Managing Concurrent Activities in Collaborative Environments*

D. Agrawal J. L. Bruno A. El Abbadi V. Krishnaswamy
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

An approach to support activities in collaborative databases is proposed. This approach merges flexible transaction models from collaborative environments and semantics based correctness criteria. A model is first presented which incorporates the notion of collaboration in the context of transactions. A protocol is then developed based on this model for efficient management of collaborative activities. This is the first attempt to formally incorporate the notion of collaboration in the database management system. Most earlier approaches provide tools and mechanisms to circumvent the isolation requirement of transactions. The protocol is a generalization of strict two phase locking since in the absence of collaboration, it reduces to strict two-phase locking.

1 Introduction

Database management systems are increasingly being used to model non-standard applications such as CAD/CAM databases, scientific databases, spatial databases, etc. Many of these applications require the efficient management of collaborative activities among its users and the traditional database paradigm of transaction is found to be too restrictive. In particular, the transaction paradigm guarantees isolation of one user from other concurrent users of the database. The isolation of transactions in multi-user database systems is enforced by maintaining entire transactions as single atomic units with respect to each other. We will refer to this notion as *absolute atomicity.* In a collaborative database absolute atomicity is overly restrictive, since a user may want to expose some of its results to one user while at the same time hiding these results from another user. Hence, we need a more powerful paradigm for transactions in collaborative environments.

*This work was supported by the NSF under grant number IRI-9117904 and by the NASA under grant number NAGW-3888.

The requirements for adequate models and support for collaboration in software engineering databases have been specified widely [Sor87, Kai90, GKM93, GHMS94]. In particular, researchers working in the area of cooperative work and participatory design [Kai90, GHMS94] have put forward several ways in which collaboration can occur among members in a design team. The first simple model is called *separate responsibilities*. In this model, the collaborative database is partitioned among several users in a disjoint manner. Cooperation then involves one user making use of designs developed by others. Another approach is referred to as *turn taking* and is basically an application of mutual exclusion or serial execution of user tasks on shared data. This approach requires significant coordination of collaborative users outside the realm of databases. A less restrictive variation of this approach is based on the notion of *alternative versions*. In this setting of collaboration, different users may develop alternate versions of the same part (thus obviating the need for mutual exclusion) in the engineering design and later team members have to interact closely in merging these alternative versions for the overall design.

The above approaches have been traditionally used in collaborative databases and cooperative design work but have significant drawbacks. In particular, they require close supervision and management of the collaboration by human users and are hard to automate if adequate support is not available at the level of the database management system. Complex tools and techniques are required to oversee this collaboration process, e.g., COSMOS [WBMN88], the ObServer System [SZ89], and Integrated Project Support Environment [VS91]. Ideally, these tools should be integrated with the database system itself. In the arena of software engineering environments, researchers would like the database management system to support the notions of *dynamic exchange* and *synchronous sessions*. In particular, the dynamic exchange model requires that the underlying database management system provide protocols and support, such that user transactions have the capability of dynamically including some subset of users in the design task while at the same time excluding others. The key requirement is that this inclusion/exclusion should occur dynamically. In contrast, the traditional transaction mechanism only supports exclusion of all users from a particular user task. The notion of synchronous sessions roughly corresponds to several users executing a single transaction simultaneously. There has been very little work in the area of transaction management that supports these latter two models of collaboration.

In this paper, we develop a coordination protocol to support the execution of collaborative transactions. The protocol is a generalization of the well-known strict two-phase locking protocol. Furthermore, if all transactions in the system are specified using the traditional notion of absolute atomicity, i.e., each user task excludes all other concurrent user tasks from seeing its partial results, the proposed protocol reduces to strict two-phase locking. The paper is organized as follows. We begin with a brief overview of the related work in the area of collaborative databases and semantics based protocols for transaction management in Section 2. In particular, we point out that the two very diverse bodies of work can be integrated with each other: the former aims to extend the notion of a traditional transaction to support collaboration and the latter aims to increase concurrency in database systems by employing application semantics. We then provide motivation for the need for collaboration in database systems in Section 3. In Section 4, we present the model for collaborative transactions. The protocol for managing such transactions is presented in Section 5. The paper concludes with a brief discussion of our results.

2 Related Work

Our approach in this paper derives from two main sources. Its main theoretical basis is from the attempts by database researchers to relax the traditional notions of atomicity and isolation that are associated with classical serializability. Its applicability and practical implications derive from the various proposals and examples developed by software engineering and design environments research, which has often advocated more ad hoc but more appropriate mechanisms than transactions for various applications that may benefit from database concepts. We believe our approach integrates these two distinct bodies of research and derives a new approach to solve some of the problems that have plagued advanced database applications, e.g., long lived transactions and the strict isolation of transactions.

Garcia-Molina [Gar83] pioneered the effort to use the notion of relative atomicity to increase concurrency among transactions in database systems. He proposed grouping transactions into compatibility sets, where transactions in one such set may be arbitrarily interleaved, but transactions in different sets observe each other as single atomic units. Lynch [Lyn83] extended Garcia-Molina's specifications from two-level compatibility sets to hierarchically structured interleaving sets, allowing transactions to have varying atomic units relative to each other. Lynch [Lyn83] introduced this notion and refers to it as *relative atomicity*. Farrag and Özsu [FÖ89] used the notion of breakpoints, which provides maximal freedom for relative atomicity specifications. In addition, they propose a graph which aids in the recognition of executions that are "equivalent" to correct executions. The most serious drawback of Farrag and Özsu's proposal is that the recognition of executions that are equivalent to correct executions has an exponential complexity [KB92]. Recently, a theory of relative atomicity [ABEK94] has been proposed for reasoning about the executions of transactions with semantic information. In particular, a graph-based tool is provided to determine if a given execution of transactions is correct. The acyclicity of this graph is both a necessary and sufficient condition for correctness. The main contribution of this theory is that unlike previous proposals [FÖ89], the recognition of correct executions can be done efficiently. Shasha et al. [SSV92] recently proposed a mechanism for chopping transaction while still preserving the traditional criterion for correctness. This approach uses limited semantic information of transactions and its main motivation is to improve performance by splitting transactions into smaller units.

