A Concurrency Control Framework for Collaborative Systems

Jonathan Munson and Prasun Dewan
Department of Computer Science
C.B. 3175, Sitterson Hall
University of North Carolina-Chapel Hill
Chapel Hill, NC 27599-3175
{munson, dewan}@cs.unc.edu

ABSTRACT
We have developed a new framework for supporting concurrency control in collaborative applications. It supports multiple degrees of consistency and allows users to choose concurrency control policies based on the objects they are manipulating, the tasks they are performing, and the coupling and merge policies they are using. Concurrency control policies are embodied in hierarchical, constructor-based lock compatibility tables. Entries in these tables may be specified explicitly or derived automatically from coupling and merge policies. In this paper, we motivate and describe the framework, identify several useful concurrency control policies it can support, evaluate its flexibility, and give conclusions and directions for future work.

Keywords
Concurrency control, collaborative systems, consistency criteria, coupling, merging, transactions

INTRODUCTION

When two or more users collaborate through jointly manipulating a shared object—be it an electronic whiteboard, a document, or a database—there is a need for them to synchronize their actions. That is, there may arise occasions where one user's action needs to be blocked so that it does not interfere with another user's action. For example, if two users are modifying a bibliography citation, one user's modification may need to be blocked until the other's completes. A concurrency control system, if one exists, would be expected to provide this synchronization.

Concurrency control has been studied extensively in the context of database systems. But traditional database concurrency control is generally too restrictive for collaborative applications, for several reasons. First, the semantics of the shared data object is usually much more complex than the read/write semantics of the database data model. As a result, the concurrency control it supports is more conservative than necessary. For instance, users may interpret the bibliography above as basically a set of citations to which insertions may be made concurrently without conflict. A database using read/write semantics will not allow concurrent insertions but will block one to let the other proceed.

Computer Supported Cooperative Work '96 , Cambridge MA USA
© 1996 ACM 0-89791-765-0/96/11 ..$3.50

Second, traditional database systems do not allow concurrent transactions to mutually depend on each other, whereas in collaborative systems users whose workspaces are coupled may be expected to influence each other in mid-transaction. For instance, two coupled users may be responsible for jointly adding a list of citations to a bibliography. Before adding a new item, a user would observe the actions of the other user to ensure that duplicate items are not inserted in the database.

Third, in collaborative systems users may wish to temporarily allow conflicting actions and delay their resolution until some later time. For instance, two users may independently add citations to a database, leaving removal of duplicates to a later stage of their work. Conventional database systems, however, do not allow the database to remain in an inconsistent state for indefinite periods.

Last, the action that a conventional database system will take when conflict is identified is to throw away all work that led to the conflict and return the database to a prior consistent state. For a user about to commit a large number of changes to a document, this would not only be very unpleasant, but is probably unnecessary; the user may only need to discard changes to one paragraph.

Concurrency control has also been addressed in collaborative systems. Early collaborative applications (e.g., [23]) used floor control (one person acting at a time) to prevent users from colliding. This was found to be too restrictive for many applications, and later systems dispensed with synchronization altogether [6, 13], or provided visual feedback to indicate which objects were "in use" [25]. These systems rely upon social protocols and users' awareness of others' actions to prevent conflicts, and hope that if conflicts do occur they can be quickly and easily resolved. But this approach may also not be acceptable in many situations. First, the need for awareness and communication may overly burden users. As transactions grow in duration or the object grows in size the problem becomes worse. Second, by not preventing conflicts or resolving them when they occur, such systems force users to resolve conflicts when they are detected. This may be undesirable because of the possibility of losing work. In short, users may prefer 'prevention' to 'cure.' Third, the quality of awareness the application can provide is highly sensitive to the delay in delivering one user's actions to other users. Thus applications that rely on awareness to prevent conflicts may not function well in environments with high-latency communications systems.

Above are two extremes of concurrency control policy: too-restrictive database policy or floor control, and no policy at all. We are concerned with how to provide concurrency control policies of intermediate restrictiveness. There has recently been significant work on concurrency control for collaborative systems, but it has primarily addressed performance. Flexible concurrency control has been addressed in advanced database systems, but in the context of programmed transactions. In this paper we present a general concurrency control framework for collaborative applications that supplies a richer data model than do database systems, provides flexible user-set policies, and defines mechanisms that automatically derive concurrency control from coupling and merge policies. In the sections that follow, we first discuss related research; we then present a set of new concurrency control requirements for collaborative applications; next we describe and evaluate our framework; and finally we present our conclusions and some thoughts for future work.

RELATED RESEARCH

Any particular concurrency control system makes assumptions regarding two basic properties of the application: the semantics of the object being shared and the consistency requirements of the application. As an example, for conventional database systems, the semantics assumed is that of storage with read and write access, and the consistency requirement is isolation of concurrent transactions. Fidelity of these assumptions with the actual semantics of the object and the expectations of the users with respect to the restrictions imposed for the sake of consistency are two important (informal) measures of quality for a concurrency control system. In this section we discuss related work in terms of these measures. We first discuss traditional database systems, then we describe some work in advanced database systems that targets increasing concurrency, and finally we discuss some work specifically focused on collaborative applications.

Traditional Database Systems

Concurrency control has been most actively studied within the domain of database systems, where its goal is usually to provide *serializable transactions* [7]. A transaction is a sequence of operations that together perform a logical task; serializable transactions are concurrent transactions whose operations are interleaved in such a way that there exists an equivalent serial (one after the other) execution of the transactions. If each transaction is correct, then a serial execution of them must be correct, and so a serializable execution of them must also be correct. The serializability analysis is based on read/write semantics, i.e., all data objects are simple data stores.

