1

SEMANTIC-BASED DECOMPOSITION OF TRANSACTIONS

Paul Ammann, Sushil Jajodia, Indrakshi Ray

Center for Secure Information Systems
and
Department of Information and Software Systems Engineering
George Mason University
Fairfax, VA 22030-4444

ABSTRACT

In some database applications the traditional approach of serializability, in which transactions appear to execute atomically and in isolation on a consistent database state, fails to satisfy performance requirements. Although many researchers have investigated the process of decomposing transactions into steps to increase concurrency, such research typically focuses on providing algorithms necessary to implement a decomposition supplied by the database application developer and pays relatively little attention to what constitutes a desirable decomposition or how the developer should obtain one. We focus on the decomposition itself. A decomposition generates proof obligations whose discharge ensures desirable properties with respect to the original collection of transactions. We introduce the notion of semantic histories to formulate and prove the necessary properties, and the notion of successor sets to describe efficiently the correct interleavings of steps. The successor set constraints use information about conflicts between steps so as to take full advantage of conflict serializability at the level of steps. We propose a mechanism based on two-phase locking to generate correct stepwise serializable histories.

1 INTRODUCTION

Performance requirements can force an application developer to decompose some transactions into smaller logical units or *steps*, especially for long-lived transactions. Consider the simple example of making a hotel reservation. A 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 – for example, when the customer makes reservations by phone – it may be desirable to execute the three steps of the reserve transaction separately. Researchers have long recognized the need to decompose transactions into steps. Our contribution is to formalize a decomposition process and identify desirable properties. We show how to assess these properties for a specific application by enumerating and discharging the proof obligations that accompany each phase of the decomposition. We describe an efficient, two-phase locking mechanism for the resulting, verified decomposition.

The traditional transaction model relies on the properties of *atomicity*, *consistency*, and *isolation*. Atomicity ensures that either all actions of a transaction complete successfully or all of the transaction's effects are absent. Consistency ensures that a transaction when executed by itself, without interference from other transactions, maps the database from one consistent state to another. Isolation ensures that no transaction ever views the partial effects of some other transaction even when transactions execute concurrently. Decomposing transactions into steps generally forces one to relinquish these three properties to some degree.

Decomposition of a transaction into steps sacrifices atomicity since the atomicity of the single logical action is lost. Interleaving steps of transactions exposes each to the partial effects of the others. Hence, if a transaction aborts after committing some of its steps, it may not be possible to remove all of its effects. This difficulty arises because transactions that read from the aborted transaction may have committed. In addition, the aborted transaction may have generated outputs. Thus traditional undo [4] is not possible; the solution is to semantically undo the actions of the aborted transactions. We achieve semantic undo with compensating steps [8, 9].

Decomposition not only sacrifices atomicity, but also impacts consistency and isolation. Execution of a step may leave the database in an inconsistent state, which other transactions or steps may access, so it is necessary to reason about the interleavings of the steps of different transactions. Although the step-by-step decomposition of a single transaction into steps may be understood easily in isolation, reasoning about the interleaving of these steps with other transactions, possibly also decomposed into steps, is more difficult. To reason about interleavings, we introduce 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 identify properties that semantic histories must satisfy for a particular decomposition to be acceptable, both when considered by itself, and also with respect to the original set of transactions.

We adopt the Object Z specification language [5] for expressing model-based specifications. Object Z is an extension of the Z specification language [14] to include object oriented features. Object Z is based on set theory, first order predicate logic, and a schema calculus to organize large specifications. Knowledge of Object Z is helpful, but not required, for reading this paper, since we narrate the formal specifications in English. Table 1 briefly explains the Object Z notation used in our examples. Other specification and analysis conventions peculiar to Object Z are explained as the need arises.

$\mathbb{N}$	Set of Natural Numbers
$\mathbb{P}\, A$	Powerset of Set A
$\# A$	Cardinality of Set A
$\setminus$	Set Difference (Also schema 'hiding')
$A \mathbin{;} B$	Forward Composition of A with B
$x \mapsto y$	Ordered Pair $(x,\ y)$
$A \nrightarrow B$	Partial Function from A to B
$A \rightarrowtail\!\!\!\!\!\!\nrightarrow B$	Partial Injective Function from A to B
$B \lhd A$	Relation A with Set B Removed from Domain
$A \rhd B$	Relation A with Range Restricted to Set B
$\mathrm{dom}\, A$	Domain of Relation A
$\mathrm{ran}\, A$	Range of Relation A
$A \oplus B$	Function A Overridden with Function B
$x?$	Variable $x?$ is an Input
$x!$	Variable $x!$ is an Output
x	State Variable x before an Operation
x'	State Variable x' after an Operation
Δx	Before and After State of Variable x
$\Box$	Temporal Operator Always
$\Diamond$	Temporal Operator Eventually
$\bigcirc$	Temporal Operator Next
op	Operation

Table 1 Relevant Object Z Notation

2 RELATED WORK

The work on semantic based concurrency control can be classified into two major categories. In the first category [10, 11, 15, 16] 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 using high level operations to access these objects. 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 [2, 7, 8, 9] 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.

3 THE HOTEL DATABASE

We illustrate our ideas with an example of a hotel database. An Object Z specification appears in figure 1. The specification assumes two types, *Guest* and *Room*, which enumerate all possible guests and all possible hotel rooms, respectively. The class *Hotel* describes the objects, the integrity constraints on these objects, and the transactions of the hotel database.

The objects may be constants or variables. The hotel database has a constant *total* which is the number of rooms in the hotel. The hotel database has three variables, namely, res, RM and ST. In Object Z states, as well as operations, are described with a two-dimensional notation called a *schema*. The state schema is nameless and contains declarations of the variables in the top part and constraints on these variables appear in the bottom. The natural number

$[Guest, Room]$

$Status ::= Available \mid Taken$

Hotel

$$total : \mathbb{N}$$

$$\begin{array}{l} res : \mathbb{N} \\ ST : Room \pfun Status \\ RM : Guest \pinj Room \\ \hline \# RM = res \\ \mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}\ RM \end{array}$$

Reserve

$$\begin{array}{l} \Delta(res, ST, RM);\ g? : Guest;\ r! : Room \\ \hline res < total;\ g? \notin \mathrm{dom}\ RM \\ ST(r!) = Available \\ res' = res + 1 \\ ST' = ST \oplus \{r! \mapsto Taken\} \\ RM' = RM \cup \{g? \mapsto r!\} \end{array}$$

Cancel

$$\begin{array}{l} \Delta(res, RM, ST) \\ g? : Guest \\ \hline g? \in \mathrm{dom}\ RM \\ res' = res - 1 \\ RM' = \{g?\} \ndres RM \\ ST' = ST \oplus \{RM(g?) \mapsto Available\} \end{array}$$

Report

$$\begin{array}{l} currentST! : Room \pfun Status \\ currentRM! : Guest \pinj Room \\ \hline currentST! = ST \\ currentRM! = RM \end{array}$$

Figure 1 Initial Specification of the Hotel Database

res counts current reservations, and the partial injection *RM* relates guests to rooms. Our particular example does not allow guests to register multiple times, which is reflected in the fact that *RM* is an injective function. The example could be modified easily with different constraints. The partial function *ST* records the status of each room. Additional integrity constraints on the objects in hotel database appear in the bottom part of *Hotel*. There are two such constraints:

1. $\#RM = res$. The number of guests who have been assigned rooms (the size of the RM function) equals the total number of reservations (res).
2. $\mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}\, RM$. The set of rooms that are taken ($\mathrm{dom}(ST \rhd \{Taken\})$) is exactly the set of rooms reserved by guests ($\mathrm{ran}\, RM$).