Kaiser has discussed various scenarios where relaxed forms of transactions may be appropriate for activity coordination among teams of researchers in multi-user software development environments[Kai90]. A notion of *commit serializability* is proposed where a transaction may be divided during operation and parts committed separately. These ideas were implemented by the notions of *split-transactions* and *join transactions* [PKH88]. Unlike relative atomicity, once a transaction splits and subsequently commits, its results are visible to all the users of the database. To incorporate the notion of teams of participants, the notion of *participation domains* is introduced [Kai90]. 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. In [Kai90] these notions are proposed but no formal model or protocols are presented.

Skarra and Zdonik [SZ89] introduce the notion of *transaction groups*, which explicitly defines a nested framework for *cooperating transactions*. Within a transaction group, cooperating transactions and subgroups are synchronized using *semantic patterns*, which are defined by abstract data types. A semantic pattern is an operation sequence that represents semantic actions and preserves consistency within and between the database objects. Traditional locks are extended to incorporate object semantics and patterns to relaxed notions of conflict. This approach provides a low level mechanism for incorporating both the semantics of objects as well as semantics patterns to increase interleavings between collaborating transactions, when the patterns of interactions are known a priori. Several other proposals have been made to adapt transactions to software development environments [BKK85, WBMN88, RRD90, SV91]. These proposals, in general, provide a set of tools for facilitating collaboration in design databases. These tools often involve managing design versions or implementing check-in/check-out techniques. A survey of version related issues for engineering databases are reported in [Kat90]. Barghouti and Kaiser [BK91] provide an excellent survey of the concurrency control issues in advanced database applications.

3 Motivating Collaboration

In a collaborative environment, teams of researchers may be executing concurrently on different tasks, often interacting closely within each team as well as across teams. Such interactions, especially across teams, need to be regulated to ensure that no incorrect flow of information takes place and that the overall goal of the collaboration is consistent and correct. To illustrate some of these ideas, we use a geographic application from the Earth Observation System (EOS) project [GM93]. Our particular example is typical of many software collaborative environments.

In the EOS project, raw data is collected from satellites at a centralized repository, and then distributed to various secondary sites, where scientists apply standardized algorithms to derive maps or data that corresponds to their particular applications. Programs/algorithms for producing derived maps are often developed by teams of scientists and software engineers, and then are made available to the science community. As more efficient, accurate, and versatile algorithms are developed the software engineers create new versions of already existing programs. These programs are often large software systems and the update process may involve several individuals who cooperate together. Furthermore, at any point, scientists from the user community may request to use/copy the programs. Note that often the user scientists are not concerned as to whether they get the latest version or not, since the functionality often doesn't change, rather, the new versions are more efficient and may provide more services and options.

A straightforward approach to implement the above requirements would be to provide the user scientists with a simple "ftp" facility for retrieving the remote program. This approach works as long as the programs are simple and are included in a single file that is updated by a single individual. However, if a team of software engineers collaborate together to update a set of files implementing the new algorithms, a user may retrieve various files at different stages of development of the final product, thus resulting in an inconsistent and possibly incorrect program. Therefore, some degree of synchronization and coordination between the software designers and the remote users/scientists becomes necessary. For simplicity, consider two software developers, *Alice* and *Bob* cooperating to write a program consisting of two components: *main* and *proc*. Remote users need to read both procedures in a consistent manner from the point-of-view of the creators/updaters of the two components. One simple approach would be to associate with each procedure a lock with read and write options. A simple concurrency control based approach would require all accesses to be coordinated through the acquisition of locks. Hence, a read (write) lock is acquired before reading (writing), and the lock is held until the end of the user's interaction. This would ensure that the components are not changed during the period when they are being used.

Assume each of the software developers needs to read *both main* and *proc* before each can start updating the program segment they have been allocated. Note that each of these developers may need a long time before updating the file (a long lived transaction). Hence, *Alice* locks both *main* and *proc* for reading, then for the entire development process, which may last for days, *Bob* may not write a new version of *proc*. Since *Alice* and *Bob* are collaborating, the strict coordination imposed by a locking protocol is too strict. In particular, we should not be concerned if the execution of

Alice and *Bob* are not atomic with respect to each other. The sole concern is that an outside scientist should have an atomic view with respect to both developers.

Depending upon the model of collaboration, different scenarios could arise. For example, *Alice* and *Bob* may choose to perform their updates independently. In this case, both software developers read *main* and *proc* but *Alice* updates *main* and *Bob* updates *proc*. Since there is no collaboration between *Alice* and *Bob*, the two developers do not put any constraint on the users. In particular, it would be perfectly reasonable to allow an outside user to access the two modules after *Alice's* updates but before *Bob's* updates. On the other hand, *Alice* and *Bob* may set up a collaboration channel between them and may want to exclude users from reading partial results created by them but may want to see each others partial results. Hence, for collaboration purposes, we need a relaxed type of locks that may be released with respect to collaborators in the same team, while not released with respect to other users. Hence, although *Alice* acquires the locks, *Bob* can also acquire locks (even write locks) on these program modules. In particular, *Alice* may acquire read locks on *main* and *proc* and *Bob* acquires a write lock on *proc* concurrently. However, outside users may not acquire read locks as long as one of the two software developers holds a write lock.

This is the type of cooperation that we strive to achieve in this paper. Unfortunately, this simplistic adaptation of two phase locking may result in incorrect retrieval by outside scientists. Consider the following scenario: *Alice* read locks and reads *main* and *proc*; *Bob* read locks and reads both *main* and *proc* and then acquires a write lock on *proc* (*Alice* collaborates with *Bob*, hence this is allowed), writes a new version of *proc* and releases all its locks. Now a remote scientist tries to read *proc* and *main* by successfully acquiring read locks on *main* and *proc* (recall *Alice* only holds read locks on these program modules, and read locks can be held concurrently), and reads *proc* and *main* and releases the locks. Finally, *Alice* write locks and installs a new version of *main*. Unfortunately, the remote user retrieves partial results of an incomplete collaboration. Our goal in this paper is to develop a model for expressing collaborative relationships among various users and to develop protocols for managing such collaboration, without sacrificing efficiency and correctness.

