Implementing Semantic-Based Decomposition of Transactions*

Sushil Jajodia, Indrakshi Ray and Paul Ammann

Center for Secure Information Systems
George Mason University, Fairfax, VA 22030-4444
Email: {jajodia, indrakshi, pammann}@gmu.edu

Abstract. In some database applications, performance requirements are not satisfied by the traditional approach of serializability, in which transactions appear to execute atomically and in isolation on a consistent database state. Although many researchers have investigated the process of decomposing transactions into steps to increase concurrency, the focus of the research is typically on implementing a decomposition supplied by the database application developer, with relatively little attention to what constitutes a desirable decomposition and how the developer should obtain such a decomposition. In our research, we focus on the decomposition process itself.
In [2], we introduced the notion of semantic histories and identified a number of properties which must be satisfied by a decomposition if the decomposition correctly models the original collection of transactions. We also formulated the notion of successor sets to describe efficiently the correct interleavings of steps. In this paper, we develop the successor set constraints more fully, and show how they can be used to take full advantage of conflict serializability at the level of steps. We give a graph-based characterization of correctness of successor set histories, and show how a verified decomposition can be implemented in a two-phase locking environment. We also discuss how the problems related to starvation, deadlocks, and recovery could be addressed.

1 Introduction

Performance requirements can force a transaction to be decomposed into smaller logical units (*steps*), especially if the transaction is long-lived. Consider the simple example of making a hotel reservation. The reserve transaction might consist of ensuring that there are still rooms vacant, selecting a vacant room that matches the customer's preferences, and recording billing information. Since the reserve transaction might last a relatively long time – when the customer makes reservations by phone – an implementation might force the three steps in the reserve transaction to occur separately.

The traditional transaction model relies on the properties of *atomicity*, *consistency*, and *isolation*. Decomposing transactions into steps generally forces one to relinquish

* This work was partially supported by DARPA under grant number N0060–96–D–3202. The work of S. Jajodia was also supported by NSF under grant number IRI–9633541 and by NSA under grants MDA904-96-1-0103 and MDA904-96-1-0104. The work of I. Ray was also supported by a George Mason University Graduate Research Fellowship Award.

these three properties to some degree. Decomposition not only sacrifices atomicity, since atomicity of the single logical action is lost, but impacts consistency and isolation as well. Execution of a step may leave the database in an inconsistent state, which may be viewed by other transactions or steps; therefore, it is necessary to reason about the interleavings of the steps of different transactions. Although the step-by-step decomposition of a single transaction may be understood in isolation, reasoning about the interleaving of these steps with other transactions, possibly also decomposed into steps, is substantially more difficult. To reason about interleavings, in [2] we introduced the notion of *semantic* histories which not only list the sequence of steps forming the history, but also convey information regarding the state of the database before and after execution of each step in the history. We identified several properties which semantic histories must satisfy to show that a particular decomposition correctly models the original collection of transactions.

In this paper, we take a closer look at the notion of successor sets which was formulated in [2] to describe efficiently the correct interleavings of steps. We give a graph-based characterization of correctness of successor set histories. We then give a two-phase locking implementation for the verified decomposition that takes full advantage of conflict serializability at the level of steps.

The remainder of the paper is organized as follows. A motivating example is presented in Section 2. Section 3 gives our model and illustrates it with refinements to the motivating example. Successor sets, which are crucial to an efficient implementation of our decomposition, are presented in Section 7. A graph-based characterization of correctness is given in Section 8, followed by an implementation in a two-phase locking environment in Section 9. Section 10 discusses how the problems related to starvation, deadlocks, and recovery could be addressed. Section 11 relates our work to the literature. Section 12 concludes the paper.

2 The Hotel Database

We use a running example of a hotel database to explain our ideas. The hotel database has a set of rooms. The status of each room is either *Available* or *Unavailable*. The object *ST* stores the status information of each room. The objects *res* and *total* denote the number of reservations and the total number of rooms in the hotel respectively. The object *RM* stores information that relates the guests to the assigned rooms. In this example, we do not allow guests to register multiple times.

The hotel database has two integrity constraints: (1) The number of guests assigned rooms equals the number of reservations. (2) The set of rooms that are unavailable is exactly the set of rooms assigned to guests.

There are three types of transactions in the hotel database – *Reserve*, *Cancel* and *Report*. The *Reserve* transaction checks to ensure that a room is available and the guest does not have a reservation, then selects a room for assignment and changes its status to *Unavailable* and assigns the room to the guest. The *Cancel* transaction checks if the guest has a reservation, then decrements *res*, changes the status of the room assigned to the guest to *Available*, and removes the guest to room assignment from *RM*. The *Report* transaction prints *ST* and *RM*.

