, , 1–22 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Bounding the Effects of Compensation under Relaxed Multi-level Serializability*

PIOTR KRYCHNIAK AND MAREK RUSINKIEWICZ AND ANDRZEJ CICHOCKI
{piotr,marek,andrzej}@cs.uh.edu
Computer Science Department, University of Houston, Houston TX 77204-3475

AMIT SHETH amit@cs.uga.edu
LSDIS Lab, Dept. of Computer Science, Univ. of Georgia, 415 GSRC, GA 30602-7404

GOMER THOMAS gomer@ctt.bellcore.com
Bellcore, RRC-1N275, 444 Hoes Lane, Piscataway NJ 08854-4182

Editor:

Abstract. The multi-level transaction concept provides a powerful tool for structuring activities in multidatabase systems. However, even multi-level serializability is sometimes too restrictive as a correctness criterion, either because of very high concurrency requirements, or because of the practical difficulties of implementing a scheduler in actual production environments. The extended multi-level transaction model presented in this paper supports higher concurrency in cases where higher level operations commute in one direction, but not in the other – i.e., when it is valid to interchange them when they occur in one order in a history, but not when they occur in the other order. We introduce a relaxed correctness criterion based on allowing a bounded number of out of order conflicts at each level in the multi-level framework, where the bound can be different for different levels. Finally we discuss the properties of compensation in this framework, developing a theory of compensation which depends only on the semantics of the operations and not on the particular state of the database. We illustrate the use of these concepts in the context of a particular class of practical applications.

Keywords: Multidatabase Transactions, Compensation, Relaxed Correctness Criteria, K-Serializability

1. Introduction

Many large applications operating in a multidatabase environment can be viewed in hierarchical terms. Each level of the hierarchy has its own set of operations and its own view of the data. The operations at any level of the hierarchy above the bottom level are implemented as transactions or procedures which invoke operations of the next lower level. This hierarchical structure may have its origin in the semantics of the applications. For example, design activities are usually hierarchical.

Nested transactions [14, 13, 1] and multi-level transactions [23, 24] have been designed for hierarchical activities. Although these transaction models use serial-

* This work was supported in part by MCC, Bellcore, and by the Texas Advanced Research Program under Grant No. 3652008. Majority of Sheth's work was performed at Bellcore.

izability as a correctness criterion, they increase the level of concurrency by taking advantage of semantic commutativity of higher level operations. This paper proposes a relaxed correctness criterion for hierarchical activities modeled as multi-level transactions, allowing a further increase in concurrency. We will show that the semantics of higher level operations can be used to define a class of non-serializable schedules that may be acceptable from the point of view of many applications.

Frequently, hierarchical activities are of long duration or access heterogeneous systems with very diverse response times. Because of this, the full isolation of conventional transactions may not be appropriate for such activities. However, relaxing the isolation requirements means that undo operations can no longer use the before-images. In case of transaction aborts and crash recovery, compensating operations may have to be used to undo the effects of committed transactions. In this paper we propose a compensation model for hierarchical activities and define a correctness criterion for it, based on the idea of limiting the period when an operation remains subject to compensation and bounding the number of operations which are exposed to its effects in the meantime. The main advantage of the approach proposed in this paper is that it allows reasoning about histories with compensating operations using only knowledge about the *semantics of operations*, without reference to the *semantics of database states*.

The paper is organized as follows. In Section 2 we present a motivating example illustrating the need for the a relaxed correctness criterion and for a new model of compensation in hierarchical transaction models. In Section 3 we propose a relaxed correctness criterion for hierarchical activities. In Section 4 we introduce a compensation model for hierarchical transaction models. In Section 5 we again refer to the example and show the possible application of the proposed compensation model. In Section 6 we discuss similar solutions proposed in the literature. Finally, in Section 7 we summarize our results.

2. Motivating example

Throughout the paper we will use an example from the telecommunication industry. Very often there is a need to build a telephone circuit, for example from a user's home to the central office, or between two users' locations. Description of various components of a circuit (lines, switches, etc.), and about their interconnections are stored in multiple databases. Since in most cases the physical components of the circuit are already in place, the only task required for building a circuit is to update the databases, so that the appropriate connections will be allowed. And, for the purpose of this paper, this is the only interesting task.

This environment can be represented as a multidatabase system consisting of a collection of semi-autonomous local databases[1], each with its own data manager. We will sometimes refer to each of these local databases as a "site," although some of them may be co-located on the same machine.

The databases contain information about *items* of various kinds. Items can be assigned to *assemblies* (denoted by $a, b, \ldots$). In our example, the items are cir-

cuit components: telephone lines, switches, etc., and the assemblies are telephone circuits. We assume that items are grouped into *classes*, e.g., lines leading from a certain place to another. Items from the same class provide similar functionality but may have different parameters such as performance or cost. We will denote items belonging to class X by $x_1, x_2, \ldots$. We assume that there is no overlap or replication among databases. Any individual item is represented in only one database.

Consider two high-level operations: an assignment operation and a deassignment operation. An assignment operation $A(X,a)$ assigns an "optimal" available item from a class X to an assembly a. The optimality criteria for selecting an item from a class X are based on the characteristics of the assembly a. For example, $A(X,a)$ may assign the least noisy line from the class of lines X to the circuit a. If $A(X,a)$ succeeds then the status of an item x_i changes from *unassigned* to *assigned to assembly a*. A deassignment operation $D(x_i,a)$ changes the status of an item x_i from *assigned to assembly a* to *unassigned*. Each of these high level operations is implemented in terms of lower level read and write operations. These high level operations have the useful property that a successful assignment $A(X,a)$ always can be compensated with the deassignment operation $D(x_i,a)$, and a deassignment $D(x_i,a)$ always succeeds, in the sense that immediately after it is executed and committed the item x_i becomes *unassigned*. Note that although these two operations may be seen as an operation – compensating operation pair, they are in fact independent from each other. Of course, the deassignment operation can be applied only to items that were previously successfully assigned.

Consider two important types of (global) activities. An assignment activity forms an assembly by assigning to it a collection of items belonging to different classes (from various sites). A deassignment activity dissolves an assembly by deassigning all the components assigned to it. Building a telephone circuit can serve as an example of such global activities. Obviously each such global activity consists of a collection of local subtransactions at various sites. Each local subtransaction is either an assignment or a deassignment operation, as discussed in the previous paragraph.

It may turn out that some subtransactions of a global assignment activity fail, in the sense that in a certain class of items there are no items available that can be assigned to the assembly and meet the optimality requirements. In some cases this may mean that assignments already made from other classes of items (possibly at other sites) must be undone, and a different combination of item types must be attempted. In rare cases it may mean the desired assembly cannot be formed, so the entire global activity must be aborted, and all assignments made by it so far must be undone.