4 The Model

In a *collaborative* database, a set of users access a common pool of data *objects*. The objects in the database can be accessed through atomic *read* and *write* operations. Users interact with the database by invoking *co-actions*. A co-action is a sequence of read and write operations that are executed on the objects. A read (write) operation executed by a co-action C_i on object x is denoted as $r_i[x]$ ($w_i[x]$). An *execution* S over $\mathbf{C} = \{C_1, \ldots, C_n\}$ is an interleaved sequence of all the operations of the co-actions in $\mathbf{C}$ such that the operations of co-actions C_i appear in the same order in S as they do in C_i, for $i = 1, \ldots, n$.

In the traditional database model, all interactions between users are through the read and write operations issued by transactions on the data objects. Since a transaction updates data objects piecemeal, each transaction must appear as a single atomic unit to all other transactions, otherwise, a transaction may observe the database in an inconsistent state. In a collaborative database, on the other hand, users may interact indirectly outside of the realm of the data, in addition to the traditional interactions through the data. Hence, the notion of *absolute atomicity*, which requires each transaction to appear as a *single* atomic unit to all other transactions, is too strict for a collaborative environment. In particular, users collaborating together in a design environment communicate with each other through *collaborative channels*. Unlike database operations that are modeled explicitly in an execution, interactions through collaborative channels cannot be explicitly modeled. Instead, interactions via collaborative channels break the boundaries of absolute atomicity leading to a new notion of atomicity specified in a pairwise manner, thus capturing the notion of a collaborative channel between any two co-actions. We term this notion as *relative atomicity* of co-actions.

Consider a set of co-actions $\mathbf{C} = \{C_1, C_2, C_3\}$ where

$$C_1 = r_1[main]r_1[proc]w_1[main]$$
$$C_2 = r_2[main]r_2[proc]w_2[proc]$$
$$C_3 = r_3[main]r_3[proc$$

$Atomicity(C_1, C_2)$: $\boxed{r_1[main]r_1[proc]}$ $\boxed{w_1[main]}$

$Atomicity(C_1, C_3)$: $\boxed{r_1[main]r_1[proc]w_1[main]}$

$Atomicity(C_2, C_1)$: $\boxed{r_2[main]r_2[proc]}$ $\boxed{w_2[main]}$

$Atomicity(C_2, C_3)$: $\boxed{r_2[main]r_2[proc]w_2[main]}$

$Atomicity(C_3, C_1)$: $\boxed{r_3[main]r_3[proc]}$

$Atomicity(C_3, C_2)$: $\boxed{r_3[main]r_3[proc]}$

Figure 1: Relative Atomicity Specifications

In the relative atomicity model, a co-action has multiply defined atomicity specifications with respect to every other co-action depending on the type of interactions that occur between the users outside the data objects through collaborative channels.. Formally, an *atomic unit of C_i relative to C_j* is a sequence of operations of C_i such that no operations of C_j are allowed to be executed within this sequence. However, C_i can establish a collaborative channel with respect to C_j and allow operations from C_j to execute both before and after this sequence. $Atomicity(C_i, C_j)$ denotes the ordered sequence of atomic units of C_i relative to C_j and $AtomicUnit(k, C_i, C_j)$ denotes the k^{th} atomic unit in $Atomicity(C_i, C_j)$.

Figure 1 illustrates the example of three co-actions considered informally in Section 3. Due to the collaborative channel between *Alice* and *Bob* the co-actions have the relative atomicity specifications as shown in the figure. In particular, the rectangular boxes represent the atomic units of each co-action with respect to other co-actions. In Figure 1, $Atomicity(C_1, C_2)$ is $\langle \boxed{r_1[main]r_1[proc]}, \boxed{w_1[main]} \rangle$, which indicates that if operations of C_2 have to be executed within C_1 then they may only be executed after $r_1[proc]$ and before $w_1[main]$. However, $Atomicity(C_1, C_3)$ is $\langle \boxed{r_1[main]r_1[proc]w_1[proc]} \rangle$. This indicates that co-action C_1 allows, through explicit co-ordination, only C_2 to execute operations after its $r_1[proc]$ operation, but allows C_3 to execute operations after C_1 has terminated. We say that an operation o of a co-action C_j is *interleaved with* an $AtomicUnit(k, C_i, C_j)$ in an execution S, if there exist operations o' and o'' of $AtomicUnit(k, C_i, C_j)$ such that o' precedes o and o precedes o'' in S. The relative atomicity specifications over $\mathbf{C}$ is the set $\{Atomicity(C_i, C_j) | C_i, C_j \in \mathbf{C}\}$.

From the relative atomicity specifications of co-actions in Figure 1, in any correct execution it is required that operations of C_2 not be interleaved within $AtomicUnit(k, C_1, C_2)$ for $k = 1, 2$. However, operations of C_2 may be executed between the two atomic units of C_1 relative to C_2. Consider, for example, the following execution:

$$S_{ra} = r_1[main]r_1[proc]r_2[main]r_2[proc]w_1[main]w_2[proc]r_3[main]r_3[proc]$$

Note that even though S_{ra} is not a serial execution, it is correct with respect to the relative atomicity specifications of the three co-actions in Figure 1. For example, operations of C_2 are executed between $r_1[proc]$ and $w_1[main]$. In spite of such interleavings, the atomicity of C_1 relative to C_2 is preserved. It can be easily shown that the same holds for the remaining interleavings. Consider another execution of the same set of co-actions:

$$S_1 = r_1[main]r_1[proc]r_2[main]r_2[proc]w_2[proc]r_3[main]r_3[proc]w_1[main]$$

The above execution should not be deemed correct since the execution violates the condition stipulated by $Atomicity(C_1, C_3)$. In particular, the read actions of co-action C_3 are interleaved within the atomic unit of C_1 relative to C_3. On the other hand, the following execution:

$$S_2 = r_2[main]r_2[proc]w_2[proc]r_3[main]r_3[proc]r_1[main]r_1[proc]w_1[main]$$

which provides the co-action C_3 the same syntactic state (*main* updated by co-action C_1) as in the previous execution will be correct. The key observation here is that the relative atomicity specifications model "collaboration" which is outside the realm of database itself. Thus in S_1, C_1 and C_2 execute in such a way that they set up the explicit collaboration between them and hence exclude C_3 until the collaborative task is completely done. In execution S_2, on the other hand, the two co-actions decide not to collaborate and hence C_3 is not prohibited from observing the state of data when only C_1 has incorporated its changes. Thus, by interleaving co-actions appropriately, we can accomplish the notion *participating domains* [Kai90].