3 The Model

In our model, a *database* is specified as a set of objects, along with some *invariants* or *integrity constraints* on these objects. At any given time, the *state* is determined by the values of the objects in the database. A change in the value of a database object changes the state. The invariants are predicates defined over the objects in the state. A database state is said to be *consistent* if the set of values satisfies the given invariants. A *transaction* is an operation that takes the database from one consistent state to another.

Let I denote the original invariants, and let $\mathbf{ST}$ denote the set consisting of all consistent states; i.e., $\mathbf{ST} = \{ST : ST \text{ satisfies } I\}$. A transaction T_i always operates on a consistent $ST \in \mathbf{ST}$; the state after the execution of T_i is also in $\mathbf{ST}$. However, when T_i is broken up into steps $< T_{i1}, T_{i2}, \ldots, T_{in} >$, each step T_{ij} is executed as an atomic operation. If ST_{ij} represents the partial execution of T_i, it is possible that after execution of step T_{ij}, the resulting database state ST_{ij} no longer satisfies the invariants I and, therefore, lies outside $\mathbf{ST}$. Once the invariants are violated, the formal basis for assessing the correctness of subsequent behavior collapses.

In our approach, we define a new set of invariants, $\hat{I}$, by relaxing the original invariants I. We decompose each transaction such that execution of any step results in a database state that satisfies $\hat{I}$. In addition, if all the steps of a transaction are executed on an initial state that satisfies the original invariants, then the final state also satisfies the original invariants. Let $\widehat{\mathbf{ST}} = \{ \text{ST} : \text{ST satisfies } \hat{I}\}$. $\mathbf{ST}$ and $\widehat{\mathbf{ST}}$ are related as follows: $\mathbf{ST} \subset \widehat{\mathbf{ST}}$. Formalizing $\widehat{\mathbf{ST}}$ allows us to investigate activities in inconsistent but acceptable states.

4 Correctness Criteria for Semantic-based Decompositions

In this section, we briefly review the relevant notions related to semantics histories. We refer the reader to [2] for additional details and their justification.

A *decomposition* of a transaction T_i is a sequence of two or more atomic *steps* $< T_{i1}, T_{i2}, \ldots, T_{in} >$. In place of T_i, these steps are executed in the given order as atomic operations on a database state. We make a distinction between a type of a step and an instance of a step. Histories, defined subsequently, reflect actual transactions, and must reference instances of steps. In general, a history may contain many instances of a step of a given type. We use the notation T_{ij} to denote an instance of a step.

Definition 1. [Stepwise Serial History] A *stepwise serial history H* over a set of transactions $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$ is a sequence of steps such that

1. for each $T_i \in \mathbf{T}$, a step of T_i either appears exactly once in H or does not appear at all,
2. for any two steps T_{ij}, T_{ik} of some $T_i \in \mathbf{T}$, T_{ij} precedes T_{ik} in H if T_{ij} precedes T_{ik} in T_i, and
3. if $T_{ij} \in H$, then $T_{ik} \in H$ for $1 \leq k < j$.

By Condition (1), we ensure that every step of a transaction should occur at most once in a stepwise serial history. Condition (2) ensures that the order of the steps in a transaction is preserved in the stepwise serial history. Condition (3) ensures that for every step in a stepwise serial history, all the preceding steps in the corresponding transaction are present in the history.

Definition 2. [Complete Execution] An execution of a transaction $T_i = < T_{i1}, T_{i2}, \ldots, T_{in} >$ in a history H is a *complete* execution if all n steps of T_i appear in H.

To emphasize the fact that we view the database from a semantic perspective, we define the term *semantic history*.

Definition 3. [Semantic History] A *semantic* history H is a stepwise serial history that is bound to (1) an initial state, and (2) the states resulting from the execution of each step in H. A semantic history H_p over $\mathbf{T}$ is a *partial* semantic history if the execution of some transaction T_i is not complete in H_p. A semantic history H over $\mathbf{T}$ is a *complete* semantic history if the execution of each T_i in $\mathbf{T}$ is *complete*.

Definition 4. [Correct Partial Semantic History] A partial semantic history H_p is a *correct* partial semantic history if

1. the initial state is in $\mathbf{ST}$,
2. all states before and after the execution of each step in H_p are in $\widehat{\mathbf{ST}}$, and
3. preconditions for each step are satisfied before it is executed.

