Applying Formal Methods to Semantic-Based Decomposition of Transactions

Paul Ammann and Sushil Jajodia and Indrakshi Ray
Center for Secure Information Systems
and Department of Information and Software Systems Engineering
George Mason University

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

Categories and Subject Descriptors: H.2.4 [**Database Management**]: System—*Transaction Processing*; D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Invariants*; *Pre- and post-conditions*; *Specification techniques*

General Terms: Database Management Systems,Formal Specifications

Additional Key Words and Phrases: Database Management Systems, Transaction Processing, Concurrency Control

1. INTRODUCTION

Performance requirements can force an application developer to decompose some transactions into smaller logical units or *steps*, especially for long-lived transactions.

The work of Paul Ammann was partially supported by National Science Foundation under grant number CCR-9202270.
The work of Sushil Jajodia was 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.
The work of Indrakshi Ray was partially supported by a George Mason University Graduate Research Fellowship Award.
Authors' Address: ISSE Department, Mail Stop 4A4, George Mason University, 4400 University Drive, Fairfax, VA 22030-4444. Email: {pammann,jajodia,indrakshi}@gmu.edu

Consider the simple example of making a hotel reservation. A reserve transaction might consist of ensuring that there are still rooms vacant, selecting a vacant room that matches the customer's preferences, and recording billing information. Since the reserve transaction might last a relatively long time – for example, when the customer makes reservations by phone – it may be desirable to execute the three steps of the reserve transaction separately. Researchers have long recognized the need to decompose transactions into steps. Our contribution is to formalize a decomposition process and identify desirable properties. We show how to assess these properties for a specific application by enumerating and discharging the proof obligations that accompany each phase of the decomposition. We describe an efficient, two-phase locking mechanism for the resulting, verified decomposition.

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

Decomposition of a transaction into steps sacrifices atomicity since the atomicity of the single logical action is lost. Interleaving steps of transactions exposes each to the partial effects of the others. Hence, if a transaction aborts after committing some of its steps, it may not be possible to remove all of its effects. This difficulty arises because transactions that read from the aborted transaction may have committed. In addition, the aborted transaction may have generated outputs. Thus traditional undo [Bernstein et al. 1987] is not possible; the solution is to semantically undo the actions of the aborted transactions. We achieve semantic undo with compensating steps [Garcia-Molina 1983; Garcia-Molina and Salem 1987].

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

The remainder of the paper is organized as follows. A motivating example is presented in Section 2. Section 3 gives our model and illustrates a decomposition of the example. Successor sets, which are crucial to efficiency of our decomposition, are presented in Section 4. A graph-based characterization of correctness is given in Section 5, followed by a mechanism in a two-phase locking environment in Section 6. Section 7 relates our work to the literature. Section 8 concludes the paper. Proofs of desirable properties for the example appear in the Appendix.

$\mathbb{N}$	Set of Natural Numbers
$\mathbb{P}\,A$	Powerset of Set A
$\#A$	Cardinality of Set A
$\setminus$	Set Difference (Also schema 'hiding')
$A \mathbin{⨾} B$	Forward Composition of A with B
$x \mapsto y$	Ordered Pair $(x,\ y)$
$A \nrightarrow B$	Partial Function from A to B
$A \rightarrowtail\!\!\!\!\!\!+\; B$	Partial Injective Function from A to B
$B \lhd A$	Relation A with Set B Removed from Domain
$A \rhd B$	Relation A with Range Restricted to Set B
$\mathrm{dom}\, A$	Domain of Relation A
$\mathrm{ran}\, A$	Range of Relation A
$A \oplus B$	Function A Overridden with Function B
$x?$	Variable $x?$ is an Input
$x!$	Variable $x!$ is an Output
x	State Variable x before an Operation
x'	State Variable x' after an Operation
ΔA	Before and After State of Schema A
ΞA	ΔA with No Change to State

Table I. Relevant Z Notation

We adopt the Z specification language [Spivey 1992] 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 I 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 appears in figure 1. The hotel database has a set of objects, two integrity 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 the number of rooms in the hotel.

In Z states, as well as operations, are described with a two-dimensional notation called a *schema*, in which declarations for the objects appear in the top part and constraints on the objects appear in the bottom part. The schema *Hotel*, which defines the state of the database, lists the objects in the hotel. The natural number *res* counts current reservations, and the partial injection *RM* relates guests to rooms. Our particular example does not allow guests to register multiple times, which is reflected in the fact that *RM* is an injective function. The example could be modified easily with different constraints. The partial function *ST* records the status of each room. Additional integrity constraints on the objects in hotel database appear in the bottom part of *Hotel*. There are two such constraints:

(1) $\#RM = res$. The number of guests who have been assigned rooms (the size of the *RM* function) equals the total number of reservations (*res*).

$$[\mathit{Guest}, \mathit{Room}]$$

$$\mathit{Status} ::= \mathit{Available} \mid \mathit{Taken}$$

$$\mathit{total} : \mathbb{N}$$

Hotel

$$\begin{array}{l} \mathit{res} : \mathbb{N} \\ \mathit{ST} : \mathit{Room} \rightarrow\!\!\!\!\!+\; \mathit{Status} \\ \mathit{RM} : \mathit{Guest} \rightarrowtail\!\!\!\!\!+\; \mathit{Room} \\ \hline \#\mathit{RM} = \mathit{res} \\ \mathrm{dom}(\mathit{ST} \rhd \{\mathit{Taken}\}) = \mathrm{ran}\, \mathit{RM} \end{array}$$

Reserve

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

Cancel

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

Report

$$\begin{array}{l} \Xi \mathit{Hotel} \\ \mathit{currentST}! : \mathit{Room} \rightarrow\!\!\!\!\!+\; \mathit{Status} \\ \mathit{currentRM}! : \mathit{Guest} \rightarrowtail\!\!\!\!\!+\; \mathit{Room} \\ \hline \mathit{currentST}! = \mathit{ST} \\ \mathit{currentRM}! = \mathit{RM} \end{array}$$

Fig. 1. Initial Specification of the Hotel Database

(2) $\mathrm{dom}(\mathit{ST} \rhd \{\mathit{Taken}\}) = \mathrm{ran}\, \mathit{RM}$. The set of rooms that are taken ($\mathrm{dom}(\mathit{ST} \rhd \{\mathit{Taken}\})$) is exactly the set of rooms reserved by guests ($\mathrm{ran}\, \mathit{RM}$).

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

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

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

NaiveR1

$\Delta Hotel$

$res < total$
$res' = res + 1$
$ST' = ST$
$RM' = RM$

NaiveR2

$\Delta Hotel$
$r! : Room$

$ST(r!) = Available$
$ST' = ST \oplus \{r! \mapsto Taken\}$
$res' = res$
$RM' = RM$

NaiveR3

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

$g? \notin \mathrm{dom}\, RM$
$RM' = RM \cup \{g? \mapsto r!\}$
$res' = res$
$ST' = ST$

Fig. 2. A Naive Decomposition

the given invariants.

