Applying Formal Methods to 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

Email: {pammann, jajodia, imukherj}@isse.gmu.edu

Contact Author: Professor Sushil Jajodia, Mail Stop 4A4, George Mason University, Fairfax, VA 22030-4444. Telephone: 703–993–1653/1640 (dept.), Fax: 703–993-1638, Internet: jajodia@gmu.edu

A preliminary version of this paper will appear under the title 'Using formal methods to reason about semantics-based decomposition of transactions' in *VLDB '95: Proceedings of the Twenty-First International Conference On Very Large Data Bases*, Zurich, Switzerland, September 1995.

*Partially supported by National Science Foundation under grant number CCR-9202270

†Partially supported by a grant from ARPA, administered by the Office of Naval Research under grant number N0014-92-J-4038, by National Science Foundation under grant number IRI-9303416, and by National Security Agency under contract number MDA904-94-C-6118

‡Partially supported by a George Mason University Graduate Research Fellowship Award

Abstract

In some important 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 this paper, we focus on the decomposition process itself. A decomposition generates a set of proof obligations that must be satisfied to show that the decomposition correctly models 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 implement the resulting, verified decomposition in a two-phase locking environment.

1 Introduction

Performance requirements can force a transaction to be decomposed into smaller logical units (we call these *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 – for example, when the customer makes reservations by phone – an implementation might force the three steps in the reserve transaction to occur separately. Although researchers have recognized the need for decomposing a set of transactions into steps, we analyze the decomposition process in a way that allows correctness with respect to the original transactions to be assessed formally. Each aspect of our decomposition process is accompanied by corresponding proof obligations, and we give an efficient, two-phase locking implementation 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 its 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 consistent state. 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 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 into steps 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, 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 which semantic histories must satisfy to show that a particular decomposition correctly models the original collection of transactions.

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 decompo-

$\mathbb{N}$	Set of Natural Numbers
$\mathbb{P}\, A$	Powerset of Set A
$\#A$	Cardinality of Set A
$\setminus$	Set Difference (Also schema 'hiding')
$A \fatsemi B$	Forward Composition of A with B
$x \mapsto y$	Ordered Pair $(x,\ y)$
$A \pfun B$	Partial Function from A to B
$A \pinj B$	Partial Injective Function from A to B
$B \ndres A$	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
ΔA	Before and After State of Schema A
ΞA	ΔA with No Change to State

Table 1: Z Notation

sition, are presented in Section 4. A graph-based characterization of correctness is given in Section 5, followed by an implementation in a two-phase locking environment in Section 6. Section 7 relates our work to the literature. Section 8 concludes the paper.

We adopt the Z specification language [Spi89] for expressing model-based specifications. Z is based on set theory, first order predicate logic, and a schema calculus to organize large specifications. Knowledge of Z is helpful, but not required, for reading this paper, since we narrate the formal specifications in English. Table 1 briefly explains the Z notation used in our examples. Other specification and analysis conventions peculiar to Z are explained as the need arises.

2 The Hotel Database

We illustrate our ideas with an example of a hotel database. A Z specification of the hotel database appears in figure 1. The hotel database has a set of objects, two integrity

$[Guest, Room]$

$Status ::= Available \mid Unavailable$

$total : \mathbb{N}$

Hotel

$res : \mathbb{N}$

$RM : Guest \rightarrowtail Room$

$ST : Room \twoheadrightarrow Status$

$guest : \mathbb{P}\, Guest$

$\#RM = res$

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

Reserve

$\Delta Hotel$

$g? : Guest$

$r! : Room$

$res < total$

$g? \notin guest$

$ST(r!) = Available$

$res' = res + 1$

$ST' = ST \oplus \{r! \mapsto Unavailable\}$

$RM' = RM \cup \{g? \mapsto r!\}$

$guest' = guest \cup \{g?\}$

Cancel

$\Delta Hotel$

$g? : Guest$

$g? \in guest$

$res' = res \Leftrightarrow 1$

$ST' = ST \oplus \{RM(g?) \mapsto Available\}$

$RM' = \{g?\} \mathbin{\Leftrightarrow} RM$

$guest' = guest \setminus \{g?\}$

Report

$\Xi Hotel$

$currentstatus! : Room \twoheadrightarrow Status$

$currentassignments! : Guest \rightarrowtail Room$

$currentstatus! = ST$

$currentassignments! = RM$

Figure 1: Initial Specification of the Hotel Database

constraints on these objects, and three types of transactions, which we identify and explain below.

The specification assumes two types, *Guest* and *Room*, which enumerate all possible guests and all possible hotel rooms, respectively. The global variable *total* is number of rooms in the hotel.

In Z states are described with a two-dimensional graphical notation called a *schema*, in which declarations for the objects in the state appear in the top part and constraints on the objects appear on the bottom part. The objects in the hotel database are listed in the schema *Hotel*, which defines the state of the hotel.

The object *res* is a natural number that records the total number of reservations, RM is a partial injection that relates guests to rooms, ST is a partial function that records the status of each room in the hotel, and *guest* records the set of guests.

The integrity constraints on the objects in hotel database appear in the bottom part of *Hotel*. There are two integrity 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. $\mathrm{dom}(ST \rhd \{Unavailable\}) = \mathrm{ran}\, RM$. The set of rooms that are unavailable ($\mathrm{dom}(ST \rhd \{Unavailable\})$) is exactly the set of rooms reserved by guests ($\mathrm{ran}\, RM$). In other words, every unavailable room must be associated with some guest.

The three types of transactions in the hotel database are *Reserve*, *Cancel*, and *Report*. *Reserve* takes as input a 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. (Our particular example does not allow guests to register multiple times.) *Reserve* has a postcondition that room $r!$ with status *Available* is chosen, the total number of reservations *res* is incremented, the status of $r!$ is changed to *Unavailable*, the ordered pair $g? \mapsto r!$ is added to the function RM, and $g?$ is added to the set *guest*.

Cancel takes as input a guest $g?$ and cancels that guest's reservation. *Cancel* has a precondition that the guest $g?$ is in *guest*. *Cancel* has a postcondition that *res* is decremented, the status of the room assigned to $g?$ is changed to *Available*, $g?$ is removed from the domain of the function RM, and $g?$ is removed from the set *guest*.

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.

3 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 in the state. A database state is said to be *consistent* if the values of the objects satisfy the given invariants.

A *transaction* is an operation on a database state. Associated with each transaction is a set of *preconditions* and a set of *postconditions* on the database objects. 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 unavailable 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 break up a transaction into steps, and execute each of these steps as an atomic unit. The decomposition exploits the semantic information associated with the transaction. Although such a decomposition process is very much application specific, nevertheless, we identify necessary properties that must be satisfied by a decomposition.

Definition 1 [Transaction Decomposition] 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.

As one check on the decomposition, we must demonstrate that the steps, when executed in the correct sequence and without interference from other transactions, model the execution of the original transaction.

One possible composition requirement is that the steps in a decomposition be treated exactly as transactions in the original system, in that the integrity constraints must hold after each step. As the decomposition below demonstrates, such a requirement is too strong in practice. After presenting a naive decomposition, we develop a more realistic composition property.

3.1 A Naive Decomposition of the Reserve Transaction

Suppose we break up the *Reserve* transaction into the following three steps:

Step 1: Increment the number of reserved rooms (*res*).

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

Step 3: Add the guest to the set of guests and assign the room to the guest.

Thus, *Reserve* will be implemented as three atomic steps, rather than as a single action. A naive specification of these steps is given in figure 2.

The above decomposition has a serious flaw in that none of the proposed steps, considered by itself, maintains the invariants in *Hotel*. For example, *NaiveR*1 does not maintain the invariant $\#RM = res$ since *NaiveR*1 increments the value of *res*, but does not alter

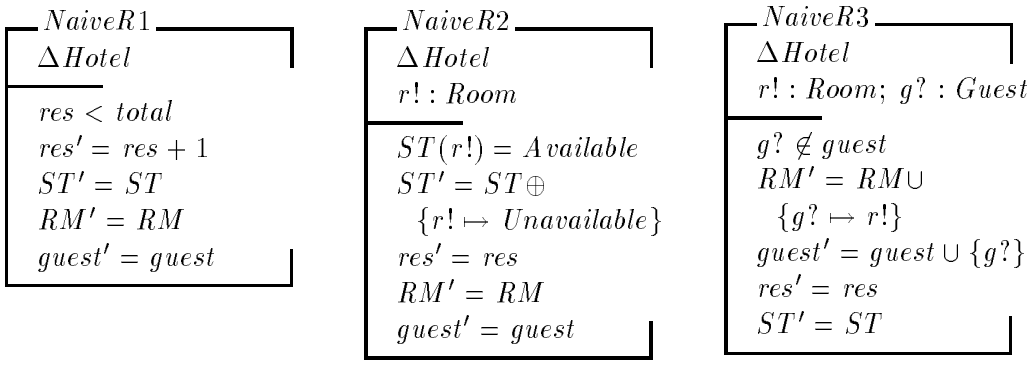


Figure 2: A Naive Decomposition

RM. Formally, the computed preconditions of all three steps simplify to false, indicating that none of the steps can be safely executed in an implementation. Executing one of the proposed steps would leave the invariants unsatisfied, and other transactions could be exposed to the inconsistent state. For example, *Report* may produce an inconsistent output if executed in a state in which the invariants do not hold.

3.2 Modification of Original Invariants

The previous example demonstrates that not all decompositions are acceptable. Specifically, a decomposition may yield steps that leave the database in a state in which the invariants are not satisfied. This possibility is illustrated for the hotel example by the arrow labeled *NaiveR1* in figure 3(a). Once the invariants are violated, the formal basis for assessing the correctness of subsequent behavior collapses.

