A Proclamation-Based Model for Cooperating Transactions

H. V. Jagadish
Oded Shmueli

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We propose a transaction model that provides a framework for transactions to cooperate without sacrificing serializability as a notion of correctness. Cooperation does not depend on detailed knowledge of the semantics of transaction operations. Semantic properties such as data dependent commutativity can be "discovered" automatically at *run time* without a need to declare these properties explicitly.

When transactions wish to cooperate, they do so by issuing "proclamations". A proclamation is an (implicitly or explicitly specified) set of values, one of which the transaction "promises" to write if it commits. So, a proclamation provides incomplete information concerning future possible database states. Transactions can compute with this incomplete information, and can commit after writing conditional multi-values.

We examine the theoretical basis for the proclamation model. We outline an implementation strategy for the model, including a simple lock-based transaction manager and a transaction compiler extension to handle sets of values.

1. INTRODUCTION

Traditionally, concurrency control for transactions has relied upon a rigorous correctness notion of serializability, see [6]. The serializability restriction may be relaxed by relying on the semantics of operations [3, 19]. In many cases, one can live with a weaker notion of correctness, and indeed, for performance reasons, one often does so. For instance, many real users of large database systems today rely on weaker ad-hoc notions of correctness such as "cursor stability" [9]. At the same time, the longer duration of transactions has increased the pressure not to apply concurrency control too strictly.

As databases are applied to non-traditional applications, such as design and software development, the possibility of cooperation between transactions increases, and the transaction management system should be able to adapt to take advantage of this. Cooperation typically requires one transaction relying on certain behavior by another transaction. While this reliance usually is based on some higher level semantic knowledge, it can often be reduced to a reliance on a particular update behavior. In particular, a transactions may be able to predict, at least partially, what value it will write for a particular data item, call it X, well in advance of the transaction completing its computation and committing. Another transaction, wishing to read the value of X, may be able to perform useful computation even if it does not know the exact value of X, but instead merely that X belongs to some *set* of values. Several examples are presented in Section 2 where such is the case.

In this paper, we propose a transaction model in which transactions are allowed to cooperate, if they so choose. We do so without sacrificing serializability. The transactions in our model are similar to traditional transactions: they are deterministic, and transform consistent states into consistent states. When transactions wish to cooperate, they do so by issuing proclamations. A proclamation is an (implicitly or explicitly specified) set of values, one of which the transaction "promises" to write if it commits. The transaction management system ensures that transactions meet some minimum guarantees with respect to these proclamations. Therefore, a malicious transaction cannot cause other transactions to err by issuing false proclamations.

Our model reduces to the conventional model if transactions choose not to cooperate. The major point of departure in the case of cooperating transactions is that a transaction may read a set of values and proclaim a set of values. Transactions are also *monotonic*: intuitively, if each read operation of a transaction is made to read a subset of what it actually reads then each update operation will produce a subset of the values it actually produces.

The paper is organized as follows. We present a few motivating examples in Section 2. In Section 3 we formalize the transaction model and present a theorem establishing a variant of view serializability that relies on reading and writing subsets of values appearing in the actual execution. Section 4 presents extensions to the basic model that greatly enhance its utility. Section 5 considers larger systems issues, in terms of how the transaction code is written and compiled. Section 6 discusses how to implement a transaction manager based on our model, and shows that small modifications of existing standard methods is all that is required. Section 7 concludes.

Related Work

Many extended transaction models have been proposed, for example [2, 8, 14, 16, 17]. There have also been many attempts at finding alternative notions of correctness. Korth et. al. treat a general transaction model, including versions and subtransactions, where correctness is defined using pre-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada, 1992

conditions and post-conditions [10]. In multi-databases one may define "local" and "global" consistency [13]. Nodine and Zdonik use finite state machines to specify the allowed interleavings of transactions[14]; these are similar in spirit to path expressions [7] and follow the work of [18]. These models either require the definition of a compensating transaction (to make up for commitment that was allowed too early), rely upon the transactions displaying certain semantics in their behavior or operations, or coordinate behavior using other mechanisms. For example, the work of [3] relies on a notion of "recoverability" that allows a transaction to commit after reading non-committed data. This data can be "fixed" in case of abort of its producing transaction.

A scheme that maintains for each object an old value and a new value has been proposed in [4]. So, transactions may utilize the old value if the new one is being updated. The scheme increases parallelism. A locking scheme is provided to implement the idea with some unwanted side effects, e.g. a transaction may obtain the old value, for consistency reasons, even after a new committed value has been produced. More generally, there is a theory of multi-version concurrency control [5], where each item is allowed to have multiple versions. This feature allows more flexibility in scheduling algorithms, one can give each read request an appropriate version, in many cases, that will preserve consistency. Our scheme is different in that we do allow access to non-committed values, based on some properties of our transactions (monotonicity). Further, we allow access to sets of possible values, one being a real version and others being possibilities.

2. MOTIVATION

It is often the case that transactions know, or are able to predict (at least partially), what values they may write at completion. Furthermore such prediction is often useful to other transactions. In fact, the guarantees about future behavior encoded in such predictions are a fundamental primitive in terms of which cooperation can be defined. In this section we present a few examples from diverse applications.

2.1 Airline Reservation

We begin by considering a classic problem of high data contention. Let X, the variable of interest, be the number of seats available on a particular flight. A typical transaction, checking the availability of seats for a passenger, needs only to know whether the current value of X is non-zero. If it can be informed, for instance, that the current value of X is 14 or 15 depending on whether some other reservation transaction commits or aborts, it can go ahead and provide a positive response regarding seat availability. If this transaction now wishes to reserve a seat, having confirmed availability, it can do so, and mark the final value of X as either 13 or 14, depending on whether the "true" value of X read by the transaction was supposed to have been 14 or 15, respectively. In airline reservations systems today, the special semantics of this process are used to provide a customized concurrency control mechanism specific to the particular application. The ideas we develop below, we claim, can have much the same effect, but with complete independence from the semantics of the application.

Now consider a different transaction, run by the airline, that checks how many seats are left unused, to determine if a smaller aircraft will suffice. This airplane assignment transaction, even though it reads the value of X from the database, does not care whether the value read is 14 or 15 or 16. It can compute and commit as long as it can be certain the value is less than a threshold for aircraft substitution.

Observe that for the seat reservation transactions efficiency improvements, similar to those we can obtain through a uniform proclamation paradigm, can be obtained in a semantics-dependent manner by using escrow locks [15] or commutativity [3]. Due to their close dependence on the semantics of seat reservations, these techniques cannot permit concurrent execution of the airplane assignment transaction.

