Semantic multigranularity locking and its performance in object-oriented database systems

Kyoung-In Kwon *, Songchun Moon 1

Graduate School of Management, Korea Advanced Institute of Science and Technology, 207-43 Cheongryangni Dongdaemun, Seoul 130-012, South Korea

Received 7 March 1996; revised 30 May 1997; accepted 28 July 1997

Abstract

Concurrency control schemes for object-oriented database systems (OODBSs) used in the area of performance-critical applications should increase the degree of concurrency and decrease locking overhead in order to offer maximum performance. However, most commercially available OODBSs use conventional concurrency control schemes that never exploit the rich semantics of object-oriented data model, thus fail to offer better performance. In this paper, we propose a concurrency control scheme for OODBSs, called in-place semantic multigranularity locking (ISMGL), that not only exploits semantics regarding methods to enhance the concurrency degree, but also utilizes multigranularity locking rules to decrease locking overhead. The novel aspect of ISMGL is that it combines conventional multigranularity locking and nested transaction model with utilizing commutativity of methods. Our concurrency control scheme is applicable for realistic OODBSs in the presence of complex objects with shared subobjects. Lastly, we evaluate the performance of ISMGL by using a simulation study.

Keywords: Concurrency control; Nested transaction; Multigranularity locking; Object-oriented database systems

1. Introduction

Object-oriented database systems (OODBSs) provide capabilities for modeling complex objects and the inheritance among classes. Methods defined in a class, furthermore, usually support rich semantics in comparison to simple read or write
operations in that the methods represent the behavioral aspects of objects. In general, exploiting semantics regarding these methods is considered to be important to alleviate potential data contention. If the execution order of any two methods never affects the semantic consistency of the execution results, they could obviously execute concurrently. In this respect, if we are able to use semantic information regarding methods, the degree of concurrency could be significantly increased so that OODBs are effective enough in the area of performance-critical applications.

Most commercial database management systems (DBMSs) use locking schemes for their concurrency control mechanism due to their simplicity of idea and performance non-inferiority in comparison to any different sort of transaction scheduling schemes so far. In locking schemes, the granule size of lockable unit is considered to have a significant effect on the concurrency degree. By using fine lockable units, in general, the concurrency degree can be enhanced, whilst the overhead of lock management would be accordingly increased due to the increased number of locks to be managed. To reduce the number of locks which must be acquired to access shared objects, multigranularity locking [6, 9] has been developed by organizing the database in the form of lock hierarchies.

Several multigranularity locking schemes [8,11,13] attempted to schedule concurrent accesses of shared objects by utilizing the notion of class and class composition hierarchies. These schemes utilized the inheritance relationship amongst classes and the aggregation relationship in the structure of complex objects. However, they were based only on read and write locks. In other words, these schemes lacked features in exploiting the semantic information regarding methods, thus seem to abandon the possibility of increasing concurrency degree for the performance-critical applications.

To exploit the semantics of a method, the method should be executed in an atomic way such that the method must be executed all or nothing and also executed without interfering against other methods. Otherwise, a concurrent execution of the two commutative methods could leave inconsistent results [17]. Therefore, the atomicity of a method execution must be ensured by a concurrency control scheme. This can be achieved by treating a method execution as a subtransaction of a top-level transaction. Subtransactions ensure only the atomicity and isolation properties among the well-known ACID properties: atomicity, consistency, isolation and durability [10]. This means that a subtransaction may commit or abort independently and appears atomic to other transactions, but the durability of the effects of a committed subtransaction depends on its top-level transaction. Note that a method could invoke other methods so that the calling-called relationship among methods establishes a nested method execution. In this respect, nested method executions can naturally be modeled as nested transactions [15,17].

Greater concurrency degree is possible by reducing the number of conflicts amongst methods, and the conflicts are determined by methods’ semantics. The semantics that is used for the determination of conflicts is the commutativity of methods. Commutative methods can execute concurrently without inducing inconsistent results, thus not having conflict relationships. Therefore, the commutativity relationships between methods must be specified prior to running of transactions to increase the concurrency degree by utilizing semantics associated with methods.

A semantic concurrency control (SCC) scheme [15] attempted to specify commutativity relationships between methods defined on the same class. In this scheme, methods defined on different classes are presumed to be commutative each other. However, this might not be true in OODBs where complex objects with shared subobjects exist and a
subobject could be accessed by methods of more than one object. The reason is that any two methods defined on different classes could access the same shared subobjects simultaneously with conflicting modes. In this case, the commutativity relationships among all methods, even for the methods defined on different classes, should be specified earlier than actual execution of the methods so that a locking scheme can determine the compatibility between methods. Note that specifying commutativity relationships for all methods in all classes in a database is practically impossible. Furthermore, in SCC, a method lock is associated with each method execution for the purpose of resolving conflicts among concurrent method executions on the same object. The employment of method locks would result in increased locking overhead as the number of lock modes increases.

This sort of commutativity presumption problem in the presence of shared subobjects in SCC can be resolved by not associating lock modes with method executions. Instead of controlling conflicts among methods by method locks, we could take an approach to detect the conflicts at execution time on the shared subobjects. That is, we are able to detect method-level conflicts via lower-level conflicts between read and write operations. We call this in-place conflict detection approach in the sense that the conflicts between methods are always detected at the right time instance that the conflicts are posed to take place. In this approach, only atomic operations have their associated lock modes, and the commutativity relationships among methods that access the shared subobjects are left unspecified. The idea is that any conflict between methods that try to access shared subobjects is detected instantly when the methods request a conflicting lock. In this way, incorrect executions are never allowed. Note that method executions are not associated with locks, so methods can continue their executions until a conflict occurs at the level of atomic operations.

2. Related work and problems

Previous work on locking-based concurrency control schemes for OODBs has concentrated on either the issue of exploiting object semantics [15,17] from the viewpoint of increasing the degree of concurrency or the issue of capitalizing on hierarchical structures of objects [11,13] from the viewpoint of reducing locking overhead. The former approach attempts to exploit method semantics based on the nested transaction model [14], and the latter approach extends the conventional multigranularity locking schemes by utilizing class hierarchies and class composition hierarchies.

