Using Formal Methods To Reason About Semantics-Based Decompositions Of Transactions

Paul Ammann* Sushil Jajodia† Indrakshi Ray‡
Center for Secure Information Systems
and
Department of Information and Software Systems Engineering
George Mason University
Fairfax, VA 22030-4444

Abstract

Many researchers have investigated the process of decomposing transactions into smaller pieces to increase concurrency. The focus of the research is typically on implementing a decomposition supplied by the database application developer, with relatively little attention to what constitutes a desirable decomposition and how the developer should obtain such a decomposition. In this paper, we argue that the decomposition process itself is worthy of attention. A decomposition generates a set of proof obligations that must be satisfied to show that a particular decomposition correctly models the original collection of transactions. We introduce the notion of semantic histories to formulate and prove the necessary properties. Since the decomposition impacts not only the atomicity of transactions, but isolation and consistency properties as well, we present a technique based on formal methods that allows these properties to be surrendered in a carefully controlled manner.

1 Introduction

Key to the success of the transaction model are the *atomicity*, *consistency*, and *isolation* properties. Atomicity ensures that either all actions of a transaction complete successfully or all of its effects are absent. Consistency ensures that a transaction when executed by itself, without interference from other transactions, maps the database from one consistent state to another consistent state. Isolation ensures that no transaction ever views the partial effects of other transactions even when transactions execute concurrently.

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

The process of breaking transactions into steps not only sacrifices atomicity (since atomicity of the single logical action is lost), but impacts consistency and isolation as well. Execution of a step may leave the database in an inconsistent state, which may be viewed by other transactions or steps; therefore, it is necessary to reason about the interleavings of the steps of different transactions. Even if the step-by-step decomposition of a single transaction into steps is understood in isolation, reasoning about the interleaving of these steps with other transactions, possibly also decomposed into steps, is not completely understood.

To reason about interleavings, we introduce the notion of *semantic* histories which not only list the sequence of steps forming the history, but also convey information regarding the state of the database before and after execution of each step in the history. We identify properties which semantic histories must satisfy to show that a particular decomposition correctly models the original collection of transactions.

The rest of the paper is organized as follows. Section 2 briefly reviews related research. Section 3 presents a motivating example. Section 4 contains the model with refinements to the motivating example. Section 5 discusses an implementation. Section 6 concludes the

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

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

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

paper.

We adopt the Z specification language [18] for expressing model-based specifications of abstract data types or ADTs. 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 specification in English. Specification and analysis conventions peculiar to Z will be explained as the need arises.

2 Related Work

Most transaction-oriented models enforce a very low level, syntactic notion of consistency, namely serializability with respect to read/write conflicts [3]. Two separate lines of work have expanded on this notion.

The first line is the work on atomic transactions [8, 9, 13, 14, 20, 21]. This line of work is based on ADTs. We also adopt the theory of ADTs to define correctness, but there is a crucial difference in focus. We are interested in achieving more concurrency by expanding the set of correct execution histories such that some transactions need not be atomic.

The second line is a set of papers on breaking transactions into steps [17], and in particular papers on semantic database concurrency control [1, 4, 16, 2, 5, 6, 10]. Researchers have introduced the notions of transaction steps, countersteps, allowed vs. prohibited interleavings of steps, and implementations in locking environments. The focus is typically on implementing a decomposition supplied by the database application developer, with relatively little attention to what constitutes a desirable decomposition and how the developer should obtain such a decomposition. We find the decomposition process itself to be worthy of attention, so we give the developer a model in which to decompose transactions, and we define properties to assure the developer as to the soundness of a given decomposition. Only then do we consider the problem of implementing our decomposition in a two-phase locking environment.

The idea of specifying transactions by its preconditions and post conditions has been illustrated in [12, 11]. The authors propose the NT/PV transaction model based on nested transactions, multiple versions and explicit predicates. 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 correct if it begins in a state that satisfies its input conditions, the subtransactions are executed in an order consistent with the partial order, and the database state after the execution of the transaction satisfies the transaction's post condition. An execution of an interleaved set of transactions is NT/PV correct if every transaction in the set executes correctly. In [12, 11], the burden of correctly specifying the preconditions, postconditions of the transactions, and determining the partial order of the subtransactions in a transaction lies with the application developer. Our work focusses on providing tools to the application developer in specifying the preconditions and postconditions of transaction, and showing how to reason about the interleaving of steps of different transactions.