The three types of transactions in the hotel database are *Reserve*, *Cancel* and *Report*. These constitute the operations in the hotel database. *Reserve* reserves a room for guest $g?$ and produces as output a room assignment $r!$. *Reserve* has a precondition that there must be fewer than *total* reserved rooms and $g?$ must be a new guest. Since the domain of RM is the set of guests with reservations, the latter is captured by checking that $g? \notin \mathrm{dom}\, RM$. *Reserve* has a postcondition that some room $r!$ with status *Available* is chosen, the number of reservations is incremented, the status of $r!$ is changed to *Taken*, and the ordered pair $g? \mapsto r!$ is added to the function RM. *Cancel* cancels a reservation for guest $g?$. *Cancel* has a precondition that $g?$ is a guest and a postcondition that *res* is decremented, $g?$ is removed from the domain of the function RM, and the status of the room for $g?$ is changed to *Available*. *Report* has no precondition, and merely produces the state components ST and RM as outputs.

Since the role of initialization is peripheral to our analysis, we omit initialization schemas here. Instead, we assume that the database has been initialized to a consistent state.

4 THE MODEL

In our model, a *database* is specified as a collection 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. A database state is said to be *consistent* if the values of the objects satisfy the given invariants.

A *transaction* is an operation that transforms one database state to another. Associated with each transaction is a set of *preconditions* and a set of *postconditions*. A precondition limits the database states to which a transaction can be applied. For example, a *Reserve* transaction has a precondition that the hotel have at least one room available. A postcondition constrains the possible database states after a transaction completes. For example, a *Reserve* transaction has a postcondition that there be some room available before the reservation that is taken after the reservation. Postconditions also constrain outputs. For example, the room $r!$ output by *Reserve* must be available initially. Together, preconditions and postconditions must ensure that if a transaction executes on a consistent state, the result is again a consistent state.

Instead of executing a transaction as an atomic unit, we wish to break a transaction into steps and execute each of these steps as an atomic unit. A *decomposition* of a transaction is a sequence of *steps*. In place of the transaction, the steps execute atomically in order. A transaction that has been decomposed into two or more steps is referred to as a *multistep* transaction. In keeping with the terms used in formal methods, we say that a decomposition is a *refinement* of the original specification, with the understanding that the various properties described subsequently are satisfied. We also use the term refinement to describe other steps towards implementation. For example, a successor set description, defined in section 6, is a refinement of a decomposition.

One possible approach to decomposition is to treat the steps as transactions. In particular, one could insist that the integrity constraints hold after each step, which is the decision taken in the Saga model [9]. As the naive decomposition below demonstrates, such a requirement is too strong for some applications, and so we develop a more flexible approach.

4.1 A Naive Decomposition of the Reserve Transaction

Suppose we break up the *Reserve* transaction into the following three atomic steps. A naive specification of these steps is given in figure 2.

Step 1: Increment the number of reserved rooms.

Step 2: Pick a room with status *Available* and change it to *Taken*.

Step 3: Assign the room selected in Step 2 to the guest.

From a formal perspective, the naive decomposition has a fatal flaw, in that none of the proposed steps, considered by itself, maintains the invariants in *Hotel*. For example, $NaiveR1$ does not maintain the invariant $\#RM = res$ since $NaiveR1$ increments the value of res, but does not alter RM. Formally,

NaiveR1
$\Delta(res)$
$res < total$
$res' = res + 1$

NaiveR2
$\Delta(ST)$
$r! : Room$
$ST(r!) = Available$
$ST' = ST \oplus$ $\{r! \mapsto Taken\}$

NaiveR3
$\Delta(RM)$
$r! : Room$
$g? : Guest$
$g? \notin \mathrm{dom}\, RM$
$RM' = RM \cup \{g? \mapsto r!\}$

Figure 2 A Naive Decomposition

the computed preconditions of all three steps simplify to false, indicating that none of the steps can be safely executed in an implementation. Executing any of the proposed steps violates the invariants, and other transactions are exposed to the inconsistent state. For example, *Report* produces an inconsistent output if executed in a state in which the second invariant does not hold.

4.2 Generalizing the Original Invariants

The example demonstrates that some decompositions are unacceptable. Specifically, a decomposition may yield steps that leave the database in a state in which the invariants do not hold. The arrow labeled *NaiveR1* in figure 3(a) illustrates this possibility. Once the invariants are violated, the formal basis for assessing the correctness of subsequent behavior collapses.

Insisting on decompositions where each step maintains database consistency does solve this problem. However, the informal description of the steps into which *Reserve* is broken is perfectly satisfactory, and it is excessive to insist that the invariants of *Hotel* hold at all intermediate steps. We need a formal model that can accommodate the notion that some – but not all – violations of the invariants are acceptable.

Figure 3(b) illustrates a model that allows inconsistent states – as defined by the invariants – that are nonetheless acceptable. The temporary inconsistency introduced by *R1* (specified below in figure 4) is allowed, and steps of some other transactions, for example *CancelD*, can tolerate the inconsistency introduced by *R1*, and so are allowed to proceed. The general approach is to modify the original set of invariants and decompose transactions such that each step satisfies the new set of invariants. The model in figure 3(b) has many advantages, including greater concurrency among steps. We formalize the model as follows.

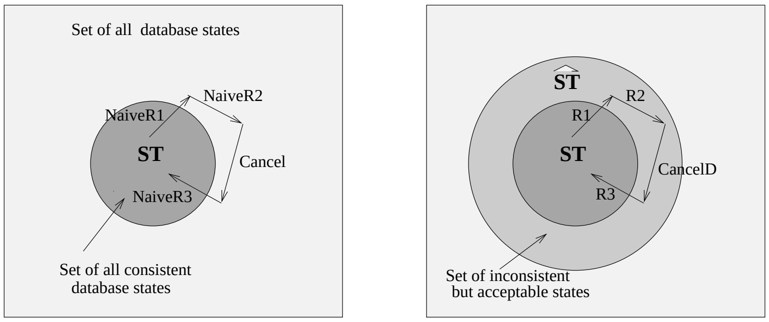
Let I denote the original invariants, and let $\mathbf{ST}$ denote the set consisting of all consistent states; that is, $\mathbf{ST} = \{ST : ST \text{ satisfies } I\}$. In the standard model, a transaction T_i always accesses a consistent $ST \in \mathbf{ST}$. If ST_i denotes the state after the execution of T_i, then ST_i is also in $\mathbf{ST}$. When T_i is broken up into steps $S_{i1}, \ldots, S_{in}$, each step S_{ij} executes atomically. If ST_{ij} represents the state resulting from the partial execution of T_i through step S_{ij}, it is possible that ST_{ij} no longer satisfies the invariants I and so lies outside $\mathbf{ST}$. Figure 3(a) illustrates this possibility for the naive decomposition of the hotel example.

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 state that satisfies $\hat{I}$. Let $\widehat{\mathbf{ST}} = \{ \text{ ST : ST satisfies } \hat{I}\}$. The relationship between $\mathbf{ST}$ and $\widehat{\mathbf{ST}}$ is shown in figure 3(b). The inner circle denotes $\mathbf{ST}$ and the outer circle denotes $\widehat{\mathbf{ST}}$ (signifying that $\mathbf{ST} \subset \widehat{\mathbf{ST}}$). The ring denotes the set of all states that satisfy $\hat{I}$ but not I. The important part about figure 3(b) is that the set of inconsistent but acceptable states is formally identified and distinguished from the states that are unacceptable. The advantage is that formal analysis can be used to investigate activities in $\widehat{\mathbf{ST}}$.