A representative SCC scheme [15] exploits semantics regarding methods based on the notion of open nested transaction [20,21]. However, SCC cannot be used in OODBs because the major assumption employed in SCC, methods defined on different classes are always compatible, is not applicable to complex objects with shared subobjects. Example 1 illustrates this. Fig. 1 shows a simple project database that is used as an example database in this paper. In this example, the SCC is assumed to use an object as a lockable granule of each method lock.

![Fig. 1. A project database with class composition hierarchy.](image-url)
Example 1 (Commutativity presumption problem associated with method locks in SCC). Suppose that a method \textit{give-bonus( )} defined on a class \textit{project} provides special allowance to all researchers participating in a project, where researchers \textit{m₁} and \textit{m₂} are members of a project \textit{p₁}. A method \textit{salary( )} of a class \textit{researcher} calculates monthly salary of a given researcher by reading the values of two attributes \textit{allowance} and \textit{base-pay}. Fig. 2 shows an execution sequence of two transactions, transaction \textit{total-monthly-salary (T₁)} and transaction \textit{bonus (T₂)}, where \textit{T₁} computes the total monthly salary for all researchers and \textit{T₂} gives special allowance to researchers who participated in project \textit{p₁}. In Fig. 2, \textit{get(m₁.\allowance)} and \textit{set(m₁.\allowance)} represent system-defined atomic operations that read or write the value of \textit{allowance} attribute of \textit{m₁} object, respectively.

In Fig. 2, assume that \textit{T₁} has finished execution of \textit{salary(m₁)}, and is currently executing \textit{salary(m₂)}. At this point, \textit{T₂} is allowed to get a method lock for the execution of \textit{give-bonus(p₁)}. This is because that the actually conflicting methods, \textit{salary( )} and \textit{give-bonus( )}, are defined on different classes so that they are presumed to be compatible. Therefore, the execution sequence of Fig. 2 could be produced by SCC.

However, \textit{salary(m₁)} of \textit{T₁} and \textit{give-bonus(p₁)} of \textit{T₂} are not compatible in the sense that they access the same attribute, \textit{m₁.\allowance}, using incompatible atomic operations, i.e. \textit{get(m₁.\allowance)} and \textit{set(m₁.\allowance)}. This leads to incorrect result in database, that is, the execution sequence of Fig. 2 is not serializable. In this respect, the execution sequence should not take place.

The commutativity presumption problem in SCC illustrated in Example 1 is a severe limitation because modeling complex objects with shared subobjects is a fundamental capability of OODBSs. Semantic locking (SL) [16] resolved this problem by determining the compatibility of methods dynamically whenever any two methods access the same subobject at execution time. In order for this to be possible, SL associates a shared (S) lock with a read operation and an X lock with a write operation, but never associates a lock with a method execution. The commutativity relationships between methods, that are specified for each class, are only used to alleviate a conflict when any two method executions access a shared object with incompatible lock modes.

Note that both SCC and SL use only one lock granule type of object or page. This could lead to severe locking overhead when a transaction accesses large amount of database. For instance, if a transaction accesses a huge amount of objects, say hundred thousand, belonging to the same class where an object-level locking is used, the transaction must acquire hundred thousand locks one
by one. Moreover, introducing new locks for all methods in this situation would be another source of locking overhead.

An object-oriented multigranularity locking (OMGL) protocol [8,13] has been proposed for an OODBS where the class hierarchy and class composition hierarchy are taken into account on forming lockable granule hierarchy (LGH). OMGL, however, has a restriction of admitting exclusive accesses on a class composition hierarchy in that there cannot be any direct writers regarding the instances of component classes even if there is only a single reader throughout the class composition hierarchy. Example 2 illustrates this concurrency degradation phenomenon in OMGL. Before starting Example 2, we give a short description of OMGL to make this example comprehensible.

OMGL locks a node of LGH with one of the five lock modes: intention shared (IS), intention exclusive (IX), shared (S), shared and intention exclusive (SIX), and X. For class composition hierarchies, OMGL uses three different lock modes: intention shared object (ISO), intention exclusive object (IXO), shared and intention exclusive object (SIXO). Note that ISO mode conflicts with IX mode, and IXO mode conflicts with both IS and IX modes. Similarly SIXO mode also conflict with both IS and IX modes. To lock a complex object, first of all OMGL locks the root class in IS, IX, S, SIX, or X mode. It, then, locks each component class of the complex object in ISO, IXO, S, SIXO, or X.

**Example 2** (Concurrency degradation in OMGL due to class composition hierarchy locking). Reconsider Fig. 1. Suppose that $T_3$ selects all *project* objects such that the progress-state is finished and $T_4$ increases the salary of a *researcher* object by executing a method *inc-base-pay( )*. According to the locking rules of OMGL, both $T_3$ and $T_4$ try to lock appropriate classes and objects in a root-to-leaf order as follows: $T_3$, lock root class *project* in IS mode; lock *researcher* class, a component class of *project*, in ISO mode; lock the selected *project* objects in S mode. $T_4$, lock *researcher* class in IX mode; lock a *researcher* object in X mode.

Here, $T_3$'s ISO lock on *researcher* class and $T_4$'s IX lock on the same class cannot be granted simultaneously, since the two lock modes are incompatible. For this reason, the two transactions cannot proceed concurrently. Therefore, if $T_3$ locks the *researcher* class in ISO mode first, $T_4$ must wait until $T_3$ has finished. $T_3$ and $T_4$, however, in fact can be allowed to execute concurrently because they never access a common object. That is, any interleaved execution sequence of $T_3$ and $T_4$ is serializable. In this respect, OMGL has a concurrency degradation with regard to locking complex objects.

This deficiency in OMGL arises from the fact that, whenever a complex object is locked, every component object of that object in the composition hierarchy is also locked even though a transaction accesses a small portion of component objects. In addition, if a class composition hierarchy has a deep complex structure, the locking overhead would be increased, since the number of class objects that must be locked becomes large. Furthermore, OMGL is based solely on the notion of read and write locks and a flat transaction model so that the protocol itself is insufficient to exploit semantics regarding methods.