	Set of Natural Numbers
A	Powerset of Set A
$\#A$	Cardinality of Set A
$\setminus$	Set Difference
$A \mathbin{;} B$	A Composed with B
$x \mapsto y$	Ordered Pair (x, y)
$A \nrightarrow B$	Partial Function from A to B
$A \quad B$	Partial Injective Function from A to B
$B - A$	Relation A: Set B Removed from Domain
$A \quad B$	Relation A: Range Restricted to Set B
$\mathrm{dom}\ A$	Domain of Relation A
$\mathrm{ran}\ A$	Range of Relation A
$A \oplus B$	Function A Overridden with Function B
$x?$	Variable $x?$ is an Input
$x!$	Variable $x!$ is an Output
x	State Variable x before an Operation
x'	State Variable x' after an Operation
ΔA	Before and After State of Schema A
ΞA	ΔA with No Change to State

Table 1: Z Notation

3 The Hotel Database

We illustrate our ideas with an example of a hotel database. A Z specification of the hotel database appears in figure 1. (Table 1 briefly explains the Z notation used in this and subsequent examples.) 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 two types, *Guest* and *Room*, enumerate all possible guests and all possible hotel rooms, respectively. The global variable *total* is the size of the hotel.

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

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

The integrity constraints on the objects in hotel

$[Guest, Room]$

$Status ::= Available \mid Unavailable$

$total :$

Hotel

$res : \;\; ; \; RM : Guest \;\; Room$

$guest : \;\; Guest; \; ST : Room \twoheadrightarrow Status$

$\#RM = res$

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

Reserve

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

$res < total; \; g? \notin guest$

$ST(r!) = Available; \; res' = res + 1$

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

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

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

Cancel

$\Delta Hotel; \; g? : Guest$

$g? \in guest; \; res' = res - 1$

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

$RM' = \{g?\} - RM$

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

Report

$\Xi Hotel$

$currentstatus! : Room \twoheadrightarrow Status$

$currentassignments! : Guest \;\; Room$

$currentstatus! = ST$

$currentassignments! = RM$

Figure 1: Initial Specification of the Hotel Database

database appear in the bottom part of *Hotel*. There are two integrity constraints:

1. $\#RM = res$. The number of guests who have been assigned rooms (the size of the RM function) equals the total number of reservations (res).
2. $\mathrm{dom}(ST \;\; \{Unavailable\}) = \mathrm{ran}\, RM$. The set of unavailable rooms ($\mathrm{dom}(ST \;\; \{Unavailable\})$) is exactly the set of rooms reserved by guests ($\mathrm{ran}\, RM$). In other words, every unavailable room must be associated with some guest.

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

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

Report has no precondition, and merely produces the state components ST and RM as outputs.

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

4 The Model

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

A *transaction* is an operation on a database state. Associated with each transaction is a set of *preconditions* and a set of *postconditions* on the database objects. A precondition limits the database states to which a transaction can be applied. For example, a *Reserve* transaction has a precondition that the hotel have at least one room available. A postcondition constrains the possible database states after a transaction completes. For example, a *Reserve* transaction has a postcondition that there be some room available before the reservation that is unavailable after the reservation. Preconditions and postconditions must be strong enough so that if a transaction executes on a consistent state, the result is again a consistent state.

Instead of executing a transaction as an atomic unit, we break up a transaction into steps, and execute each of these steps as an atomic unit. The decomposition exploits the semantic information associated with the transaction. Although such a decomposition process is very much application specific, nevertheless, we identify necessary properties that must be satisfied by a decomposition to ensure the system's proper functioning.

Definition 1 [Transaction Decomposition] A *decomposition* of a transaction T_i is a sequence of two

NaiveR1

$$\begin{array}{l} \Delta Hotel \\ \hline res < total \\ res' = res + 1 \\ ST' = ST \\ RM' = RM \\ guest' = guest \end{array}$$

NaiveR2

$$\begin{array}{l} \Delta Hotel;\ r! : Room \\ \hline ST(r!) = Available \\ ST' = ST \oplus \\ \quad \{r! \mapsto Unavailable\} \\ res' = res;\ RM' = RM \\ guest' = guest \end{array}$$

NaiveR3

$$\begin{array}{l} \Delta Hotel \\ r! : Room;\ g? : Guest \\ \hline g? \notin guest \\ RM' = RM \cup \{g? \mapsto r!\} \\ guest' = guest \cup \{g?\} \\ res' = res;\ ST' = ST \end{array}$$

Figure 2: A Naive Decomposition

or more atomic *steps* $< T_{i1}, T_{i2}, \ldots, T_{in} >$. In place of T_i, these steps are executed in the given order as atomic operations on a database state.