To reason about decomposing transactions into steps and to avoid the problems of a naive decomposition, we use *auxiliary variables* to generalize the invariants. Auxiliary variables are a standard method of reasoning about concurrent executions [12] and, in particular, have been applied to the problem of semantic-based concurrency control [8, Appendix C]. Our work focuses more on the decomposition than does [8], and so we emphasize the role of auxiliary variables more strongly. Before we generalize the invariants in the hotel example, we present two properties that a decomposition must possess. We stress that the auxiliary variables are introduced for purposes of analysis; the goal is to eliminate such variables from an implementation.

4.3 Composition Property

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



(a) Standard classification of database states

(b) Database states as classified in our model

Figure 3 Classification of the Database States

Composition Property Let $S_{i1}, \ldots, S_{in}$ be the steps of transaction T_i and ST be a state that satisfies the original integrity constraints I. Then executing the sequence of steps $S_{i1}, \ldots, S_{in}$ in isolation on ST is equivalent to executing T_i on ST, except for constraints on auxiliary variables.

If aux and aux' denote the auxiliary variables, the composition property in Object Z is:

$$I \wedge (S_{i1} \mathbin{;} \ldots \mathbin{;} S_{in}) \setminus (aux, aux') \Leftrightarrow T_i$$

The left hand side gives the composition of the steps where the initial state is constrained to satisfy I and the auxiliary variables are 'hidden' or suppressed. The right hand side is the original transaction T_i. In Object Z a propositional relation between schemas – equivalence, in this case – translates into the same relation between the predicates defining the schemas.

At this stage we do not state what happens if the precondition of some step is not satisfied and thus the execution cannot complete. We address this point after we introduce the notion of compensating steps.

From an implementation perspective, the composition property is similar to requiring that the sequential execution of the steps be view equivalent to that of the original transaction.

4.4 Sensitive Transaction Isolation Property

In our model, we allow transactions to access database states that do not satisfy the original invariants (that is, 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 called *sensitive* transactions [8]. 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 $\mathbf{ST}$, even though the decomposition of T_i may access database states in $\widehat{\mathbf{ST}} - \mathbf{ST}$.

In our model, we ensure the sensitive transaction isolation property by construction. There are two aspects to such a construction. First, for each sensitive transaction, we compute the subset of the original integrity constraints, I, relevant to the calculation of any outputs. Second, as pointed out by Rastogi, Korth, and Silberschatz [13], if outputs are generated by multiple steps, interleavings between these steps must be controlled to ensure that outputs from later steps are consistent with outputs from earlier steps.

We specify the sensitive transaction isolation property using history invariants in Object Z; we illustrate this later in Section 5.1.

4.5 Compensating Steps

When transactions are decomposed into steps a transaction may not complete successfully. A transaction may not complete successfully if a precondition of a step is not satisfied, the user aborts the transaction, or the system crashes. Incomplete transactions pose special problems in semantic oriented models because steps may commit before it is determined whether the transaction can complete.

Suppose a *Reserve* transaction aborts after its first step, say $R1$, commits. Some mechanism must undo the effects of $R1$. Nullifying the effects of $R1$ using the backwards recovery method of traditional *undo* [4], where the state that existed before $R1$ is physically restored, is not possible because steps of other transactions may have read the updates of $R1$. Instead we adopt the forward recovery solution of a compensating step [8, 9], which semantically undoes the effects of the committed step $R1$ and does not disturb transactions that may have read from $R1$.

Like other steps of a transaction, compensating steps execute atomically. However, the role of a compensating step differs from that of other steps. A compensating step is not considered part of the normal processing of a transaction; it is initiated only to semantically undo a transaction.

For a transaction T_i that has been decomposed into n steps $S_{i1}, \ldots, S_{in}$ we specify $n-1$ compensating steps denoted by $C_{i2}, \ldots, C_{in}$. The compensating step C_{ij} semantically undoes the cumulative effect of steps $S_{i1},\ldots,S_{i(j-1)}$. This is in contrast to the approach used in [8, 9] where a compensating step C_{ij} is used to semantically undo the operations performed by a single step S_{ij}. The difference between the approaches is not significant; our choice simplifies our presentation. The steps and compensating steps for the hotel example is presented later in Section 5.1.

4.6 Semantic Histories

We are interested in the relationship between the original specification and the specification with the generalized invariants. In particular, we would like to know if and when the database returns to a consistent state.

Before we proceed further, we make a distinction between a type of a step and an instance of a step. Suppose the *Reserve* transaction in the hotel example is decomposed into steps $R1$, $R2$ and $R3$; $R1$, $R2$, $R3$ will represent the different types of steps.

Histories, defined subsequently, reflect actual transactions, and must reference instances of steps and compensating steps. A history may contain many instances of a step of a given type. In cases where it not necessary to distinguish the role of steps from that of compensating steps, we use the term 'step' generically and denote an instance of either a step or a compensating step of transaction T_i as T_{ij}. Where the roles differ, we use S_{ij} to denote an instance of a step and C_{ij} to denote an instance of a compensating step. The type of an instance of a step T_{ij} is denoted by $ty(T_{ij})$.

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

1. *a step T_{ij} either appears exactly once in H or does not appear at all,*
2. *for any two steps S_{ij} and T_{ik}, S_{ij} precedes T_{ik} in H if S_{ij} precedes T_{ik} in T_i,*
3. *if $T_{ik} \in H$, then $S_{ij} \in H$ for $j = 1, \ldots, (k-1)$,*
4. *if a compensating step $C_{ij} \in H$, then $S_{ij} \notin H$ and $T_{ik} \notin H$ for $k > j$.*

Condition (1) ensures that every step of a transaction occurs at most once. Condition (2) ensures that the order of the steps in a transaction is preserved. Condition (3) ensures that for every step, preceding steps in the corresponding transaction are present. Condition (4) ensures that a compensating step terminates a transaction.

Unlike typical definitions of histories, our notion of a history does not reference the operations on data elements, such as read and write. Such operations are introduced and integrated into the definition of histories as we further refine our specifications.

Example 1 $< S_{11}, S_{11} >$ *is not a stepwise serial history since it violates condition 1.* $< S_{13}, S_{12} >$ *is not a stepwise serial history since it violates conditions 2 and 3.* $< S_{11}, C_{12}, S_{12} >$ *is not a stepwise serial history since it violates condition 4.* $< S_{11}, S_{21}, S_{12}, S_{13} >$ *and* $< S_{11}, S_{21}, C_{12}, S_{22} >$ *are stepwise serial histories.*

Definition 2 [**Complete Execution**] *Consider a transaction T_i decomposed into steps $S_{i1}, \ldots, S_{in}$ with compensating steps $C_{i2}, \ldots, C_{in}$. The execution of T_i in a history H is a complete execution if either (i) all n steps of T_i appear in H or (ii) some steps of T_i, namely, $S_{i1}, \ldots, S_{ij}$ appear in H followed by the corresponding compensating step $C_{i(j+1)}$, where $j < n$.*

The sequences $S_{i1}, \ldots, S_{in}$ and $S_{i1}, \ldots, S_{ij}, C_{i(j+1)}$ are examples of *execution sequences* [8] of transaction T_i. The sequence $S_{i1}, \ldots, S_{in}$ is a *successful* execution sequence of T_i, and the sequence $S_{i1}, \ldots, S_{ij}, C_{i(j+1)}$ is an *unsuccessful* execution sequence of T_i.

Example 2 *For the hotel database, an execution of a Reserve transaction T_i is complete in H if either (i) all three steps S_{i1}, S_{i2}, and S_{i3} of T_i appear in H, or (ii) S_{i1} and C_{i2} appear in H, or (iii) S_{i1}, S_{i2} and C_{i3} appear in H. Case(i) is an example of successful complete execution. Cases(ii) and (iii) are examples of unsuccessful complete executions.*

To introduce state information, we define *semantic history*.