In this respect, it is desirable to improve the OMGL by adopting a SL scheme based on the notion of in-place conflict detection approach. We propose a semantic multigranularity locking scheme for OODBSs called *in-place semantic multigranularity locking* (ISMGL) in the next section.

### 3. The model and lockable granule hierarchy

The model of an object-oriented database and nested transaction are described in this section. We also describe the LGH for our locking scheme in detail.
3.1. Database model and assumptions

Since there is no standard object-oriented data model that is unanimously accepted, we use a commonly accepted object-oriented data model [3,7,13, 19]. We assume that an object can participate in more than one complex object; in this case the object is said to be a shared subobject. This assumption is appropriate in the sense that an object in real world might have arbitrary many relationships with other objects so that it could be referenced by a number of other objects.

We assume that get and set are the system-defined methods which are used to directly read or write a given attribute, respectively. In addition, we assume that the get and set operations are atomic. Methods of an object are generally allowed to invoke methods of other objects. Moreover, query facilities and methods of complex objects are allowed to access the attributes of component objects directly by invoking system-defined methods [15,17]. Introducing these system-defined methods might seem to be a violation to the notion of encapsulated objects. Some of the previous work [15,17] allow this, and others [8,19] do not. The reasons for allowing this direct access via system-defined methods, in this paper, are twofold: First, respecting the notion of encapsulation fully is impractical from the viewpoint of efficiency in that most queries require the structure of objects to be revealed and it is hard to define all possible methods for objects [15]. Second, the possibility of accidental corruption of objects is considered to be very low in the sense that methods are defined by qualified database administrators and the query facilities are verified component of database systems.

We do not assume that any two methods defined on different classes are compatible. This is because they could be in conflict with each other in the presence of shared subobjects and the conflict relationships would be determined dynamically. Here, the compatible methods mean that they could execute concurrently without damaging the database consistency. Therefore, the compatibility relationship between methods defined on different classes is assumed to be unknown prior to their execution. Analogously, the compatibility relationships between methods defined on the same class are also assumed to be unknown if they could access other objects. Except the unknown compatibility cases, it is generally possible to specify compatibility relationships among methods defined on the same class. For each class, we assume that a commutativity table that shows compatibility relationships between methods exists as a special property of the class.

3.2. Transaction model

A method defined on a class could access both a local state of an object and a local state of other component objects by invoking methods defined on the component classes. This establishes a method invocation hierarchy and results in a nested method execution, and it can naturally be modeled as nested transactions [17]. Formally, a transaction that allows nesting can be defined as follows.

**Definition 1.** A transaction is a tuple $T_i = (o_j \cup t_j, < T_i)$, where $o_j$ is a set of atomic operations and $t_j$ is a set of other transactions, and $< T_i$ is a partial order on $o_j \cup t_j$, which at least orders all conflicting atomic operations and transactions in $o_j \cup t_j$.

Since we concern only the problem of concurrency control, the abortion of transaction is not depicted in the definition. In the definition of transaction, every transaction in $t_j$ should appear to be atomic to $T_i$. The members of $o_j \cup t_j$ are children of $T_i$, and $T_i$ is their parent. If a transaction does not have its parent, it is called top-level transaction, otherwise it is called subtransaction. A transaction may contain any number of
subtransactions, which again may hold any number of subtransactions. The term method execution and subtransaction will be used in an interchangeable manner hereafter in this paper, since a method execution is modeled as a subtransaction. The ancestor (descendant) relation is the reflexive transitive closure of the parent (child) relation. The least common ancestor of a set of transactions \( T \) is a transaction \( T_i \) such that \( T_i \) is an ancestor of every transaction \( T_j \in T \) but no proper descendant of \( T_i \) meets this property. \( T_i \) and \( T_j \) are incomparable if neither is a descendant of the other.

The notion of transaction makes use of the conflict relationships between atomic operations and subtransactions. Non-conflicting operations are said to be commutative. We need to define the commutativity between methods for the purpose of exploiting semantics of methods. Two method executions, say \( t \) and \( t' \), on the same object are said to be commutative if and only if the two possible execution sequences, i.e. \( \langle t, t' \rangle \) and \( \langle t', t \rangle \), are indistinguishable for both \( t \) and \( t' \) and at the same time indistinguishable for all possible sequences of methods that may be invoked subsequently [15].

On the basis of this definition of commutativity, database class designers are able to define a commutativity table for each class. For instance, the commutativity table for methods defined on a class researcher is depicted in Table 1.

<table>
<thead>
<tr>
<th>Executed</th>
<th>Requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpt-special-area ( )</td>
<td>Yes</td>
</tr>
<tr>
<td>inc-allowance ( )</td>
<td>Yes</td>
</tr>
<tr>
<td>inc-base-pay ( )</td>
<td>Yes</td>
</tr>
<tr>
<td>salary ( )</td>
<td>Yes</td>
</tr>
</tbody>
</table>

3.3. Lockable granule hierarchy

MGL allows a transaction to acquire locks on appropriate size of data granules for its operations. MGL, however, must prevent any two concurrent transactions from acquiring conflicting locks on two data granules that are non-disjoint. This can be achieved by organizing a database as LGH by exploiting the natural form of hierarchical relationships between different data granules in the database. Each node of the LGH represents a lockable data item containing all its descendants.