To show that the decomposition has been performed correctly, we must check that the steps, when executed in the correct sequence and without interference from other transactions, model the execution of the original transaction.

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

4.1 A Naive Decomposition of Reserve

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

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

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

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

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

The above decomposition has a serious flaw in that none of the proposed steps, considered by itself, maintains the invariants in *Hotel*. For example, $NaiveR1$ 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 one of the proposed steps would leave the invariants unsatisfied, and other transactions could be exposed to the inconsistent state. For example, *Report* may produce an inconsistent output if executed in a state in which the invariants do not hold.

4.2 Modification of Original Invariants

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

As noted earlier, one way to solve this problem is to allow only those decompositions that have the property that partial executions leave the database state consistent. Such an approach is exceedingly restrictive, and so we reject it. In the hotel example, the informal description of the steps into which *Reserve* is broken is perfectly satisfactory; what is unreasonable is the insistence that the invariants of *Hotel* hold at all intermediate steps. We need a formal model that can accommodate the notion that some – but not all – violations of the invariants are acceptable.

Figure 4 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 5) is allowed, and steps of some other transactions, e.g. *RefinedCancel*, 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 4 has many advantages, including greater concurrency among steps. We formalize the model as follows.

Let I denote the original invariants. Let **ST** denote the set consisting of all consistent states; i.e., $\mathbf{ST} = \{ST : ST \text{ satisfies } I\}$. A transaction T_i always operates on a consistent $ST \in \mathbf{ST}$. If ST_i denotes the state after the execution of T_i, then ST_i is also in **ST**. However, when T_i is broken up into steps $< T_{i1}, T_{i2}, \ldots, T_{in} >$, each step T_{ij} is executed as an atomic operation. If ST_{ij} represents the partial execution of T_i, it is possible that after execution of step T_{ij}, the resulting database state ST_{ij} no longer satisfies the invariants I and, therefore, lies outside **ST**. Figure 3

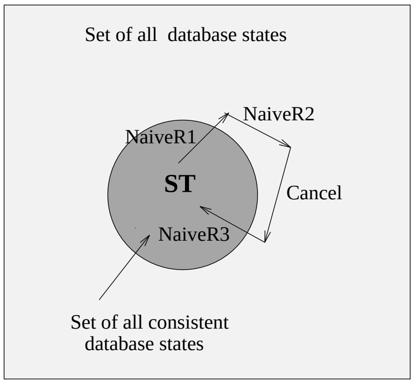

Figure 3: General classification of database states

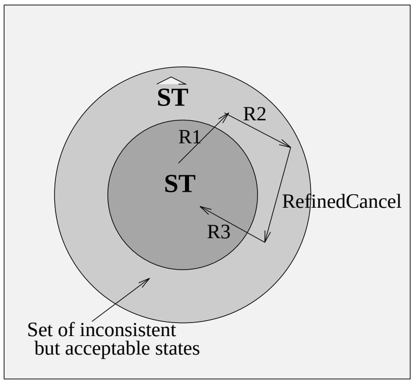

Figure 4: Database states as classified in our model

illustrates this possibility for the naive decomposition of the hotel example.

In our approach, we define a new set of invariants, $\hat{I}$, by relaxing the original invariants I. We decompose each transaction such that execution of any step results in a database state that satisfies $\hat{I}$; if all the steps of a transaction are executed serially on a consistent initial state, the final state satisfies the original set of invariants. Let $\widehat{\mathbf{ST}} = \{$ ST : ST satisfies $\hat{I}\}$. The relationship between $\mathbf{ST}$ and $\widehat{\mathbf{ST}}$ is shown in figure 4. 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 4 is that the set of inconsistent but acceptable states is formally identified and distinguished from the states that are unacceptable. The advantage is that formal analysis can be used to investigate activities in $\widehat{\mathbf{ST}}$.