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

Instead of executing a transaction as an atomic unit, we wish to break a transaction into steps and execute each of these steps as an atomic unit. A *decomposition* of a transaction is a sequence of *steps*. In place of the transaction, the steps execute atomically in order. A transaction that has been decomposed into two or more steps is referred to as a *multistep* transaction. In keeping with the terms used in formal methods, we say that a decomposition is a *refinement* of the original specification, with the understanding that the various properties described subsequently are satisfied. We also use the term refinement to describe other steps towards implementation. For example, a successor set description, defined in section 4, is a refinement of a decomposition.

One possible approach to decomposition is to treat the steps as transactions. In particular, one could insist that the integrity constraints hold after each step, which is the decision taken in the Saga model [Garcia-Molina and Salem 1987]. As the naive decomposition below demonstrates, such a requirement is too strong for some applications, and so we develop a more flexible approach.

3.1 A Naive Decomposition of the Reserve Transaction

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

Step 1:. Increment the number of reserved rooms.

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

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

From a formal perspective, the naive decomposition has a fatal flaw, in that none of the proposed steps, considered by itself, maintains the invariants in *Hotel*. For example, $NaiveR1$ does not maintain the invariant $\#RM = res$ since $NaiveR1$ increments the value of *res*, but does not alter *RM*. Formally, the computed preconditions of all three steps simplify to false, indicating that none of the steps can be safely executed in an implementation. Executing any of the proposed steps violates the invariants, and other transactions are exposed to the inconsistent state. For example, *Report* produces an inconsistent output if executed in a state in which the second invariant does not hold.

3.2 Generalizing the Original Invariants

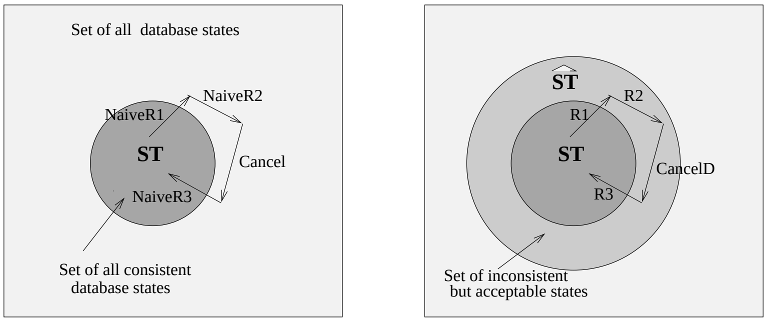
The example demonstrates that some decompositions are unacceptable. Specifically, a decomposition may yield steps that leave the database in a state in which the invariants do not hold. The arrow labeled *NaiveR1* in figure 3(a) illustrates this possibility. Once the invariants are violated, the formal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

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

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

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

In our approach, we define a new set of invariants $\hat{I}$ by relaxing the original invariants I. We decompose each transaction such that execution of any step results in a state that satisfies $\hat{I}$. Let $\widehat{\mathbf{ST}} = \{ \text{ST} : \text{ST satisfies } \hat{I}\}$. 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



(a) Standard classification of database states

(b) Database states as classified in our model

Fig. 3. Classification of the Database States

investigate activities in $\widehat{\mathbf{ST}}$.

To reason about decomposing transactions into steps and to avoid the problems of a naive decomposition, we use *auxiliary variables* to generalize the invariants. Auxiliary variables are a standard method of reasoning about concurrent executions [Owicki and Gries 1976] and, in particular, have been applied to the problem of semantic-based concurrency control [Garcia-Molina 1983, Appendix C]. Our work focuses more on the decomposition than does [Garcia-Molina 1983], and so we emphasize the role of auxiliary variables more strongly. Before we generalize the invariants in the hotel example, we present two properties that a decomposition must possess. We stress that the auxiliary variables are introduced for purposes of analysis; the goal is to eliminate such variables from an implementation.

3.3 Composition Property

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

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

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

$$I \wedge (S_{i1} \,{}_{9}^{\circ} \ldots {}_{9}^{\circ}\, S_{in}) \setminus (aux, aux') \Leftrightarrow T_i$$

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

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

From an implementation perspective, the composition property is similar to requiring that the sequential execution of the steps be view equivalent to that of the original transaction. One complicating factor is that the specification of the transaction, and hence of the steps, may be nondeterministic. In addition, the auxiliary variables introduced by the decomposition do not appear in the original database; compare *Hotel* in figure 1 with *HotelD* in figure 4.

3.4 Sensitive Transaction Isolation Property

In our model, we allow transactions to access database states that do not satisfy the original invariants (that is, states in $\widehat{\mathbf{ST}} - \mathbf{ST}$). But we may wish to keep some transactions from viewing any inconsistency with respect to the original invariants. For example, some transactions may output data to users; these transactions are called *sensitive* transactions [Garcia-Molina 1983]. We require sensitive transactions to appear to have generated outputs from a consistent state.

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

In our model, we ensure the sensitive transaction isolation property by construction. There are two aspects to such a construction. First, for each sensitive transaction, we compute the subset of the original integrity constraints, I, relevant to the calculation of any outputs. This subset of I must be implied by the preconditions of the step or steps that generate the outputs. Second, as pointed out by Rastogi, Korth, and Silberschatz [Rastogi et al. 1995], if outputs are generated by multiple steps, interleavings between these steps must be controlled to ensure that outputs from later steps are consistent with outputs from earlier steps.

3.5 A Valid Decomposition

In this section, we provide a decomposition of the hotel database. The problems noted in the previous decomposition are avoided, and the two properties identified so far hold. After presenting the example, we derive additional properties for decompositions. The example is valid in the sense that it satisfies all of the properties that we require for a decomposition.

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

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{HotelD}} \\
\hline
res : \mathbb{N} \\
ST : Room \pfun Status \\
RM : Guest \pinj Room \\
underway : \mathbb{N} \\
acquired : \mathbb{P}\, Room \\
\hline
\#RM + underway = res \\
\mathrm{dom}(ST \rres \{Taken\}) = \mathrm{ran}\, RM \\
\qquad\qquad\qquad \cup\ acquired \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{R1}} \\
\hline
\Delta HotelD \\
\hline
res < total \\
res' = res + 1 \\
underway' = underway + 1 \\
ST' = ST \\
RM' = RM \\
acquired' = acquired \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{R2}} \\
\hline
\Delta HotelD \\
r! : Room \\
\hline
ST(r!) = Available \\
ST' = ST \oplus \{r! \mapsto Taken\} \\
acquired' = acquired \cup \{r!\} \\
res' = res \\
underway' = underway \\
RM' = RM \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{R3}} \\
\hline
\Delta HotelD \\
g? : Guest \\
r! : Room \\
\hline
g? \notin \mathrm{dom}\, RM \\
RM' = RM \cup \{g? \mapsto r!\} \\
acquired' = acquired \setminus \{r!\} \\
underway' = underway - 1 \\
res' = res \\
ST' = ST \\
\hline
\end{array}
$$

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

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ReportD}} \\
\hline
\Xi HotelD \\
currentST! : Room \pfun Status \\
currentRM! : Guest \pinj Room \\
\hline
acquired = \varnothing \\
currentST! = ST \\
currentRM! = RM \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CompR2}} \\
\hline
\Delta HotelD \\
\hline
res > 0 \\
underway > 0 \\
res' = res - 1 \\
underway' = underway - 1 \\
ST' = ST \\
RM' = RM \\
acquired' = acquired \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CompR3}} \\
\hline
\Delta HotelD;\ r! : Room \\
\hline
res > 0 \\
underway > 0 \\
r! \in acquired \\
res' = res - 1 \\
underway' = underway - 1 \\
acquired' = acquired \setminus \{r!\} \\
ST' = ST \oplus \{r! \mapsto Available\} \\
RM' = RM \\
\hline
\end{array}
$$

Fig. 4. A Valid Decomposition for the Hotel Database

The single step decomposition *CancelD* is nearly identical to the transaction *Cancel*, except that the auxiliary variables *acquired* and *underway* are specified as remaining unchanged.