A typical LGH for MGL in relational DBMSs consists of database, relation, and record, where the database is the most coarser granule and the record is the finest granule. This LGH corresponds to the notion of database, class, and object in OODBSs, from top to down. However, the class hierarchy must be taken into account in the MGL for OODBSs in the sense that it has a crucial effect on the data granules that should be locked due to the following reasons. First, a class inherits attributes and methods defined in its superclasses. Hence, while a transaction is accessing an instance of a class, the definition of the class and its superclasses must not be modified by another transaction. Second, whenever a MGL scheme tries to lock a class node in a LGH, it must consider the class hierarchy associated with the class because

| Table 1
<p>| Commutativity table for methods of class researcher |
|----------------|------------|</p>
<table>
<thead>
<tr>
<th>Executed</th>
<th>Requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpt-special-area ( )</td>
<td>Yes</td>
</tr>
<tr>
<td>inc-allowance ( )</td>
<td>Yes</td>
</tr>
<tr>
<td>inc-base-pay ( )</td>
<td>Yes</td>
</tr>
<tr>
<td>salary ( )</td>
<td>Yes</td>
</tr>
<tr>
<td>Held</td>
<td>Requested</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
</tr>
<tr>
<td>IS</td>
<td>IX</td>
</tr>
<tr>
<td>S</td>
<td>SIX</td>
</tr>
<tr>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IS</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>IX</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>S</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SIX</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>X</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

We conclude this section by establishing the correctness of our concurrency control scheme.

4.1. Notion of in-place conflict detection

The notion of in-place conflict detection is employed to solve the commutativity presumption problem associated with method locks in the presence of shared subobjects mentioned in the introduction. The primary goal is to detect a conflict between methods which might access shared subobjects without explicitly specifying commutativity relationships among all the methods defined on all classes. To achieve this goal, we associate S and X locks only with read and write operations, respectively. Note that the read and write operations on local attributes are appeared as the system-defined get and set methods from the viewpoint of application programmers. Methods, however, do not possess their lock modes. In this approach, the commutativity relationships among methods, that are not specified explicitly, are determined in-place at the time when those methods request conflicting locks on the shared subobject simultaneously. Accordingly, the commutativity presumption problem is resolved since all the conflicts among methods can be discovered and controlled correctly at execution time.

This in-place conflict detection approach is conceivable only when we lock component objects of a query on a class usually requires an evaluation against both that class and all its subclasses. Therefore, we lock both a class and all its descendant classes on the class hierarchy whenever a database schema change operation or a query operation involving a class and its descendants is executed. Note that the LGH is assumed to be a tree for the simplicity of our locking protocol.

We do not consider a complex object including all its component objects a lockable granule. Instead, we lock the component objects of complex objects separately, following the actual access patterns of database operations, in order for the in-place conflict detection approach be achievable, as elucidated in the next section. By taking this way, concurrency degradation in OMGL shown in Example 2 would also be alleviated. The compatibility relationships among lock modes in MGL scheme [9] are presented in Table 2 for the completeness of our locking protocol.

4. Semantic multigranularity locking

In this section, we propose a concurrency control protocol that synchronizes the execution of nested transactions with exploiting method commutativity and a multigranularity locking scheme. Prior to presenting the protocol, we discuss the notion of in-place conflict detection approach.
complex object individually. That is, a complex object including all its component objects should not be locked altogether. Otherwise, any complex objects that share a single component object cannot be accessed concurrently by different methods in conflicting modes. In this case, conflicts between methods cannot be detected at the time instance that those conflicts take place, since the methods never have a chance to access the shared component object in conflicting modes. Therefore, individual locking on component objects is indispensable to the in-place conflict detection approach. Note that this approach takes an optimistic way in deciding conflicts between methods. That is, it never presumes that the conflicts would exist among them. Rather it decides whether a method-level conflict takes place only when the actual low-level conflict occurred.

4.2. Locking protocol

The concepts of nested transaction model and multigranularity locking are orthogonal from the viewpoint of the locking protocol. The former is dealt with a transaction structure, and the latter is about the data granule for locking. In this respect, we have to combine the locking rules for nested transaction and multigranularity. We first present a locking protocol for nested transaction, then combine it with multigranularity locking rules.

To exploit the commutativity of methods, we make use of the SL [16]. However, it turned out that SL could allow non-serializable executions whenever a retained S lock is converted into a retained X lock [12], and this problem must be solved. We now present a SL protocol called enhanced semantic locking (ESL) that prohibits the potential non-serializable executions of SL. Note that the notion of multigranularity locking is not considered in ESL. This ESL protocol will be a basis of our semantic multigranularity locking protocol, ISMGL. The ESL rules are described in the following. In the following rules, if a lock is held, then the holding transaction or subtransaction has a right to do a corresponding operation, while a retained lock is a placeholder indicating that some of committed subtransactions of the retainer had held the lock and the top-level transaction of the retainer is not terminated yet.

4.2.1. ESL rules

(1) A method execution \( m \) can execute read (or write) operation on a data granule \( x \), iff \( S_m[x] \) lock (or \( X_m[x] \) lock) is requested and granted so that it is held by the method.

(2) A method execution \( m \) cannot commit or abort until all its children have terminated. When \( m \) commits or aborts:
   (2.1.) \( m \) is not at top-level and it commits: its locks are inherited and retained by its parent. However, lock conversion for the inherited locks does not occur in its parent.
   (2.2.) \( m \) is not at top-level and it aborts: its locks are discarded.
   (2.3.) \( m \) is at top-level: its locks are discarded.

(3) A \( S_m[x] \) lock (\( X_m[x] \) lock) is granted to \( m \) iff
   (3.1.) no other method execution holds a \( X[x] \) (\( S[x] \) or \( X[x] \) lock), and
   (3.2.) all method executions, \( m'' \), that retain \( X[x] \) locks (\( S[x] \) or \( X[x] \) locks) are such that
         (a) \( m'' \) is an ancestor of \( m \), or
         (b) \( m'' \) is not an ancestor of \( m \), and a method, \( m' \), among the lock inherited path for each conflicting \( X[x] \) lock (\( S[x] \) or \( X[x] \) lock) but not including the retainer is commutative with an ancestor of \( m \).