As noted earlier, one way to solve this problem is to allow only those decompositions that have the property that partial executions leave the database state consistent. Such an approach is exceedingly restrictive, and so we reject it. In the hotel example, the informal description of the steps into which *Reserve* is broken is perfectly satisfactory; what is unreasonable is the insistence 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, e.g. *ValidCancel*, 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

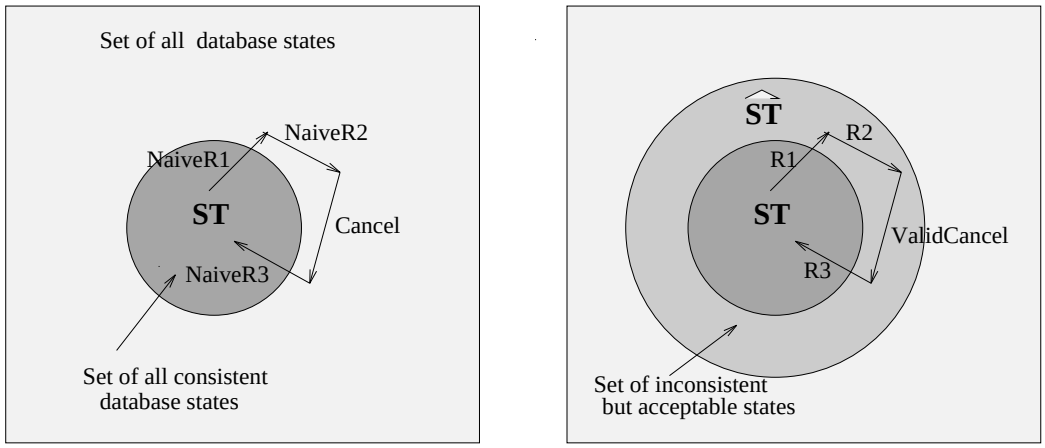
such that each step satisfies the new set of invariants. The model in figure 3(b) has many advantages, including greater concurrency among steps. We formalize the model as follows.

Let I denote the original invariants, and let $\mathbf{ST}$ denote the set consisting of all consistent states; i.e., $\mathbf{ST} = \{ST : ST \text{ satisfies } I\}$. A transaction T_i always operates on a consistent $ST \in \mathbf{ST}$. If ST_i denotes the state after the execution of T_i, then ST_i is also in $\mathbf{ST}$. 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 $\mathbf{ST}$. 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 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 $\widehat{\mathbf{ST}} = \{ \text{ ST} : \text{ST satisfies } \hat{I}\}$. 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 the correctness of decomposing transactions into steps, and 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 [OG76] and, in particular, have been applied to the problem of semantic database concurrency control [GM83, Appendix C]. Our work focuses more on the decomposition process than does [GM83], and so we emphasize the role of auxiliary variables more strongly.

In the hotel example, we generalize the invariant $\#RM = res$ by introducing an auxiliary variable to express the fact that number of guests with rooms might differ from total reservations by the number of reserve transactions in progress. We generalize the invariant $\text{dom}(ST \rhd \{Unavailable\}) = \text{ran}\, RM$ by introducing another auxiliary variable to express the fact that the unavailable rooms might differ from the rooms assigned to guests by those rooms selected by reserve transactions in progress. Before we show these changes to the 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 the implementation.



(a) General classification of database states

(b) Database states as classified in our model

Figure 3: Classification of the Database States

3.3 Composition Property

With the notion of generalized invariants 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 $T_{i1}, T_{i2}, \ldots, T_{in}$ denote the corresponding steps. T_i and its steps are related as follows:

$$T_i \Leftrightarrow (T_{i1} \fatsemi T_{i2} \fatsemi \ldots \fatsemi T_{in}) \wedge I$$

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.

From an implementation perspective, the composition property is similar to requiring that the stepwise execution of the steps be view equivalent to that of the original transaction. A complicating factor is that the decomposition may introduce additional database objects; the composition property does not limit the values of these additional database objects (for example, compare *Hotel* in figure 1 with *ValidHotel* in figure 4).

3.4 Sensitive Transaction Isolation Property

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 [GM83]. 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 T_i may be running on a database state in $\widehat{\mathbf{ST}} \Leftrightarrow \mathbf{ST}$.

In our model, we ensure the sensitive transaction isolation property by construction. For each sensitive transaction, we compute the subset of the original integrity constraints, I, relevant to the calculation of any outputs. This subset of I must be implied by the precondition of the sensitive transaction.

3.5 A Valid Decomposition

In this section, we provide a valid decomposition of the hotel database. The problems noted in the previous decomposition are avoided, and the necessary properties identified so far hold. After presenting the example, we derive additional properties that valid decompositions must have.

To make the invariants more general, we add auxiliary variables and define a new state *ValidHotel*. 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 invariants are modified accordingly. The schema *ValidHotel* together with the modified 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 three steps satisfy the composition property; see the appendix for a further discussion of this point. 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 the appendix for details.

The refined version of the single step *Cancel* transaction is nearly identical to the unrefined version, except that the auxiliary variables *tempassigned* and *tempreserved* are specified as remaining unchanged.

Report is a sensitive transaction, and we establish the sensitive transaction isolation

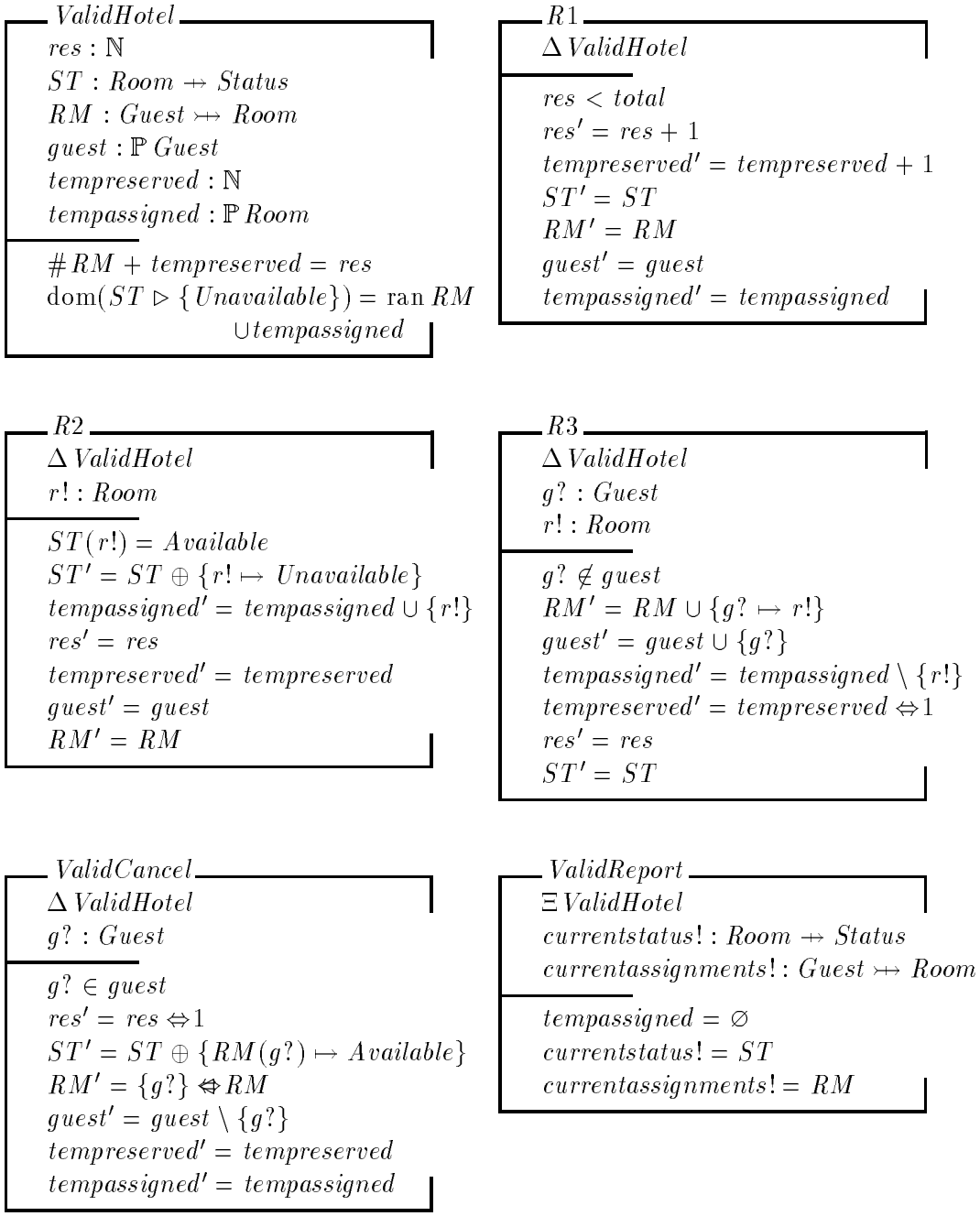


Figure 4: A Correct Decomposition for the Hotel Database

property by construction. A formal treatment is given in the appendix. Informally, *Report* transaction outputs values of ST and RM. ST and RM involve the following original invariant:

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

which can be derived from

$$\mathrm{dom}(ST \rhd \{Unavailable\}) = \mathrm{ran}\, RM \cup tempassigned$$

if the auxiliary variable *tempassigned* satisfies $tempassigned = \varnothing$.