Definition 5. [Correct Complete Semantic History] A complete semantic history H is a *correct* complete semantic history if

1. H is a correct partial semantic history, and
2. the final state is in $\mathbf{ST}$.

In a correct semantic history the steps of transactions are executed serially. To improve the throughput, steps must be executed concurrently. Later, in Section 8, we present our notion of correct concurrent executions.

5 Properties of a Valid Decomposition

With the notion of correctness in place, we can state the property relating steps in a decomposition to the original transaction. We call this requirement the *composition property*. Formally:

Composition Property: Let T_i denote the original transaction and T_{i1}, T_{i2}, …, T_{in} denote the corresponding steps. Executing the steps $T_{i1}, T_{i2}, \ldots, T_{in}$ serially on a state satisfying the original invariants I, changes the original database objects in the same way as executing the original transaction T_i on the same state.

Similar to the consistency property for traditional databases, we place the following requirement on semantic histories:

Consistent Execution Property: If we execute a complete semantic history H on an initial state (i.e., the state prior to the execution of any step in H) that satisfies the original invariants I, then the final state (i.e., the state after the execution of the last step in H) also satisfies the original invariants I.

In our model, we allow steps or transactions to see database states that do not satisfy the original invariants (i.e., states in $\widehat{\mathbf{ST}} - \mathbf{ST}$). But we may wish to keep some transactions from viewing any inconsistency with respect to the original invariants. For example, some transactions may output data to users; these transactions are referred to as *sensitive* transactions in [6]. We require sensitive transactions to appear to have generated outputs from a consistent state.

Sensitive Transaction Isolation Property: All output data produced by a sensitive transaction T_i should have the appearance that it is based on a consistent state in **ST**, even though T_i may be running on a database state in $\widehat{\mathbf{ST}} - \mathbf{ST}$.

The fourth property which we describe is the *complete execution property*. When transactions have been broken up into steps, the interleaving of steps may lead to deadlock (i.e., a state from which we cannot complete some partially executed transaction). The complete execution property ensures that deadlock is avoided; if a transaction has been partially executed, then it can eventually complete.

Complete Execution Property: Every partial correct semantic history H_p is a prefix of some complete correct semantic history.

6 A Valid Decomposition

In [2], we provide a valid decomposition of the hotel database. It satisfies all the properties identified in the previous section.

To make the invariants more general, we add the auxiliary variable *tempreserved*, which is a natural number, to denote the reservations that have been partially processed. We also add the auxiliary variable *tempassigned*, which is a set of rooms, to denote the rooms that have been reserved but which have not yet been assigned to guests. The generalized invariants are as follows: (1) The number of reservations (*res*) equals the number of guests assigned rooms plus the number of reservations in progress (*tempreserved*). (2) The set of rooms with status *Unavailable* equals the set of rooms assigned to the guests union the set of rooms in *tempassigned*.

We decompose the reserve transaction into steps $R1$, $R2$ and $R3$. $R1$ checks if *res* is less than *total*, and then increments *res* and *tempreserved*. $R2$ selects an *Available* room and changes its status to *Unavailable* and includes this room in the set *tempassigned*. $R3$ checks if the guest does not have an existing reservation, then it assigns the room to the guest, decrements *tempreserved*, removes the room from the set *tempassigned*. The single step of *Cancel* transaction, denoted by *ValidCancel*, is nearly identical to the original transaction. *Report* is a sensitive transaction, and we establish the sensitive transaction isolation property by construction. *Report* transaction outputs values of *ST* and *RM*. *ST* and *RM* involve the following original invariant: The set of rooms with status *Unavailable* equals the set of rooms assigned to the guests. This original invariant is satisfied when the auxiliary variable *tempassigned* is empty. To ensure consistent output, we add the extra precondition, *tempassigned* is empty, in step *ValidReport* which is the only step of *Report* transaction.

7 Successor Sets

Decomposing transactions into steps yields improved performance, but the interleaving of these steps must be constrained so as to avoid inconsistencies. In the decomposition we have given so far, the interleaving is constrained by additional preconditions on the auxiliary variables. Although auxiliary variables facilitate analysis, it is expensive to implement them. Also performing additional precondition checks involves extra run time overhead. To avoid implementing auxiliary variables and performing additional precondition checks we introduce the concept of successor sets.

