Increasing Concurrency in Object Oriented Databases for Software Engineering Environments

Hardeep Singh, Jun Han
Peninsula School of Computing and Information Technology
Monash University, McMahons Road, Frankston, Vic 3199, Australia
{hsingh,jhan}@monash.edu.au

Abstract

Databases for software engineering environments (SEEs) store and manage the various software artifacts involved. Object oriented databases (OODBs) have been identified to be most suitable for meeting the needs of SEEs. To provide effective support, transaction models in these databases need to take into account specific characteristics of the long-duration tasks and uncertain activities in SEEs. In this paper we present a locking protocol that exploits the relationships among groups of users in SEEs to grant access for conflicting locks and achieve greater concurrency in long-lived SEE transactions. We also extend an existing semantic locking protocol for OODBs to incorporate our group-based locking protocol to obtain a comprehensive locking protocol for use in SEEs.

Keywords object oriented databases, databases for CASE, software engineering environments, concurrency control, user groups, locking protocol, transaction management, long-lived transactions.

1 Introduction

Software engineering environments (SEEs) are complex systems and involve various software artifacts manipulated by users and tools. Databases for SEEs store and manage these software artifacts and need to address specific issues such as support for long-lived transactions. Semantics associated with conventional database transactions are too restrictive for use in SEEs. This is because transactions in SEEs are characterized by their long-lived and open-ended tasks and it is unacceptable for users to wait indefinitely.

Our work presented in this paper addresses the concurrency issues for databases in SEEs. Several extended transaction models have been proposed in the past. These models enhance concurrency by relaxing one or more of the ACID properties enforced in the conventional transaction models and allow sharing of uncommitted work. Some of these approaches are not suitable in SEEs, and others require extensive user intervention which is undesirable. This problem has also been addressed in the context of user dominated cooperative work, where concurrency control is managed by the users and not by the environment. This area of research is relatively new. Thus the issue of the most suitable transaction model for databases in SEEs is very much open.

We address this issue in the context of tailorable SEEs, i.e., a generic environment that cannot be used "as is" and has to be augmented with method/project-specific information by the environment builder (or the administrator) before being used. In such environments, the environment builder has full knowledge of the software development process and the interactions among the various users and tools. Therefore it is possible to exploit project-related contextual knowledge when building an environment. In this paper we present a locking protocol for databases in SEEs. This protocol allows the builder to augment the transaction manager (TM) with project-related information (i.e., information related to group relationships in the project team, to be precise), which is exploited dynamically by the TM to achieve increased concurrency.

As is already evident from some existing research [10, 7], object oriented database management systems (OODBMSs) are most appropriate for use in SEEs. This is because OODBMSs readily support concepts like encapsulation, objects (abstract data types), inheritance and composition, combined with advances and experiences in data modeling such as semantic data models. Such support is highly desirable for modeling software artifacts [24], which can be regarded as composite objects and are hierarchical in nature. For example, a requirements document could be composed of sections, each section could be composed of paragraphs, and paragraphs could be further composed of sentences, and so on.

We propose a locking protocol in the context of object oriented databases.

Our main contributions in this paper are as follows:

1. Group-based locking protocol: An SEE supports groups of users involved in a software development project. These groups have relations which can be exploited for increased concurrency. We employ these group-based relations and present a novel locking protocol for transactions in SEEs. The group relations are known a priori to the environment builder (or the administrator) and can be augmented to the TM. We have recognized that not all relations can be determined statically and may depend upon dynamic scenarios. Thus our protocol also allows users to interact with the transaction manager and to tailor certain relations dynamically (when necessary).

2. Enhancement of a semantic locking protocol for object oriented databases: A promising semantic locking protocol which employs method semantics has been proposed for object oriented databases in [23]. We enhance this existing protocol with our group-based protocol. This results in a rather well suited protocol for object oriented databases in SEEs that exploits relations among user groups, method semantics and ordered shared locks for enhanced concurrency.