A type of transaction that is widely used in many telecommunication applications (and many other industrial applications) is the *queued message model* [3]. In such systems a local transaction is initiated by an incoming message on the message queue. It then executes and commits unilaterally, without any opportunity for global commit coordination. In the future, the systems that use the queued message model may be replaced by the systems that support global commit coor-

dination. However, in order to improve the performance of the system it may still be appropriate to use unilateral commit of assignment transactions. This is due to the fact that the global assignment activities are typically of long duration and may access heterogeneous systems with very different response times. Therefore, the full isolation of those activities may need to be relaxed. By committing the local assignment subtransactions and allowing their results to become visible as soon as they are completed we can reduce the time the local systems are locked and thus we can achieve a considerable improvement in performance. We assume therefore, that local assignment subtransactions commit unilaterally with no global coordination.

The question then is how to guarantee correct concurrent execution of such global activities, assuming that each activity would be correct if executed in isolation from all the others. We will consider this problem in the framework of multi-level transactions with three levels. The global activities are the top level transactions. The assignments and deassignments at local sites are the middle level operations. The database reads and writes are the bottom level operations.

Let us consider two global assignment activities G_1 and G_2 that build two assemblies (circuits): a and b respectively. We assume that at site i G_1 performs a successful assignment $A_1(X,a)$ of an optimal available item x_1 from a class X to an assembly a. Similarly, G_2 at site i successfully assigns an optimal available item from a class X to an assembly b by performing an operation $A_2(X,b)$. Suppose that some other G_1's assignment to the assembly a has failed on some other site and a new combination of item types for a is attempted. As a result of this the previously assigned item x_1 of class X has to be deassigned and G_1 performs a deassign operation $D_1(x_1,a)$ at site i.

The above scenario may result in the following history of assignment and deassignment operations performed by G_1 and G_2 at site i: $H_1 = A_1(X,a) \bullet A_2(X,b) \bullet D_1(x_1,a)$. If we assume that the local data managers guarantee serializable execution schedules, then the executions of the database reads and writes are serializable with respect to the assignment and deassignment operations. The problem then is to demonstrate that the execution of the assignment and deassignment operations is correct with respect to the set of global transactions. In our example, if the global assignment activities G_1 and G_2 were executed in isolation the history of assignment/deassignment operations at site i would be as follows: $H_2 = A_1(X,a) \bullet D_1(x_1,a) \bullet A_2(X,b)$. This history is considered to be correct because it represents a serial execution of G_1 and G_2 at site i. Since the deassignment $D_1(x_1,a)$ semantically undoes the effects of the assignment $A_1(X,a)$ both operations can be purged from the history and therefore $H_2 = A_2(X,b)$.

To prove that the history H_1 is correct it is enough to show that it is equivalent to the history H_2, i.e. to the hypothetical history where G_1 and G_2 are executed in isolation. If the deassignment operation $D_1(x_1,a)$ could be moved ahead of $A_2(X,b)$ in history H_1 then H_1 would be correct since both histories H_1 and H_2 would be equal. The order of $A_2(X,b)$ and $D_1(x_1,a)$ could be exchanged only if executing $A_2(X,b)$ before $D_1(x_1,a)$ gives the same result as executing $D_1(x_1,a)$

before $A_2(X, b)$. [2] In such a case we will call exchanging the order of operations a *valid interchange*. An example of a valid interchange of operations in the context of assignment activities would be exchanging the order of assignments or deassignments of items belonging to different classes like e.g.: $A_1(X, a)$ and $A_2(Y, b)$, and $A_2(X, b)$ and $D_1(y_1, a)$.

In our example, executing $A_2(X, b)$ ahead of $D_1(x_1, a)$ may give a different result than executing $D_1(x_1, a)$ before $A_2(X, b)$, and thus exchanging their execution order cannot be considered a *valid interchange*. This is because the item x_1 assigned by $A_1(X, a)$ might be the last available item in the class X, therefore $A_2(X, b)$ will fail if executed before $D_1(x_1, a)$ and will succeed if executed after $D_1(x_1, a)$.

However, taking into account the semantics of higher level assignment operations may help us to reason about the effects of applying invalid interchanges to the histories with assignment and deassignment operations. This in turn may result in defining methods to bound the amount of inconsistency caused by relaxing the isolation of global assignment activities. We should stress that the analysis of the effects of invalid interchanges of operations in a history is possible only because of the rich semantics of assignment operations. It would not be possible at the read/write level where no semantic information is available. At the read/write level even one invalid interchange of operations in the history may have unpredictable results. Hence, the methods proposed below are not applicable in the context of traditional serializability theory based on read/write operations.

If we consider the problem from the standpoint of the optimality of assignments, moving $D_1(x_1, a)$ ahead of $A_2(X, b)$ should not be considered as a valid interchange. This is due to the fact that assignment $A_2(X, b)$ in history H_1 is given a more limited choice of class X items than in the correct history H_2. In our example, $A_1(X, a)$ could have assigned the last of fiber optics lines, and $A_2(X, b)$ had to choose an old copper one. In history H_2, the operation $A_2(X, b)$ has its choice expanded, since $D_1(x_1, a)$ made the fiber optics line available again. In history H_1 it may also happen that assignment $A_2(X, b)$ fails (if $A_1(X, a)$ assigns the very last line) and the global assignment activity must attempt a less desirable combination of items. The knowledge of the semantics of the assignment operation and in particular the knowledge about the criteria for the optimal choice of an item from a class can be used to trade-off some loss of the optimality of assignments for improved performance coming from relaxing the isolation of global assignment activities. Of course no loss of optimality occurs if the items belonging to a class are functionally equivalent with respect to assembly requirements. However, even if they are not, the loss of optimality can be controlled by imposing a limit on the number of such sub-optimal assignments. We will formalize the approach presented above in the remainder of this paper.

3. Commutativity in the framework of multi-level transactions

Hierarchical activities like those presented in section 2 can be modeled as nested transactions [14] or open-nested transactions [13, 1, 8]. In the open-nested

transaction model the effects of committed subtransactions may become visible to other concurrent transactions before their parent transaction commits. Relaxing the isolation properties of the conventional nested transaction model allows for more concurrency at the expense of more complicated recovery that may require compensating subtransactions.

Multi-level transactions are a special case of open-nested transactions where the sibling subtransactions in the transaction tree have the same nesting depth [23, 24]. A hierarchical activity modeled as a multi-level transaction will have a layered structure of operations, where an operation corresponds to a subtransaction in the open-nested transaction model.

A multi-level hierarchy of operations with a root transaction T_i is a $N+1$ level tree with the operation o_i^N at the root and atomic data operations o_j^0 at the tree leaves. We use a superscript to denote a level to which an operation belongs. Let $O_j^L = \{o_1^{L-1}, \ldots, o_n^{L-1}\}$ be a set of operations issued by o_j^L. The operation o_j^L is a pair $\langle\ O_j^L\ ,\ \prec_L\ \rangle$, where $\prec_L$ is a partial order defined on operations from the set O_j^L. The exact set of level $L-1$ operations issued by a parent operation o_j^L is determined at run-time, since some operations may fail to achieve their goals and alternative operations may be issued by the parent.