Definition 3 [**Semantic History**] *A semantic history H is a stepwise serial history bound to*

1. *an initial state, and*
2. *the states resulting from the execution of each step in H.*

Informally, we use the term partial semantic history for cases in which the execution of at least one transaction actually is incomplete, but from a formal perspective, partial semantic histories are just semantic histories. Complete semantic histories are a special case of a semantic histories:

Definition 4 [**Complete Semantic History**] *A semantic history H over a set of transactions $\mathbf{T}$ is a complete semantic history if the execution of each T_i in $\mathbf{T}$ is complete.*

Next we define what it means for a semantic history to be correct.

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

1. *the initial state is in* $\mathbf{ST}$,

2. *the states before and after each step in H are in* $\widehat{\mathbf{ST}}$, *and*
3. *the precondition for each step is satisfied in the corresponding state.*

Definition 6 [**Correct Complete Semantic History**] *A complete semantic history H is a correct complete semantic history if*

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

4.7 Consistent Execution Property

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.

Like the sensitive transaction isolation property, we specify the consistent execution property using history invariants. We show this later in Section 5.1.

4.8 Semantic Atomicity Property

When transactions have been broken up into steps, it may not be possible to complete a transaction. This may happen if the precondition of some later step is not satisfied and the effects of the partially executed transactions cannot be undone by executing compensating steps. The semantic atomicity property ensures that such a situation is avoided; if a transaction has been partially executed, then it can complete.

Semantic Atomicity Property Every correct semantic history H_p defined over a set of transactions $\mathbf{T}$ is a prefix of some correct complete semantic history H over $\mathbf{T}$.

Like all the other properties stated so far, semantic atomicity is a necessary property. The definition of semantic atomicity property is very general. Some applications may require a stronger property, the successful execution property, stated below.

4.9 Successful Execution Property

When transactions have been broken up into steps, it may not be possible to complete a transaction successfully. This may happen if a precondition of some later step of the transaction is not satisfied. Also, the interleaving of steps of different transactions may result in a state from which it is not possible to successfully complete some transaction. The successful execution property ensures that such a situation is avoided; if a transaction has been partially executed, then it can complete without resorting to compensation.

Successful Execution Property Every correct semantic history H_p defined over a set of transactions $\mathbf{T}$ is a prefix of some correct complete semantic history H over $\mathbf{T}$ such that for each $T_i \in \mathbf{T}$ that is incomplete in H_p, H contains a successful execution sequence of T_i.

Unlike the other properties we have stated so far, successful execution is an optional property. Successful execution property requires that all the preconditions of a transaction should appear in the first step. This in turn would require a large number of updates to be performed in the first step. Thus insisting on successful execution property may force too many operations in the first step of the transaction - which is undesirable from the performance point of view. Hence we do not insist that applications have the successful execution property.

5 EXAMPLES OF DECOMPOSITION

5.1 A Valid Decomposition

In this section, we provide a decomposition of the hotel database. The example is valid in the sense that it satisfies all of the properties that we require for a decomposition.

To make the invariants more general, we add auxiliary variables and define a new state *HotelD*, which is short for *Hotel Decomposition*. We add the auxiliary variable *underway*, which is a natural number, to denote the reservations that have been partially processed. We also add the auxiliary variable *acquired*, which is a set of rooms, to denote the rooms that have been taken but which have not yet been assigned to guests. The invariants are modified accordingly. The schema *HotelD* together with the generalized invariants is shown in figure 4.

$R1$, $R2$ and $R3$ are three steps of the *Reserve* transaction. To implement a *Reserve*, the three steps execute in order. The composition property for the hotel example, formally stated in Object Z, is as follows.

$$Hotel \land ((R1 \mathbin{⨾} R2 \mathbin{⨾} R3) \setminus (underway, acquired, underway', acquired')) \Leftrightarrow Reserve$$

The three steps satisfy the composition property; we omit the details of the proof and refer the interested reader to [3]. Although *Reserve* is a sensitive transaction, it turns out that no additional preconditions are needed to ensure that the output $r!$ reflects a consistent state; again, see [3] for details.

The specification of the single step *CancelD* is identical to the transaction *Cancel*.

Report is a sensitive transaction, and we establish the sensitive transaction isolation property by construction. A formal treatment is given in [3]. Informally, *Report* transaction outputs values of ST and RM. ST and RM appear in the following original invariant:

$$\mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}\, RM$$

which can be derived from the generalized invariant

$$\mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}\, RM \cup acquired$$

if the auxiliary variable *acquired* satisfies $acquired = \varnothing$. Hence, to ensure that *ReportD* does not output inconsistent data we specify the following restriction as a history invariant.

$$\Box((acquired \neq \varnothing) \Rightarrow (\bigcirc \mathbf{op} \neq ReportD)).$$

The above notation means that it is always true when the auxiliary variable *acquired* is not the empty set, the next operation must not be the step *ReportD*.

HotelD

$$
\begin{array}{l}
res, underway : \mathbb{N} \\
ST : Room \nrightarrow Status \\
RM : Guest \rightarrowtail Room \\
acquired : \mathbb{P}\, Room \\
\hline
\#RM + underway = res \\
\mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}\ RM \cup acquired
\end{array}
$$

R1

$$
\begin{array}{l}
\Delta(res, underway) \\
\hline
res < total \\
res' = res + 1 \\
underway' = underway + 1
\end{array}
$$

R2

$$
\begin{array}{l}
\Delta(ST, acquired);\ r! : Room \\
\hline
ST(r!) = Available \\
ST' = ST \oplus \{r! \mapsto Taken\} \\
acquired' = acquired \cup \{r!\}
\end{array}
$$

R3

$$
\begin{array}{l}
\Delta(RM, acquired, underway) \\
g? : Guest;\ r! : Room \\
\hline
g? \notin \mathrm{dom}\ RM \\
RM' = RM \cup \{g? \mapsto r!\} \\
acquired' = acquired \setminus \{r!\} \\
underway' = underway - 1
\end{array}
$$

CancelD

$$
\begin{array}{l}
\Delta(ST, RM, res) \\
g? : Guest \\
\hline
g? \in \mathrm{dom}\ RM \\
res' = res - 1 \\
ST' = ST \oplus \{RM(g?) \mapsto Available\} \\
RM' = \{g?\} \ntriangleleft RM
\end{array}
$$

ReportD

$$
\begin{array}{l}
currentST! : Room \nrightarrow Status \\
currentRM! : Guest \rightarrowtail Room \\
\hline
currentST! = ST \\
currentRM! = RM
\end{array}
$$

CompR3

$$
\begin{array}{l}
\Delta(res, underway, acquired, ST) \\
r! : Room \\
\hline
res > 0 \\
underway > 0 \\
r! \in acquired \\
res' = res - 1 \\
underway' = underway - 1 \\
acquired' = acquired \setminus \{r!\} \\
ST' = ST \oplus \{r! \mapsto Available\}
\end{array}
$$

CompR2

$$
\begin{array}{l}
\Delta(res, underway) \\
\hline
res > 0;\ underway > 0 \\
res' = res - 1 \\
underway' = underway - 1
\end{array}
$$

$$
\Box(acquired \neq \varnothing) \Rightarrow \bigcirc(\mathbf{op} \neq ReportD)
$$

$$
\Box\Diamond(acquired = \varnothing \wedge underway = 0)
$$

Figure 4 A Valid Decomposition for the Hotel Database

We specify the consistent execution property for the hotel example using the following history invariant.

$\Box(\Diamond(underway = 0 \wedge acquired = \varnothing))$

This notation means it is always true that eventually the auxiliary variables satisfy the condition $underway = 0 \wedge acquired = \varnothing$. Note that when $underway = 0 \wedge acquired = \varnothing$, the database is in a consistent state.