The paper is organized as follows. We first present some related work in the context of extended transaction models for SEEs in section 2. Possible relations of various groups of users and tools are defined in section 3. Next we present our group-based locking protocol in section 4, followed by a detailed example in section 5. In section 6 we discuss the primitives required for interaction with the transaction manager. In section 7 we extend a semantic locking protocol with our group-based protocol. Implementation issues of our protocol are discussed in section 8.

2 Related Work

Several extended transaction models have been proposed in various contexts. However question still remains as to which is the most appropriate for use in databases for SEEs.

The nested transaction model and the related concurrency control algorithm were proposed by Moss in [20]. It is based upon the strict two-phase locking (2PL) protocol, where a transaction cannot release any locks until it commits. Most advanced transaction models use the same structure as nested transactions, as it introduces the idea of structuring a transaction into a tree of subtransactions. However the semantics associated with two-phase locking are too restrictive for SEEs.

Sagas were proposed by Garcia Molina in [12] to deal with the problem of long-lived transactions. They are based upon the notion of compensating transactions. The contribution is that once a subtransaction is complete it can be commit without waiting for other subtransactions to complete. The release of partial results is of importance in long-lived transactions in SEE. However the major problem with Sagas is that they are useful only when each subtransaction is relatively independent and can be compensated, which is not always the case in SEEs.

Split transactions were proposed in [18] for supporting long duration activities whose actions and interactions cannot be foreseen at the beginning. In this model a transaction can be split into two serializable transactions. It allows the commitment of subtransactions and thus release useful results from the original transaction. This is useful only in user controlled transactions. The main problem with this model is that the user has to understand exactly how the transaction split should happen, and what objects should be associated with each of the split transactions. This puts a significant burden on the user.

In [2] an approach to support activities in a collaborative database environment is proposed. The proposed model is based upon identifying which set of actions collaborate and thus identifying and relaxing the notion of atomicity for the collaborative actions. Since the set of operations cannot be predetermined in SEEs this information has to be supplied by the users.

In [8] the notion of visibility domain has been introduced. A visibility domain is a set of users who can share the same data items. Each transaction has a particular visibility domain associated with it and any member of the visibility domain may start their own subtransactions on the copy of data that belongs to the transaction. This is similar to the the notion of participation domains in [17], where a set of transactions with a particular set of participants is called a domain and all other users of the database are observers. The execution of transactions in a domain is not necessarily serializable with respect to other participants in the same domain, but all transactions in different domains are serializable with respect to each other. These notions as proposed in [17] have no formal model or associated protocol.

We acknowledge that not all work in the context of advanced transaction models has been been dealt with here and there exists several other transaction models addressing similar issues. [3] provides a good survey of concurrency control issues in advanced database applications.

After analysing of the existing models for long-lived and uncertain transactions we have developed a locking-based concurrency control protocol for OODBMSs in SEEs. Our protocol exploits the rich semantics associated with groups of users and their relations in a project. We facilitate specification of
this semantic information statically, to be exploited by the transaction manager during environment execution. Our model is flexible as it also allows tailoring of group relations dynamically. By facilitating static specification we attempt to minimise the burden on the users. We have also extended an existing semantic locking protocol with our group-based locking protocol. This results in a protocol where operation semantics from OODBMSs and group semantics from projects can be exploited to achieve enhanced concurrency in SEEs.

3 Groups in SEEs and Their Relationships

Users and tools interact with an SEE to accomplish the software development tasks in a project. User interactions may be direct or via a tool. Tools on the other hand may operate in batch or interactive mode. The users and tools normally operate under a process framework which associates the tasks with different users or user groups. This results in a division of a project team into several groups, where each group may be responsible for carrying out a task such as work on a particular subsystem. It is also possible that a single user or tool may be associated with several groups possibly with different roles. The division of users into groups is not the subject of this paper and has been dealt with extensively in the context of process modeling and enactment in SEEs [13]. For the reader it is sufficient to assume that such a division exists and the division may be task-oriented. For example, in Merlin [16] groups can be constructed by the use of roles associated with the users and in [22] a model of SEEs is presented where programmers are divided into small groups who work closely.