Typically higher level operations can be easily assigned particular semantics, e.g. allocate an optimal line to a circuit or report the number of lines that are available for allocation. At the lower levels the operations lose their semantic properties, and at the level 0 they become atomic read/write operations. A multi-level transaction model may take advantage of *semantic commutativity* among operations at higher levels of the operation hierarchy.

Semantic commutativity is defined in terms of the state of a database and the return value of an operation. As in [4] we assume that a result of an operation P executed on the database in state s, denoted as $result(P, s)$, is a pair $\langle state(P, s),\ return(P, s)\rangle$, where $return(P, s)$ is a value returned by P and $state(P, s)$ is the state of the database after the execution of P. It is possible that for a specific state of the database an operation P is unable to achieve its goal. Then the operation *fails* (or *aborts*), leaving the state of the database unchanged, and returning undefined value. The operation P is *defined* in the state s if it does not fail, otherwise it is undefined. For example, the operation $Assign(X, a)$ executed while all items in the class X are already assigned, is not defined. A sequence of operations is defined if all operations in this sequence are defined.

In general two operations may or may not commute depending on the state of the database in which they are applied. Since we are interested in reasoning about the correctness of histories based only on the semantics of the operations in histories (i.e. to the context in which those operations are executed) we will introduce below two notions of commutativity that *do not depend* on the state of the database.

DEFINITION 1 *Two operations P and Q semantically commute* iff, *for each database state s the following two conditions hold:*

1. *The composition $P \bullet Q$ is defined in state s* iff *the composition $Q \bullet P$ is defined in state s.*

2. *If $P \bullet Q$ and $Q \bullet P$ are defined in state s, then $result(Q \bullet P, s) = result(P \bullet Q, s)$.*

This is essentially equivalent to the notion of *backward commutativity* introduced in [21]. We use the term semantic commutativity to emphasize that it is symmetric with respect to P and Q.

Multi-level serializability considers a history of operations at level 0 to be valid if the operations at each level L ($0 \leq L < N$) can be transformed into a serial history (with respect to the operations at level $L+1$) by a series of interchanges among the commuting operations at level L. The ordering of the operations at each level $L > 0$ is based on the serialization produced at level $L-1$, and an interchange is valid if the two operations semantically commute.

In our model we increase the possibility for interchanges between operations, introducing the notion of *left-to-right commutativity* or *LTR-commutativity.* In contrast to semantic commutativity, the LTR-commutativity relation may be non-symmetrical; it is possible to have P LTR-commutative with Q, but Q not LTR-commutative with P. That allows more operations to be interchanged, and, consequently, increases the concurrency in the system.

Definition 2 *Operation P is LTR-commutative with an operation Q* iff *for every state s of the database the following two conditions hold:*

1. *If the composition $P \bullet Q$ is defined in state s, then $Q \bullet P$ is defined in state s.*

2. *If $P \bullet Q$ is defined in state s, then $result(Q \bullet P, s) = result(P \bullet Q, s)$*

LTR-commutativity is related to the *right backward commutativity* defined by Weihl [22]. However, unlike Weihl we assume that *for every state of the database* two operations LTR-commute if the results of their execution are independent of their order.

One consequence of Definition 2 is that if P is LTR-commutative with Q, and if history H_1 (represented as a composition of operations)

$$H_1\colon\ H_p \bullet P \bullet Q \bullet H_s$$

is defined (H_p and H_s are subhistories of H_1), then the history

$$H_2\colon\ H_p \bullet Q \bullet P \bullet H_s$$

is also defined and the result of compositions of operations in histories H_1 and H_2 is the same; i.e., it is valid to transform H_1 into H_2. Note, however, that it may not be valid to transform H_2 into H_1. If P is semantically commutative with Q, then P is LTR-commutative with Q, and Q is LTR-commutative with P.

To illustrate the difference between LTR-commutativity and semantic commutativity let us consider the *Withdraw* and *Deposit* operations. Let's assume that s

Table 1. LTR-commutativity relation for the banking example.

	$Withdraw(i)$	$Deposit(i)$
$Withdraw(i)$	do not commute	commute
$Deposit(i)$	do not commute	commute

indicates the initial state of a bank account that represents the database state. The $Deposit(i)$ is defined as follows: for every state s of the database $s := s + i$. The $Withdraw(i)$ is defined in the following way: if $s > i$ then $s := s - i$ otherwise the operations fails and state s remains unchanged. The $Withdraw$ operation is LTR-commutative with $Deposit$, but $Deposit$ is not LTR-commutative with $Withdraw$. This is because of the fact that, for every state of the database, if the history $Withdraw \bullet Deposit$ is defined then also the history $Deposit \bullet Withdraw$ is defined and both histories give the same result. However the reverse does not always hold. For example, consider the following histories:

$$(s = 100)\ H1 : Deposit(100) \bullet Withdraw(150)$$
$$(s = 100)\ H2 : Withdraw(150) \bullet Deposit(100).$$

For the initial state of the database $s = 100$, the history $H1$ gives a different result than the history $H2$, since $Withdraw$ in H_2 fails. The LTR-commutativity relation for this example is illustrated in Table 1.

Below, we will introduce definitions of *LTR-equivalence* of histories and *LTR-conflict* between operations. These definitions are based on the concept of LTR-commutativity, and therefore LTR-equivalence and LTR-conflicts are not symmetric.

DEFINITION 3 *An operation o_i^L LTR-conflicts with an operation o_j^L in a history H* iff *o_i^L precedes o_j^L in the history and o_i^L is not LTR-commutative with o_j^L.*

DEFINITION 4 *Transformation of history H_k containing two consecutive operations o_i^L and o_j^L into history H_l where order of these two operations is reversed is an* LTR-commutative interchange iff *operation o_i^L does not LTR-conflict with operation o_j^L.*

DEFINITION 5 *The history H_i is LTR-equivalent to the history H_j* iff *H_i can be transformed into H_j by means of a series of LTR-commutative interchanges.*

We are now ready to define our relaxed correctness criteria. First we use the notion of LTR-commutativity to extend the set of allowable execution histories. It should be noted that level 0 read and write operations that access the same object are not LTR-commutative (in this case the notion of LTR-conflict reduces to the traditional notion of read/write and write/write conflict). Hence, no relaxing of