Serializability is an appropriate consistency model for database systems because users expect their transactions not to be interfered with by any other transactions. And, because their transactions are usually fixed programs, they could not respond to interference if it did happen. But users of collaborative systems, which are typically interactive, can respond to interference. Also, what constitutes interference in a database system may in a collaborative system be entirely proper, e.g., one user sharing with another work that is not yet complete. Thus serializability is for collaborative applications often too restrictive a consistency model.

Exploiting Type Semantics

The restrictive nature of conventional database concurrency control is also due to its reliance on primitive read/write semantics. A number of researchers have demonstrated that basing concurrency control on the semantics of the interface an object presents—its type—rather than the reads and writes of the object's constituent members, results in less restrictive scheduling [19, 24, 26]. Concurrency control based on read/write semantics must necessarily be conservative because there is no awareness of what operations in a transaction actually depend on each other. It must be assumed, for example, that a write following a read depends on that read. Consideration of the object semantics may reveal that there is no dependency, but this information is not available to the concurrency control system.

Korth [19] and Schwarz and Spector [24] developed lock compatibility tables based on this idea, using a data type's operations as lock modes rather than, or in addition to, simple read and write operations. They demonstrated that greater concurrency resulted, still using serializability as the consistency criterion. Schwarz and Spector also suggest that at times other, weaker, consistency criteria can be used, and provide an example of a "weakly FIFO queue" that allows two users to interleave their enqueue and dequeue operations if the operations are independent, that is, if one user does not dequeue an object enqueued by another user in a concurrent transaction. Such a queue preserves the property of 'fairness' rather than serializability.

Being lock-based, the mechanisms above are pessimistic concurrency control techniques. Herlihy [16] uses similar type information in an optimistic scheme, in which transactions run without synchronization and are checked for consistency when they attempt to commit.

Exploiting Application Semantics

An application will typically have semantics beyond those coded in the data types it employs. For instance, an application may need an object with set semantics but have available to it only a sequence type; it will therefore use the sequence in set fashion. If the concurrency control system is made aware of this information, significantly less restrictive control may result. This approach is of particular interest to the database community because it layers on top of conventional read/write operations.

Garcia-Molina [9], Lynch [20], and Farrag and Ozsu [8] described transaction models in which transaction programmers explicitly declared how a particular kind of transaction may be interleaved with other transactions in ways that would have otherwise been disallowed.

Sagas, as defined by Garcia-Molina in [10], are sequences of subtransactions that can be interleaved arbitrarily with other transactions. Thus sagas can view partial results of other sagas. To avoid the problem of cascading aborts that sagas would otherwise be prone to, each subtransaction has a *compensation function* that will undo its effects according to application semantics. In case a saga aborts, its subtransactions' compensation functions are invoked rather than a rollback to a previous database state. This increases concurrency by allowing schedules that otherwise would have been disallowed to avoid cascading aborts.

Other researchers have presented work based on formal notions of program correctness, in which application semantics is encoded in predicates that must be satisfied by transactions. Bancilhon et al. [1] require each transaction to provide an invariant of the database that its execution maintains; Korth and Speegle [18] require each transaction to supply precondition and postcondition predicates. Since satisfaction of arbitrary predicates would be infeasible, the authors provide for an efficiently executed subset.

As will be seen when we describe our own work, we have borrowed much from the large body of database concurrency control research. But our work diverges from the database research because of its focus on a different class of applications. Collaborative applications are typically interactive and often designed to support unstructured activities, with dynamic and context-specific consistency requirements. Thus concurrency control systems that require predicates or compensating functions programmed at application-definition time could not well-support these applications. Furthermore, mechanisms intended for database applications are not designed to bring to bear, in addition to the machine's capabilities, the more sophisticated, albeit less reliable, consistency-checking powers of human beings, that are available in interactive applications. Our work identifies high-level concurrency control mechanisms for supporting concurrency control that recognizes that humans may cooperate in maintaining consistency.

Multiple Granularity Locking

Data objects are often hierarchically structured. A technical manual may be composed of chapters, which are composed of sections, which are composed of subsections. A software project may be composed of modules, which are composed of code and documentation. Given such a structure, it may not be clear what the best unit, or granularity, of locking is: i.e., for a document, should users lock chapters, or sections, or subsections? Large-grained locks are convenient when users need to make many changes, but small-grained locks allow more concurrent activity. Gray et al. addressed this problem with *multiple granularity locking* [11, 12]. Document locking under this scheme would allow a user to lock either a chapter, a section, or a subsection. In addition to conventional read and write locks (Share and Exclusive in their terminology), the authors defined two additional locks, Intent Share and Intent Exclusive. An Intent *mode* lock on a structure indicates that a user intends to place a *mode* lock on some substructure. When attempting to lock a structure in *mode* mode, a transaction attempts to place an Intent *mode* lock on each parent structure, beginning with the root. The lock compatibility table of Table 1 determines, at each level of the hierarchical structure, an attempt's success or failure. In case of failure a transaction removes all Intent locks from parent nodes.

Lock	Lock held				
request	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

Table 1. Lock compatibility table for multiple granularity locking

Table 1 shows an SIX lock, which stands for Share Intent Exclusive. It is essentially a Share lock that allows the holder to set an Exclusive lock on a substructure.

To illustrate multiple granularity locking, consider users A and B jointly editing a document, where A has an Exclusive lock on the first section of the first chapter. Suppose B wants to lock the second section of the first chapter. B attempts to place an IX lock on the first chapter, which succeeds because it is compatible with A's IX lock. B's attempt to lock Section 2 in Exclusive mode succeeds as well. But suppose B had decided to lock the entire chapter in Exclusive mode. This would fail because an X lock is not compatible with an IX lock.

