The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment

Yoav Raz

Digital Equipment Corporation, 151 Taylor St. (TAY1), Littleton, Ma 01460

Abstract

Commitment Ordering (CO) is a serializability concept, that allows *global serializability* to be effectively achieved across multiple *autonomous Resource Managers* (RMs). The RMs may use different (any) concurrency control mechanisms. Thus, CO provides a solution for the long standing global serializability problem. RM *autonomy* means that no concurrency control information is shared with other entities, except *Atomic Commitment* (AC) protocol (e.g. *Two Phase Commitment - 2PC*) messages. CO is a necessary condition for guaranteeing global serializability across autonomous RMs. CO generalizes the popular *Strong-Strict Two Phase Locking* concept (S-S2PL; "release locks applied on behalf of a transaction only after the transaction has ended"). While S-S2PL is subject to *deadlocks*, CO exhibits *deadlock-free* executions when implemented as *nonblocking* (optimistic) concurrency control mechanisms.

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

1 Introduction

Distributed transaction management services are intended to provide coordination for transactions that span multiple *resource managers* (RMs).

A RM is a software component that manages resources under transactions' control. A *resource* is any medium with well defined *states* that are being modified and retrieved while obeying transaction's ("all or nothing") semantics (*atomicity*). This means that effects of failed transactions are undone, which requires that resources' states be *recoverable* (i.e. if a resource is modified by a transaction, the state it had when the transaction started can be restored before the transaction ends). A resource is typically (but not necessarily) a data item. The scope of any specific resource (e.g. granularity units, versions, or replications) is defined as a part of a RM's semantics. Examples of resource managers are database systems (DBSs), queue managers, cache managers, some types of management entities/objects (e.g. see [EMA], [OSI-SMO]) etc.

A RM may impose a certain property of the generated transaction *histories* (transaction event schedules) to guarantee correctness and certain levels of fault tolerance. However, the *global history*, i.e. the combined history of all the RMs involved, does not necessarily inherit such a property even if it is provided by all the RMs. The *serializability* (SER) property is an example. Serializability is the most commonly accepted general criterion for the correctness of concurrent transactions (e.g. see [Bern 87], [Papa 86]), and supported in most RMs. When transactions involve more than one RM, this property may be violated in general, unless special measures are taken, or certain conditions exist to guarantee it. This issue is dealt with, for example, in [Brei 90], [Brei 91], [Elma 87], [Geor 91], [Glig 85], [Litw 89] and [Pu 88]. [Weih 89] deals with the relationships between local and global serializability in the framework of abstract data types. Achieving global serializability with reasonable performance, especially across RMs that implement different concurrency control mechanisms, has been considered a difficult problem (e.g. [Shet 90], [Silb 91]).

Global serializability can be guaranteed, in principle, by several methods if the RMs involved share relevant concurrency control information. *Timestamp Ordering (TO)* is an example (e.g. [Bern 87], [Lome 90]). If all the RMs involved support TO-based concurrency control and share the same timestamps, then the entire system can exhibit a coherent behavior based on TO, which guarantees global serializability. However, this technology requires a certain RM synchronization as well as timestamp propagation, and is currently unavailable in heterogeneous environments. Another known method, based on *locking,* allows RM *autonomy.* We define a RM to be *autonomous* if it does not share any resources and concurrency control information (e.g. timestamps) with another entity (external to the RM), and is being coordinated (at the nonapplication level[1]) solely via *Atomic Commitment* (AC) protocols (to achieve global atomicity). Most systems that support distributed transaction services provide AC protocols and related interfaces. These protocols guarantee atomicity even in the presence of certain types of recoverable failures. It means that either a distributed transaction is *committed,* i.e. its effects on all the resources involved become permanent, or it is *aborted (rolled back),* i.e. its effects on all the resources are undone. The most commonly used atomic commitment protocols are variants of the *Two Phase Commitment* protocol (*2PC* - [Gray 78], [Lamp 76]). Examples are *Digital Equipment Corporation's Distributed Transaction Manager - DECdtm* ([DECdtm]), *Logical Unit Type 6.2* of International Business Machines Corporation ([LU6.2]), and the *ISO - OSI standard for Distributed Transaction Processing* ([OSI-DTP]). A well known *local* (i.e. local to each RM) concurrency control mechanism that together with AC guarantees global serializability is *Strong Strict Two Phase Locking* (S-S2PL; "release locks issued on behalf of a transaction only after the transaction has ended"). This fact has been known for several years, and has been the major correctness foundation for distributed transactions. Various technical documents about distributed transaction management (e.g. [OSI-CCR]) have mentioned it. The observation that local S-S2PL guarantees global serializability appears explicitly at least in [Pu 88], [Brei 90] and [Brei 91][2]. The disadvantage of this approach is that *all* the RMs involved have to implement S-S2PL based concurrency control, even if other types are preferable for some RMs.

In this paper we examine the relationships between histories of individual RMs and the global history that comprises them, and generalize the above observation. We define a history property named *Commitment Ordering (CO),* and show that guaranteeing it is a *necessary* and *sufficient* condition for *guaranteeing* global serializability under the conditions of RM autonomy. CO can be implemented as standalone serializability mechanisms as well as being incorporated with other concurrency control mechanisms. Since CO can be enforced solely by controlling the order of transactions' commit events, it can be combined with any other concurrency control mechanism without affecting the mechanism's resource access scheduling strategy. This allows selecting and optimizing concurrency control for each RM according to the nature of transactions involved. Enforcing CO does not require aborting more transactions than those needed to be aborted for global serializability violation prevention, which is determined exclusively by the resource access orders, and is independent of the commit orders. S-S2PL based RMs provide CO already, since S-S2PL is a special case of CO.

In summary, serializability of transaction histories across (any) different RM types, which may use different concurrency control mechanisms but provide the CO property, is guaranteed without any global coordination or services but AC. Thus, the CO solution is fully distributed.

Section 2 is an overview and reformulation of serializability theory, which provides the foundation for analyzing CO. Section 3 defines CO and describes its

[1] Typically, a RM is unaware of any resource state dependency with states of resources external to the RM, implied by applications. This is also true in the cases where RMs are coordinataed by *multi-database systems,* which provide applications with integrated views of resources.

[2] [Brei 91] uses the term *rigorousness* for S-S2PL. [Brei 91] also redefines CO (naming it *strong recoverability*) and uses it to show that applying S-S2PL locally guarantees global serializability. No algorithm for enforcing CO (beyond S-S2PL) is given there.

properties. Section 4 examines CO schedulers and presents generic CO algorithms. Section 5 deals with multi RM histories, atomic commitment, and relationships between local and global properties. Section 6 shows that CO is exactly the property required to guarantee global serializability across autonomous RMs. Section 7 provides a conclusion. This paper is an abridged version of [Raz 90].

2 Histories and their properties - an overview

This section summarizes and reformulates known concepts and results of concurrency control theory (see also [Bern 87]), as well as introducing some new concepts, as a foundation for the following sections. Some reformulation is required to express and prove results that follow.

2.1 Transactions and histories

A *transaction*, informally, is an execution of a set of programs that access shared resources. It is required that a transaction is *atomic*, i.e. either the transaction completes successfully and its effects on the resources become permanent[1], or all its effects on the resources are undone. In the first case, the transaction is *committed.* In the second, the transaction is *aborted.* Formally, we use an abstraction that captures only events and relationships among them, which are necessary for reasoning about concurrency control:

A *single RM transaction* T_i is a *partial order* of events (specific events within the above informally defined transaction).
The (binary, asymmetric, transitive, irreflexive) relation that comprises the partial order is denoted "$<_i$".

Remarks:

- $\text{event}_a <_i \text{event}_b$ reads: event_a *precedes* event_b (in T_i).
- The subscript i may be omitted when the transaction's identifier is known from the context.

The events of interest are the following[2]:

- The *operation* of *reading* a resource; $r_i[x]$ denotes that transaction T_i has retrieved (read) the (partial) state of the resource x.
- The *operation* of *writing* a resource; $w_i[x]$ means that transaction T_i has modified (written) the state of the resource x.
- *Ending* a transaction; e_i means that T_i has ended (has been either committed or aborted) and will not introduce any further operations.

A transaction obeys the following *transaction rules* (*axioms*):

- **TR1**
  A transaction T_i has exactly a single event e_i.
  A value is assigned to e_i: $e_i = c$ if the transaction is *committed*; $e_i = a$ if the transaction is *aborted.*
  Notation: e_i may be denoted c_i or a_i when $e_i = c$ or $e_i = a$, respectively.
- **TR2**
  For any operation $p_i[x]$ (either $r_i[x]$ or $w_i[x]$) $p_i[x] <_i e_i$

Two operations on a resource x, $p_i[x]$, $q_j[x]$ are *conflicting* if they are *noncommutative*, i.e. applying them in different orders results in two different states[3] of x.
A more restrictive[4] approach assumes them to be conflicting, if at least one of them is a write operation.

A *complete history* H *over* a set T of transactions is a partial order with a relation $<_H$ defined according to the following *history rules* (*axioms*):

- **HIS1**
  If T_i is in T and $\text{event}_a <_i \text{event}_b$ then $\text{event}_a <_H \text{event}_b$
- **HIS2**
  If T_i and T_j are in T then for any two conflicting operations $p_i[x]$, $q_j[x]$, either $p_i[x] <_H q_j[x]$ or $q_j[x] <_H p_i[x]$
- **HIS3**
  Let T_i, T_j be transactions in T, where $e_i = a$.
  If $w_i[x] <_H r_j[x]$ then either $e_i <_H r_j[x]$ or $r_j[x] <_H e_i$ (Without this rule a history's

[1] The term *permanent* is relative and depends on a resource's volatility (e.g. sensitivity to process or media failure).