Before formally introducing successor sets, let us consider one of the problems associated with the interleaving of steps of different transactions in an implementation. This will help us to establish the necessary background for our notion of successor sets. Suppose a transaction T_i introduces an inconsistency in step T_{ij} and removes the inconsistency in some later step T_{ik}. Another transaction, say T_{pq}, is not allowed to see the inconsistency introduced by step T_{ij}. Now, T_{pq} will see an inconsistency caused by T_{ij} only if T_{pq} tries to read or write a variable which has been modified by T_{ij}. In other words, T_{pq} must conflict with T_{ij}. In such a case T_{pq} should not be allowed to execute after step $T_{ij}, T_{i(j+1)}, \ldots, T_{i(k-1)}$ as shown below.

$$T_{i1}\ \overbrace{T_{i2}\ \ldots\ T_{i(j-1)}}^{T_{pq}\ can\ execute}\ T_{ij}\ \underbrace{T_{i(j+1)}\ T_{i(j+2)}\ \ldots\ T_{i(k-1)}}_{T_{pq}\ cannot\ execute}\ T_{ik}\ \overbrace{T_{i(k+1)}\ \ldots\ T_{in)}}^{T_{pq}\ can\ execute}$$

Note that this goal can be achieved if we implement the steps obtained using the decomposition based on generalized invariants. In the generalized invariant scheme, preconditions are used to control the interleaving of steps of different transactions, and in any history, resulting from the implementation of the generalized invariant scheme, the preconditions involving auxiliary variables is false if T_{pq} appears between steps T_{ij} and T_{ik}. In other words, preconditions involving auxiliary variable are false if we try to execute T_{pq} after T_{il}, where T_{pq} conflicts with T_{il} or any step previous to T_{il}. However, as mentioned earlier, such a scheme will be undesirably costly.

We now formally introduce our notion of successor sets.

Definition 6. [Successor Set] The *successor set* of $ty(T_{ij})$, denoted $SS(ty(T_{ij}))$, is a set of types of steps.

Note that, at this point, the notion of successor sets is purely syntactic. Subsequently, we define the constraints under which a successor set description is correct with respect to a particular decomposition. But first we define correct successor set histories.

Definition 7. [Correct Successor Set History] Let H be a correct semantic history. H is a correct successor set history if it satisfies the following additional requirement: If T_i is incomplete in the prefix of H that ends at T_{pq}, and T_{ij} is the last step in T_i such that (1) T_{ij} conflicts with T_{pq} and (2) T_{ij} precedes T_{pq} in H, then $ty(T_{pq}) \in SS(ty(T_{ij}))$.

The above successor set rule enforces the requirement that in any correct successor set history, the step T_{pq} must be in the successor set of step T_{ij} where T_{ij} is the last step of T_i which conflicts with and precedes step T_{pq}.

Example 1. Successor set descriptions are obtained by examining the preconditions with auxiliary variables. In the hotel example, the only precondition with auxiliary variable involves checking whether *tempassigned* is empty in step *ValidReport*. This precondition is satisfied as long as a step of type *ValidReport* does not appear between a step of type $R2$ and a step of type $R3$ of reserve transaction. The successor sets of $R1$ and $R2$ are given below.

$$SS(R1) = \{R1,\ R2,\ R3,\ ValidCancel,\ ValidReport\}$$
$$SS(R2) = \{R1,\ R2,\ R3,\ ValidCancel\}$$

The successor set for $R1$ includes every other possible type of step; it is possible to execute a step of type $R1$, and then execute a step of any other type $R1$, $R2$, $R3$, *ValidCancel*, *ValidReport*. The successor set for $R2$ is more restrictive. $ValidReport \notin SS(R2)$ means that any step of type *ValidReport* cannot execute after step of type $R2$ if a step of type *ValidReport* conflicts with a step of type $R2$ or $R1$. In other words, *ValidReport* is not allowed to the see the inconsistencies with respect to the original invariants that are introduced by a step of type $R2$. Also note that after the last step of a transaction has been executed, it will be possible to execute any step of any transaction; thus the successor set of $R3$, *ValidReport*, *ValidCancel* contain all types of steps.

With respect to the specifications given with auxiliary variables, not all successor set descriptions are correct. We describe correct successor set descriptions with the *valid successor set property*:

Definition 8. [Valid Successor Set Property] A specification S_2 that uses successor set description is *valid* with respect to specification S_1 with generalized invariants if

1. Any correct semantic history generated by S_2 is also a correct semantic history generated by S_1.
2. S_2 satisfies the complete execution property.

If the sets of correct semantic histories generated by S_1 and S_2 are identical, then it follows that S_2 enjoys the complete execution property if and only if S_1 does. In the case where S_2 generates fewer correct semantic histories than S_1 – a byproduct of the successor set descriptions of S_2 being less expressive than the first order logic preconditions of S_1 – the complete execution property needs to be explicitly reverified with respect to S_2.