The user groups in an SEE have relations based upon their communication protocol and the employed development process. Relations between user groups are called inter group relations. The nature of these relations make it feasible for some groups to share uncommitted work. Moreover, most of these relations are known a priori to the environment builder and can be used to augment the transaction manager (TM) statically. The TM can then exploit these relations dynamically to allow sharing of uncommitted results. Below we formally define these relations.

The inter-group relations are classified into three simple categories: friendly, hostile and neutral.

Friendly: This form of relation between groups specifies that sharing of incomplete or uncommitted work is allowed. This is the case even though the groups may be associated with different tasks or activities.

Hostile: This relation has opposite consequences as compared to the earlier one. Groups related via a hostile relation work in isolation and are not allowed to share each other’s uncommitted work. This form of relation is thus most uncooperative.

Neutral: Neutral groups on the other hand are regarded as neither friendly or hostile. Interpretation of their relationship as being friendly nor hostile is determined dynamically in a particular context and is under direct user control. This form of relation is necessary when the environment builder may not know the exact relation a priori or the relation may vary in different scenarios. Thus the user is allowed to dynamically give an exact meaning to this relation in the context of a particular scenario. This relation permits flexibility and its use will be discussed in more detail in section 4.

Friendly, hostile and neutral relations are directed. For example, if it is specified that an arbitrary group A is a friend of B, then B may access A’s uncommitted work, whereas A may not be allowed to do the same unless the reverse relation is also specified to be friendly. Each of these relations may either be global or specific. A global relation applies in the context of any artifact or activity that is associated with transactions of the participating groups. (Here the concept of activity is the same as in the context of development processes [13]). Whereas a specific relation allows us to associate a context (artifacts/activities) that must be satisfied for the relation to exist. For example two groups A and B can have a specific friendly relation which exists only when they manipulate an artifact X during an activity A_i. This gives more control to the environment builder by allowing different relations between the same groups to be specified in different contexts.

Assume two users Bart and Maggie belonging to two different groups A and B shown in Figure 1. A and B may have friendly, hostile or neutral relation. If A and B have a global and bidirectional friendly relation then Bart and Maggie may share uncommitted work in all transaction scenarios. On the other hand if a friendly relation exists from A to B when A is manipulating a requirement document, Maggie as a member of B can view any uncommitted work done by Bart on requirement document.

The issue of sharing uncommitted work between users of a single group can be regarded as a special
case where the relation of a group is expressed with itself. For example if a group is regarded to have a friendly relation with itself then group members may share uncommitted work among themselves. Besides, more precise control over sharing of uncommitted work between individual members can be achieved by using a similar mechanism with group members as separate entities. We thus shall not explicitly address this intra-group issue in the remainder of this paper.

In the next section we present our locking protocol which exploits the inter-group relations to relieve conflicting transactions, and thus to achieve greater concurrency.

4 Locking Protocol Based upon Group Relations

We assume here that the reader is familiar with some of the existing work on transactions, including the atomic transaction model, its implementation using two-phase locking [11] and the nested transaction model [20].

Object oriented databases manage composite objects and are most suitable for SEE [7, 9]. The traditional notion of conflict based upon read and write operations imposes undesirable constraints on the execution of transactions in such systems. In an object oriented database an operation invoked on a higher level object results in an entire execution hierarchy on lower level objects. Since objects in such databases have a nested structure, nested transactions are a natural choice for object oriented databases. User or tool transactions in SEE access objects by executing methods or atomic operations on objects. Atomic operations access only the local state of the object, whereas methods access both the local states and other component objects.

4.1 Assumptions

Our assumptions are as follows:

1. SEE databases are object oriented where objects are arranged in hierarchies and a method on one object can only call methods on objects that are lower in the hierarchy.

2. Locks are required only for atomic operations and not on objects or methods. Thus before a method can execute an atomic operation, it must acquire the associated lock.

3. Any transaction in an SEE is associated with a user group and an activity specified in the process model, and manipulates one or many software artifacts.