Concurrency Control For Collaborative Applications

We discuss here some recent efforts that address the need for concurrency control in collaborative applications. DistView [22] is a collaborative applications framework employing a replicated architecture. Concurrency control is provided via locking. For performance reasons lock tables are replicated so that all accesses are local.

GroupKit [14] is an applications toolkit that also employs locking. In DistView, user actions are blocked between the time a lock is requested and the time it is granted; in GroupKit, however, user actions are allowed to continue while waiting for the lock to be granted. If the lock is refused the actions are undone automatically.

A non-locking, non-blocking approach is offered by Ellis and Gibbs [5, 6] through their dOPT algorithm for replicated architectures. The algorithm is designed to provide consistent displays in the face of out-of-order message delivery. The dOPT algorithm relies on operation transformation rather than blocking and so provides excellent response time, but it is not a synchronization mechanism.

The efforts described above are primarily concerned with maintaining a desirable property of systems without concurrency control, namely good response time. They do not, however, address the need for user-defined consistency in collaborative systems.

REQUIREMENTS

Transactions in collaborative applications have several characteristics that, taken together, distinguish them from transactions in other applications requiring concurrency control. Being typically interactive, they may be of long duration; they are often unstructured and unplanned; they are more sensitive to response-time performance than total system throughput. Because they are collaborative applications, concurrency control's role of preventing inconsistent concurrent actions must compete with the need of users to share their partial results with each other. As a result of these characteristics, collaborative applications have their own peculiar concurrency control requirements. Some, such as support for long transactions, user control over transaction execution, relaxed consistency criteria, and integration with access control, may be found in [2] and [15]. We offer the following additional requirements which so far have not been addressed in previous work. Concurrency control for collaborative applications should:

- permit high-level definition of application-specific consistency criteria. Application developers should be able to specify policies that reflect the structure and semantics of the application objects, in a high-level language.
- allow users to specify policy at time of collaboration. Application developers do not have complete knowledge of how an application will be used, so users should be able to specify or modify the concurrency control policy based on their particular collaboration.
- integrate concurrency control policy with merge policy. The extent of conflicts that cannot be automatically merged, as indicated by the merge policy in force, may determine the concurrency control policy users select. For example, the merge policy for bibliography entries may indicate that two sets of changes to the same entry cannot be automatically merged. The concurrency control policy should then reflect this by not allowing concurrent editing of different fields of a single entry.
- integrate concurrency control policy with coupling policy. The extent of awareness of each other's activities facilitated by coupling may affect the extent of mediation required by the concurrency control mechanism. If, for example, users of a shared drawing application specify a coupling policy in which users immediately see the editing actions of others, they may also desire a non-restrictive concurrency control policy.
- provide implicit "read" operations. In collaborative applications where users view and edit a display of the shared data, there is a need to distinguish, for the sake of the concurrency control system, which data a particular action actually depends upon, and which data is merely visible on the screen. The application developer should be able to provide this dependency information.

In the next section we describe a concurrency control framework that we have designed to meet these requirements. We next evaluate with respect to related work, and then present our conclusions and thoughts for future work.

A CONCURRENCY CONTROL FRAMEWORK FOR COLLABORATIVE SYSTEMS

Our solution has aspects of several of the related systems discussed earlier. Like database systems, it provides concurrency control for a range of types; like advanced transaction models, it allows concurrency control to be depend on object semantics; and like dOPT, DistView and GroupKit, it considers the domain of collaborative applications. It extends these works in several ways. First, it offers the concept of composable constructor-based hierarchical lock compatibility tables, which are defined for two important constructors that appear in collaborative applications. Second, it identifies methods for automatically filling these tables based on coupling and merge policies. Third, it motivates and describes several new concurrency control policies. Fourth, it provides programmer-defined dependency relations that automatically link items not related through structural hierarchy.

Suite Data Model

The basis of our concurrency control framework is the Suite collaboration framework [3]. Suite provides general structured objects, fine-grained object attributes, and flexible coupling among object views. Suite applications consist of user-defined data structures hierarchically composed of simple values, records, and sequences, and user-written functions that respond to users' changes to the data structure. This data model has proved sufficient for creating a variety of collaborative applications, including collaborative text editors, outline editors, spreadsheets, and debuggers.

Constructor-Specific Lock Compatibility Tables for Hierarchical Structures

Our framework's lock compatibility tables combine features of multiple granularity locking with type-specific locking. They are based on the type constructors provided by the Suite data model: simple types (integers, strings, etc.), records, and sequences. We discuss the lock tables for each in turn.

Lock Tables for Simple Types

A lock table for a simple type is shown in Table 2. Instead of Exclusive and Share, we use the terms Modify and Read. These are the only lock modes provided for simple types as these are the only operations that may be performed on them.

Lock	Lock held	
request	Modify	Read
Modify	No	No
Read	No	Yes

Table 2. Simple lock compatibility table

Lock Tables for Records

Lock tables for structured types reflect the operations that may be performed on these types. Table 3 shows a lock table for a record type. RM (Read-Modify) serves the same purpose as Gray's SIX lock. M(*field*) and R(*field*) serve much the same purpose as Intent Exclusive and Intent Read in multiple granularity locking. An M(*field*) lock indicates that the holder has a Modify lock on the record field named *field*, or on some part of the field (if the field is itself a structured object); similarly with R(*field*). Rows with *field* parameters represent lock requests on fields for which a lock is already held; rows with *field'* parameters represent lock requests where the field parameter does not match the field parameter of any lock already held.

