

## Appendix A – Z Notation

$\mathbb{N}$	Set of Natural Numbers
$\mathbb{P}A$	Powerset of Set $A$
$\#A$	Cardinality of Set $A$
$\setminus$	Set Difference (Also schema ‘hiding’)
$A \circ B$	Forward Composition of $A$ with $B$
$x \mapsto y$	Ordered Pair $(x, y)$
$A \mapsto B$	Partial Function from $A$ to $B$
$A \mapsto\!\!\!\rightarrow B$	Partial Injective Function from $A$ to $B$
$B \triangleleft A$	Relation $A$ with Set $B$ Removed from Domain
$A \triangleright B$	Relation $A$ with Range Restricted to Set $B$
$\text{dom } A$	Domain of Relation $A$
$\text{ran } A$	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
$\Delta A$	Before and After State of Schema $A$
$\Xi A$	$\Delta A$ with No Change to State

Table 2: Relevant Z Notation

## Appendix B – Proofs of the Theorems

### Proofs of Theorems from Section 3

**Theorem 1** If two histories  $H$  and  $H'$  are conflict equivalent, then the histories are semantic equivalent. The converse is not, generally, true.

**Proof 1** Suppose  $H$  and  $H'$  are conflict equivalent. To prove that they are semantic equivalent, we must show that they satisfy all the three conditions given in Definition 9. Condition 1 is satisfied because conflict equivalence also requires the same criterion. To show that  $H$  and  $H'$  satisfy conditions 2 and 3, we make use of the following property of conflict equivalent histories: the histories  $H$  and  $H'$  have the same reads from relationships and the same final writes [7]. That is, if  $w_i[x]$  is the final write in history  $H$ , then  $w_i[x]$  is also the final write in history  $H'$ . If  $T_i$  reads  $x$  from  $T_j$  in  $H$ , then  $T_i$  reads  $x$  from  $T_j$  in  $H'$ . The value written by a transaction  $T_i$  on a data item  $x$  depends on the the values of the data items read by  $T_i$ . Thus, the value written by a final write on some data item  $x$  is identical in both the histories. Thus, if the two histories  $H$  and  $H'$  are executed on the same initial state, they will produce the same final state. Condition 2 of Definition

9 is therefore satisfied. The output produced by a transaction  $T_i$  depends on the values of data items it reads. If  $H$  and  $H'$  are conflict equivalent, a transaction  $T_i$  in  $H$  will have the same reads from relation as the transaction  $T_i$  in  $H'$ . Thus, the output produced by any transaction  $T_i$  is the same in both the histories and condition 3 of Definition 9 is satisfied.

To show that the converse is not, generally, true, consider the following two transactions  $T_1$  and  $T_2$ :

$$T_1 = r[x]; x = x - 100; w[x]; r[y]; y = y + 100; w[y];$$

$$T_2 = r[y]; y = y - 50; w[y]; r[x]; x = x + 50; w[x];$$

Consider the following two histories  $H''$  and  $H'''$  that consists of operations of  $T_1$  and  $T_2$  (we do not show the internal operations of the transactions):

$$H'' = r_1[x]; w_1[x]; r_2[y]; w_2[y]; r_2[x]; w_2[x]; r_1[y]; w_1[y];$$

$$H''' = r_1[x]; w_1[x]; r_1[y]; w_1[y]; r_2[y]; w_2[y]; r_2[x]; w_2[x];$$

$H''$  and  $H'''$  are not conflict-equivalent because they do not order conflicting operations in the same way. The two histories are semantic equivalent because they produce the same state and output when executed on the same initial state.

**Theorem 2** The committed projection of a history  $H$  consisting of a set of well-formed two-phase transactions is policy-compliant.

**Proof 2** Assume that the history  $H$  is not policy-compliant. This means that one or more transactions in the history  $H$  are not policy-compliant. Suppose  $T_i$  is one such transaction. Without loss of generality, assume that the transaction  $T_i$  does not have write access to an object  $O_r$  but nevertheless updates object  $O_r$ . We show that this cannot happen. Since  $T_i$  is well-formed, it will deploy the appropriate policy object before it performs the update. Moreover, before the deploy operation can take place,  $T_i$  has to obtain the deploy lock for a policy object  $\alpha$  that authorizes  $T_i$  to update  $O_r$ . Thus, when  $T_i$  initially accessed  $O_r$ , there was a policy object  $\alpha$  that allowed  $T_i$  to update  $O_r$ . So, the only possibility is that while  $T_i$  was updating  $O_r$ , the policy object  $\alpha$  got deleted or modified. But according to well-formed rules this is not possible. Any transaction  $T_j$  modifying the policy object  $\alpha$  has to obtain a write lock ( $WL$ ) on  $\alpha$ . Before the write lock on  $\alpha$  can be granted, the transaction  $T_i$  holding the deploy lock ( $DL$ ) has to be aborted and the deploy lock released. Thus, the above scenario of  $T_i$  updating  $O_r$  without any policy authorizing  $T_i$  to do so, does not arise in our case. Therefore,  $T_i$  is policy-compliant. Our assumption, that the history  $H$  is not policy-compliant, is wrong.