Before concluding this section, we give one more definition. Consider the following scenario. Suppose that in a partial correct semantic history step T_{ij} of transaction T_i has been executed, and suppose we wish to execute step T_{pq}, where T_{pq} conflicts with T_{ij} or some step previous to T_{ij}. The *next* function, defined below, gives the earliest step in T_i after T_{ij} where T_{pq} is allowed to execute such that the resulting history remains correct. (The *next* function is exactly the F function in [5].)

Definition 9. [Next Function] The *next function*, denoted by $NF(T_{ij}, T_{pq})$, gives the first step T_{ik} of T_i in the sequence T_{ij}, $T_{i(j+1)}, \ldots, T_{i(j+n)}$ such that $ty(T_{pq}) \in SS(ty(T_{ik}))$.

Stated more formally, $T_{ik} = NF(T_{ij}, T_{pq})$ if the following conditions hold: (i) $ty(T_{pq}) \in SS(ty(T_{ik}))$, $k \geq j$ and (ii) for each step T_{ir} appearing between T_{ij} and T_{ik} in T_i (if any), $ty(T_{pq}) \notin SS(ty(T_{ir}))$.

Suppose a step T_{kl} conflicts with a step T_{ij} and suppose $ty(T_{kl}) \notin SS(ty(T_{ij}))$. Let $NF(T_{ij}, T_{kl}) = T_{ip}$. T_{kl} is not allowed to execute after the steps T_{ij}, $T_{i(j+1)}$, $T_{i(j+2)}$, $\ldots$ $T_{i(p-1)}$, but T_{kl} is allowed to execute after T_{ip}. It must be the case that T_{ip} alters the database state in such a way that T_{kl} can execute. Thus T_{kl} and T_{ij} must conflict. We capture this property as an explicit constraint in our model.

Constraint 10. If T_{kl} conflicts with T_{ij} or some step in T_i previous to T_{ij} and $ty(T_{kl}) \notin SS(ty(T_{ij}))$, then T_{kl} also conflicts with $NF(T_{ij}, T_{kl})$.

8 Correct Concurrent Executions

In a correct successor set history all operations of one step must appear before any operation of any other step. However, if steps execute atomically and without interleaving the system makes poor use of system resources. The standard solution, which we adopt, is to increase the class of allowable histories to include the histories that are conflict equivalent to correct successor set histories. We call these histories *correct stepwise serializable histories*.

We show how to decide whether a history is a correct stepwise serializable history by analyzing a graph, called a precedence graph, derived from that history.

Definition 11. [Precedence Graph] Let H be a history defined over a set of transactions $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$. The *precedence graph* of H, denoted by $PG(H)$, is a directed graph where the nodes are the steps of the transactions in $\mathbf{T}$ and the edges graph are of three types constructed as follows:

1. Internal Edges – For each pair of consecutive steps T_{ij} and $T_{i(j+1)}$ of transaction T_i, there is an *internal* or *I* edge (T_{ij},$T_{i(j+1)}$).
2. Conflict Edges – For each pair of conflicting steps T_{ij} and T_{kl} belonging to different transactions T_i and T_k, there is a *conflict* or *C* edge (T_{ij},T_{kl}) if there is an operation in T_{ij} that conflicts with and precedes another operation in T_{kl}.
3. Successor Edges – For each pair of steps T_{ij} and T_{kl}, belonging to different transactions T_i and T_k respectively, such that there is a conflict edge from T_{ij} to T_{kl}, there is a successor edge ($NF(T_{ij}, T_{kl})$,T_{kl}).

Observation 12. For a correct successor set history H, if there is an edge (T_{ij}, T_{kl}) in $PG(H)$ then T_{ij} precedes T_{kl}. It follows that for a correct successor set history H, the precedence graph $PG(H)$ is acyclic.

Theorem 13. *A history H is a correct stepwise serializable history iff $PG(H)$ is acyclic.*

Proof. $\Leftarrow$: Since $PG(H)$ is acyclic it can be topologically sorted. Let H_c be any topological sort of $PG(H)$. We prove H_c is a correct successor set history by contradiction. Suppose H_c is not a correct successor set history. There must be some step T_{pq} which interleaves with transaction T_i such that T_{ij} is the last step in T_i conflicting with and preceding T_{pq}, and $ty(T_{pq}) \notin SS(ty(T_{ij}))$. Let $NF(T_{ij}, T_{pq}) = T_{ik}$. Since $ty(T_{pq}) \notin SS(ty(T_{ij}))$, T_{ik} is some step after T_{ij}. In $PG(H)$ there must be a successor edge (T_{ik}, T_{pq}) corresponding to the conflict edge (T_{ij}, T_{pq}). By Constraint 10 T_{pq} and T_{ik} conflict. As T_{ij}

is the last operation which conflicts and precedes T_{pq}, T_{pq} must precede T_{ik}, and the edge (T_{pq}, T_{ik}) is in H. As figure 1 shows, the result is a cycle in $PG(H)$ – a contradiction. The assumption that H_c is not a correct successor set history is wrong. Since H is equivalent to H_c, H is also a correct stepwise serializable history.