2.2 Modular VLSI Design

Suppose that one designer is currently modifying module A. If this designer can guarantee certain aspects of the interface to this module, then another designer can start working on module B that has to interface with A. In fact, the update to module B can be committed even before the update to module A is complete! All that has to be guaranteed is that at commit time, the promise made by the designer of A is kept – the updated module A must indeed meet the promised interface specification.

More specifically, consider (a grossly simplified view of) integrated circuit design. Let transaction A be working on the layout of module `adder`. The `length` and `width` of a module are two important attributes. Let transaction B be a floor planner. The task of transaction B is to place the different modules in the chip to minimize some objective function, such as total area. Transaction B does not care about the internal details of the design of module `adder`: all it needs are the length and width.

Often, transaction A can proclaim a small set of possible bounding rectangles for its implementation of `adder` before the design is completely done. Transaction B can then make a number of floor plans, one for each A-proclaimed rectangle, and proceed with its computation without waiting for A to finish. Alternatively, transaction B can determine reasonable upper bounds on the length and width of the layout of module `adder`, allocate sufficient space to accommodate any of these rectangles, and proceed.

2.3 Software Engineering

Consider the development of an object-oriented system. Let transaction A begin work to modify the *definition* of a class `cl_foo`. Let transaction B develop code that requires the definition of class `cl_foo`. If transaction A proclaims a set of data members and member functions that it will (not) modify, and if transaction B requires use only of members not being modified by A, then transaction B could utilize this knowledge to proceed without waiting for A to finish. This is because the definitions of members that B uses are guaranteed to stay stable. (Of course, in most programming languages the code written by the two transaction must be compiled together prior to execution. Here we focus only on the code development process and not on code execution).

The "value" of an object, such as the definition of class `cl_foo`, is a "string" representing its entire definition. Transaction A, in this example, proclaims only some property of this string that will remain invariant, still leaving open many different strings that could finally result. In spite of the proclamation not being an explicitly enumerated set of values, transaction B is able to make use of it.

Most design and other cooperative applications are very complex. Of necessity, the examples presented above are simple. Our purpose is to convince the reader that there are indeed applications where it is possible to predict at least part of the outcome ahead of time, and where such predictions can be utilized.

3. THE MODEL

3.1 Basic Features of the Model

Our model is that transactions wishing to cooperate can issue *proclamations* regarding the value of a particular data item that they will write at the end. Each proclamation is a set of possible values that the data item may take at the end of a transaction. (This set may be specified implicitly or explicitly, and could even be infinite in some cases. In particular, a proclamation may often be just an invariant with nothing said about data items not included in the invariant). A transaction may issue multiple proclamations with regard to an individual data item. In this case, each successive proclamation has to be a subset of the preceding proclamation for that data item.

When a transaction attempts to read a data item, it picks up the latest information regarding the data item, whether this is from a committed write, or through a proclamation. By *latest* we mean the latest value written by a *non-aborted* transaction. If it reads a proclamation, it can go ahead and compute with the set of values presented to it for the data item read. If it completes its computation, it may commit.

There is an additional requirement regarding transactions that issue proclamations. They must have read the value of the data item they are issuing a proclamation for, either through a true read, or from a proclamation, prior to issuing any proclamations. We require that the set of values written in a proclamation include not only the values the transaction at that point thinks it might update the data item to, but also the value(s) of the data item as read by the transaction. This is required to take into account the possibility of an abort. (In Sec. 6.4 we show how this requirement can effectively be guaranteed by the transaction management system even if the transaction code indicates a "blind write").

The major requirement is that any value written by a transaction for a data item be invariant irrespective of which of the possible instantiations of the possible set of values in any input obtained by read proclamation is eventually "declared" to be the value to be read by the transaction. (We will relax this requirement later).

When a transaction commits, it first performs W operations for all items it proclaimed values for, and perhaps some additional ones. Its last operation is C. So, W operations write *committed values*, as in [11]. The last operation of an aborted transaction is A.

We present an example to give the reader a flavor of the sorts of executions possible under the scheme we propose. $P[X]$ denotes a proclamation on X. $R[X]$ denotes a read operation on X (either of a committed value or a proclamation), $W[X]$ denotes a final declaration of a committed value for X. We shall use $U[X]$ to denote an update operation on X ($P[X]$ or $W[X]$) and $O[X]$ to denote any operation on X. C and A indicate successful completion and abortion, respectively.

Example 1

```
-----------------------------------------------------> (time)
t1: R1[X] P1[X]            R1[Y]         W1[X] C1
t2:               R2[X] W2[Y] C2
```

Transaction t_1 reads "from" t_2 and vice versa. The two transactions execute concurrently. It is still possible to serialize t_2 ahead of t_1 because the value that t_2 writes is the same whether it reads the value of X prior to t_1 or the one produced by t_1.

We summarize the main features of transactions in this model:

i. When a transaction t_i issues a read operation, it reads from the latest value (written by a write or a write proclamation) produced by a non-aborted transaction t_j. The result obtained may be a single unique value, or a set of values. The former is a true read, the latter is the read of a proclamation. In this paper, we do not distinguish between the two and refer to both by the symbol R.

ii. A transaction must read (or read proclamation) an item it later updates.

iii. If a transaction issues a proclamation on item X, the set of values proclaimed must include the latest value(s) it read for that item.

iv. If a transaction issues more than one proclamation on X, then each proclamation must be a subset of the previous proclamation.

v. If a transaction writes item X, for which it has made a proclamation, the value must be one of the values in its latest proclamation for X.

vi. Once a transaction writes (W) any item, it reads no more items and it produces no more proclamations for any item.

vii. A transaction writes (W) to all items it proclaimed values for (and perhaps some additional ones) iff its last operation is C. A transaction performs a write (W) to any item only if its last operation is C. (We show in Sec. 6.4 how such a requirement can effectively be met even with in-place updating).

viii. If a transaction aborts then it writes (W) to no item and A is its last operation.

ix. Consider an execution of a transaction t. In this execution it reads and updates, both type of operations can refer to single values (reads and writes) and multiple values (read proclamation or issue proclamation). Transaction t is said to possess the *monotonic computation property* if it satisfies the following condition: If t produces sets $K_1, \ldots, K_n$ as issued

proclamations when supplied sets $J_1, \ldots, J_m$ for read proclamations then when supplied for its read proclamations sets $J'_1, \ldots, J'_m$, where J'_i is a (non-empty) subset of J_i (not necessarily proper), $i=1, \ldots, m$, it produces $K'_1, \ldots, K'_n$ and produces the *same* values in all write (W) operations, with K'_i a (non-empty) subset of K_i (not necessarily proper), $i=1,..,n$. Furthermore, the set of operations and the order of operations within a transaction remains unchanged.