[2] More event types such as locking and unlocking may be introduced when necessary.

[3] We deal with states informally only. State distinction, and thus operation commutativity, may depend on the RM's semantics.

[4] Two write operations on the same resource may commute, e.g. *increment* and *decrement* of a *counter*.

semantics (as reflected by resource states) is not uniquely determined, since if $e_i = a$ the effect of $w_i[x]$ is undone; i.e. reading x after e_i results in retrieving the last state of x that was written by other (unaborted) transaction than T_i.)

Remarks:

- The subscript H in $<_H$ may be omitted when H is known from the context.
- The graphic symbol $\rightarrow_H$ may be used instead of $<_H$ when convenient.
- $<_H$ may be omitted for total orders, i.e., a history may be represented by an event sequence.

For modeling executions with incomplete transactions, we define a *history* to be any *prefix*[1] of a complete history.

2.2 On guaranteeing a property

In the following sections we examine conditions for *guaranteeing* that a system (any collection of interacting components or objects) generates histories with certain properties. This concept is formalized as follows:

Definition 2.1

Let S_A be the set of all reachable states of a system A.

The system A *guarantees* a property P, if every state in S_A has property P.

§

We concentrate on the case where a system's state is a history generated by the system[2].

2.3 History classes

Remark: A property's acronym is also used as the name for the class of all histories with this property.

2.3.1 Serializability

Transaction T_2 *is in a conflict with* transaction T_1 if $p_1[x] < q_2[x]$ for respective conflicting operations $q_2[x], p_1[x]$.

The conflict types are *ww, wr, rw,* when $p_1[x]$, $q_2[x]$ are write-write, write-read, and read-write, respectively.

Remark: Note the asymmetry in the definition above.

There is a *conflict equivalence* between two histories H and H' (the two are *conflict equivalent*) if they are both defined over the same set of transactions T, and consist of the same transaction events (for partially executed transactions), and

$p_i[x] <_H q_j[x]$ if and only if $p_i[x] <_{H'} q_j[x]$

for any *conflicting* operations $p_i[x], q_j[x]$ of any *committed* transactions T_i, T_j, respectively, in T (i.e. H and H' have the same conflicts between operations of committed transactions).

A history H over a transaction set T is *serial*, if for every two transactions T_i, T_j in T all the operations and the end of one of them precede all the operations and the end of the other (i.e., if $p_i[x] <_H q_j[y]$ or $e_i <_H q_j[y]$ then for any operations $s_i[u], t_j[v]$ in H, $s_i[u] <_H t_j[v]$, and $e_i <_H t_j[v]$).

The *commit projection* of a history H, is its *projection* (*restriction*)[3] on its set of committed transactions.

A history is *serializable* (SER; is in SER), if its commit projection is conflict equivalent to some serial history.

Transaction states (in addition to *committed* and *aborted*) are defined as follows:

A transaction is *decided*, if it is either *aborted* or *committed;* otherwise, it is *undecided.*

An undecided transaction is *ready* if it has completed its processing, and is prepared either to be committed or aborted; otherwise it is *active.*

The following diagram defines the possible transitions between transaction states:

[1] A *prefix* of a partial order P over a set S is a partial order P' over a set $S' \subseteq S$, with the following properties:
If $b \in S'$ and $a <_P b$ then also $a \in S'$
If $a,b \in S'$ then $a <_P b$ if and only if $a <_{P'} b$

[2] The related state transition function has a history and an event set as arguments. Its values on a given history and its prefixes, or on a given event set and its subsets are compatible. Such a function is neither formalized nor explicitly used in this work.

[3] Let P be a partial order over a set S. A *projection (restriction)* of P on a set $S' \subseteq S$ is a partial order P', a subset of P, that consists of all the elements in P, involving elements of S' only.

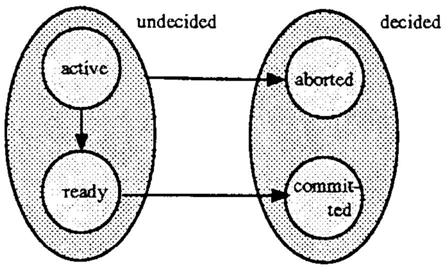


Figure 2.1: Transaction states and their transitions

The *Serializability Graph* of a history H, SG(H), is the following directed graph:

SG(H) = (*T*, *C*) where

- *T* is the set of all unaborted (i.e. committed and undecided) transactions in H
- *C* (a subset of *T*x*T*) is a set of edges that represent transaction conflicts: Let T_1, T_2 be any two transactions in *T*. There is an edge from T_1 to T_2 if T_2 is in a conflict with T_1.

The *Committed Transaction Serializability Graph* of a history H, CSG(H), is the subgraph of SG(H) with all the committed transactions as nodes and all the respective edges.

The *Undecided Transaction Serializability Graph* of a history H, USG(H), is the subgraph of SG(H) with all the undecided transactions as nodes and all the respective edges.

Theorem 2.1 provides a criterion for checking serializability:

Theorem 2.1 - The Serializability Theorem

A history H is *serializable* (in SER) if and only if CSG(H) is cycle-free.

(For a proof see, for example, [Bern 87]).

2.3.2 Recoverability

This section defines history properties that guarantee certain desired behavior patterns when aborts occur (see also [Bern 87]).

Recoverability is an essential property of histories when aborted transactions are present (i.e., in all real situations). Recoverability guarantees that *committed* transactions read only resource states written by committed transactions, and hence, no committed transactions read corrupted states. Recoverability also ensures that a serializable history has the same semantics (i.e., the history's outcome as reflected by the resources' states) as a conflict-equivalent serial history (when exists; e.g., for complete histories). This may not be true without recoverability, if aborted transactions are present.

Let T_1 and T_2 be two distinct transactions. We say that a transaction T_2 *reads* (a resource x) *from* (*in a read-from* conflict, or *wrf conflict with*; *wrf* is a special case of a *wr* conflict) transaction T_1 if T_2 reads x before T_1 is aborted (if aborted), and T_1 is the last transaction to write x before being read by T_2 (i.e., $w_1[x] < r_2[x]$ and there is no event t such that $w_1[x] < t < r_2[x]$, where t is either a_1 or $w_3[x]$ of some T_3).

It is required that for any two transactions T_1, T_2 in H, whenever T_2 reads any resource from T_1, aborting T_1 implies aborting T_2 (i.e., (T_2 *reads from* T_1) *implies* ($e_1 = a$ *implies* $e_2 = a$)). To guarantee this, T_2 should be decided only after T_1 has been decided (this is a necessary condition[1]). Thus, a history H is defined to be *recoverable* (REC; in REC) if for any two transactions T_1, T_2 in H, whenever T_2 reads any resource from T_1, T_1 ends before T_2 does ($e_1 < e_2$), and aborting T_1 implies aborting T_2.
Formally, (T_2 *reads from* T_1) *implies*
($e_1 < e_2$ *and* ($e_1 = a$ *implies* $e_2 = a$)).
The above formulation of recoverability allows it to be enforced effectively.

Aborts caused by transactions reading states written by aborted transactions (*cascading aborts*) are prevented if any transaction in H reads only data written by already committed transactions (i.e.,
(T_2 *reads from* T_1) *implies* $e_1 = c$). *Avoiding cascading aborts* (ACA; *cascadelessness*) is the property which is necessary and sufficient to guarantee the above condition[2]: H *is ACA* (in ACA), if for any two transactions

[1] The claim is proven by assuming the contrary, i.e., $e_2 < e_1$, and having T_2 committed, while T_1 is later aborted.

[2] Sufficiency is obvious. Necessity is proven by assuming $r_2[x] < e_1$ and having T_1 aborted.

T_1, T_2 in H, $\quad$ (T_2 *reads* x *from* T_1) *implies*
$\quad (e_1 = c$ *and* $e_1 < r_2[x]$).

Let T_1, T_2 be any two transactions in H. H is *strict* (ST; is in ST; has the *strictness* property) if
$w_1[x] < p_2[x]$ *implies* $e_1 < p_2[x]$,
where $p_2[x]$ is either $r_2[x]$ or $w_2[x]$.

Strictness simplifies the restoration of a resource's state after aborting transactions that have written that resource. The recovery procedures of most existing database systems rely on strictness.

Theorem 2.2 follows immediately from the definitions above:

Theorem 2.2 ([Bern 87])

REC $\supset$ ACA $\supset$ ST
where "$\supset$" denotes a strict containment.

2.3.3 Two Phase Locking

Two Phase Locking (2PL) is a serializability mechanism that implements two types of locks[1]: *write locks* and *read locks*. A write lock on a resource blocks both read and write operations of that resource, while a read lock blocks write operations only. 2PL consists of partitioning a transaction's duration to two phases: in the first, locks are acquired; in the second, locks are released ([Eswa 76]).

A history is defined to be a *2PL history* (it is in the class 2PL), if it can be generated by the 2PL mechanism.

When combining strictness (ST) with 2PL we get *Strict Two Phase Locking* (S2PL = ST$\cap$2PL). To enforce S2PL, *write locks* issued on behalf of a transaction are not released until its end. *Read locks*, however, can be released earlier, after the end of phase one of 2PL.