5.2 An Invalid Decomposition

In this section, we give an example of an invalid decomposition. Unlike the naive decomposition, the decomposition given below generates correct semantic histories. The decomposition is invalid because it does not satisfy one of the necessary properties, namely, the semantic atomicity property.

To illustrate the possibility, we modify the *Hotel* database as shown in fig. 5.

In the example specification, the cancel transaction is decomposed into steps C1 and C2. We introduce the auxiliary variable *underwayC* which keeps count of the cancel transactions that have completed step $C1$ but not step $C2$. The invariant $\#RM = res - underwayR$ in the original *HotelD* is changed to $\#RM = res - underwayR + underwayC$.

Moreover, we introduce a new structure *clist* which keeps track of the guests whose cancelations are in progress. The guest whose reservation is being canceled is added to the *clist* in step $C1$ and is removed from the *clist* in step $C2$. To ensure that a guest whose cancelation is in progress is not canceled by some other transaction, we include precondition $g? \notin clist$ in step $C1$. $C1$ also has another precondition $res > 0$ which ensures that $C1$ executes when there is at least one reservation. The postcondition of $C1$ decrements *res*, increments *underwayC* and inserts $g?$ in *clist*. The preconditions of $C2$ check that $g?$ has a valid reservation, $g?$ is in *clist* and *underwayC* is positive. The postcondition of $C2$ removes $g?$ from the domain of RM, makes the room, which was assigned to $g?$, *Available*, and removes $g?$ from *clist*.

DeadlockHotel

$$
\begin{array}{l}
res, underwayR, underwayC : \mathbb{N} \\
RM : Guest \rightarrowtail\!\!\!\!\!\rightarrow Room;\ ST : Room \nrightarrow Status \\
clist : \mathbb{P}\, Guest;\ acquired : \mathbb{P}\, Room \\
\hline
\# RM = res - underwayR + underwayC \\
\mathrm{dom}(ST \rhd \{Taken\}) = \mathrm{ran}\, RM \cup acquired
\end{array}
$$

Res1

$$
\begin{array}{l}
\Delta(res, underwayR) \\
\hline
res < total \\
res' = res + 1 \\
underwayR' = underwayR + 1
\end{array}
$$

Res2

$$
\begin{array}{l}
\Delta(ST, acquired);\ r! : Room \\
\hline
ST(r!) = Available \\
ST' = ST \oplus \{r! \mapsto Taken\} \\
acquired' = acquired \cup \{r!\}
\end{array}
$$

Res3

$$
\begin{array}{l}
\Delta(RM, underwayR, acquired) \\
g? : Guest;\ r! : Room \\
\hline
underwayR > 0;\ r! \notin \mathrm{ran}\, RM \\
g? \notin \mathrm{dom}\, RM;\ g? \notin clist \\
RM' = RM \cup \{g? \mapsto r!\} \\
underwayR' = underwayR - 1 \\
acquired' = acquired \setminus \{r!\}
\end{array}
$$

C1

$$
\begin{array}{l}
\Delta(res, clist, underwayC) \\
g? : Guest \\
\hline
res > 0 \\
g? \notin clist \\
res' = res - 1 \\
clist' = clist \cup \{g?\} \\
underwayC' = underwayC + 1
\end{array}
$$

C2

$$
\begin{array}{l}
\Delta(ST, RM, clist);\ g? : Guest \\
\hline
g? \in \mathrm{dom}\, RM;\ g? \in clist \\
underwayC > 0 \\
ST' = ST \oplus \{RM(g?) \mapsto Available\} \\
RM' = \{g?\} \lhd RM;\ clist' = clist \setminus \{g?\}
\end{array}
$$

CompRes2

$$
\begin{array}{l}
\Delta(res, underwayR) \\
\hline
res > 0 \\
underwayR > 0 \\
res' = res - 1 \\
underwayR' = underwayR - 1
\end{array}
$$

CompRes3

$$
\begin{array}{l}
\Delta(res, underwayR, acquired, ST) \\
r! : Room \\
\hline
res > 0;\ underwayR > 0 \\
r! \in acquired;\ ST(r!) = Taken \\
res' = res - 1 \\
underwayR' = underwayR - 1 \\
acquired' = acquired \setminus \{r!\} \\
ST' = ST \oplus \{r! \mapsto Available\}
\end{array}
$$

CompC2

$$
\begin{array}{l}
\Delta(res, clist, underwayC) \\
g? : Guest \\
\hline
res < total \\
g? \in clist;\ g? \in \mathrm{dom}\, RM \\
underwayC > 0 \\
res' = res + 1 \\
clist' = clist \setminus \{g?\} \\
underwayC' = underwayC - 1
\end{array}
$$

$$\Box\Diamond(underwayR = 0 \wedge underwayC = 0 \wedge acquired = \varnothing)$$

Figure 5 Example Specification Lacking Complete Execution Property

Since the cancel transaction is decomposed into two steps, we must specify $CompC2$, a type of compensating step, which semantically undoes the actions of $C1$. $CompC2$ has four preconditions: res must be less than $total$, $underwayC$ must be positive, $g?$ must be in $clist$ and $g?$ must have a valid reservation. The postcondition of $CompC2$ increments res, decrements $underwayC$ and removes $g?$ from $clist$.

We impose an additional constraint that a room cannot be reserved for a guest whose cancelation is in progress; note the precondition $g? \notin clist$ in step $Res3$. The reserve transaction is broken into steps $Res1$, $Res2$ and $Res3$, similar to $R1$, $R2$ and $R3$ of the *HotelD* specification. We assume that this example has no *Report* transaction.

Consider the partial history $H_p =< S_{11} >$ where $ty(S_{11}) = C1$. Suppose in the initial state of H_p, $John \notin \operatorname{dom} RM$. The cancel transaction T_1 attempts to cancel the reservation for guest $John$. The execution of step S_{11} results in $John$ being inserted in the $clist$. Now step S_{12} cannot execute since the precondition $John \notin \operatorname{dom} RM$ is not satisfied as $John$ does not have any reservation. The compensating step C_{12} of type $CompC2$ also checks for these preconditions; since these preconditions are not satisfied, the compensating step cannot be executed. The specification is therefore an invalid specification because it lacks the semantic atomicity property.

The deadlock could be avoided by including the invariant $clist \subseteq guest$ in *DeadlockHotel*. Omission of this constraint allows the database to enter an undesirable state where $c? \in clist \wedge c? \notin guest$, from which neither the next step or the compensating step could be executed.

6 SUCCESSOR SETS

The decomposition process introduces additional database objects (auxiliary variables) and imposes additional constraints (history invariants) on the execution of steps. The additional objects are present primarily to support analysis. For efficient implementation, we want to avoid instantiating the objects. The specifications also require the satisfaction of history invariants; checking the satisfaction of history invariants is expensive and our goal is to avoid such checks in the implementation. Successor sets are the mechanism we use to achieve this objective.

Definition 7 [**Successor Set**] *The successor set of $ty(T_{ij})$, denoted $SS(ty(T_{ij}))$, is a set of types of steps.*

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; the correctness characterization is specified in terms of semantic histories.

But before giving the correctness characterization, we wish to define histories in terms of conflicts. To achieve this goal we introduce the notion of conflict into our model. Two operations *conflict* if both operate on the same data item and at least one is a Write. Two steps T_{ij} and T_{pq} *conflict* if they contain conflicting operations.

It is easy to determine the set of conflicting steps once the code for the decomposed transactions is given. At this stage we only have the specification, but we would still like to define a notion of conflict. We define the following variables as being *written* in the specification:

1. Any state variable modified in a postcondition.

Similarly, we define the following variables as being *read* in the specification:

1. Any state variable referenced in a precondition, or
2. Any state variable referenced in a postcondition.

The read and write set of the steps of the Hotel Database, as obtained from the specifications is given in Table 2. Table 3 gives the set of conflicting steps in the Hotel Database.