The reason for not including the retainer in Rule (3.2) is to guarantee the atomicity of method execution that owns the retained lock. There are two differences between our ESL and the SL as
follows. First, since ESL does not allow the lock conversion for retained locks, it has the same or more number of retained locks compared to the retained locks which SL might have. Therefore, ESL is more restrictive than SL with respect to granting locks, thus allows less execution sequence of transactions. Second, the corresponding rule of SL to Rule (3.2) of ESL disallows lock granting in the cases that the ancestor of \( m \) is a directly or indirectly commuting ancestor has \( m' \). This kind of directly or indirectly commuting ancestor has never happened in our model, since we define the commutativity relationships only for the methods that access only local state of objects. Therefore, ESL protocol accepts the same class of execution sequences as SL accepts in our model if SL is modified with disallowing lock conversions for retained locks.

Since the MGL and ESL are orthogonal, we must enforce every transaction satisfies the rules for granularity locking and the rules for nested transactions independently. We now propose a locking protocol called ISMGL that integrates both the locking rules.

### 4.2.2. ISMGL rules

1. If \( x \) is not the root of the LGH, then to set \( S_i[x] \) or \( IS_i[x] \) (or \( X_i[x] \) or \( IX_i[x] \)), \( T_i \) must have an IS or IX lock (or IX lock) on \( x \)'s parent.

2. To execute read (or write) operation on a data granule \( x \), \( T_i \) must hold an S or X (or X) lock on some ancestor of \( x \) in the LGH.

3. \( T_i \) cannot commit or abort until all its children have terminated. When \( T_i \) commits or aborts:
   - (3.1) \( T_i \) is at top-level: its locks are inherited and retained by its parent. However, lock conversion for the retained locks does not occurred in its parent.
   - (3.2) \( T_i \) is not at top-level and it aborts: its locks are released.
   - (3.3) \( T_i \) is at top-level: its locks are released.
   - (4) A lock is granted to \( T_i \) if and only if
     - (4.1) no other transaction holds a conflicting lock, and
     - (4.2) all transactions, \( T_j \), that retain conflicting locks are such that
       - (a) \( T_j \) is an ancestor of \( T_i \), or
       - (b) \( T_j \) is not an ancestor of \( T_i \), and a transaction within the lock inherited path for each conflicting lock but not including the retainer is commutative with an ancestor of \( T_i \).
   - (5) \( T_i \) may not release an intention lock on a node \( x \), if it is currently holding a lock on any child of \( x \).

Rules (1), (2), and (5) are derived from conventional MGL protocol, and Rules (2)-(4) are derived from ESL. According to ISMGL, a transaction \( T_i \) must request its lock from the root-to-leaf order in the LGH following Rules (1) and (2), and release its lock from the leaf-to-root order by Rule (5). Whether or not the requested lock is granted is determined by Rule (4). Note that Rule (3) guarantees that ISMGL obeys the strict two-phase protocol, thus free from the cascading abort problem.

We now present two examples that show the operations and benefits of ISMGL. Example 3 explains the commutativity presumption problem of SCC illustrated in Example 2 does not occur in ISMGL, and Example 4 shows the efficient locking on a complex object in ISMGL compared to OMGL when a transaction accesses a small portion of the component objects. Note that only class-level granules and object-level granules are considered to be lockable granules in ISMGL of the following examples.

**Example 3 (Prevention of commutativity presumption problem in ISMGL).** In this example we use the same transactions used in Example 1 where the execution sequence of the two transactions is
depicted in Fig. 2. In Fig. 2, we again assume that
T₁ has finished the execution of salary(m₁), and is
currently executing salary(m₂). At this point, in
accordance with Rule (3.1), the top-level transaction
of T₁ retained salary method's S lock on m₁, de-
noted as Ssalary[m₁], and IS lock on the class research-
er, denoted as ISsalary[researcher], to which
the researcher object m₁ belongs. When the give-
bonus(p₁) of T₂ is requesting X lock on m₁ in
order to execute set(m₁, allowance) after acquiring
IXgiben-bonus[project], a conflict with the Ssalary[m₁]
of T₁ is detected.

Here the ISMGL concurrency controller refers
the commutativity table of the class project in ac-
cordance with Rule (4.2) to see if give-bonus(p₁)
is commutative with salary(m₁). However, since
the two methods are not commutative, the execu-
tion of give-bonus(p₁) is blocked until the conflict-
ing Ssalary[m₁] lock is released. Consequently, the
execution sequence of Example 2 could not be pro-
duced by ISMGL, so that the commutativity pre-
sumption problem of SCC is prevented from
occurring.

Example 4 (Efficient locking on complex objects in
ISMGL). Reconsider the project database depicted
in Fig. 1 and transactions T₃ and T₄ explained in
Example 2. In accordance with the locking rules
of ISMGL which do not lock a complex object as a whole, both T₃ and T₄ try to lock appropriate
classes and objects in a root-to-leaf order as fol-
lows: T₃, lock root class project in IS mode; lock
the selected project objects in S mode. T₄, lock re-
searcher class in IX mode; lock a researcher object
in X mode.

Unlike the previous OMGL example, the two
transactions can proceed concurrently, since
there is no lock conflict between them. In this
respect, the concurrency degradation problem of
OMGL depicted in Example 2 does not occur
in ISMGL so that some concurrency gain is ex-
pected.

4.3. Proof of correctness

In this section, we prove the correctness of IS-
MGL in that it allows only semantically serializ-
able execution. The correctness is proved in step
by step manner. We first discuss the correctness
of ESL based on the proof of SL presented in
[16]. Then, we argue that the ISMGL is equivalent
to ESL from the viewpoint of explicit locks setting
on the leaves of LGH. These are sufficient to show
the correctness of ISMGL.

We use the notion of semantically serializable
execution [5] for the correctness proof of ISMGL.
An execution E is semantically equivalent to a serial
execution Eₛ if we can get Eₛ from E using sub-
stitution and commutativity-based reversal trans-
fomations. An execution is semantically serializable
if it is semantically equivalent to some serial exe-
cution.

In order to prove that only semantically serial-
able executions are allowed in SL, Resende [16]
makes use of the bottom-up proof paradigm [5]
which uses the notion of commutativity-based re-
versal and the notion of substitution. For employ-
ing the bottom-up proof paradigm, he develops
separate leftmost and reduce (SLR) algorithm.
SLR separates leftmost node that has only leaves
as children by using commutativity-based reversals
and substitutes it with an atomic subtransaction,
and it repeats this procedure until only top-level
transactions are left.