To reason about the correctness of decomposing transactions into steps, and avoid the problems of a naive decomposition, we use *auxiliary variables* to generalize the invariants. Auxiliary variables are a standard method of reasoning about concurrent executions [15], and, in particular, have been applied to the problem of semantic database concurrency control [6]. Our work focuses more on the decomposition process than does [6], and so we emphasize the role of auxiliary variables more strongly.

In the hotel example, we generalize the invariant $\#RM = res$ by introducing an auxiliary variable to express the fact that number of guests with rooms might differ from total reservations by the number of reserve transactions in progress. We generalize the invariant $\mathrm{dom}(ST \quad \{Unavailable\}) = \mathrm{ran}\, RM$ by introducing another auxiliary variable to express the fact that the unavailable rooms might differ from the rooms assigned to guests by those rooms selected by reserve transactions in progress. Before we show these changes to the example, we present two properties that a decomposition must possess. We note that the auxiliary variables are introduced for purposes of analysis, and are eliminated in our implementation.

4.3 Composition Property

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

Composition Property Let T_i denote the original transaction and $T_{i1}, T_{i2}, \ldots, T_{in}$ denote the corresponding steps. T_i and its steps are related as follows:

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

Executing the steps $T_{i1}, T_{i2}, \ldots, T_{in}$ serially on a state satisfying the original invariants I, changes the original database objects in the same way as executing the original transaction T_i on the same state.

From an implementation perspective, the composition property is similar to requiring that the stepwise execution of the steps be view equivalent to that of the original transaction. A complicating factor is that the decomposition may introduce additional database objects; the composition property does not limit the values of these additional database objects (for example, compare *Hotel* in figure 1 with *RefinedHotel* in figure 5).

4.4 Sensitive Transaction Isolation

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

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

In our model, we ensure the sensitive transaction isolation property by construction. For each sensitive transaction, we compute the subset of the original integrity constraints, I, relevant to the calculation of any outputs. This subset of I is included as an explicit precondition for the sensitive transaction.

4.5 A Valid Decomposition

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

To make the invariants more general, we add auxiliary variables and define a new state *RefinedHotel*. We add the auxiliary variable *tempreserved*, which is a natural number, to denote the reservations that have been partially processed. We also add the auxiliary variable *tempassigned*, which is a set of rooms, to denote the rooms that have been reserved but which have not yet been assigned to guests. The invariants are modified accordingly. The schema *RefinedHotel* together with the modified invariants is shown in figure 5.

$R1$, $R2$ and $R3$ are three steps of the reserve transaction. To correctly implement a *Reserve*, the three steps are to be run in the serial order $R1, R2, R3$. The three steps satisfy the composition property; see the appendix for a further discussion of this point. Although *Reserve* is a sensitive transaction, it turns out that no additional preconditions are needed to ensure that the output $r!$ reflects a consistent state; see the appendix for details.

The refined version of the single step *Cancel* transaction is nearly identical to the unrefined version, except that the auxiliary variables *tempassigned* and *tempreserved* are not changed.

Report is a sensitive transaction, and we establish the sensitive transaction isolation property by construction. A formal treatment is in the appendix. Informally, *Report* transaction outputs values of ST and RM. ST and RM involve the original invariant:
$\mathrm{dom}(ST \quad \{Unavailable\}) = \mathrm{ran}\, RM$
which can be derived from
$\mathrm{dom}(ST \quad \{Unavailable\}) = \mathrm{ran}\, RM \cup tempassigned$
if the auxiliary variable *tempassigned* satisfies
$tempassigned =$ The refined version of *Report* is shown in figure 5.

4.6 Semantic Histories

Since we modify the invariants, several questions must be answered. In particular, we would like to know if and when the database state returns to a consistent state. We will answer these questions after we give some definitions.

