

Exploiting User-Group Relationships for Increasing Concurrency in Software Engineering Environments

Hardeep Singh, Jun Han

*Peninsula School of Computing and Information Technology
Monash University, McMahons Road, Frankston, Vic 3199, Australia
{hsingh,jhan}@mars.fcit.monash.edu.au*

Abstract

Software engineering environments (SEEs) emerged in order to address the problem associated with developing and maintaining large software projects. Databases in SEEs store and manage the various software artifacts that result from the different phases in the software development cycle. The traditional notion of atomicity and serializability is too restrictive in SEE databases. This is because transaction models in SEE databases need to take into account long-duration activities that may last from days to months. Users need to share partial results of their ongoing activities as it is undesirable to keep other users waiting indefinitely. Here we address the concurrency issue in SEEs. We present a locking protocol that exploits relations among groups of users for achieving enhanced concurrency in SEEs.

1. Introduction

Software development in a project is a complex task which involves several team members. These team members perform related activities and operate on a variety of software artifacts which have complex dependencies. Software engineering environments (SEEs) have been developed to support the various aspects of the software development process. Databases for SEEs provide the facility to manage the project data, mainly the software artifacts. These databases are custom-made or general purpose systems adapted to a specific SEE.

Extensive work has been done in the area of appropriate data and transaction models for design applications including CASE. The traditional data models (such as the relational model) and the traditional transaction model of flat, short and serializable transactions

have been successfully used in traditional database applications. However these have been identified to be lacking in providing enough support to meet the needs of non-standard applications such as CASE. Recent work [8, 5] has shown that object-oriented database systems (OODBSs) are most suitable to meet the needs of SEEs. They are especially desirable for SEEs because their data model supports concepts like encapsulation, objects (abstract data types), inheritance, composition, etc. Such support is highly desirable for modeling software artifacts [22], which can be regarded as composite objects and are hierarchical in nature.

A major issue that databases for SEEs need to address is transaction management. Short, atomic and conflict-serializable transactions are not appropriate in this domain. This is because transactions in SEEs may last over long periods and are characterized by the need to let the developers cooperate. Correctness criterion of conflict serializability used traditionally is by far too restrictive, prevents cooperation and forces users to wait indefinitely which is highly undesirable. Thus research on databases for SEEs has to address the issue of long-lived transactions. Several non standard 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 are often combined with the notion of private user workspaces (checkout/checkin). Also a transaction model may be adequate in one development process and may be unsuitable in another. Thus ideally a database for SEEs should allow specification of different cooperation strategies depending upon the needs of a specific process and not impose a particular one on all processes.

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

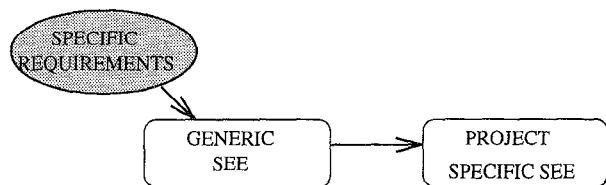


Figure 1. Generation of a project specific SEE from a generic SEE

administrator) before being used (see figure 1). 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.

Our contributions in this paper are as follows:

- We have identified a set of relations between groups of users in SEEs.
- We provide a novel locking protocol which employs relations among user-groups to achieve enhanced concurrency in SEEs. Several user-group relations for a particular project are known to the environment builder and can be augmented to the TM before the environment is enacted and used. We also recognise the fact that not all user-group relations can be determined statically and thus 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).

The paper is organized as follows. We first present the various user-group relations in section 2. In section 3 we present our locking protocol. A detailed example to facilitate complete understanding of our protocol is presented in 4. In section 5 we discuss the primitives required for interaction with the transaction manager. In section 6 we discuss some existing work on extended transaction models.

2. User Groups and their Relationships in SEEs

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 [11]. For the reader it is sufficient to assume that such a division exists and the division may be task-oriented. For example, in Merlin [14] groups can be constructed by the use of roles associated with the users and in [20] 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 3.

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 [11]). 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. 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.

3. Advanced Locking Protocol for Long Transactions

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 [9] and the nested transaction model [18].

Object oriented databases manage composite objects and are most suitable for SEEs [5, 7]. 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 SEEs 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.