We now discuss the correctness of ESL based
on the correctness proof of SL. Note that ESL is
essentially the same locking protocol as SL in
our model if SL prohibits lock conversion for re-
tained locks.

Theorem 1. ESL allows only semantically serializ-
able executions.

Proof. Immediate from the fact that SL allows
only serializable executions [16] and ESL adopts
the same locking rules as SL except the prohibition of retained lock conversion. □

We now prove that ISMGL is equivalent to ESL which uses only two lock modes, and which explicitly locks the leaves of LGH. Let \( L = (N, E) \) be a lock instance tree where \( N \) is the set of nodes and \( E \) is the set of edges. Each node is a lockable data granule that contains all its child nodes, and an edge represents this containment relationship. Let \( Q \) be the set of leaves of \( L \). To show that ISMGL is equivalent to ESL from the viewpoint of leaf-level locking, we should ensure that transactions never granted to hold conflicting explicit or implicit locks on the same leaf-level data item, excepting the semantics of transactions involved in the specific lock granting process neglects the conflicts.

**Lemma 1 (Prevention of conflicting lock holders on leaf nodes).** Suppose all transactions observe the ISMGL protocol with respect to a given lock instance tree \( L \). If a transaction holds an explicit or implicit lock on a node \( n \in Q \) of \( L \), then no other transaction holds a conflicting explicit or implicit lock on that node.

**Proof.** We prove this lemma by using the same method in [6]. Suppose that transactions \( T_i \) and \( T_j \) hold conflicting locks on a leaf node \( n \). There are five possible cases.

1. \( T_i \) holds implicit \( S \) lock, \( T_j \) holds explicit \( X \) lock.
2. \( T_i \) holds implicit \( S \) lock, \( T_j \) holds implicit \( X \) lock.
3. \( T_i \) holds explicit \( S \) lock, \( T_j \) holds implicit \( X \) lock.
4. \( T_i \) holds explicit \( X \) lock, \( T_j \) holds explicit \( X \) lock.
5. \( T_i \) holds implicit \( X \) lock, \( T_j \) holds implicit \( X \) lock.

The case that \( T_i \) holds explicit \( S \) lock and \( T_j \) holds explicit \( X \) lock is excluded from this discussion since it is obviously impossible. In addition, the case that \( T_i \) holds explicit \( X \) lock and \( T_j \) holds explicit \( X \) lock is also disregarded for the same reason.

**Case 1.** By Rule (2) of ISMGL, \( T_i \) holds \( S_i[n'] \) for some proper ancestor \( n' \) of \( n \). By Rule (1), \( T_j \) must hold an \( IX \) lock on every proper ancestor of \( n \). Thus, \( T_j \) must hold \( IX_j[n'] \). This, however, is impossible because the \( S \) mode lock and \( IX \) mode lock are in conflict.

**Case 2.** By Rule (2) of ISMGL, \( T_i \) holds \( S_i[n'] \) for some proper ancestor \( n' \) of \( n \), and \( T_j \) holds \( X_j[n''] \) for some proper ancestor \( n'' \) of \( n \). There are three subcases: (a) \( n' = n'' \), (b) \( n' \) is a proper ancestor of \( n'' \), and (c) \( n'' \) is a proper ancestor of \( n' \).

Case (a) is impossible since \( T_i \) and \( T_j \) cannot hold conflicting locks on the same node. Case (b) is impossible because \( T_j \) must hold \( IX_j[n''] \), which conflicts with \( S_i[n'] \). Case (c) is, also, impossible since \( T_i \) must hold \( IS_i[n''] \), which conflicts with \( X_j[n''] \). Therefore, Case 2 is impossible.

Cases 3 and 4 follow the same discussion as Case 1, and Case 5 follows the discussion made for Case 2.

**Lemma 2 (Prevention of conflicting lock retainers on leaf nodes).** Suppose all transactions observe the ISMGL protocol with respect to a given lock instance tree \( L \). If a transaction \( T_i \) retains an explicit or implicit lock on node \( n \in Q \) of \( L \), then no other transaction holds a conflicting explicit or implicit lock on that node except the retainer that is commutative with an ancestor of \( T_j \).

**Proof.** Assume that \( T_i \) retains a lock, and \( T_j \) is not a proper descendant of \( T_i \) and there is no transaction within the lock inherited path for the retained lock except \( T_i \) that is commutative with an ancestor of \( T_j \). Suppose that transaction \( T_j \) is granted to hold a conflicting lock on a leaf node \( n \). There are eight cases:
1. $T_i$ retains implicit S lock, $T_j$ holds explicit X lock.
2. $T_i$ retains implicit S lock, $T_j$ holds implicit X lock.
3. $T_i$ retains explicit S lock, $T_j$ holds explicit X lock.
4. $T_i$ retains explicit S lock, $T_j$ holds implicit X lock.
5. $T_i$ retains implicit X lock, $T_j$ holds explicit X lock.
6. $T_i$ retains implicit X lock, $T_j$ holds implicit X lock.
7. $T_i$ retains explicit X lock, $T_j$ holds explicit X lock.
8. $T_i$ retains explicit X lock, $T_j$ holds implicit X lock.

**Case 1.** By Rule (2) of ISMGL, $T_i$ retains $S_i[n']$ for some proper ancestor $n'$ of $n$. By Rule (1), $T_j$ must hold an IX lock on every proper ancestor of $n$. Thus, $T_j$ must hold $IX_j[n']$. This, however, is impossible since the retained $S_i[n']$ lock and $IX_j[n']$ conflict each other and, by assumption, $T_j$ is not a proper descendant of $T_i$ and there is no transaction within the lock inherited path for the retained lock except $T_i$ that is commutative with an ancestor of $T_j$. Therefore, Case 3 is impossible.

Cases 4, 5, and 8 follow the same discussion as Case 1. Case 6 follows the same discussion as Case 2. Case 7 follows the same discussion as Case 3.