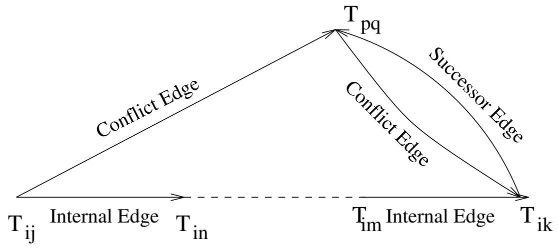

Fig. 1. Edges of Precedence Graph involving T_{ij},T_{ik},T_{pq}

$\Rightarrow$: Since H is a correct stepwise serializable history, it must be equivalent to some correct semantic history H'. Let $PG(H')$ be the precedence graph of the history H'. Since H and H' are equivalent, the internal edges and the conflict edges in $PG(H')$ are the same as the corresponding edges in $PG(H)$. Successor edge ($NF(T_{ij}$,$T_{kl})$,T_{kl}) is added if there is a conflict edge (T_{ij}, T_{kl}) where $i \neq k$. Since the set of internal edges and conflict edges are the same for $PG(H)$ and $PG(H')$ and the successor sets are the same for H and H', the set of successor edges is the same for $PG(H)$ and $PG(H')$. Thus $PG(H) = PG(H')$. Since $PG(H')$ is acyclic (Observation 12), the result follows.

9 Concurrency Control Mechanism

We now propose a mechanism based on a two-phase locking environment. First, we describe some notation. A step is a sequence of read and write operations followed by a commit operation: $T_{ij} = O_{ij}(x_1), O_{ij}(x_2), \ldots, O_{ij}(x_n), C_{ij}$, where $O_{ij}(x)$ is either $R_{ij}(x)$ or $W_{ij}(x)$. A transaction is a sequence of steps followed by a termination operation: $T_i =< T_{i1}, T_{i2}, \ldots, T_{in}, TR(T_i) >$.

9.1 Data Structures

In addition to the data structures required by the standard two phase locking protocol, we require the following data structures: (i) Active-Set - Set of Active Transactions and (ii) Int-Set - Interleaving Sets.

Active-Set(x) – The active set for x keeps the list of all active transactions that have accessed x. Whenever any step T_{ij} that reads or writes x commits, the transaction T_i is added to Active-set(x). After the transaction T_i terminates, T_i is removed from the Active-Set(x).

Int-Set(T_i,x) - The interleaving set for x is associated with each active transaction T_i that accesses x. The interleaving set gives the types of the steps that can access the data item. If data item x has been accessed by step T_{ij} of T_i and T_{ij} or any step occurring after T_{ij} commits, then Int-set(T_i,x) is replaced by the successor set of the corresponding committed step.

9.2 Algorithms

The mechanism requires the following assumptions

1. Lock management is centralized.
2. The steps of a transaction are submitted in order; that is, an operation in step $T_{r(s+1)}$ is submitted only after step T_{rs} commits.
3. If a transaction reads and writes the same data entity x, the read operation precedes the write operation.
4. A transaction reads or writes an entity x at most once.
5. The algorithms specified below execute atomically.

Algorithm 14. Algorithm for Read

Procedure Process-read (R_{ij}(x))
begin
    if a step T_{lm} is holding an exclusive lock on x
        exit;   /* Lock unavailable - T_{ij} can retry later */
    for each $T_k \in$ Active-set(x)
        if ty(T_{ij}) $\notin$ Int-set(T$_k$,x)
            exit; /* Lock unavailable - T_{ij} can retry later */
    lock x in shared mode;
    accept(R_{ij}(x));
end

Algorithm 15. Algorithm for Write

Procedure Process-write (W_{ij}(x))
begin
    if a step T_{lm} is holding any lock on x
        exit;   /* Lock unavailable - T_{ij} can retry later */
    for each $T_k \in$ Active-set(x)
        if ty(T_{ij}) $\notin$ Int-set(T_k,x)
            exit; /* Lock unavailable - T_{ij} can retry later */
    lock x in exclusive mode
    accept(W_{ij}(x));
end

