$(T_i, st_1.v)(st_1.v, st_2.v) \cdots (st_{j-1}.v, st_j.v)(st_j.v, w)$ such that $(st_1.v, st_2.v)$ is not a committed edge. Thus, since w is marked "special" (edge (w, T_i) is in the TSG) and aborted edges (which are also not committed) are deleted from the TSG in Step 1 of Detect_Cycles1, there is a cycle in the TSG $(T_i, st_1.v)(st_1.v, st_2.v) \cdots (st_j.v, w)(w, T_i)$ such that $(st_1.v, st_2.v)$ is not a committed edge and all the edges are either unmarked or committed edges. Thus, insertion of T_i's edges into the TSG violates the edge insertion rule. □

Algorithm Detect_Cycles1 thus determines if the insertion of a transaction T_i's edges into the TSG violates the edge insertion rule (point 1 of the augmented edge insertion rule). Algorithm Detect_Cycles2, presented below, determines if insertion of transaction T_i's edges into the TSG violates point 2 of the augmented edge insertion rule.

procedure Detect_Cycles2($(V, E), exec(T_i)$):
for (every transaction $T_j \in V$)
begin
$S = exect(T_i) \cup exec(T_j)$;
if for all sites $s_q \in S$, (T_j, s_q) is a committed edge or for all sites $s_q \in S$, (T_j, s_q) is an aborted edg
then skip
else return ("violates");
end
return("does not violate")

Lemma 8: Consider a state st_j of Detect_Cycles1 and states $st_1, st_2, \ldots, st_{j-1}$ such that $st_j \rightarrow st_{j-1}$, $st_{j-1} \rightarrow st_{j-2}$, ..., $st_1 \rightarrow st_0$ denote the sequence of reverse transitions from st_j to st_0. If edge (u, w) is marked "used" by Detect_Cycles1 when it is in state st_j ($st_j.v = u$) and $w \neq st_k.v$, for all k, $k = 0, 1, 2, \ldots, j$, then there is a path from T_i to w, $(T_i, st_1.v)(st_1.v, st_2.v) \cdots (st_{j-1}.v, st_j.v)(st_j.v, w)$ such that $(st_1.v, st_2.v)$ is not a committed edge (if $j = 1$, $st_2.v = w$).

Proof: We prove the lemma by induction on j.

Basis ($j = 0$): If (u, w) is marked "used" by Detect_Cycles1 when it is in state st_0, then since $st_0.v = u = T_i$, the path from T_i to w is (T_i, w).

Induction: Assume the lemma is true for $j = r$, $r \geq 0$. We show that the lemma is true for $j = r + 1$. Thus, edge (u, w) is marked "used" by Detect_Cycles1 when it is in state st_{r+1}, $st_{r+1}.v = u$ and $w \neq st_k.v$, $k = 0, 1, 2, \ldots, r + 1$, where $st_{r+1} \rightarrow st_r$, ..., $st_1 \rightarrow st_0$ denote the sequence of reverse transitions from st_{r+1} to st_0. Consider the transition $st_r \rightarrow st_{r+1}$ that results in $st_{r+1}.v$ being marked "visited" and edge $(st_r.v, st_{r+1}.v)$ being marked "used". Since $st_{r+1}.v$ is not marked "visited" just before the transition $st_r \rightarrow st_{r+1}$ is made, $st_{r+1}.v \neq st_k.v$ for all k, $k = 0, 1, 2, \ldots, r$. Thus, by the induction hypothesis, there is a path from T_i to $st_{r+1}.v$: $(T_i, st_1.v)(st_1.v, st_2.v) \cdots (st_r.v, st_{r+1}.v)$. Further, since there is a path from T_i to $st_{r+1}.v$, $w \neq st_k.v$, for all k, $k = 0, 1, 2, \ldots, r + 1$, and $(st_{r+1}.v, w)$ is an edge in the TSG, there is a path from T_i to w, $(T_i, st_1.v)(st_1.v, st_2.v) \cdots (st_r.v, st_{r+1}.v)(st_{r+1}.v, w)$. If $r \geq 1$, then by induction hypothesis, $(st_1.v, st_2.v)$ is not a committed edge. If, on the other hand, $r = 0$, then since $st_1.v$ is marked "special" in Step 1 and $(st_1.v, w)$ is marked "used" when Detect_Cycles1 is in state st_1, $(st_1.v, w)$ is not a committed edge.□

Theorem 8: If Detect_Cycles1 returns "violates", then insertion of T_i's edges into the TSG violates the edge insertion rule.

Proof: Let us assume that just before Detect_Cycles1 returns "violates", it is in state st_j and it marks edge (u, w) "used" ($st_j.v = u$). Thus, w must be marked "special", $current \neq w$ and $u \neq T_i$. Let $st_j \rightarrow st_{j-1}$, $st_{j-1} \rightarrow st_{j-2}$, ..., $st_1 \rightarrow st_0$ denote the sequence of reverse transitions from st_j to st_0. Since $st_j.v \neq T_i$ and the TSG is bipartite, $j \rangle 1$. Also, since w is marked "special", $w \neq T_i$ or $w \neq st_0.v$. Further, by Lemma 6, since until the reverse transition $st_1 \rightarrow st_0$ is made, no nodes that are marked "special" can be marked "visited" (as a result, none of $st_k.v$ for all k, $k = 2, \ldots, k$ are marked "special"), $w \neq st_k.v$, for all k, $k = 2, \ldots, j$ and $current = st_1.v$. Since $current \neq w$, $w \neq st_1.v$. Thus, by Lemma 8, there is a path from T_i to w:

Theorem 7: If insertion of T_i's edges into the TSG violates the edge insertion rule, then Detect_Cycles1((V,E), $exec(T_i)$) returns "violates".

Proof: If the edge insertion rule is violated by the insertion of transaction T_i's edges, then there exists a cycle in the TSG of the form $(T_i, v_1), (v_1, v_2), \ldots, (v_{r-1}, v_r), (v_r, T_i)$, $r \rangle 2$, such that edge (v_1, v_2) is not a committed edge. Further edges $(T_i, v_1), (v_1, v_2), \ldots, (v_{r-1}, v_r), (v_r, T_i)$ are either unmarked or committed edges, and nodes $v_2, \ldots, v_{r-1}$ are not marked "special", while v_1 and v_r are marked "special" in Step 1.

Since all of the edges in the cycle are either unmarked or committed edges, none of them are deleted from the TSG in Step 1. By lemmas 4 and 5, Detect_Cycles1 must make a transition $st_0 \rightarrow st_j$ during its execution, where $st_0.v = T_i$ and $st_j.v = v_1$ (unless Detect_Cycles1 terminates earlier in which case it returns "violates" and the theorem is proved). Also, none of $v_2, \ldots, v_{r-1}$ are marked "visited" when Detect_Cycles1 makes the transition $st_0 \rightarrow st_j$. (Suppose at least one of $v_2, \ldots, v_{r-1}$ is marked "visited" before Detect_Cycles1 makes the transition $st_0 \rightarrow st_j$. Let v_m, for some $m = 2, \ldots, r-1$, be the first to be marked "visited" among $v_2, \ldots v_{r-1}$ as a result of Detect_Cycles1 making the transition $st_k \rightarrow st_l$ ($st_l.v = v_m$). There is a path from v_m to v_1: $(v_m, v_{m-1}), \ldots, (v_2, v_1)$, such that none of $v_m, \ldots, v_2$ are marked "special". Further, none of $v_{m-1}, \ldots, v_1$ are marked "visited" when v_m is marked "visited". Thus, by Lemma 7, (v_2, v_1) is marked "used" before the reverse transition $st_l \rightarrow st_k$ takes place. As a result, v_1, a node that is marked "special", is marked "visited" before the reverse transition $st_l \rightarrow st_k$ takes place, thus contradicting Lemma 6.)

Thus, by Lemma 7, since $v_2, \ldots, v_{r-1}$ are not marked "special" and are not marked "visited" when v_1 is marked "visited" due to transition $st_0 \rightarrow st_j$, and edge (v_1, v_2) is not a committed edge, edge (v_{r-1}, v_r) is marked "used" before the reverse transition $st_j \rightarrow st_0$ is made. Further, since by Lemma 6, no node marked "special" is marked "visited" before Detect_Cycles1 makes the reverse transition $st_j \rightarrow st_0$, $current = v_1$ at least until the reverse transition $st_j \rightarrow st_0$ is made. Thus, $current = v_1$ when (v_{r-1}, v_r) is marked "used". Since v_r is marked "special", $r \rangle 2$, and by the definition of path, $v_1 \neq v_r$, Detect_Cycles1 returns "violates". □

We now prove that if Detect_Cycles1 returns "violates", then insertion of T_i's edges into the TSG violates the edge insertion rule.

Lemma 6: If Detect_Cycles1 makes a state transition $st_k \rightarrow st_l$ (causing node $st_l.v$ to be marked "visited"), then no other node marked "special" is marked "visited" before Detect_Cycles makes the reverse transition $st_l \rightarrow st_k$.

Proof: Suppose a state transition $st_i \rightarrow st_j$ results in a node marked "special" being marked "visited". By Lemma 5, $st_i = st_0$. However, the only way Detect_Cycles1 can be in state st_0 (the initial state) is by first making a reverse transition $st_l \rightarrow st_k$. □

Lemma 7: If there is a path from node v_0 to node v_j, $(v_0, v_1)(v_1, v_2) \cdots (v_{j-1}, v_j)$, in the TSG (V, E) such that

- for all $k = 1, 2, \ldots, j-1$, if v_k is marked "special", then edge (v_k, v_{k+1}) is not a committed edge, and
- when v_0 is marked "visited" due to state transition $st_j \rightarrow st_k$ ($st_k.v = v_0$), none of $v_1, v_2, \ldots, v_j$ are marked "visited",

then edge (v_{j-1}, v_j) is marked "used" during the execution of Detect_Cycles1 before the reverse transition $st_k \rightarrow st_j$ is made.

Proof: We prove the above lemma by induction on j.