**Theorem 2** (Leaf-level locking equivalence between ISMGL and ESL). *If the explicit locks retained or held by two incomparable transactions do not conflict, then the implicit or explicit locks retained or held on leaf nodes do not conflict.*

**Proof.** By Rules (1) and (2), implicit or explicit lock modes on leaf nodes are X or S modes that are the same lock modes used in ESL. In addition, by Lemmas 1 and 2, it is obvious that the implicit or explicit locks retained or held on leaf nodes by two incomparable transactions do not conflict. 

5. **Performance evaluation**

In this section, we evaluate the performance characteristics of the protocol which exploits method semantics and multigranularity locking technique by means of an actual simulation. The simulation is based on the implementation of nested two-phase locking (N2PL) [14], ESL, and ISMGL with CSIM discrete-event simulation package [18]. Note that SL is not considered here because it is almost the same protocol with ESL as mentioned before.

5.1. **Simulation model**

In order to evaluate the performance of the protocol, we have constructed a database simulation model similar to the one described in [1,4]. The simulation model, shown in Fig. 3, is basically a closed queuing model which consists of terminals, transaction manager (TM), concurrency controller (CC), and data manager (DM).
Terminals have roles to generate nested transactions one by one. Before generating a new nested transaction, it waits for the \textit{inter\_arrival\_delay} time which is assumed to be exponentially distributed. Each nested transaction has arbitrary number of subtransactions and a subtransaction consists of actual read and write operations. To simplify our simulation study, all nested transactions are assumed to be three levels in the sense that each of them has one top-level transaction, a number of subtransactions, and actual read or write operations on objects at the leaf-level. In addition, we assume that the top-level transactions never have actual data access operations.

To study the effect of multigranularity locking, terminals generate two types of nested transactions: coarse granularity transaction (CGT) and fine granularity transaction (FGT). CGT has a subtransaction that accesses predefined percentage of objects that are instances of a specific class in random manner so that a coarse granule class lock can be utilized. Note that CGT type transactions never exploit method semantics since they set class-level locks explicitly. On the other hand, FGT has subtransactions that access arbitrary objects that are uniformly distributed in the database, and it exploits method semantics as the methods are modeled as subtransactions. When a subtransaction aborts, the top-level transaction tries to restart the subtransaction. Note that the subtransactions of a nested transaction executed sequentially so that the parallelism among subtransactions is not allowed in our simulation. By disallowing the sibling parallelism, we could see the effect of method semantics exploitation more clearly since the effect of sibling parallelism is excluded.

TM coordinates transactions' operations between terminals and CC by placing necessary lock requests for the operations to \textit{cc\_queue} on behalf of the transactions. Note that class- and object-level lock granules are utilized in the simulation of ISMGL, and only object-level locks are applied to N2PL and ESL. The CC processes the lock requests in \textit{cc\_queue} according to its concurrency control protocol. If a lock request is granted, the CC forwards the corresponding operation to DM or sends lock grant message to TM. Otherwise, the transaction that requests the lock is blocked by CC until the CC wakes it up when some other transactions release or inherit the conflicting locks. Deadlocks are resolved by time-out mechanism where the time-out value is specified by the parameter \textit{time\_out}. DM consists of a variable number of disks, \textit{num\_disks}. In our simulation model, TM and CC utilize CPU servers, and DM utilizes disk servers. Note that the requests to CPU and disk resources are serviced on first-come, first-served (FCFS) manner.

5.2. Parameter settings and performance metrics

Table 3 lists the simulation parameters and their brief descriptions. The values chosen to conduct the experiments are also listed in Table 3. The number of classes, \textit{num\_classes}, and objects,
Table 3  
Simulation model parameter descriptions and values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_size</td>
<td>Number of objects in database</td>
<td>1000 Data objects</td>
</tr>
<tr>
<td>num_classes</td>
<td>Number of classes</td>
<td>20 Classes</td>
</tr>
<tr>
<td>num_cpus</td>
<td>Number of CPUs</td>
<td>2 CPUs</td>
</tr>
<tr>
<td>num_disks</td>
<td>Number of disks</td>
<td>4 Disks</td>
</tr>
<tr>
<td>num_terms</td>
<td>Number of terminals</td>
<td>10-100 Terminals in steps of 10</td>
</tr>
<tr>
<td>int_arrival_delay</td>
<td>Transaction inter-arrival delay</td>
<td>5 s</td>
</tr>
<tr>
<td>cpu_delay</td>
<td>CPU time for accessing an object</td>
<td>12 ms</td>
</tr>
<tr>
<td>io_delay</td>
<td>I/O time for accessing an object</td>
<td>35 ms</td>
</tr>
<tr>
<td>cc_req_delay</td>
<td>CPU time for servicing a cc request</td>
<td>3 ms</td>
</tr>
<tr>
<td>cgt_pct</td>
<td>Percentage of CGT</td>
<td>0-75% in steps of 25</td>
</tr>
<tr>
<td>nest_tran_size</td>
<td>Mean nested transaction size</td>
<td>16 Subtransactions (24 operations)</td>
</tr>
<tr>
<td>update_method_pct</td>
<td>Update method percentage</td>
<td>50%</td>
</tr>
<tr>
<td>commute_pct</td>
<td>The percentage of commutative methods</td>
<td>45%</td>
</tr>
<tr>
<td>time_out</td>
<td>Time-out value for deadlock detection</td>
<td>1.5 s</td>
</tr>
</tbody>
</table>

$db_size$, in database are set to 20 and 1000, respectively. These values are chosen to induce the necessary amount of data contention so that we could observe the relative performance behavior of the protocols. In our experiments, we varied the number of terminals, $num_terms$, from 10 to 100 in steps of 10. This $num_terms$ parameter set on an experiment determines the multiprogramming level (MPL) for the experiment.