Algorithm 16. Algorithm for Step Commit

Procedure Process-stepcommit(C_{ij})
begin
  for each x locked by the transaction in this or previous step **do**
    Int-set(T_i,x) = SS(ty(T_{ij}));
  for each entity x locked by the transaction in this step **do**
    begin
      if $T_i \notin$ Active-set(x)
        Active-set(x) = Active-set(x) $\cup T_i$;
      Release the lock on x which was acquired by T_{ij};
    end
end

Algorithm 17. Algorithm for Transaction Terminate

Procedure Process-terminate($TR(T_i)$)
begin
  for each entity x which was accessed by T_i **do**
    begin
      Active-set(x) = Active-set(x) $-$ T_i;
      delete the structure Int-set(T_i,x) ;
    end
end

In section 8 we imposed Constraint 10 on successor set descriptions. Constraint 10 requires step $NF(T_{ij}, T_{pq})$ and step T_{pq} to conflict if T_{pq} and T_{ij} conflict and $ty(T_{pq}) \notin SS(ty(T_{ij}))$. It turns out that our particular mechanism does not require Constraint 10 on successor set descriptions as it implements this constraint independent of whether the successor set description enjoys the property.

Theorem 18. *Any history generated by our mechanism is a correct stepwise serializable history.*

We omit the proof due to lack of space.

10 Other Issues

In this section, we discuss how problems related to starvation, deadlocks, and recovery could be addressed.

10.1 Starvation

In our mechanism we do not specify how to grant locks if multiple steps are contending for the same lock. If the scheme for granting lock requests is unfair, starvation may occur. The problem of starvation can be handled at the specification level or at the implementation level. To keep the specification simple, we recommend handling the problem of starvation at the implementation level. The problem of starvation can be solved using the standard techniques [4]. For example we can associate a FIFO queue with each data item; steps are allowed to lock the item only in the order in which they have requested a lock on the data item. Other mechanisms are possible in which priorities are assigned to steps; locks are assigned on the basis of priorities. The priority of a step increases the longer it waits and so eventually it will be possible for a particular step to acquire locks.

10.2 Deadlock

There are two reasons why a step T_{ij} may be blocked because of step T_{kl} in our mechanism, thus causing deadlock: (1) T_{ij} wants to lock an item which is already locked by T_{kl} in a conflicting mode, or, (2) $ty(T_{ij}) \notin SS(ty(T_{k(l-1)}))$ and $ty(T_{ij}) \in SS(ty(T_{kl}))$; hence step T_{ij} cannot proceed until step T_{kl} completes.

We provide a modified wait-for graphs to detect deadlocks in our mechanism. The nodes in the MWFG correspond to active steps (a step is active from the time it tries to acquire the first lock to the time it completes). The edges in graph are of two types: (i) L edges (waiting for Lock edges) - the directed edge (T_{ij}, T_{kl}) indicates that T_{ij} cannot execute because it is waiting for a lock held by T_{kl}, and (ii) S edges (waiting for Successor edges) - the directed edge (T_{ij}, T_{kl}) indicates that T_{ij} cannot execute until T_{kl} completes because $ty(T_{ij}) \in SS(ty(T_{kl}))$ and $ty(T_{ij}) \notin SS(ty(T_{k(l-1)}))$.

The following algorithm can be used to generate the Modified Wait-For Graph:

1. The following operations are performed when step T_{pq} becomes active.
   (a) it checks the MWFG to see if there is a node corresponding to T_{pq}; if the node is absent the wait-for graph is updated to include this node.
   (b) T_{pq} checks in the wait-for list to see if an S edge has to be drawn from T_{ij} to T_{pq}; if yes, this edge is drawn in MWFG.
2. The following operations are performed when T_{pq} requests for a lock on data item x and the lock request cannot be granted.
   (a) If T_{pq} cannot get a lock on x because some T_{kl} has locked the data item in conflicting mode draw the L edge (T_{pq}, T_{kl}) in MWFG.
   (b) If T_{pq} cannot lock x because $ty(T_{pq}) \notin$ Int-Set(T_r,x) (this is because step $ty(T_{pq}) \notin SS(ty(T_{rs}))$ where T_{rs} is the last step of T_r to have completed execution) then the S edge $(T_{pq}, NF(T_{rs}, T_{pq}))$ must be drawn in the wait-for graph. However at this point it may not be possible to draw this edge because the step $NF(T_{rs}, T_{pq})$ may not be active and there may not be any node corresponding to this step. For this reason we have a list, called the wait-for list, which stores the list of edges that have to be drawn when the respective nodes are created.
3. The following operations are executed when T_{pq} terminates.
   (a) The node corresponding to T_{pq} is deleted.
   (b) All edges directed at node T_{pq} is deleted.