3.6 Semantic Histories

We are interested in the relationship between the original specification and the specification with the modified invariants. In particular, we would like to know if and when the database state returns to a consistent state, and whether outputs reflect a state that satisfies the original invariants, and not just the modified ones. We return to these questions after giving some definitions.

In the previous subsections we showed how a *Reserve* transaction can be decomposed into steps $R1$, $R2$ and $R3$. Before we proceed further, we make a distinction between a type of a step and an instance of a step. $R1$, $R2$, $R3$, *ValidReport*, *ValidCancel* represent the different *types* of steps in the Hotel Database. On the other hand, 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 2 [Type] Steps of a transaction in a system are classified into a set of *types*. Let $TYPES$ be the set of all types of steps which are run by the system. The type of step T_{ij} is denoted by $ty(T_{ij})$.

Example 1 The set of all types of steps for the Hotel Database is given by,

$$TYPES = \{R1, R2, R3, ValidReport, ValidCancel\}$$

□

Definition 3 [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 $< T_{i_1 j_1}, T_{i_2 j_2}, \ldots, T_{i_p j_q} >$, $1 \leq i_1, \ldots, i_p \leq m$, $T_{i_r j_s}$ is a step in T_{i_r}, $1 \leq i_r \leq m$, $1 \leq j_s \leq n$, n is the number of steps in transaction T_{i_r}, 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.

Note: At this point, our notion of a history does not reference the typical operations on data elements, such as read and write. These operations are introduced and integrated into the definition of histories as we refine our specifications.

Example 2 $< T_{11}, T_{13}, T_{21}, T_{12} >$ is not a stepwise serial history since it violates condition (2) and (3). $< T_{11}, T_{12}, T_{13}, T_{12} >$ is not a stepwise serial history since it violates condition (1). $< T_{11}, T_{21}, T_{12}, T_{13} >$ is a stepwise serial history. □

Definition 4 [Complete Execution] An execution of a transaction $T_i = < T_{i1}, T_{i2}, \ldots, T_{in} >$ in a stepwise serial history H is a *complete* execution if all n steps of T_i appear in H.

Example 3 For the hotel database, an execution of the reserve transaction T_i will be complete in a stepwise serial history H if all three steps T_{i1}, T_{i2}, and T_{i3} of T_i appear in H. Note that $ty(T_{i1}) = R1$, $ty(T_{i2}) = R2$, $ty(T_{i3}) = R3$. □

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

Definition 5 [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.

Although we usually use the term partial semantic history for cases in which the execution of at least one transaction actually is incomplete, we find it convenient to define complete semantic histories as a special case of partial semantic histories.

Definition 6 [Partial Semantic History] A partial semantic history is simply a semantic history.

Definition 7 [Complete Semantic History] A semantic history H over $\mathbf{T}$ is a *complete* semantic history if the execution of each T_i in $\mathbf{T}$ is *complete*.

3.7 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.

Although consistent execution property is definitely desirable, it is not enough because it does not capture the cumulative effect of each transaction. For a semantic history to be correct, we require that all intermediate states be in $\widehat{\mathbf{ST}}$, which is formalized in following definitions. Note that the consistency of outputs is ensured by the sensitive transaction isolation property.

Definition 8 [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 9 [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 $\mathbf{ST}$.

3.8 Complete Execution Property

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 correct partial semantic history H_p is a prefix of some correct complete semantic history.

In the hotel database suppose we have a correct partial semantic history H as shown below:

$$H =< T_{11}, T_{12} >$$

where $ty(T_{11}) = R1$ and $ty(T_{12}) = R2$. Let the reserve transaction T_1 execute with $g? =$ John. Consider step T_{13} where $ty(T_{13}) = R3$. The precondition John $\notin$ *guest* of T_{13} requires that John not have a existing reservation, but it is possible that in the final state in H, John *is* an element of *guest*. We may cancel John's existing reservation, by executing the single step cancel transaction T_{21}. This will allow the incomplete reserve transaction T_1 to complete. First, the precondition of T_{21}, John $\in$ *guest*, is guaranteed to hold if the precondition of T_{13} does not hold. Second, the postcondition of T_{21} establishes the precondition of T_{13}. Thus the reserve transaction for John can complete, and the correct partial semantic history H can be extended to be a correct complete semantic history.

3.9 A Decomposition Lacking Complete Execution Property

In this section, we show that some otherwise plausible decompositions do *not* satisfy the complete execution property, which is clearly undesirable. To illustrate the possibility, we modify the hotel database to yield the *DeadLockHotel* database shown in figure 5.

In the example specification, the cancel type of transaction is decomposed into steps of type $C1$ and $C2$. We introduce the auxiliary variable *tempcanceled* which keeps count of the cancel transactions that have completed a step of type $C1$ but not one of type $C2$. The invariant

$$\#RM = res \Leftrightarrow tempreserved$$

in the original *ValidHotel* is changed to:

$$\#RM = res \Leftrightarrow tempreserved + tempcanceled$$

Moreover, we introduce a new structure *clist* which keeps track of the guests whose cancellations are in progress. The guest whose reservation is being canceled is added to the *clist* in first step of cancel and is removed from the *clist* in the second step. We impose an additional constraint that a room cannot be reserved for a guest whose cancellation is in

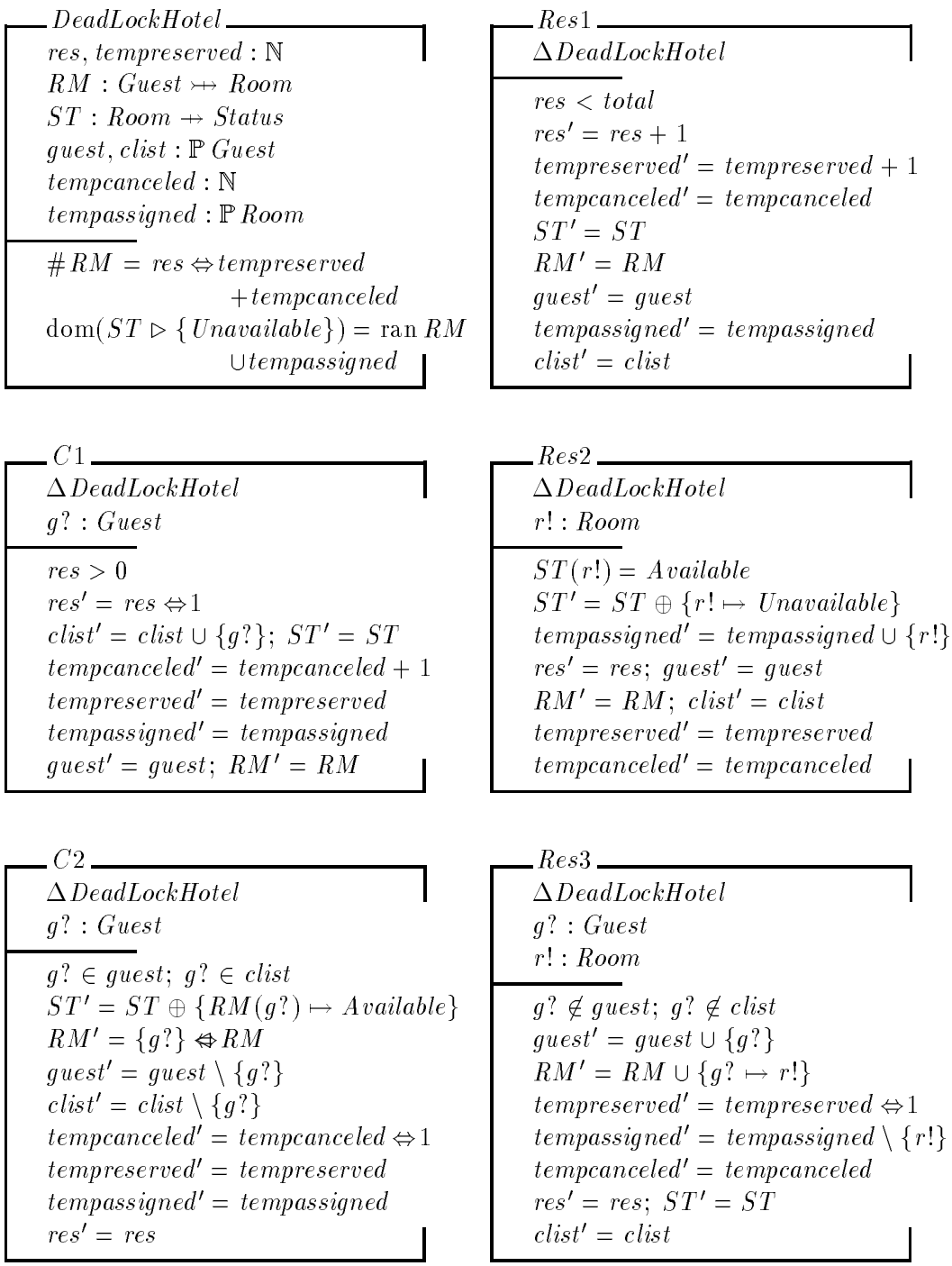


Figure 5: Example Specification Lacking Complete Execution Property

progress; note the precondition $g? \notin clist$ in step $Res3$. The reserve transaction is broken into steps $Res1$, $Res2$ and $Res3$, which are similar to $R1$, $R2$ and $R3$ of the *ValidHotel* specification.

Consider the correct partial semantic history $H_p =< T_{11}, T_{21}, T_{22} >$, where $ty(T_{11}) = C1$, $ty(T_{21}) = Res1$ and $ty(T_{22}) = Res2$.