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

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

which can be derived from the generalized invariant

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

if the auxiliary variable *acquired* satisfies $\mathit{acquired} = \varnothing$. Hence, this predicate is a precondition in the single step decomposition *ReportD*.

3.6 Compensating Steps

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

Suppose a *Reserve* transaction aborts after step $R1$ commits. Some mechanism must undo the effects of $R1$. Nullifying the effects of $R1$ using the backwards recovery method of traditional *undo* [Bernstein et al. 1987], where the state that existed before $R1$ is physically restored, is not possible because steps of other transactions may have read the updates of $R1$. Instead we adopt the forward recovery solution of a compensating step [Garcia-Molina 1983; Garcia-Molina and Salem 1987], which semantically undoes the effects of the committed step $R1$ and does not disturb transactions that may have read from $R1$.

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

For a transaction T_i that has been decomposed into n steps $S_{i1}, \ldots, S_{in}$ we specify $n-1$ compensating steps denoted by $C_{i2}, \ldots, C_{in}$. The compensating step C_{ij} semantically undoes the cumulative effect of steps $S_{i1},\ldots,S_{i(j-1)}$. This is in contrast to the approach used in [Garcia-Molina 1983; Garcia-Molina and Salem 1987] where a compensating step C_{ij} is used to semantically undo the operations performed by a single step S_{ij}. The difference between the approaches is not significant; our choice simplifies our presentation. In the *Hotel* example, we specify compensating steps *CompR*2 and *CompR*3 for the *Reserve* transaction. Figure 4 includes the specification of *CompR*2 and *CompR*3.

3.7 Semantic Histories

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

In the previous subsections we showed how a *Reserve* transaction can be decomposed into steps $R1$, $R2$ and $R3$. We also showed the compensating steps $CompR2$ and $CompR3$ associated with the *Reserve* transaction. Before we proceed further, we make a distinction between a type of a step and an instance of a step. $R1$, $R2$, $R3$, *CancelD*, *ReportD*, $CompR2$ and $CompR3$, represent the different *types* of steps and compensating steps in the Hotel Database.

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

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

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

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

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

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

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

The sequences $S_{i1}, \ldots, S_{in}$ and $S_{i1}, \ldots, S_{ij}, C_{i(j+1)}$ are examples of *execution sequences* [Garcia-Molina 1983] of transaction T_i. The sequence $S_{i1}, \ldots, S_{in}$ is a *successful* execution sequence of T_i, and the sequence $S_{i1}, \ldots, S_{ij}, C_{i(j+1)}$ is an *unsuccessful* execution sequence of T_i.

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

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

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

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

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

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

3.8 Consistent Execution Property

To develop a notion of consistent executions, we define what it means for a semantic history to be correct.

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

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

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

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

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

Consistent Execution Property All histories generated by an application are correct semantic histories.

3.9 Successful Execution Property

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

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

In the hotel database suppose we have a correct semantic history $H_p = < S_{11}, S_{12} >$ defined over $\{T_1\}$, where $ty(S_{11}) = R1$ and $ty(S_{12}) = R2$. Let the reserve transaction T_1 execute with $g? =$ John. Consider step S_{13} where $ty(S_{13}) = R3$. The precondition John $\notin \operatorname{dom} RM$ of S_{13} requires that John not have an existing reservation, but it is possible that in the final state in H_p, John already is a guest. The correct semantic history H_p cannot be extended to a correct complete semantic history which contains the successful execution sequence of T_1. Note that if we instead define H_p over $\mathbf{T} = \{T_1, T_2\}$, where T_2 cancels John's existing reservation, then H_p can be extended to a complete history $H = < S_{11}, S_{12}, S_{21}, S_{13} >$ over $\mathbf{T}$, where $ty(S_{21}) = CancelD$. The first example, where $\mathbf{T} = \{T_1\}$, shows that the hotel specification does *not* enjoy the successful execution property.

Unlike the other properties we have stated so far, successful execution is an optional property. Successful execution property requires that all the preconditions of a transaction should appear in the first step. If the hotel example were to have the successful execution property, the precondition $g? \notin \operatorname{dom} RM$ must appear in step $R1$ and not in step $R3$. This in turn would require an update of the variable RM in the very first step. Thus insisting on successful execution property may force too many operations in the first step of the transaction - which is undesirable from the performance point of view. Hence we do not insist that applications have the successful execution property.

3.10 Compensation Property

A transaction may not always complete successfully; it may abort. When a multistep transaction aborts after committing one or more steps, the effects of the committed steps must be semantically undone, which is accomplished by executing the compensating step. The *compensation property* ensures that it will be possible to execute the compensating step at some point. Unlike the successful execution property, which is optional, the compensation property is required.

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

4. SUCCESSOR SETS

The decomposition process introduces additional database objects and additional preconditions on steps. The additional objects and preconditions are present primarily 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.

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

At this point, the notion of successor sets is purely syntactic. Subsequently, we define the constraints under which a successor set description is correct with respect to a particular decomposition; the correctness characterization is specified in terms of semantic histories.

We need to ensure that in 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, but it is not always possible to determine statically how the precondition being removed will actually evaluate during execution. 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. In the executable code we may use a separate object for each room. Thus, updates to RM always conflict at the specification level, but only conflict at the code level if the updates are to the same specific room.

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 are 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 conservatively 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. To achieve this goal we introduce the notion of conflict into our model. Two operations *conflict* if both operate on the same data item and at least one is a Write. Two steps T_{ij} and T_{pq} *conflict* if they contain conflicting operations.

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

(1) 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 $x' = x$.

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

Table II. Read and Write Sets for Steps of Hotel Example

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

Table III. Conflicting Steps for Hotel Example

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

Armed with a notion of conflict that is applicable for both specifications and implementations, we define a history by using conflict in a manner similar to [Bernstein et al. 1987].

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

(1) $H = \cup_{i=1}^{m} \cup_{j=1}^{n_i} T_{ij}$;
(2) $\prec_H \supseteq \cup_{i=1}^{m} \cup_{j=1}^{n_i} \prec_{ij}$; and
(3) for any two conflicting operations $p, q \in H$, either $p \prec_H q$ or $q \prec_H p$.