Type of Step	Variables Read	Variables Written
$R1$	res, $total$, $underway$	res, $underway$
$R2$	ST, $acquired$	ST, $acquired$
$R3$	RM, $underway$, $acquired$	RM, $underway$, $acquired$
$ReportD$	ST, RM	
$CancelD$	res, ST, RM	res, ST, RM
$CompR2$	res, $underway$	res, $underway$
$CompR3$	res, ST, $underway$, $acquired$	res, ST, $underway$, $acquired$

Table 2 Read and Write Sets for Steps of Hotel Example

Type of Step	Types of Conflicting Steps
$R1$	$R1$, $R3$, $CancelD$, $CompR2$, $CompR3$
$R2$	$R2$, $R3$, $ReportD$, $CancelD$, $CompR3$
$R3$	$R1$, $R2$, $R3$, $ReportD$, $CancelD$, $CompR2$, $CompR3$
$ReportD$	$R2$, $R3$, $CancelD$, $CompR3$
$CancelD$	$R1$, $R2$, $R3$, $ReportD$, $CancelD$, $CompR2$, $CompR3$
$CompR2$	$R1$, $R3$, $CancelD$, $CompR2$, $CompR3$
$CompR3$	$R1$, $R2$, $R3$, $CancelD$, $ReportD$, $CompR2$, $CompR3$

Table 3 Conflicting Steps for Hotel Example

Armed with a notion of conflict that is applicable for both specifications and implementations, we define a history by using conflict in a manner similar to [4].

Definition 8 **[History]** *A history H defined over a set of transactions $\mathbf{T} = \{T_1, \ldots, T_m\}$, where each transaction T_i is composed of n_i steps, is a partial order of operations with ordering relation $\prec_H$ where:*

1. $H = \cup_{i=1}^{m} \cup_{j=1}^{n_i} T_{ij}$;
2. $\prec_H \supseteq \cup_{i=1}^{m} \cup_{j=1}^{n_i} \prec_{ij}$; *and*

3. *for any two conflicting operations* $p, q \in H$, *either* $p \prec_H q$ *or* $q \prec_H p$.

Condition (1) says that the execution represented by H involves precisely the operations of the steps of $\mathbf{T}$. Condition (2) says that H preserves the order of operations in each step. Condition (3) says that every pair of conflicting operations are ordered in H.

The definition of conflict allows us to define a notion of correctness with respect to successor set descriptions that is not overly restrictive. The possibility of being overly permissive is addressed subsequently.

Definition 9 [**Correct Successor Set History**] *Let H be a correct semantic history. 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 $ty(T_{pq}) \in SS(ty(T_{ij}))$.

Example 3 *In the hotel example, there are two history invariants corresponding to the consistent execution property and sensitive transaction isolation property. The specification of the steps in the hotel example is such that the database always returns to a consistent state when all transactions complete and no extra mechanism is required to enforce the consistent execution property.*

The other history invariant forbids the execution of steps of type ReportD when the auxiliary variable acquired $\neq \varnothing$. *This history invariant is satisfied as long as a step of type ReportD does not appear between steps of type* $R2$ *and* $R3$ *of reserve transaction. To ensure this we specify the successor sets as shown in Table 4.*

For the hotel example, all of the history invariants in auxiliary variables are captured by the successor set description, and so none of the the auxiliary variables need to be implemented.

With respect to the specifications given with history invariants and auxiliary variables, not all successor set descriptions are valid. Informally, a successor

Successor Set of Type of Step	Types of Steps in Successor Set
$SS(R1)$	$R1$, $R2$, $R3$, $ReportD$, $CancelD$, $CompR2$, $CompR3$
$SS(R2)$	$R1$, $R2$, $R3$, $CancelD$, $CompR2$, $CompR3$
$SS(R3)$	$R1$, $R2$, $R3$, $ReportD$, $CancelD$, $CompR2$, $CompR3$
$SS(ReportD)$	$R1$, $R2$, $R3$, $ReportD$, $CancelD$, $CompR2$, $CompR3$
$SS(CancelD)$	$R1$, $R2$, $R3$, $ReportD$, $CancelD$, $CompR2$,$CompR3$
$SS(CompR2)$	$R1$, $R2$, $R3$, $ReportD$, $CancelD$, $CompR2$,$CompR3$
$SS(CompR3)$	$R1$, $R2$, $R3$, $ReportD$, $CancelD$, $CompR2$,$CompR3$

Table 4 Successor Sets for the Hotel Example

set is valid with respect to a specification containing history invariants if any correct successor set history can also be generated by the specification containing history invariants. Although desirable, the converse property does not hold in general since first-order logic history invariants have more expressive power than the successor set mechanism. Formally, we describe valid successor set descriptions with the *valid successor set property*:

Definition 10 [Valid Successor Set Property] *A specification S_2 that employs a successor set description is valid with respect to specification S_1 with history invariants if*

1. *any correct successor set history generated by S_2 is also a correct semantic history generated by S_1.*
2. *S_2 satisfies the semantic atomicity property.*

The second condition can be easily satisfied by ensuring that all compensating steps are contained in each successor set description. The hotel example has the valid successor set property, where it turns out that the successor set specification generates exactly the same set of histories as the specification with history invariants.

Suppose an application requires the successful execution property. Since successor set descriptions are less expressive than the first order predicates they replace, the set of histories for S_2 may be a proper subset of the set of histories for S_1. Therefore, the successful execution property must be reverified explicitly on histories generated by S_2.

7 CORRECT STEPWISE SERIALIZABLE HISTORIES

For every pair of steps in a correct successor set history, all operations of one step appear before any operations of the other step. However if the steps of a transaction execute atomically and without any interleaving, the database system uses resources poorly. In this section we describe the notion of a *correct stepwise serializable history.* In a correct stepwise serializable history the steps of transactions need not be executed serially, but nevertheless the effect is the same as that of a correct successor set history.

We develop stepwise serializability by defining equivalence in a standard way [4]. Two histories H and H' are said to be *equivalent* if they are defined over the same set of steps, they have the same operations, and they order conflicting operations of steps in the same way. That is, for any conflicting operations p_{ij} and q_{kl} belonging to T_{ij} and T_{kl} (respectively), if $p_{ij} \prec_H q_{kl}$, then $p_{ij} \prec_{H'} q_{kl}$. A *correct stepwise serializable history* is one which is equivalent to a correct successor set history. Before we build a graph model to reason about stepwise serializability of histories we introduce a helpful function.

7.1 The Next Function

Consider the following scenario. Suppose that in a partial correct successor set 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. The role of the next function in our model is similar to that of the F function in [7].

Definition 11 [**Next Function**] *The next function, denoted by $NF(T_{ij}, T_{pq})$, gives the first step T_{ik} after T_{ij} in an execution sequence of T_i such that $ty(T_{pq}) \in SS(ty(T_{ik}))$. Formally, $T_{ik} = NF(T_{ij}, T_{pq})$ if the following conditions hold:*

1. *$ty(T_{pq}) \in SS(ty(T_{ik}))$, $k \geq j$ and*
2. *for each step T_{ir} appearing between T_{ij} and T_{ik} in T_i (if any), $ty(T_{pq}) \notin SS(ty(T_{ir}))$.*

A transaction T_i may have multiple execution sequences, one successful, and others unsuccessful. The value returned by the next function $NF(S_{ij}, T_{pq})$ depends on the particular sequence of T_i. For example, for the successful execution sequence, $S_{i1}, \ldots, S_{ij}, \ldots, S_{ik}, \ldots, S_{in}$, $NF(S_{ij}, T_{pq}) = S_{ik}$. For the unsuccessful execution sequence, $S_{i1}, \ldots, S_{ij}, \ldots, C_{im}$, $NF(S_{ij}, T_{pq}) = C_{im}$ for $m = j+1, \ldots, k$. In our subsequent analysis, we consider histories where the execution sequence for each transaction is fixed, and so this nondeterministic nature of the next function is not an issue in our presentation.