3.1. Assumptions

The work presented in this paper is based upon the following assumptions:

- i. 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.
- ii. 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.
- iii. 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.
- iv. We also make use of the notion of *transaction delegation* as first proposed in ASSET [3] and recently implemented [17]. Delegation has been recognised as a powerful mechanism for extended transaction models and is supported via a primitive $\text{delegate}(t_i, t_j, ob)$. It specifies that transaction t_i transfers to t_j the responsibility for the operations performed by t_i on object ob . These operations are committed if and only if t_j commits, provided t_j 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 $\text{delegate}(t_i, t_j, \text{oper}_i)$, which specifies that transaction t_i transfers to transaction t_j the responsibility of a particular operation oper_i . 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 [21, 19]. Assumption 3 is not unrealistic and in fact existing work [13] 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 [17].

3.2. 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 $\text{group}(T)$ and $\text{user}(T)$ 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 $\text{lock}(t)$ 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 $\text{lock}(t)$ on an atomic operation t , with top-level parent transaction T_y will be granted with a *surrogate* relation with respect to a conflicting $\text{lock}(x)$, if $\text{lock}(x)$ is retained by top-level transaction T_x of x and either

- i $\text{group}(T_x)$ and $\text{group}(T_y)$ have a friendly relation that applies in the current context associated with (T_y) ; or
- ii $\text{group}(T_x)$ and $\text{group}(T_y)$ have a neutral relation in the current context and:
 - a. $\text{user}(T_x)$ grants friendly relation privileges to $\text{group}(T_y)$ for the duration of T_y ; or
 - b. $\text{user}(T_x)$ 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 5.

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_y the method execution hierarchy rooted at M' , where M' is a subtransaction of T_x and x is a descendent 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 $\text{lock}(x)$ 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 groups¹. 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

$$\begin{aligned} & \forall i, j \mid \text{adopt}(G_i.T_x, G_j.T_y, ST_i) \\ & \Rightarrow \text{CommitConsent}(G_i.T_i, G_j.T_j) \end{aligned} \quad (1)$$

¹Even 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).

$$\begin{aligned} \forall i, j \mid \text{adopt}(G_i.T_x, G_j.T_y, ST_i) \\ \Rightarrow \text{AbortConsent}(G_i.T_i, G_j.T_j) \end{aligned} \quad (2)$$

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 3.3 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 [21]. 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).

3.3. 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 $owner_{T_i}$ and $owner_{T_j}$ respectively:

i. *delegate*(T_i, T_j, M) : Both $owner_{T_i}$ and $owner_{T_j}$ 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, $owner_{T_i}$ may further issue notifications to $owner_{T_j}$. These indicate whether $owner_{T_i}$ intends to *commit*, *abort* or is *undecided* on the operations associated with artifact M. The use of such a notification to $owner_{T_j}$ is obvious. Until such a notification is received $owner_{T_j}$ may assume $owner_{T_i}$'s intention being undecided.

ii. **Commit request on T_j after a surrogate relation is established between T_i and T_j :** If T_i is still active then $owner_{T_i}$ is notified and until a mutual consent is reached between $owner_{T_i}$ and $owner_{T_j}$, 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.

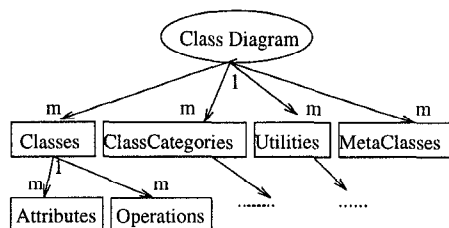


Figure 2. Composition of a Class Diagram

iii. **T_i or T_j issues an abort request:** The $owner_{T_i}$ and $owner_{T_j}$ 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 subtraction 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 sole purpose is to become the delegatee for M and then commit. This can be achieved by execution of the primitive *delegate*(T_j, T', M).