The *monotonic computation assumption* is that each transaction in our system possesses the monotonic computation property. We show in Sec. 5 how this property can be guaranteed by the compiler for the transaction code.

3.2 Serializability

Consider a general transaction system supporting the interleaving of transaction operations. We define a *realizable history* as a pair (S,E) where E is the binary read-from relation (which we view as a directed graph) among operations and S is the sequence of system (read and write) operations in the order they executed. The sequence S is constructed once the system has ceased operating, i.e. no transaction is active, or equivalently, with all active transactions aborted. $(U_j[X], R_i[X])$ is an edge in E iff $R_i[X]$ read the value that was produced by $U_j[X]$ in the execution; we use $j=0$ when reading from the initial database. In particular, $(U_j[X], R_i[X])$ is an edge in E implies that U_j preceded R_i in S, i.e. reading causality. Note that E is not in general deducible from S.

In our transaction system, we postulate a final transaction t_f that reads for each item in the database its latest value which is written by a committed transaction; t_f performs no updates, and the state it reads is defined as the *final state*. We create appropriate corresponding edges for operations of t_f in E.

An *execution history* is a realizable history which could be produced in a transaction system conforming with our model. Now, consider an actual execution history (S,E) in such a transaction system. There is exactly one edge in E for each read operation (R), from the immediately preceding update operation (W or P) of a non-aborted transaction. The *effective sequence* of system operations, s, is derived from the execution history in three steps. The first step deals with aborted transactions. The second is an optional step in which transactions may "choose" a source for some of the values they read. The third step effectively gets rid of all P operations.

Reduction Modification Procedure

Consider an execution history containing transactions t_1, t_2 and item X such that transaction t_2 reads X via an operation $R_2[X]$ from an update operation $P_1[X]$ performed by transaction t_1.

Let $R_1[X]$ be the reading operation of t_1 on X which is most recently preceding $P_1[X]$. Such an R_1 must exist because of the requirement of reading before updating. E is modified by deleting $(P_1[X], R_2[X])$ and adding $(U_3[X], R_2[X])$, where the edge $(U_3[X], R_1[X])$, for some U_3, is in E. (I.e., making t_2 read from where t_1 read just prior to proclaiming the value for X read by t_2).

Transformation Procedure

1. Let t_1 be the first aborted transaction. Consider each transaction t_2 and item X such that t_2 reads X from t_1, and apply the reduction modification above to t_1, t_2 and X. Then remove from S all operations performed by t_1, and from E all edges relating reads performed by t_1.
   Repeat the above procedure (stated for t_1) for each aborted transaction t_j, in the order of abort events, A_j. Thereby obtain a modified pair (S',E'), which we shall call the *committed history*.
2. Consider a transaction t_2 that has an operation $R_2[X]$ reading from an operation $P_1[X]$. Apply the reduction modification procedure to t_1, t_2, and X. (I.e., we think, temporarily, of t_1 as "aborted" and make $R_2[X]$ read the earlier value of X).
   This transformation is optional, and may be applied to none, some, or all reads from proclamations. Once this transformation has been applied in as many places as desired, the resulting pair (s',e') is called the *pre-effective history*.
3. We now treat edges of the form $(U_1[X], R_2[X])$ in e'. If the edge is $l=(W_1[X], R_2[X])$ then move $R_2[X]$ immediately to the right of $W_1[X]$ in s'. If the edge is $l=(P_1[X], R_2[X])$ then move $R_2[X]$ immediately to the right of $W_1[X]$ in s' and replace l in e' with the edge $(W_1[X], R_2[X])$. (The intuitive justification for this transformation is that by retaining edge l in step 2 we "intend" t_2 to read X from t_1). The resulting pair (s,e) is called the *effective history*. In general, this effective history may look strange: for example, there may be an $R_2[X]$ operation *following* the C_2 operation in s (in that case (s,e) is *not* even a realizable history). The s component of the effective history is called the *effective sequence*.

Fig. 1 illustrates the transformation procedure.

Next, we define *conflicting operations* in the usual way [6]: $R[X]$ does not conflict with $R[X]$, $R[X]$ conflicts with $W[X]$, $W[X]$ conflicts with $W[X]$. Create from s a *conflict graph* with (committed) transactions as nodes, and a directed edge (t_i, t_j) iff transaction t_i has an operation in s that precedes a conflicting operation in s of transaction t_j. We are now ready to state the main observation, the proof is omitted for lack of space.

Theorem 1:
Let G be the conflict graph for the effective sequence, obtained by the procedure described above. If G is acyclic then the transactions in S' can be executed serially, i.e. one after another, with the order being a topological sort on G, such that:

i. Each transaction in this serial execution, for each item X, reads one of the values that it read for X in the actual execution (a limited version of "view equivalence"), and
ii. The net effect of the entire serial execution is to move the database to exactly the same final state as in the actual execution with execution history (S,E).

Consider the following execution history (transaction t_a is the only one aborted):

$S = W_0[X] R_1[X] P_1[X] R_a[X] P_a[X] R_2[X] R_3[X] P_3[X] A_a \; C_2 \; W_1[X] C_1 \; W_3[X] C_3 \; R_f[X]$
$E = \{(W_0[X], R_1[X]), (P_1[X], R_a[X]), (P_a[X], R_2[X]), (P_a[X], R_3[X]), (W_3[X], R_f[X])\}$

After step 1 of the transformation procedure, we get:

$S' = W_0[X] R_1[X] P_1[X] R_2[X] R_3[X] P_3[X] C_2 \; W_1[X] C_1 \; W_3[X] C_3 \; R_f[X]$
$E' = \{(W_0[X], R_1[X]), (P_1[X], R_2[X]), (P_1[X], R_3[X]), (W_3[X], R_f[X])\}$

Apply the transformation of step 2 to t_2 that read X from t_1:

$s' = W_0[X] R_1[X] P_1[X] R_2[X] R_3[X] P_3[X] C_2 \; W_1[X] C_1 \; W_3[X] C_3 \; R_f[X]$
$e' = \{(W_0[X], R_1[X]), (W_0[X], R_2[X]), (P_1[X], R_3[X]), (W_3[X], R_f[X])\}$

Finally, after step 3 of the transformation procedure, we have:

$s = W_0[X] R_2[X] R_1[X] P_1[X] P_3[X] C_2 \; W_1[X] R_3[X] C_1 \; W_3[X] R_f[X] C_3$
$e = \{(W_0[X], R_1[X]), (W_0[X], R_2[X]), (W_1[X], R_3[X]), (W_3[X], R_f[X])\}$

The conflict graph that is obtained from the above transformed history:

G is t_0 ----> t_2 ----> t_1 ----> t_3 ----> t_f

G is acyclic and the serial execution $t_2 \; t_1 \; t_3$ produces the same final state as the original execution.