Based on the relative atomicity specifications, the correctness of an execution S over $\mathbf{C}$ can be outlined as follows. An execution S will adhere to the specifications if for all co-actions C_i and C_l no operation of C_i is interleaved with an $AtomicUnit(k, C_l, C_i)$ for any k. In general, defining correct executions in these terms is too restrictive and does not allow enough concurrency among co-actions that do not interfere with each other (in the same way that serial executions are too restrictive in the traditional serializability model). To capture the various ways in which co-actions can interfere with each other, we need to introduce the notions of *conflict* and *dependency*. The notion of *conflict* between operations is used in the traditional concurrency control theory [Pap79, BSW79]. Two operations of different co-actions *conflict* if they access the same data object and at least one of them is a write operation. Two executions are *conflict equivalent* if they both order conflicting operations in the same manner. For collaborative databases, we need to define an additional notion called *depends on* which is derived from conflicts and the internal structure of co-actions. We say that o_2 *directly depends on* o_1 in S if o_1 precedes o_2 in S and either o_1 and o_2 are operations of the same co-action or o_1 conflicts with o_2. The *depends on* relation is the transitive closure of the directly depends on relation. The notion of dependency captures information flow that may occur between co-actions as a result of a sequence of conflicts. Ideally, this dependency should be from write operations to read operations, but for simplicity we use all conflict pairs. Thus even if two co-actions do not conflict with each other, one may influence the other due to the information flow resulting from the dependencies between them. For notational convenience, in the rest of the paper, we index co-actions and the operations within a co-action. For example, operation o_{ij} is the j^{th} operation of co-action C_i. We now define correct executions in our model that are referred to as *relatively serial* executions [ABEK94].

Definition 1 S is a *relatively serial* execution over $\mathbf{C}$ if for all co-actions C_i and C_l, if an operation o_{ij} of C_i is interleaved with an $AtomicUnit(k, C_l, C_i)$ for some k, then o_{ij} does not depend on any operation $o_{lm} \in AtomicUnit(k, C_l, C_i)$ and no operation $o_{ln} \in AtomicUnit(k, C_l, C_i)$ depends on O_{ij}.

In order to illustrate the notion of *depends on* we introduce an additional operation in co-action C_3 which performs a write operation on some other module, *DEM* (denoting digital elevation maps). Consider the following execution S_{rs} over $\mathbf{C}$ given in Figure 1:

$$S_{rs} = r_3[main]r_3[proc]r_1[main]r_2[main]w_3[DEM]r_1[proc]r_2[proc]w_2[proc]w_1[main]$$

The above schedule does not conform to the relative atomicity specified via the collaborative channels among the co-actions. In particular, an operation of C_2 is interleaved with $AtomicUnit(1, C_1, C_2)$. In spite of such interleaving, note that $r_2[main]$ does not depend on $r_1[main]$ and $r_1[proc]$ does not depend on $r_2[main]$. Similarly for the other interleavings: $w_3[DEM]$ interleaved with $AtomicUnit(1, C_1, C_3)$ and $AtomicUnit(1, C_2, C_3)$ but $w_3[DEM]$ does not have any depends on relations with any of the other operations. Hence S_{rs} is relatively serial. Thus, the notion of dependency is used in such a way that the relative atomicity specifications are enforced only when it is necessary, i.e., when co-actions affect each other.

The notion of relatively serial executions is analogous to the notion of serial executions in serializability theory [BSW79, Pap79]. In particular, there exist executions that are not relatively serial but may have the same behavior as some relatively serial executions. We define an execution to be *relatively serializable* [ABEK94] if it is conflict equivalent to some relatively serial execution. For example, the following execution of co-actions over $\mathbf{C}$ given in Figure 1:

$$S_{rsr} = r_1[main]r_1[proc]r_3[main]r_3[proc]w_3[DEM]w_1[main]$$

is not relatively serial since $r_1[main]$ is interleaved with $AtomicUnit(1, C_1, C_3)$ and $w_1[main]$ depends on $r_3[main]$. However, S_{rsr} is relatively serializable since it is conflict equivalent to the following relatively serial execution:

$$r_3[main]r_1[main]r_1[proc]r_3[proc]w_3[DEM]w_1[main]$$

The relative atomicity model is a generalization of the traditional absolute atomicity model. That is, if the specifications are such that each co-action is a single atomic unit with respect to all other co-actions, then relative atomicity reduces to absolute atomicity [ABEK94]. In [ABEK94] we also introduce a graph that can be used to determine whether a given execution is relatively serializable. This graph is therefore a powerful tool for determining the correctness of protocols that manage co-actions.

5 A Locking Protocol

In this section, we present a new locking protocol, which guarantees the relatively serializable execution of co-actions. The protocol utilizes the specifications resulting from collaborative channels among co-actions to enforce correct executions in the system. Such executions would be non-serializable under the traditional theory but are still acceptable to users. We start with a general overview of the protocol and then we describe the protocol in greater detail.

5.1 Overview of the Protocol

One of our main design criteria is to develop a protocol that has similar characteristics to the strict two-phase locking protocol [EGLT76]. Ideally, we would like to design the protocol such that when the specifications of co-actions are restricted to absolute atomicity, the proposed locking protocol reduces to the standard strict two-phase locking protocol.