Let the reserve and cancel transactions in H_p have the same $g?$ as their input. Also assume that $g? \notin guest$. We try to complete the reserve and cancel transactions. T_{23} cannot be executed because the precondition $g? \notin clist$ is not satisfied as step T_{11} has inserted $g?$ in *clist*. T_{12} cannot be executed because the precondition $g? \in guest$ is not satisfied. It is possible to execute any number of steps of other transactions, but the reserve and cancel transactions in H_p still cannot complete.

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 T_{23} nor T_{12} can complete.

At this point in our presentation we have described a method by which transactions can be decomposed into steps in a manner that supports reasoning about the correctness of the decomposition. The decomposition process introduces additional database objects and additional preconditions on steps. The additional objects and preconditions are present exclusively to support analysis. For efficient implementation, we want to avoid instantiating the objects and checking the preconditions. Successor sets are the mechanism we use to achieve this objective.

4 Successor Sets

4.1 Conflict

For the semantic-based decomposition in this paper to be useful, the specifications must be implementable in the framework of some concurrency control mechanism. The specification given so far indicates what operations are performed by the steps. Corresponding to the specification, an implementation must describe the details of the steps in term of database read and write operations and also some local processing. Our eventual aim is to get a concurrency control mechanism in which steps need not be executed atomically, but the read, write operations of one step can be interleaved with those of another. To meet this requirement we introduce the notion of conflicting operations, conflicting steps, and conflict-oriented histories.

Type of Step	Variables Read	Variables Written
$R1$	res, $total$	res
$R2$	ST	ST
$R3$	RM, $guest$	RM, $guest$
$ValidReport$	ST, RM	
$ValidCancel$	res, ST, RM, $guest$	res, ST, RM, $guest$

Table 2: Read and Write Sets for Steps of Hotel Example

Definition 10 [Conflicting Operations] Two operations are said to *conflict* if they both operate on the same data item and at least one of them is a Write.

Definition 11 [Conflicting Steps] Two steps T_{ij} and T_{kl} *conflict* if they contain conflicting operations.

It is easy to determine the set of conflicting steps once an implementation is given. However at this stage we only have the specification. From the specification, we would like to define a notion of conflict. In a specification, it is not always possible to accurately identify the exact set of variables that will be read or written in the final implementation. Nonetheless we define the following variables as being *written* in the specification:

1. Explicitly modified variables – Any state variable where the after state has some new explicit value, or
2. Unconstrained variables – Any state variable where the after state is unconstrained. In such cases the implementation is free to alter the variable, and we assume that the variable is, in fact, modified.

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 x referenced in a postcondition other than $x' = x$.

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.

Armed with a notion of conflict that is valid for both specifications and implementations, we can now proceed to define a history in terms of conflicts in a manner similar to [BHG87].

Type of Step	Types of Conflicting Steps
$R1$	$R1$, *ValidCancel*
$R2$	$R2$, *ValidReport*, *ValidCancel*
$R3$	$R3$, *ValidReport*, *ValidCancel*
ValidReport	$R2$, $R3$, *ValidCancel*
ValidCancel	$R1$, $R2$, $R3$, *ValidReport*, *ValidCancel*

Table 3: Conflicting Steps for Hotel Example

Definition 12 [History] A history H defined over a set of transactions $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$, where each transaction T_i has been decomposed into i_n steps, is a partial order with ordering relation $\prec_H$ where:

1. $H = \cup_{i=1}^{m} \cup_{j=1}^{i_n} T_{ij}$;
2. $\prec_H \supseteq \cup_{i=1}^{m} \cup_{j=1}^{i_n} \prec_{ij}$; 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 T_1, T_2, ..., T_m. Condition (2) says that the H preserves the order of operations of the steps. Condition (3) says that every pair of conflicting operations are ordered in H.

We point out that in an actual implementation, the set of conflicts may be different than those obtained at the specification level. For example, two steps may conflict in the specification but not at the implementation. Consider the hotel example. In the specification, the object RM covers the entire hotel; the guest information for all rooms is integrated into a single object. An reasonable implementation may use a separate object for each room. Thus, updates to RM always conflict at the specification level, but only conflict at the implementation level if the updates are to the same specific room.

4.2 Avoiding Precondition Checks

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, which is based on generalizing invariants with auxiliary variables, the interleaving is constrained by additional preconditions on the auxiliary variables. Although the generalized invariants facilitate analysis, it is expensive to implement the auxiliary

variables. Also performing checks on the additional preconditions (which may be quite complex) involves extra run time overhead. To avoid implementing auxiliary variables and to avoid 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}, $T_{i(j+1)}$, ..., $T_{i(k-1)}$ 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 semantic 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 13 [Successor Set] The *successor set* of $ty(T_{ij})$, denoted $SS(ty(T_{ij}))$, contains a set of types of steps such that if $ty(T_{pq}) \in SS(ty(T_{ij}))$, then step T_{ij} leaves the database in a state where it may be safe for T_{pq} to execute.

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

With the help of successor sets we wish to eliminate the preconditions involving auxiliary variables. For this purpose, we need to ensure that in all the allowable histories obtained from the successor set description, the precondition being removed always evaluates to true. The computation of successor sets is done statically at the specification level; however it is

not always possible to determine statically how the precondition being removed will actually evaluate during execution. For example, the precondition may depend in some subtle way on the order of prior steps in the history. Fortunately, the uncertainty about preconditions can be overcome easily.

Suppose steps of type B have a precondition involving an auxiliary variable. We wish to replace this precondition check with successor set description. Our problem is to decide whether B should be in the successor set of step A. There can be three cases,

1. The precondition of a step of type B is always falsified by the execution of a step of type A. In this case we exclude B from the successor set of A.

2. The precondition of a step of type B is never falsified by the execution of a step of type A. In this case we include B in the successor set of A.

3. The precondition of a step of type B is sometimes, but not always, falsified by the execution of a step of type A. In this case we act conservatively and exclude B from the successor set of A.

We do not want the conservative approach taken in the third case above to overly constrain the implementation. Let T_{il} be a step of type A and T_{pq} be a step of type B. During execution, if T_{pq} does not conflict with step T_{il}, or any step previous to T_{il}, then step T_{pq} may proceed concurrently with step T_{il}. On the other hand if a step T_{pq} conflicts with step T_{il}, or some step previous to T_{il}, we allow T_{pq} to execute only if B is in the successor set of A. Note that preconditions involving auxiliary variables evaluate to false only when T_{pq} is executed after T_{il} where T_{pq} conflicts with T_{il} or some step previous to T_{il}.

More generally, any semantic history H generated using successor sets must meet the following additional requirement besides those given in either Definition 8 or Definition 9, depending on whether H is partial or complete:

Definition 14 [Correct Successor Set History] In a correct semantic history H, if 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}.

Successor Set of Type of Step	Types of Steps in Successor Set
$SS(R1)$	$R1$, $R2$, $R3$, *ValidReport*, *ValidCancel*
$SS(R2)$	$R1$, $R2$, $R3$, *ValidCancel*
$SS(R3)$	$R1$, $R2$, $R3$, *ValidReport*, *ValidCancel*
$SS($*ValidReport*$)$	$R1$, $R2$, $R3$, *ValidReport*, *ValidCancel*
$SS($*ValidCancel*$)$	$R1$, $R2$, $R3$, *ValidReport*, *ValidCancel*

Table 4: Successor Sets for the Hotel Example

Example 4 Successor set descriptions are obtained by examining the preconditions with auxiliary variables. In the hotel example, the only precondition with auxiliary variables is $tempassigned = \emptyset$ in step type *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. We specify the successor sets as in Table 4. Also note that after the last step of a transaction has been executed, it should be possible to execute any step of any other transaction. Thus the successor set of $R3$, *ValidReport* and *ValidCancel* contains all types of steps.

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 $R2$.

Note that for the hotel example, all of the preconditions in auxiliary variables are captured by the successor set description, and so none of the auxiliary variables need to be implemented. For some applications, it may not be possible to eliminate all of the preconditions in auxiliary variables, and so the referenced auxiliary variables must indeed be implemented. □

Successor set descriptions are intended to allow the removal of predicates that reference auxiliary variables, and ultimately the auxiliary variables themselves. Therefore, with respect to the specifications given with generalized invariants and auxiliary variables, not all successor set descriptions are valid. Informally, a successor set is valid with respect to a generalized invariant specification if any correct successor set history can also be generated by the generalized invariant specification. Although desirable, the converse property does not hold in general since first-order logic preconditions have more expressive power than

the successor set mechanism. Formally, we describe valid successor set descriptions with the *valid successor set property*:

Definition 15 [Valid Successor Set Property] A specification S_2 that employs a successor set description is *valid* with respect to specification S_1 with generalized 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 complete execution property.

If the set of correct semantic histories generated by S_1 is identical to the set of correct successor set histories generated by S_2, 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. The appendix contains a proof of the valid successor set property for the hotel example, where it turns out that the successor set specification generates exactly the same set of histories as the specification with preconditions in auxiliary variables.

Before concluding this section, we give one more definition. 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 such that the resulting history remains correct. (The *next* function is exactly the F function in [FÖ89].)

Definition 16 [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}, $T_{i(j+1)}$, ..., $T_{i(j+n)}$ 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:

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), $ty(T_{pq}) \notin SS(ty(T_{ir}))$.

Example 5 Consider two transactions T_1 and T_2 from our Hotel example.

$T_1 = T_{11}, T_{12}, T_{13}$
$T_2 = T_{21}$
$ty(T_{11}) = R1, ty(T_{12}) = R2, ty(T_{13}) = R3$
$ty(T_{21}) = ValidReport$

Successor set descriptions are given in Table 4.