4. We also make use of the notion of transaction delegation as first proposed in ASSET [5] and recently implemented [19]. Delegation has been recognised as a powerful mechanism for extended transaction models and is supported via a primitive delegate(ti,tj,ob). It specifies that transaction ti transfers to tj the responsibility for the operations performed by ti on object ob. These operations are committed if and only if tj commits, provided tj does not delegate them to other transactions. In ASSET it has been suggested that the granularity can be lowered from objects to specific operations. However this option is not further considered in this work. In our context we assume that such support is indeed provided and assume the support of primitive operation delegate(ti,tj,oper), which specifies that transaction tj transfers to transaction tj the responsibility of a particular operation oper. We tailor the semantics of this delegation primitive by disallowing the delegating transaction from committing or aborting completely independently of the delegatee as will be evident in the discussion of our protocol below.

Here assumptions 1 and 2 are exactly the same as in [23, 91]. Assumption 3 is not unrealistic and in fact existing work [15] allows us to extract this process-based information associated with a transaction. The primitive suggested in assumption 4 is the result of some recent work on extended transaction models and its use and implementation has already been demonstrated [19].

4.2 The Protocol

User or tool operations result in the invocation of methods on objects in the SEE database. These operations may be regarded as top-level transactions. Such an operation may invoke several database methods on the software artifacts and these may be regarded as subtransactions of the top-level transaction. These database methods result in nested execution hierarchies whose roots are the methods invoked directly by the top-level operation. We assume group(T) and user(T) refer to the group and the user associated with a transaction T.

1. A method execution t' on a software artifact can execute an atomic operation t, if and only if lock(t) is requested and is granted to t'.

2. A method execution cannot terminate until all its children have terminated. When a method execution commits, its locks are inherited by its parent. If the top-level transaction commits then the locks are discarded. Locks are also discarded when any method execution aborts.

3. A lock can be granted to a method execution if no other method execution holds a conflicting lock or the retainers of the conflicting lock are ancestors of the requesting method execution.

4. A lock(t) on an atomic operation t, with top-level parent transaction Tp will be granted with a surrogate relation with respect to a conflicting
lock(x), if lock(x) is retained by top-level transaction \( T_x \) of x and either

i. group(\( T_x \)) and group(\( T_y \)) have a friendly relation that applies in the current context associated with (\( T_y \)) ; or

ii. group(\( T_x \)) and group(\( T_y \)) have a neutral relation in the current context and:
   a. user(\( T_x \) ) grants friendly relation privileges to group(\( T_y \)) for the duration of \( T_y \); or
   b. user(\( T_x \) ) postpones the access to the conflicting lock to an event in the future context of its current transaction \( T_x \). In this case the conflicting lock is temporarily refused and is only granted during a future event. The specification of when the lock will be granted in the future is in direct control of the user and can be specified using the primitives in section 6.

In the above cases a surrogate relation is established between \( T_x \) and \( T_y \) upon the grant of access to the conflicting lock and its direction is from \( T_x \) to \( T_y \).

5. For an established surrogate relation the primitive \( \text{delegate}(T_x, T_y, M') \) is executed. This gives \( T_x \) the method execution hierarchy rooted at \( M' \), where \( M' \) is a subtransaction of \( T_x \) and x is a descendant of \( M' \). In other words it now appears as if \( T_y \) had initiated and carried out the method execution rooted at \( M' \). Consequently the lock(x) is released by \( T_x \) and is acquired by \( T_y \). Either of \( T_x \) or \( T_y \) may commit or abort with mutual consent from both the associated groups and is discussed next in the commitment rule.

6. Commitment Rule: Transactions having a surrogate relation, commit and abort with a mutual consent amongst their associated groups\(^1\). If a consent cannot be resolved immediately the transaction is left as it is, to be committed or aborted at a later stage when a consent could be reached. More formally this can be specified as follows

\[
\forall i,j \mid \text{adopt}(G_i, T_x, G_j, T_y, ST_i) \Rightarrow \text{CommitConsent}(G_i, T_i, G_j, T_j) \tag{1}
\]

\[
\forall i,j \mid \text{adopt}(G_i, T_x, G_j, T_y, ST_i) \Rightarrow \text{AbortConsent}(G_i, T_i, G_j, T_j) \tag{2}
\]

These two rules specify that if any transaction \( T_y \) of group \( G_j \) adopts a subtransaction \( ST_i \) of another transaction \( T_x \) of group \( G_i \), then a mutual consent to commit or abort has to exist between \( G_i \) and \( G_j \) for \( T_i \) and \( T_j \). This consent has to be achieved via communication and notification. The SEE provides limited support in the form of user and environment notifications but the decision is made by the user possibly through negotiations outside of the SEE (section 4.3 provides more details).