4. Example

Assume that an SEE being used in a project, supports specification of design in Booch's design language [4] and implementation in C++. Further assume the existence of two users Bart and Maggie, 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 implementation in C++ of the design developed by the class-designers. Figure 2 represents part of the composition 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 software product which requires changing its design and implementation. More specifically changes are required 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 compo-

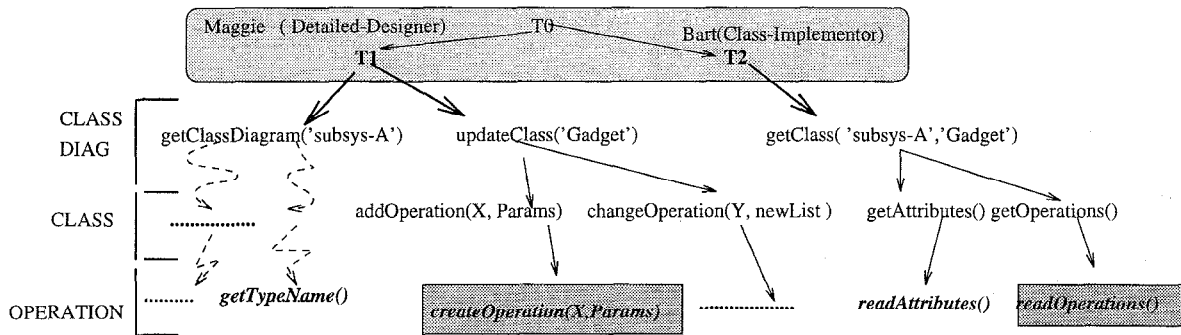


Figure 3. Method Execution Hierarchy

ment class *Gadget* of *subsys-A* and goes about editing *Gadget*, making changes to some of the its existing functions and adding new functions. Maggie 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 making changes to *Gadget*, the results cannot be visible outside till Maggie completes her entire transaction and finishes making changes to the *Driver* and *Trigger* classes.

Bart is eager to get started on his implementation 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 hierarchy for Maggie's transaction T1 and Bart's transaction T2. As in [12] T0 represents a fictitious object, 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 levels in the tree are associated with which component objects according to the

composition hierarchy of Figure 2. Our particular interest is in the method invocation `updateClass()` by T1. Execution 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 conflicting 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 invocation where methods `getAttributes()` and `getOperations()` require lock to atomic operations `readAttributes()` and `readOperations()`. However T1 holds a lock on atomic operation `createOperation()` which conflicts with the lock request for `readOperations()`.

We can relieve this conflict by exploiting the relations 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 intolerable 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 re-

sults 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 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).

5. 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:

i. *SuspendFriendship(Group_i)*: 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 *Group_i* as the parameter allows a suspension of the friendly relation only with that group.

ii. *ResumeFriendship(Group_i)*: 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.

iii. *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 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.

iv. *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.

6. 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 [18]. 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 [10] 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

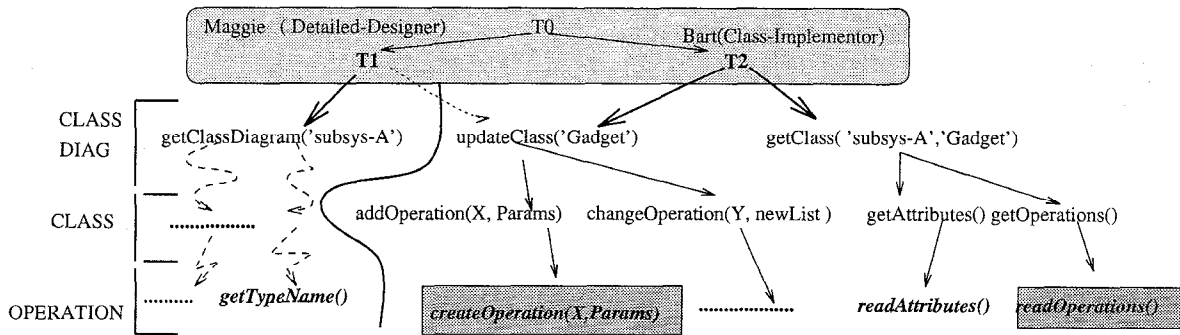


Figure 4. Method Execution Hierarchy Using Group Protocol

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 [16] 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 significant burden on the user.

In [1] 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 [6] 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 [15], 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 [15] have no formal model or associated protocol.

We acknowledge that not all work in the context of advanced transaction models has been dealt with here and there exists several other transaction models addressing similar issues. [2] 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 OODBs 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.