The strict two-phase locking protocol uses shared and exclusive mode locks to synchronize read and write operations on an object. A co-action is granted an exclusive lock on an object only if no other co-action holds a lock on that object and is granted a shared lock on an object only if all other locks on that object are shared. All the locks acquired by a co-action are released at the end of the co-action. It can be easily shown that the strict two-phase locking protocol maintains each co-action as a single atomic unit with respect to other co-actions in the system [Pap79, BSW79].

Several proposals have been made to adapt strict two-phase locking for co-actions to eliminate the restriction that locks must be held until co-actions terminate [PKH88, Kai90]. In particular, based on the relative atomicity specifications of a co-action, the co-action should now be allowed to release its locks to those co-actions that are allowed at the end of its atomic units. As before shared locks can be held concurrently by several co-actions. Unfortunately, in the context of relative atomicity specifications, the following example illustrates that the approach which seems intuitively appealing is not correct. Consider the execution presented in Section 4.

$$S_1 = r_1[main]r_1[proc]r_2[main]r_2[proc]w_2[proc]r_3[main]r_3[proc]w_1[main]$$

S_1 is accepted since C_2 is allowed to collaborate with C_1 and, since $r_3[main]$ does not conflict with $r_1[main]$ the shared lock for $r_3[main]$ is granted even though C_3 is not allowed in the atomic unit of C_1 with respect to C_3. However, as was argued in the previous section, S_1 is not equivalent to any relatively serial execution, since the dependencies from $w_2[proc]$ to $r_3[proc]$ and from $r_3[main]$ to $w_1[main]$ restrain C_3 from being moved outside the atomic units of C_1 and C_2. Hence we need to add more restrictions to develop a correct locking protocol.

Instead of generalizing strict two-phase locking, we start by developing a simple protocol that would accept all relatively serial executions. From the definition of relatively serial executions, it can be seen that when an operation o is interleaved within an atomic unit of a co-action C, the following two conditions must hold:

Consider a set of co-actions $\mathbf{C}_2 = \{C_1, C_2, C_3\}$ where

$$C_1 = r_1[x]w_1[y]w_1[z]$$
$$C_2 = r_2[y]w_2[z]$$
$$C_3 = w_3[x]$$

$Atomicity(C_1, C_2)$: $\boxed{r_1[x]w_1[y]}$ $\boxed{w_1[z]}$ $Atomicity(C_2, C_1)$: $\boxed{r_2[y]}$ $\boxed{w_2[z]}$

$Atomicity(C_1, C_3)$: $\boxed{r_1[x]w_1[y]}$ $\boxed{w_1[z]}$ $Atomicity(C_2, C_3)$: $\boxed{r_2[y]}$ $\boxed{w_2[z]}$

Figure 2: Example Set of co-actions $\mathbf{C}_2$

1. Operation o does not depend on any operation of that atomic unit.

2. No operation of that atomic unit depends on o.

Suppose that an operation o' of C has been executed. In order to ensure the first condition, the simple protocol must delay the execution of o if it results in a dependency from o' to o, until after the atomic unit in which it is interleaved (if any) has "completed", i.e., each operation of the atomic unit has been executed. The following example illustrates the actions taken by the simple protocol to ensure that the first condition holds. Consider the execution

$$S_3 = r_1[x]w_3[x]w_1[y]r_2[y]w_2[z]w_1[z]$$

on the set of co-actions $\mathbf{C}_2$ given in Figure 2. The simple protocol must delay $w_3[x]$ since it is interleaved with the atomic unit $\boxed{r_1[x]w_1[y]}$ of C_1 and there is a dependency from $r_1[x]$ to $w_3[x]$. In other words, C_3 is not granted a lock for $w_3[x]$ until after the atomic unit has completed.

The effect of the delay of execution of an operation o is to "push o forward" past the end of the atomic unit in which it causes dependencies. We say that a co-action C_i can obtain a *pushforward lock* for o_{ij} with respect to a dependency from o_{kl} if the last operation of the atomic unit of C_k, relative to C_i which contains operation o_{kl}, has been executed. We refer to this last operation as $PushForward(o_{kl}, C_i)$ Hence, a co-action C_i can obtain a pushforward lock for o_{ij} with respect to a dependency from o_{kl} if $PushForward(o_{kl}, C_i)$ has been executed. Thus we have the following locking rule.

- Locking Rule. If a pushforward lock for o_{ij} with respect to o_{kl} can be acquired by C_i for each operation o_{kl} such that o_{ij} depends on o_{kl}, $i \neq k$, then C_i is granted a lock for o_{ij}. An operation o is executed if a lock can be granted according to the above locking rule. Otherwise the operation o is delayed.

When an operation o is interleaved with an atomic unit, and later an operation o' of that atomic unit arrives, which depends on o, then the execution of o' will result in a violation of the second condition of relatively serial executions. In this case, the simple protocol must either abort all the operations of the atomic unit which have executed prior to the execution of o or it should abort o. We choose to abort o. Note that delaying operation o' will not result in the second condition being satisfied. This scenario is illustrated in the following example. Consider the execution

$$S_4 = r_1[main]r_1[proc]r_3[main]r_3[proc]w_1[main]$$

on the set of co-actions **C** given in Figure 1. In S_4, $r_3[main]$ is allowed to execute after $r_1[main]r_1[proc]$ since there is no dependency to $r_3[main]$ from either of them, even though C_3 is not allowed within the atomic unit of C_1. The protocol aborts $r_3[main]$ since C_3 is not allowed in the atomic unit of C_1 and there is a dependency from $r_3[main]$ to $w_1[main]$. However, the execution S_4 is relatively serializable and equivalent to C_3 being executed before C_1. From the collaboration point of view, also, this execution is intuitively correct. In fact, if strict two phase locking was used to synchronize C_1 and C_3 without using any collaborative information, the above execution would have been accepted. Hence, the simple protocol is too restrictive and we next explore how to relax this protocol.