Figure 1. Example illustrating the Transformation Procedure to determine serializability

An execution history for a set of transactions is *serializable*, if it produces the same final database state as some serial, one by one, execution of the transactions. The conclusion is that if the conflict graph is acyclic then the execution history is serializable. Observe that the conflict graph for a given actual execution is not unique. In fact, it is quite possible that while some conflict graphs that can be obtained for an execution history have cycles, others are acyclic. An execution is serializable if it has *at least one* conflict graph that is acyclic. Indeed, Theorem 1 shows that a topological sort on such an acyclic conflict graph is a valid serialization of the execution history.

4. EXTENSIONS TO THE BASIC MODEL

4.1 Multiple reads

A basic feature of our model is conveyed by the monotonicity assumption. In a real world situation, a transaction may continue operating on the assumption that the value for X is taken out of a known set up to a point where it needs more specific information. One way to do this is for a transaction to perform multiple conditional computations, and select one of them at the end. If the number of alternative computations to be performed is not too high, this may be worth doing, particularly in parallel processing situations. At that point it may issue a $R[X]$ again in the hope of refining its knowledge.

The first concern is to specify what value it reads on this subsequent $R[X]$ operation. If the transaction t, from which it has read the value in the preceding $R[X]$ operation, is not aborted, then the value received is the latest value posted (through W or P) by t. In case t has aborted, the value is obtained from the transaction from which t read X, in case that one has not aborted - and so on, recursively. Observe that in any case, the subsequent read is a subset of the previous read.

Furthermore, with the above scheme, Theorem 1 still holds as its proof does not depend on having, for each X, only a single read operation. Of course, the transactions must still obey the monotonicity assumption. This implies that care must be taken in compiling transactions that perform multiple reads so that the same sequence of operations, or a sub-sequence leading to the same written values, would result in considering a serial execution history.

4.2 Conditional Writes

Another possible extension is to allow transactions to perform W operations that write *conditional multi-values*, a conditional multi-value is a set of conditional values. An example $W[X = (\text{if } Y=1 \text{ then } 7, \text{ if } Y=3 \text{ then } 8, \text{ otherwise } 5)]$. Such a multi-value should specify what to write for each combination of currently possible values for items read. Once such multi-values can be written, they can also be read. This presents no problem since these reads behave as if they are reading proclamations, as far as their utilization is concerned.

To deal with such conditional values we refine the meaning of a C operation. Up till now, C signified "completion of committed transaction". We shall now re-think about C as an "effective completion" and introduce a new symbol D for the "true completion". *True completion* means that all writes (Ws) performed by the transaction have been refined into single values, *effective completion* means that the transaction has written to all items for which it made proclamations, but some of the written values may be conditional. Recall that, in all cases, no W operations are performed until a decision to commit has already been reached.

Multi-values can be refined into single values *after* execution. Once there are no more active transactions these

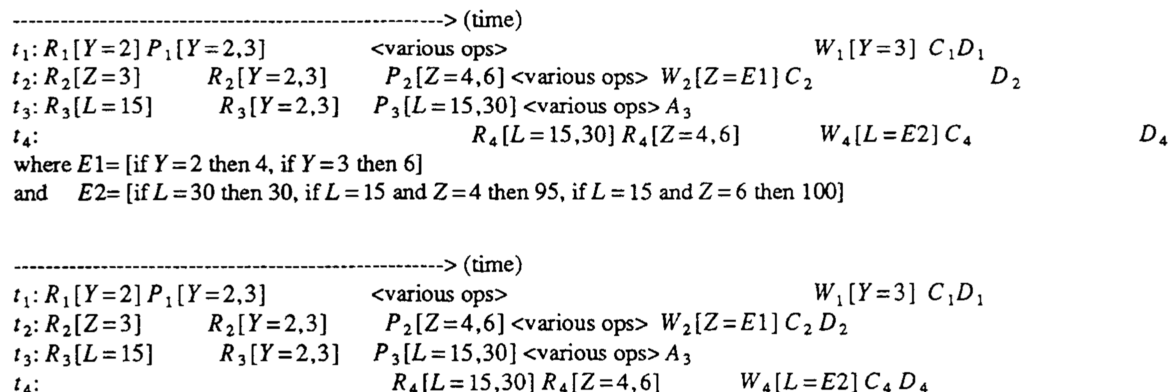

Figure 2. Two possible execution histories corresponding to an execution with multi-valued writes

active values are resolved as much as possible. This refinement depends on the conflict graph chosen. If the conflict graph chosen is acyclic, each W, involving multiple conditional values, can eventually be refined into a single value. In proof, consider the first transaction in Ss that wrote a multi-value, since it only reads single values it can resolve its Ws into single values; then proceed inductively.

The reason some refinements are performed after the execution is that the refinement may depend on the particular conflict graph chosen. In fact, multi-values can also be refined by the system *during* execution. Conceptually, a multi-value remains *active* and at regular intervals the transaction system tries to resolve its state. It does so by reading the items that determine the conditions. This is similar to performing multiple reads (Section 3.1) but here the reading is performed by the transaction system.

Operationally, the transaction management system can "help" transactions resolve their conditional values by applying the reduction modification of step 2 of the transformation *at run time*. This means "ignoring" the transaction t that X was read from, and "deciding to read X" from where t read it. This way, the possible values of X are restricted, which facilitates resolving conditional writes. In this case, during execution, the transaction system makes decisions that restrict the possible set of conflict graphs.

Example 2

Consider the execution history shown in first part of Fig. 2. Since t_1 D-completes (reaches true completion) before t_2, t_2 effectively writes $Z=6$. Transaction t_4 writes a conditional value; the actual value is determined by t_2 and t_3. Since t_2 writes $Z=6$ and since t_3 aborts with $L=15$ still in effect, the conditional write of t_4 is $L=100$. Observe that the order of value determination is t_1, t_2, t_4. This also happens to be a serialization order.