Some of the next functions of the above example are given by:

$NF(T_{11}, T_{21}) = T_{11}$
$NF(T_{12}, T_{21}) = T_{13}$
$NF(T_{13}, T_{21}) = T_{13}$

□

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}, $T_{i(j+1)}$, $T_{i(j+2)}$, ... $T_{i(p-1)}$, 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 1 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 $NF(T_{ij}, T_{kl})$. □

5 Correct Stepwise Serializable Histories

Recall that 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.

Definition 17 [Equivalence of Histories] Two histories H and H' are said to be equivalent if [BHG87]:

1. they are defined over the same set of steps (transactions) and have the same operations; and
2. they order conflicting operations of steps in the same way; that is, for any conflicting operations p_{ij} and q_{kl} belonging to steps T_{ij} and T_{kl} (respectively), if $p_{ij} \prec_H q_{kl}$, then $p_{ij} \prec_{H'} q_{kl}$. [$p_{ij} \prec_H q_{kl}$ denotes p_{ij} precedes q_{kl} in H].

Definition 18 [Correct Stepwise Serializable History] A *correct stepwise serializable history* is one which is equivalent to a correct successor set history.

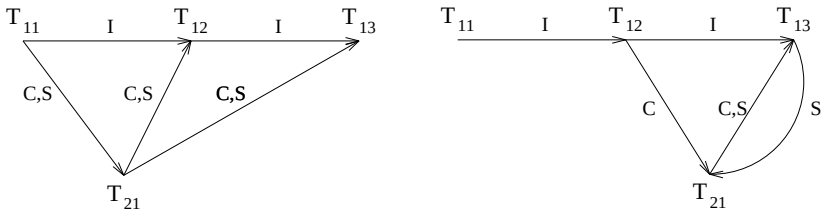


Figure 6: Precedence Graph For Histories In Examples 6,7

5.1 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.

Definition 19 [Precedence Graph][FÖ89] 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 – There is a *successor* or S edge according to the following recursive rule. 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 path from T_{ij} to T_{kl}, there is a successor edge $(NF(T_{ij}, T_{kl}), T_{kl})$.

Each edge $T_{ij} \rightarrow T_{kl}$ in $PG(H)$ means that at least one of T_{ij}'s operation should precede T_{kl}. This means that in a correct successor set history equivalent to H, T_{ij} precedes step T_{kl}.

Example 6 Consider the following history H:

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

where

$ty(T_{11}) = R1$
$ty(T_{12}) = R2$
$ty(T_{13}) = R3$
$ty(T_{21}) = ValidCancel$

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

Example 7 Consider the following history H:

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

where

$ty(T_{11}) = R1$
$ty(T_{12}) = R2$
$ty(T_{13}) = R3$
$ty(T_{21}) = ValidReport$

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:

$\Leftarrow$: 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 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 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 also a correct stepwise serializable history.

$\Rightarrow$: Since H is a correct stepwise serializable history, it must be equivalent to some correct successor set history H'. Let $PG(H')$ be the precedence graph of the history H'. Since H

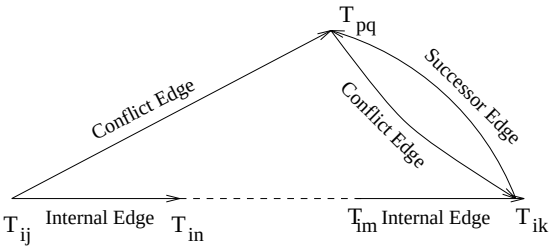


Figure 7: Edges of Precedence Graph involving T_{ij}, T_{ik}, T_{pq}

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 edges $(NF(T_{ij}, T_{kl}), T_{kl})$ are added recursively if there is a path between T_{ij} and 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 $PG(H) = PG(H')$. Assume there is a cycle in $PG(H)$. Let the cycle be $T_{ij} \rightarrow T_{kl} \rightarrow T_{mn} \cdots T_{ij}$. This implies that in H', $T_{ij} \prec T_{kl} \prec T_{mn} \cdots \prec T_{ij}$ – a contradiction since H' is serial. Hence $PG(H)$ is acyclic. □

The theorem presented here is stronger than the corresponding result in [FÖ89]. Specifically, the result in [FÖ89] also requires that there exist a linearization that corresponds to a correct successor set history. Constraint 1 permits us to avoid the additional requirement.

6 Implementation

In this section, we implement our mechanism in a two-phase locking environment. Although the mechanism we propose is based on the scheme in [FÖ89], ours is considerably simpler; see Section 7 for details. Our mechanism uses the standard 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. The appropriate lock is acquired before the step reads or writes the data item. A lock can be acquired only if the current step is in the successor set of all the active transactions that have locked the data item. In our mechanism the locks acquired by a step are released when the step commits.

Before describing our mechanism, 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:

$T_i =< T_{i1}, T_{i2}, \ldots, T_{in}, TR(T_i) >$

6.1 Data Structures

In addition to the data structures required by the standard two phase locking protocol, we require the following data structures.

1. Active-Set - Set of Active Transactions
2. 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} reads or writes x, 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. The interleaving set Int-set(T_i,x) is initialized to empty at the time a step T_{ij} of T_i reads or writes x. 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.

6.2 Algorithms

The mechanism requires the following assumptions:

1. 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.
2. If a transaction reads and writes the same data entity x, the read operation precedes the write operation.

Algorithm 1 Algorithm for Read

Procedure Process-read (R_{ij}(x))
begin
  if a step T_{lm} is holding an exclusive lock on x
    begin
      delay R_{ij}(x);
      exit;

```
        end
    for each T_k ∈ Active-set(x)
        if ty(T_ij) ∉ Int-set(T_k,x)
            begin
                delay R_ij(x);
                exit;
            end
    if T_i ∉ Active-set(x)
        Active-set(x) = Active-set(x) ∪ T_i;
    lock x in shared mode;
    accept(R_ij(x));
    Int-set(T_i,x) = ∅;
end
```

Algorithm 2 Algorithm for Write

```
Procedure Process-write (W_ij(x))
begin
    if a step T_lm is holding any lock on x
        begin
            delay W_ij(x);
            exit;
        end
    for each T_k ∈ Active-set(x)
        if ty(T_ij) ∉ Int-set(T_k,x)
            begin
                delay W_ij(x);
                exit;
            end
    if T_i ∉ Active-set(x)
        Active-set(x) = Active-set(x) ∪ T_i;
    lock x in exclusive mode
    accept(W_ij(x));
    Int-set(T_i,x) = ∅
end
```

Algorithm 3 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**

Release the lock on x which was acquired by T_{ij};

end

Algorithm 4 Algorithm for Transaction Terminate

Procedure Process-terminate(TR(T_i))

begin

for each entity x which was accessed by T_i **do**

begin

delete the values Int-set(T_i,x) ;

Active-set(x) = Active-set(x) $\Leftrightarrow T_i$;

end

end

6.3 Correctness of the Mechanism

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

Proof: This proof proceeds in two parts.

1. Any history generated by our mechanism is equivalent to some semantic history H.
2. The semantic history H is a correct successor set history.

(Proof of Part 1)
Consider any history generated by our mechanism. Assume we have a centralized lock manager and the lock release operation takes place atomically. The order of first lock release establishes a serialization order on the steps. The serialization order represents some semantic history H. All histories generated by our mechanism are conflict equivalent to H.

(Proof of Part 2)
We prove by induction on the number of steps in H that H is a correct successor set history.

In basis case where H has exactly one step, it is clear that H is a correct successor set history.

Assume that the result holds for any semantic history H_n consisting of n steps. Let the order of the first lock release of the steps be given by $< S_1, S_2, \cdots, S_n >$, where S_i denotes a step T_{kl} which has been renamed for convenience. Thus, by assumption, $H_n =< S_1, S_2, \cdots, S_n >$ is a correct successor set history.

Consider H_{n+1}, a semantic history of $n+1$ steps. Let the order of steps in which locks are released be given by $< S_1, S_2, \cdots, S_n, S_{n+1} >$. Thus $H_{n+1} =< S_1, S_2, \cdots, S_{n+1} >$.

Let $S_{n+1} = T_{pq}$. To show that H_{n+1} is a correct successor set history, we need to show that for all active transactions T_i, if T_{ij} is the last step of T_i which conflicts with T_{pq} and precedes T_{pq}, then $ty(T_{pq}) \in SS(ty(T_{ij}))$.

We proceed by contradiction. Let T_{ij} be the last step of an active transaction T_i that conflicts with and precedes T_{pq}. Assume that $T_{pq} \notin SS(T_{ij})$. Let $NF(T_{ij}, T_{pq}) = T_{ir}$. Since $ty(T_{pq}) \notin SS(ty(T_{ij}))$, $T_{ir} \neq T_{ij}$. In other words T_{ir} is some step after T_{ij}. By Constraint 1, T_{ir} conflicts with T_{pq}. Clearly T_{ir} does not occur before step T_{pq}, as T_{ij} is the last step which precedes and conflicts with T_{pq}. (The semantic history H_{n+1} does not contain step T_{ir}). Since T_{ij} and T_{pq} conflict, T_{ij} and T_{pq} must be accessing some common data item x. Let T_{ik} be the last step of T_i to be executed before T_{pq}. The order in which T_{pq}, T_{ij}, T_{ik} occur in the H_{n+1} is given by the relation $T_{ij} \prec T_{ik} \prec T_{pq}$. Now before the execution of T_{pq}, Int-set(T_i,x) = $SS(ty(T_{ik}))$. Since our mechanism allowed T_{pq} to access item x, $ty(T_{pq}) \in$ Int-set(T_i,x). This implies that $ty(T_{pq}) \in SS(ty(T_{ik}))$. But this is a contradiction, since T_{ik} occurs after T_{ij} and from the definition of next function, T_{ir} is the first step after T_{ij}, such that $ty(T_{pq}) \in SS(ty(T_{ir}))$. Therefore the assumption that $ty(T_{pq}) \notin SS(ty(T_{ij}))$ is false, and H_{n+1} is a correct successor set history.