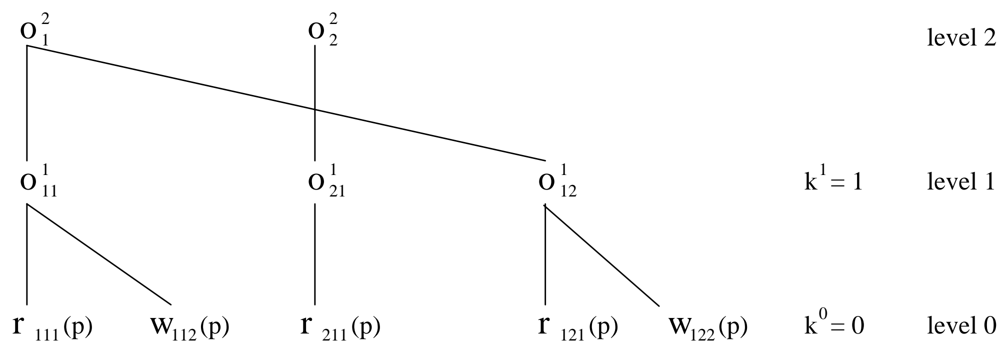

Figure 1. LTR-conflicts in a three level schedule.

serializability is possible on level 0 since even one invalid interchange at this level may produce unpredictable results. This is not the case at the higher levels of the hierarchy of operations. As we have shown in our example involving assignment and deassignment operations, allowing for non-LTR-commutable exchange may result in a loss of optimality of assignments or even a failed assignment but not in erroneous assignments resulting in e.g., allocating the same item to multiple assemblies.

DEFINITION 6 *A multi-level history H is LTR-serializable* iff *the history at each level L ($0 \leq L < N$) is LTR-equivalent to a serial history with respect to the operations at level $L+1$. The LTR-serialization order of operations at each level $L < N$ determines the LTR-serialization order of their parent operations at level $L+1$.*

DEFINITION 7 *A history H_L of operations at level L is k-serializable at level L* iff *it can be transformed into a serial history with respect to the operations at level $L+1$ by a series of interchanges of operations at level L such that each operation is involved in no more than k non-LTR-commutative interchanges.*

DEFINITION 8 *A history H of operations is multi-level ($k^0, \ldots, k^{N-1}$)-serializable (often abbreviated as multi-level K-serializable, or just K-serializable)* iff *it is k^L-serializable at every level L ($0 \leq L < N$), where as before the initial order of the operations at each level $L > 0$ is determined by the k^L-serialization order at level $L-1$.*

In most situations, it is inappropriate to allow non-serializable conflicts at level 0, since this may result in corrupted data and unpredictable results. In contrast, richer semantics of higher level operations may allow setting $k^L > 0$ for $L > 0$. Increasing k^L results in increased concurrency in the system coming from relaxing the isolation of operations at level $L+1$.

Consider the three level schedule in Figure 1. If we assume that operation o^1_{11} is not LTR-commutative with operation o^1_{21} and that operation o^1_{21} is not LTR-commutative with operation o^1_{12}, then at level 1 we have LTR-conflicts between o^1_{11} and o^1_{21}, and between o^1_{21} and o^1_{12}. At level 0 the history is serializable with respect to the operations at level 1. Therefore the history is multi-level (k^0, k^1)-serializable, with $k^0 = 0$ and $k^1 = 1$.

4. K-compensable histories

In this section we will discuss compensating operations in the context of hierarchical activities. In the nested transaction model [14], the effects of a child operation are revealed to other concurrently running transactions only after the parent operation commits. If blocking concurrency control mechanisms such as 2PL are used, then no other transaction can access the data items until the root transaction commits. The full isolation provided in this model may not be appropriate for long-running hierarchical activities in situations when blocking data for a long time may not be acceptable. In such situations, it may be desirable for subtransactions to commit and reveal their results before their parent commits. The consequence of this fact is that compensating operations must be used.

Our objective is to develop a compensation model for hierarchical activities in which we can reason about correctness of a history with compensating operations knowing *only the semantics of operations* in the history. Our compensation model does not take into account other semantic information, such as the the state of the database. We will use *state-independent* compensation in which a compensating operation undoes a compensated-for operation regardless of the state of the database. For example: in every state of the database deassignment $D_1(x_1, a)$ compensates assignment $A_1(X, a)$ (that allocated x_1 to a), $Withdraw(50)$ compensates $Deposit(50)$ regardless of the initial balance. We focus on the hierarchical multi-level activities. To deal with the problems of intervening transactions, we limit the scope of compensation by introducing the notion of *horizon of compensation*. The compensation of an operation is allowed only within its horizon of compensation.

Following the ideas in [23] we assume that in hierarchical multi-level activities, a child operation can be compensated only before its parent operation commits. The compensating operation can be issued by a parent operation and is a part of the parent operation activity. When a parent operation commits, we no longer can compensate a child operation. If compensation is needed after that, the entire parent operation must be compensated. Thus, the horizon of compensation for a child operation is limited by the commitment of the parent operation. Having such a horizon of compensation can be very valuable, as it limits the circumstances under which we must be concerned about another transaction seeing the effects of an operation which may later be compensated.

We adopt this model of compensation for any level $L \leq N$ of a multi-level hierarchy. To make it possible, we have to add an extra level $N + 1$ that represents a parent of all root transactions. The activity on this new level never commit-

s, and thus all root transactions can be compensated the same way as any other operations[3].

In our proposed model we assume that an operation that has been issued may be in one of three states: *uncommitted*, *committed but subject to compensation*, and *committed and non-compensable*. Unlike the nested transaction model, in which the visibility of the results of a committed subtransaction is restricted to the parent subtransaction and siblings, in our model the results of a committed operation become visible to all other concurrently executing operations.

Definition 9 *An operation o_i^L of level L is in the* uncommitted *state if it has been issued by its parent operation o_m^{L+1} at level $L+1$, but it has not been committed.*

Definition 10 *An operation o_i^L of level L is in the* committed but subject to compensation *state if it has been already committed but its parent operation o_m^{L+1} has not committed yet.*

Definition 11 *An operation o_i^L of level L is in the* committed and non-compensable *state if it has been committed and its parent operation o_m^{L+1} has already committed.*

Let co_i^L be a compensating operation for an operation o_i^L. If for a given operation o_i^L there is no compensating operation co_i^L then the commitment of o_i^L before its parent operation commits is not allowed and the operation o_i^L may be either uncommitted or committed and non-compensable. For a given operation o_i^L only its parent operation o_m^{L+1} may invoke a compensating operation co_i^L, if such an operation exists. Once the parent operation o_m^{L+1} commits, the only way to undo the effects of the operation o_i^L is to compensate the entire operation o_m^{L+1} by means of a compensating operation co_m^{L+1}.

Let us consider the three level schedule in Figure 2. Figure 2a) shows level 0 operations o_{111}^0 and o_{112}^0 that have been committed already. Since their parent operation o_{11}^0 has also been committed, both operations are in the "committed and non-compensable" state. The operation o_{121}^0 has been already committed and is in the "committed but subject to compensation" state (assuming that a compensating operation co_{121}^1 is available), since its parent operation o_{12}^0 has not committed. The operation o_{122}^0 is in the "uncommitted" state. At level 1 the operation o_{11}^1 is in the "committed but subject to compensation" state while the operation o_{12}^1 is in the "uncommitted" state.