The rule 1 above specifies that locks are only acquired for atomic operations. Rules 2 and 3 specify the semantics associated with nested two phase locking protocol. These three rules are identical to the ones suggested in [23]. According to rule 4 conflicting locks may be acquired if the groups of the participating transactions are identified to be friendly. This results in establishment of a surrogate relation and subtransaction being delegated (along with its associated locks).

4.3 Notification and Negotiation

The TM may notify the owners of the transaction on several occasions and prompt their response regarding delegated subtransactions and requests for commit or abort consent. These are summarized as follows in the context of any two transactions \( T_i \) and \( T_j \) and their owners \( \text{owner}_{T_i} \) and \( \text{owner}_{T_j} \), respectively:

1. \( \text{delegate}(T_i, T_j, M) \): Both \( \text{owner}_{T_i} \) and \( \text{owner}_{T_j} \) are notified only in terms of associated artifact(s) with method M. This is because to the user, the lower-level database method itself may not be meaningful. As a result of this notification, \( \text{owner}_{T_i} \) may further issue notifications to \( \text{owner}_{T_j} \). These indicate whether \( \text{owner}_{T_i} \) intends to commit, abort or is undecided on the operations associated with artifact M. The use of such a notification to \( \text{owner}_{T_i} \) is obvious. Until such a notification is received \( \text{owner}_{T_j} \) may assume \( \text{owner}_{T_i} \)'s intention being undecided.

2. Commit request on \( T_j \) after a surrogate relation is established between \( T_i \) and \( T_j \): If \( T_i \) is still active then \( \text{owner}_{T_i} \) is notified and until a mutual consent is reached between \( \text{owner}_{T_i} \) and \( \text{owner}_{T_j} \), the commit request is not granted. If \( T_i \) has already committed then no notification is sent and the consent is immediately granted to \( T_j \). Note that there is no notification required if \( T_i \) commits and \( T_j \) is still ongoing.

3. \( T_i \) or \( T_j \) issues an abort request: The \( \text{owner}_{T_i} \) and \( \text{owner}_{T_j} \) are both notified and their mutual consent is required on the request. A possible consent could be to abort the transaction \( T_j \), provided the subtransaction M delegated to \( T_j \) by \( T_i \) is committed. In this case the subtransaction M has to be delegated back to the original parent transaction \( T_i \) via the use of the same delegate primitive, if \( T_i \) is active. If \( T_i \) has terminated then a new transaction \( T' \) may be initiated by the TM whose

\(^1\) Even though it is obvious that consent has to be reached between the owner of two transactions, we use groups. This is because within a group the responsible person could be the owner of the transaction or some other individual. The use of the term owner in our context means the person(s) responsible for the transaction, these may not necessarily be always the owner(s).
ponent class Gadget of
ing changes to Gadget, the results cannot be visi-
requires making changes to the two classes Driver
existing functions and adding new functions. Mag-
tation work and incorporate the changes made in
Trigger classes.
the design by Maggie into his C++ code. Bart
has to wait till Maggie completes all her changes
made to subsys-A. She first browses
realises that the changes need to be made to a com-
atomic operations applicable on an object of that
type.
Figure 2: Composition of a Class Diagram