The property *Strong-S2PL* (S-S2PL) requires that all locks are not released before the transaction ends (either committed or aborted).

Formally: A history H is S-S2PL (in S-S2PL) if for any conflicting operations $p_1[x]$, $q_2[x]$ in H (of transactions T_1, T_2 respectively) $p_1[x] < q_2[x]$ implies $e_1 < q_2[x]$.

Theorem 2.3 summarizes the relationships among the 2PL classes:

Theorem 2.3

2PL $\supset$ S2PL $\supset$ S-S2PL

2.4 On inherently blocking and noninherently blocking properties

Some history properties can be enforced only by blocking mechanisms. A mechanism is *blocking*, if in some situations it delays some transaction's *event* until a certain *event*(s) occurs in some *other* transaction(s).

A mechanism is *operation-blocking*, if in some situations it delays a transaction's *operation* until a certain *event*(s) occurs in some *other* transaction(s), or aborts all transactions with blocked operations (to avoid operation-blocking, that otherwise would occur).

We define a history property to be *inherently-blocking*, if it can be enforced by operation-blocking mechanisms only. Otherwise it is *noninherently-blocking*.

Both serializability and recoverability are noninherently-blocking, since they can always be guaranteed by aborting a violating transaction any time before it ends, without having any operations blocked. This observation is the basis for *optimistic concurrency control* ([Kung 81]), where transactions run without blocking each other's operations, and are aborted before ending, if they violate serializability or any other desired property. 2PL, ACA and their special cases, on the other hand, are inherently-blocking.

Note that the mutual blocking of two or more transactions is the cause of *deadlock* situations. Thus, non-blocking mechanisms guarantee *deadlock-freeness*.

Remark: In this work we deal with blocking and deadlocks informally only.

2.5 On commit-decision delegation

In some situations the decision whether to commit or abort a ready transaction is delegated from one system (object, component) to another system (object, component) via a notification. This notification is denoted as a *YES vote on the transaction*.

Definition 2.2

Let transaction T be in a *ready* state. System A *delegates* the commit decision on a transaction T to system B by *voting* YES *on* T, if system A is prepared to either commit T or abort it, according to the decision taken by

[1] A lock is considered any mechanism that blocks resource-access operations.

system B. After voting YES system A cannot affect the decision anymore.

Remark: System A can abort transactions. It cannot vote YES on a transaction after aborting it.

Let y_i denote the YES voting event by system A on transaction T_i, and d_i the decision event that takes place in system B. d_i takes the values a or c, and may be denoted a_i or c_i respectively (when a distinction between e_i and d_i is clear by the context). The following *commit decision delegation (CDD) rules (axioms)* involving these events hold true:

- **CDD1**
  $p_i[x] < y_i$ for any operation $p_i[x]$ of T_i (i.e., all the transaction's operations are completed before voting YES on the transaction[1]).
- **CDD2**
  $y_i < d_i$ (i.e., when commit-decision delegation is applied, an explicit vote is required *before* any decision to commit or abort can be made).
- **CDD3**
  $d_i < e_i$ (i.e., the transaction is ended by system A only after being notified of the decision).
- **CDD4**
  $e_i = c$ if and only if $d_i = c$ (the obedience rule).
- **CDD5**
  event $< d_i$ implies event $< y_i$ for any event $\neq y_i$ in system A (i.e., all such precedence dependencies with d_i are through y_i).
- **CDD6**
  $d_i <$ event implies $e_i <$ event for any event $\neq e_i$ in system A (i.e., all such precedence dependencies with d_i are through e_i).

§

Note that CDD1,2,3 are consistent with TR2. CDD5,6 reflect that systems A and B interact through the voting mechanism only (to generate interaction history events).

In some situations, where dependencies exist between decision events of different transactions (see sections 4.4 and 5 below), the following condition is needed to guarantee such dependencies:

Definition 2.3

A system (object) that *delegates* commit decisions *obeys* the *commit-decision delegation, dependency condition (CD^3C) for transactions* T_1 and T_2, if it votes YES on T_2 only after commiting or aborting T_1, i.e., the following relationship holds true:

- **CD^3C**
  $e_1 < y_2$

§

Note the asymmetry in the definition above.

Theorem 2.3 summarizes the conditions for decision event dependencies.

Theorem 2.3

Let system A delegate the commit decision on transactions T_1 and T_2 to system B.

Then CD^3C for T_1 and T_2 (i.e. $e_1 < y_2$) is a necessary and sufficient condition for $\text{event}_1 < \text{event}_2$, when event_1 is y_1 or d_1 or e_1, and event_2 is y_2 or d_2 or e_2.

Proof:

Follows by CDD.

§

3 Commitment Ordering (CO)

Commitment Ordering (CO) is a property of histories that guarantees serializability. It generalizes S-S2PL. A history is *CO* if the order (<) of any two conflicting operations in any two *committed* transactions matches the order of the respective commit events.

After a transaction accesses a resource, S-S2PL blocks any conflicting operations on the resource until the end of that transaction. CO, on the other hand, allows access by conflicting operations, while using any access scheduling strategy. This allows CO to be implemented also in a nonblocking manner, which guarantees *deadlock-freeness*. The price for this, however, is the possibility of *cascading aborts* when recoverability is applied.

[1] Commiting T_i by system A *after* the decision is made may involve the completion of *write* operations that have been written before the voting to a temporary storage, and not to the resource itself. However, in such cases the resource is locked for *any* operation until the transaction ends, and thus CDD1 can be assumed also for this case.

Definition 3.1

A history is in CO if for any conflicting operations $p_1[x]$, $q_2[x]$ of any *committed* transactions T_1, T_2 respectively, $p_1[x] < q_2[x]$ implies $e_1 < e_2$.

Formally:

$(e_1 = c$ and $e_2 = c$ and $p_1[x] < q_2[x])$ implies $e_1 < e_2$.

§

We now show that CO implies serializability:

Theorem 3.1

SER $\supset$ CO

Proof:

(i) Let a history H be in CO, and let

$\ldots \rightarrow T_i \rightarrow \ldots \rightarrow T_j \rightarrow \ldots$ be a (directed) path in CSG(H). By the CO definition (definition 3.1) and an induction by the order on the path above, we conclude that $c_i < c_j$.

(ii) Now, suppose that H is not in SER.

By theorem 2.1 (without loss of generality) there is a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ in CSG(H), where $n \geq 2$.

First, let T_i and T_j in (i) be T_1 and T_2 above, respectively (consider an appropriate prefix of the expression representing the cycle above).

This implies by (i) that $c_1 < c_2$.

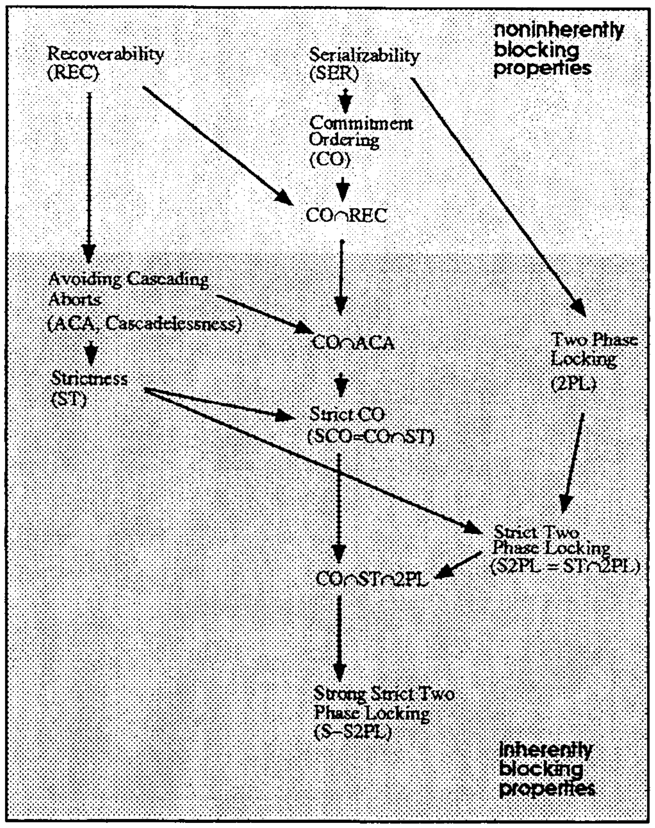
Now, let T_i and T_j in (i) be T_2 and T_1 above, respectively (consider an appropriate suffix of the expression representing the cycle above). This implies that $c_2 < c_1$. However, $c_1 < c_2$ and $c_2 < c_1$ contradict each other, since the relation "<" is asymmetric. Hence CSG(H) is acyclic, and H is in SER by Theorem 2.1.

Now examine the following serializable, non CO history to conclude that the containment is strict:

$r_1[x]\ w_2[x]\ c_2\ c_1$

§

The following diagram summarizes the containment relationships between history classes (some relationships are introduced here without proofs).



noninherently blocking properties
Recoverability (REC)
Serializability (SER)
Commitment Ordering (CO)
CO∩REC
Avoiding Cascading Aborts (ACA, Cascadelessness)
CO∩ACA
Two Phase Locking (2PL)
Strictness (ST)
Strict CO (SCO=CO∩ST)
Strict Two Phase Locking (S2PL = ST∩2PL)
CO∩ST∩2PL
Strong Strict Two Phase Locking (S-S2PL)
inherently blocking properties

Figure 3.1: Class containment relationships

An arrow from a class A to a class B indicates that class A strictly contains B; a lack of a directed path between classes means that the classes are incomparable.