Basis ($j = 1$): Trivially, if (v_0, v_1) is not marked "used", then before making the reverse transition $st_k \rightarrow st_j$, Detect_Cycles1 marks (v_0, v_1) "used" (in case v_0 is marked special, then since (v_0, v_i) is not a committed edge, it will be marked "used").

Induction: Assume the lemma is true for $j = r$, $r \rangle 0$. We need to show that the lemma is true for $j = r + 1$. Let the path be $(v_0, v_1)(v_1, v_2) \cdots (v_{r-1}, v_r)(v_r, v_{r+1})$. By the induction hypothesis, (v_{r-1}, v_r) is marked "used" before Detect_Cycles1 makes the reverse transition $st_k \rightarrow st_j$. Since v_r is not marked "visited" when Detect_Cycles1 makes transition $st_j \rightarrow st_k$, v_r is marked "visited" before Detect_Cycles1 makes the reverse transition $st_k \rightarrow st_j$ and after it makes the transition $st_j \rightarrow st_k$. Thus, if v_r is marked "visited" due to transition $st_l \rightarrow st_m$ ($st_m.v = v_r$), then Detect_Cycles1 makes the reverse transition $st_m \rightarrow st_l$ before it makes the reverse transition $st_k \rightarrow st_j$. Since Detect_Cycles1 marks edge (v_r, v_{r+1}) "used" before making the reverse transition $st_m \rightarrow st_l$, Detect_Cycles1 marks (v_r, v_{r+1}) "used" before making the reverse transition $st_k \rightarrow st_j$ (in case v_r is marked special, then since (v_r, v_{r+1}) is not a committed edge, it will be marked "used"). □

Corollary 2: Detect_Cycles1 terminates in a finite number of steps.

Proof: Every time a transition $st_0 \rightarrow st_j$ is made, by Lemma 4, a reverse transition $st_j \rightarrow st_0$ is made in a finite number of steps. Since there are a finite number of choices in state st_0, each choice is eliminated when a transition $st_0 \rightarrow st_j$ is made, and no further transitions can be made once all choices have been eliminated, Detect_Cycles1 terminates in a finite number of steps. □

In order to prove that Detect_Cycles1 returns "violates" iff insertion of T_i's edges into the TSG violates the edge insertion rule, we first define the notion of a *path* in the TSG.

Definition 4: In a TSG (V, E), $(v_0, v_1)(v_1, v_2) \cdots (v_{k-1}, v_k)$, $k \rangle 0$, is a path from v_0 to v_k iff

- for all i, $i = 0, 1, 2, \ldots, k-1$, $(v_i, v_{i+1}) \in E$, and
- for all pairs (i, j), such that $i, j = 0, 1, 2, \ldots, k$, $i < j$, the following is true: $v_i \neq v_j$.

If, in addition, $(v_k, v_0) \in E$, then the set of edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)(v_k, v_0)$ form a *cycle*. □

We begin by showing that if insertion of T_i's edges into the TSG violates the edge insertion rule, then Detect_Cycles1((V,E), $exec(T_i)$) returns "violates". We first need to prove the following lemmas.

Lemma 5: Consider a state transition $st_k \rightarrow st_l$ that results in node $st_l.v$ being marked "visited". If node $st_l.v$ is marked "special", then $st_k.v = T_i$ (or alternatively, $st_k = st_0$).

Proof: Suppose $st_k.v \neq T_i$. Then, before $st_l.v$ is marked "visited", Detect_Cycles1 would terminate and return "violates" since just before Detect_Cycles1 makes transition $st_k \rightarrow st_l$, $current \neq st_l.v$ (since $st_l.v$ is not marked "visited" before the transition $st_k \rightarrow st_l$), and $st_l.v$ is marked "special". This leads to a contradiction since it is given that transition $st_k \rightarrow st_l$ results in $st_l.v$ being marked "visited". Thus, $st_k.v = T_i$. Since no state transitions are caused by nodes that are already marked "visited", and T_i is marked "visited" in Step 2, the only state st_k for which $st_k.v = T_i$ is st_0. Thus, $st_k = st_0$. □

where $v_0, v_1, \ldots, v_n$ are nodes in the TSG. We denote the values of v, $F(v_i)$, $v_i \in V$ in state st_j, by $st_j.v$, $st_j.F(v_i)$ respectively. Detect_Cycles1 is said to be in state st_j at a point in between the execution of any two of its steps, if at that point, $v = st_j.v$, and for all nodes $v_i \in V$, $F(v_i) = st_j.F(v_i)$. Certain steps in Detect_Cycles1 cause it to move from one state to another. When a step causes Detect_Cycles1 to move from state st_j to state st_k, Detect_Cycles1 is said to make a *state transition* $st_j \rightarrow st_k$. Note that only steps 4 and 5 cause state transitions (we assume that the state of Detect_Cycles1 is undefined before the execution of Step 2 and after Detect_Cycles1 terminates).

Lemma 4: If Detect_Cycles1 makes a state transition $st_j \rightarrow st_k$ due to Step 4, then after the execution of a finite number of steps, Detect_Cycles1 also makes the reverse transition $st_k \rightarrow st_j$ (due to Step 5).

Proof: We prove the above lemma by induction on num, where num is the number of edges marked "unused" in the TSG just after the transition $st_j \rightarrow st_k$ is made.

Basis ($num = 0$): In this case, since all the edges in the TSG are marked "used", there is no further choice of edges from v in state st_k (Step 3). As a result, Detect_Cycles1 executes Step 5, and thus makes the reverse transition $st_k \rightarrow st_j$ in a finite number of steps.

Induction: Assume the lemma is true for $num = r$. We show that the lemma is true for $num = r + 1$. Thus, just after the transition $st_j \rightarrow st_k$ is made, the number of edges marked "unused" in the TSG is $r + 1$. For every choice of edges (v, u) while in state st_k, if u is marked "visited", then the choice is eliminated in one step by marking edge (v, u) "used". If u is not marked "visited", then Detect_Cycles1 marks edge (v, u) "used" and makes a state transition $st_k \rightarrow st_l$ due to Step 4, where $st_l.v = u$, $st_l.F(v) = st_k.v$. As a result, the number of edges marked "used" in the TSG just after the transition $st_k \rightarrow st_l$ is made is r. Thus, by the induction hypothesis, Detect_Cycles1 makes a reverse transition $st_l \rightarrow st_k$ in a finite number of steps. Since there are a finite number of choices of edges in state st_k, each choice is eliminated when a transition $st_k \rightarrow st_l$ is made, and no further state transitions (due to Step 4) can be made once all choices have been eliminated, Detect_Cycles1 makes the reverse transition $st_k \rightarrow st_j$ (due to Step 5) in a finite number of steps. □

A consequence of the Lemma 4 is that algorithm Detect_Cycles1 terminates in a finite number of steps. We refer to the initial state (the state immediately after the execution of Step 2 of Detect_Cycles1) as st_0. Thus, $st_0.v = T_i$ and $st_0.F(v) = null$ for all nodes v in the TSG.

-Appendix C-

The following algorithm, Detect_Cycles1, has complexity similar to the depth-first search algorithm for graphs, and determines if insertion of T_i's edges into the TSG violates the edge insertion rule. Detect_Cycles1 takes as arguments the TSG and the set of sites at which T_i executes. It returns "violates" iff insertion of T_i's edges into the TSG violates the edge insertion rule. It traverses edges in the TSG marking them "used" as it goes along so that an edge is not traversed multiple times. In Detect_Cycles1, v is the current node being visited, and $F(v)$ is the node from which v is visited and to which backtracking from v must take place.

procedure Detect_Cycles1($(V, E), exec(T_i)$):

1. Delete aborted edges (which are also not committed edges) from the TSG. Add T_i's edges t the TSG each of which is neither committed nor aborted.
2. Mark all edges "unused". For all nodes v in the TSG, mark v "unvisited" and set $F(v) = nul$ Mark all site nodes at which T_i executes "special". Also, $v := T_i$, and mark T_i "visited".
3. If every edge (v, u) is marked "used", then go to step (5).
   If v is marked "special" and for every edge (v, u), either
   - (v, u) is marked "used", or
   - (v, u) is a committed edge,

   then go to step (5).
4. Choose an edge (v, u) that is not marked "used" and in addition, if v is marked "special" then (v, u) is not a committed edge.
   Mark edge (v, u) "used".
   If $v = T_i$, then $current := u$.
   If $v \neq T_i$, $u \neq current$ and u is marked "special", **return**("violates").
   If u is not marked "visited", then $F(u) := v$, $v := u$ and u is marked "visited".
   Go to step (3).
5. If $v \neq T_i$, then $temp := F(v)$, $F(v) := null$ and $v := temp$.
6. **return**("does not violate").

We now prove that Detect_Cycles1 returns "violates" iff insertion of T_i's edges into the TSG violates the edge insertion rule. For this purpose, we introduce the notion of a state of Detect_Cycles1. A state of Detect_Cycles1, denoted by st_j, is a tuple $(v, F(v_0), F(v_1), \ldots, F(v_n))$,

wait for other transactions). Thus, T_i must be a compensating transaction whose edges have not been inserted into the TSG.

We now show that it is impossible for there to be a cycle consisting of only compensating transactions. Suppose there is a cycle consisting of compensating transactions $CT_1, CT_2, \ldots, CT_n$, CT_1 such that CT_1 waits for CT_2, CT_2 waits for CT_3, ..., CT_n waits for CT_1. Thus, there must exist a cycle of weakly terminated transactions $T_1, T_2, \ldots, T_n, T_1$ that have not strongly terminated such that $T_{(i \ mod \ n)+1}$ commits and aborts at two sites at which T_i commits (since CT_i executes at sites at which T_i commits and CT_i waits for $CT_{(i \ mod \ n)+1}$). Due to the augmented edge insertion rule, $T_{(i \ mod \ n)+1}$'s edges must have been inserted into the TSG after T_i's edges are inserted into the TSG. Thus T_2's edges are inserted into the TSG after T_1's edges are inserted into the TSG, T_3's after T_2's and so on. Thus, T_n's edges are inserted into the TSG after T_1's edges are inserted and T_1's edges are inserted into the TSG after T_n's edges are inserted. This leads to a contradiction, and thus, the GTM concurrency control protocol is deadlock-free. □

ensures that if T_i is serialized before T_j, then T_j's edges cannot be inserted into the TSG before T_i's edges are inserted into the TSG.