Definition 2 [History] A *history* H over a set of transactions $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$ is a sequence of steps $< T_{i_1 j_1}, T_{i_2 j_2}, \ldots, T_{i_p j_q} >$, $1 \le i_1, \ldots, i_p \le m$, $T_{i_r j_s}$ is a step in T_{i_r}, $1 \le r \le m$, $1 \le s \le n$, such that

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

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

Example 1 $< R1, R3, Report, R2 >$ is not a history since it violates condition (2) and (3). $< R1, R2, R3, R2 >$ is not a history since it violates condition (1). $< R1, Report, R2, R3 >$ is a history. □

To emphasize the fact that we view the database as an abstract data type and transactions as operations on this abstract data type, we define the term *semantic history* to distinguish it from the term history used in database literature (e.g., [3]).

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

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

From now on, we will deal with semantic histories only.

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

$$\begin{array}{l}
\textit{RefinedHotel} \\
\hline
res, tempreserved : \\
ST : Room \twoheadrightarrow Status;\ RM : Guest \rightarrowtail Room \\
guest : \ Guest;\ tempassigned : \ Room \\
\hline
\#RM + tempreserved = res \\
\mathrm{dom}(ST \quad \{Unavailable\}) = \mathrm{ran}\, RM \\
\qquad\qquad \cup tempassigned
\end{array}$$

$$\begin{array}{l}
R1 \\
\hline
\Delta RefinedHotel \\
\hline
res < total;\ res' = res + 1 \\
tempreserved' = tempreserved + 1 \\
ST' = ST;\ RM' = RM;\ guest' = guest \\
tempassigned' = tempassigned
\end{array}$$

$$\begin{array}{l}
R2 \\
\hline
\Delta RefinedHotel;\ r! : Room \\
\hline
ST(r!) = Available \\
ST' = ST \oplus \{r! \mapsto Unavailable\} \\
tempassigned' = tempassigned \cup \{r!\} \\
res' = res;\ tempreserved' = tempreserved \\
guest' = guest;\ RM' = RM
\end{array}$$

$$\begin{array}{l}
R3 \\
\hline
\Delta RefinedHotel;\ g? : Guest;\ r! : Room \\
\hline
tempreserved > 0 \\
r! \in tempassigned;\ g? \notin guest \\
RM' = RM \cup \{g? \mapsto r!\};\ res' = res \\
guest' = guest \cup \{g?\};\ ST' = ST \\
tempassigned' = tempassigned \setminus \{r!\} \\
tempreserved' = tempreserved - 1
\end{array}$$

$$\begin{array}{l}
RefinedCancel \\
\hline
\Delta RefinedHotel;\ g? : Guest \\
\hline
g? \in guest;\ res' = res - 1 \\
ST' = ST \oplus \{RM(g?) \mapsto Available\} \\
RM' = \{g?\} - RM;\ guest' = guest \setminus \{g?\} \\
tempreserved' = tempreserved \\
tempassigned' = tempassigned
\end{array}$$

$$\begin{array}{l}
RefinedReport \\
\hline
\Xi RefinedHotel \\
currentstatus! : Room \twoheadrightarrow Status \\
currentassignments! : Guest \rightarrowtail Room \\
\hline
tempassigned = \\
currentstatus! = ST \\
currentassignments! = RM
\end{array}$$

Figure 5: A Correct Decomposition for the Hotel Database

Example 2 An execution of the reserve transaction will be complete in a history H if all three steps $R1$, $R2$, and $R3$ of reserve appear in H. □

Definition 5 [Partial Semantic History] A semantic history H_p over $\mathbf{T}$ is a *partial* semantic history if the execution of some transaction T_i is not complete in H_p.

Definition 6 [Complete Semantic History] A semantic history H over $\mathbf{T}$ is a *complete* semantic history if the execution of each T_i in $\mathbf{T}$ is *complete*.

4.7 Consistent Execution Property

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

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

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

Definition 7 [Correct Partial Semantic History] A partial semantic history H_p is a *correct* partial semantic history if

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

Definition 8 [Correct Complete Semantic History] A complete semantic history H is a *correct* complete semantic history if

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

4.8 Complete Execution Property

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

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

In the hotel database suppose we have a partial semantic history H where $H =< R1, Report, R2 >$. where the reserve transaction executes with $g? =$ John. Consider step $R3$. The precondition $g? \notin guest$ of $R3$ requires that John not have a existing reservation, but it is possible that in the final state in H, John *is* an element of *guest*. We may cancel John's existing reservation, thereby allowing the reserve transaction to complete. First, the precondition of *Cancel*, $g? \in guest$, is guaranteed to hold if the precondition of $R3$ does not hold. Second, the postcondition of *Cancel* establishes the precondition of $R3$. Thus the reserve transaction for John can complete.