A property is *inherently blocking* if it can be enforced only by blocking transaction's operations until certain events occur in *other* transactions.

4 Commitment Ordering schedulers

A *scheduler* is a RM's component that schedules certain transactions' events. *Commitment Ordering (CO) schedulers* are schedulers that generate CO histories. Generic mechanisms, that can be combined in various ways to implement CO schedulers are presented in the following sections. The algorithms described below provide additional (algorithmic) characterizations for the properties CO and CO∩REC.

4.1 Schedulers: components and classification

Schedulers typically deal with three types of transaction events:

- Transaction initiation
- Resource access
- Transaction termination

Concentrating on the latter two types[1], we model a (complete) scheduler as consisting of two components:

- Resource Access Scheduler (RAS)
  A component that manages the resource access requests arriving on behalf of transactions, and decides when to execute which resource access operation.
- Transaction Termination Scheduler (TTS)
  A competent that monitors the set of transactions and decides when and which transaction to commit or abort. In a multi RM environment this component participates in *atomic commitment* procedures on behalf of its RM and controls (within the respective RM) the execution of the decision reached via atomic commitment for each relevant transaction.

A scheduler component is *blocking* if it executes certain transaction's event requests only after certain events have occurred in some *other* transaction(s). Otherwise, it is *nonblocking*.

Nonblocking schedulers implement the so called *optimistic concurrency control* approach ([Kung 81]). When a scheduler is *nonblocking*, it provides *deadlock-free* executions.

4.2 A "pure" CO TTS - The Commitment Order Coordinator (COCO)

The following TTS type, the *Commitment Order Coordinator (COCO)*, checks for CO only and generates CO histories. The generated histories are not necessarily recoverable. Recoverability, if required, can be applied by enhancing the COCO (see section 4.2) or by an external mechanism (e.g., see section 4.4).

A COCO maintains a serializability graph, the USG, of all *undecided* transactions. Every new transaction processed by the RM is reflected as a new node in the USG; every conflict between transactions in the USG is reflected by a directed edge (an edge between two transactions may represent several conflicts).

USG(H) = *(UT,C)* where

- *UT* is the set of all undecided transactions in a history H
- *C* (a subset of *UT*x*UT*) is the set of directed edges between transactions in *UT*. There is an edge from T_1 to T_2 , if T_2 is in a conflict with T_1.

The set of transactions aborted as a result of committing a transaction T (to prevent a future commitment ordering violation) is defined as follows:

$\mathrm{ABORT}_{CO}(T) = \{ T' \mid \text{The edge } T' \to T \text{ is in } C \}$

These aborts cannot be compromized, as stated by lemma 4.1:

Lemma 4.1

Aborting all the members of $\mathrm{ABORT}_{CO}(T)$, after T is committed, is necessary for guaranteeing CO (assuming that every transaction is eventually decided).

Proof:
Suppose that T is committed. Let T' be some transaction in $\mathrm{ABORT}_{CO}(T)$. Thus T' is undecided when T is committed. If T' is later committed, then c < c', where c and c' are the commit events of T and T' respsctively. However, T is in a conflict with T', and thus, CO is violated.

§

Lemma 4.1 is the key for the CO algorithm.

Algorithm 4.1 - The CO Algorithm

Repeat the following steps:

- Select any transaction T in the USG in the *ready* state (using any criteria, such as by priorities assigned to each transaction; a priority can be changed dynamically as long as the transaction is in the USG), and commit it.

[1] A transaction is usually initiated as soon as computing resources are available, without considering the effect on the history's properties. Situations where an advantage can be taken of controlling initiation are ignored here.

- Abort all the transactions in the set $ABORT_{CO}(T)$, i.e. all the transactions (both *ready* and *active*) in the USG that have an edge going to T .
- Remove any *decided* transaction (T and the aborted transactions) from the graph (they do not belong in the USG by definition).

Remark: During each iteration the USG should reflect *all* operations' conflicts until commit.

§

Example 4.1

The following figure demonstrates one iteration of the algorithm:

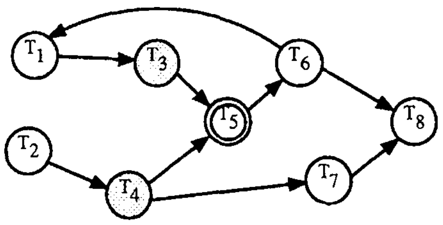


Figure 4.1: A USG

If T_5 is selected to be committed, then T_3 and T_4 are aborted; T_3, T_4 and T_5 are then removed from the graph.

§

The following theorem states the algorithm's correctness:

Theorem 4.1

Histories generated by a scheduler involving a COCO (algorithm 4.1) are in CO.

Proof:

The proof is by induction on the number of iterations by the algorithm, starting from an empty history H_0 , and an empty graph $USG_0 = USG(H_0)$. H_0 is CO.

Assume that the history H_n , generated after iteration n, is CO. USG_n (in its *UT* component) includes all the undecided transactions in H_n.

Now perform an additional iteration, number n+1, and commit transaction T_1 (without loss of generality - wlg) in USG_n. H_{n+1} includes all the transactions in H_n and the new (undecided) transactions that have been generated after completing step n (and are in USG_{n+1}).

Examine the following cases after completing iteration n+1:

- Let T_2, T_3 (wlg) be two committed transactions in H_n. If T_3 is in a conflict with T_2 then $c_2 < c_3$ since H_n is CO by the induction hypothesis.
- Obviously, $c_2 < c_1$ for every (previously) committed transaction T_2 in H_n with which T_1 is in a conflict.
- Suppose that a committed transaction T_2 is in a conflict with T_1. This means that T_1 is in $ABORT_{CO}(T_2)$, and thus aborted when T_2 was committed. A contradiction.

The cases above exhaust all possible pairs of conflicting committed transactions in H_{n+1}. Hence, H_{n+1} is CO.

§

By theorems 3.1 and 4.1 we conclude the following:

Corollary 4.1

Histories generated by a system that includes a COCO are serializable.

Note that aborting the transactions in $ABORT_{CO}(T)$ when committing T prevents any cycle involving T being generated in the CSG in the future. This observation is a direct way to show that the algorithm above guarantees serializability. If a transaction exists, that does not reside on any cycle in the USG, then a transaction T exists with no edges *from* any other transaction. T can be committed without aborting any other transaction since $ABORT_{CO}(T)$ is empty. If all the transactions in the USG are on cycles, at least one transaction has to be aborted when committing another one. If the COCO is combined with a scheduler (see also section 4.4 below) that guarantees (local) serializability, cycles in the USG are either prevented or eliminated by the scheduler aborting transactions.

In a multi-RM environment, a COCO decides when to vote YES on a transaction in an atomic commitment (AC) protocol, rather than deciding when to commit it. After a notification to commit has arrived, when committing a transaction, the actions taken by the COCO are the same as for a single RM environment. When using AC, delaying the voting or blocking it usually reduces the number of aborted transactions (see more in section 5 below).

4.3 The CO Recoverability Coordinator (CORCO)

A CORCO is a CO TTS generating histories that are both CO and recoverable. This TTS is an enhancement of the COCO (section 4.2 above), and differs from it only in processing additional information to guarantee recoverability, and thus, possibly aborting additional transactions, to prevent recoverability violations.

A CORCO maintains an enhanced serializability graph, *wrf*-USG:

wrf-USG(H) = $(UT, C \cup C_{wrf})$ where

- *UT* is the set of all undecided transactions in the history H.
- *C* is a set of edges between transactions in *UT*: There is a *C* edge from T_1 to T_2, if T_2 is in a conflict (conflicts) with T_1 but has not read from T_1.
- C_{wrf} is a set of edges between transactions in *UT* as well: There is a C_{wrf} edge from T_1 to T_2, if T_2 has read from T_1 (and possibly is also in conflicts of other types with T_1).

Note that *C* and C_{wrf} are disjoint.

The set of transactions aborted as a result of committing T (to prevent future CO violation) is defined as follows:

$ABORT_{CO}(T) = \{ T' \mid T' \rightarrow T \text{ is in } C \text{ or } C_{wrf} \}$

The above definition of $ABORT_{CO}(T)$ has the same semantics as the definition of $ABORT_{CO}(T)$ for the COCO.

The set of aborted transactions due to recoverability, as a result of aborting transaction T', is defined as follows:

$$ABORT_{REC}(T') = \{ T'' \mid T' \rightarrow T'' \text{ is in } C_{wrf} \text{ or } T''' \rightarrow T'' \text{ is in } C_{wrf} \text{ where } T''' \text{ is in } ABORT_{REC}(T') \}$$

Note that the definition is recursive. This reflects the nature of cascading aborts.

A CORCO executes the following algorithm:

Algorithm 4.2

Repeat the following steps:

- Select any *ready* transaction T in the *wr*-USG, that does not have any in-coming C_{wrf} edge (i.e. such that T is not in $ABORT_{REC}(T')$ for any transaction T' in $ABORT_{CO}(T)$; this avoids the need to later abort T itself), and commit it.
- Abort all the transactions T' (both *ready* and *active)* in $ABORT_{CO}(T)$.
- Abort all the transactions T" (both *ready* and *active)* in $ABORT_{REC}(T')$ for every T' aborted in the previous step (cascading aborts).
- Remove any *decided* transaction (T and all the aborted transactions) from the graph.