For the purposes of our graph model, we constrain the next function with respect to conflict. Suppose step T_{pq} conflicts with T_{ij} or some earlier step of T_i and $ty(T_{pq}) \notin SS(ty(T_{ij}))$. Let $NF(T_{ij}, T_{pq}) = T_{ik}$. T_{pq} is not allowed to execute after steps T_{ij}, $T_{i(j+1)}$, ..., $T_{i(k-1)}$. But it is allowed to execute after step T_{ik}. This scenario is shown below.

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

T_{ij} introduces some inconsistency that prevents T_{pq} from being executed. This inconsistency is resolved in step T_{ik}. If T_{ik} alters the database state in such a way that T_{pq} can execute, then T_{pq} and T_{ik} must conflict. We capture this property as an explicit constraint in our model. The specifier of a decomposition is obligated to verify the constraint against a successor set description. The constraint lets us show a stronger relationship between acyclic precedence graphs and correct stepwise serializable histories than was possible in [7].

Constraint 1 *If T_{pq} conflicts with T_{ij} or some step in T_i previous to T_{ij} and $ty(T_{pq}) \notin SS(ty(T_{ij}))$, then T_{pq} also conflicts with $NF(T_{ij}, T_{pq})$.*

7.2 Acyclic Graph Model

In this section 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. Our precedence graph is related to the one developed in [7].

Definition 12 [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*

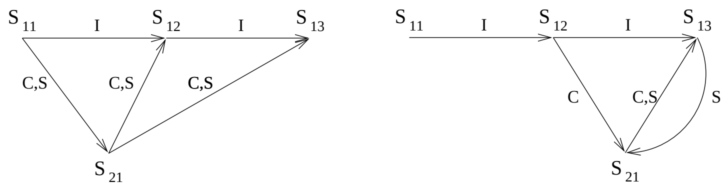


Figure 6 Precedence Graph for Histories in Examples 4 and 5

$PG(H)$, is a directed graph where the nodes are the steps of the transactions in **T** *and the edges are of three types constructed as follows:*

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

Observation 1 *For a correct successor set history H, if there is an edge (T_{ij}, T_{pq}) in $PG(H)$ then T_{ij} precedes T_{pq}. It follows that for a correct successor set history H, the precedence graph $PG(H)$ is acyclic.*

Example 4 *Consider the following history H:*

$$H = < S_{11}, S_{21}, S_{12}, S_{13} >$$

where $ty(S_{11}) = R1$, $ty(S_{12}) = R2$, $ty(S_{13}) = R3$, and $ty(S_{21}) = CancelD$. Successor set descriptions are given in Table 4. Conflicting steps in the Hotel Database are given in Table 3. The precedence graph is shown in the left of figure 6. The labels I,C,S on the edges of the graph shown in the figure denote

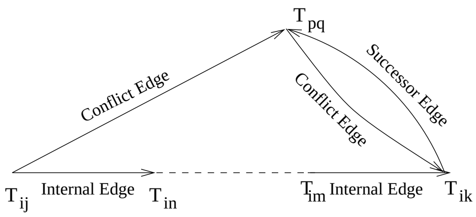


Figure 7 Edges of Precedence Graph involving T_{ij}, T_{ik}, T_{pq}

internal, conflict and successor edge respectively. Note that the precedence graph is acyclic.

Example 5 *Consider the following history H:*

$$H = < S_{11}, S_{12}, S_{21}, S_{13} >$$

where $ty(S_{11}) = R1$, $ty(S_{12}) = R2$, $ty(S_{13}) = R3$, and $ty(S_{21}) = ReportD$. The precedence graph is shown in the right of figure 6. Note that the precedence graph is cyclic.

Theorem 1 *A history H is a correct stepwise serializable history iff $PG(H)$ is acyclic.*

Proof 1 $\Leftarrow$*: 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} and in $PG(H)$ there must be a successor edge (T_{ik}, T_{pq}) corresponding to the conflict edge (T_{ij}, T_{pq}). By Constraint 1 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 conflict edge (T_{pq}, T_{ik}) is in H. As figure 7 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 a correct stepwise serializable history.*

⇒: Since H is a correct stepwise serializable history, it must be equivalent to some correct successor set history H'. H and H' have identical precedence graphs: $PG(H) = PG(H')$. From Observation 1, $PG(H')$ is acyclic. Hence $PG(H)$ is acyclic.

8 CONCURRENCY CONTROL MECHANISMS

In this section, we propose a concurrency control mechanism for our model and identify the issues relevant to an implementation.

We make 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.

Our mechanism uses two phase locking on the steps of the transactions. There are two modes in which a data item may be locked by a step - shared mode or exclusive mode. A step acquires an appropriate lock as a prerequisite for accessing a data item. A step is denied a lock if either another step holds a conflicting lock or if the step fails a test based on successor sets. Locks acquired by a step are released when the step commits or aborts.

For the purposes of this section, we define a step as a sequence of read and write operations followed by a commit or an abort operation,

$$T_{ij} = O_{ij}(x_1), O_{ij}(x_2), \ldots, O_{ij}(x_n), E_{ij},$$

where $O_{ij}(x)$ is either $R_{ij}(x)$ or $W_{ij}(x)$ and E_{ij} is either $C_{ij}(x)$ or $A_{ij}(x)$, and a transaction is a sequence of steps followed by a termination operation,

$$T_i =< T_{i1}, \ldots, T_{in}, TR(T_i) >$$

We require the following data structures in addition to those required by the two phase locking protocol.

1. Active-Set – Set of Active Transactions

   Active-Set(x) – The active set for x keeps the list of all active transactions whose committed steps 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 Active-Set(x).

2. Int-Set – Interleaving Sets

   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.

8.1 Algorithms

Before a read operation can proceed, step T_{ij} needs a shared lock for x. There are two conditions for T_{ij} to acquire the shared lock: (i) No other step has an exclusive lock on x and (ii) T_{ij} is in Int-Set(T_k,x) for all active transactions T_k whose committed steps have accessed x. If either condition is not satisfied, the lock is not granted and step T_{ij} must try again later. When R_{ij}(x) is retried, it must be re-executed from the first step of the algorithm.

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 ∈ Active-set(x)
        if ty(T_ij) ∉ Int-set(T_k,x)
            exit;            /* Lock unavailable - T_ij can retry later */
    lock x in shared mode;
    accept(R_ij(x));
end
```

Before a write operation can proceed, step T_{ij} needs an exclusive lock for x. There are two conditions for T_{ij} to acquire the exclusive lock: (i) No other step has any lock on x and (ii) T_{ij} is in Int-Set(T_k,x) for all active transactions T_k whose committed steps have accessed x. If either condition is not satisfied, the lock is not granted and step T_{ij} must try again later. When W$_{ij}$(x) is retried, it must be re-executed from the first step of the algorithm.

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 ∈ Active-set(x)
        if ty(T_ij) ∉ Int-set(T_k,x)
            exit;          /* Lock unavailable - T_ij can retry later */
    lock x in exclusive mode
    accept(W_ij(x));
end
```

A step commits when all the operations of a step are complete. For each data item x locked by the transaction in the current or previous steps, the interleaving set associated with this transaction and data item x is replaced by the successor set of the step. The transaction is included in the list of active transactions that have accessed x. All locks acquired by this step are released.