Figure 2b) shows the actions required to abort the root operation o_1^2 (transaction T_1) at this point. At level 0, the operation o_{122}^0 is aborted, since it was in the "uncommitted" state, and the operation o_{121}^0 is compensated, since it was in the "subject to compensation" state. At level 1 the operation o_{12}^1 is aborted, since it was in the "uncommitted" state, and the operation o_{11}^1 is compensated, since it was in the "subject to compensation" state. Notice that the schedule does not show explicitly compensation of the operations o_{111}^0 and o_{112}^0. This is because of the fact that those operations have been in the "committed and non-compensable"

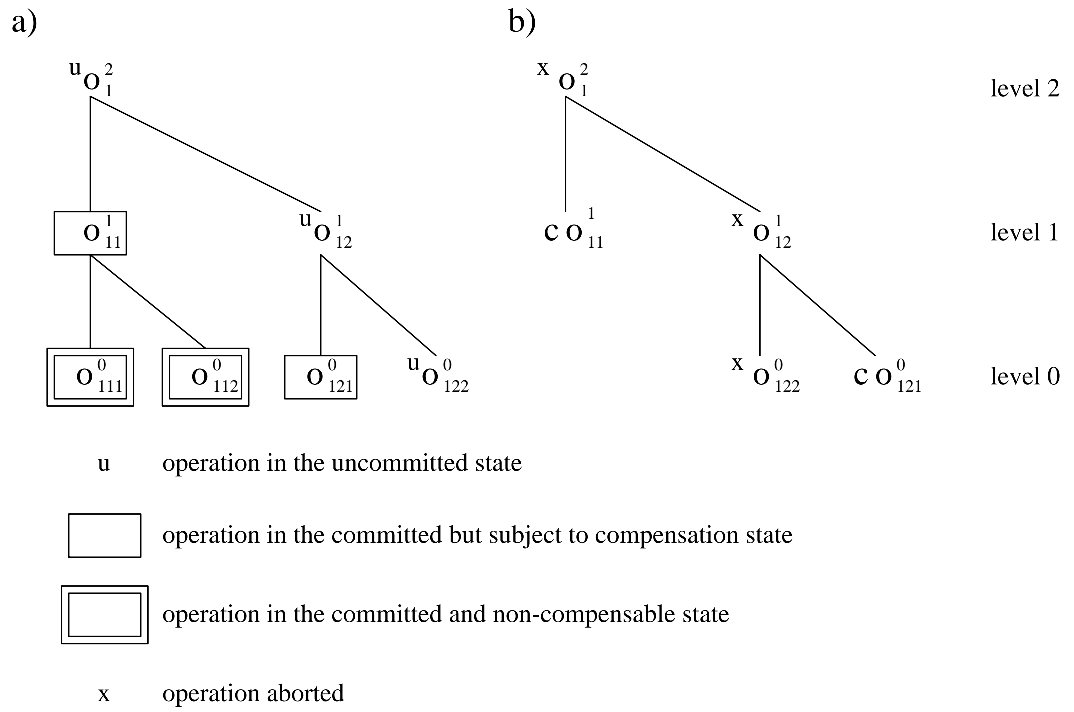

Figure 2. Compensation in three level schedule after a root transaction aborts.

state and their effects are undone by compensating their parent operation o^1_{11}. The compensating operation co^1_{11} can reverse operation o^1_{11} by issuing operations that do not necessarily undo operations o^0_{111} and o^0_{112}. This may be possible even if compensating operations for o^0_{111} and o^0_{112} are not provided. This example illustrates the fact that certain lower level operations may not be compensable while it is possible to compensate their ancestor operations.

Below we use the concept of an *augmented operation* to formally define *K-compensability* of histories – a proposed correctness criterion for compensation in hierarchical activities. The idea of an augmented operation at level L is to combine into one hypothetical operation ao^L_i a given operation o^L_i and its compensating operation co^L_i if both appear in the history of operations at level L.

Definition 12 *An augmented operation ao^L_i in a history H is a pair $\langle o^L_i, co^L_i \rangle$. If co^L_i is not in the history H then the second element of the pair is empty.*

For example, let us consider history H_1 at level 1 (in a three level hierarchy), where operation o^1_{31} is not commutative with any of the operations o^1_{11}, co^1_{11} or o^1_{21}, co^1_{21}, but o^1_{21} and co^1_{21} LTR-commute with co^1_{11}:

$$H_1 : o^1_{11} \bullet o^1_{21} \bullet o^1_{31} \bullet co^1_{21} \bullet co^1_{11}.$$

In history H_1, there are three augmented operations: ao^1_{11} (pair $\langle o^1_{11}, co^1_{11} \rangle$), ao^1_{21} (pair $\langle o^1_{21}, co^1_{21} \rangle$), ao^1_{31} (equivalent to o^1_{31} since co^1_{31} is not present in the history).

Let us consider a history H'_1 derived from the history H_1 by removing operation o^1_{31}:

$$H'_1 : o^1_{11} \bullet o^1_{21} \bullet co^1_{21} \bullet co^1_{11}.$$

History H'_1 is LTR-serializable with respect to augmented operations ao^1_{11} and ao^1_{21} since it can be shown equivalent to the serial history with augmented operations: $ao^1_{11} \bullet ao^1_{21} = o^1_{11} \bullet co^1_{11} \bullet o^1_{21} \bullet co^1_{21}$. Note that the interchanges of an execution order are not applied to augmented operations. Rather, the history H'_1 is converted to a serial history over augmented operations by moving co^1_{11} ahead of o^1_{21} and co^1_{21} based on LTR-commutativity.

History H_1 is not LTR-serializable with respect to the augmented operations ao^1_{11} and ao^1_{21} and ao^1_{31} because o^1_{31} does not LTR-commute with any other operation in the history. However, H_1 can be transformed into a serial history with respect to the augmented operations ao^1_{11} and ao^1_{21} and o^1_{31} by $k^1 = 2$ non-LTR-commutable interchanges (between o^1_{31} and co^1_{21}, and between o^1_{31} and co^1_{11}) and the interchanges described above that transformed the LTR-serializable history H'_1 into a serial history with augmented operations. Therefore the history H_1 is $k^1 = 2$ serializable at level $L = 1$ with respect to augmented operations ao^1_{11}, ao^1_{21}, and ao^1_{31}.

Definition 13 *A history with compensating operations is k^L-compensable at level L* iff *it is k^L-serializable at level L with respect to all augmented operations ao^L_i.*

According to the Definition 13 the history H_1 is $k^1 = 2$ compensable at level $L = 1$ with respect to the augmented operations ao^1_{11}, ao^1_{21}, and ao^1_{31}.