Lock request	Lock held				
	M	R	RM	M(*field*)	R(*field*)
M	No	No	No	No	No
R	No	Yes	No	No	Yes
RM	No	No	No	No	Yes
M(*field*)	No	No	No	No	No
M(*field′*)	No	No	No	Yes	Yes
R(*field*)	No	Yes	Yes	No	Yes
R(*field′*)	No	Yes	Yes	Yes	Yes

Table 3. Lock compatibility table for records

Our lock modes for records are similar to Gray's multiple granularity lock modes. Ours differ in that instead of IX and IS, which indicate that *some* child of the object is locked, we have M(*field*) and R(*field*) modes, which indicate that *field*, or some child of *field*, is locked. Giving parameters to the modes allows multiple policies. For example, instead of the policy in Table 3, which allows one user to modify one field and another user to concurrently modify another field, we may wish to specify that users can concurrently read separate fields, but that only one user may modify a field, allowing only reading of other fields. This policy is not possible using only S, X, IS, IX, and SIX modes. (The same effect is achieved using an SIX lock but what lock a user chooses cannot be enforced by policy.)

Lock Tables for Sequences
Sequence lock tables share the Modify, Read, and Read-Modify modes of record lock tables, but differ in two respects. First, there are lock modes for Insert and Delete operations, which are parameterized by a sequence position number. An Insert(*i*) lock allows insertions at position *i*; Delete(*i*) allows deletions at position *i*. Second, Modify and Read Intent modes are parameterized by sequence position number instead of field name. An example sequence lock table is shown in Table 4. The rows of the table with an *i′* parameter represent the case where a lock request has a position parameter that does not match the position parameter of any lock already held. The table below prevents concurrent insertions, deletions, and modifications at the same position, but allows concurrent access to different positions. This policy may be preferred for a WYSIWIS (What You See Is What I See) text editor for synchronous collaboration, in order to prevent two users inserting characters at the same position. The DistEdit [17] collaborative editor uses a like policy. Entries in rows and columns for Modify and Read locks are as expected and are the same for all policies.

Lock request	Lock held						
	M	R	RM	I(*i*)	D(*i*)	M(*i*)	R(*i*)
M	No	No	No	No	No	No	No
R	No	Yes	No	No	No	No	Yes
RM	No	No	No	No	No	No	Yes
I(*i*)	No	No	No	No	No	No	No
I(*i′*)	No	No	No	Yes	Yes	Yes	Yes
D(*i*)	No	No	No	No	No	No	No
D(*i′*)	No	No	No	Yes	Yes	Yes	Yes
M(*i*)	No	No	No	No	No	No	No
M(*i′*)	No	No	No	Yes	Yes	Yes	Yes
R(*i*)	No	Yes	Yes	No	No	No	Yes
R(*i′*)	No	Yes	Yes	Yes	Yes	Yes	Yes

Table 4. Lock compatibility table for sequences.

Locking Algorithm
A user request for a lock on a particular node in the object tree spawns a series of requests, beginning with a request for the appropriate Intent lock (Read or Modify, parameterized by field or sequence index) on the parent object. This in turn generates a request for a lock on the parent's parent, and so on up the tree until the root is reached. (As our locking processes are currently single-threaded, whether a request goes leaf to root or root to leaf does not matter, as it does in Gray's scheme.) Only if all requests succeed is the user's lock request granted.

The kind of lock requested on a parent structure depends on the kind of lock requested on the child. A request for a Read, Read(*field*), Read(*i*) lock, or for any lock whose corresponding operation does not modify the object, causes a Read(*field*) or Read(*i*) lock request on the parent, depending on the constructor type of the parent. Otherwise, a Modify(*field*) or Modify(*i*) lock is requested. Because the framework supplies the constructors, it ensures that a lock request on a particular constructor generates the correct parent lock request, whether Read or Modify. For example, requests for an Insert(*i*) or Delete(*i*) lock on a sequence that is a field of a record must generate a request for a Modify(*field*) lock on the parent record because Insert and Delete operations change the "value" of the sequence.

A sequence insert or delete operation changes the mapping between sequence positions and the elements in the sequence, and consequently the position parameters of the element locks on the sequence must change. The following rules specify how a set of sequence locks changes as a result of insert and delete operations, for the display of the user performing the operations.

Insert(*i*):	For all locks with parameters greater than or equal to *i*, increment the parameters. Lock element *i* in Modify and Insert modes.
Delete(*i*):	For all locks with parameters greater than *i*, decrement the parameters. Delete locks associated with deleted element.

Because of Suite's flexible coupling, not all user displays receive all insert and delete operations. Therefore we maintain maps that translate an element's original position to its new position, and vice versa, and use them to map lock requests between corresponding elements.

Comparison With Type-Specific Locking
Our lock tables are similar to the type-specific tables of Schwarz and Spector [24]. They differ in a subtle respect, however, in that ours are constructor-specific rather than type-specific. Type-specific locking forces developers to create specific lock tables for each type created, whereas with constructor-specific locking, the lock tables of a specific type are composed from the lock tables of the constructors used to compose the type. In other words, a type's concurrency control policy is defined at the level of the type's implementation. From the perspective of our requirement for application-specific consistency criteria, this represents a step backwards, away from fidelity with actual type semantics. We feel, however, that the benefits to ap-

plication developers outweigh the loss of perhaps some useful policies. Since in any programming framework the number of type constructors is finite and typically small, the framework can predefine lock tables for them and provide rules for composing these tables.

Alternative Sequence Concurrency Control Policies

In this section we present two alternative concurrency control policies for sequences. As entries for Modify, Read, and Read-Modify rows and columns do not change from policy to policy, we omit these from the lock tables that follow. Taking again the scenario of two users collaboratively writing a paper and adding citations to the bibliography, the first example reflects the situation where the writers wish to simply accumulate citations as they write and remove duplicates later. This may be the case in the early stages of the paper's development. The only concurrency control they feel they need is at the individual citation level, where they will enforce exclusive access.