However, a different serialization order may also be imposed. This order is t_2, t_4, t_1. In this case t_2 reads the value of Y read by t_1 (by application of the reduction modification of optional step 2 of the transformation procedure to this (uncommitted) read from t_1). Therefore, the committed value of Z is 4, and the value of L written by t_4 is 95. In this case the sequence of events is as shown in the second part of Fig. 2.

The final database state obtained is different in the two serialization orders. However, they are both "correct" from the perspective of serializability. We have permitted the concurrent execution of interdependent transactions and let them commit with conditional writes. Different legitimate serializations may resolve these conditional values differently.

4.3 Refining Monotonicity

Monotonicity was defined in terms of set inclusion. There are other possibilities. Define a partial ordering, denoted "$<$", on values. Define a set A to be a $<$-subset of a set B if for each element a of A there exists an element b of B such that $a<b$. Now replace the subset specification in the definition of monotonicity with $<$-subset.

This allows us to look at more general notions of monotonicity. For example, chip floor plan P may be $<$ than another floor plan Q if each module placed in P has sufficient space allocated for it in module Q. With this extended notion we can handle the floor plan design example option of reserving sufficient space. Here, the particular monotonicity property ensured by a transaction is left to the transaction code writer to enforce. In case we use the (ordinary) subset notion of monotonicity, a compiler can produce monotonic transactions from user specified ones. This is the topic of the next Section.

5. SYSTEM DESIGN

An important task is to produce a monotonic transaction given the code of an "ordinary" transaction. In this section we sketch how our ideas can be incorporated into transaction code. In particular, we show how it is often possible to make minimal modifications to transaction code to be able to read and write multi-values rather than single values.

One burden we place on the user is to declare explicitly a list of proclamation predicates of relevance. These are the predicates whose truth a transaction may proclaim, and the ones which when proclaimed can be used by the transaction. (Since proclamations are only used between cooperating transactions, it is reasonable to expect the transaction writer to know what kinds of proclamations cooperating transactions

may make). This information is captured in a `proclamation_definition.h` file. In particular, a predicate of the form "`member_of (`<*enumerated set*>`)`", is likely to be used often.

The user writes transaction code as before, reading and writing single values. When appropriate, explicit proclamations are inserted by the user into the code. This is done by means of a new keyword `proclaim`, which takes as arguments, the (identifier of the) database entity being proclaimed for, the specific predicate (of the ones already declared) being proclaimed, and parameters supplied to this predicate, if any.

Let us consider a simple case in which a transaction reads a number of database items, performs some calculations, and then updates a set of database items. The code for the transaction's calculations is iterated by the compiler, once for each possible combination of values for proclamations of interest. Each iterated execution has its own local copies of variables that it updates. The value proclaimed for a variable is the union of values in each iterated case. The final value of a variable is determined uniquely (and can be written into the database) if it is the same across all local copies of the iterated executions performed, otherwise, it is conditionally dependent on the specific values of inputs obtained imprecisely. If the transaction system is not capable of handling conditional multi-values then, the code is augmented with repeated reads until unique values may be determined.

We do not, at the present, have a compiler implemented. Nevertheless, in the Appendix, we present some pseudocode to give the reader an idea of the transformations a compiler would make. The key point is that it is often possible to to take an ordinary user transaction code and transform it into transaction code that can read and write sets of values in a monotonic way. The concurrency control ideas in this paper can be used, even without such a compiler, provided users are willing to write monotonic transactions that are capable of reading and writing multi-values.

6. REALIZATION

In this section we show how to build a transaction manager to realize the conceptual model described in section 3. In particular, we show that proclamation-based concurrency control can be implemented using a locking protocol that is a slight extension of two-phase locking. We discuss how to integrate the proposed scheme with a standard write-ahead-log based recovery scheme.

Other realizations of the conceptual model are possible. Our intention in describing the schemes below is simply to show that relatively simple techniques can be used to generate correct executions according to our criteria.

6.1 A Locking Scheme

There are three types of locks on an object – *shared*, *exclusive*, and *preferred*. The first two are standard. Only one transaction can hold an exclusive lock on a data item, and in that case, no transactions may hold any other locks. Only one transaction can hold a preferred lock on a data item, but at the same time any number of additional transactions can hold it in shared mode. To be able to read a data item, a transaction must hold at least one of the locks listed. To be able to update a data item, a transaction must hold either an exclusive lock or a preferred lock.

Standard two-phase locking is implemented with the modification that after the first proclamation of a transaction, the transaction system changes an exclusive lock on the corresponding data item to preferred. In fact, a transaction is never permitted to "upgrade" to a preferred lock from no lock or a shared lock; it must have an exclusive lock on a data item, and then voluntarily downgrade it to a preferred lock.

When a transaction wishes to read an item, it first tries to obtain a shared lock on the item. (Not required if the transaction already has a stronger lock – exclusive or preferred – on the item). When a transaction wishes to write an item, it obtains an exclusive lock on the item. If it wishes to cooperate, the transaction may issue a proclamation and downgrade an exclusive lock to preferred.

All the rest of the locking protocol is standard. Except for the downgrading discussed above, no locks are given up until all locks required have been obtained. All locks are given up at completion, by commit or abort, time. If a transaction desires a lock that is currently unavailable, it waits for the current holder of the lock to finish. Where there is competition for locks, any transaction scheduling algorithm may be used. Any standard deadlock prevention technique, or deadlock detection and resolution technique, may be used.

Lock actions are $h[X]$ ($h'[X]$) for obtaining (resp., releasing) shared locks on X, $x[X]$ ($x'[X]$) for obtaining (resp., releasing) exclusive locks on X, $d[X]$ for downgrading an exclusive lock to a preferred lock, and $p'[X]$ for releasing a preferred lock.

6.2 Correctness

Consider transactions in the order of appearance of their first write, denoted by F, operation (or C in case of no W operation) $F_1, \ldots, F_n$, in (S', E'), the committed execution. (This is the same as the order in the actual execution history, except that aborted transactions have been dropped, since step 1 of the transformation procedure does not alter any W or C points). We shall argue below that this order is a serialization order for an execution according to the scheme above. Without loss of generality operation F_i is of transaction t_i, $i = 1, \ldots, n$.