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

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

Definition 9. [**Correct Successor Set History**] Let H be a correct semantic history. If T_i is incomplete in the prefix of H that ends at T_{pq}, and T_{ij} is the last step in T_i such that

(1) T_{ij} conflicts with T_{pq} and

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

Table IV. Successor Sets for the Hotel Example

(2) T_{ij} precedes T_{pq} in H

then $ty(T_{pq}) \in SS(ty(T_{ij}))$.

Example 3. In the hotel example, the only precondition with auxiliary variables is $acquired = \varnothing$ in step type $ReportD$. This precondition is satisfied as long as a step of type $ReportD$ does not appear between a step of type $R2$ and a step of type $R3$ (or $CompR3$) of reserve transaction. We specify the successor sets as shown in Table IV.

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, in which case the referenced auxiliary variables must indeed be implemented.

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 10. [**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 compensation property.

The second condition can be easily satisfied by ensuring that all compensating steps are contained in each successor set description. 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.

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

5. CORRECT STEPWISE SERIALIZABLE HISTORIES

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

We develop stepwise serializability by defining equivalence in a standard way [Bernstein et al. 1987]. Two histories H and H' are said to be *equivalent* if they are defined over the same set of steps, they have the same operations, and they order conflicting operations of steps in the same way. That is, for any conflicting operations p_{ij} and q_{kl} belonging to T_{ij} and T_{kl} (respectively), if $p_{ij} \prec_H q_{kl}$, then $p_{ij} \prec_{H'} q_{kl}$. A *correct stepwise serializable history* is one which is equivalent to a correct successor set history. Before we build a graph model to reason about stepwise serializability of histories we introduce a helpful function.

5.1 The Next Function

Consider the following scenario. Suppose that in a partial correct successor set history step T_{ij} of transaction T_i has been executed, and suppose we wish to execute step T_{pq}, where T_{pq} conflicts with T_{ij} or some step previous to T_{ij}. The *next* function, defined below, gives the earliest step in T_i after T_{ij} where T_{pq} is allowed to execute. The role of the next function in our model is similar to that of the F function in [Farrag and Özsu 1989].

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

(1) $ty(T_{pq}) \in SS(ty(T_{ik}))$, $k \geq j$ and

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

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

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

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

T_{ij} introduces some inconsistency that prevents T_{pq} from being executed. This inconsistency is resolved in step T_{ik}. If T_{ik} alters the database state in such a way that T_{pq} can execute, then T_{pq} and T_{ik} must conflict. We capture this property as an explicit constraint in our model. The specifier of a decomposition is obligated to verify the constraint against a successor set description. The constraint lets us show a stronger relationship between acyclic precedence graphs and correct stepwise serializable histories than was possible in [Farrag and Özsu 1989].

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

5.2 Acyclic Graph Model

In this section we show how to decide whether a history is a correct stepwise serializable history by analyzing a graph, called a precedence graph, derived from that history. Our precedence graph is related to the one developed in [Farrag and Özsu 1989].

Definition 12. **[Precedence Graph]** Let H be a history defined over a set of transactions $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$. The *precedence graph* of H, denoted by $PG(H)$, is a directed graph where the nodes are the steps of the transactions in $\mathbf{T}$ and the edges are of three types constructed as follows:

(1) Internal Edges – For each pair of consecutive steps S_{ij} and $T_{i(j+1)}$ of transaction T_i, there is an *internal* or I edge $(S_{ij}, T_{i(j+1)})$. (Note that S_{ij} is a step of T_i; $T_{i(j+1)}$ is a step or compensating step of T_i.)

(2) Conflict Edges – For each pair of conflicting steps T_{ij} and T_{pq} belonging to different transactions T_i and T_p, there is a *conflict* or C edge (T_{ij}, T_{pq}) if there is an operation in T_{ij} that conflicts with and precedes another operation in T_{pq}.

(3) Successor Edges – For each pair of steps T_{ij} and T_{pq}, belonging to different transactions T_i and T_p respectively, such that there is a conflict edge from T_{ij} to T_{pq}, there is a successor edge $(NF(T_{ij}, T_{pq}), T_{pq})$.

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

Example 4. Consider the following history H:

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

where $ty(S_{11}) = R1$, $ty(S_{12}) = R2$, $ty(S_{13}) = R3$, and $ty(S_{21}) = CancelD$. Successor set descriptions are given in Table IV. Conflicting steps in the Hotel Database are given in Table III. The precedence graph is shown in the left of figure 5. The

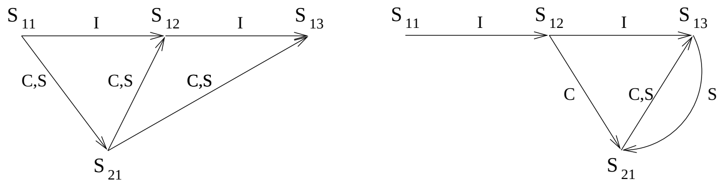


Fig. 5. Precedence Graph for Histories in Examples 4 and 5

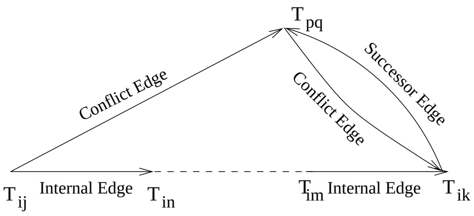


Fig. 6. Edges of Precedence Graph involving T_{ij}, T_{ik}, T_{pq}

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 5. Consider the following history H:

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

where $ty(S_{11}) = R1$, $ty(S_{12}) = R2$, $ty(S_{13}) = R3$, and $ty(S_{21}) = ReportD$. The precedence graph is shown in the right of figure 5. 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} and in $PG(H)$ there must be a successor edge (T_{ik}, T_{pq}) corresponding to the conflict edge (T_{ij}, T_{pq}). By Constraint 1 T_{pq} and T_{ik} conflict. As T_{ij} is the last operation which conflicts and precedes T_{pq}, T_{pq} must precede T_{ik}, and the conflict edge (T_{pq}, T_{ik}) is in H. As figure 6 shows, the result is a cycle in $PG(H)$ – a contradiction. The assumption that H_c is not a correct successor set history is wrong. Since H is equivalent to H_c, H is a correct stepwise serializable history.
$\Rightarrow$: Since H is a correct stepwise serializable history, it must be equivalent to some correct successor set history H'. H and H' have identical precedence graphs:

$PG(H) = PG(H')$. From Observation 1, $PG(H')$ is acyclic. Hence $PG(H)$ is acyclic. □

The theorem presented here is stronger than the corresponding result in [Farrag and Özsu 1989]. In our model, the acyclicity of the precedence graph is a necessary and sufficient condition for a history to be a correct stepwise serializable history. In [Farrag and Özsu 1989], acyclicity of the precedence graph is a necessary but not sufficient condition for allowable histories. Our stronger result depends on Constraint 1, which ties the next function to conflict between steps. Put differently, Constraint 1 proscribes arbitrary successor set descriptions; there must be a reason for excluding a step from the successor set of another step.