4.9 Decomposition with Deadlock

In this section, we show that some otherwise plausible decompositions do *not* satisfy the complete execution property, which is clearly undesirable. To illustrate the possibility, we modify the *Hotel* database as shown in figure 6.

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

Moreover, we introduce a new structure *clist* which keeps track of the guests whose cancellations are in progress. The guest whose reservation is being canceled is added to the *clist* in step $C1$ and is removed from the *clist* in step $C2$. We impose an additional constraint that a room cannot be reserved for a guest whose cancellation is in progress; note the precondition $g? \notin clist$ in step $Res3$. The reserve transaction is broken into steps $Res1$, $Res2$ and $Res3$, similar to $R1$, $R2$ and $R3$ of the *RefinedHotel* specification.

Consider the partial history H_p, where
$H_p =< C1, Res1, Res2 >$. Let the reserve and cancel transactions in H_p have the same $g?$ as their input. Also assume that $g? \notin guest$. We try to complete the reserve and cancel transactions. $Res3$ cannot be executed because the precondition $g? \notin clist$ is not satisfied because step $C1$ has inserted $g?$ in *clist*. $C2$ cannot be executed because the precondition $g? \in guest$ is not satisfied. It is possible to execute any number of steps of other transactions, but the reserve and cancel transactions in H_p still cannot complete.

The deadlock could be avoided by including the invariant $clist \subseteq guest$ in *DeadlockHotel*. Omission of this constraint allows the database to enter an undesirable state where $c? \in clist \wedge c? \notin guest$, from which neither $Res3$ nor $C2$ can complete.

5 Implementation

5.1 Successor Set Mechanism

Decomposing transactions into steps yields improved performance, but the interleaving of these steps must be constrained so as to avoid inconsistencies. In the decomposition we have given so far, which is based on generalizing invariants with auxiliary variables, the interleaving is constrained by additional preconditions on the auxiliary variables. Although the generalized invariants facilitate analysis, it is expensive to implement the auxiliary variables and to check the additional preconditions.

To avoid implementing auxiliary variables and to avoid checking additional preconditions, we propose two mechanisms: a *queuing mechanism* to ensure that steps of a transaction execute in order and a *successor set* mechanism somewhat similar to the notion of *compatibility sets* in [6] and *breakpoint sets* in [5]. The specification of a queuing mechanism is straightforward, and is therefore omitted. Implementation concerns for the queuing mechanism are considered later in the paper. Here we concentrate on the successor set mechanism.

We note that the semantics of successor sets differ substantially from that of the analogous notions in [6] and in [5]. Also, via specific decomposition steps and corresponding proof obligations, we assist the specifier in verifying the correctness of successor sets with respect to the original specification; in [6] and in [5], the burden of arguing correctness rests entirely with the application developer.

Definition 9 [Successor Set] The *successor set* of a step T_{ij}, denoted $SC(T_{ij})$, is the set of steps that can appear after step T_{ij} and before step $T_{i(j+1)}$ in any correct semantic history.

Example 3 Successor set descriptions are obtained by examining the preconditions with auxiliary variables. Some of these preconditions will always be satisfied if the steps in each transaction are executed in order; the queuing mechanism guarantees the satisfaction of these preconditions. In the hotel example, the only remaining precondition with auxiliary variables is $tempassigned =$ in *RefinedReport*. We observe that this precondition will always be satisfied as long as *RefinedReport* is not executed between steps $R2$ and $R3$ of a reserve transaction. Thus we specify the successor sets as follows.

$$\begin{array}{l}
\textit{DeadlockHotel} \\
\hline
res, tempreserved, tempcanceled : \\
RM : Guest \quad Room;\ ST : Room \nrightarrow Status \\
guest, clist : \quad Guest;\ tempassigned : \quad Room \\
\hline
\#RM = res - tempreserved \\
\qquad\qquad + tempcanceled \\
\mathrm{dom}(ST \quad \{Unavailable\}) = \mathrm{ran}\, RM \\
\qquad\qquad \cup tempassigned
\end{array}$$