5.2 Specifications of the Protocol

An appropriate locking protocol should at least perform as well as the standard concurrency control protocol, strict two-phase locking. The simple protocol discussed above even after using the collaborative information fails to accept some executions that strict two-phase locking would have accepted without taking into account the collaborative information. Note, however, that strict two-phase locking is not appropriate since it prohibits collaborating completely and its relaxation results in incorrect executions. We can rectify this shortcoming of the simple protocol as follows. Instead of aborting $r_3[main]$ in S_4 dependencies of the form $r_3[main]$ to $w_1[main]$ can be allowed, since S_4 is equivalent to a relatively serial execution in which $r_3[main]$ executes before C_1. Thus, optimistically, we can allow dependencies from an operation o to operations within an atomic unit, and validate later that the operation o can be "pulled backward" before the start of the atomic unit. An operation o interleaved within an atomic unit can be *pulled backward* if the effect of executing o before the start of the atomic unit is the same as that when the execution of o is interleaved. The effect of executing o before the start of the atomic unit (pulling backwards) should not affect the operations that have already been executed. Hence the basic idea of the protocol can be summarized as follows. Unlike the strict two-phase locking rule, a co-action waiting to execute a conflicting operation needs to obtain "pushforward" locks with respect to the atomic units of co-actions with which it conflicts. If a co-action executes an operation within an atomic unit of another co-action and if later conflicts are detected, the co-action must obtain a "pullbackward" lock with respect to that atomic unit. Otherwise, the co-action is aborted.

We now present the following rules that must be observed by a collaborative locking protocol to accept relatively serializable executions.

- Locking Rule 1. A co-action C_i can acquire a $pushforward$ lock for $o_{ij}[x]$ if for each operation $o_{kl}[y]$ such that $o_{ij}[x]$ depends on $o_{kl}[y]$ $(i \neq k)$, $PushForward(o_{kl}, C_i)$ has been executed.

- Locking Rule 2. A co-action C_i can acquire a *pullbackward* lock for $o_{ij}[x]$ if for each operation $o_{kl}[y]$ such that $o_{ij}[x]$ depends on $o_{kl}[y]$ $(i \neq k)$, o_{kl} can be pulled backward before $PullBackward(o_{ij}, C_k)$, which is the first operation of the atomic unit of C_k relative to C_i in which o_{ij} is interleaved.

- Locking Rule 3. A co-action C_i can acquire a lock on x for $o_{ij}[x]$ if it can obtain a $pushforward$ lock and a *pullbackward* lock for $o_{ij}[x]$.

We illustrate the mechanism using the following scenarios. Consider an operation, o_{kl} interleaved in an atomic unit $\boxed{o_{in} \ldots o_{ij}}$ of C_i. If o_{kl} depends on o_{in} as shown in Figure 3 then the protocol as per *Locking Rule 1* must delay the execution of o_{kl} until the end of the atomic unit of C_i (relative to C_k) that contains o_{in}, i.e., until after o_{ij} has been executed.

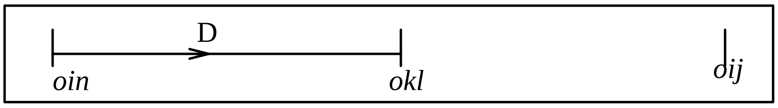

Figure 3: Dependency from o_{in} to o_{kl}

In the above scenario, if o_{kl} does not depend on o_{in}, then o_{kl} can be executed immediately even though o_{kl} is not allowed in the atomic unit $\boxed{o_{in} \ldots o_{ij}}$ as per the specifications. A later dependency that may arise from o_{kl} to o_{ij} (as shown in Figure 4) can cause the executions to be relatively non-serializable if o_{kl} is constrained to be executed after o_{in}. The constraint is not a dependency from o_{in} to o_{kl} but is an execution such that o_{kl} cannot be pulled out before o_{in}.

We illustrate the notion of such constraints with the help of an example illustrated in Figure 5. In the execution S shown in Figure 5, $w_{in}[x]$, $r_{kl}[y]$ and $w_{ij}[y]$ correspond to o_{in}, o_{kl} and o_{ij} respectively. It can be verified that in S, $r_{kl}[y]$ is constrained to be after $w_{in}[x]$ due to the dependency from

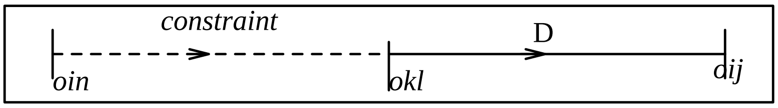

Figure 4: Dependency from o_{kl} to o_{ij}

$w_{in}[x]$ to $r_{ps}[x]$ and the pushforward dependency from $r_{ps}[x]$ to $r_{kl}[y]$, even though $r_{kl}[y]$ does not depend on $w_{in}[x]$. When the above dependency from o_{kl} to o_{ij} arises, the protocol will not be able to obtain a pullbackward lock for o_{ij} since o_{kl} is constrained to be after o_{in}. Note that delaying the execution of o_{ij} will not alter the situation. The protocol chooses to abort o_{kl} so that C_i is not blocked indefinitely. The protocol should also abort all operations that are affected by o_{kl}. Clearly, if this causes any aborts of C_i's operations, then o_{ij} should not be executed and the pullbackward lock for o_{ij} is not granted to C_i.

Consider $S = w_{in}[x]w_{pq}[y]r_{ps}[x]r_{kl}[y]\ldots w_{ij}[y]$
where
$\boxed{w_{pq}[y]r_{ps}[x]}$ is an atomic unit of C_p w.r.t. C_k.

$\boxed{w_{in}[x]w_{ij}[y]}$ is an atomic unit of C_i w.r.t. C_k.

Figure 5: Example for constraint and dependency within an atomic unit

Finally consider the scenario of Figure 6. Since there are no dependencies between o_{kl} and any of C_i's operations, the locking rules are satisfied and a execution corresponding to the above scenario can be generated by the protocol. It is easy to see that this execution is relatively serializable.