5 Example
Assume that an SEE being used in a project, sup-
supports specification of design in Booch’s design lan-
guage [6] and implementation in C++. Further as-
sume the existence of two users Bart and Maggie, as-
associated with groups class-implementors and
detailed designers respectively. A particular activity
associated with detailed-designers is to define class
diagrams in Booch’s design language. Role of the
class-implementors is simply to write the implemen-
tation in C++ of the design developed by the class-
designers. Figure 2 represents part of the composi-
tion structure of class diagram as represented in the
SEE database. Here class diagram is composed of
many classes (indicated by adornment 1 and m on
the edges of the link), class categories, metaclasses,
etc. A class in turn has attributes and operations as
its components. In an object oriented SEE database
each entity in Figure 2 has a corresponding type
associated with it which dictates the methods and
atomic operations applicable on an object of that
type.
Suppose that a bug X is discovered in the soft-
ware product which requires changing its design and
implementation. More specifically changes are re-
quired to class diagram subsys-A. Maggie goes about
making changes to subsys-A. She first browses
subsys-A to identify the location of changes. She
realises that the changes need to be made to a com-
ponent class Gadget of subsys-A and goes about
editing Gadget, making changes to some of its ex-
isting functions and adding new functions. Mag-
lie then realises that her changes to Gadget also
requires making changes to the two classes Driver
and Trigger. Maggie now proceeds to make these
changes. Now even though Maggie has finished mak-
ing changes to Gadget, the results cannot be visi-
tible outside till Maggie completes her entire trans-
saction and finishes making changes to the Driver
and Trigger classes.
Bart is eager to get started on his implemen-
tation work and incorporate the changes made in
the design by Maggie into his C++ code. Bart
has to wait till Maggie completes all her changes
and commits. This waiting may be unacceptable
because Bart could be idle for a long duration.
Using traditional two phase locking Bart would
not be able to access the changes done to Gadget.
This is because Maggie acquires a write lock on
Gadget and Bart requires a read lock. Since read
and write locks are incompatible such an access will
not be granted. However since Maggie is done with
her changes to Gadget it is desirable that her changes
be revealed to Bart.
This problem would not be solved by supporting
parallel versions of Gadget. This is because Bart
requires the latest version of Gadget as his changes
to the code need to incorporate what Maggie just
changed in the design of Gadget.
In Figure 3 we show the partial execution hierar-
chy for Maggie’s transaction T1 and Bart’s transac-
tion T2. As in [14] TO represents a fictitious ob-
ject, called environment whose methods are user
transactions. Atomic operations are shown in bold
and form the leaves of the execution tree. Internal
nodes of the tree are the methods. If a method
wants to execute an atomic operation it must first
acquire a lock on it. We also indicate which lev-
els in the tree are associated with which compo-
nent objects according to the composition hierar-
chy of Figure 2. Our particular interest is in the
method invocation updateClass() by T1. Execu-
tion of this method results in a nested invocation
where a method addOperation() requires a lock on
atomic operation createOperation(). Since at this
time no other method execution is holding a conflict-
ning lock, the lock is granted and method
addOperation() executes successfully. When the
method completes, all its locks are handed over to
the parent. In this case the lock is passed to method
updateClass(). As update class finishes executing,
it passes its locks to its parent which is now the
enclosing user level transaction T1. Using two phase
locking, only when T1 commits are the locks on
atomic operations released and the changes to class
Gadget are revealed.
Now consider the transaction T2 invoked by Bart.
Bart is interested in knowing the changes done to
class Gadget so he invokes a getClass() method of
class diagram. This method results in a nested invo-
cation where methods readAttributes() and
getOperations() require lock to atomic operations
readAttributes() and readOperations(). How-
ever T1 holds a lock on atomic operation
createOperation() which conflicts with the lock
request for readOperations().
We can relieve this conflict by exploiting the re-
lations between the groups detailed-designers and
class-implementors associated with Maggie and Bart
and consequently with T1 and T2. Assume it is
known statically that Maggie and Bart’s groups need
not work in isolation as that may lead to intolera-
able delays. If detailed-designer group and class-implementor group have a friendly relation or they have neutral relation and Maggie converts it into a friendly relation then the conflict can be relieved. According to our locking protocol Bart is allowed access and the entire method execution hierarchy rooted at updateClass() now becomes a subtransaction of T2 via delegation. This results in all the locks for atomic operations passed by updateClass to its parent T1 to be handed over to T2. Commit and abort of T1 and T2 is now done with mutual agreement between the detailed-designer group and class-implementors group. The effect of relieving conflict results in the execution hierarchy shown in Figure 4.