**Theorem 3** The committed projection of a history  $H$  consisting of a set of well-formed two-phase transactions is conflict serializable.

**Proof 3** We prove this by contradiction. Assume that the history  $H$ , produced by transactions  $\{T_1, T_2, \dots, T_n\}$ , is not conflict serializable. Then the serialization graph [7] produced from this history contains a cycle. Without loss of generality, assume that this cycle is  $T_1 \rightarrow T_2 \rightarrow T_3 \dots T_n \rightarrow T_1$ . The presence of the arrow  $T_1 \rightarrow T_2$ , signifies that there is an operation in  $T_1$  that conflicts with and precedes another operation in  $T_2$ . The unlock operation in  $T_1$  must precede the lock operation in  $T_2$  (this is because the data object involved in a conflicting operation can be locked by only one transaction at any time). That is,  $U_1(O_a) < L_2(O_a)$ . Using similar arguments, we can argue that for the edge  $T_2 \rightarrow T_3$ , there is an unlock operation in  $T_2$  that precedes a lock operation in  $T_3$ . That is,  $U_2(O_b) < L_3(O_b)$ . Since transaction  $T_2$  is two-phase,  $L_2(O_a) < U_2(O_b)$ . Therefore, we can conclude that  $U_1(O_a) < L_3(O_b)$ . This argument can be extended and we can arrive at the conclusion that  $U_1(O_a) < L_1(O_k)$ . This is not possible because  $T_1$  is two-phase. Thus, we arrive at a contradiction. Hence, our initial assumption that the history is not conflict serializable is wrong. Therefore, the history  $H$  is conflict serializable.

## Proofs of Theorems from Section 4

**Theorem 4** The committed projection of a history  $H$  generated by this mechanism is policy-compliant.

**Proof 4** Assume that the history  $H$  is not policy-compliant. This means that one or more transaction in the history  $H$  is not policy-compliant. Suppose transaction  $T_i$  is not policy-compliant. Without loss of generality, assume that the transaction  $T_i$  does not have write access to an object  $O_r$  but nevertheless updates object  $O_r$ . We show that this cannot happen. Since  $T_i$  satisfies all but Condition 4 of the Definition 6, it will deploy the appropriate policy object before it performs the update. Moreover, before the deploy operation can take place,  $T_i$  has to obtain the deploy lock for the policy object. In other words, before  $T_i$  can access  $O_r$ , it has to obtain the deploy lock for a policy object  $\alpha$  that authorizes  $T_i$  to update  $O_r$ . Thus, when  $T_i$  initially accessed  $O_r$ , there was a policy object  $\alpha$  that allowed  $T_i$  to update  $O_r$ . So, the only possibility is that while  $T_i$  was updating  $O_r$ , the policy object  $\alpha$  got modified such that  $T_i$  no longer has the privilege to update  $O_r$ . But this scenario cannot occur according to our algorithm. Any transaction  $T_j$  modifying the policy object  $\alpha$  has to obtain a write lock ( $WL$ ) on  $\alpha$ . Before the write lock on  $\alpha$  can be granted, the transaction  $T_i$  holding the deploy lock ( $DL$ ) has to be aborted and the deploy lock released (because  $T_j$  re-

stricts the policy). Thus, the above scenario of  $T_i$  updating  $O_r$  without any policy authorizing  $T_i$  to do so, does not arise in our case. Therefore,  $T_i$  is policy-compliant. Our assumption, that the history  $H$  is not policy-compliant, is wrong.