Remarks:
During each iteration the *wrf*-USG should reflect *all* operations' conflicts till commit.

Transactions on *wrf* cycles, that are not aborted by the algorithm, are aborted asynchronously.

§

Example 4.2

The following figure demonstrates one iteration of the algorithm:

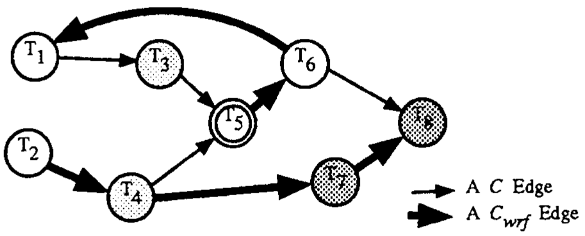


Figure 4.3: A *wrf*-USG

If T_5 is selected to be committed then T_3, T_4, T_7, T_8 are aborted; all committed and aborted transactions are then removed from the graph.

$ABORT_{CO}(T_5) = \{ T_3, T_4 \}$
$ABORT_{REC}(T_3) = \phi$ (empty set)
$ABORT_{REC}(T_4) = \{ T_7, T_8 \}$

§

The algorithm's correctness is stated as follows:

Theorem 4.2

Histories generated by a scheduler involving a CORCO (algorithm 4.2) are both CO and recoverable.

Proof:

The histories generated are CO by theorem 4.1, since a CORCO differs from a COCO only in possibly aborting additional transactions during each iteration (due to the recoverability requirement).

Since all the transactions that can violate recoverability (transactions in $\mathrm{ABORT}_{\mathrm{REC}}(T')$ for every aborted transaction T' in $\mathrm{ABORT}_{\mathrm{CO}}(T)$) are aborted during each iteration (i.e. transactions that read resources written by an aborted transaction before the abort), the generated histories are recoverable.

§

By theorems 3.1 and 4.2 we conclude the following:

Corollary 4.2

Histories generated by a system that includes a CORCO are both serializable and recoverable.

4.4 Combining a CO TTS with a RM's scheduler

The CO TTSs above can be combined with any RAS or a complete scheduler. When a CO TTS is combined with a scheduler, the scheduler delegates the commit decision (see section 2.5 above) to the CO TTS. If all the components are *non-blocking*, then also the combined mechanism is non-blocking, and hence ensures *deadlock-freeness*:

Corollary 4.3

There exist schedulers incorporating a COCO or CORCO that generate deadlock free executions only.

By Corollaries 4.1 and 4.2 the combined RAS (or scheduler) does not necessarily need to produce serializable histories in order to guarantee serializability, since the CO TTSs above take care of this.

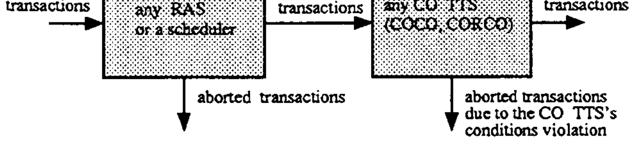


Figure 4.3
The commit/abort decision process using any CO TTS combined with a RAS or scheduler

The combined mechanism executes as follows (See Figure 4.3 above):

First, a transaction interacts with a RAS (or a scheduler). Then if unaborted, when ready, the transaction is considered by a CO TTS as a candidate to be committed. A transaction may be aborted to prevent a CO TTS's condition violation.

An important property of the CO TTSs is their total passivity with regard to resource access operations. The RAS (or the complete scheduler) can implement any resource access scheduling strategy without being affected by a combined CO TTS. The only requirement is that a CO TTS has updated conflict information (e.g. a USG or a *wrf*-USG).

When a scheduler delegates the commit decision, the following statements about recoverability hold true :

Theorem 4.3 - The Recoverability Inheritance Theorem

Let a system consist of a scheduler that guarantees *recoverability (cascadelessness, strictness)* and some *component* to which it delegates the commit decisions on all unaborted transactions. Let the scheduler vote YES on a transaction when it can be committed (by the scheduler) without violating *recoverability (cascadelessness, strictness)*. Then the above system *guarantees recoverability (cascadelessness, strictness* respectively*)* as well.

Proof:

Since the decision notification can arrive any time after voting, the scheduler has to guarantee the following condition in order to prevent a possible recoverability violation (see a definition of recoverability in section 2.3 above):
(T_2 *reads from* T_1) *implies*
($e_1 < y_2$ *and* ($e_1 = a$ *implies* $e_2 = a$))
Thus, if (T_2 *reads from* T_1) the scheduler does not vote YES on T_2 before T_1 is decided ($e_1 < y_2$) and hence $e_1 < e_2$ follows by rules CDD2,3. Since e_2 follows e_1, if $e_1 = a$ then (by the above condition) the scheduler can and has to enforce ($e_1 = a$ *implies* $e_2 = a$) by aborting T_2. Thus, (T_2 *reads from* T_1) *implies* ($e_1 < e_2$ *and* ($e_1 = a$ *implies* $e_2 = a$)) and recoverability is guaranteed. Note that CD^3C for T_1 and T_2 is maintained.

The cases ACA, ST are straightforward.

§

From theorem 4.3 we conclude corollary 4.4:

Corollary 4.4

If guaranteeing both CO and recoverability is required, a CORCO is necessary to be combined with a scheduler only if the scheduler does not guarantee recoverability by itself. Otherwise a COCO is sufficient.

Remark: Note that if the combined scheduler above is S-S2PL based, then the USG of the respective CO TTS does not have any edges. This means that no aborts by the CO TTS are needed, as one can expect, and the entire CO TTS is unnecessary. This is an extreme case. Other scheduler types may induce other properties of the respective USGs. No *wrf* conflicts are reflected in the USG when the scheduler guarantees cascadelessness; no *ww* and *wr* when it guarantees strictness.

Theorem 4.4 - The Recoverability Enforcement Condition

Let a system consist of a scheduler and some *component* to which the scheduler delegates the commit decisions on all unaborted transactions.

Then

- Guaranteeing CD^3C for any T_1 and T_2 such that T_2 is in a *read-from (wrf)* conflict with T_1 is a *necessary* condition for the system to *guarantee* recoverability.
- Guaranteeing CD^3C for any T_1 and T_2 such that T_2 is in a *wr* or *ww* conflict with T_1 is a *necessary* condition for the system to *guarantee* strictness (ST).

Proof:

(i) Suppose that recoverability is guaranteed and that CD^3C is not guaranteed for all T_1, T_2 such that T_2 has read from T_1. Thus there may exist T_1 and T_2 such that T_2 reads from T_1 , where the scheduler has voted YES on both transactions before T_1 is decided, violating CD^3C. Now suppose that T_1 is aborted by the deciding component and then T_2 is committed. This is a violation of recoverability, contrary to the assumption that the system guarantees recoverability.

(ii) Suppose that strictness is guaranteed. Let $w_1[x]$, $p_2[x]$ be conflicting operations of T_1, T_2 , respectively, where $p_2[x]$ is either a read or write operation. Due to strictness, $e_1 < p_2[x]$ (see section 2.3.2). Since $p_2[x] < y_2$ by CDD1 (see definitin 2.2), also $e_1 < y_2$ follows, which is CD^3C.

§

5 Multiple Resource Manager environment

5.1 The underlying transaction model

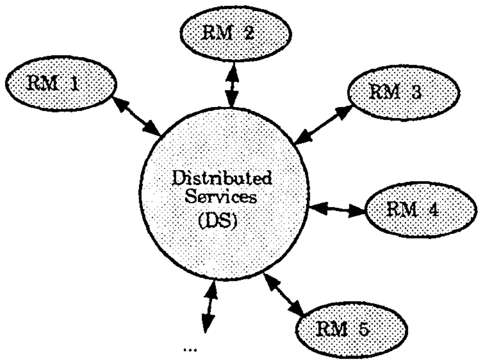


Figure 5.1: A multiple RM environment. RMs are invoked and coordinated through the DS

A multi RM *environment* consists of several (more than one) *autonomous* RMs, and a *Distributed Services (DS) system* component.

The DS component provides:

- Application (programs) execution environment
- Application communication services
- Application RM access services
- Transaction Management services (transaction demarcation, RM transaction participation registration, synchronization, atomic commitment, etc.)

No distinction is made between a centralized RM (i.e. confined to a single node) or a distributed one (i.e. executing and accessing resources on several nodes).

Note that the RMs' autonomy implies resource partitioning among the RMs.

The following terminology and notation are used as well:

- An *environment* is a DS and a set of RMs, where a transaction can span any subset. The RMs involved with a transaction are the *participants* in the transaction.

- Each RM in an environment has an *identifier* (e.g. RM 2).
- Events are qualified by both a transaction's identifier and a RM's identifier (e.g. $w_{3,2}[x]$ means a write operation of resource x by RM 2 on behalf of transaction T_3).

A multiple RM transaction is a generalization of a single RM transaction:

Definition 5.1

A transaction T_i consists of one or more *local subtransactions*[1].
A local subtransaction $T_{i,j}$ accesses *all* the resources under the control of a *participating* RM j, that T_i needs to access, and *only* these resources (i.e. all its events are qualified with j).

A local subtransaction obeys the definition of a transaction and rules TR1, TR2 in section 2.

A local subtransaction has *states* as defined in section 2.