$$\begin{array}{l}
\textit{Res1} \\
\hline
\Delta DeadlockHotel \\
\hline
res < total;\ res' = res + 1 \\
tempreserved' = tempreserved + 1 \\
tempcanceled' = tempcanceled \\
ST' = ST;\ RM' = RM \\
clist' = clist;\ guest' = guest \\
tempassigned' = tempassigned
\end{array}$$

$$\begin{array}{l}
\textit{C1} \\
\hline
\Delta DeadlockHotel;\ g? : Guest \\
\hline
res > 0;\ res' = res - 1 \\
clist' = clist \cup \{g?\};\ ST' = ST \\
tempcanceled' = tempcanceled + 1 \\
tempreserved' = tempreserved \\
tempassigned' = tempassigned \\
guest' = guest;\ RM' = RM
\end{array}$$

$$\begin{array}{l}
\textit{Res2} \\
\hline
\Delta DeadlockHotel;\ r! : Room \\
\hline
ST(r!) = Available \\
ST' = ST \oplus \{r! \mapsto Unavailable\} \\
tempassigned' = tempassigned \cup \{r!\} \\
res' = res;\ guest' = guest;\ clist' = clist \\
RM' = RM;\ tempreserved' = tempreserved \\
tempcanceled' = tempcanceled
\end{array}$$

$$\begin{array}{l}
\textit{C2} \\
\hline
\Delta DeadlockHotel;\ g? : Guest \\
\hline
g? \in guest;\ g? \in clist;\ tempcanceled > 0 \\
ST' = ST \oplus \{RM(g?) \mapsto Available\} \\
RM' = \{g?\} - RM;\ guest' = guest \setminus \{g?\} \\
clist' = clist \setminus \{g?\};\ res' = res \\
tempcanceled' = tempcanceled - 1 \\
tempreserved' = tempreserved \\
tempassigned' = tempassigned
\end{array}$$

$$\begin{array}{l}
\textit{Res3} \\
\hline
\Delta DeadlockHotel;\ g? : Guest;\ r! : Room \\
\hline
tempreserved > 0;\ r! \notin \mathrm{ran}\, RM \\
g? \notin guest;\ g? \notin clist \\
guest' = guest \cup \{g?\};\ clist' = clist \\
RM' = RM \cup \{g? \mapsto r!\};\ res' = res \\
tempreserved' = tempreserved - 1 \\
tempassigned' = tempassigned \setminus \{r!\} \\
tempcanceled' = tempcanceled;\ ST' = ST
\end{array}$$

Figure 6: Example Specification Lacking Complete Execution Property

$SC(R1) = \{R1, R2, R3, \mathit{RefinedCancel}, \mathit{RefinedReport}\}$
$SC(R2) = \{R1, R2, R3, \mathit{RefinedCancel}\}$

The successor set for $R1$ includes every other possible step; it is possible to execute an $R1$, and then execute any of another $R1$, an $R2$ (of the same or of a different reserve transaction), an $R3$ (of a different reserve transaction), a *RefinedCancel*, or a *RefinedReport*. The successor set for $R2$ is more restrictive. $\mathit{RefinedReport} \notin SC(R2)$ means that the *RefinedReport* step cannot execute after step $R2$. In other words, *RefinedReport* is not allowed to the see the inconsistencies with respect to the original invariants that are introduced by step $R2$.

Note that with the queuing mechanism and the successor set description, the auxiliary variables themselves need not be implemented, and all constraints on the auxiliary variables may be ignored. □

Any complete semantic history generated using successor sets must meet an additional requirement to those given in Definition 8:

> Consider every pair of steps T_{ij} and $T_{i(j+1)}$ of transaction T_i. For every step T_{kl} of a different transaction T_k appearing between T_{ij} and $T_{i(j+1)}$ (if any), $T_{kl} \in SC(T_{ij})$.

Successor set descriptions are intended to allow the removal of predicates that reference auxiliary variables, and ultimately the auxiliary variables themselves. Therefore, with respect to the specifications given with generalized invariants and auxiliary variables, not all successor set descriptions are correct. Informally, a successor set is correct with respect to a generalized invariant specification if any semantic history generated using successor sets 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 correct 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 semantic history generated by S_2 is also a correct semantic history generated by S_1.
2. S_2 satisfies the complete execution property.