Proof: Suppose T_j's edges are inserted before T_i's edges are inserted into the TSG. Since T_j commits at least two sites at which T_i executes, T_i's edges cannot be inserted into the TSG until T_j commits at all sites s_k, $s_k \in (commit(T_j) \cap exec(T_i))$ (otherwise, the edge insertion rule may be violated). Thus, T_i is not serialized before T_j at any site $s_k \in (commit(T_i) \cap commit(T_k))$. However, since T_i is serialized before T_j, by Lemma 3, T_j's edges cannot be inserted into the TSG before T_i's edges are inserted. □

Proof of Theorem 3: By Theorem 1, the GTM commit protocol ensures that every non-atomic transaction is compensated for. The LTMs are assumed to ensure that schedules at local sites are serializable. This, coupled with the fact that aborted subtransactions are not redone from logs, but are retried, ensures the schedules at local sites are serializable even in the presence of failures. Thus, from Theorem 2, we can conclude that the schedule S is serializable.

We now show that for every transaction T_j in the schedule, if T_i is serialized before T_j in S_j, and T_i aborts at any other site at which T_j commits, then CT_i is not serialized after T_j. Since T_i and T_j have more than one site in common at which they execute (let s_q and s_r denote sites such that T_j commits at both s_q and s_r, while T_i commits and aborts at sites s_q and s_r respectively), and T_i is serialized before T_j, by Corollary 1, edges (T_i, s_q) and (T_i, s_r) must be inserted into the TSG before edges (T_j, s_q) and (T_j, s_r) are inserted. Furthermore, edge (T_i, s_r) is either an unmarked or an aborted edge, while edge (T_i, s_q) is either an unmarked, committed or aborted edge. By the augmented edge insertion rule, edges (T_j, s_q) and (T_j, s_r) can be inserted into the TSG only if either edges (T_i, s_q) and (T_i, s_r) are deleted from the TSG, or both are aborted edges. In both cases, due to the augmented edge deletion rule, CT_i must commit at every site $s_k \in (commit(CT_i) \cap exec(T_j))$ before T_j's edges are inserted into the TSG. Thus, by Lemma 3, CT_i is not serialized after T_j. □

Proof of Theorem 4: We show that no cycle of the form $T_1, T_2, \ldots, T_n, T_1$ such that T_1 waits for T_2, T_2 waits for T_3, ..., T_n waits for T_1 exists. Consider any transaction T_i, $i = 1, 2, \ldots, n$ in the cycle. If T_i's edges are in the TSG, then T_i cannot wait for any other transaction since the concurrency control followed by the local DBMSs does not cause transactions to wait. As a result, T_i's edges are not in the TSG. Furthermore, no transaction can wait for T_i if T_i's edges are not in the TSG unless T_i is a compensating transaction (since local DBMSs do not cause transactions to

- there exists a site s_r such that $s_r \in (commit(T_i) \cap commit(T_j))$, and
- for all $s_q \in (commit(T_i) \cap commit(T_j))$, T_i is not serialized before T_j at s_q.

If the edge management scheme is used, then T_j's edges are not inserted into the TSG before T_i's edges are inserted.

Proof: Since for all $s_q \in (commit(T_i) \cap commit(T_j))$, T_i is not serialized before T_j at site s_q, and T_i is serialized before T_j, without loss of generality, there must exist global transactions $T_1, T_2, \ldots, T_n$, and sites $s, s_1, s_2, \ldots, s_n$, $n \rangle 0$, such that $sub(T_i, s)$ is serialized before $sub(T_1, s)$ at s, $sub(T_1, s_1)$ is serialized before $sub(T_2, s_1)$ at s_1, ..., $sub(T_n, s_n)$ is serialized before $sub(T_j, s_n)$ at s_n, and $s \neq s_1$, $s_1 \neq s_2$, ..., $s_{n-1} \neq s_n$ (for the purpose of simplifying the proof, we assume that $i \rangle n$, $j \rangle n$, and the sites are numbered similar to the transactions). Suppose T_j's edges are inserted into the TSG before T_i's edges are inserted. By Lemma 2, since T_i is serialized before T_j, no edge incident on T_j is deleted before T_i weakly terminates. Further, since $commit(T_i) \cap commit(T_j) \neq \emptyset$, and T_j's edges are inserted into the TSG before T_i's edges are inserted, by the edge deletion rule, T_j weakly terminates before any of T_i's edges are deleted from the TSG. Due to Lemma 2, for all k, $k = 1, 2, \ldots, n$, no edge incident on T_k is deleted before T_i weakly terminates, and no edge incident on T_j is deleted before T_k weakly terminates. As a result, since a transactions' edges (at least one of whose subtransactions commits) are inserted into the TSG before it weakly terminates, at some time during the execution, the TSG contains edges incident on all the transactions $T_i, T_1, T_2, \ldots, T_n, T_j$. Let T_k be the last transaction among $T_i, T_1, T_2, \ldots, T_n$, whose edges are inserted into the TSG. The insertion of T_k's edges into the TSG violates the edge insertion rule since all edges in the cycle $(s_r, T_i), (T_i, s), (s, T_1), \ldots, (T_n, s_n), (s_n, T_j), (T_j, s_r)$ are either unmarked or committed edges and

- if $k = i$, then (s, T_1) is not a committed edge at the time of insertion of t's edges, (since $sub(T_i, s)$ is serialized before $sub(T_1, s)$).
- if $k = 1, 2, \ldots, n-1$, then (s_k, T_{k+1}) is not a committed edge at the time of insertion of T_k's edges, (since $sub(T_k, s_k)$ is serialized before $sub(T_{k+1}, s_k)$).
- if $k = n$, then (s_n, T_j) is not a committed edge at the time insertion of T_n's edges (since $sub(T_n, s_n)$ is serialized before $sub(T_j, s_n)$). $\Box$

Corollary 1: Let T_i, T_j be global transactions, and s_q, s_r be sites such that $s_q, s_r \in commit(T_j)$, $s_q, s_r \in exec(T_i)$, and either $s_q \in commit(T_i)$ or $s_r \in commit(T_i)$. The edge management scheme

are deleted, then trivially due to the edge deletion rule, T_i weakly terminates before T_1's edges are deleted. On the other hand, if T_i's edges are in the TSG when the first edge incident on T_1 is deleted from the TSG, then since both $sub(T_i, s)$ and $sub(T_1, s)$ commit, edges (T_i, s) and (T_1, s) are either committed or unmarked edges. Thus, by the edge deletion rule, T_i must have weakly terminated. □

Proof of Theorem 2: Suppose schedule S is not serializable. Since schedules at local sites are serializable, without loss of generality, there exists a cycle $T_1, T_2, \ldots, T_n, T_1, n\rangle 1$, in S consisting of only global transactions, and sites $s_1, s_2, \ldots, s_n$, such that $sub(T_1, s_2)$ is serialized before $sub(T_2, s_2)$ at site s_2, $sub(T_2, s_3)$ is serialized before $sub(T_3, s_3)$ at site s_3, ..., $sub(T_n, s_1)$ is serialized before $sub(T_1, s_1)$ at site s_1 (for the purpose of simplifying the proof, the sites are numbered similar to the transactions). Since for all i, $i = 1, 2, \ldots, n$, $commit(T_i) \neq \emptyset$, the edge insertion rule ensures that all of T_i's edges are inserted into the TSG before T_i weakly terminates. Further, by Lemma 2, no edge incident on T_2 is deleted before T_1 weakly terminates, ..., no edge incident on T_1 is deleted before T_n weakly terminates.

Thus, at some time during the execution of the transactions, for all i, $i = 1, 2, \ldots, n$, the TSG contains all the edges belonging to T_i and T_i has weakly terminated. Due to serializability of schedules at local sites, for some i, $i = 1, 2, \ldots, n$, $s_i \neq s_{(i \ mod \ n)+1}$. Since there exists a path in the TSG from T_i to $T_{(i \ mod \ n)+1}$, $i = 1, 2, \ldots, n$, (both T_i and $T_{(i \ mod \ n)+1}$ have subtransactions that execute at site $s_{(i \ mod \ n)+1}$), there exist two different paths from T_i to $T_{(i \ mod \ n)+1}$, one through $(T_i, s_{(i \ mod \ n)+1})$, and another through (T_i, s_i). Thus, there exists a cycle (T_1, s_2), (s_2, T_2), (T_2, s_3), (s_3, T_3), ..., (T_n, s_1), (s_1, T_1) in the TSG. Since $T_1, T_2, \ldots, T_n$ have weakly terminated, every edge in the cycle is a committed edge. Furthermore, for all i, $i = 1, 2, \ldots, n$, since $sub(T_i, s_{(i \ mod \ n)+1})$ is serialized before $sub(T_{(i \ mod \ n)+1}, s_{(i \ mod \ n)+1})$, $ser(sub(T_i, s_{(i \ mod \ n)+1}))$ executes before $ser(sub(T_{(i \ mod \ n)+1}, s_{(i \ mod \ n)+1}))$ executes. However, this leads to a contradiction since the edge insertion rule ensures that for some i, $i = 1, 2, \ldots, n$, $ser(sub(T_i, s_{(i \ mod \ n)+1}))$ does not execute before $ser(sub(T_{(i \ mod \ n)+1}, s_{(i \ mod \ n)+1}))$ executes. Thus, S is serializable. □

In order to prove Theorem 3, we need to first establish the following results.