6. CONCURRENCY CONTROL MECHANISMS

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

We make the following assumptions:

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

Although the mechanism we propose is based on the scheme in [Farrag and Özsu 1989], ours is considerably simpler; see Section 7 for a comparison. Our mechanism uses two phase locking on the steps of the transactions. There are two modes in which a data item may be locked by a step - shared mode or exclusive mode. A step acquires an appropriate lock as a prerequisite for accessing a data item. A step is denied a lock if either another step holds a conflicting lock or if the step fails a test based on successor sets. Locks acquired by a step are released when the step commits or aborts.

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

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

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

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

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

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

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

6.1 Algorithms

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

Algorithm 1. Algorithm for Read

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

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

Algorithm 2. Algorithm for Write

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

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

Algorithm 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**
    begin
      if $T_i \notin$ Active-set(x)
        Active-set(x) = Active-set(x) $\cup$ T_i;
      Release the lock on x which was acquired by T_{ij};
    end
end

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

Algorithm 4. Algorithm for Step Abort

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

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

Algorithm 5. Algorithm for Transaction Terminate

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

The Mechanism and Constraint 1. In section 5 we imposed Constraint 1 on successor set descriptions. Constraint 1 requires step $NF(T_{ij}, T_{pq})$ and step T_{pq} to conflict if T_{pq} and T_{ij} conflict and $ty(T_{pq}) \notin SS(ty(T_{ij}))$. It turns out that our particular mechanism does not require Constraint 1 on successor set descriptions. Suppose step T_{pq} and T_{ij} conflict, and the conflicting operations use the data item x. In our mechanism after step T_{ij} commits, Int-Set(T_i,x) is updated. Each subsequent step of T_i updates Int-Set(T_i,x) on commit, whether the step accesses x or not. In particular step $NF(T_{ij}, T_{pq})$ updates Int-Set(T_i,x) on commit. Thus the mechanism behaves as if T_{pq} conflicts with T_{ij} and all subsequent steps, and in particular, the mechanism behaves as if T_{pq} conflicts with step $NF(T_{ij}, T_{pq})$. Hence the mechanism implements Constraint 1 independent of whether the successor set description enjoys the property.

6.2 Correctness of the Mechanism

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

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

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

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

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

□

6.3 Discussion

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

In the Process-read and Process-write algorithms we do not specify how to grant locks when multiple steps contend for the same lock. If the scheme for granting lock requests is unfair, starvation may occur. Starvation can be addressed with the standard techniques [Elmasri and Navathe 1989]. For example we can associate a FIFO queue with each data item; steps are allowed to lock the item only in the order in which they have requested a lock on the data item. Other mechanisms are possible in which priorities are assigned to steps and locks are assigned on the basis of priorities.

Deadlock occurs in standard two phase locking [Bernstein et al. 1987] – two or more transactions may be blocked, each waiting for the other to complete. In standard two phase locking, transactions block only because a transaction wants to lock an item which is already locked by a different transaction in a conflicting mode. In our mechanism steps may block for the same reasons transactions do in standard two phase locking. In addition, a step may block because it is not in the successor set of some other committed step. One approach is to allow deadlock to occur and periodically check for the presence of deadlock using deadlock detection techniques, such as wait-for graphs [Bernstein et al. 1987]. The usual definition of a wait-for graphs must be modified to take into account deadlocks caused by successor set constraints.

There are a variety of issue for supporting compensation. One issue is reliably storing data items which may be needed by a compensating step in case the multistep transaction does not complete. A second issue deals with triggering of the compensating steps. In our model, as in [Garcia-Molina 1983] the initiation of the compensating step is done by the system. Such an approach has the advantage that all transaction aborts, whether user-initiated or failure-related, can be treated in a uniform way. A third issue is recovery from system crash. Transactions that are incomplete at the time of the crash can either be compensated or continued.

7. RELATED WORK

Most transaction-oriented models enforce a very low level, syntactic notion of consistency, namely serializability with respect to read/write conflicts [Bernstein et al. 1987]. Some researchers [Herlihy 1987; Herlihy and Weihl 1991; Lynch et al. 1994; Lynch 1983; Weihl 1984; Weihl 1988] have expanded on this notion by classifying the data objects into abstract data types; transactions access the data objects through operations on the data types. Commutativity of these operations, and not the read/write operations, is used to determine conflicts between transactions, resulting in more concurrency. Still further concurrency can be achieved if recoverability [Badrinath and Ramamritham 1992] of operations is used instead of commutativity. Although we also adopt the theory of abstract data types for the purpose of defining correctness of transaction execution, there is a significant difference in the two approaches. Unlike [Herlihy 1987; Herlihy and Weihl 1991; Weihl 1988], we define the entire database as a state, with the transactions being the operations on the state. 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. [Agrawal et al. 1993; Du and Elmagarmid 1989; Farrag and Özsu 1989; Garcia-Molina 1983; Korth and Speegle 1988; Korth and Speegle 1994; Levy et al. 1991a; Levy et al. 1991b; Wachter and Reuter 1992]. For example, in [Du and Elmagarmid 1989], the authors develop *quasi-serializability* as a correctness criterion for transactions executing on heterogeneous distributed database systems. A global history is quasi-serializable if it is equivalent to a quasi serial history; quasi serial histories require only the global transactions (transactions accessing more than one local database) to execute serially. In some of these works, researchers have decomposed transactions into steps [Agrawal et al. 1993; Farrag and Özsu 1989; Garcia-Molina 1983; Jajodia and Meadows 1987; Korth and Speegle 1988; Korth and Speegle 1994; Sha et al. 1988; Shasha et al. 1992] and developed semantic based correctness criteria [Agrawal et al. 1993; Farrag and Özsu 1989; Garcia-Molina 1983]. 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 [Farrag and Özsu 1989] and Korth and Speegle [Korth and Speegle 1988; Korth and Speegle 1994] bear close resemblance to our work and are discussed in separate subsections.

In [Shasha et al. 1992], the authors decompose a transaction into steps based only on conflict information among transactions. Since they do not use any semantic knowledge, certain decompositions, valid in our model, are not allowed by their system. Specifically, the decomposition of *Reserve* into steps $R1$, $R2$ and $R3$ in the Hotel Database, will not be a correct decomposition according to their model.

In [Sha et al. 1988], 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 [Agrawal et al. 1993] propose three correctness criteria: *consistency*, *orderability* and *strong orderability* for non-serializable executions. The data objects in their model are classified into various abstract data types. These objects are accessed through user programs that 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 is 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, 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 ConTract model [Wachter and Reuter 1992] is a formal model for specifying and controlling long-lived complex applications like office automation, CAD applications, etc. Consistency is not based on serializability but on application dependent correctness criteria. A ConTract is composed of a number of steps that are elementary units of work; the control flow of the steps is specified by a script. A ConTract is allowed to externalize its partial results before it completes. Compensation of a ConTract takes place when a user wants to abort a ConTract that has committed some of its steps. Corresponding to each step in the ConTract, a compensating step is specified which must be executed when the user cancels a ConTract. Cascading compensations may occur in the ConTract model. The motivation for our model is different; our model is suitable for applications consisting of long duration transactions. Thus, we do not provide a generalized model for modeling complex activities requiring cooperation among different transactions.