□

7 Related Work

Most transaction-oriented models enforce a very low level, syntactic notion of consistency, namely serializability with respect to read/write conflicts [BHG87]. Some researchers [Her87, HW91, LMWF94, Wei84, Lyn83, Wei88] have expanded on this notion using the theory of abstract data types or ADTs. In these works, the data objects are classified into ADTs and high level operations are defined for the ADTs. These operations are the only means by which the transactions can access the data objects. Commutativity of the operations of the ADT, and not the read/write operations, is used to determine conflicts between transactions - this results in more concurrency. Still further concurrency can be achieved if recoverability [BR92] of operations is used instead of commutativity. In our work, we also adopt the theory of ADTs for the purpose of defining correctness of trans-

action execution. However, there is a significant difference in the two approaches. Unlike [Her87, HW91, Wei88], we define the entire database as an ADT, with the transactions being the operations on the ADT. Furthermore we are interested in achieving more concurrency by expanding the set of correct execution histories such that some transactions need not be atomic.

Other extensions to the classical transaction model include generalizing the notion of serializability and proposing new correctness criteria for transaction executions [AAS93, DE90, SLJ88, KS88, KS94, FÖ89, GM83]. For example, in [DE90], the authors develop *quasi-serializability* as a correctness criterion for transactions executing on heterogeneous distributed database systems. In some of these works, researchers have decomposed transactions into steps [AAS93, FÖ89, GM83, JM87, KS88, KS94, SLJ88, SSVR92] and developed semantic based correctness criteria [AAS93, FÖ89, GM83]. Researchers have variously introduced the notions of transaction steps, countersteps, allowed vs. prohibited interleavings of steps, and implementations in locking environments. Among these, the work of Farrag and Özsu [FÖ89] and Korth and Speegle [KS88, KS94] bear close resemblance to our work and are discussed in separate subsections.

In [SSVR92], the authors describe how a transaction can be decomposed into steps, without using any semantic knowledge of the transactions. The authors use the standard notion of serializability as the correctness criterion. The authors show that if the steps of the transactions in a history are serializable, then the history is equivalent to a serial history. Unlike our work, the decomposition of transactions into steps is based on conflict information only. Also, certain decompositions are valid in our model that are not allowed by their system. Specifically, the decomposition of the *Reserve* transaction into steps $R1$, $R2$ and $R3$ in the Hotel Database, will not be a correct decomposition according to their model.

In [SLJ88], the authors propose a notion of correctness based on database partitioning and transaction decomposition. The database is partitioned into *atomic data sets* using the knowledge of consistency constraints. The transactions are decomposed into *elementary transactions* using only the semantic information associated with a transaction. An elementary transaction must maintain the consistency constraints of the accessed atomic data sets. The authors introduce a correctness criteria known as *generalized setwise serializable*, in which it is required that the effect of execution of the elementary transactions on any atomic data set is equivalent to a serial execution of the elementary transactions on the data set. Generalized setwise serializable schedules maintain database consistency and also satisfy the post conditions of the transactions.

The authors in [AAS93] propose three correctness criteria - *consistency*, *orderability* and *strong orderability* for non-serializable executions. The data objects in their model are clas-

sified into various abstract data types. These objects are accessed through user programs. User programs are characterized by the presence of *consistency assertions*. Consistency assertions are predicates on database objects that are required to hold before an operation accessing the data objects are executed. The correctness criterion, consistency, is based on the users specifications and allows nonserializable executions that are acceptable to the users. The other two correctness criteria, viz. orderability and strong orderability are generalizations of view serializability and conflict serializability respectively. The user is responsible for writing correct programs and for specifying the correct consistency assertions in the user programs.

The work of Garcia-Molina [GM83] is based on the classification of the transactions into a number of *semantic types*. *Compatibility sets* are associated with each of these types. The author defines the notion of *sensitive transactions* and *semantically consistent schedules*. A transaction is sensitive if it has to output consistent database data to be viewed by the user. A semantically consistent schedule is one such that it maintains the consistency of the database, and any sensitive transaction in it obtains a consistent view of the data. In a semantically consistent schedule, the steps of transactions belonging to the same compatibility set can be interleaved arbitrarily, whereas steps of transactions that do not belong to the same compatibility set can be interleaved as much as permitted by the serializability constraints. The user is responsible for classifying the transactions into types, for decomposing the transaction into steps, and for providing the compatibility sets.

ACTA [CR94] is a framework for formally specifying extended transaction models. ACTA provides components, (e.g. operations, significant events, conditions on events, transaction dependencies) for describing the transaction models. However, our model cannot be represented within the ACTA framework. This is because ACTA conditions cannot be used to express the preconditions and postconditions of a semantic history. Thus extensions to the ACTA framework are required to represent our model.

7.1 Korth and Speegle

In [KS88, KS94] the authors introduce a transaction model called NT/PV model for long duration transactions. The NT/PV model incorporates the idea of nested transactions, multiple versions and explicit predicates to increase concurrency. A transaction, denoted by (T, P, I, O), is characterized by the set of subtransactions T constituting the transaction, the partial order P among the subtransactions, the input conditions or preconditions I and the output conditions or postconditions O. An execution of a transaction is characterized by a pair (R, X), where R is a run time dependency relation between the subtransactions of the transaction, and X denotes the state associated with the transaction. An execution

of a transaction is NT/PV correct if (i) the subtransactions are executed in a state that satisfies the preconditions of the subtransactions, and (ii) the post conditions of the last subtransaction satisfy the post conditions of the transaction. An execution of an interleaved set of transactions is NT/PV correct if every transaction in the set executes correctly. The authors show how classical transactions and various notions of serializability can be represented by their model and propose an implementation which generates only NT/PV correct executions.

Though a transaction in our model can be represented as a NT/PV transaction, our notion of correct execution of a set of transaction is very different from that proposed in [KS94]. A transaction in our model can be represented in their framework as follows - $t = (T, P, I, O)$, where the set T represents the set of steps of the transactions, P is a total order on the steps, I represent the preconditions of the transaction, O represent the post conditions of the transaction. In the NT/PV model, the post conditions of top level transactions imply the satisfaction of database integrity constraints. As stated in [KS94], one of the requirements for correct executions is that the post conditions of the last subtransaction of a transaction should satisfy the transaction's post conditions. This implies that in a correct execution, the last subtransaction of a top level transaction should satisfy the database integrity constraints. In our notion of correct execution of a set of transactions (which we call *correct semantic history*) we do not require the post condition of last step of a transaction to satisfy the database integrity constraints.

Example 8 For example consider the following history.

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

where $ty(T_{11}) = R1$, $ty(T_{12}) = R2$, $ty(T_{13}) = R3$, $ty(T_{21}) = ValidCancel$.

The history H is shown in Figure 8. Recall that $\mathbf{ST}$ represent the consistent database states (i.e. states satisfying the original invariants), and $\widehat{\mathbf{ST}}$ represent the states satisfying the relaxed invariants. The ring in the figure denotes the states $\widehat{\mathbf{ST}} \Leftrightarrow \mathbf{ST}$, that is the states that satisfy the modified invariants but do not satisfy the original invariants. Executing the first step of the reserve transaction, T_{11}, takes us to a state in $\widehat{\mathbf{ST}} \Leftrightarrow \mathbf{ST}$. After executing the only step of the Cancel transaction, T_{21}, we are still in one of the states in $\widehat{\mathbf{ST}} \Leftrightarrow \mathbf{ST}$. Note that such an execution is correct in our model. Since the last step of the Cancel transaction does not satisfy the integrity constraints of the database, this is not a correct execution according to the NT/PV model. □

However we do have other requirements for a correct execution. For example, we require the post condition of any step of any transaction to satisfy the modified invariants (which are the relaxed integrity constraints). For a correct execution of a set of transactions, we

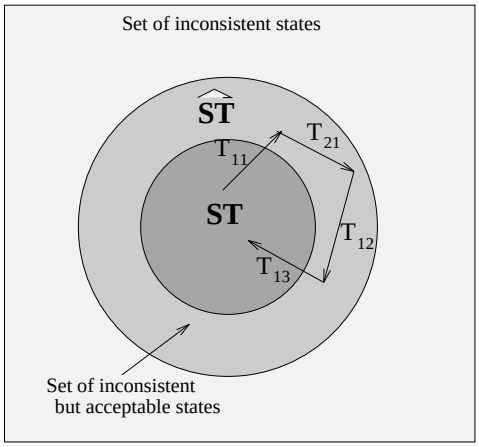


Figure 8: Example Execution Not Permitted by the NT/PV Model

also require that after the execution of all steps of all the transactions, the database be in a consistent state.

The mechanism proposed by the authors for generating NT/PV correct executions requires support for multiple versions. As our transaction model does not deal with multiple versions, and our notion of correct execution is different from that proposed by the authors in [KS94], our implementation differs from that in [KS94].