A transaction T_i has an event d_i of *deciding* whether T_i is committed or aborted. d_i is usually distinct from any event $e_{i,j}$ of any local subtransaction $T_{i,j}$ and takes place in the DS component[2].

§

A distinction between an individual RM's history and the global history is required as well:

Definition 5.2

A *local history* is generated by a single RM, and defined over the set of its local subtransactions.

A local history obeys the definition of a history in section 2.

Notation: H_i is the history generated by RM i .

§

A *global history* obeys the definition of a history in section 2.

It is assumed that an *atomic commitment* (AC) protocol is applied to guarantee *atomicity* in the distributed environment. An AC protocol implements the following general scheme each time a transaction is decided:

- **AC**
  Each participating RM delegates the commit decision by voting either YES or NO (also absence of a vote within a time limit may be considered NO) after its respective local subtransaction has reached the ready state, or votes NO if unable to reach the ready state. The transaction is committed by all RMs if and only if all have voted YES. Otherwise it is aborted by all RMs.

Remarks:

- The YES vote is an obligation to end the local subtransaction (commit or abort) as decided by the AC protocol. After voting YES, a RM cannot affect the decision.
- After voting NO, a local subtransaction may be aborted immediately (thus a NO vote may be represented by the event $e_{i,j} = a$).
- Note that 2PC ([Gray 78], [Lamp 76]) is a special case of AC.
- The AC protocol type determines under what failure and recovery conditions atomicity is guaranteed (e.g. see [Bern 87]). No specific AC protocols are dealt with here.

AC enforces the following *atomic commitment rules (axioms)* in addition to the rules CDD (see section 2.5):

- **AC1**
  If $d_i = c$ then $y_{i,j}$ exists and $y_{i,j} < d_i$ for all local subtransaction $T_{i,j}$ (i.e. a transaction is decided to be committed only after receiving YES votes from all the local subtransactions).
- **AC2**
  If $d_i = c$ then $d_i < e_{i,j}$ for all local subtransaction $T_{i,j}$ (i.e. all local subtransactions are committed only after the AC protocol has decided to commit the transaction).

For environments that implement AC we conclude the following:

[1] Local subtransactions reflect transaction partitioning over RMs, and are independent of a possible explicit transaction's partitioning into nested subtransactions by an application.

[2] A *local* transaction is a transaction that consists of a single local subtransaction; a *global* transaction consists of two or more. Although a local transaction T_i can be decided locally by some RM j (and not necessarily in the DS), assume for convenience that $y_{i,j}$, d_i exist and obey CDD.

Lemma 5.1 - The Commit Fusion Lemma

Let T_1 and T_2 be any transactions decided via an AC protocol.
If $d_1 = c$ then

- event $< d_1$ if and only if event $< y_{1,j}$ for *some* j, for any event distinct from $y_{1,j}$ and d_1.
- $d_1 <$ event if and only if $e_{1,j} <$ event for *some* j, for any event distinct from $e_{1,j}$ and d_1.
- For event_1 and event_2 of T_1 and T_2 respectively, where event_i, i=1,2 is either $y_{i,j}$ or d_i or $e_{i,j}$, RM j *enforces*[1] $\text{event}_1 < \text{event}_2$ if and only if it enforces $e_{1,j} < y_{2,j}$ (i.e. CD^3C for $T_{1,j}$ and $T_{2,j}$; see also theorem 2.3).

Thus if $d_i = c$ then d_i and all the events $y_{i,j}$ and $e_{i,j}$, for every participating RM j, can be fused together into a single event without affecting order relations among other events.

Proof:

Follows by the rules CDD and AC.

§

The *commitment fusion lemma* allows in many situations a simplified transaction model to be used. This model ignors the AC mechanism and assumes that a (committed) multi RM transaction (like a single RM transaction) has a single commitment event. Note that this assumption is not valid for the abort events in a multi-RM transaction.

Example 5.1

The following two transactions both access resources x and y.
x, y are under the control of RMs 1, 2 respectively.

RM 1	$T_{1,1}$	$r_{1,1}[x]$	$e_{1,1}$	$T_{2,1}$	$w_{2,1}[x]$	$e_{2,1}$
RM 2	$T_{1,2}$	$r_{1,2}[y]$	$e_{1,2}$	$T_{2,2}$	$w_{2,2}[y]$	$e_{2,2}$
		T_1			T_2	

Figure 5.2: T_1 and T_2 and their local subtransactions

The RMs generate the following histories ($y_{i,j}$ events are omitted):

RM 1:	H_1	$r_{1,1}[x]\ w_{2,1}[x]\ c_{2,1}\ c_{1,1}$
RM 2:	H_2	$w_{2,2}[y]\ c_{2,2}\ r_{1,2}[y]\ c_{1,2}$

Figure 5.3: The local histories H_1 and H_2

Note that the history H_1 violates CO, which results in a (global) serializability violation.

The respective global history H is the following:

$$\begin{tikzcd}
r_{1,1}[x] \arrow[r] & w_{2,1}[x] \arrow[dr] & & c_{2,1} & & & c_{1,1} \\
& & c_2 \arrow[ur] \arrow[dr] & & & c_1 \arrow[ur] & \\
w_{2,2}[y] \arrow[urr] & & & c_{2,2} \arrow[r] & r_{1,2}[y] \arrow[ur] & & c_{1,2} \arrow[ul]
\end{tikzcd}$$

$$\text{CSG(H)} = \begin{tikzcd} T_1 \arrow[r, bend left] & T_2 \arrow[l, bend left] \end{tikzcd}$$

Figure 5.4: The history H and its CSG.

Since the transactions are committed, end-transaction events can be fused with respective decision events (by the fusion lemma).

§

5.2 Local and global properties

This section examines the relationship between properties of histories generated by the individual RMs and the global history generated in the environment. History properties are redefined in a way that allows definitions and results for single RM histories also to be used in the multi-RM case.

Definition 5.3 - Global Properties

Let X be a history property, well defined for a single RM's histories. Let X', the *associated property of* X, be a (global) history property defined by a modified definition of X, where every event of ending a *committed* transaction (e_i) in the definition of X is replaced with the respective commit-decision event (d_i).

A global history H has property X (is in X) if property X' is well defined and H has property X' (i.e. H obeys

[1] "Enforces" means that actions taken by RM j generate events that reflect or imply such an order.

the definition of X').

§

Remarks:

- Let X be a history property. In what follows, X also denotes the class of global histories with property X.
- Note that if a property X does not impose constraints on the order of events e_i, then the definitions of X and X' are identical (e.g. serializability).
- Note that when the fusion lemma (5.1) is implemented, the properties X and X' are identical.

Example 5.2

CO', the associated property of CO, is defined as follows:

A history is in CO' if for any conflicting operations $p_{1,j}[x]$, $q_{2,j}[x]$ of any *committed* transactions T_1, T_2 respectively, $p_{1,j}[x] < q_{2,j}[x]$ implies $d_1 < d_2$. Thus if a (global) history is in CO', it is also in CO by definition 5.3.

§

The following is a representative example for a result of definition 5.3 :

Corollary 5.1

SER $\supset$ CO (i.e. theorem 3.1 is valid also for global history classes).

The other class containment releationships described in section 3 follow similarly for global history classes.

By lemma 5.1 and the definition of CO we also conclude the following:

Theorem 5.1 - The Global CO Enforcement Condition

- Let H be a global history. Then CD^3C for any *committed* $T_{1,j}$ and $T_{2,j}$ (i.e. $e_{1,j} < y_{2,j}$), such that $T_{2,j}$ is in a conflict with $T_{1,j}$, is a *necessary* and *sufficient* condition for H to be in CO.

and thus

- Guaranteeing CD^3C for any $T_{1,j}$ and $T_{2,j}$, such that $T_{2,j}$ is in a conflict with $T_{1,j}$, and $y_{1,j}$ exists already, is a *necessary* and *sufficient* condition for guaranteeing CO.

(Without guaranteeing CD^3C, CO may be violated, and vise versa.)

We now define properties of global histories that reflect properties of their respective local histories:

Definition 5.4 - Local-X

Let H be global history generated in some environment, and H_j its respective local history generated by RM j in the environment.

H'_j, the *augmented* history of a local history H_j, is a history defined over the local subtransactions of RM j, where each local subtransaction $T_{i,j}$ is augmented with the event d_i.

Let X be a history property, well defined for single RM histories, and X' the associated property of X (see def. 5.3). H is in *Local-X* (is *locally* X) if H'_j of every RM j in the environment is in X',

§

Usually (e.g. for all the history properties that we have explicitly defined) Local-X $\supseteq$ X , i.e. if a global history is in X, it is in Local-X. In particular, the following relationships exist as well:

Theorem 5.2

Local-X = X (i.e. a (global) history is in X if and only if it is in Local-X),
where X is any of the following properties:

- REC, ACA, ST, CO, S-S2PL

Proof outline:

The theorem follows by the definitions of global properties (5.3), Local-X (5.4), the commitment fusion lemma (5.1), the RMs' resource partitioning, and the definitions of REC, ACA, ST, CO and S-S2PL.
A proof is demonstrated below for the case X = CO (without using the fusion lemma; it is trivial with the lemma):

Let H be a global history and H_j its respective local history of RM j, and let H'_j be the augmented history of H_j. With out loss of generality, let T_1 and T_2 be committed transactions in H.
Let H be in CO. Suppose that for some RM j , $T_{2,j}$ is in a conflict with $T_{1,j}$. This means that T_2 is in a conflict with T_1 , which implies $d_1 < d_2$ (CO and definition 5.3). Thus for every RM j , H'_j is in CO' (see definition 5.4) and H is in Local-CO.
Now let H be in Local-CO. If T_2 is in a conflict with T_1 , it means that for some RM j , $T_{2,j}$ is in a conflict with

$T_{1,j}$. Since H'_j is in CO' (Local-CO; definition 5.4) this implies that $d_1 < d_2$. Thus H is in CO (definition 5.3).