Garcia-Molina [Garcia-Molina 1983] classified transactions into a number of *semantic types*, associated *compatibility sets* with each of these types, and defined 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 idea of using compensation has also been explored [Garcia-Molina 1983]. If a transaction T cannot be successfully completed, compensating steps are executed to compensate for the updates caused by the transaction. This avoids the aborts of compatible transactions that used the data produced by the partial execution of T. In this model, corresponding to each step s_i of a transaction, except the last, the application developer specifies a counterstep c_i. The counterstep c_i semantically undoes the effect of the step s_i. The counter steps are initiated by the system when a transaction aborts. For reasons of performance the authors state that countersteps should not be delayed or aborted. The burden of classifying the transactions into types, decomposing the transaction into steps, providing the compatibility sets, and designing countersteps rests with the application developer. A formal treatment of compensating transactions is presented in [Korth et al. 1990] in which the authors propose a generalized model that allows the definition of various types of compensating transactions.

The Saga model [Garcia-Molina and Salem 1987] also addresses long duration transactions. A Saga consists of a collection of transactions, each of which maintains database consistency. However any partial execution of the Saga is undesirable; either all the transactions in a Saga complete successfully or compensating transactions should be run to amend for the partial execution of the Saga. Thus corresponding to every transaction in the Saga, except the last one, a compensating transaction is specified. The compensating transaction semantically undoes the effect of the corresponding transaction. Note that the role of compensating transactions in the Saga is different than the role of compensating steps in our model. In a Saga, each transaction as well compensating steps always maintain database consistency. In our model, on the other hand, execution of step violates database consistency; compensating steps are therefore required to restore consistency.

ACTA [Chrysanthis and Ramamritham 1994] 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 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.

The idea of a threefold classification of states – safe, unsafe but acceptable, and unsafe and unacceptable – is proposed in the context of fault tolerance in [Arora and Kulkarni 1995]. The idea is to tolerate faults that take a system to the unsafe but acceptable states. The proof obligations are to show that certain actions are not attempted in unsafe states and that eventually the system returns to a safe state. Our model differs in that the intermediate states are the result of design, not of faulty behavior. Further, information about committed steps, which is captured by histories, allows stronger arguments to be made about system behavior.

7.1 Korth and Speegle

In [Korth and Speegle 1988; Korth and Speegle 1994] 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 [Korth and Speegle 1994]. In the NT/PV model, the post conditions of top level transactions imply the satisfaction of database integrity constraints. As stated in [Korth and Speegle 1994], 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.

As an example, consider the history $H =< S_{11}, S_{21}, S_{12}, S_{13} >$ where $ty(S_{11}) = R1$, $ty(S_{12}) = R2$, $ty(S_{13}) = R3$, $ty(S_{21}) = CancelD$. The history H is shown in Figure 7. Executing the first step of the reserve transaction,S_{11}, takes us to a state in $\widehat{\mathbf{ST}} - \mathbf{ST}$. After executing the only step of the Cancel transaction,S_{21}, we are still in one of the states in $\widehat{\mathbf{ST}} - \mathbf{ST}$. Such an execution is correct in our model, but not in NT/PV model because the last step of the Cancel transaction does not satisfy the integrity constraints of the database. Note, however, that we do

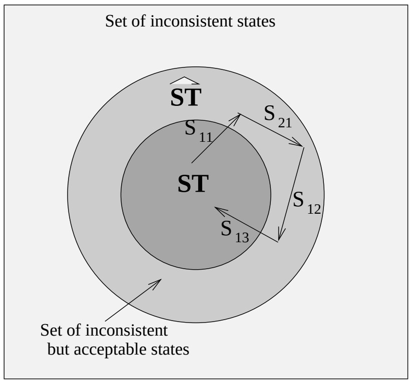


Fig. 7. Example Execution Not Permitted by the NT/PV Model

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) and 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 [Korth and Speegle 1994], our algorithm differs from that in [Korth and Speegle 1994].

7.2 Farrag and Özsu

Our work is closely related to that of [Farrag and Özsu 1989] and hence we discuss this work at a more detailed level. In [Farrag and Özsu 1989] the authors introduce the notion of *breakpoint sets* associated with the steps of transactions. The breakpoint set of a step S_{ij} of a transaction T_i gives the types of transactions that may interleave transaction T_i after step S_{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 S_{jk} and $S_{j(k+1)}$ of another transaction T_j, T_i is in the breakpoint set of the step S_{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

```
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;
```

Fig. 8. RESERVE and CANCEL transactions as given in Farrag and Özsu's work

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 [Farrag and Özsu 1989]. However, the semantics of the successor set differs from that of breakpoint sets. The 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 [Farrag and Özsu 1989]. We use only two kinds of locks - shared and exclusive as opposed to the four kinds of locks of [Farrag and Özsu 1989]. 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 [Farrag and Özsu 1989] 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.

To illustrate the differences further, we repeat below the motivating example from Section 2 of [Farrag and Özsu 1989]. It turns out that this example fails to satisfy Assumption 5.1 in [Farrag and Özsu 1989], which is important because the mechanism given in [Farrag and Özsu 1989] requires this assumption. Afterwards, we show how our model and mechanism can handle this example.

The example in Section 2 of [Farrag and Özsu 1989] involves three transaction types, RESERVE, CANCEL, and REPORT. We show the details of the RESERVE and CANCEL transaction types in Figure 8.

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 described 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 [Farrag and Özsu 1989, 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 [Farrag and Özsu 1989] 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.

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. 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 algorithm that ensures execution histories that are stepwise conflict-serializable and also respect the successor set constraints. The algorithm is substantially simpler than the corresponding one in [Farrag and Özsu 1989]. We have also identified some of the issues which must be resolved before an actual implementation can be developed.

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

An important question is how well our model scales up to real-world applications. The necessary properties must be demonstrated for applications which must be implemented by our model. For the purpose of this paper we have used the Z specification language and all the analysis are done by hand. However for real life applications this may not be feasible; for such applications it will be necessary to automate to the extent possible the process of discharging the proof obligations. Industrial-level tool support for such an endeavor is essential, and the use of other specification languages which have adequate tool support, such as the B Method [Abrial 1993], needs to be investigated.

ACKNOWLEDGMENTS

It is a pleasure to acknowledge the valuable suggestions made by the anonymous referees.

A preliminary version of this paper appeared under the title "Using Formal Methods To Reason About Semantics-Based Decomposition of Transactions" in *Proceedings of the 21st International Conference on Very Large Data Bases*, pp. 218 – 227, Zurich, Switzerland, Sept. 1995.

REFERENCES

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

Agrawal, D., Abbadi, A. E., and Singh, A. K. 1993. Consistency and orderability: Semantics-based correctness criteria for databases. *ACM Trans. Database Syst. 18,* 3 (September), 460–486.

Arora, A. and Kulkarni, S. S. 1995. Designing masking fault-tolerance via nonmasking fault-tolerance. In *Proceedings of Symposium on Reliable Distributed Systems*, Bad Neuenahr, Germany, pp. 174–185.

Badrinath, B. and Ramamritham, K. 1992. Semantics-based concurrency control: Beyond commutativity. *ACM Trans. Database Syst. 17,* 1 (March), 163–199.