Lemma 3: Let T_i and T_j be two global transactions such that:

- T_i is serialized before T_j,

-Appendix B-

Proof of Theorem 1: If the pivot subtransaction commits, then all cohorts are sent $\langle commit, T_i$ message. Further, due to the commit protocol, before the pivot subtransaction commits all compensatable cohorts have sent the GTM a $\langle ack\_commit, T_i\rangle$ message. Thus, all compensatable subtransactions have committed. Hence any subtransaction that aborts is a retriable subtransactions. Since each cohort receives a $\langle commit, T_i\rangle$ message from the GTM, the retriable subtransactions are retried and eventually commit. Thus, all subtransactions commit. If the pivot subtransaction does not commit, then all subtransactions are aborted by either the LTM or executing compensating transactions. The proof of this is similar to the proof of the previous case and thus omitted. □

In order to prove Theorem 2, we need the following lemma. For the proof of the lemma and the theorem we represent the subtransaction of a transaction T_i at a site s_j (that is, T_{ij}) by $sub(T_i, s_j)$.

Lemma 2: If the GTM follows the edge management scheme, and global transaction T_i is serialized before global transaction T_j in a schedule, then no edge incident on T_j is deleted from the TSG before T_i weakly terminates.

Proof: If T_i is serialized before T_j, without loss of generality, there exist global transactions $T_1, T_2, \ldots, T_n$, $n \geq 0$, and sites $s, s_1, s_2, \ldots, s_n$ such that $sub(T_i, s)$ is serialized before $sub(T_1, s)$ at site s, $sub(T_1, s_1)$ is serialized before $sub(T_2, s_1)$ at site s_1, ..., $sub(T_n, s_n)$ is serialized before $sub(T_j, s_n)$ at site s_n (for the purpose of simplifying the proof, we assume that $i \rangle n$, $j \rangle n$, and the sites are numbered similar to the transactions). We show that no edge incident on T_1 is deleted from the TSG before T_i weakly terminates. By a similar argument, no edge incident on T_2 is deleted from the TSG before T_1 weakly terminates, and so on. Thus, we conclude that no edge incident on T_j is deleted from the TSG before T_i weakly terminates.

We now show that no edge incident on T_1 is deleted from the TSG before T_i weakly terminates. Due to the edge insertion rule, all of T_i's edges are inserted into the TSG before $ser(sub(T_i, s))$ executes. By the definition of serialization event, $ser(sub(T_i, s))$ executes before $ser(sub(T_1, s))$ since $sub(T_i, s)$ is serialized before $sub(T_1, s)$. Also, $ser(sub(T_1, s))$ executes before $sub(T_1, s)$ commits, and due to the edge deletion rule, $sub(T_1, s)$ commits before any of the edges incident on T_1 are deleted. Thus, all of T_i's and T_1's edges are inserted into the TSG before any of the edges incident on T_1 are deleted. If any of T_i's edges are deleted from the TSG before any of T_1's edges

Proof: Since S is an SRC schedule, S is serializable and every non-atomic transaction in S is compensated for. Consider a schedule S' resulting from the execution of transaction programs corresponding to atomic (committed) transactions in S such that the serialization order of transactions in S' is consistent with that in S. Since S' is serializable and consists of only atomic transactions, S' preserves database consistency. By the semantics of compensation, S preserves database consistency.

We now show that every transaction T_i in S sees a consistent database state. Let S result from the execution of transaction programs from a consistent database state DS, and $d = \bigcup_{s_k \in commit(T_i)} D_k$. Since transaction programs have no data dependencies, in order to show that T_i reads consistent data, we need to show that $state(T_i, d, DS, S)$ is consistent. Since S is an SRC schedule, S is serializable and for all non-atomic transactions T_j serialized before T_i in S^d such that $abort(T_j) \cap commit(T_i) \neq \emptyset$, CT_j is not serialized after T_i in S^d. Consider a schedule S' resulting from the execution, from database state DS, of transaction programs corresponding to

- atomic (committed) transactions in S, and
- non-atomic transactions T_j and their compensating transactions in S that satisfy the follwing condition: CT_j is serialized after T_i in S^d, and

such that the serialization order of transactions in S'^d is consistent with that in S^d. If T_i' denotes the transaction corresponding to T_i in S', then in S'^d, none of the non-atomic transactions serialized before T_i' abort at any of the sites in $commit(T_i)$. Thus, by Theorem 5, $state(T_i', d, DS, S')$ is consistent. By the semantics of compensation, $state(T_i, d, DS, S)$ is consistent. □

an earlier seat reservation transaction for flight 101 which now needs to be compensated for. The compensating transaction for the seat reservation transaction would be the cancellation of the seat reserved. Assume that just before the execution of the compensating transaction, flight 101 was overbooked, and $flag = true$. Also assume that, as a result of the cancellation, flight 101 is no longer overbooked. Thus, unless the compensating transaction sets $flag = false$ in addition to cancelling the seat, it would leave the database in an inconsistent state. As a result, the appropriate compensating transaction for the seat reservation transaction would be one which would cancel the seat, and then set $flag = false$ if the flight was no longer overbooked.

We assign the following semantics to compensating transactions in terms of database consistency. Let S be a non-atomic schedule (containing compensating transactions) resulting from the execution of transaction programs with no data dependencies from database state DS. Let d be an integral subset of D such that S^d is serializable and let $T_1, T_2, \ldots, T_n$ be a serialization order of transactions in S^d, where T_i is the transaction resulting from the execution of transaction program TP_i. Let T_i, for some i, $i = 1, 2, \ldots, n$ be a transaction such that for some j, $j = 1, 2, \ldots, i-2$, T_j is a non-atomic transaction, and for some k, $k = j+1, \ldots, i-1$, $T_k = CT_j$. Consider the schedule S' resulting from the execution of transaction programs in the order $TP_1, TP_2, \ldots, TP_{j-1}, TP_{j+1}, \ldots, TP_{k-1}, TP_{k+1}, \ldots, TP_i, \ldots, TP_n$ from database state DS. Let the serialization order of transactions in S' be $T'_1, T'_2, \ldots, T'_{j-1}, T'_{j+1}, \ldots, T'_{k-1}, T'_{k+1}, \ldots, T'_i, \ldots, T'_n$, where T'_i is the transaction resulting from the execution of transaction program TP_i and $commit(T_i)$ $commit(T'_i)$. If $state(T'_i, d, DS, S')$ is consistent, then $state(T_i, d, DS, S)$ is consistent. Note that the above assertion is much weaker than the assertion that the databases states seen by T_i and T'_i are the same. Informally, an execution S preserves database consistency if the execution S' preserves database consistency, where

- S' results from the execution of transaction programs corresponding to atomic transactions and non-atomic transactions that have not been compensated for in S,
- S' has a serialization order which is consistent with that in S, and
- transactions resulting from the execution of the same transaction program in both S and S' commit at the same sites.

Theorem 6: Every SRC schedule S resulting from the execution of transaction programs with no data dependencies preserves database consistency, and transactions in S see consistent database states.

cally realized in a schedule.

Definition 3: Let S be a schedule that results due to the execution of transaction programs from database state DS_1, and $d \subseteq D$ such that S^d is serializable. Let $T_1, T_2, \ldots, T_n$ be a serialization order of transactions in S^d. The state of the database before the execution of each transaction with respect to data items in d is defined as follows:

$$state(T_i, d, S, DS_1) = \begin{cases} DS_1^d & \text{if } i = 1 \\ DS_2^d, \text{ where } DS_1\{T_1, T_2, \ldots, T_{i-1}\}DS_2 & \text{if } i\rangle 1 \end{cases} \quad \square$$

$state(T_i, d, S, DS_1)$ is the state of the database with respect to data items in d as seen by T_i. The state of a transaction depends on the initial state and the serialization order chosen and thus, may not be unique. The following theorem states conditions under which non-atomic schedules preserve database consistency.

Theorem 5: Let S be a non-atomic schedule (containing no compensating transactions) resulting from the execution of transaction programs with no data dependencies from database state DS, and C be a set of sites. Let $d = \bigcup_{s_k \in C} D_k$ and S^d be serializable with serialization order $T_1, T_2, \ldots, T_n$. If DS^d is consistent, and for all j, $j = 1, 2, \ldots, i-1$, if $abort(T_j) \cap C = \emptyset$, then $state(T_i, d, S, DS)$ is consistent, for all i, $i = 1, 2, \ldots, n$.

Proof: We prove the theorem by induction on i.

Basis ($i = 1$): $state(T_1, d, S, DS) = DS^d$ is given to be consistent.

Induction: We assume the theorem is true for $i = r$. In order to show that the theorem is true for $i = r + 1$, we first use the induction hypothesis to conclude that $state(T_r, d, S, DS)$ is consistent. Since $abort(T_r) \cap C = \emptyset$ (given), $d \bigcup_{s_k \in C} D_k$ is an integral subset of D, and transaction programs have no data dependencies, by Lemma 1, $state(T_{r+1}, d, S, DS)$ is consistent. $\square$

We now consider non-atomic schedules containing compensating transactions. Compensation, as was mentioned earlier, is a semantically rich recovery paradigm which is used to undo committed transactions without resorting to cascading aborts, and to restore database consistency. A simple example would help in explaining our approach to compensation. Consider an airline database in which reservation of a seat, cancellation of a seat etc. constitute transactions. In addition, let there be the following integrity constraint: if $flag = true$, then flight 101 is overbooked. Consider

define the notion of data dependencies as follows.

Definition 2: A transaction program TP_i has no data dependencies if for all d such that d is an integral subset of D, all pairs (DS_1, DS_2) of database states such that $DS_1^d = DS_2^d$, and $DS_1\{TP_i\}DS_3$ and $DS_2\{TP_i\}DS_4$, the following is true: $DS_3^d = DS_4^d$. □

For a transaction program with no data dependencies and whose execution results in a non-atomic transaction, the following is true.

Lemma 1: Let T_i be a non-atomic transaction resulting from the execution of transaction program TP_i that has no data dependencies from database state DS_1, and d be an integral subset of D. If DS_1^d is consistent and $DS_1\{T_i\}DS_2$, then $DS_2^{(d-\bigcup_{s_k \in abort(T_i)} D_k)}$ is consistent.