The horizon of compensation concept allows scheduling an operation o^L_j after any number of operations o^L_i that are in committed and non-compensable state, with no violation of multi-level serializability. Moreover, o^L_j can be scheduled after any number of operations o^L_i that are in committed but subject to compensation state, as long as either o^L_j is LTR-commutative with every co^L_i or every o^L_i is LTR-commutative with o^L_j. If k^L-compensability is the correctness criterion, we can relax the commutativity requirements by allowing an operation o^L_j to be scheduled as long as the number of prior non-commuting operations o^L_i that are in the committed but subject to compensation state does no exceed k^L.

Earlier, we argued that the main problems with compensation are due to the fact that intervening transactions may see the results of transactions that get compensated later. Analyzing compensation in the context of hierarchical activities allowed us to define the horizon of compensation. This, in turn, provides a mechanism to restrict the side-effects of compensation. Having enough semantic knowledge about operations, we can predict the effects of possible compensation on concurrently executing intervening operations. We can impose a limit on side-effects of compensation that reflects a trade-off between increased concurrency and relaxed correctness criteria. K-serializability allows more concurrency than multi-level serializability, at the expense of relaxing consistency requirements. Moreover, it should be pointed out that the possibility of choosing different values for k^L at different levels of the hierarchy of operations allows considerable flexibility for imposing bounds on the effects of relaxing serializability.

On the higher levels more semantic knowledge about operations is available. On those levels it is possible to predict the effects of a non-serializable multi-level schedule for different values of $k^L > 0$. One can choose k^L to reflect the semantics of a particular application. On lower levels of the hierarchy the choice of $k^L > 0$ becomes more difficult since the weak semantics of operations on those levels may not allow prediction of the effects of non-serializable schedules. At level 0 the sequence of operations becomes a sequence of bare read and writes that makes the choice of $k^0 > 0$ in most cases inappropriate, since it would result in corrupted data.

5. K-compensability of assignments

The concepts described in Sections 3 and 4 can be applied to the practical problem of managing assignment/deassignment transactions in a multidatabase system, discussed in Section 2.

In Section 2 we have described the problems that can be caused by relaxing the isolation of the global assignment activities. If an item is assigned in a subtransaction of a global transaction, and if the assignment is later canceled by a compensating deassignment in another subtransaction of the same global transac-

tion, the overall effect is not the same as if the initial assignment had never been made. In the interim other global tasks may have made less desirable assignments because of the unavailability of the item.

However, one can control this effect by enforcing k-compensability at the level of the assignments and deassignments – i.e., by enforcing multi-level K-serializability with $k^0 = 0$ and $k^1 > 0$. The effect of setting $k^0 = 0$ is to ensure that the basic assignments and deassignments are atomic and isolated, so that there is no corruption of the database, for example no possibility of assigning the same item to two different assemblies. The effect of setting a value for $k^1 > 0$ can be viewed from several standpoints. From the standpoint of any particular assignment-deassignment compensating pair, it says that at most k^1 other operations have their choices reduced (as compared to what would have happened if the initial assignment had never taken place). From the point of view of any particular assignment, it says that at most k^1 other assignments which are still subject to compensation will have artificially deprived it of choices.

The choice of a particular value of k^L is highly dependent on the semantics of the applications. For the telecommunication example k^1 denotes possible "non-optimality" of assignments. For example, if $k^1 = 1$, the assignment operation may chose the second-best item, if $k^1 = 2$, it may be third-best etc. Therefore, the application designer should specify k^1 in such a way that the optimality criteria are satisfied. If he decide that the item assigned should always be among the best 10% of available items, the k^1 should have value of 0.1 * (average number of available items) etc.

Allowing K-serializability instead of requiring full multi-level serializability has the additional effect that assignments and deassignments of different global transactions are allowed to interleave in different orders at different sites. However, from an application standpoint, this is not a problem.

Of course, in practice there are other types of global activities besides just assignments and deassignments. Other important types are:

- Add items to inventory or delete items from inventory. In these cases adding items to the inventories at different sites are essentially independent from a concurrency control point of view, so there is no problem. However, the interleaving of the add operations and assignment operations should be controlled by K-compensability, since adding some items to the inventory may influence the choice made by the assignment. Similar considerations regard the delete operation.

- Generate report of available items. Since this is not a debit-credit type of situation, it is not important that the report may include partial results of some global tasks. However, it is important that it may read some items as assigned when the assignment will be canceled later by a compensating deassignment. The effect of this can be limited by enforcing K-compensability.

Implementation Considerations

A scheduler enforcing multi-level K-compensability can be implement as follows. We propose enforcing k^L on each level by the means of special purpose locks called *c-locks*. C-locks are managed independently for each level and have granularity corresponding to the semantics of operations on this level. For example, on level 0 (with operations of read/write type) the c-locks would operate on particular database objects such as specific telecommunication lines or switches. On level 1, with operations like assignment or deassignment, the c-locks would operate on collections of objects such as particular classes of lines or sets of equivalent switches.

For every accessed data item, or set of data items as appropriate for a given level L, a c-lock manager creates a *c-lock table*. This table has rows and columns indexed by the operations of this level, so that each entry represents an ordered pair $< o_r^L,\ o_c^L >$. There are two types of entries in the c-lock table. If o_r^L is LTR-commutative with o_c^L the entry is empty, otherwise it contains a c-lock list. A c-lock list is a list containing identifiers of operations that access the data, and is initialized to () (i.e., empty).

The c-lock manager adheres to the following protocol:

1. For every operation o_i^L submitted for execution the c-lock manager creates a c-lock table, or finds the appropriate c-lock table if it was created earlier.

2. For every entry in the column indexed by o_i^L (i.e., for every entry $< o_r^L,\ o_i^L >$) the c-lock manager calculates the number j_r of operations that are included in the c-lock list and have different parent than o_i^L. The sum $j = \sum j_r$ represents the total number of non-LTR-commutative operations that precede o_i^L and that are in the committed but subject to compensation state.

3. If $j = 0$, i.e., if there are no previous conflicting operations, $o_i^L$is allowed to proceed.

4. If $j > 0$, there are j conflicts. If $j \leq k^L$ the operation is allowed to proceed, otherwise it is aborted or blocked.

5. If o_i^L is allowed to proceed, it adds its identifier to all c-lock lists in its row (i.e., to all c-lock table entries $< o_i^L,\ o_c^L >$). In case the operation is aborted by the system, it must remove its identifiers from these lists.

6. When an operation commits, the identifiers of all its children operations are removed from the appropriate c-lock lists.

7. When a compensating operation co_i^L commits, the identifier of the compensated-for operation o_i^L is removed from the appropriate c-lock lists.

This algorithm assures that for each operation o_i^L the number of preceding operations that are not LTR-commutative and are still subject to compensation will never exceed k^L. Thus no operation can be involved in more than k^L non-LTR-commutative interchanges, and K-compensability can be preserved.