7. Conclusions

We have presented here a locking protocol that exploits relations among user-groups to grant conflicting locks. This allows sharing of uncommitted results among project members thus enhancing concurrency. Relations among user-groups in a project team are classified as friendly, hostile or neutral. Environment builder specifies these relationships statically. These may be tailored dynamically by customising neutral relationships for different scenarios. Due to the fact that most of the group relations are specified statically, our protocol requires less user intervention.

The protocol presented in this paper will be implemented on top of O2, an existing object oriented database management system, and used in an SEE to determine its effectiveness. The results of these experiments will be made available in our future publications.

References

- [1] D. Agrawal, J. Bruno, A. Abbadi, and V. Krishnaswamy. Relative serializability: An approach for relaxing the atomicity of transactions. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems PODS 94*, pages 139–149. ACM Press, May 1994.
- [2] N. Barghouti and G. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.
- [3] A. Biliris, S. Dar, N. Gehani, H. Jagadish, and K. Ramamritham. Asset: A system for supporting extending transactions. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 44–54, May 1994.
- [4] G. Booch. *Object Oriented Analysis and Design with Applications*. Benjamin/Cummings Publishing Company, Inc, 1994.
- [5] A. Dahanayake and G. Florijn. Evaluation of object-oriented database support for software engineering environments. In *Seventh Conference on Software Engineering Environments*, pages 21–33. IEEE, April 1995.
- [6] M. Dowson and B. Nejme. Nested transactions and visibility domains. In *Proceedings of the 1989 ACM SIGMOD Workshop on Software CAD Databases*, pages 36–38, February 1989.
- [7] W. Emmerich. *Tool Construction for Process Centered Software Development Environments Based on Object Database Systems*. PhD Thesis, University of Paderborn, Germany, 1995.
- [8] W. Emmerich, W. Schafer, and J. Welsh. Databases for software engineering environments – the goal has not yet been attained. In *Proceedings of the 4th European Software Engineering Conference*, volume LNCS, pages 145–162, Garmish-Partenkirchen, Germany, 1993. Springer.
- [9] K. Eswaran, J. N. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–632, November 1976.
- [10] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*. ACM Press, 1987.
- [11] P. Garg and M. Jazayeri. *Process-centered software engineering environments*. IEEE Computer Society Press, 1995.
- [12] T. Hadzilakos and V. Hadzilakos. Transaction synchronization in object bases. *Journal of Computer and System Sciences*, pages 2–24, 1991.
- [13] G. Heineman and G. E. Kaiser. The Cord approach to extensible concurrency control. Technical Report CUCS-024-95, Department of Computer Science, Columbia University, 1996.
- [14] G. Junkermann, B. Peuschel, W. Schafer, and S. Wolf. Merlin: Supporting cooperation in software development through a knowledge-based environment. In G. J. et al., editor, *Software Process Modelling and Technology*. Research Studies Press, Somerset, UK, 1994.
- [15] G. Kaiser. A flexible transaction model for software engineering. In *Proceedings of Sixth International Conference On Data Engineering*, pages 560–567, February 1990.
- [16] G. Kaiser and C. Pu. Dynamic restructuring of transactions. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 8, pages 265–295. Morgan Kaufmann, 1992.
- [17] C. Martin and K. Ramamritham. Delegation: Efficiently rewriting history. Technical Report 1995-090, Department of Computer Science, University of Massachusetts, October 1995.
- [18] J. Moss. *Nested Transactions: An approach to Reliable Distributed Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, 1981.
- [19] P. Muth, T. Rakow, G. Weikum, P. Brossler, and C. Hasse. Semantic concurrency control in object oriented database systems. In *Proceedings of International Conference on Data Engineering*, 1993.
- [20] D. Perry and G. Kaiser. Models of software development environments. In *10th International Conference on Software Engineering*, pages 60–68, April 1988.
- [21] R. Resende, D. Agrawal, and A. E. Abbadi. Semantic locking in object-oriented database systems. In *OOPSLA Proceedings*, pages 388–402, 1994.
- [22] H. Singh and J. Han. Modelling software artifacts and their relationships in software engineering environments. Technical Report 96-06, Peninsula School of Computing, Monash University, Melbourne, Australia, May 1996.