§

Theorem 5.3

Local-X $\supset$ X (i.e. being in Local-X does not imply that a history is in X),
where X is any of the following properties:

- SER, 2PL, S2PL

Proof:

Local-X $\supseteq$ X is true for SER (using theorem 2.1) and 2PL (follows by definition).
It is also true for strictness:
If H is a global history in ST then ($w_{i,j}[x] < p_{k,j}[x]$ implies $d_i < p_{k,j}[x]$) for any relevant RM j by definition 5.3. Thus by definition 5.4 H is in Local-ST.
Thus the containment is also true for S2PL = ST$\cap$2PL.

Now let H be the history in example 5.1 above.

The history H is in Local-SER, Local-2PL and Local-S2PL since both H'_1 and H'_2 are in SER', 2PL' and S2PL'.

However H is not in SER, 2PL or S2PL:

- CSG(H) has a cycle, so by theorem 2.1 H is not in SER .
- If it is in 2PL or S2PL it is also in SER, and we have a contradiction.

§

5.3 On generating global CO histories

This section describes how the Commitment Order Coordinator (COCO) defined in section 4.2 takes part in AC to guarantee global CO histories. (The CORCO described in section 4.3 is handled similarly.)

In a multi-RM environment that implements AC, a COCO typically receives a request via an AC protocol to commit some transaction T in the USG. If the COCO can commit the transaction, it votes YES on it via AC, which is an obligation to either commit or abort according to the decision reached by the AC protocol. Later, after being notified of the decision, if T is committed, all transactions in $\text{ABORT}_{\text{CO}}(T)$ need to be aborted (by algorithm 4.1). Thus the COCO (say, of RM i) has to delay its YES vote on T, if it has voted YES on any transaction in $\text{ABORT}_{\text{CO}}(T)$ (CD^3C by theorem 5.1). When YES vote on T is possible, the COCO may either choose to do so immediately upon being requested (the *non-blocking without delays* approach), or to delay the voting for a given, predetermined amount of time (*non-blocking with delays*). During the delay the set $\text{ABORT}_{\text{CO}}(T)$ may become smaller or empty, since its members may be decided and removed from the USG, and since $\text{ABORT}_{\text{CO}}(T)$ cannot increase after T has reached the ready state.
Instead of immediately voting, or delaying the voting for a given amount of time (which may still result in aborts) the COCO can *block* the voting on T until all transactions in $\text{ABORT}_{\text{CO}}(T)$ are decided. However, if another RM in the environment also blocks, this may result in a global deadlock (e.g., if T' is in $\text{ABORT}_{\text{CO}}(T)$ for one RM, and T is in $\text{ABORT}_{\text{CO}}(T')$ for another RM). Aborting transactions by *timeout* is a common mechanism for resolving such deadlocks. Controlling the timeout by the AC protocol, rather than aborting independently by the RMs, is preferable for preventing unnecessary aborts.
Note that aborting transactions by the COCO is necessary only if a local cycle in its USG is not eliminated by some external entity (e.g. a scheduler that generates a cycle-free local USG or one that uses aborts to resolve local cycles), or if a global cycle (across two or more local USGs) is generated. Since the cycles are generated exclusively by the way the RASs operate and are independent of the commit order, the COCO does not have to abort more transactions that need to be aborted (also when using any other concurrency control) for serializability violation prevention.

6 Guaranteeing global serializability by Local Commitment Ordering

This section shows necessary and sufficient conditions for guaranteeing global serializability when the RMs are autonomous, i.e. the only exchanges between them for the purpose of transaction management are those of atomic commitment protocols (with no piggy-backing of any additional concurrency control information; e.g. transaction timestamps).

6.1 Local-CO is sufficient for global serializability

The following is a consequence of theorem 3.1 (corollary 5.1) and theorem 5.2 :

Theorem 6.1

SER $\supset$ Local-CO (i.e. if a history is in Local-CO then it is globally serializable).

Remark: Local-CO is maintained, if all the RMs in the environment use any types (possibly different, e.g. CORCO and S-S2PL based) of CO mechanisms.

6.2 Conditions when Local-CO is necessary to guarantee global serializability

Theorem 6.1 states that local CO is a *sufficient condition* for global serializability. We now use (informally) Knowledge Theory based arguments (see for example [Halp 87], [Hadz 87]) to prove that Local-CO is also *necessary for guaranteeing* global serializability in a multi-RM environment, when the RMs support local serializability and use atomic commitment exchanges only for coordination. (*necessary for guaranteeing* means that otherwise a violation may occur; see definition 2.1.)

The necessity in CO is proven by requiring that each RM avoid committing any transaction that can potentially cause a serializability violation when committed. If it is clear that a transaction remains in such a situation forever (based on knowledge available to the RM locally), the transaction is aborted. We name such a transaction a *permanent risk (PR).* The PR property is relative to a RM. The above requirement implies that each RM in the environment has to implement the following *commitment strategy* (CS):

- CS
  - Starting from a history with no decided transactions, commit any ready transaction via an AC protocol. Every other transaction that is a PR is aborted[1].
  - Repeat (asynchronously when possible) the following procedure:
    Commit (via AC) any ready transaction, that *cannot cause a serializability violation*, and abort all the PR transactions.

The resulting global histories are proven to be in CO.

Theorem 6.2

- If all the RMs in the environment are autonomous and provide local serializability, then CS is a *necessary* strategy for each RM, in order to guarantee global serializability.
- If CS is implemented by all the RMs, the global histories generated are in Local-CO..

Proof:

The Serializability Theorem (theorem 2.1) implies that the serializability graph provides all the necessary information about serializability. We assume that every RM, say RM i, "knows" its local serializability graph SG_i (it includes all the committed and undecided transactions) and its subgraphs CSG_i (includes committed transactions only) and USG_i (includes all undecided transactions). We also assume (based on AC) that each RM has committed a transaction, if and only if it has voted Yes, and "knows" that all other RMs participating in a transaction have voted Yes, and will eventually commit it.

The goal for each RM is to guarantee a cycle-free (global) CSG (committed transaction serializability graph), by avoiding any action that may create a global cycle (local cycles in CSG_i are eliminated by RM i, since local serializability is assumed in the theorem).

First, CS is trivially necessary for the following reasons: since a PR transaction remains PR for ever (by definition), it cannot be committed and thus must be aborted to free computing resources. On the other hand, any ready transaction that cannot cause a serializability violation can be committed.

We now need to identify PR transactions, while implementing CS. We show that this implies that each RM executes algorithm 4.1.

Each RM implements CS as follows:

[1] A hidden axiom is assumed, that computing resources are not held unnecessarily. Otherwise, PR transactions can be marked and kept undecided forever. Aborting such transactions and reexecuting them also supports a general concept of *fairness* that requests a transaction's successful completion within a reasonable time interval.

- Base stage:
  Assume that CSG_i does not include any transaction. Commit any ready transaction T (via AC).
  Suppose that prior to committing T there is an edge $T' \to T$ in USG_i. It is possible that there exists an edge $T \to T'$ in some USG_j of some RM j, j≠i, but RM i, though, cannot verify this (due to the autonomy requirement). This means that committing T' later may cause a cycle in CSG. Since committing T cannot be reversed (see transaction state transitions in section 2), no future event can change this situation. Hence T' is a PR (in RM i), and RM i must abort it (by voting No via AC) upon committing T.

- Inductive stage:
  Suppose that CSG_i includes at least one transaction. We show that no ready transaction can cause a serializability violation if committed, and hence can be committed (provided that a consensus to commit is reached by all the participating RMs via AC):
  Commit any ready transaction T.
  (i) Examine any undecided transactions T' (in USG_i).
  Suppose that prior to committing T there is an edge $T' \to T$ in USG_i. Using again the arguments given for the base stage, T' is a PR, and RM i must abort it. If there is no edge from T' to T, there is no path possibly left from T' to T, after aborting the PR transactions above. Thus no additional T' is a PR and no decision on T' is taken at this stage.
  (ii) Examine now any previously committed transaction T" (in CSG_i).
  It is impossible to have a path $T \to ... \to T"$ in CSG_i or in CSG_j for any RM j, j≠i , since, if this path existed at the stage when T" was committed, it would have been disconnected during that stage, when aborting all the PR transactions (with edges to T"; using (i) above), and since no incoming edges to T" could have been generated after T" has been committed. Hence, only a path $T" \to ... \to T$ can exist in CSG_i or in CSG_j for any RM j, j≠i. This means that no cycle in CSG through T and T" can be created, and no T" needs to be aborted (which is impossible since T" is committed, and would fail the strategy).
  The arguments above ensure that no ready transaction can cause a serializability violation when committed at the beginning of an inductive stage, as was assumed, and hence (any ready transaction) T could have been committed. Note that committing a transaction can start before the commit process is completed for a previous one (i.e. a concurrent implementation of the strategy), as long as CD^3C is maintained for T' and T, where there exists an edge $T' \to T$ in USG_i. Without enforcing CD^3C, a committed transaction can be identified later as a PR in RM i, and can cause a serializability violation.