7.2 Farrag and Özsu

Our work is closely related to that of [FÖ89] and hence we discuss this work at a more detailed level. In [FÖ89] the authors introduce the notion of *breakpoint sets* associated with the steps of transactions. The breakpoint set of a step T_{ij}, of a transaction T_i, gives the types of transactions that may interleave transaction T_i after step T_{ij}. The burden of specifying breakpoint sets lies with the application developer and the authors assume that the developer will define a correct breakpoint set description. A schedule is *correct* if it is stepwise serial and for every transaction T_i that appears between steps T_{jk} and $T_{j(k+1)}$ of another transaction T_j, T_i is in the breakpoint set of the step T_{jk}. To identify a larger class of allowable schedules, called *relatively consistent schedules*, the authors propose a *precedence graph*. For a schedule to be relatively consistent, the precedence graph of the schedule must satisfy two criteria - (i) the precedence graph must be acyclic and, (ii) there must be a topological sort of the graph that gives a correct schedule. As noted, our model requires only the first constraint.

The authors describe a lock based mechanism that generates only relatively consistent

schedules. The mechanism uses four types of locks - relatively shared, relatively exclusive, shared and exclusive. Before reading or writing a data item, the appropriate lock (shared or exclusive) is acquired. The lock on an item can be acquired only if the transaction is in the breakpoint set of the most recently executed steps of the transactions holding relatively shared or relatively exclusive lock on the data item. After a step completes, all shared locks and exclusive locks acquired by this step are converted to relatively shared and relatively exclusive locks respectively. After a transaction terminates, a release graph is checked to see whether the locks of the transaction can be released. In addition to its inherent complexity, the mechanism requires breakpoint sets to satisfy an additional assumption that may be difficult to obtain in practice.

Our successor set notion is similar to the breakpoint sets of [FÖ89]. However, the semantics of the successor set differs from that of breakpoint sets. Th most significant difference is that successor sets are sets of types of *steps*, whereas breakpoint sets are sets of types of *transactions*. Successor sets have the flexibility to express that type of step T_{km} is in the successor set of type of step T_{ij} even though the type of some other step T_{kl} of T_k may not be in the successor set of type of step T_{ij}.

The mechanism we propose is simpler than that of [FÖ89]. We use only two kinds of locks - shared and exclusive as opposed to the four kinds of locks of [FÖ89]. Before reading or writing a data item, the appropriate lock (shared or exclusive) has to be acquired. A lock can be acquired only if the type of the current step is in the successor set of all the active transactions that had locked the data item. The lock release process is also very different. The locks acquired by a step are released when the step commits. In contrast, in [FÖ89] the earliest time when locks acquired by a step can be released is after the entire transaction terminates. Our mechanism can be adapted more easily to a standard two-phase locking environment.

Example

In this subsection we repeat the motivating example from Section 2 of [FÖ89]. It turns out that this example fails to satisfy Assumption 5.1 in [FÖ89], which is important because the mechanism given in [FÖ89] requires this assumption. Afterwards, we show that our model – and mechanism – can indeed handle this example.

The example in Section 2 of [FÖ89] involves three transaction types, RESERVE, CANCEL, and REPORT. We only show the details of the RESERVE and CANCEL transaction types.

RESERVE Transaction

Step 1: if Res < Total

then begin
Res ← Res + 1;
add new guest g_i to database;
end
else exit;
Step 2: Find a (free) room r_k with ST(r_k) = A;
Step 3: RM(g_i) ← r_k;
ST(r_k) ← U;

CANCEL Transaction
Step 1: if RM(g_i) = r_k
then begin
ST(r_k) ← A;
Res ← Res - 1;
remove guest g_i from database
end
else exit;

The prohibition on interleaving is that a RESERVE transaction may not begin at the second breakpoint of another RESERVE transaction, or else the same room might be assigned to multiple guests. The interleaving is formalized as follows. Let T_i be a RESERVE transaction. Let B_{i1}, B_{i2}, B_{i3} be the breakpoints associated with the three steps of the transaction T_i. Let T_k be a CANCEL transaction. Let B_{k1} be the breakpoint associated with the single step of the CANCEL transaction T_k. The notation $ts(B_x)$ denotes the set of transaction types that may interleave at breakpoint B_x. The allowed interleavings are describe by:

RESERVE $\in ts(B_{i1})$,
RESERVE $\notin ts(B_{i2})$,
RESERVE $\in ts(B_{i3})$,
RESERVE $\in ts(B_{k1})$,
CANCEL $\in ts(B_{i1})$,
CANCEL $\in ts(B_{i2})$,
CANCEL $\in ts(B_{i3})$.

The Farrag and Özsu mechanism requires the following assumption [FÖ89, Assumption 5.1]:

> Suppose that transaction T_j is allowed to interleave at a breakpoint B_{ip} of transaction T_i. Then any other transaction that can interleave at any breakpoint

in T_j can also interleave at the breakpoint B_{ip}. (Stated formally: If $ty(T_j) \in ts(B_{ip})$ and $ty(T_k) \in ts(B_{jq})$ then $ty(T_k) \in ts(B_{ip})$.)

The above assumption is very strong, and the specifications of the breakpoint sets in the motivating example of [FÖ89] fail to satisfy it. Specifically, since a CANCEL transaction can interleave at the breakpoint B_{i2}, and since RESERVE is in the breakpoint B_{k1} of the CANCEL transaction, RESERVE must be in the breakpoint set of B_{i2}, which it is not.

Implementing the Example by Our Model

As the semantics of successor sets are different from those of breakpoint sets, it is not possible to directly map breakpoint sets into successor sets. In this subsection we give a possible successor set specification which conforms to our model, and also achieves the restrictions on the interleavings from the example given above.

Let $RES1$, $RES2$, $RES3$ be the types of the three steps of the RESERVE transaction. Let $CANCEL$ be the type of the single step in CANCEL. As above, we ignore the REPORT transaction.

A reasonable successor set description is given below. Our mechanism easily implements this description.

$SS(RES1) = \{RES1, RES2, RES3, CANCEL\}$
$SS(RES2) = \{RES3, CANCEL\}$
$SS(RES3) = \{RES1, RES2, RES3, CANCEL\}$
$SS(CANCEL) = \{RES1, RES2, RES3, CANCEL\}$

8 Conclusion

In this paper, 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 implementation that ensures execution histories that are stepwise conflict-serializable and also respect the successor set constraints. The implementation is substantially simpler than the corresponding implementation in [FÖ89].

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, which means that all integrity constraints relevant to the calculation of any output will have to be included as explicit preconditions for the transaction (see section 3.4). 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 somewhat problematic since deletion may impact the complete execution property, which ensures that any transaction that has been partially executed can eventually complete.

References

[AAS93] 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.

[AJR95] Paul Ammann, Sushil Jajodia, and Indrakshi Ray. Using formal methods to reason about semantics-based decomposition of transactions. In *VLDB '95: Proceedings of the Twenty-First International Conference On Very Large Data Bases*, Zurich, Switzerland, September 1995. To appear.

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

[BR92] B.R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, March 1992.

[CR94] P. K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, September 1994.

[DE90] W. Du and A.K. Elmagarmid. Quasi serializability: A correctness criterion for global concurrency control in interbase. In *Proc.* 16^{th} *VLDB*, pages 347–355, 1990.

[FÖ89] 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.

[GM83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[Her87] M. Herlihy. Extending multiversion time-stamping protocols to exploit type information. *IEEE Transactions on Computers*, 36(4):443–448, April 1987.

[HW91] 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.

[JM87] S. Jajodia and C. Meadows. Managing a replicated file in an unreliable network. In *Proceedings of 3rd IEEE International Conference on Data Engineering*, pages 396–404, Los Angeles, CA, February 1987.

[KS88] H. F. Korth and G. D. Speegle. Formal model of correctness without serializability. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 379–386, June 1988.

[KS94] H. F. Korth and G. Speegle. Formal aspects of concurrency control in long-ouration transaction systems using the NT/PV model. *ACM Transactions on Database Systems*, 19(3):492–535, September 1994.

[LMWF94] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[Lyn83] Nancy A. Lynch. Multilevel atomicity—A new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[OG76] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[PST91] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice-Hall, New York, 1991.

[SLJ88] L. Sha, J. P. Lehoczky, and E.D. Jensen. Modular concurrency control and failure recovery. *IEEE Transactions on Computers*, 37(2):146–159, February 1988.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, New York, 1989.

[SSVR92] D. Shasha, E. Simon, P. Valduriez, and P. Rodin. Simple rational guidance for chopping up transactions. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 298–307, San Diego, CA, June 1992.

[Wei84] W. E. Weihl. *Specification and Implementation of Atomic Data Types*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1984.

[Wei88] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

Appendix A – Properties of the Decomposition

We show that the decomposition as given in Section 4.5 has the necessary properties.

The Composition Property

In the Hotel Database, the original invariants I are satisfied when the auxiliary variables *tempreserved* and *tempassigned* satisfy the following conditions:

$$tempreserved = 0 \land tempassigned = \varnothing$$

The reserve transaction is the only transaction in the Hotel Database that has been decomposed into multiple steps. For the reserve transaction the proof obligation is :

$$Reserve \Leftrightarrow (R1 \fatsemi R2 \fatsemi R3) \land tempreserved = 0 \land tempassigned = \varnothing \qquad \ldots(i)$$

The right hand side of the equivalence ($\Leftrightarrow$) in the expression (i) obtained by schema composition [PST91] evaluates to:

$(R1 \fatsemi R2 \fatsemi R3) \land tempreserved = 0 \land tempassigned = \varnothing$

$\Delta ValidHotel$
$g? : Guest$
$r! : Room$

$res < total$
$g? \notin guest$
$ST(r!) = Available$
$res' = res + 1$
$guest' = guest \cup \{g?\}$
$ST' = ST \oplus \{r! \mapsto Unavailable\}$
$RM' = RM \cup \{g? \mapsto r!\}$
$tempreserved' = 0$
$tempassigned' = \varnothing$

The schema for the original *Reserve* transaction is given by:

Reserve

$\Delta Hotel$
$g? : Guest$
$r! : Room$

$res < total$
$g? \notin guest$
$ST(r!) = Available$
$res' = res + 1$
$guest' = guest \cup \{g?\}$
$ST' = ST \oplus \{r! \mapsto Unavailable\}$
$RM' = RM \cup \{g? \mapsto r!\}$

The constraints on $tempreserved$, $tempreserved'$, $tempassigned$, and $tempassigned'$ are implied by the invariants in $\Delta Hotel$. The converse is also true. Thus the predicate parts of the schemas are equivalent, and hence the schemas are equivalent.