This example has shown that our protocol is very useful in the case when a subtransaction is considered done by the owner but results are simply not released due to the semantics associated with strict two phase locking protocol. The application of our protocol achieves increased concurrency especially in the context of long-lived transactions (e.g., where Maggie's activities could last a couple of days).

6 User Interaction Primitives with the Transaction Manager

Users or resources of the software engineering environment may interact with the transaction manager via a limited set of primitives. These are in addition to the ones normally required in any transaction model such as begin, commit, abort etc. These primitives are important for specifying the complete semantics of our model and are explained briefly as follows:

1. SuspendFriendship(Groupi): Suspend friendly relation in the context of the current transaction. The parameter is optional and absence of the parameter results in the suspension of all friendly relations in the context of the ongoing transaction. No longer are the intermediate results of the ongoing transaction revealed to any other transaction and neither does the user issuing the primitive, interact with the transaction manager to resolve any neutral relations. This implies that for the associated transaction, the group-based locking protocol is no longer used and the default semantics of an ordinary two-phase protocol for nested transactions applies. The parameter allows a finer control on friendly relations. Specification of a group Groupi as the parameter allows a suspension of the friendly relation only with that group.

2. ResumeFriendship(Groupi): This resumes friendly relations suspended using the above primitive. The parameter is optional as in the previous case, and has similar effect. The use of the parameter is the same as in the earlier primitive.

3. MakeFriend()/DenyFriend(): In case a request for a conflicting lock occurs and the relation between two participating groups is identified to be neutral, the TM notifies the owner of the transaction holding the conflicting lock. This owner decides whether the transaction requesting the conflicting lock can be allowed access or not. Access is allowed by invoking the MakeFriend primitive. This results in the requesting transaction to be treated as if it was initiated by a friendly group and the semantics of our group-based locking protocol is used. The DenyFriend primitive is used to deny establishing a friendly relation. In this case the relation is still regarded as neutral in the context of the current transaction.

4. PostponeDecision(): A user may wish to postpone his decision regarding a neutral relation. In this case the request for a conflicting lock has to wait for further response from the user. The TM may choose to prompt the user at the end of each subtransaction to determine if the user has reached a decision. The user may again reuse this primitive or one of the earlier primitives to postpone, grant or deny a friendly relation. It is possible that the user may have postponed his decision regarding several such requests. In that case the transaction manager allows the user to see the requests that were postponed and grant them a friendly relation selectively. However if more than one transaction, out of the ones selected to be friends, is waiting on
the same conflicting lock only one of them acquires it and proceeds successfully on its method execution. The selection is according to the order of request for the conflicting lock.

7 Extending Semantic Locking with Group-Based Relations

In [23] a locking protocol for object oriented databases has been presented. This protocol generalises Moss’s original nested transaction protocol by employing rich semantic information available in object oriented databases and incorporates the notion of ordered shared locks. The protocol incorporates the notions of higher level objects [4, 21], nested executions [14, 1] and dynamic conflicts. Locks are required only for atomic operations. The main contribution of this protocol is that the semantics of higher level operations are exploited to resolve conflicts regarding atomic operations, and supports referential sharing among disjoint objects.

This protocol is very promising for object oriented databases. In this section we extend this semantic locking protocol to incorporate our group-based locking protocol. The resultant protocol is rich and highly suitable for use in SEEs. For simplicity we keep the discussion of semantic locking protocol to a minimum. Its complete description can be found in [23].

Protocol: The extended semantic locking protocol is described as follows.

1. A method execution executes an atomic operation after acquiring its lock.
2. A method execution cannot terminate until all its children have terminated. When a method execution commits it passes its locks to its parent. If the parent is the top-level user transaction and it commits then all the locks it holds are released.
3. A request for a conflicting lock can be granted if no other method execution holds a conflicting lock or the retainers of the conflicting lock are ancestors of the requesting method.
4. A lock request for an atomic operation that conflicts with another lock can be granted using semantic information. This is possible if any descendant of the retainer method of the conflicting operation and any ancestor of the method requesting the conflicting lock commute.
5. If conflicting lock cannot be granted by any of the above cases then the group-based protocol semantics presented in section 4 is employed to see if the request can be granted. If the request is granted then a surrogate relation is established.
6. If a conflicting lock cannot be granted by any of the previous cases, it is granted via an ordered shared relationship which imposes a commitment order on the requesting method. In this case the requesting method waits for the method with which an ordered shared relation has been established to terminate. The ordered shared relation may be removed if the method execution reaches some point where commutativity of ancestor methods or group relations can be used.