Table 5 below implements this policy. It reflects slightly different semantics for sequences than does Table 4; changed entries are highlighted and unchanged rows are not shown. The policy regards locks as rights to edit particular elements rather than particular positions. Thus it allows one user to insert at a position where another user is modifying, because inserting at that position does not affect modifications to the element at that position—the existing element is merely bumped up one position, and its locks are accordingly adjusted.

Lock	Lock held			
request	I(*i*)	D(*i*)	M(*i*)	R(*i*)
I(*i*)	Yes	Yes	Yes	Yes
D(*i*)	Yes	No	No	No
M(*i*)	Yes	No	No	No
R(*i*)	Yes	No	No	Yes

Table 5. Sequence lock compatibility table, less restrictive

When it comes time to properly prepare the bibliography for the final version of the paper, the users select a more restrictive concurrency control policy that allows only one of them to insert or delete entries, to ensure that they do not add duplicates. They do, however, wish to allow concurrent modification of (different) individual entries. This policy, shown in Table 6, reflects a more restrictive policy than that of Table 4; again, changed entries are highlighted and unchanged rows are not shown. This policy maintains the property that only one person may insert or delete on a sequence at a time.

Lock	Lock held			
request	I(*i*)	D(*i*)	M(*i*)	R(*i*)
I(*i′*)	No	No	Yes	Yes
D(*i′*)	No	No	Yes	Yes

Table 6. Sequence lock compatibility table, more restrictive

Concurrency Control Policies Derived from Coupling and Merge Policies

We have demonstrated how the concurrency control policy can reflect different object semantics and collaboration scenarios. To this point we have assumed that users choose a policy by either selecting from pre-defined policies or modifying an existing policy entry by entry. In this section we would like to present a third way, which is to transform existing concurrency control policies based on the policies of other collaboration functions.

Concurrency Control Policy from Merge Policy

A merge policy specifies how conflicts are to be resolved at merge time [21]. This information can be used during the transaction to resolve conflicts as they occur, in ways different from a conventional concurrency control policy. Merge policies typically use different criteria than do concurrency control policies. At merge time transactions are complete, so as a result, merge policies may place more emphasis on preserving each user's work and less on considerations of correctness, assuming that corrections can be made later. In the concurrency control domain this translates to a bias in favor of those operations that construct or preserve, and against those operations that destroy.

There are several policies possible for merging. In Table 7 we show a merge policy for a sequence. The row of the merge table corresponds to the operations performed by one user and the column corresponds to operations performed by the other user. The entries in the table indicate if the merged version should include operations of the row user, the column user, or both users (which makes sense in case of insertions and deletions), or if the result is to be determined by a user-defined function or interactively by users, or if the changes should be merged at a lower level of the structure.

User 1	User 2 operation			
operation	I(*i*)	D(*i*)	M(*i*)	No op *i*
I(*i*)	Both			Row
D(*i*)		Both	Column	Row
M(*i*)		Row	Merge	Row
No op *i*	Column	Column	Column	

Table 7. Sequence merge policy

In general, merging of inconsistent versions is necessary when consistency-maintaining concurrency control cannot be used either because the users are disconnected from each other or because it is too strict and unduly reduces concurrency. In these situations, merging is responsible for curing inconsistencies that were not prevented by concurrency control. However, merging is not an ideal solution since users may expend considerable effort in interactively resolving conflicts.

Therefore, users manipulating an object may switch between using merging and concurrency control to maintain consistency, using concurrency control when the merge cost is high and merging when the cost of reduced concurrency is high. Moreover, they may use a liberal concurrency control policy and a merge policy together to maintain consistency, where concurrency control prevents the subset of inconsistencies that are associated with high merge costs, and merging cures the remaining inconsistencies. In these situations, the objects being manipulated have associated with them both merge tables and lock tables.

The information in these tables, however, may not be independent. For instance, if the merge table accepts insertions of both users, then the implication for concurrency control may be that simultaneous insertions are allowed. Therefore, it would be useful if such implications could be made automatically by the system, thereby relieving users from the effort of specifying duplicate and possibly inconsistent information in the two tables.

We support such implications through tables that map merge actions to lock actions. Table 8 is an example of such a table.

Merge entry	*Merge action*	*Mapping to lock compatibility table*
Both	Do both ops	Yes
Column	Do column op	No
Row	Do row op	Undo lock-held operation, then Yes
(blank)	(no action)	(no mapping)
Users	Interactive selection by users	No
Merge	Merge changes to substructure	No
User-defined	User-defined function	No

Table 8. Mapping from merge policy to conc. control policy

Applying the mappings of Table 8 to the lock compatibility table of Table 4 yields the lock compatibility table of Table 9, below. Note the highlighted entries that changed.

Lock request	Lock held			
	I(*i*)	D(*i*)	M(*i*)	R(*i*)
D(*i*)	Yes	Yes	No	No
M(*i*)	Yes	Undo	No	No

Table 9. Sequence lock table, after merge policy application

In Table 9 we see a departure from a Yes/No action which undoes one user's deletion so that another user may keep and modify that element. This action anticipates the merge action that would have done the same thing. Such a feature is, of course, only practical in an environment that provides a robust undo facility. We recognize that by undoing one action, other actions of the transaction may become invalid. At present we rely on application developers and users to recognize when this is the case and not select this option for the lock compatibility table, but we are investigating including the option to undo all changes to a structure when changes to any substructure are undone.

