April, pp. 253-260.
[23] D. Olsen and S. Ram, "Towards a comprehensive concurrency control mechanism for object-oriented databases", Journal of Database Management, vol. 6, no. 4, **(1995)**, pp. 24-35.
[24] D. Riehle and S. P. Berczuk, "Types of Member Functions in Java", Report, **(2000a)**.
[25] D. Riehle and S. P. Berczuk, "Properties of Member Functions in Java", Report, **(2000b)**.
[26] D. Saha and J. Morrissey, "A self – Adjusting Multi-Granularity Locking Protocol for Object – Oriented Databases", Second International Conference on the Applications of Digital Information and Web Technologies, IEEE, **(2000)**, pp. 832-834.

[27] Servio, Servio Logic Corp., "Transactions and Concurrency Control", In: Gemstone Product Overview, Alameda, **(1990)**, CA, USA.
[28] C. Szyperski, D. Gruntz, S. Murer, "Component Software–Beyond object -oriented programming", Pearson Education, second edition, **(2002)**.
[29] Versant, "TechView Product Report: Versant Object Database", **(2008)**, www.odbms.org.

Authors

V. Geetha

She has received B.Tech (CSE) degree in 1990 from Pondicherry engineering college, Puducherry, India and M.Tech (CSE) degree in 1999 from Pondicherry University. She had been working in polytechnic under PIPMATE for 11 years before joining Pondicherry Engineering College in 2002. She is working as Assistant Professor (Senior) in Information Technology department, Pondicherry engineering college. She has completed Ph.D in Computer science and Engineering in Pondicherry University. Her research areas of interest includes client/ server architecture, distributed systems, object oriented system design and middleware technologies.

N. Sreenath

He has received B.Tech (ECE) degree at JNTU college of Engineering, Anantapur, India in 1987 and M.Tech (CSE) at University of Hyderabad in 1990. He has done his Ph.D in Computer Science and Engineering at Indian Institute of Technology, Chennai, India. He is working in Pondicherry Engineering College since from 1991. He is currently working as Professor of Computer Science and Engineering in Pondicherry Engineering College. He has more than 30 publications in various international conference proceedings and journals. His areas of interest are distributed computing, high speed networks and WDM optical networks.