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

Chrysanthis, P. K. and Ramamritham, K. 1994. Synthesis of extended transaction models using ACTA. *ACM Trans. Database Syst. 19,* 3 (September), 450–491.

Du, W. and Elmagarmid, A. 1989. Quasi serializability: A correctness criterion for global concurrency control in interbase. In *Proceedings of the International Conference on Very Large Data Bases*, Amsterdam, Netherlands, pp. 347–355.

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

Farrag, A. A. and Özsu, M. T. 1989. Using semantic knowledge of transactions to increase concurrency. *ACM Trans. Database Syst. 14,* 4 (December), 503–525.

Garcia-Molina, H. 1983. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst. 8,* 2 (June), 186–213.

GARCIA-MOLINA, H. AND SALEM, K. 1987. Sagas. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, San Francisco, CA, pp. 249–259.

HERLIHY, M. 1987. Extending multiversion time-stamping protocols to exploit type information. *IEEE Trans. Comput. 36,* 4 (April), 443–448.

HERLIHY, M. P. AND WEIHL, W. E. 1991. Hybrid concurrency control for abstract data types. *J. Comput. Syst. Sci. 43,* 1 (August), 25–61.

JAJODIA, S. AND MEADOWS, C. 1987. Managing a replicated file in an unreliable network. In *Proceedings of the IEEE International Conference on Data Engineering*, Los Angeles, CA, pp. 396–404.

KORTH, H., LEVY, E., AND SILBERSCHATZ, A. 1990. A formal approach to recovery by compensating tran sactions. In *Proceedings of the International Conference on Very Large Data Bases*, Brisbane, Australia, pp. 95–106.

KORTH, H. F. AND SPEEGLE, G. 1994. Formal aspects of concurrency control in long-ouration transaction systems using the NT/PV model. *ACM Trans. Database Syst. 19,* 3 (September), 492–535.

KORTH, H. F. AND SPEEGLE, G. D. 1988. Formal model of correctness without serializability. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pp. 379–386.

LEVY, E., KORTH, H. F., AND SILBERSCHATZ, A. 1991a. An optimistic commit protocol for distributed transaction management. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Denver, CO, pp. 88–97.

LEVY, E., KORTH, H. F., AND SILBERSCHATZ, A. 1991b. A theory of relaxed atomicity. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 95–109.

LYNCH, N., MERRITT, M., WEIHL, W., AND FEKETE, A. 1994. *Atomic Transactions.* Morgan Kaufmann, San Mateo, CA.

LYNCH, N. A. 1983. Multilevel atomicity—A new correctness criterion for database concurrency control. *ACM Trans. Database Syst. 8,* 4 (December), 484–502.

OWICKI, S. AND GRIES, D. 1976. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM 19,* 5 (May), 279–285.

POTTER, B., SINCLAIR, J., AND TILL, D. 1991. *An Introduction to Formal Specification and Z.* Prentice-Hall, New York.

RASTOGI, R., KORTH, H. F., AND SILBERCHATZ, A. 1995. Exploiting transaction semantics in multidatabase systems. In *Proceedings of the International Conference on Distributed Computing Systems*, Vancouver, Canada, pp. 101–109.

SHA, L., LEHOCZKY, J. P., AND JENSEN, E. 1988. Modular concurrency control and failure recovery. *IEEE Trans. Comput. 37,* 2 (February), 146–159.

SHASHA, D., SIMON, E., AND VALDURIEZ, P. 1992. Simple rational guidance for chopping up transactions. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, San Diego, CA, pp. 298–307.

SPIVEY, J. 1992. *The Z Notation: A Reference Manual, Second Edition.* Prentice-Hall, Englewood Cliffs, NJ.

WACHTER, H. AND REUTER, A. 1992. The ConTract model. In A. K. ELMAGARMID (Ed.), *Database Transaction Models for Advanced Applications*, pp. 219–263. San Mateo, CA: Morgan Kauffman.

WEIHL, W. E. 1984. Specification and implementation of atomic data types. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA.

WEIHL, W. E. 1988. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput. 37,* 12 (December), 1488–1505.

Appendix – Properties for the Example

The Composition Property

The *Reserve* transaction is the only multistep transaction in the example. The proof obligation is :

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

The left hand side of the equivalence is obtained by schema composition [Potter et al. 1991] followed by schema conjunction and evaluates to:

$$\begin{array}{|l}
\hline
Hotel \land ((R1 \mathbin{⨟} R2 \mathbin{⨟} R3) \setminus (underway, acquired, underway', acquired')) \\
\Delta Hotel \\
g? : Guest \\
r! : Room \\
\hline
res < total \\
g? \notin \operatorname{dom} RM \\
ST(r!) = Available \\
res' = res + 1 \\
ST' = ST \oplus \{r! \mapsto Taken\} \\
RM' = RM \cup \{g? \mapsto r!\} \\
\hline
\end{array}$$

The predicate part of this schema is equivalent to the predicate part of the *Reserve* schema given in Figure 1; hence the two schemas are equivalent.

The Sensitive Transaction Isolation Property

In the hotel database we have two sensitive transactions: *Report* and *Reserve*. In each case, the Sensitive Transaction Isolation Property is achieved by construction.

Consider *Report* transactions. We compute the subset of the original invariants that must hold as a precondition for *Report*. We obtain these from *Hotel* by hiding the state variables *not* involved in producing the outputs of *Report*. The only state variable not involved in producing the output of *Report* is *res*. Hiding the variable *res* from *Hotel* produces the following schema:

$$\begin{array}{|l}
\hline
Hotel \setminus (res) \\
RM : Guest \rightarrowtail\!\!\!\!\!\rightarrow Room \\
ST : Room \nrightarrow Status \\
\hline
\exists\, res : \mathbb{N} \bullet (\\
\#RM = res \land \\
\operatorname{dom}(ST \rhd \{Taken\}) = \operatorname{ran} RM) \\
\hline
\end{array}$$

The constraint simplifies to

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

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

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

and $acquired = \varnothing$. Hence we include $acquired = \varnothing$ as a precondition for *ReportD*.

Next we consider *Reserve* transactions. 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:

$$
\begin{array}{|l}
\hline
\mathit{Hotel} \setminus (\mathit{res}, \mathit{RM}) \\
ST : \mathit{Room} \nrightarrow \mathit{Status} \\
\hline
\exists\, \mathit{res} : \mathbb{N};\ \mathit{RM} : \mathit{Guest} \rightarrowtail \mathit{Room} \bullet (\\
\#\mathit{RM} = \mathit{res} \wedge \\
\mathrm{dom}(ST \rhd \{\mathit{Taken}\}) = \mathrm{ran}\, \mathit{RM}) \\
\hline
\end{array}
$$

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.

For the hotel example, we do not have any multistep transaction in which outputs are generated in multiple steps. Thus the problem of outputs from later steps not being consistent with outputs from earlier steps does not arise in this example.