In the CS implementation above, all the PR transactions are identified and aborted at each stage. Examining this implementation we conclude that it results in exactly performing algorithm 4.1 in each RM (at the stage when T is committed in RM j, the set of all PR transactions is exactly the set $ABORT_{CO}(T)$). Hence, by theorem 4.1 every RM involved guarantees CO, and by enforcing CD^3C, also CO'. This means that the generated (global) history is in Local-CO. The only possible deviation from the implementation above is by aborting additional transactions at each stage. Such a deviation still maintains the generated history in Local-CO.

§

Theorems 6.1 and 6.2 imply the following:

Corollary 6.1

Guaranteeing *Local-CO* is a *necessary* and *sufficient condition* for *guaranteeing* (global) *serializability* in an environment of *autonomous* RMs.

7 Conclusion

This work generalizes a previously known result, that Strong Strict Two Phase Locking (S-S2PL) together with Two Phase Commit (2PC) guarantee *global serializability* in a multi *resource manager* (RM) environment. The new concept defined here, *Commitment Ordering* (*CO*), provides additional ways to achieve global serializability, through different concurrency control mechanisms, that may provide *deadlock-free* executions. This allows the levels of concurrency to be controlled by local trade-offs between blocking implementations of CO (e.g. S-S2PL), which are subject to deadlocks, and deadlock-free CO implementations, which are subject to cascading aborts. To guarantee global serializability, no services, but those of *atomic commitment*, are necessary for the coordination of transactions across RMs, if each RM supports CO. Another result shown is that guaranteeing CO is *necessary* for guaranteeing global serializability, when the RMs involved are *autonomous* (i.e. when only atomic commitment is used for RM coordination).

The relationships between various properties of the histories generated by the individual RMs and respective properties of the respective global history are examined, and assuming that atomic commitment is used, it is shown which properties, CO in particular, are preserved globally when applied locally by the RMs.

Generic CO enforcing mechanisms are described as well, and their behavior in a multi-RM environment is examined. Since CO can be enforced locally in each RM (most existing commercial database systems are S-S2PL based, and already provide CO), no change in existing atomic commitment protocols and interfaces is required to utilize the CO solution.

The study presented in this work suggests that CO is a practical, fully distributed solution for the global serializability problem in a distributed, high-performance transaction-processing environment (see also [Raz 91a] for implementation-oriented aspects of CO).

Autonomy implies that a RM has no knowledge of whether a transaction is *local*, i.e. confined to the RM, or *global*, i.e. spanning more than one RM. If a RM is coordinated with other RMs via AC protocols only, and in addition can identify its local transactions (e.g. by notifications from applications (either implicitly or explicitly), or through AC protocols), it is said to have *extended knowledge autonomy (EKA).* Since local transactions do not need to be coordinated across RMs via AC protocols, they do not need to obey the CO condition for the purpose of global serializability. Under EKA a more general property, *Extended Commitment Ordering (ECO),* is necessary to guarantee global serializability (see [Raz 91b]). ECO reduces to CO when all the transactions are assumed to be global.

Acknowledgments

Many thanks are due to several people for useful discussions on the subject of this work or comments on an early version of this paper: Rob Abbott, Phil Bernstein, Edward Braginsky, Jeff East, Bill Emberton, Maurice Herlihy, Mei Hsu, Walt Kohler, Dave Lomet, Bob Taylor, Vijay Trehan and Mark Tuttle.
Special thanks are due to Vijay Trehan for continued encouragement and support.

References

[Bern 87] P. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Data-base Systems,* Addison-Wesley, 1987.

[Brei 90], Y. Breibart, A. Silberschatz, G. Thompson, "Reliable Transaction Management in a Multidatabase System", in *Proc. of the ACM SIGMOD Int. Conf. on Management of Data,* Atlantic City, New Jersey, June 1990

[Brei 91], Y. Breibart, Dimitrios Georgakopoulos, Marek Rusinkiewicz, A. Silberschatz, "On Rigorous Transaction Scheduling", *IEEE Trans. Soft. Eng.,* Vol 17, No 9, September 1991.

[DECdtm] J. Johnson, W. Laing, R. Landau, "Transaction Management Support in the VMS Operating System Kernel", *Digital Technical Journal,* Vol 3, no. 1, Winter 1991.
Philip A. Bernstein, William T. Emberton, Vijay Trehan, "DECdta - Digital's Distributed Transaction Processing Architecture", *Digital Technical Journal,* Vol 3, no. 1, Winter 1991.

[EMA] *Enterprise Management Architecture - General Description,* Digital Equipment Corporation, EK-DEMAR-GD-001.

[Elma 90] A. Elmagarmid, W. Du, "A Paradigm for Concurrency Control in Heterogeneous Distributed Database Systems", *Proc. of the Sixth Int. Conf. on Data Engineering,* Los Angeles, California, February 1990.

[Eswa 76] Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L., "The Notions of Consistency and Predicate Locks in a Database System", *Comm. ACM* 19(11), pp. 624-633, 1976.

[Geor 91] Dimitrios Georgakopoulos, Marek Rusinkiewicz, Amit Sheth, "On serializability of Multi database Transactions Through Forced Local Conflicts", in *Proc. of the Seventh Int. Conf. on Data Engineering,* Kobe, Japan, April 1991.

[Glig 85] V. Gligor, R. Popescu-Zeletin, "Concurrency Control Issues in Distributed Heterogeneous Database Management Systems", in F. A. Schreiber, W. Litwin editors, *Distributed Data Sharing Systems,* pp. 43-56, North Holland, 1985.

[Gray 78] Gray, J. N., "Notes on Database Operating Systems", *Operating Systems: An Advanced Course,* Lecture Notes in Computer Science 60, pp. 393-481, Springer-Verlag, 1978.

[Hadz 87] Vassos Hadzilacos, "A Knowledge Theoretic Analysis of Atomic Commitment Protocols", *Proc. of the Sixth ACM Symposium on Principles of Database Systems*, pp. 129-134, March 23-25, 1987.

[Halp 87] Joseph Y. Halpern, "Using Reasoning about Knowledge to Analyze Distributed Systems", Research Report RJ 5522 (56421) 3/3/87, Computer Science, IBM Almaden Research Center, San Jose, California, 1987.

[Kung 81] Kung, H. T., Robinson, J. T., "On Optimistic Methods for Concurrency Control", *ACM Tras. on Database Systems* 6(2), pp. 213-226, June 1981.

[Lamp 76] Lampson, B., Sturgis, H., "Crash Recovery in a Distributed Data Storage System", Technical Report, Xerox, Palo Alto Research Center, Palo Alto, California, 1976.

[Litw 89] Litwin, W., H. Tirri, "Flexible Concurrency Control Using Value Date", in *Integration of Information Systems: Bridging Heterogeneous Databases*, ed. A. Gupta, IEEE Press, 1989.

[Lome 90] David Lomet, "Consistent Timestamping for Transactions in Distributed Systems", Technical Report CRL 90/3, Digital Equipment Corporation, Cambridge Research Lab, September 1990.

[LU6.2] System Network Architecture - Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2, SC30-3269-3, International Business Machines Corporation, 1985.

[OSI-CCR] ISO/IEC IS 9804, 9805, JTC1/SC21, *Information Processing Systems - Open Systems Interconnection - Commitment, Concurrency and Recovery service element*, October 1989.

ISO/IEC JTC1/SC21 N4611 Addendum, *CCR Tutorial* - Annex C of ISO 9804, August 1990.

[OSI-SMO] ISO/IEC DP 10040, JTC1/SC21, *Information Processing Systems - Open Systems Interconnection - System Management Overview*, September 1989.

[OSI-DTP] ISO/IEC DIS 10026 (1, 2, 3), JTC1/SC21, *Information Processing Systems - Open Systems Interconnection - Distributed Transaction Processing*, October 1989.

[Papa 86] Papadimitriou, C. H., *The Theory of Concurrency Control*, Computer Science Press, 1986.

[Pu 88] Calton Pu, "Transactions across Heterogeneous Databases: the Superdatabase Architecture", Technical Report No. CUCS-243-86 (revised June 1988), Department of Computer Science, Columbia University, New York, NY.

[Raz 90] Yoav Raz, "The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers", DEC-TR 841, Digital Equipment Corporation, November 1990, revised April 1992.

[Raz 91a] Yoav Raz, "The Commitment Order Coordinator (COCO) of a Resource Manager, or Architecture for Distributed Commitment Ordering Based Concurrency Control", DEC-TR 843, Digital Equipment Corporation, December 1991, revised April 1992.

[Raz 91b] Yoav Raz, "Extended Commitment Ordering, or Guaranteeing Global Serializability by Applying Commitment Order Selectively to Global Transactions", DEC-TR 842, Digital Equipment Corporation, November 1991, revised April 1992.

[Shet 90] Amit Sheth, James Larson, "Federated Database Systems", *ACM Computing Surveys*, Vol. 22, No 3, pp. 183-236, September 1990.

[Silb 91] Avi Silberschatz, Michael Stonebraker, Jeff Ullman, "Database Systems: Achievements and Opportunities", *Communications of the ACM*, Vol. 34, No. 10, October 1991.

[Weih 89] William E. Weihl, "Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types", *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 2, pp. 249-282, April 1989.