Proof: Let $d' = d - \bigcup_{s_k \in abort(T_i)} D_k$, DS_3 be a consistent database state such that $DS_3^{d'} = DS_1^{d'}$ and $DS_3\{TP_i\}DS_4$. Note that d' is an integral subset of D. Since T_i results from the execution of TP_i from DS_1, T_i does not abort at any site s_k such that $D_k \subseteq d'$, and TP_i has no data dependencies, $DS_2^{d'} = DS_4^{d'}$. Since TP_i preserves database consistency, DS_4 is consistent. Thus, $DS_2^{(d-\bigcup_{s_k \in abort(T_i)} D_k)}$ is consistent. □

In Lemma 1, it is important that transaction program TP_i has no data dependencies. If TP_i has data dependencies, then Lemma 1 may not hold as is illustrated by the following example.

Example 3: Consider an MDBS consisting of sites s_1, s_2 and s_3. Let $D_1 = \{a\}$, $D_2 = \{b\}$ and $D_3 = \{c\}$. Let $IC = (c \rangle a) \wedge (c \rangle b) \wedge (a = b)$. Consider the following transaction program TP_1 that contains data dependencies.

$$TP_1: \quad c := a + 1$$

Consider an atomic transaction T_1 that results when TP_1 executes from database state $DS_1 = \{(a,1),(b,3),(c,5)\}$ (T_1 commits at both sites s_1 and s_3). Let $DS_1\{T_1\}DS_2$. Even though $DS_1^{\{b,c\}}$ is consistent, $DS_2^{\{b,c\}} = \{(b,3),(c,2)\}$ is inconsistent. □

We next introduce the notion of "state" which is a possible database state that a transaction might have seen. The state of a transaction is an abstract notion and may never have been physi-

-Appendix A-

We begin by first exploring conditions under which non-atomic schedules (that do not contain compensating transactions) preserve database consistency. We then extend these results to non-atomic schedules containing compensating transactions. We denote the set of data items at site s_k by D_k. We denote the set of all the data items in the MDBS by D; thus $D = \bigcup_{i=1}^{m} D_i$. We assume that the local databases are disjoint; that is, $D_i \cap D_j = \emptyset$, $i \neq j$. For each data item $d' \in D$, $Dom(d')$ denotes the domain of d'. A *database state* maps every data item d' to a value v', where $v' \in Dom(d')$. Thus, a database state, DS, can be expressed as a set of ordered pairs of data items in D and their values, $DS = \{(d', v') : d' \in D \text{ and } v' \in Dom(d')\}$[3].

Integrity constraints (denoted by IC) in a database distinguish inconsistent database states from consistent ones. A database state DS is consistent iff it satisfies the integrity constraints of the database. DS^d denotes the state of the database restricted to data items in the set d, where $d \subseteq D$. Thus, $DS^d = \{(d', v') : d' \in d \text{ and } (d', v') \in DS\}$. DS^d is consistent iff there exists a consistent database state DS_1 such that $DS_1^d = DS^d$. For example, consider a database consisting of data items a and b and $IC = (a = b)$. A database state $DS = \{(a, 5), (b, 6)\}$ is not consistent. However, $DS^{\{a\}} = \{(a, 5)\}$ is consistent and $DS^{\{b\}} = \{(b, 6)\}$ is consistent.

A transaction is a sequence of operations resulting from the execution of a *transaction program*. A transaction program is usually written in a high level programming language with assignments, loops, conditional statements and other complex control structures. Thus, execution of a transaction program starting at different database states may result in different transactions. A transaction program, when executed in isolation, is always assumed to preserve database consistency (the integrity constraints of the database). We use the notation $DS_1\{TP_i\}DS_2$ to denote the fact that execution of transaction program TP_i from database state DS_1 results in database state DS_2. Similar notation is used to denote execution of transactions and schedules (the intended meaning will be clear from the context). For a schedule S and a set of data items d, S^d denotes the restriction of S to operations that access data items in d. In addition, we use the following terminology. A set of data items d is an *integral subset* of D if, for some set of sites C, $d = \bigcup_{s_k \in C} D_k$. We formally

[3] DS has the property that if $(d', v_1') \in DS$ and $(d', v_2') \in DS$, then $v_1' = v_2'$.

[GRS91] D. Georgakopolous, M. Rusinkiewicz, and A. Sheth. On serializability of multidatabase transactions through forced local conflicts. In *Proceedings of the Seventh International Conference on Data Engineering, Kobe, Japan*, 1991.

[LKS91a] E. Levy, H. F. Korth, and A. Silberschatz. An optimistic commit protocol for distributed transaction management. In *Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data, Denver, Colorado*, pages 88–97, May 1991.

[LKS91b] E. Levy, H. F. Korth, and A. Silberschatz. A theory of relaxed atomicity. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1991.

[Pap86] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Rockville, Maryland, 1986.

[Pu88] C. Pu. Superdatabases for composition of heterogeneous databases. In *Proceedings of the Fourth International Conference on Data Engineering, Los Angeles*, 1988.

[Ske82] D. Skeen. Non-blocking commit protocols. In *Proceedings of ACM-SIGMOD 1982 International Conference on Management of Data, Orlando*, pages 133–147, 1982.

to relax some of the restrictions on transactions that need to be imposed if traditional recovery techniques were followed. However, since such executions may no longer consist of atomic transactions, thereby preservation of database consistency may be jeopardized. It was necessary for us to develop a new correctness criteria that ensures that transactions see consistent database states, and database consistency is preserved.

We also developed a new commit protocol and a new concurrency control scheme that ensures that all generated schedules are correct. The new commit protocol eliminates the problem of blocking, which is characteristic of the standard 2PC protocol. The concurrency control protocol we presented can be used in any MDBS environment irrespective of the concurrency control protocol followed by the local DBMSs in order to ensure serializability.

Acknowledgements.

We would like to thank Eliezer Levy for discussions that helped us develop a better understanding of compensation and its role in transaction management.

References

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[BS88] Y. Breitbart and A. Silberschatz. Multidatabase update issues. In *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data, Chicago*, pages 135–141, 1988.

[BST90] Y. Breitbart, A. Silberschatz, and G. R. Thompson. Reliable transaction management in a multidatabase system. In *Proceedings of ACM-SIGMOD 1990 International Conference on Management of Data, Atlantic City, New Jersey*, pages 215–224, 1990.

[DE89] W. Du and A. K. Elmagarmid. Quasi serializability: a correctness criterion for global concurrency control in InterBase. In *Proceedings of the Fifteenth International Conference on Very Large Databases, Amsterdam*, pages 347–355, 1989.

[ED90] A.K. Elmagarmid and W. Du. A paradigm for concurrency control in heterogeneous distributed database systems. In *Proceedings of the Sixth International Conference on Data Engineering*, 1990.

- If the insertion of T_i's edges into the TSG results in a violation of the edge insertion rule due to a transaction T_j that has not weakly terminated, then T_i waits for T_j.
- Let T_i and T_j have two sites in common at which they execute, T_j commits at one of the sites and aborts at the other. If T_j has not yet been compensated for, then the insertion of T_i's edges into the TSG results in a violation of the augmented edge insertion rule. As a result, T_i waits for CT_j.

It turns out that waits that result from the GTM protocol cannot result in a deadlock situation. Thus, if no waits are introduced by the local DBMSs, no deadlock can occur. The following theorem states this more precisely.

Theorem 4: If each of the local DBMSs follow a concurrency control protocol that does not require a transaction to wait for another transaction, then the GTM concurrency control protocol is deadlock-free.

Proof: See Appendix B. □

If the local DBMSs follow a concurrency control protocol that requires a transaction to wait on another transaction (e.g., 2PL), then deadlocks can occur. A transaction T_i may wait for another transaction T_j at the local site, while T_j may be waiting for T_i, since T_i's edges may have been inserted into the TSG and it may not be possible to insert T_j's edges until T_i's edges have either committed or aborted. These deadlocks can be detected and resolved by the GTM either by using timeouts or the scheme presented in [BST90].

6 Conclusion

We proposed a transaction model for MDBS applications that exploits the semantics of the transactions. Global subtransactions in our model are either one of the following types: *compensatable*— if the subtransaction commits, then it can be compensated for, *retriable*— if a subtransaction aborts, then it can be retried. Further, every global transaction has a pivot subtransaction that may be neither compensatable nor retriable.

Our enriched transaction model impacts the transaction management and recovery techniques employed in an MDBS environment. Instead of using traditional recovery techniques like redoing transactions from logs, we use compensation and retrying for recovery purposes. Thus, it is possible

- If there exists a transaction T_j (different from T_i) and distinct sites s_q and s_r such that $(T_j, s_q), (s_q, T_i), (T_i, s_r), (s_r, T_j)$ are edges in the cycle, then either (T_j, s_q) and (s_r, T_j) are both committed edges, or (T_j, s_q) and (s_r, T_j) are both aborted edges.

Augmented Edge Deletion Rule: Let τ be a set of transactions such that for any pair of transactions T_i, T_j, if $T_i \in \tau$ and T_j is connected to T_i by a path consisting of either committed or unmarked edges, then $T_j \in \tau$. If every transaction in τ has strongly terminated, then edges incident on all transactions in τ are deleted from the TSG.

Since a strongly terminated transaction is also weakly terminated, if the augmented edge deletion rule holds, then the edge deletion rule also holds. Thus, the augmented edge deletion rule is more restrictive than the edge deletion rule developed earlier.

Theorem 3: If every LTM ensures the serializability of local schedules, and the GTM commit protocol and the augmented edge management scheme are used, then every resulting schedule S is SRC.

Proof: See Appendix B. □

An efficient algorithm to ensure that the augmented edge management scheme is not violated can be found in Appendix C. Permitting the TSG to contain cycles, besides providing a higher degree of concurrency, is essential for the recovery of non-atomic transactions using compensation. Due to the augmented edge deletion rule, a non-atomic transaction's edges are not deleted from the TSG until it is compensated for, since otherwise non-SRC schedules may result. Further, the compensating transaction for a non-atomic transaction executes at those sites the non-atomic transaction has committed. As a result, it is essential that the TSG be permitted to contain cycles, since otherwise it may be impossible to compensate for a non-atomic transaction.