Table 8 is only one example of a mapping between merge and lock actions. It assumes that concurrency control is responsible for resolving all inconsistencies. We can define a more liberal policy that maps the 'User' merge action to a 'Yes' lock action, thereby letting the inconsistencies introduced by this change to be resolved at merge time. This policy would be used when the cost of reducing concurrency is high and the cost of fixing inconsistencies is low. For example, two users may each be responsible for creating alternate versions of a paper abstract, and the merge decision is simply to choose one or the other.

Similarly we can define a more conservative mapping that maps 'Both' to 'No', on grounds that the automatic merging is imperfect and users would later be expected to choose one or the other.

Note that we have so far defined several distinct concurrency control policies for sequences, differing in their degree of restrictiveness. Yet the most restrictive offers a reasonable amount of concurrency and the least restrictive maintains a significant amount of control.

Concurrency Control Policy from Coupling Policy
At times a subset of users may wish to abandon concurrency control between themselves and allow unrestricted operation, relying on social protocols conducted over a separate channel to synchronize. And yet they do not want to relax concurrency control with respect to others. Our model achieves this with shared Modify locks.

When a user with a Modify lock receives a request for the same Modify lock, the request should be granted if it is from a user with whom the first user wishes to share the object. Lock compatibility tables may make this decision if *user* is included as a parameter in the rows of the tables. Table 10 shows a partial table that grants an already-held Modify lock to Munson but not to others. (When the *user* parameter does not appear "All" is assumed; but a row with a specific user overrides this.)

Lock request	Lock held			
	I(*i*)	D(*i*)	M(*i*)	R(*i*)
...	...	...	...	...
M(*i*)	No	No	No	No
M(*i*, Munson)	Yes	Yes	Yes	No
...	...	...	...	...

Table 10. Sequence lock table with shared locks policy

Shared lock rows can be added to lock compatibility tables directly, but our framework includes an automatic method for doing so. It is based on the expectation that shared locks are most desirable between those users who are closely coupled, i.e., those users who see each others' partial results. So we offer a mechanism that consults an object's coupling tables [4] and adds shared lock rows for those users who are closely coupled. The user may customize the new entries to achieve the desired sharing.

Lock rows that specify a specific user as a parameter allow lock policies to be based on user roles/identities. One application of this would be in support of a user who has the task of adding citations and cross-references throughout a document, and thus requires fine-grained access to every part. A policy with this user given shared locks allows the user to perform the task without locking everyone else out.

Dependency Relations for Implicit Locking

Classical concurrency control typically assumes a user invokes a read operation to observe the state of the shared data. In most collaborative systems, users do not invoke read commands—a user's view receives another user's write operation as an update according to the coupling established between the users. If read/write serializability is used as the consistency criterion, a safe approach would be to assume that what the receiving transaction does following the receipt of an update—any update—depends on that update. This is clearly too restrictive since users may receive many updates that are unrelated to their own task. So current collaborative systems assume that all concurrent updates are independent of each other.

Because our goal is a framework that supports a range of policies, including serializability, we are not satisfied with such an assumption. Our solution is to provide for programmer-defined dependencies. If a programmer declares that data item X is dependent on data item Y , then if a user requests a Modify lock on X, a Read lock on Y is automatically requested, and the lock on X is granted only if the lock on Y is granted. The problem of deadlock does not arise because a Modify lock and all its Read dependencies are obtained atomically.

Note that to some extent Read dependencies duplicate the function of Read-Modify locks, in that they both allow a user to atomically request Read locks as a result of a Modify lock request. In contrast with Read-Modify locks, however, Read dependencies are not restricted to parent structures of the item being Modify-locked. Also, a Read dependency may be established by the application developer, and in that case a Read lock is requested each time a Modify lock for the dependent object is requested. A Read-Modify lock may not be requested automatically, but is up to the user to request.

To alleviate the burden on programmers to declare Read dependencies, we are considering implementing default, overridable, dependencies, such as between all fields in records and all elements in sequences.

Implementation in Suite

In this section we briefly describe how our framework is implemented in the Suite collaboration system. The Suite architecture is a hybrid of replicated and centralized architectures. User display and input processing is replicated in the Dialogue Managers (DM) and application code is centralized. For the sake of performance, lock tables and lock-held lists are replicated at each user's DM.

While a DM is waiting for its lock request to be granted or denied, the user is allowed to proceed editing the structure for which the lock was requested, and may move on to edit other structures. Greenberg and Marwood [14] refer to this as *fully optimistic* locking. If another request is received before its own is received, the DM undoes the user's changes and blocks further editing on the corresponding item. Only the changes pertaining to that lock request are undone; other changes made between the time the undone changes were made but prior to when they were undone are not undone. We presume the user will be able to determine whether these changes should be undone as well, in the case that they depended on the undone changes, and undo them manually.

A user commits a transaction by issuing the Accept command, which causes the Dialogue Manager to update the application object with the user's changes. At this time, and not before, the user's locks are released. (We thus use the conventional two-phase locking protocol.) A user may perform a partial commit by selecting a (well-defined) portion of the application object(s) and issuing the Accept command, which commits only the changes to the selected substructure. We discuss this feature later in the Evaluation section.

Lock compatibility tables are stored as Suite object attributes, with each object inheriting its lock table from its type. Each type receives default lock tables according to the constructors it is based on. Lock tables may be specialized for particular object instances. The normal Suite attribute mechanism is used for storing the lock tables but the normal attribute lookup mechanism was altered for performance reasons.

Lock requests and lock table modifications may be performed interactively through the Dialogue Manager user interface. To lock an object, the user selects the object and clicks on one of the Read, Modify, or Read-Modify buttons that appear in the panel of other edit operations. Editing a simple type or invoking a sequence operation automatically generates the appropriate lock request. So while the lock tables may suggest that users have an overwhelming number of lock choices, there are in fact only three lock buttons. Lock table entries are presented in tabular form, largely as shown here, with pop-up menus of available options.