The Sensitive Transaction Isolation Property

In the hotel database we have two sensitive transactions which are of type *Report* and *Reserve*. In each case, the proof of the Sensitive Transaction Isolation Property is by construction.

First we consider transactions of type *Report*. We compute the subset of the original invariants that must be satisfied as a precondition for *Report*. We obtain these from *Hotel* by hiding the state variables *not* involved in producing the outputs of *Report*. The state variables not involved in producing the output of *Report* are *res* and *guest*. Hiding the variables *res* and *guest* from *Hotel* produces the following schema:

$Hotel \setminus (res, guest)$

$RM : Guest \rightarrowtail Room$
$ST : Room \twoheadrightarrow Status$

$\exists\, res : \mathbb{N};\ guest : \mathbb{P}\, Guest \bullet$
$\#RM = res$
$\mathrm{dom}(ST \rhd \{Unavailable\}) = \mathrm{ran}\, RM$

The constraint simplifies to

$$\mathrm{dom}(ST \rhd \{Unavailable\}) = \mathrm{ran}\, RM \qquad \cdots (i)$$

The constraint is implied by an invariant in $\hat{I}$, namely,

$$\mathrm{dom}(ST \rhd \{Unavailable\}) = \mathrm{ran}\, RM \cup tempassigned$$

and $tempassigned = \varnothing$. Hence we include $tempassigned = \varnothing$ as a precondition for *ValidReport*.

Next we consider the transactions of type *Reserve*. The output of a *Reserve* transaction is the room $r!$ assigned to the guest. The only constraint on $r!$ is that the function ST evaluated at $r!$ be $Available$. Therefore, to compute the subset of the original invariants that must hold as a precondition on *Reserve*, we hide all the state variables except ST in *Hotel*. The schema obtained by hiding all the variables but ST is as follows:

$Hotel \setminus (res, guest, RM)$

$$RM : Guest \rightarrowtail Room$$
$$ST : Room \nrightarrow Status$$

$$\exists\, res : \mathbb{N};\ guest : \mathbb{P}\, Guest;\ RM : Guest \rightarrowtail Room \bullet$$
$$\#RM = res$$
$$\mathrm{dom}(ST \rhd \{Unavailable\}) = \mathrm{ran}\, RM$$

The constraint simplifies to **true**, which means that no additional preconditions need be placed in the steps $R1$, $R2$, or $R3$ of the reserve transaction.

The Consistent Execution Property

For the hotel database system, the original invariants in I are satisfied when $tempassigned = \varnothing$ and $tempreserved = 0$. Thus we have to prove that if the initial state of a complete semantic history satisfies $tempassigned = \varnothing$ and $tempreserved = 0$, the final state of the history will also satisfy $tempassigned = \varnothing$ and $tempreserved = 0$.

Let n_1, n_2 and n_3 be the number of steps of type $R1$, $R2$ and $R3$ respectively present in any complete history. The variable *tempreserved* is modified by steps of type $R1$ and $R3$ of a reserve transaction. *tempreserved* is incremented in steps of type $R1$ and is decremented in step of type $R3$. Thus *tempreserved* is given by the following expression

$$tempreserved = n_1 \Leftrightarrow n_3$$

The variable *tempassigned* is modified in steps of type $R2$ and $R3$. In step of type $R2$ a room is added to the set *tempassigned* and in step of type $R3$ of the reserve transaction the room is taken out from the set *tempassigned*. Thus we can write,

$$\mid tempassigned \mid = n_2 \Leftrightarrow n_3$$

In a complete history since all the reserve transactions have completed, the number of steps of type $R1$, $R2$ and $R3$ that have been executed are equal, and so

$$n_1 = n_2 = n_3$$

Therefore, in a complete history,

1. $tempreserved = 0$ and
2. $\mid tempassigned \mid = 0$ or $tempassigned = \varnothing$.

Hence we can conclude that the hotel database has the consistent execution property.

The Complete Execution Property

To prove that the Hotel Database System has the complete execution property, we take any partial correct semantic history and show that it is the prefix of some complete correct semantic history.

In the Hotel Database, only the reserve transactions have been broken into steps. So in any partial semantic history the only incomplete transactions are reserve transactions. Suppose we have n incomplete reserve transactions in the correct partial semantic history. We show how an incomplete reserve transaction can be completed, thereby decreasing the number of incomplete reserve transactions from n to $n \Leftrightarrow 1$. By repeated applications of our argument, it is possible to reduce the number of incomplete reserve transactions to zero, at which point the history is a complete correct semantic history.

Consider an incomplete reserve transaction. If the reserve transaction has completed a step of type $R1$, the preconditions of the next step, which is of type $R2$, is always satisfied and so the next step can be executed. If the reserve transaction has completed step of type $R2$, the precondition of the next step, which is of type $R3$, $(g? \notin guest)$ may or may not be satisfied. If the precondition of the step of type $R3$ is not satisfied, the precondition of another transaction, which is of type *ValidCancel* $(g? \in guest)$, is satisfied. Moreover the postcondition of step of type *ValidCancel* establishes the precondition of step of type $R3$, and so the step of type $R3$ can complete. Thus all reserve transactions can complete.

Appendix B – Properties of the Histories Generated using Successor Set Mechanism

The Valid Successor Set Property

Part 1

In the first part, we prove that the set of correct semantic histories generated using the successor set mechanism is a subset of the set of correct histories generated using invariants and precondition checks.

Let

$\mathbf{H_1}$ = set of correct semantic histories generated by decomposition of transactions and

$\mathbf{H_2}$ = set of correct semantic histories generated by the queuing and successor set mechanism.
The proof obligation is $\mathbf{H_2} \subseteq \mathbf{H_1}$, i.e. we have to show that for any correct semantic history in $\mathbf{H_2}$, there is a corresponding correct semantic history in $\mathbf{H_1}$.

Let H_2 be any correct semantic history generated using the queuing and successor set mechanism. From H_2 we construct a semantic history H_1 as follows:

1. make the initial state of H_1 the same as that of H_2,
2. for any step in H_2, include the same step in H_1,
3. add the precondition $tempassigned = \emptyset$ to any and all occurrence(s) of steps of type *ValidReport* in H_1.

Clearly H_1 is the history corresponding to H_2, which uses precondition checks to control the ordering instead of successor set mechanism as done in H_2. It remains to be shown that $H_1 \in \mathbf{H_1}$, i.e. the semantic history H_1 is a correct semantic history. Note that the only way in which history H_1 differs from H_2 is that, some preconditions present in the steps of type *ValidReport* in H_1 are not present in the corresponding steps in H_2.

Let us consider the precondition ($tempassigned = \emptyset$) that was added to *ValidReport* in H_1. From the specifications of $R1$, $R2$, $R3$, *ValidCancel*, and *ValidReport* we see that the only situation in which the precondition of a step of type *ValidReport* is not satisfied, is when at least one step of type $R2$ has been executed but the next step, which is of type $R3$, has not yet been executed. However, as $ValidReport \notin SC(R2)$, a step of type *ValidReport* will never be allowed to execute between a step of type $R2$ and a step of type $R3$. Thus in H_2 $tempassigned = \emptyset$ is always satisfied before *ValidReport* is executed.

Note that the set of operations that modify the database state are the same for steps in H_1 and H_2. The initial states are the same for both the histories. So the state obtained by applying a step in H_1 is the same as that obtained by applying H_2.

Summing up, for the semantic history H_1, the following observations can be made.

1. The initial state is in $\mathbf{ST}$ because it is the same as that of the correct semantic history H_2.
2. The state obtained by applying each step of the history is in $\widehat{\mathbf{ST}}$ because it is the same as applying the corresponding step in the correct semantic history H_2.
3. Preconditions are satisfied for each step.

Thus the history H_1 is a correct semantic history, i.e. $H_1 \in \mathbf{H_2}$. Therefore for any history $H_2 \in \mathbf{H_2}$, we have a corresponding $H_1 \in \mathbf{H_1}$.

For the Hotel Database example, we can also show that corresponding to any history in $\mathbf{H_1}$ we can generate a corresponding history in $\mathbf{H_2}$ using the successor set and queuing mechanisms. In any correct history in $\mathbf{H_1}$, the preconditions are satisfied at each step and a step of type *ValidReport* can never appear between steps of type $R2$ and $R3$. The successor set mechanism ensures that a step of type *ValidReport* does not occur between steps of type $R2$ and an $R3$ and does not impose any further restrictions. Thus for any history in $\mathbf{H_1}$, we have a corresponding history in $\mathbf{H_2}$.

Thus for the hotel database example, the set of histories in $\mathbf{H_1}$ is equivalent to the set of histories in $\mathbf{H_2}$.

Part 2

In the second part of the valid successor set property, it is required to show that all partially correct semantic histories will eventually complete. For the Hotel Database the set $\mathbf{H_1}$ is equivalent to the set $\mathbf{H_2}$. Since we have proved earlier that the set of histories in $\mathbf{H_1}$ has the complete execution property, it is implied that the set of histories in $\mathbf{H_2}$ has the complete execution property.