5.4 Deadlocks

We now shift our attention to the deadlock problem. Deadlocks within a site are handled by the local DBMS. Global deadlocks may occur if either the local DBMSs or the GTM follow a concurrency control protocol that may require transactions to wait for other transactions. In our scheme a transaction T_i is required to wait in the following two circumstances:

commits at both s_1 and s_2 (since G_1 does not abort in Example 2, and insertion of G_2's edges would violate the edge insertion rule). As a result, G_2 does not execute until G_1 completes execution, and thus the non-serializable schedule in Example 2 cannot result.

Edge Deletion Rule: Let τ be a set of transactions such that for any pair of transactions T_i, T_j, if $T_i \in \tau$ and T_j is connected to T_i by a path consisting of either committed or unmarked edges, then $T_j \in \tau$. If every transaction in τ has weakly terminated, then edges incident on all transactions in τ are deleted from the TSG.

It must be noted that if edges incident on any transaction in τ are deleted before all transactions in τ have weakly terminated, then non-serializable schedules may result.

Theorem 2: If every LTM ensures the serializability of local schedules, and the GTM follows the edge management scheme, then every global schedule is serializable.

Proof: See Appendix B. □

5.3 The Augmented Edge Management Scheme

The edge management scheme described above ensures that the resulting schedules are serializable. However, it does not guarantee that such schedules are SRC. For example, the schedule in Example 1 is serializable but not SRC; it could be generated by the above scheme as follows. The fund transfer transaction T_1 executes first and edges corresponding to it are inserted. After the GTM receives a message that T_1 at site s_2 is aborted, edge (T_1, s_2) is an aborted edge. Thus, the above edge insertion rule will allow the audit transaction T_2 to execute since the cycle caused by it in the TSG contains an aborted edge, resulting in a non-SRC schedule.

In order to ensure that schedules are SRC, the edge insertion and deletion rules are augmented as follows:

Augmented Edge Insertion Rule: For every cycle that results due to the insertion of T_i's edges, the following two requirements must hold:

- The edge insertion rule of the edge management scheme, is satisfied.

5.2 The Edge Management Scheme

We now present the rules used by the GTM for deciding when edges can be safely inserted and deleted in the TSG. In contrast to the scheme used in [BST90], our scheme permits the TSG to contain cycles.

In order to describe the scheme we need to define the following terminology. If the GTM receives a $\langle ack\_commit, T_i \rangle$ message from the cohort at site s_j, then edge (T_i, s_j) is referred to as a *committed edge*. Similarly, if the GTM receives a $\langle ack\_abort, T_i \rangle$ message from the cohort at site s_j, edge (T_i, s_j) is referred to as an *aborted edge*. If the GTM has not received any message from the cohort at site s_j, then edge (T_i, s_j) is referred to as an *unmarked* edge. Thus, edge (T_i, s_j) is a committed edge only if T_{ij} has committed at site s_j. If edge (T_i, s_j) is an aborted edge but not a committed edge, then T_{ij} must have aborted at site s_j. Further, if edge (T_i, s_j) is both a committed and an aborted edge, then the compensating transaction for T_{ij} has committed at site s_j.

Edges of a transaction T_i are inserted into the TSG only if the insertion of T_i's edges does not violate the *edge insertion rule* described below. The GTM inserts edges belonging to only one transaction at a time.

Edge Insertion Rule: For every cycle that results due to the insertion of T_i's edges at least one of the following condition holds:

1. The cycle contains an edge that is aborted but not committed.
2. There exists transactions T_j, T_k (different from T_i) and distinct sites s_q, s_r such that (T_j, s_q), (s_q, T_i), (T_i, s_r) and (s_r, T_k) are edges in the cycle, and both (T_j, s_q), (s_r, T_k) are committed edges.

The edge insertion rule ensures that cycles in the TSG do not cause cycles in the *serialization graph* [BHG87]. To see this, consider a cycle in the TSG that satisfies condition 1. Since an aborted subtransaction does not conflict with any other transaction, such a cycle does not cause a cycle in the serialization graph. For a cycle in the TSG that satisfies condition 2 we see that, since $ser(T_i)$ occurs after $ser(T_j)$ at site s_q, and $ser(T_i)$ occurs after $ser(T_k)$ at site s_r, T_i is serialized after T_j and T_k at s_q and s_r respectively. Thus, such a cycle does not cause a cycle in the serialization graph.

In Example 2, if the GTM follows the edge insertion rule, G_1's edges are first inserted into the TSG before G_1 executes at site s_1. Further, G_2's edges are not inserted into the TSG until G_1

Edges in the TSG are inserted as a result of the execution of certain *serialization events* [ED90]. For a given concurrency control protocol, the serialization event, ser, is a function that maps transactions to operations such that, for any pair of transactions T_i and T_j in a schedule that results from the protocol, if T_i is serialized before T_j, then $ser(T_i)$ executes before $ser(T_j)$ in the schedule[2]. Further, for every transaction T_i, $ser(T_i)$ does not execute after T_i commits. For example, for the timestamp ordering scheme, $ser(T_i)$ is the operation that results in transaction T_i being assigned a timestamp. Similarly, in case of 2PL, $ser(T_i)$ is the operation that results in T_i obtaining its last lock.

Serialization events may not exist for certain protocols (e.g., serialization graph testing) [Pu88]. For such protocols, serialization events can be introduced by forcing conflicts between transactions [GRS91]. For example, we can require that every transaction update a particular data item, say, *ticket*. If some transaction T_i is serialized before another transaction T_j, then T_i must have updated *ticket* before T_j updated it. Thus, $ser(T_i)$ is the write operation of transaction T_i on *ticket*.

We require that all the edges of a global transaction T_i will be inserted in the TSG before $ser(T_{ik})$ is submitted to the server at s_k, for any $s_k \in exec(T_i)$. For example, consider a global transaction T_1 that executes at sites s_1 and s_2 which follow the 2PL protocol and a timestamp ordering scheme respectively. Further, suppose that in the timestamp ordering scheme followed by the LTM at s_2, a timestamp is assigned to a transaction before any of its operations execute. Since s_1 follows the 2PL protocol and s_2 follows the timestamp ordering protocol, $ser(T_{11})$ is the operation that results in T_{11} obtaining its last lock, and $ser(T_{12})$ is the first operation of T_{12}. In our scheme, the GTM inserts both of T_1's edges into the TSG before submitting the last database operation of T_{11}, and the first operation of T_{12} to the respective servers.

Note that as the compensating transaction CT_i corresponding to a non-atomic transaction T_i is also considered as a global transaction, edges corresponding to CT_i must also be inserted into the TSG. Let CT_i consist of subtransactions $CT_{i1}, CT_{i2}, \ldots, CT_{ir}$, where $s_1, s_2, \ldots, s_r$ are the sites in $commit(T_i)$. Since the GTM does not control the execution of CT_{ij} (which are directly executed by the server on receipt of an $\langle abort, T_i \rangle$ message from the GTM), the GTM inserts edges corresponding to CT_i before dispatching the $\langle abort, T_i \rangle$ message to the servers at the sites in $commit(T_i)$.

[2]For a given protocol, various operations may satisfy the property required of a serialization event. We assume that one of them is chosen to be ser.

strongly terminated are SRC. Our protocol involves insertion and deletion of edges from a graph. We first develop an edge management scheme that ensures schedules consisting of global, local and compensating transactions are serializable. We then augment the edge management scheme developed in order to ensure that schedules are SRC.

5.1 The Transaction-Site Graph

Since local transactions execute outside the control of the GTM, the GTM may be unaware of the indirect conflicts between global transactions at the local DBMSs due to local transactions. This may result in non-serializable executions.

Example 2: Consider an MDBS environment consisting of two sites: s_1 with data items a and b, and s_2 with data items c and d. Suppose that each local DBMS uses the 2PL protocol to ensure serializability. Let:

$$\begin{array}{lll} G_1: & w_1(a) & w_1(c) \\ G_2: & w_2(b) & w_2(d) \end{array}$$

be two global transactions, and let:

$$\begin{array}{lll} L_3: & r_3(a) & r_3(b) \\ L_4: & r_4(c) & r_4(d) \end{array}$$

be two local transactions (L_3 at s_1, and L_4 at s_2).

Consider an execution in which transaction G_1 first executes at site s_1 followed by the execution of L_3. G_2 then executes at both s_1 and s_2, followed by L_4. Finally, G_1 executes at site s_2 resulting in the following non-serializable schedule (we denote the local schedules at sites s_1 and s_2 by S_1 and S_2 respectively).

$$\begin{array}{llllll} S_1: & w_1(a) & r_3(a) & r_3(b) & w_2(b) & \\ S_2: & w_2(d) & r_4(c) & r_4(d) & w_1(c) & \square \end{array}$$

In order to prevent such non-serializable schedules, the GTM maintains a graph, called the *transaction-site graph* (TSG), which is similar to the commit graph presented in [BST90]. A TSG is an undirected bipartite graph consisting of nodes corresponding to local sites (site nodes) and global transactions (transaction nodes). Edges in the TSG may be present only between transaction nodes and site nodes. An edge between a transaction node T_i and a site node s_j indicates that $s_j \in exec(T_i)$, and is denoted by either (s_j, T_i) or (T_i, s_j). Edges (T_i, s_k) in the TSG, for all sites $s_k \in exec(T_i)$, are referred to as either T_i's edges, or edges incident on T_i.

a correctness criterion for non-atomic schedules called *serializable with respect to compensation* (SRC). In order to define SRC, we need to define the following notation. The set of sites at which a global transaction T_i executes is denoted by $exec(T_i)$. The sites at which T_i commits and aborts are denoted by $commit(T_i)$ and $abort(T_i)$, respectively. Note that $exec(T_i) = commit(T_i) \cup abort(T_i)$. Let $s_1, s_2, \ldots, s_r$ be the sites on which a non-atomic transaction T_i commits; thus $s_j \in commit(T_i)$, where $1 \leq j \leq r$. Let CT_{ij} be the compensating transaction that executes to undo semantically the effects of T_{ij}. For the purpose of defining SRC, $CT_{i1}, CT_{i2}, \ldots, CT_{ir}$ are considered as subtransactions of a global transaction CT_i, and CT_i is referred to as the compensating transaction corresponding to T_i.