Our framework provides C functions for setting lock table entries, setting Read dependencies, and requesting locks on behalf of users. At present we offer no means for the application to request locks on its on behalf, as in most applications the application object itself is not considered an independent agent. So that we may interface Suite applications with non-Suite applications, which would likely communicate with Suite processes through the Suite application object, we are considering adding support to allow an application to request locks for external agents.

Correctness

A concurrency control system is correct if it maintains the application's consistency criteria. For a system that promotes flexible definition of consistency criteria, as ours does, a proof of correctness must account for all possible definitions of consistency. Fortunately, the core of our framework—hierarchical constructor-based lock tables—is a straightforward extension of mechanisms (multi-granularity locking, type-specific locking) for which formal proofs of correctness already exist [12, 24]. It only remains for us to prove that their interaction will not admit of incorrect behavior, which we do here informally.

In our framework a user's locks are held until the end of the transaction (marked by the user indicating "Accept"), as in conventional two-phase locking. Thus correct behavior is a matter of ensuring that each lock request, as it is propagated up the structure, is handled at each level in a manner consistent with lower levels. We assume that each type's lock table is correct, by virtue of the application developer, or possibly the user, having determined it. Then the only way incorrect behavior could happen would be if a type-specific lock request at one structural level were incorrectly propagated to a parent level. Since propagated lock requests are either Read(*component*) or Modify(*component*), it is necessary only to ensure that those operations that cause a state change of the object are propagated as Modify(*component*) locks, and those that do not as Read(*component*) locks. This is ensured by our framework for each type we provide. Since we assumed that each lock table is correct, we can assume that the upward-propagated lock requests are handled correctly.

EVALUATION

We evaluate the flexibility of our approach by examining to what extent it can simulate the concurrency control policies offered by previous work in database systems and collaborative systems.

DistEdit

If we model a text buffer as a sequence of characters, then Table 4 simulates the DistEdit concurrency control policy, with the exception that it provides additional Read locks. If we model the text with a more hierarchical structure, such as a sentence being a sequence of words, a paragraph being a sequence of sentences, etc., then users gain the ability to lock at various levels, depending on their needs for non-interference.

GroupKit

GroupKit allows concurrent operations on a collection of graphics objects but serializes access to a single object. If we model the collection as a sequence, the following lock table provides this policy. Only Delete, Modify, and Read access to elements is serialized; Insert operations are allowed to proceed concurrently. In GroupKit, a single lock covers Insert, Delete, and Modify operations.

Lock	Lock held			
request	I(*i*)	D(*i*)	M(*i*)	R(*i*)
I(*i*)	Yes	Yes	Yes	Yes
I(*i'*)	Yes	Yes	Yes	Yes
D(*i*)	Yes	No	No	No
D(*i'*)	Yes	Yes	Yes	Yes
M(*i*)	Yes	No	No	No
M(*i'*)	Yes	Yes	Yes	Yes
R(*i*)	Yes	No	No	No
R(*i'*)	Yes	Yes	Yes	Yes

Table 11. GroupKit policy

Multiple Granularity Locks

Our framework subsumes in functionality the multiple granularity locking scheme of [11], with the exception that we do not offer an Update mode, which gives read access to the holder with the option of upgrading the lock to an Exclusive lock.

Serializable Queues

Schwarz and Spector give examples of policies for queues, which Suite does not directly support. So we will assume they are implemented with sequences, and that queue behavior is enforced through normal Suite mechanisms such as attributes and user-written validation functions. That is, only insertions after the last element and deletions of the first element are allowed. Furthermore we assume that the value of the dequeued element is obtained as a side effect of the deletion. We first show a policy for serializable queue access. We show only relevant entries.

Lock	Lock held	
request	I(*i*)	D(*i*)
I(*i*)	No	Yes
D(*i*)	No	No
D(*i'*)	Yes	No

Table 12. Serializable queue policy

This policy simulates the concurrency control policy of Schwarz and Spector's strictly FIFO queue. The fact that a new Insert lock is obtained upon each Insert operation, and that these are incompatible with Delete locks at the same position, prevents one user from dequeueing an element that another has enqueued in a concurrent transaction.

Table 13, again abbreviated, presents our policy for the weakly FIFO queue, which was described earlier. Multiple users may insert and delete, and only Delete operations on the same element are exclusive.

Lock	Lock held	
request	I(*i*)	D(*i*)
I(*i*)	Yes	Yes
D(*i*)	No	Yes
D(*i'*)	Yes	No

Table 13. Weakly FIFO queue policy

In addition to these policies simulating the policies of other systems, we have defined several new policies for concurrency control, as found in Tables 5, 6, 9, and 10.

Predicate Satisfaction

In the transaction model of Korth and Speegle [18], a transaction's precondition must be satisfied before it can begin executing, and its postcondition satisfied before it can commit. The merge matrix of our merge procedure, to which transaction updates are subject when users have been working asynchronously (uncoupled Dialogue Manager views) is a language for postconditions. Furthermore, our merge framework allows application developers to supply merge procedures (written in C) for special cases. Thus we offer a high-level, but restricted language for postconditions, as well as a low-level, but unrestricted language. We are currently investigating a language of intermediate level and restrictiveness.

Other Advanced Transaction Models

Barghouti and Kaiser [2] describe many advanced transaction models, among them Split/Join transactions and participant transactions. Our framework offers some of the capabilities of these models, in a lightweight fashion.

The Suite "Accept" command by default commits changes to all application structures. But through selection a user can selectively commit changes to substructures. This is in effect a Split operation, for it frees the committed substructure and enables other users to lock it. A partial accept fails if any part in the substructure to be committed is locked through a Read dependency.