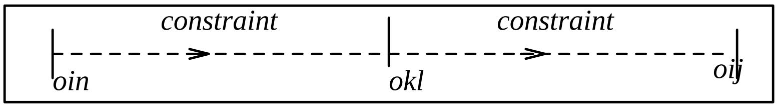

Figure 6: No dependencies from o_{kl} or to o_{kl}

The proposed locking protocol accepts all relatively serial executions since locking rules 1, 2, and hence 3 are satisfied for all operations of a relatively serial execution. This follows due to the fact that there are no dependencies between interleaved operations within an atomic unit. In addition, under absolute atomicity of co-actions, the protocol reduces to the strict two-phase locking protocol. Consider any execution that is produced by the strict two phase locking protocol. *Locking* *rule* 1 of the proposed protocol is satisfied since conflicting operations (hence dependencies) are executed only after the end of the atomic unit of a co-action, which is the entire co-action itself. In addition, *locking rule* 2 is satisfied since interleaved operations cannot be constrained. This is due to the fact that each constraint is associated with at least one conflict, and therefore the entire co-action has to be completed before the conflict can arise. This is because in a strict two-phased locked execution locks are released only after the completion of a co-action. Thus the interleaved operations can be pulled backward satisfying *locking rule* 2. Therefore any strict two-phase locking execution is accepted by the protocol. Consider any execution S generated by the protocol under absolute atomicity. Since S satisfies *locking rule* 1, conflicting operations are moved past the end of a co-action in S. Hence, the strict two-phase locking protocol can also grant the locks on conflicting operations as needed by S. Since S satisfies *locking rule* 2, interleaved operations are not constrained and thus the strict two-phase locking protocol can grant the locks for the interleaved operations. Therefore when relative atomicity specifications are not available the protocol we have developed performs like the strict two-phase locking protocol.

5.3 Implementation of the Protocol

The main effort in implementing this protocol entails determining dependencies and constraints between operations. Collecting dependency information is relatively simple because it is basically a transitive closure of conflict and program order of read and write operations of co-actions. Constraints, on the other hand, require careful consideration. While conflicts and pushforward dependencies between two operations can be determined at the time of the execution of the second operation, the same is not true for pullbackward dependencies. Pullbackward dependencies can arise later in the execution (long after both the operations have been executed). It is therefore complicated to determine the constraint from o_{in} to o_{kl} (see Figure 4). To simplify the protocol, we use the following approach to determine the possibility of a constraint from o_{in} to o_{kl}. We consider that there is a possibility of a constraint from o_{in} to o_{kl} in an execution if there is a dependency or a pushforward or pullbackward dependency from o_{in} or from any operations of T_i that follow o_{in}.

In order to implement the proposed protocol, we need to maintain the following two main data structures to compute dependencies and constraints between operations of co-actions.

- $LockQueue(x)$ is the queue of operations of active co-actions that have obtained locks on the data item x, along with their dependencies. Each entry in the queue is of type $LockEntry$ given in Figure 7. The operation *op* depends on each of the operations in *dep*. Associated with each entry are two sets of operations – $pfset$ and *pbset* which contain the pushforward and pullbackward dependencies of *op* which arise from pushforward and pullbackward locks.

```
TYPE
    LockEntry : RECORD
        op : operation;
        dep : set of operation;
        pfset : set of operation;
        pbset : set of operation;
    end;

    OpEntry : RECORD
        op : operation;
        pfl : set of operation;
        pbl : set of (operation, operation);
        pbd : set of operation;
    end;
```

Figure 7: Basic Data Structures for Lock and Co-action Management

An operation o is in $pfset$ ($pbset$) if there is a pushforward (pullbackward) dependency from op to o. There is one $LockQueue$ for each data item.

- A co-action maintains an $OpEntry$ record (as given in Figure 7) for each of its operations. For each of its operation $o_{ij}[x]$, the co-action C_i keeps track of the lock obtained on x. C_i keeps track of the pushforward locks of o_{ij} in pfl, which is a set of operations o_{km} such that there exists an $o_{kl}[y]$ and $o_{ij}[x]$ depends on $o_{kl}[y]$ and $PushForward(o_{kl}, C_i) = o_{km}$. In addition, C_i keeps track of pullbackward locks of o_{ij} in pbl which is a set of pairs. If $o_{ij}[x]$ depends on $o_{kl}[y]$ and $PullBackward(o_{ij}, C_k) = o_{in}$ then pbl contains the pair $(o_{kl}[y], o_{in}[z])$. In addition, C_i keeps track of the pullbackward dependencies to $o_{ij}[x]$ in the set pbd of $o_{ij}[x]$.

The above two data-structures correspond to the state information associated with the data objects and with the co-actions.

A high-level description of the algorithm for executing operations is illustrated in Figure 8. On receiving an operation $o_{ij}[x]$ the scheduler executing the protocol determines if the lock on x for $o_{ij}[x]$ can be granted to C_i. This is done by considering all the dependencies to the operation $o_{ij}[x]$. The dependencies to o_{ij-1} (if any), and the dependencies to those operations that directly conflict with and precede $o_{ij}[x]$, capture all the dependencies to $o_{ij}[x]$. The dependency information for these conflicting operations is obtained from $LockQueue(x)$. In accordance with the *Locking Rules*, the lock on x for $o_{ij}[x]$ is granted if a pushforward lock and a pullbackward lock can be obtained for $o_{ij}[x]$. A function, $AcquirePfLock$, returns true if for each dependency from $o_{kl}[y]$ to $o_{ij}[x]$, the last operation of the atomic unit of C_k (relative to C_i) containing $o_{kl}[y]$ has been

```
BEGIN Protocol

DATA
        Dep : set of operation;

    (* Determine the dependencies to o_ij[x] *)
    Dep = ComputeDependency(o_ij[x]);

    (* Determine if pushforward and pullbackward locks for o_ij[x] can be acquired by C_i *)
    IF Dep ≠ φ THEN
        IF (not(AcquirePfLock(o_ij[x], Dep))) THEN
            delay(o_ij[x]);
            exit;
        ELSE
            IF (not(AcquirePbLock(o_ij[x], Dep))) THEN
                delay(o_ij[x]);
                exit;
            END; (* IF *)
        END; (* IF *)
    END; (* IF *)

    (* A lock on x for o_ij[x] can be granted since locking rules 1 and 2 are satisfied *)
    (* The set Dep reflects those dependencies to o_ij[x] which can be pulled backward *)

    (* Execute the operation and record information about the locks obtained
        and dependencies created *)

    Execute o_ij[x];

END (* Protocol *)
```