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 ∉ Active-set(x)
                Active-set(x) = Active-set(x) ∪ T_i;
            Release the lock on x which was acquired by T_ij;
        end
end
```

A step may not always complete successfully; it may have to abort. The abort causes all the locks held by the step to be released. The abort of step T_{ij} does not affect the data structures Active-set(x) or Int-set(T_i,x); these data structures are adjusted with the transaction termination processing. Traditional recovery for aborted transactions, such as undo, is required for the aborted step, but details are omitted.

Algorithm for Step Abort

> **Procedure** Process-stepabort(A_{ij})
> **begin**
>   /* Restore values written by T_{ij} */
>   **for** each entity x locked by the transaction in this step **do**
>   Release the lock on x which was acquired by T_{ij};
> **end**

Termination removes a transaction from the set of active transactions. Since interleaving sets are associated only with active transactions, the interleaving set Int(T_i,x) is deleted when the transaction terminates.

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**

The Mechanism and Constraint 1

In section 7 we imposed Constraint 1 on successor set descriptions. Constraint 1 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 1 on successor set descriptions. Suppose step T_{pq} and T_{ij} conflict, and the conflicting operations use the data item x. In our mechanism after step T_{ij} commits, Int-Set(T_i,x) is updated. Each subsequent step of T_i updates Int-Set(T_i,x) on commit, whether the step accesses x or not. In particular step $NF(T_{ij}, T_{pq})$ updates Int-Set(T_i,x) on commit. Thus the mechanism behaves as if T_{pq} conflicts with T_{ij} and all subsequent steps, and in particular, the mechanism behaves as if T_{pq} conflicts with step $NF(T_{ij}, T_{pq})$. Hence the mechanism implements Constraint 1 independent of whether the successor set description enjoys the property.

8.2 Correctness of the Mechanism

Theorem 2 *Any history generated by our mechanism is a correct stepwise serializable history.*

Proof 2 *To prove that the history H generated by our mechanism is a correct stepwise serializable history we show that the precedence graph $PG(H)$ is acyclic and invoke Theorem 1. Constraint 1 is addressed by our mechanism as discussed above.*

The assumptions of atomic lock release and a centralized lock manager ensure that lock release operations of steps form a total order, which we denote $\prec_{lr}$. Our mechanism submits step $T_{r(s+1)}$ only after step T_{rs} commits. Thus all I edges in $PG(H)$ are consistent with $\prec_{lr}$. Since two phase locking controls concurrency between steps, all C edges in $PG(H)$ are also consistent with $\prec_{lr}$.

It remains to show that all S edges in $PG(H)$ are consistent with $\prec_{lr}$. Let $NF(T_{ij}, T_{pq}) = T_{ik}$ and consider the S edge (T_{ik}, T_{pq}). The reason for this S edge is that there is a C edge (T_{ij}, T_{pq}) and $ty(T_{pq}) \notin SS(ty(T_{ij}))$. The existence of the C edge provides two relevant facts: First T_{pq} and T_{ij} conflict on some x. Second, T_{ij} performs a Step Commit operation that releases the lock on x before T_{pq} can acquire the conflicting lock. Note that in the Step Commit operation, Int-set(T_i,x) is set to the $SS(ty(T_{ij}))$. Any request by T_{pq} for a conflicting lock on x through the Process-read or Process-write operations is rejected until step after T_{ik} commits and updates Int-set(T_i,x). Thus all S edges in $PG(H)$ are consistent with $\prec_{lr}$.

Since all I, C, and S edges in $PG(G)$ are consistent with the total order $\prec_{lr}$, $PG(H)$ is acyclic.

8.3 Discussion

As with other locking protocols, our mechanism has potential for starvation of transactions and deadlock. Since these issues can be addressed in standard ways, we do not describe detailed algorithms, but instead present a general outline.

In the Process-read and Process-write algorithms we do not specify how to grant locks when multiple steps contend for the same lock. If the scheme for granting lock requests is unfair, starvation may occur. Starvation can be addressed with the standard techniques [6]. 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 and locks are assigned on the basis of priorities.

Deadlock occurs in standard two phase locking [4] – two or more transactions may be blocked, each waiting for the other to complete. In standard two phase locking, transactions block only because a transaction wants to lock an item which is already locked by a different transaction in a conflicting mode. In our mechanism steps may block for the same reasons transactions do in standard two phase locking. In addition, a step may block because it is not in the successor set of some other committed step. One approach is to allow deadlock to occur and periodically check for the presence of deadlock using deadlock detection techniques, such as wait-for graphs [4]. The usual definition of a wait-for graphs must be modified to take into account deadlocks caused by successor set constraints.

There are a variety of issue for supporting compensation. One issue is reliably storing data items which may be needed by a compensating step in case the multistep transaction does not complete. A second issue deals with triggering of the compensating steps. In our model, as in [8] the initiation of the compensating step is done by the system. Such an approach has the advantage that all transaction aborts, whether user-initiated or failure-related, can be treated in a uniform way. A third issue is recovery from system crash. Transactions that are incomplete at the time of the crash can either be compensated or continued.

9 CONCLUSION

In this work, we have provided the database application developer writing the specification conceptual tools necessary to reason about systems in which 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. The developer begins with a specification produced via standard formal methods, transforms some transactions in the specification into steps, and assesses the properties of the resulting system. The formal analysis at each step of this process provides assurance that the resulting system possesses the desired properties.

We have provided a two-phase locking algorithm that ensures execution histories that are stepwise conflict-serializable and also respect the successor set constraints. The algorithm is substantially simpler than the corresponding one in [7]. We have also identified some of the issues which must be resolved before an actual implementation can be developed.

We can easily permit ad hoc transactions to be dynamically added in our model, although they will require some special intervention. An ad hoc transaction could be executed as an atomic, sensitive transaction if all integrity constraints relevant to the transaction were included as explicit preconditions. Alternatively, an ad hoc transaction could be included at the successor set stage by simply excluding it from all successor set descriptions. Deletion of a transaction in our model is not problematic since deletion of a transaction does not impact any other property.

An important question is how well our model scales up to real-world applications. The necessary properties must be demonstrated for applications which must be implemented by our model. In this work, we have used the Object Z specification language and all the analysis are done by hand. However for real life applications this may not be feasible; for such applications it will be necessary to automate to the extent possible the process of discharging the proof obligations. Industrial-level tool support for such an endeavor is essential, and the use of the B Method [1] and other specification languages which have adequate tool support needs to be investigated.

REFERENCES

[1] J. R. Abrial. *B-Technology Technical Overview.* B-Core(UK) Ltd., The Oxford Science Park, Oxford, UK, 1993.

[2] 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.

[3] P. Ammann, S. Jajodia, and I. Ray. Applying formal methods to semantic-based decomposition of transactions. To appear in ACM Transactions on Database Systems.

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

[5] D. Duke and R. Duke. Towards a semantics for Object Z. In D. Bjorner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z*, volume 428 of *Lecture Notes in Computer Science*, pages 242–262. Springer-Verlag, 1990.

[6] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, Redwood City, CA, 1989.

[7] 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.

[8] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[9] 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.

[10] M. Herlihy. Extending multiversion time-stamping protocols to exploit type information. *IEEE Transactions on Computers*, 36(4):443–448, April 1987.

[11] 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.

[12] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[13] R. Rastogi, H. F. Korth, and A. Silberchatz. Exploiting transaction semantics in multidatabase systems. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 101–109, Vancouver, Canada, June 1995.

[14] J.M. Spivey. *The Z Notation: A Reference Manual, Second Edition*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[15] W. E. Weihl. *Specification and Implementation of Atomic Data Types*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1984.

[16] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.