Note that for level 0 operations (read and write), and for and $k^0 = 0$, this algorithm reduces to strict 2PL. That assures the atomicity and isolation of the level 1 operations, and prevents possible inconsistency of the database state.

As we can see from the description, this algorithm executes several steps for each submitted operation o_i^L. The number of executed steps is proportional to the number of operations that are not LTR-commutative with o_i^L, and to the value of k^L. Since the number of non-LTR-commutative operations is predefined in the system, we can consider the complexity of this algorithm to be O(k) (i.e., linear).

This algorithm could be easily extended to maintain different values of k^L for each operation o_i^L. The only modification needed is a check in step 5 whether adding of the identifier of o_i^L to the c-lock list causes the sum j_c of any column c to exceed k_c^L assigned to it. If so, o_i^L is aborted or blocked, otherwise it may proceed. This modification would allow for greater flexibility of the system, since each kind of operations would have its own limits of possible inconsistency.

It may happen in some applications that it is not feasible or acceptable to block subtransactions. The concept of multi-level serializability and c-locks is still useful for estimating the impact of non-serializability on the application. In such situations, operations do not block or abort, but each operation records in the log the number of c-locked items which it encountered. This information can be used to give guidance on how much excess inventory should be maintained to allow for the expected number of items which are artificially unavailable at any given time because of assignments which end up getting compensated later in the same global transaction. This approach should be also adopted for the root level transactions, since their parent operation never commits, and therefore their c-locks are never removed. Alternatively, we can assume that all root transactions commute with each other.

6. Discussion

The notions of K-serializability and K-compensability are related in many ways to similar concepts discussed in the literature: backward and forward commutativity [21], approximate soundness [11], or ϵ-serializability [17]. In this section we discuss these relations in more detailed way.

It is clear that activities like those presented in section 2 can be modeled as nested transactions [14] or open-nested transactions [13, 1, 8]. In the open-nested transaction model the effects of committed subtransactions may become visible to other concurrent transactions before their parent transaction commits. Relaxing the isolation properties of the conventional nested transaction model allows for more concurrency at the expense of more complicated recovery that may require compensating subtransactions. Recovery issues in nested transactions have been thoroughly studied by Moss, Griffeth, and Graham [15]. They defined the notion of "abstract serializability" to determine which operations can be correctly undone and which have to be dealt with by compensating subtransactions.

Observation 1 *Abstract serializability by layers [15] is a correctness criterion strictly more restrictive than K-serializability.*

Abstract serializability by layers requires that a history of operations on every level of a multi-level transaction can be transformed into a serial history with respect to the operations on the higher level by the means of interchanging the order of *commutative* operations. K-serializability admits the possiblity of exchanging the order of LTR-commutative operations, but also allows for k^L *non-commutative* interchanges on each level L. That extends the class of admissible histories beyond abstract serializability.

Multi-level transactions are a special case of open-nested transactions where the sibling subtransactions in the transaction tree have the same nesting depth [23, 24]. A hierarchical activity modeled as a multi-level transaction will have a layered structure of operations, where an operation corresponds to a subtransaction in the open-nested transaction model. The multi-level transaction framework can be applied in a natural way to multidatabase systems, with levels corresponding to global transactions, local subtransactions, and local database operations [16, 24]. In our discussion we concentrate on hierarchical activities that can be described using the multi-level transaction model, but the model proposed in this paper can be generalized for the open-nested transaction model such as described in [1]. This generalization would be natural if the transactions, subtransactions, and basic data operations in an open-nested transaction system access the data contained in the database in the hierarchical manner assumed in our paper, i.e., if some transactions access items, some access classes of items (but not items), some superclasses of items (but again not items and not classes of items), etc. In this case the open-nested transactions can be mapped into multilevel transactions with some operations on some levels missing, i.e., with operations on level i that issue operations of level $i-2$ (or $i-3\ldots$) instead of operations from level $i-1$. Otherwise, when an open-nested transaction is allowed to access any data in the database, our model would not be really applicable.

Our notion of semantic commutativity corresponds to the *backward commutativity*, and LTR-commutativity to the *right backward commutativity* defined by Weihl [21, 22] in the context of recovery. However, unlike Weihl, we assume that *for every state of the database* two operations LTR-commute if the results of their execution are independent of their order. Weihl's notion of an operation include the result of the execution of the operation, and therefore the right backward commutativity relation is dependent on the state of the database.

Observation 2 *Backward commutativity relation [21] is strictly larger than semantic commutativity.*

Observation 3 *Right backward commutativity relation [21] is strictly larger than LTR-commutativity.*

These two observations are simple consequences of the fact that analysis of the database state yields more semantic information that can be used for determining the commutativity, and therefore allows finer control of commutativity. In consequence right backward commutativity allows more operations to commute than LTR-commutativity. Consider an example of a withdrawal operation. Two withdrawal operations do not LTR-commute. However, two withdrawal operations may right backward commute depending on the state of the database, e.g., if the amount of money on the account is such that both withdrawals are successful. The advantage of LTR-commutativity is that it is possible to decide whether two operations LTR-commute by examining the semantics of the operations without considering the state of the database in which those operations are executed. Moreover, right backward commutativity can be determined only *after* the operations are executed (and their results are known), whereas LTR-commutativity is determined statically.

The notion of multi-level K-serializability may appear similar to other correctness criteria based on non-serializable schedules. However, in many cases this similarity is misleading. Consider for example the original concept of ϵ-serializability introduced in [17, 25]. In our environment, where most transactions have to update the database, it is inappropriate to allow non-serializable conflicts at level 0 (postulated by ϵ-serializability), since this may result in corrupted data, and incorrect results. In contrast, richer semantics of higher level operations may allow setting $k^L > 0$ for $L > 0$. The *bounded inconsistency* [20] is an extension of ϵ-serializability to abstract data objects, and therefore to the operations on level $L > 0$. However, it is still not applicable in the environment where non-serializable (sub)transactions can update the database, or when the data items have only qualitative values ("item is assigned" vs. "item is not assigned").

Another seemingly similar notion is *bounded ignorance* [10]. However, it is a concept basically incomparable with K-serializability. It does not consider problems of commutativity of compensating transactions, but problems of missing updates in replicated databases. Bounded ignorance allows transactions to read out-of-date replicas of data within restrictions that at most N updates are missing. Note that in our environment this protocol would allow N transactions to assign the same item to N different assemblies, which is clearly in violation of our correctness criteria.

Compensation as a tool for recovery has been thoroughly studied by Korth, Levy and Silberschatz [11, 12]. The model of compensation presented in their papers assumes two level activities with read/write operations at level 0 and transactions at level 1. The discussion in [11] is based on the intuitively clear notion that if a compensating transaction CT_i immediately follows the transaction T_i it is supposed to compensate, then the result is as if T_i had never occurred. Problems may arise if intervening transactions are scheduled between T_i and CT_i. To deal with this problem, the authors introduced the notion of *soundness* of the history. If the intervening transactions semantically commute with the compensating transaction CT_i then the history involving T_i, the intervening transactions and CT_i is considered sound. To deal with the problem of intervening transactions that do not commute with CT_i, they proposed the notion of *approximate soundness* – a correctness cri-

terion for compensation that is based on semantic information about the state of the database in which the operations are applied.