In order to evaluate the effect of multigranularity locking, we varied the percentage of CGT, $cgt_pct$, from 0 to 75 in steps of 25. The nested transaction size, $nest_tran_size$, which is only applicable to FGT, represents the average number of subtransactions performed by a nested transaction, the mean of a uniformly distributed random variable between $nest_tran_size \pm 3$. Each subtransaction represents either simple read or simple update method where a method accesses only one object. The percentage of simple update method, $update_method_pct$, which accesses an object with read and write operations sequentially, is set to 50%. The approximate sizes of a CGT and FGT type nested transactions are 45 and 24 operations, respectively. The percentage of commutative methods, $commute_pct$, is fixed at 45%. This parameter shows the percentage of read/write operation conflicts that could be ignored due to method semantics. Observe that only FGT in the protocols which exploit method semantics makes use of this parameter.

The CPU access delay, $cpu_delay$, is set to 12 ms, the I/O access delay, $io_delay$, is set to 35 ms, and the concurrency control service delay, $cc_req_delay$, is set to 3 ms. These parameter values are similar to those in previous performance studies of locking protocols [1,2]. In addition, our simulation experiments are performed under limited resource environments in which two CPU resources and four disk resources are utilized.

The transaction throughput rate and response time are used as main performance metrics in our simulation. The throughput rate is measured by the number of committed transactions per second, and the response time in seconds is measured as the difference between a transaction's initiation time and commit time. Since our simulation model is a closed queuing system, the response time can
be calculated from throughputs and MPLs by applying Little's law [22]. Hence, we do not explicitly show the response times. In addition to the two main performance metrics, the class- and object-level blocking ratios, which give average number of times that a transaction has to block to commit, are used to analyze the effect of utilizing multigranularity locking rules and method semantics in concurrency control schemes. The class- or object-level blocking ratio is computed as the ratio of the number of transaction blocking events with waiting for class- or object-level locks to the number of transaction commits, respectively.

5.3. Results and analysis

We now present and analyze the results of the simulation experiments conducted for the protocols: N2PL, ESL, and ISMGL. The graphs in Figs. 4–8 show the average results of two runs with a run time of 10,000 s. In order to eliminate the initial transient effect, we discarded the execution history of the first 5% of the total run time. In the experiment, we studied the throughput characteristics of the three protocols by varying the ratio of CGT and FGT with the effects of MPL. From the experiment of CGT 0%, we could see the effect
of method semantics exploitation by comparing throughputs between ESL and N2PL, since the transactions only consist of FGTs in this experiment. In addition, from the experiments of CGT 25%, 50% and 75%, we could see the effect of multigranularity locking in throughput characteristics.

Fig. 4 illustrates that the maximum throughput is obtained with ESL and the thrashing point of ESL is higher than those of ISMGL and N2PL. In this experiment, the throughput of ISMGL is considerably lower than those of the other protocols. This is to be expected as ISMGL just aggravates locking overhead via introducing additional intention mode locks on classes while having no chances to reduce locking overhead by using explicit class-level locks. Observe that ESL outperforms N2PL, and this is because the advantage of exploiting commutativity between methods is only taken in ESL.

Figs. 5–7 show the effect of CGT percentage increase in throughput characteristics of the three protocols. Observe that ISMGL shows better throughputs as the CGT percentage increases, and it outperforms ESL beginning from the CGT 50% experiment. This is because that ISMGL is able to profit more from the use of explicit class-level locks that could reduce the need of object-level locks substantially as the CGT percentage increases.

The interesting fact is that the peak throughputs of ISMGL occur at MPL = 40 and around, which is quite smaller compared to ESL that shows the highest throughputs at MPL = 50 or MPL = 60. The primary reason that attributes to this result is the difference of class- and object-level blocking ratios among the protocols. To show this, we illustrate the blocking ratios of CGT 75% experiment in Fig. 8 as a representative case. Note that only object-level blocking ratios are applicable to ESL and N2PL, since they only utilize the object-level locking. For ISMGL, Fig. 8 shows that the class-level blocking is dominant source of transaction blocking, and the blocking ratio is increased rapidly beginning at MPL = 60. This results in the rapid throughput degradation of ISMGL when the MPL goes beyond that point.
Consequently, the above simulation results verify the fact that the ESL which exploits method semantics outperforms N2PL. Thus a more sophisticated semantics-based locking protocol deserves to be applied in high contention environments. In addition, ISMGL outperforms ESL and N2PL if the percentage of CGT is higher than a certain point. Thus the multigranularity locking approach to semantics-based concurrency control is justifiable in OODBSs.

6. Conclusion

In this paper, we have presented a semantic multigranularity locking scheme for OODBSs, called ISMGL, and its brief performance evaluation. The beneficial properties of ISMGL are that (1) it exploits the commutativity of methods to enhance the degree of concurrency, (2) it decreases locking overhead by utilizing multigranularity locking rules. We resolve the commutativity presumption problem by taking the in-place conflict detection approach. This approach, also, alleviates the concurrency degradation of OMGL in some degree.

ISMGL is novel since it combines conventional multigranularity locking scheme and nested transaction model with exploiting commutativity of methods, and is applicable for performance-critical OODBSs in the presence of complex objects with shared subobjects. In addition, as our performance evaluation indicates, the integration of a multigranularity locking and a semantics-based concurrency control can produce better performances in OODBSs.

Acknowledgements

The authors wish to acknowledge the anonymous referees for their helpful comments.

References

Kyoung-In Kwon received his B.S. degree in Computer Science and Statistics from Seoul National University in 1987 and his M.S. degree in Information and Communication Engineering from Korea Advanced Institute of Science and Technology (KAIST) in 1993. Currently, he is working towards his Ph.D. at the KAIST in Seoul. His research interests include database concurrency control, object-oriented database systems, and data models. He has also been working for LG Information & Communications Ltd. as a senior software engineer in developing switching systems since 1987.

Songchun Moon received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 1985. He has been working for KAIST since then. He was a distinguished scholar at the Hungarian Academy of Science and still serves for EURO-MICRO as a director. He has developed a multi-user relational database management system IM which is the first prototype ever in Korea in 1990, and also a distributed database management system DIME in 1992, which is another first prototype ever in Korea. He is currently involved in database security management as a part of transaction manager development activities and also in enterprisewise global data warehousing.