Figure 8: A high-level description of the Locking Protocol

executed. Another function, *AcquirePbLock*, returns true if a pullbackward lock can be obtained by C_i for $o_{ij}[x]$. First, this function checks to see if each operation $o_{kl}[y]$ on which $o_{ij}[x]$ depends on can be pulled backward before the start of the atomic unit of C_i (relative to C_k) containing $o_{ij}[x]$. As described earlier, $o_{kl}[y]$ can be pulled backward before $PullBackward(o_{ij}[x], C_k)$ (say $o_{im}[t]$) if $o_{kl}[y]$ is not constrained to be after $o_{im}[t]$. As explained earlier, this is ascertained by checking that there are no dependencies (except internal dependencies), and no pushforward and pullbackward dependencies from any operation of C_i between $o_{im}[t]$ and $o_{ij-1}[.]$. It keeps track of all the operations that cannot be pulled backward. If both the pushforward and pullbackward locks for $o_{ij}[x]$ are granted to C_i then the scheduler also grants the lock on x for $o_{ij}[x]$ to C_i. If the lock on x is acquired by C_i, then $o_{ij}[x]$ is executed and is appended to $LockQueue(x)$. The dependencies to $o_{ij}[x]$ and pushforward and pullbackward dependencies are maintained in the data-structure *OpEntry*. The detailed implementation of this protocol appears in [Kri93]. In particular, details regarding the computation of dependencies, termination, and garbage-collection of co-actions are presented. Furthermore, a formal proof of correctness is also presented in [Kri93].

6 Conclusion

In this paper, we present an approach to support collaborative activities that occur in software engineering databases, design databases, and scientific databases. Our approach is novel since it incorporates collaboration into the database management system itself. Furthermore, our work is based on two fairly diverse bodies of research: collaborative tools and environments, and semantics based transaction management. In particular, the former aims to extend the traditional transaction model to weaken the isolation property and to support long-lived activities. The latter, on the other hand, is motivated to find alternate correctness criteria when semantics of transactions and database are available. We extend the latter approach and instead of using semantics, rely on users to provide collaborative information which is used to determine relative atomic units of collaborative transactions. With the relative atomicity model, we do not restrict collaborations to particular domains or teams, rather interactions may be dynamically decided during the course of the execution. After adapting the formal model to collaborative databases we present a locking protocol for supporting correct executions of collaborative transactions. An important aspect of this protocol is that in the absence of collaboration, i.e., in the traditional database model, the protocol reduces to the standard strict two-phase locking protocol widely used in databases for transaction management.

References

[ABEK94] D. Agrawal, J. L. Bruno, A. El Abbadi, and V. Krishnaswamy. Relative Serializability: An Approach for Relaxing the Atomicity of Transactions. In *Proceedings of the ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, May 1994.

[BK91] N. S. Barghouti and G. E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–318, September 1991.

[BKK85] F. Bancilhon, W. Kim, and H. F. Korth. A model of cad transactions. In *Proceedings of the Conference on Very Large Data Bases*, pages 25–33, Stockholm, Sweden, August 1985.

[BSW79] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, 5(5):203–216, May 1979.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[FÖ89] A. A. Farrag and M. T. Özsu. Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

[Gar83] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GHMS94] K. Gronbaek, J. A. Hem, O. L. Madsen, and L. Sloth. Cooperative Hypermedia Systems: A Dexter-Based Architecture. *Communications of the ACM*, 37(2):64–74, February 1994.

[GKM93] K. Gronbaek, M. Kyng, and P. Mogensen. CSCW challenges: Cooperative Design in Engineering Projects. *Communications of the ACM*, 36(6):67–77, June 1993.

[GM93] N. D. Gherson and C. G. Miller. Dealing with the Data Deluge. *IEEE Spectrum*, 30(7):28–32, July 1993.

[Kai90] G. E. Kaiser. A Flexible Transaction Model for Software Engineering. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 560–567, 1990.

[Kat90] R. Katz. Towards a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408, December 1990.

[KB92] V. Krishnaswamy and J. Bruno. On the Complexity of Concurrency Control using Semantic Information. Technical Report TRCS 92-21, Department of Computer Science, University of California, Santa Barbara, CA 93106, 1992.

[Kri93] V. Krishnaswamy. *Semantics Based Synchronization in Database Systems*. PhD thesis, Department of Computer Science, University of California, Santa Barbara, May 1993.

[Lyn83] N. A. Lynch. Multilevel Atomicity – A New Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems*, 8(4):485–502, December 1983.

[Pap79] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[PKH88] C. Pu, G. Kaiser, and N. Hutchinson. Split-Transactions for Open-Ended Activities. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pages 26–37, August 1988.

[RRD90] M. A. Ranft, S. Rehm, and K. R. Dittrich. How to Share Work on Shared Objects and Design Databases. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 575–583, 1990.

[Sor87] P. Sorgaard. A Cooperative Work Perspective on Use and Development of Computer Artifacts. In *Proceedings of the Tenth Information Systems Research Seminar in Scandinavia*, pages 10–12, August 1987.

[SSV92] D. Shasha, E. Simon, and P. Valduriez. Simple Rational Guidance for Chopping Up Transactions. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 298–307, June 1992.

[SV91] S. Sarkar and V. Venugopal. Transaction Mechanism for Software Environment Databases. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 511–518, 1991.

[SZ89] A. H. Skarra and S. B. Zdonik. Concurrency Control and Object-Oriented Databases. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 16, pages 395–421. ACM Press, 1989.

[VS91] V. Venugopal and S. Sarkar. Requirement of Object-Oriented Databases for Tool Integration in IPSEs. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 543–552, 1991.

[WBMN88] J. Walpole, G. S. Blair, J. Malik, and J. R. Nicol. A Unifying Model for Consistent Distributed Software Development Environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 183–190, 1988.