Observation 4 *Approximate soundness [11] is a correctness criterion strictly less restrictive than k-serializability for any given k.*

Again it should be noted that we develop a compensation model for hierarchical activities in which we can reason about correctness of a history with compensating operations knowing *only the semantics of operations* in the history. Our compensation model does not take into account other semantic information, such as the the state of the database. We use *state-independent* compensation in which a compensating operation undoes a compensated-for operation regardless of the state of the database. This makes our correctness criterion more restrictive than those that use the semantic information based on the state of the database, but simplifies the reasoning about the correctness of the history, which in turn enables efficient implementation. This is not unlike the relationship between view serializability that allows for more concurrency and conflict serializability that has an efficient implementation.

7. Conclusions

This paper is motivated by a telecommunications application which involves creating circuits (assemblies) by assigning components to them, deleting circuits by deassigning their components, and generating reports on the available components. This general type of activity is common to other applications as well. The semantics of the application involves one-sided commutativity of operations, compensation of operations, and weak consistency requirements. (Assignments need not be 100% optimal, and reports need not be 100% accurate.)

The multi-level transaction model is extended to encompass LTR-commutativity, where operations in an execution history may be exchanged in one direction, but not necessarily in the other. It is then further extended to include *K-serializability*, in which a bounded number of conflicts are allowed at each level of the multi-level framework, with the bound depending on the level. This concept allows exploiting richer semantics at the higher levels (and hence larger values of k) than at the lower levels. As discussed in Section 3, this extension differs from ϵ-serializability.

Early commitment of subtransactions and the use of compensation to restore consistency is sometimes required in order to support high concurrency and/or to accommodate unilateral or optimistic commit mechanisms. The use of compensation can create problems when other transactions (or operations) access data updated by a committed transaction which may later be compensated. In general, as discussed in [KLS90], to determine the correctness of compensation it is not sufficient to know only the semantics of the operations. It is also necessary to have semantic information about the state of the database. Knowing such a context and utilizing it in the compensating transaction may be difficult. To formulate a

more limited but more easily supported compensation strategy, we introduced the notion of a *horizon of compensation.* This is defined by introducing an additional state of an operation called *committed but subject to compensation* and by tying the compensability at any level to the status of a parent operation. We also introduced the notion of k-compensability to bound the number of operations which can be affected by an operation which is still compensable. k-compensability provides a mechanism to limit side-effects of a compensation by restricting the number of conflicting transactions that could have seen the database state change made by a transaction that is later compensated.

Acknowledgments

We thank Nancy Griffeth, Narayanan Krishnakumar, and Jari Veijalainen for helpful comments on an earlier version of the paper.

Notes

1. Semi-autonomous local databases are systems that have been modified to facilitate multi-database operations, but such that their local operations and administration remain autonomous.
2. The result of a history refers to both the final state of a database and the return value of each operation. This problem will be discussed in greater detail in Section 3.
3. However, compensation of a root transaction usually results from a human intervention, and not from the need of aborting a higher-level operation.

References

1. C. Beeri, P.A. Bernstein, and N. A. Goodman. A model for concurrency control in nested transaction systems. *Journal of ACM*, 1(1), Jan. 1989.
2. P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
3. P. Bernstein, M. Hsu, and B. Mann. Implementing Recoverable Requests Using Queues. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1990.
4. B.R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. In *Proceedings of the ACM SIGMOD Conference*, 1987.
5. Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On Rigorous Transaction Scheduling. *IEEE Transactions on Software Engineering*, 17(9):954–960, September 1991.
6. H. Garcia-Molina and K. Salem. SAGAS. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1987, pages 249–259.
7. J.N. Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th International Conference on VLDB*, September 1981, pages 144–154.
8. S. Ben Hassen and M. Rusinkiewicz. Concurrency Control for Distributed Nested Transactions. In *Proceedings of the 12th International Conference of Distributed Computing Systems*, June 1992.
9. M. Hsu and A. Silberschatz. Persistent Transmission and Unilateral Commit. In *Proceedings of the 7th IEEE Conference on Data Engineering*, April 1991.
10. N. Krishnakumar and A. Bernstein Bounded Ignorance in Replicated Systems. In *Proceedings of the Symposium on Principles of Database Systems*, May 1992.

11. H. F. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the 16th International Conference on VLDB*, 1990.
12. E. Levy, H.F. Korth, and A. Silberschatz. An Optimistic Commit Protocol for Distributed Transaction Management. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1990.
13. N. Lynch. Multi-level Atomicity - a New Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems*, 8(4):485–502, December 1983.
14. J.E.B Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, MIT Press, Cambridge, MA, 1985.
15. E. Moss, N. Griffeth, and M. Graham. Abstraction in Recovery Management. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1986.
16. P. Muth, W. Klas and E.J. Neuhold. How to Handle Global Transactions in Heterogeneous Database Systems. In *Proc. 2nd Intl. Workshop on Research Issues in Data Engineering: Transaction and Query Processing*, 1992.
17. C. Pu and A. Leff. Epsilon-Serializability. Technical Report CUCS-054-90, Dept. of Computer Science, Columbia University, January 1991.
18. C. Pu and A. Leff. Autonomous Transaction Execution with Epsilon Serializability. In *Proceedings of 1992 RIDE Workshop on Transaction and Query Processing*, 1992.
19. H.-J. Schek, G. Weikum and W. Schaad. A Multi-level Transaction Approach to Federated DBMS Transaction Management. In *Proc. 1st Intl. Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*, 1991.
20. M. H. Wong and D. Agrawal. Tolerating Bounded Inconsistency for Increasing Concurrency in Database Systems. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, June 1992.
21. W. E. Weihl. Commutativity-based Concurrency control for Abstract Data Type. *IEEE Transactions on Computers*, 37(12):205–218, Dec. 1988.
22. W. E. Weihl. The Impact of Recovery on Concurrency Control. *Technical Report MIT/LCS/TM-382*, Laboratory for Computer Science, Massachusetts Institute of Technology Feb. 1989.
23. G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM TODS*, 16(1):132–180, March 1991.
24. G. Weikum and H. Schek. Concepts and Applications of Multilevel Transactions and Open-Nested Transactions. In A. Elmagarmid, editor, *Transaction Models for Advanced Database Applications*, chapter 13. Morgan-Kaufmann, February 1992.
25. K.L. Wu, P.S. Yu, and C. Pu. Divergence Control for Epsilon-serializability. In *Proceeding of the 8th IEEE Conference on Data Engineering*, February 1992.

Received Date
Accepted Date
Final Manuscript Date