A basic assumption is that a transaction performs its first W operation once it knows it commits, and in particular it will need no more locks. (In a write-ahead log based system with in-place updates, all W operations, in the sense used here, take place at commit time. See Sec. 5.4 below). So, the F_i point identifies the point the ith decision was reached to commit a transaction. We shall argue that in the final conflict graph there will be no edge from t_i into t_j such that $i > j$. This implies that the graph is acyclic, with the F_i order being a topological sort, and the history is serializable. We start with a fundamental observation (the proof is in Appendix 1):

Lemma 4:
For all i, j, X such that both $W_i[X]$ and $W_j[X]$ are in the execution history, F_i precedes F_j iff $W_i[X]$ precedes $W_j[X]$.

Example 1

--> (time)

t_1: $x[X]\ R_1[X]\ P_1[X]\ d[X]$ $h[Y]\ R_1[Y]$ $W_1[X]\ h'[Y]\ p'[X]\ C_1$
t_2: $h[X]\ R_2[X]\ x[Y]\ W_2[Y]\ x'[Y]\ h'[X]\ C_2$

Example 2

--> (time)

t_1: $x[Y]\ R_1[Y]\ P_1[Y]\ d[Y]$ <various ops> $W_1[Y=3]\ p'[Y]\ C_1$
t_2: $x[Y]\ R_2[Z]$ $h[Y]\ R_2[Y]\ P_2[Z]\ d[Z]$ $W_2[Z]\ p'[Z]\ h'[Y]\ C_2$
t_3: $x[Y]\ R_3[L]$ $h[Y]\ R_3[Y]\ P_3[L]\ d[L]$ <various ops> $p'[L]\ h'[Y]\ A_3$
t_4: $h[L]\ h[Z]\ R_4[L]\ R_4[Z]$ $x[L]\ W_4[L]\ x'[L]\ h'[Z]\ C_4$

Figure 3. Examples 1 and 2 reproduced with locking operations shown explicitly

Theorem 2:
There is a conflict graph derived from an execution following the locking scheme described above, such that, for $i=1,\ldots,n$, the only edges entering t_i are due to transactions $t_0,\ldots,t_{i-1}$.

(Recall that the conflict graph obtained through the transformation procedure of Section 2 is not unique. The Theorem here postulates the existence of at least one graph meeting the specified conditions. The proof is by construction and is presented in Appendix 1).

6.3 Uniqueness of Written Values

Now that we know a particular serialization order that can be induced on transactions executing according to the locking scheme specified here, we can show that there will never be a *necessity* to perform a multi-valued write at commit time, or delay committing to resolve conditional values. The idea is to always take steps consistent with the eventual conflict graph's topological sort being the commit order. The proof is by induction. Consider multiple valued write operations. The first transaction in the serialization can always immediately perform a unique valued write as it can consider its effective read to be from the initial database. This means there is no need for a wait (although one may choose to wait) to resolve the multi-value later on).

Consider t_i the ith transaction to commit. t_i need identify, for each item X, which value in what it read was written by an already committed transaction. Then, it can determine a unique write value. Now assume that all transactions serialized ahead of t_i have performed unique-valued writes. We show that t_i can perform a unique-valued write. By Theorem 2, we can think of all t_i read operations as done from operations of previously committed transactions (in F order). But, all these values are written when t_i decides to commit. Furthermore, by the induction hypothesis, they are single values. By the locking discipline, the needed value for X is the latest committed value written for X. Therefore, t_i can write only single values. Hence proved by induction.

6.4 Locking Scheme Implementation

We consider how the locking scheme described above might be implemented. The lock table is essentially the hash table as described in [6] with some new features. In particular, the current lock holders are partitioned into those having an exclusive lock, those having a preferred lock, and those having a shared lock on X. In case there is a preferred lock on X by transaction t, the value read by t for X is recorded in the *read field*. The proclamations made by t are recorded in the *proclamation field*. Since a transaction may make a number of proclamations, the proclamation field holds the latest proclamation.

The above description is conceptual; physically, if keeping an augmented lock table is too costly in terms of memory consumption, a pointer to disk resident data might be kept (one possibility is to keep such data as part of the system's log). By the locking discipline, at any point in time, there is at most one proclamation on X. We require that a physical write operation into the database always installs a single value. This practical restriction can be satisfied even though, conceptually, transactions may proclaim and write (conditional) multi-values.

In what follows we shall consider both P and W operations as "write" operations and use "value" to refer to either a single value or a multi-value.

When a transaction first reads item X, it obtains a shared or exclusive lock on X. If there is no current proclamation for X the value is read from the database, and recorded in the read field of the lock manager. If there is a proclamation on X, the value taken is the latest proclamation found in the proclamation field, it may be a multi-value.

To expose proclamations the transaction must:

a. hold an exclusive lock on X (which is now downgraded to a preferred lock),
b. have previously read X from the database, i.e. a committed value (it suffices if the latest committed value is in the read field – see optimization below), and
c. have updated X (and the latest updated value of X initializes the proclamation field).

An expose-proclamation is a one-time action: once it is performed, all subsequent updates are "visible" to other transactions. This visibility is made possible by converting the exclusive lock to a preferred lock.

To write an item, a transaction must have an exclusive or preferred lock on that item. (Which of the two depends only on whether the transaction has exposed its proclamations). We

consider these two cases in order:

1. The transaction writes an item to which it has not exposed proclamations. If the write is a single value, the value is simply written into the database (update in place) and the proclamation field. Appropriate logging is done to assure recovery in case of an abort or a crash, as in [12]. When a transaction writes a (possibly conditional) multi-value, the value, or a pointer to it, is written to the proclamation field in the lock table (overwriting whatever was there previously). As far as recovery is concerned this is an internal transaction operation and no logging is required.

2. The transaction writes an item to which it has exposed proclamations. The system checks that the written value is a subset of the latest proclamation; if not then the transaction is aborted. The written value, or a pointer to it, replaces the one in the proclamation field. If this is a single value it is (also) written into the database as in the previous case.

If a transaction aborts and it has previously issued proclamations for X, then the proclamations, including the read field, are erased and item X has no outstanding proclamations (see the optimization below). For each item in the database which is actually updated by the transaction, an undo operation is performed as in [12]. Space for any multi-values in, or pointed by, the proclamation field is reclaimed. Transactions that read proclamations made by aborted transactions are not immediately affected. If such a transaction attempts a read on X, a proclamation of which it previously read from an aborted transaction, X is obtained from the database.

A transaction is eligible to commit if for each item X to which it has written, its latest write operation defines a unique value. A transaction writes a unique value either by writing a single value or by writing a conditional multi-value which is exhaustive, i.e. covers all the possible cases for the values of items mentioned in the condition. Eligibility can be assured during compilation, we shall assume that only eligible transactions issue commit. Let t be a transaction eligible to commit. We shall describe two alternative basic ways for committing a transaction.