Definition 1: A non-atomic schedule S is *serializable with respect to compensation* (SRC) if all of the following hold.

- For each non-atomic transaction T_i in S, there exists a compensating transaction CT_i that is committed in S.
- S is serializable.
- Let T_j be an arbitrary global transaction in S, and $S^{commit(T_j)}$ denote the projection of S on the data items at sites at which T_j commits. For all non-atomic transactions T_i serialized before T_j in $S^{commit(T_j)}$, if CT_i is serialized after T_j in $S^{commit(T_j)}$, then $abort(T_i) \cap commit(T_j) = \emptyset$. □

Note that in Example 1, in which the audit transaction T_2 sees an inconsistent database state, $commit(T_2) = \{s_1, s_2\}$. Further, since T_1 (the fund transfer transaction) aborts at s_2, we have $s_2 \in abort(T_1)$. Thus, as T_2 at site s_1 is serialized between T_1 and CT_1, and $abort(T_1) \cap commit(T_2) \neq \emptyset$, the resulting serializable schedule is not SRC. In an SRC schedule the audit transaction will execute only after the debit subtransaction is compensated, and thus will see a consistent database state.

It can be shown that that if the schedule S is SRC, then each transaction sees a consistent database state. The proof of this claim is substantial and can be found in Appendix A.

5 The GTM Concurrency Control Protocol

In this section, we present a GTM concurrency control protocol that irrespective of the concurrency control protocol followed by the local DBMSs, ensures that schedules in which every transaction has

4 Correctness of Non-atomic Schedules

A global schedule S contains local, global, and compensating transactions. Consider a global schedule S in which each of the global transactions has strongly terminated. Schedule S may contain transactions which are *non-atomic*. A non-atomic transaction can be more easily defined by stating what constitutes an *atomic* transaction. A transaction T_i in a schedule S is atomic if either of the following hold.

- T_i is committed in S (in a distributed system, all subtransactions of T_i are committed in S).
- If T_i is aborted in S (in a distributed system if any of T_i's subtransactions are aborted in S), then it does not have *any* effect on the execution of other transactions in S or on the final database state.

A non-atomic transaction is one that does not satisfy the above two conditions. Consider a partially committed global transaction T_1 that commits at site s_1 but aborts at s_2. Depending upon the type of subtransaction T_{11} and T_{12}, either T_{12} is retried, or T_{11} is compensated for. Let S be the global schedule in which T_1 has strongly terminated. If T_{12} is retried, then T_1 is atomic (since all its subtransactions are committed). However, if T_{11} is compensated, then T_1 is non-atomic (since it is committed at some sites but aborted at others). We refer to a schedule that contains non-atomic transactions as a *non-atomic* schedule. In [LKS91b] it was shown that even though a non-atomic schedule S is serializable, and the commit protocol ensures semantic atomicity, it is possible that certain transactions "see" an inconsistent state of the database. To see this, consider the following example, from our banking enterprise domain.

Example 1: Let T_1 be a transaction that transfers money from an account A at site s_1 to an account B at site s_2 and consists of a debit subtransaction T_{11} and a credit subtransaction T_{12}. Consider the scenario in which T_{11} commits but T_{12} aborts. Let T_2 be an audit transaction that now executes (before T_{11} is compensated for by crediting account A); T_2 reads the balances in both accounts A and B, and sees a database state in which the sum of the balances of accounts A and B is less than the actual sum. This situation is clearly unacceptable. □

To prevent transactions from "seeing" an inconsistent database state, we must place restrictions on the concurrency permitted in the system. To develop these restrictions we must first introduce

schedules CT_{ij} for execution, where CT_{ij} is a compensating transaction corresponding to the subtransaction executing at the site. On commitment of CT_{ij} at the local DBMS, the cohort sends a $\langle ack\_abort, T_i \rangle$ message to the GTM.

In the above protocol, if any of the compensatable subtransactions or the pivot subtransaction aborts, the GTM aborts the global transaction. If the pivot subtransaction commits, then the GTM commits the global transaction. It should be noted that it is possible for the GTM to receive a $\langle ack\_commit, T_i \rangle$ message followed by a $\langle ack\_abort, T_i \rangle$ message from a c-cohort. However, from a p-cohort or a r-cohort, the GTM only receives either a $\langle ack\_commit, T_i \rangle$ or a $\langle ack\_abort, T_i \rangle$ message.

Transaction T_i is said to have *strongly terminated* if the GTM receives either a $\langle ack\_commit, T_i \rangle$ message from every cohort, or a $\langle ack\_abort, T_i \rangle$ messages from every cohort. It is said to have *weakly terminated* if the GTM receives either a $\langle ack\_commit, T_i \rangle$ or a $\langle ack\_abort, T_i \rangle$ messages from every cohort. Thus, a strongly terminated transaction is also weakly terminated.

At various stages of the GTM commit protocol, processes are required to wait for messages before progressing. This, in presence of communication and site failures could potentially result in blocking. However, the problem is easily alleviated by using a timeout scheme. If a server is interrupted by a timeout while waiting for a message from the GTM, it assumes that the GTM process has failed and submits an abort operation to the local DBMS, thereby releasing the resources held by the transaction. Since the GTM does not manage any data items directly we do not specify any timeout actions for the GTM. Note, that it is possible that a retriable subtransaction that has been aborted by the server on timeout may need to be retried if in case the GTM commits the transaction. The GTM commit protocol, thus, does not cause blocking of local applications. Also, if there are no failures, the protocol requires $2n$ messages and 6 rounds as compared to $3n$ messages and 3 rounds needed by the standard 2PC protocol [BHG87], where n is the number of sites on which the transaction executes.

Theorem 1: The GTM commit protocol preserves semantic atomicity of transactions.

Proof: See Appendix B. □

- When the GTM receives $\langle ack\_commit, T_i \rangle$ messages from all the c-cohorts, it sends a $\langle commit,$ message to the p-cohort. If, however, it receives at least one $\langle ack\_abort, T_i \rangle$ message from any of the c-cohorts, it aborts the global transaction, and sends all cohorts a $\langle abort, T_i \rangle$ message.

- When the p-cohort receives a $\langle commit, T_i \rangle$ message from the GTM, it submits the commit operation for T_i to the local DBMS. On receiving an acknowledgement from the local DBMS that the subtransaction has committed at the local DBMS, it sends a $\langle ack\_commit, T_i \rangle$ message to the GTM. If, however, the pivot subtransaction is aborted by the local DBMS, it sends a $\langle ack\_abort, T_i \rangle$ message to the GTM.

Phase 3:

- When the GTM receives a $\langle ack\_commit, T_i \rangle$ message from the p-cohort, it commits the global transaction and sends $\langle commit, T_i \rangle$ messages to each of the r-cohorts. If, however, it receives a $\langle ack\_abort, T_i \rangle$ message from the p-cohort, it aborts the global transaction, and sends all cohorts $\langle abort, T_i \rangle$ messages.

- When a r-cohort receives a $\langle commit, T_i \rangle$ message from the GTM, it submits the commit operation for T_i to the local DBMS. In case the local DBMS aborts the subtransaction, it retries the subtransaction until it is committed at the local DBMS. When the subtransaction finally commits at the local DBMS, it sends a $\langle ack\_commit, T_i \rangle$ message to the GTM.

The above protocol specifies the actions taken by a cohort when it receives the $\langle commit, T_i \rangle$ message from the GTM. It also specifies the actions the GTM takes when it receives either a $\langle ack\_commit, T_i \rangle$ or a $\langle ack\_abort, T_i \rangle$ message from each of the cohorts. We now specify the actions taken by the cohorts when they receive a $\langle abort, T_i \rangle$ message from the GTM.

- If the subtransaction has been aborted by the local DBMS, then the cohort sends a $\langle ack\_abort, T_i \rangle$ message to the GTM (if it has not already done so).

- If the subtransaction has neither been aborted nor committed by the local DBMS, the cohort submits the abort operation for T_i to the local DBMS. On receiving an acknowledgement from the local DBMS that the subtransaction has been aborted, it sends a $\langle ack\_abort, T_i \rangle$ message to the GTM.

- If the subtransaction has been committed by the local DBMS (note that only c-cohorts can receive a $\langle abort, T_i \rangle$ message from the GTM after T_i has committed at the local DBMS), it

other transactions or the final database state). Transactions that transfer money between accounts belonging to the same site are local transactions as are those that deposit or withdraw money from an account.

3 The GTM Commit Protocol

The GTM commit protocol must ensure that either all subtransactions of a global transaction are committed (i.e., they are either committed or retried), or all subtransactions are undone (i.e., they are either aborted by the LTM or are compensated for). A commit protocol that ensures the above property of transactions is said to preserve *semantic atomicity* [LKS91a]. Since retriable subtransactions of a global transaction in our model may not be compensatable, and compensatable subtransactions may not be retriable, the GTM commit protocol must control the order in which the subtransactions are committed. The protocol consists of three phases, each of which deals with the commit of one of the subtransaction types. The commit protocol is invoked once all the subtransactions of a global transaction have completed execution.

To describe the protocol we must first define some terminology. The servers at sites at which a global transaction T_i executes are referred to as T_i's *cohorts*. The server process executing at the site at which T_i's pivot subtransaction executes is referred to as the *p-cohort*. Similarly, servers at sites on which compensatable and retriable subtransactions execute are referred to as *c-cohorts* and *r-cohorts* respectively.