The Consistent Execution Property

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

Let n_1, n_2, n_3, cn_1, cn_2 be the number of steps of type $R1$, $R2$, $R3$, $CompR2$ and $CompR3$ respectively present in any complete history. The variable *underway* is modified by steps of type $R1$ and $R3$, and by compensating steps of type $CompR2$, $CompR3$ of reserve transaction. *underway* is incremented in steps of type $R1$ and it is decremented in steps of type $R3$ and by compensating steps of type $CompR2$ and $CompR3$. Thus *underway* is given by the following expression

$$underway = n_1 - (n_3 + cn_1 + cn_2) \qquad \cdots \qquad (1)$$

The variable *acquired* is modified by steps of type $R2$, $R3$ and by compensating steps of type $CompR3$. In step of type $R2$ a room is added to the set *acquired* and in steps of type $R3$ or compensating steps of type $CompR3$ the room is taken out from the set *acquired*. Thus we can write,

$$\mid acquired \mid = n_2 - (n_3 + cn_2) \qquad \cdots \qquad (2)$$

In a complete history all the reserve transactions have completed execution. Therefore corresponding to each step of type $R1$ in the history there is a step of type $R2$ or a compensating step of type $CompR2$. Thus

$$n_1 = n_2 + cn_1 \qquad \cdots \qquad (3)$$

Similarly corresponding to each $R2$ there is either a step of type $R3$ or a compensating step of type $CompR3$. Thus

$$n_2 = n_3 + cn_2 \qquad \cdots \qquad (4)$$

From (1),(2),(3) and (4) we can conclude that in a complete history, 1) $underway = 0$ and 2) $\mid acquired \mid = 0$ or $acquired = \varnothing$. Hence the hotel database has the consistent execution property.

The Compensation Property

The proof that the hotel database has the compensation property proceeds by induction. In the hotel example only the reserve transaction is a multistep transaction. Suppose we have n incomplete reserve transactions in the correct partial semantic history H_p. We show how an incomplete reserve transaction can be compensated, thereby decreasing the number of incomplete *Reserve* transactions from n to $n-1$. 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.

The proof of how a correct partial semantic history containing one incomplete *Reserve* transaction can be compensated proceeds in two parts. In the first part we prove that in a history if a *Reserve* transaction has completed only step of type $R1$ then it is possible to execute the compensating step of type $CompR2$ no matter what steps of other transactions execute between $R1$ and $CompR2$. In the second part we prove that if a *Reserve* transaction has completed the first two steps of type $R1$ and $R2$ respectively, then it is possible to execute a step of type $CompR3$ after $R2$ even though steps of other transactions may have run after an $R2$.

Proof of the first part:

The step $CompR2$ is scheduled only after a step $R1$ has been executed but the corresponding step $R2$ or $CompR2$ has not been executed. The step $CompR2$ has two preconditions, (i) $res > 0$ and (ii) $underway > 0$. The variable *res* is modified by reserve and cancel transactions. *res* is incremented by steps of type $R1$ and it is decremented by steps of type $CompR2$, $CompR3$ and *CancelD*. Steps of type *CancelD* cancels a previously completed reservation. Thus the cancel transaction decrements the value of *res* which was incremented by some other *Reserve* transaction that completed previously. Steps of type $CompR2$ or $CompR3$ belonging to other transactions also decrement the value of *res*; however this decrement is done to neutralize the increment caused by step $R1$ of the corresponding reserve transaction. Thus the precondition of $CompR2$ will always be satisfied if $CompR2$ is executed after step $R1$ of the same *Reserve* transaction.

The variable *underway* is modified only by steps of *Reserve* transaction. The *Reserve* transaction increments the *underway* variable in the first step which is of type $R1$. Corresponding to every step of type $R1$ either a step of type $R3$ is executed or a compensating step of type $CompR2$ or $CompR3$ is executed. Thus the steps of other transactions of type $R3$, $CompR2$ or $CompR3$ decrements the variable *underway* which was incremented by the corresponding $R1$. Hence even if steps of other transactions appear after $R1$, the precondition $underway > 0$ of $CompR2$ will always be satisfied.

Proof of the second part:

The step $CompR3$ is executed after a step of type $R2$ has been executed and for which the corresponding step of type $R3$ or the compensating step of type $CompR3$ has not been executed. The step $CompR3$ has three preconditions (i) $res > 0$, (ii) $underway > 0$ and (iii) $r! \in acquired$.

The variable *res* is incremented by the *Reserve* transaction in step of type $R1$ and is decremented by the compensating steps of type $CompR2$ and $CompR3$. The variable *res* also is decremented by the cancel transaction. Now other transaction steps of type $CompR2$ or $CompR3$ decrement the variable *res* to neutralize the increment of *res* done by the corresponding $R1$. Also the step *CancelD* cancels a complete reservation and so decrements *res* which was earlier incremented by some previously completed *Reserve* transaction. Thus steps of other transactions decrementing *res* do not violate the precondition $res > 0$ of $CompR3$ of the *Reserve* transaction.

The variable *underway* is incremented by steps of type $R1$ and is decremented by steps of type $R3$, $CompR2$ or $CompR3$. Now steps of type $R3$, $CompR2$ or $CompR3$ of other transactions decrement the value of *underway*; this decrement is to neutralize the increment caused by the corresponding $R1$. Thus even if steps of other *Reserve* transactions execute after step of type $R2$, the precondition, $underway > 0$ of $CompR3$, still get satisfied.

The variable *acquired* is updated in steps of type $R2$, $R3$ and $CompR3$ of the *Reserve* transaction. A room is inserted in the set *acquired* by a step of type $R2$ of the *Reserve* transaction. The room can be removed from the set *acquired* only by steps of type $R3$ or $CompR3$ of the same *Reserve* transaction. Thus steps of other transactions executing after $R2$ do not violate the precondition of $CompR3$.

The Valid Successor Set Property

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 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 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 $acquired = \varnothing$ to any and all occurrence(s) of steps of type *ReportD* 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 *ReportD* in H_1 are not present in the corresponding steps in H_2.

Let us consider the precondition ($acquired = \varnothing$) that was added to *ReportD* in H_1. From the specifications of $R1$, $R2$, $R3$, *CancelD*, and *ReportD* we see that the only situation in which the precondition of a step of type *ReportD* is not satisfied, is when at least one step of type $R2$ has been executed but the next step, which is of type $R3$ or $CompR3$, has not yet been executed. However, as $ReportD \notin SS(R2)$, a step of type $ReportD$ will never be allowed to execute between a step of type $R2$ and a step of type $R3$ or $CompR3$. Thus in H_1 $acquired = \varnothing$ is always satisfied before $ReportD$ 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 mechanism. In any correct history in $\mathbf{H_1}$, the preconditions are satisfied at each step and a step of type $ReportD$ can never appear between steps of type $R2$ and $R3$ or $CompR3$. The successor set mechanism ensures that a step of type $ReportD$ does not occur between steps of type $R2$ and an $R3$ or $CompR3$ 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 $\mathbf{H_2}$.

In the second part of the valid successor set property, it is required to show that the specification generated from successor set description has the compensation property. We achieve this property by construction - we include the types of compensating steps in the successor set of any type of step.

Received August 1995; revised May 1996; accepted September 1996