1. The first step in committing a transaction is to replace any conditional multi-value it has written by a unique single value. This is always possible according to the preceding sub-section. If a transaction has finally written conditional multi-values, these values are resolved as follows. Let X be an item for which the transaction read a proclamation. If item X is still locked in preferred mode, then the value of X is taken to be that in the read field, inductively, this value is a single value because it was written by a previously committed transaction. Otherwise, the value is read from the database. (An optimization is possible that will keep the last committed value for X, in the read field, as long as there are transactions holding shared locks on X.)

2. There is another option when a transaction is ready to commit; its advantage is that items reflect later updates; its disadvantage is that of holding locks for long durations. If a value read by the transaction still has an outstanding proclamation by some transaction t, this transaction waits for t to commit. The transaction will actually commit once all transactions it waits for have committed. Once the written value is resolved into a single value, that value is written into the database and the read field, as in the previous option. Thus, inductively, the read field only contains single values. It is possible that there is a cycle in the implied wait-for-commit relation (which we view as a directed graph). In that case a transaction is chosen to commit according to the previous scheme, by resolving to single values.

A mixed option is also possible, in which the transaction resolves some values and waits for other transactions to terminate on other values; when a cycle results some values are resolved to eliminate an edge in the wait-for-commit relation.

7. CONCLUSIONS

In this paper we have presented a proclamation-based model for cooperating transactions. We permit transactions to use uncommitted data in a controlled fashion, through proclamations. Transactions have the monotonic computation property which enables achieving serializable execution schedules, including reads of such uncommitted updates, without the need for cascading aborts. We presented a simple lock-based protocol that is able to obtain many of the benefits of our model.

We also permit transactions to commit while writing values that are conditioned upon incompletely specified values read by it. While we have assumed that complete specification of these values will happen "eventually", there is no conceptual reason why we could not delay this indefinitely. Thus our model can support transaction management in a database storing incomplete information.

While the presentation in this paper has mostly been in terms of proclamations being specified as an explicit set of values for a data item, clearly the set of possible values could be specified implicitly and could even be infinite. For instance, in an object-oriented system with object-level locking, a transaction may issue a proclamation regarding the value of a particular attribute of the object, while leaving the possible final state of the object otherwise unspecified. Another transaction, interested only in the value of this particular attribute, may be able to proceed in parallel even though it has no other knowledge of the state of an object.

Finally, in this paper we have not addressed the question of what motivates a transaction to issue a proclamation. Our expectation is that the issuing of proclamations may be autonomous in some high data contention situations. However, a more likely scenario, especially for design databases, is that a transaction, upon finding unavailable a data item that it wishes to access, may request the current lock-holder for a proclamation.

We believe that humans execute "transactions" in a manner more like our proclamation-based model than the traditional transaction model. We expect that our model will be of particular value in systems with long-duration

```
do {
  return_val_set->initialize() ;
  read(proclaim_set_for_num_seats) ;          // Read proclamation
  foreach x in proclaim_set_for_num_seats
    xact_copy(x)                   // Execute a copy of the transaction
}
while (return_val_set->count() >1)
                    // Repeat above until a unique return value is obtained
                    // The count of values must decrease monotonically
                    // since the proclaim_set becomes smaller monotonically
return return_val_set->unique() ;

xact_copy(y)
int y ;
{
  if (y == 0 ) return_val_set->addtoset(0) ;  // Failure to find seat
  else return_val_set->addtoset(1) ;          // Seat available
}
```

Figure 4. Transformed Code for Airline Seat Availability Query

transactions and cooperation, such as design databases. Using our model, data contention can be decreased without a detailed knowledge of the semantics of the particular application, and without sacrificing serializability as a correctness criterion.

Acknowledgement

We thank Shaul Dar for reading several drafts of the paper and for his many suggestions. We also thank Narain Gehani and Inderpal Mumick for their useful suggestions.

APPENDIX

To understand how a compile system may work, we walk through some pseudo-code in the spirit of O++, the programming language interface to the Ode object-oriented database system [1]. Consider the airline reservations example described in Sec. 2.1. `num_seats` is a database variable stating the number of seats available on a particular flight. The user writes the transaction, as shown below, with no regard for possible multi-values read.

```
if (*num_seats == 0 ) return 0 ;
else return 1 ;
```

This code is transformed, by the compiler, into the code shown in Fig. 4. `proclaim_set_for_num_seats` is the set of proclaimed values for `num_seats` that is read by the current transaction. `num_seats_val_set` and `return_val_set` are the set of values for `num_seats` and for the return value, respectively, that the transaction may produce. These sets are implemented by (and encapsulated in) an object class with member functions `count()`, `initialize()`, `addtoset()`, and `unique()`. The member function `unique()` returns a single value, which is the value of the unique set element, provided it is invoked on a singleton set. Even if multiple values for `num_seats` are present in the proclaim set, the transaction can complete with a unique return value immediately, provided all these values are non-zero. The code in the while loop is executed exactly once. If there is a zero value in the proclaimed set, then the transaction cannot complete until it can be sure whether there are seats available. It does so by repeatedly executing the while loop. If such "busy-waiting" is not desired, a `sleep()` for an appropriate duration can be added to the loop.

Now consider a more complex user transaction that actually wishes to record a reservation, updating the variable `num_seats`. Once more, the user code, as shown in Fig. 5, does not need any awareness of the proclamation-based concurrency control protocol in use. Following the same recipe as in the previous case, the compiler can generate the code in Fig. 6

This (intermediate) transformed code is very naive. In fact, it does not enhance concurrency at all. Several improvements are possible. First off, code independent of the set-valued proclamations being considered does not have to be iterated for each value. Second, one can define the notion of a *material predicate*, whose truth value affects the flow of computation, and iterate the code only once for each possible value of the material predicate. Third, the transaction can write a (conditional) multi-value and commit, and the transaction management system will determine the appropriate unique value in time.

In our example, we find `( num_seats == 0 )` to be a material predicate. The transaction code can then be transformed into a two phase computation, with a first phase where the truth of the material predicate is determined, and a second phase where the actual computation is performed. Fig. 7 exhibits these ideas; once the material predicate is decided a unique return value can be determined, and a reservation, if appropriate, can be made; the number of seats is a conditional multi-value.