We are in a position now to define the three phases in the GTM commit protocol, which are:

Phase 1:

- The GTM sends each c-cohort a $\langle commit, T_i \rangle$ message.
- When a c-cohort receives the $\langle commit, T_i \rangle$ message from the GTM, it submits the *commit* operation for T_i to the local DBMS. On receiving an acknowledgement from the local DBMS that the subtransaction has committed, the c-cohort sends a $\langle ack\_commit, T_i \rangle$ message to the GTM. If, however, the subtransaction is aborted by the local DBMS, it sends an $\langle ack\_abort, T_i \rangle$ message to the GTM.

Phase 2:

A global transaction has at most one pivot subtransaction. As in [DE89], we assume that no data dependencies exist between the subtransactions of a global transaction; that is, the execution of a global transaction at one site is independent of its execution at other sites.

With each compensatable subtransaction, a *compensating transaction* is associated. Compensating transactions are transactions that restore database consistency by semantically undoing committed transactions, without resorting to cascading aborts [LKS91a]. Let T_i be a global transaction and T_{ij} be a compensatable subtransaction of T_i that committed at site s_j. To undo the effects of T_{ij}, a compensating transaction for T_{ij}, denoted by CT_{ij}, is executed. CT_{ij} follows T_{ij} in the execution, is a separate transaction from T_{ij}, and is always serialized after T_{ij} in any schedule. Executing CT_{ij}, however, does not guarantee that all the effects of T_{ij} are undone and thus ensures only a weaker form of atomicity [LKS91a]. In our model, we further assume the following about compensating transactions:

- Since no data dependencies exist between subtransactions of a global transaction, CT_{ij} executes only at the site at which T_{ij} commits.
- CT_{ij} may itself be aborted by the local DBMSs, but if retried a sufficient number of times, it eventually succeeds.
- CT_{ij} is independent of the transactions that execute between T_{ij} and CT_{ij} in the schedule. It depends only on T_{ij}, and the integrity constraints of the database.

We now illustrate the expressive power of above developed transaction model by applying it to a banking enterprise. In such an environment, transfer of money between accounts, audits that return the current balance in accounts, deposits and withdrawals from accounts, constitute transactions that can be modeled using our scheme. For example, transactions that transfer money between accounts belonging to different sites can be modeled as global transactions with two subtransactions, one which credits a bank account, and another which debits a bank account. The credit subtransaction is retriable, while the debit subtransaction is compensatable (the compensating transaction for a debit transaction is a credit transaction, which can be assumed to succeed if retried a sufficient number of times[1]). Similarly, an audit transaction can be modeled as a global transaction, all of whose subtransactions are compensatable (the compensating transaction for a read only transaction does nothing, since a read only transaction has no effects on the execution of

[1] In case the account is deleted, we assume that an exception is raised, and the money is mailed directly to the account-holder.

2 The MDBS Model

A heterogeneous distributed database consists of a set of autonomous pre-existing centralized local database systems located at sites $s_1, s_2, \ldots, s_m$ respectively. Transactions, in our model, are a sequence of read and write operations followed by either a commit operation or an abort operation. A local schedule consists of a sequence of operations resulting from the concurrent execution of transactions at a site. A global schedule is a *distributed schedule* [Pap86] consisting of operations belonging to transactions (global and local) with a partial order on them. The LTM at each site s_i ensures the atomicity of transactions and serializability of local schedules at s_i.

The execution of global transactions is carried out by the GTM which communicates with the LTMs by means of a *server* process that executes at each site on top of the local DBMSs. We assume that the interface between a server and an LTM provides for operations to be submitted by the server to the LTM, and the LTM to acknowledge the completion of operations to the server. The GTM does not schedule an operation belonging to a global transaction for execution unless it receives an acknowledgement from the server that the previous operation of the global transaction has been executed at the local site. We also assume that the GTM is centrally located, and the sites at which a global transaction executes are known to the GTM *a priori*. In addition, the LTM does not distinguish between local transactions and global subtransactions executing at its site.

In order to exploit the semantic recovery options of compensation and retrial we develop an extended transaction model. Each global transaction, in our model, consists of a number of subtransactions, each of which is one of the following:

- **Compensatable:** a subtransaction whose execution at the site can be undone by running a compensating transaction after it commits. For example, a subtransaction that reserves a seat in an airline reservation system can be compensated for by a subtransaction that cancels the reservation.

- **Retriable:** a subtransaction that can be retried and eventually succeeds if retried a sufficient number of times. Cancellation of a seat in an airline reservation system, or crediting a bank account, are examples of retriable subtransactions.

- **Pivot:** a subtransaction that is neither retriable nor compensatable.

refer to such a global transaction as a *partially committed transaction*. Partially committed transactions may result in the loss of consistency of the MDBS. The proposed approaches to deal with this problem can be characterized as being either *forward* or *backward* in nature. The forward approach was first suggested in [BST90], where the aborted subtransactions of a partially committed transaction are redone from logs maintained by the GTM. Since the GTM has no control over the execution of local transactions, certain local transactions may execute before the GTM completes the redo of an aborted global subtransaction resulting in a non-serializable execution [BST90]. Thus, in order to ensure both atomicity and serializability, restrictions are placed on the data items read and updated by global and local transactions.

The backward approach was adopted in [LKS91a] in the context of a general distributed system to alleviate the problem of *blocking*. It utilizes compensating transactions to undo the effects of committed subtransactions of a partially committed transaction. Since compensation only guarantees a weaker form of atomicity, the authors recognize the need for, and propose a correctness criterion for executions in which compensation is used for recovery purposes.

In this paper, we develop a fault-tolerant transaction management scheme for an MDBS environment that combines both the forward and backward approaches. In contrast to the scheme developed in [BST90], where redo logs were used to restore database consistency, our forward approach is based upon retrying of the appropriate aborted subtransactions. Our combined recovery approach allows us to relax some of the restrictions imposed on which data items the transactions can read and update [BST90]. A correctness criterion similar to the one in [LKS91a] is adopted. We also present new commit and concurrency control protocols used by the GTM that ensure the correctness of executions and do not violate the local autonomy of sites.

The remainder of the paper is organized as follows. In Section 2, we introduce the MDBS model and the global transaction model based on retriable and compensatable types of transactions. Section 3 discusses the GTM commit protocol. In Section 4, a correctness criterion for executions containing compensating transactions is proposed. In Section 5, we present the GTM concurrency control protocol and show that the protocol ensures resulting schedules are correct. Section 6 contains concluding remarks.

- **Local transactions,** those transactions that execute at a single site, and execute outside the control of the GTM.

- **Global transactions,** those transactions that may execute at several sites, and execute under the control of the GTM. Each global transaction consists of a set of subtransactions each of which is executed as a local transaction.

A distinguishing feature of MDBSs is the requirement that each local DBMS preserve its local autonomy. In this paper, as in [DE89], local autonomy is defined to consist of:

- **Design Autonomy.** The local sites are free to follow any concurrency control protocol in order to ensure local database consistency.

- **Communication Autonomy.** The LTMs do not communicate any information (e.g., conflict graph) relevant for concurrency control or transaction management to the GTM.

- **Execution Autonomy.** Each LTM has complete control over those transactions that are executing at its site. Thus, the LTM is free to abort a transaction as long as the transaction in question has not committed yet.

The execution autonomy requirement is essential in an MDBS environment since local DBMSs may not, in general, permit transactions to hold onto resources, or execute, for an unbounded period of time. This requirement, however, has a serious impact on the way the atomicity of global transactions in an MDBS environment can be achieved [BST90]. For example, the two-phase commit (2PC) protocol, which is the standard protocol used to ensure the atomicity of global transactions [BHG87] cannot be used. The main problem with using 2PC protocol, or any other atomic commit protocol (e.g., three phase commit protocol [Ske82]), in an MDBS environment is that such protocols require that in presence of failures, a global subtransaction in the *prepared state* be allowed to hold onto resources for an unbounded period of time [Ske82]. As a result, local transaction may be blocked to ensure atomicity of global transactions which is clearly unacceptable due to the execution autonomy requirement. Furthermore, since the pre-existing local DBMSs may not provide for a prepared state, substantial changes may need to be made to the existing DBMS software to support an atomic commit protocol, the result is the violation of the design autonomy!

In the absence of an atomic commit protocol it is possible that certain subtransactions of a global transaction abort, whereas others commit, thereby violating the atomicity property. We

A Transaction Model for Multidatabase Systems*

Sharad Mehrotra
Rajeev Rastogi
Henry F. Korth†
Abraham Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188 USA

Abstract

A multidatabase system (MDBS), consists of a number of sites, each of which runs a distinct commercial database management system (DBMS). The goal of an MDBS is to integrate the various DBMSs to allow applications to access data from several DBMSs, without requiring modifications to the individual DBMSs. This implies that each site is allowed a high degree of local autonomy. This autonomy requirement makes the task of ensuring both, the atomicity and isolation properties of transactions, in the presence of failures, difficult. In this paper, we develop a semantically rich transaction model for MDBS applications. We relax the atomicity requirement on transactions and propose a new suitable correctness criterion. We also develop new commit and concurrency control protocols that ensure correctness and do not violate the local autonomy of the various sites.

1 Introduction

The problem of transaction management in a multidatabase system (MDBS) has received considerable attention from the database community in recent years (e.g., [BS88], [BST90], [ED90], [GRS91]). The basic problem is to integrate a number of pre-existing local database management systems (DBMSs) located at different sites, into an MDBS environment that allows transactions to access data residing at multiple sites. Each local DBMS has a *local transaction manager* (LTM) which is responsible for ensuring local database consistency. The *global transaction manager* (GTM), built on top of the existing databases, is responsible for ensuring global database consistency.

Transactions in an MDBS are of two types:

*Work partially supported by NSF grants IRI-8805215, IRI-9003341, grants from the IBM corporation, and the NEC corporation.

†Current address: Matsushita Information Technology Laboratory, 182 Nassau street, Princeton, NJ 08542-7072

A TRANSACTION MODEL FOR MULTIDATABASE SYSTEMS

Sharad Mehrotra
Rajeev Rastogi
Henry F. Korth
Avi Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-92-14 March 1992

DEPARTMENT OF COMPUTER SCIENCES
THE UNIVERSITY OF TEXAS AT AUSTIN

AUSTIN, TEXAS 78712