The MWFG must be periodically checked for the presence of cycles. A cycle in the MWFG indicates the presence of a deadlock and one or more steps in the cycle must be aborted; the steps to be aborted are known as the victims. The process of victim selection needs to be investigated. Nodes in the MWFG corresponding to steps that are currently being executed are chosen as victims. The aborted victims must be re-executed later on.

10.3 Recovery Issues

In this paper, we assume that once transactions are initiated they can be successfully completed. However, this may not always be true. For example, while the transaction is being executed, the system may crash. A complicating factor is that when a system crash occurs the transaction may have committed some but not all of its steps. In such a scenario, executing the traditional undo and redo [3] on the steps often does not restore the database to a consistent state.

To solve this problem we propose using an additional data structure which stores the state of active transactions. We term this data structure as the table of active transactions. This table contains an entry for each active transaction. When a transaction commits a step, it updates the corresponding entry to indicate the last committed step of the transaction. Once the transaction terminates, the entry corresponding to this transaction is deleted. This data structure must be stored in the stable storage so that its contents can survive system crashes. Once the system comes up after a system crash, this table is consulted to find out the transactions that were active at the time of the crash. These transactions can then be completed and the database restored to a consistent state.

11 Related Work

The work on semantic based concurrency control can be classified into two major categories. In the first category [9, 10, 12, 13] the authors exploit the semantics of operations to increase concurrency. Instead of using low level database operations like read or write to access the database objects, the authors propose the use of high level operations for this purpose. Commutativity of these operations, and not the read/write operations, is used to determine conflicts between transactions, resulting in more concurrency. In these works, the authors use serializability as the correctness criterion.

Our work falls in the second category [1, 5, 6, 7] which is based on exploiting semantics of transactions to increase concurrency. In these works, the researchers decomposed transactions into steps and developed semantic based correctness criteria. Researchers have variously introduced the notions of transaction steps, countersteps, allowed vs. prohibited interleavings of steps, and implementations in locking environments. The focus is typically on implementing a decomposition supplied by the database application developer, with relatively little attention to what constitutes a desirable decomposition and how the developer should obtain such a decomposition. We find the decomposition process itself to be worthy of attention, so we give the developer a model in which to decompose transactions, and we define properties to assure the developer as to the soundness of a given decomposition. Only then do we consider the problem of implementing our decomposition in a two-phase locking environment.

12 Conclusion

In some applications transactions that ideally should be treated as atomic – for reasons of analysis – must instead be treated as a composition of steps – for reasons of performance. In a previous paper [2] we show how to decompose transactions into atomic steps and assess the correctness of the decomposition. In this paper we have provided a two-phase locking mechanism in section 9 that ensures execution histories that are stepwise conflict-serializable and also respect the successor set constraints. We still must address the problem of reliably transmitting parameters between steps of a transaction, a problem that is considered in [8, 11]. However, the semantic aspects of our implementation have been thoroughly addressed.

References

1. D. Agrawal, A. El Abbadi, and A. K. Singh. Consistency and orderability: Semantics-based correctness criteria for databases. *ACM Transactions on Database Systems*, 18(3):460–486, September 1993.
2. P. Ammann, S. Jajodia, and I. Ray. Using formal methods to reason about semantics-based decomposition of transactions. In *Proceedings of the International Conference on Very Large Data Bases*, pages 218–227, Zurich, Switzerland, September 1995.
3. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.
4. R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, CA, 1989.
5. A. A. Farrag and M. T. Özsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.
6. H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.
7. H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 249–259, San Francisco, CA, 1987.
8. H. Garcia-Molina and K. Salem. Services for a workflow management system. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 17(1):40–44, March 1994.
9. M. Herlihy. Extending multiversion time-stamping protocols to exploit type information. *IEEE Transactions on Computers*, 36(4):443–448, April 1987.
10. M. P. Herlihy and W. E. Weihl. Hybrid concurrency control for abstract data types. *Journal of Computer and System Sciences*, 43(1):25–61, August 1991.
11. H. Wachter and A. Reuter. The ConTract model. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219–263. Morgan Kauffman, San Mateo, CA, 1992.
12. W. E. Weihl. *Specification and Implementation of Atomic Data Types*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1984.
13. W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

This article was processed using the LaTeX macro package with LLNCS style