One could get even more sophisticated. For instance, instead of writing explicit values for the conditional multi-values output, one could write a functional dependence. With this, one can get rid of the loop over the proclamation set entirely. In general, depending on the complexity of the compiler implementation that we are willing to tolerate, we can

```
  if (*num_seats == 0 ) return 0 ;       // Failure to find seat
  else {
    proclaim (num_seats, member_of (*num_seats, *num_seats-1)) ;
    record_reservation();           // Does not reference num_seats
    *num_seats-- ;
    return 1 ;                      // Reservation made successfully
  }
```

Figure 5. User Code for Airline Seat Reservation Transaction

```
do {
  num_seats_val_set->initialize() ;
  return_val_set->initialize() ;
  read(proclaim_set_for_num_seats) ;
  foreach x in proclaim_set_for_num_seats      // Read proclamation
    xact_copy(x)                    // Execute a copy of the transaction
  // Now proclaim for variable num_seats a predicate that
  // states that the value of num_seats is one of the values appearing
  // in num_seats_val_set
  proclaim (num_seats, member_of(num_seats_val_set)) ;
}
while (return_val_set->count() >1 || num_seats_val_set->count() >1) ;
                    // Repeat above until unique values are obtained for
                    // both return value and num_seats
if (return_val_set->unique()==1)
    record_reservation() ;
*num_seats = num_seats_val_set->unique() ;
return return_val_set->unique() ;

xact_copy(y)
int y ;
{
  if (y == 0 ) return_val_set->addtoset(0) ;    // Failure to find seat
  else {
    num_seats_val_set->addtoset(y-1) ;
    return_val_set->addtoset(1) ; // Reservation made successfully
  }
}
```

Figure 6. Partially Transformed Code for Airline Seat Reservation Transaction

choose how involved the logical inter-relationships between proclamations can be. The more sophisticated the compiler, the more efficient the transformed code can be.

REFERENCES

[1] R. Agrawal and N. H. Gehani, "ODE (Object Database and Environment): The Language and the Data Model," *Proc. ACM SIGMOD 1989 Int'l Conf. Management of Data*, Portland, Oregon, May-June 1989.

[2] B. R. Badrinath and B. Sriram, "Modeling Cooperative Transactions Using Obligations," *Unpublished Manuscript*, 1991.

[3] B. R. Badrinath and K. Ramamritham, "Semantics-based Concurrency Control: Beyond Commutativity," *ACM Trans. Database Syst.*, **17**(1), March 1992. (Preliminary version in *Proc. Fourth IEEE Conf. on Data Engg*, Feb. 1987, pp. 132-140)..

[4] R. Bayer, H. Heller, and A. Reiser, "Parallelism and Recovery in Database Systems," *ACM Trans. on Database Systems*, **5**(2), June 1980, 139-156.

[5] P. A. Bernstein and N. Goodman, "Multiversion Concurrency Control - Theory and Algorithms.," *ACM Trans. on Database Systems*, **8**(4), December 1983, 465-483.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[7] R. H. Campbell and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," in *Lecture Notes in Computer Science*,

```
material_predicate_evaluated = 0 ; // initialization
while (!material_predicate_evaluated) {
  read(proclaim_set_for_num_seats) ;
  if (all x in proclaim_set_for_num_seats are >0) {
    material_predicate = 0 ;
    material_predicate_evaluated = 1 ;
  }
  else if (proclaim_set_for_num_sets->count() ==1) {
    material_predicate = 1 ;
    material_predicate_evaluated = 1 ;
  }
}
      // At this point the material predicate has been evaluated
      // That is, we know whether we can successfully reserve a seat,
      // even if we do not have a unique value for num_seats

if (material_predicate) return 0 ;      // Failure to find seat
else {
  num_seats_val_set->initialize() ;
  foreach x in proclaim_set_for_num_seats
    num_seats_val_set->addtoset([x-1,x])
                    // num_seats_val_set now has conditional multi-values
                    // So each set element is a [value, condition] tuple
  proclaim ([num_seats,num_previous], ([num_seats,num_previous] is in num_seats_val_set)) ;
  record_reservation() ;
  return 1 ;                     // Reservation made successfully
}
```

Figure 7. Transformed Code for Airline Seat Reservation Transaction

vol. 16, 1974, Springer-Verlag.

[8] H. Garcia-Molina and K. Salem, "Sagas," *Proc. ACM-SIGMOD 1987 Int'l Conf. on Management of Data*, San Francisco, California, May 1987, 249-259.

[9] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of Locks and Degrees of COnsistency in a Shared Database," *Proc. of the IFIP Working COnference on Modeling Database Management Systems*, 1979, 1-29.

[10] H. F. Korth and G. D. Speegle, "Formal Model of Correctness Without Serializability," *Proc. ACM-SIGMOD 1988 Int'l Conf. Management of Data*, Chicago, IL, June 1988, 379-386.

[11] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Syst.*, **6**(2), June 1981, 213-226.

[12] B. G. Lindsay, *Single and Multi-Site Recovery Facilities in Distributed DataBases, Chapter 10*, Cambridge University Press, Cambridge, UK, 1980.

[13] S. Mehrorta, R. Rastogi, H. F. Korth, and A. Silberschatz, "Non-Serializable Executions in Hetrogeneous Distributed Database Systems," *Proc. PDIS 1st Int'l Conf. on Parallel and Distributed Information Systems*, Miami Beach, Florida , Dec. 1991, 245-252.

[14] M. H. Nodine and S. B. Zdonik, "Cooperative Transaction Hierarchies: A Transaction Model for Supporting Design Applications," *Proc. 16th Int'l Conf. Very Large Data Bases*, Brisbane, Australia, Aug. 1990, 83-94.

[15] P. E. O'Neil, "The Escrow Transactional Method," *ACM Trans. on Database Systems*, **11**(4), December 1986, 405-430.

[16] C. Pu, G. E. Kaiser, and N. Hutchinson, "Split Transactions for Open-Ended Activities," *Proc. of the 14th Int'l Conf. on Very Large Databases*, Los Angeles, California, Aug. 1988, 26-37.

[17] A. Silberschatz and E. Levy, "A Formal Approach to Recovery by Compensating Transactions," *Proc. 16th Int'l Conf. Very Large Data Bases*, Brisbane, Australia, Aug. 1990, 95-106.

[18] A. Skarra, "Concurrency Control for Cooperating Transactions in an Object Oriented Database," *SIGPLAN Notices*, **24**(4), April. 1989.

[19] W. Weihl, "Commutativity-Based Concurrency Control for Abstract Data Types," *IEEE Trans. on Computers*, **10**(4), Dec. 1988.