Note that since the correct semantic histories of S_2 might be a proper subset of those of S_1, the complete execution property needs to be explicitly verified with respect to S_2. The appendix contains a proof of the valid successor set property for the hotel example.

5.2 Two-Phase Locking Mechanism

For the semantic-based decomposition given in this paper to be useful, the specifications must be implementable with a standard database concurrency control mechanism, such as two-phase locking or timestamp ordering. The notions of interleaving described in [6] and [5] are both implemented in a locking environment. Since our interleaving mechanism differs from those of [6, 5], we sketch a two-phase locking implementation of our mechanism. Other implementations are possible, as are various optimizations.

By treating steps as complete transactions, two-phase locking can guarantee that any history is stepwise conflict serializable (i.e., conflict serializable with steps as primitive operations). However, stepwise serializability is not sufficient, since some stepwise serial histories violate the successor set description. Specifically, step T_{ij} of one transaction might serialize between steps T_{km} and $T_{k(m+1)}$ of another transaction, even if the T_{ij} is not in the successor set of T_{km}.

To ensure that the successor set descriptions are met, we introduce a control mechanism for dispatching of steps to the data manager, which is responsible for implementing two-phase locking. We assume that the data manager produces an explicit prefix of a serialization order for all steps that commit.

Consider a set of steps under the control of the data manager. Since the data manager may serialize these steps in any order, we require that the concatenation of the committed prefix of the serialization order and any order of the steps in the data manager follow the successor set rules. Additional steps can be submitted to the data manager only if doing so maintains this constraint, which we refer to as the *order constraint*. The order constraint can be checked by ensuring that each active step is an element of the intersection of the successor sets of all other active steps and also of all committed steps of incomplete transactions. (Note that completed transactions do not affect the interleaving of active transactions.)

The control mechanism is as follows. For each active transaction, a queue of the steps in that transaction is maintained. The next step in a transaction can be dispatched to the data manager if the previous step has committed and the dispatch does not violate the order constraint. The data manager ensures that all histories are stepwise serializable, and the control mechanism ensures that such histories also satisfy the successor set constraints. Conversely, any stepwise serial history that satisfies the successor set description is achievable.

6 Conclusion

In this paper, we have provided the database application developer 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 guarantees that the resulting system possesses the desired properties.

As we have indicated, the syntactic aspect of the implementation given in section 5.2 (i.e., stepwise conflict-serializability via two-phase locking), is preliminary. We anticipate developing a more efficient implementation, as was done in [6] and [5]. We also must address the problem of reliably transmitting parameters between steps of a transaction, a problem that is considered in [7, 19]. However, the semantic aspects of our implementation have been thoroughly addressed.

We can easily permit ad hoc transactions to be dynamically added in our model, although they will require some special intervention. An ad hoc transaction could be executed as an atomic, sensitive transaction, which means that all integrity constraints relevant to the calculation of any output will have to be included as explicit preconditions for the transaction (see section 4.4). Alternatively, an ad hoc transaction could be included at the successor set stage by simply excluding it from all successor set descriptions. Deletion of a transaction in our model is somewhat problematic since deletion may impact the complete execution property, which ensures that any transaction that has been partially executed can eventually complete.

References

[1] D. Agrawal, A. El Abbadi, and Ambuj K. Singh. Consistency and orderability: Semantics-based correctness criteria for databases. *ACM Transactions on Database Systems*, 18(3):460–486, September 1993.

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

[3] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[4] W. Du and A.K. Elmagarmid. Quasi serializability: a correctness criterion for global concurrency control in interbase. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 347–355, 1990.

[5] A. A. Farrag and M. T. Ozsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

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

[7] Hector Garcia-Molina and Kenneth Salem. Services for a workflow management system. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 17(1):40–44, March 1994.

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

[9] Maurice P. Herlihy and William E. Weihl. Hybrid concurrency control for abstract data types. *Journal of Computer and System Sciences*, 43(1):25–61, August 1991.

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

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

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

[13] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions.* Morgan Kaufmann Publishers, San Mateo, CA, 1994.

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

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

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

[17] Dennis Shasha, Eric Simon, and Patrick Valduriez. Simple rational guidance for chopping up transactions. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 298–307, San Diego, CA, June 1992.

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

[19] Helmut Wachter and Andreas Reuter. The contract model. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219–263. Morgan Kauffman, 1992.

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

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