Here the rule 1 requires locks to be acquired only for atomic operations. Rule 2 and 3 enforce the strict two-phase locking protocol for nested method executions as discussed in section 4. Rule 4 relaxes the nested two phase locking constraint by using a commutativity relation between method executions based on semantic information. Rule 5 specifies that a lock can also be granted by exploiting the group-based relations. A lock can be granted if the group of the requester enjoys a friendly status or is temporarily granted a friendly status, thus establishing a surrogate relation. Rule 6 allows the access to be granted via an ordered shared lock. Rules 1, 2, 3, 4 and 6 have been adopted from [23].

This extended semantic locking protocol is very suitable for SEEs as it exploits method and group semantics to achieve maximum concurrency.
We discuss the implementation issues of our group-based locking protocol. Implementation issues for semantic locking have already been discussed in [23].

A matrix as shown in Figure 5 is employed by the TM. This matrix maintains information regarding inter-group relations. A particular entry in the matrix, say $C_{ij}$, specifies the relation between two groups $G_i$ and $G_j$. This entry could either be a single value specifying that the associated groups have friendly, hostile or unfriendly relation in all contexts or alternatively could refer to three lists. One for specifying the friendly relation, one for hostile (unfriendly) relations and one for neutral relations.

Each entry in these lists specifies a pair: an artifact type and an activity, which indicates the context in which the relation exists. For example, two groups could have a friendly relation when an artifact A is being manipulated in activity subsystem-design. Not both elements of the pair are mandatory, i.e., either the activity or the artifact type can be specified alone. The shaded area in the matrix indicates the diagonal elements of the matrix. These basically express relations regarding transactions from the same group. By default, two transactions from the same group are regarded as if they are from two different groups with a friendly relation. However, this can be changed to incorporate group semantics in the same manner as for relations between different groups.

We assumed that each transaction in a software engineering environment has an activity and a group associated with it. This information may be conveyed to the TM as in [15]. The F.U.N matrix is specified statically by the environment builder, and is referred dynamically by the transaction manager to relieve conflicts. If the conflict is relieved then the corresponding nested execution trees are adjusted accordingly as shown in Figure 4.

The TM also provides support for automatic redelegation to the original parent if both transactions are active or by automatically issuing a new transaction that is delegated the desired subtransaction and then commits. However, we will not further deal with these issues in the context of this paper.

9 Conclusions

We have presented in this paper a group-based locking protocol for OODBMSs in the context of long-lived, uncertain, and interactive transactions required by SEES. It exploits domain knowledge of software engineering, i.e., the nature of relationships between user groups in a project team. In particular, such a relationship can be classified as friendly, hostile or neutral. If desired, it can be stated relative to an artifact and/or an activity, to provide finer-grained control. Our protocol achieves greater concurrency by allowing the sharing of uncommitted results between friendly groups.

The group relationships in our protocol are specified statically by the environment builder, and are used dynamically by the transaction manager to relieve conflicting locks. The static specification of the group relationships minimizes the burden on the user in reducing the need of user intervention regarding group relationships. Besides, our model also provides the flexibility of tailoring relationships dynamically (when necessary), through run-time customisation of neutral relationships. The combination of static specification and dynamic customisation of group relationships provides a practical and flexible approach to interaction with the transaction manager.

Our group-based locking protocol has also been incorporated into an existing semantic locking protocol for OODBMSs using method commutativity. This results in a novel protocol that exploits both group and method semantics to achieve increased concurrency. We are currently planning to incorporate the combined protocol in the object management system of a (generic) software engineering environment.

Acknowledgements

We would like to thank Dr. Rodolfo Resende for invaluable comments on an earlier draft of this paper.

References


