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. \( \text{RType} \) (resource type) can be attribute definition, attribute value or method definition. \( \text{RType} \) - attribute definition is used by design time transactions. \( \text{RType} \) - 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. \( \text{Resource ID} \) holds the name of the attribute or method. \( \text{Class ID} \) is used to distinguish attributes or methods whose names are used in more than one class. Shared list maintains one more field called \( \text{RefCount} \) to maintain the number of transactions sharing the resource.

Initially, the attributes and methods of all the classes are included in the \( \text{Available} \) 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 \( \text{Shared} \) or \( \text{Exclusive} \) list, depending on the lock mode. Though all the resources are in the \( \text{Available} \) 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 \( \text{Available} \) list. Shared lock mode is allowed, only if the resource is not in \( \text{Exclusive} \) list. It is allowed when the resource is in \( \text{Available} \) list or \( \text{Shared} \) list. Searching the resource in both lists is time consuming. So if it is not available in \( \text{Exclusive} \) list, grant message can be sent and resource can be accordingly updated from \( \text{Available} \) list or \( \text{Shared} \) list in the background. Several transactions can share a resource in read mode. \( \text{RefCount} \) field in \( \text{Shared} \) list is used to count the number of transactions that are currently sharing the resource. First grant message adds the resource to the \( \text{Shared} \) list and sets the \( \text{RefCount} \) to 1. Every grant message after that, increments the count. Every release message decrements it. When the \( \text{RefCount} \) is 0, the resource is removed and added to the \( \text{Available} \) list.
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 IDs 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

<table>
<thead>
<tr>
<th></th>
<th>RRA</th>
<th>RMA</th>
<th>MDD</th>
<th>RDD</th>
<th>AE</th>
<th>DE</th>
<th>MCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRA</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>RMA</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>MD</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>RD</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>AE</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>DE</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>MCL</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

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.
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](image)

**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.

![Performance varying design time transaction to runtime transaction ratio](image)

**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.