Shared locks offer some of the capabilities of participant transactions in that they allow users to share work in a manner not permitted by the application's consistency constraints, but maintain consistency with respect to non-lock-sharing users.

CONCLUSIONS

This paper makes five main contributions. First, it considers how applicable database concurrency control schemes are to collaborative applications and discusses their limitations. Second, it presents a new framework based on hierarchical constructor-based lock compatibility tables that strikes a balance between flexibility—the ability to capture application semantics—and ease-of-programming. Third, it shows the relationship between concurrency control and coupling/merging and describes how concurrency control policies can be automatically derived from coupling and merge policies. Fourth, it motivates and presents several new concurrency control policies. Finally, it provides programmer-defined dependency relations that automatically link items not related through structural hierarchy.

Important future work includes adding to our framework support for other commonly used constructors such as trees and tables (directories). Work is also necessary to integrate our work with advanced transaction models further supporting nested transactions, participant transactions, altruistic locking, and split/join transactions [2], and groupware concurrency control schemes addressing performance issues.

ACKNOWLEDGMENTS

This research was supported in part by an IBM Fellowship and National Science Foundations grant IRI-9408708. We would also like to thank the reviewers, whose insightful comments on our work significantly improved it.

REFERENCES

1. Bancilhon, F., Kim, W., and Korth, H. A model of CAD transactions. In *Proceedings of 11th International Conference on Very Large Databases* 1985, pp. 25–33.
2. Barghouti, N.S. and Kaiser, G.E. Concurrency control in advanced database applications. *ACM Computing Surveys 23*, 3 (September 1991), 223–317.
3. Dewan, P. and Choudhary, R. A high-level flexible framework for implementing multi-user user interfaces. *ACM Transactions on Information Systems 10*, 4 (October 1992), 345–380.
4. Dewan, P. and Choudhary, R. Coupling the user interfaces of a multiuser program. *ACM Transactions on Computer Human Interaction 2*, 1 (March 1995), 1–39.
5. Ellis, C. and Gibbs, S. Concurrency control in groupware systems. In *Proceedings of ACM SIGMOD Conference on Management of Data* 1989, pp. 399–407.
6. Ellis, C.A., Gibbs, S.J., and Rein, G.L. Groupware: some issues and experiences. *Communications of the ACM 34*, 1 (January, 1991), 9–28.
7. Eswaran, K.P., Gray, J., Lorie, R., and Traiger, I.L. The notions of consistency and predicate locks in a database system. *Communications of the ACM 19*, 11 (November 1976), 624–633.
8. Farrag, A.A. and Ozsu, M.T. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems 14*, 4 (December 1989), 503–525.
9. Garcia-Molina, H. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems 8*, 2 (June 1983), 186–213.
10. Garcia-Molina, H. and Salem, K. Sagas. In *Proceedings of ACM SIGMOD Annual Conference* 1987, pp. 249–259.
11. Gray, J. Notes on database operating systems, In *Operating Systems: An Advanced Course*. Springer-Verlag, New York, 1978.
12. Gray, J., Lorie, R.A., Putzolu, G.R., and Traiger, I.L. Granularity of locks and degrees of consistency in a shared database, In *Modeling in Database Management Systems*. Elsevier North-Holland, Amsterdam, 1976.
13. Greenberg, S. and Bohnet, R. GroupSketch: a multi-user sketchpad for geographically distributed small groups. In *Proceedings of Graphics Interface* 1991, pp. 207–215.
14. Greenberg, S. and Marwood, D. Real time groupware as a distributed system: concurrency control and its effect on the interface. In *Proceedings of ACM Conference on Computer Supported Cooperative Work* 1994, pp. 207–217.
15. Greif, I. and Sarin, S. Data sharing in group work. In *Proceedings of ACM Conference on Computer Supported Cooperative Work* ACM, New York, 1986, pp. 175–183.
16. Herlihy, M. Apologizing versus asking permission: optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems 15*, 1 (March 1990), 96–124.
17. Knister, M. and Prakash, A. Issues in the design of a toolkit for supporting multiple group editors. *Computing Systems 6*, 2 (Spring 1993), 135–166.
18. Korth, H. and Speegle, G. Formal model of correctness without serializability. In *Proceedings of ACM SIGMOD International Conference on Management of Data* 1988, pp. 379–386.
19. Korth, H.F. Locking primitives in a database system. *Journal of the Association for Computing Machinery 30*, 1 (January 1983), 55–79.
20. Lynch, N.A. Multilevel atomicity: a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems 8*, 4 (December 1983), 484–502.
21. Munson, J. and Dewan, P. A flexible object merging framework. In *Proceedings of ACM Conference on Computer Supported Cooperative Work* 1994, pp. 231–242.
22. Prakash, A. and Shim, H.S. DistView: support for building efficient collaborative applications using replicated objects. In *Proceedings of ACM Conference on Computer Supported Cooperative Work* 1994, pp. 153–164.
23. Sarin, S. and Greif, I. Computer-based real-time conferencing systems. *Computer 18*, 10 (October 1985), 33–45.
24. Schwarz, P.M. and Spector, A.Z. Synchronizing shared abstract types. *ACM Transactions on Computer Systems 2*, 3 (August 1984), 223–250.
25. Stefik, M., Bobrow, D.G., Foster, G., Lanning, S., and Tatar, D. WYSYIWIS revised: early experiences with multiuser interfaces. *ACM Transactions on Office Information Systems 5*, 2 (April 1987), 147–167.
26. Weihl, W. and Liskov, B. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems 7*, 2 (April 1985), 244–269.