**Theorem 5** The committed projection of a history  $H$  generated by this mechanism is serializable.

**Proof 5** The only change from the algorithms given in Section 3 is with regards to a transaction (say,  $T_i$ ) being able to acquire a write lock on a policy object (say,  $\alpha$ ), while another transaction (say,  $T_j$ ) holds a deploy lock on the same policy object. That is, the only time two conflicting locks are acquired simultaneously, is when one transaction  $T_j$  already has a deploy lock on a policy object and another transaction  $T_i$  acquires a write lock on the policy object for the purpose of relaxing the policy. This allows a policy relaxation transaction to be executed between the operations of a transaction deploying the policy. Without loss of generality assume that  $H$  is one such history containing the transactions  $T_i$  and  $T_j$  mentioned above:  $H = d_j[\alpha]; r_j[O_a]; d_i[\beta]; w_i[\alpha]; w_j[O_b]; c_j; c_i$ . We permit this history because the policy relaxation operation does not conflict with the deploy operation and can take place concurrently. The effect will be equivalent to executing the transaction that is deploying the policy followed by the transaction that is relaxing the policy. In other words, the history  $H$  is semantic equivalent to the following serial history  $H'$ , where  $H' = d_j[\alpha]; r_j[O_a]; w_j[O_b]; c_j; d_i[\beta]; w_i[\alpha]; c_i; .$  Thus, allowing a transaction to hold onto a deploy lock on a policy object while another transaction acquires a write lock on the same policy object in order to relax the policy, still produces serializable histories.

**Theorem 6** The concurrency control algorithm given in Section 4 provide more concurrency than the one given in Section 3.

**Proof 6** Let **H1** and **H2** be the set of all possible histories generated by the locking rules given in Section 3 and Section 4 respectively. We need to prove that **H1** is a proper subset of **H2**, that is, **H1**  $\subset$  **H2**. The proof will proceed in two parts: (i) First, we will prove that for any history  $H_1$ , if  $H_1 \in \mathbf{H1}$ , then  $H_1 \in \mathbf{H2}$ . (ii) Next, we will prove that for some history  $H_2$  where  $H_2 \in \mathbf{H2}$ ,  $H_2 \notin \mathbf{H1}$ . In the following paragraphs we outline the two proofs.

Proof of (i): Let  $H_1 \in \mathbf{H1}$ . We need to prove that  $H_1 \in \mathbf{H2}$ . Assume that  $H_1 \notin \mathbf{H2}$ . This is possible only if there is some operation in  $H_1$  that cannot be scheduled by the concurrency control rules of Section 4. In other words, the locking rules of Section 4 prohibit obtaining locks necessary for this operation. For ordinary data objects, the locking rules are the same for both the approaches. So the only possibility is that this is an operation on a policy object. Suppose this a Write operation. The locking rules prevents obtaining a write lock only when some other transactions have a read

lock or write lock on the policy object. But in this case the locking rules given in Section 3 would have also disallowed the write lock and hence the Write operation in  $H_1$ . Thus, the operation is not a Write operation. Similar arguments can be made for the Deploy and Read operations. Hence any operation in  $H_1$  will also be permitted by the locking rules of Section 4. In other words  $H_1 \in \mathbf{H2}$ .

Proof of (ii): We need to show that for some  $H_2 \in \mathbf{H2}$ ,  $H_2 \notin \mathbf{H1}$ . Let  $H_2 = d_i[\alpha]; r_i[O_a]; d_j[\beta]; w_j[\alpha]; w_i[O_b]; c_i; c_j$ .  $H_2$  is a history generated by interleaving the operations of two transactions  $T_i$  and  $T_j$ , where  $T_i = d_i[\alpha]; r_i[O_u]; w_i[O_v]; c_i$  and  $T_j = d_j[\beta]; w_j[\alpha]; c_j$ .  $T_i$  Reads and Writes the data objects  $O_u$  and  $O_v$  respectively.  $T_i$  executes by virtue of policy  $\alpha$ .  $T_j$  updates policy  $\alpha$  by performing a policy relaxation operation  $w_j(\alpha)$ .  $T_j$  executes due to the privileges given by policy  $\beta$ . Let us see how the concurrency control mechanism in Section 4 executes  $H_2$ . First, a deploy lock will be obtained on policy object  $\alpha$  and the deploy operation performed by transaction  $T_i$ . Then, a read lock will be obtained on object  $O_a$  and the read operation performed by  $T_i$ . After that,  $T_j$  will acquire a deploy lock on policy object  $\beta$  and perform the deploy operation. Then,  $T_j$  will acquire a write lock on policy object  $\alpha$  to perform the write operation. Since this is a policy relaxation, the write lock will be granted without aborting  $T_i$ . After this, the write operation is performed. Continuing in this manner, the concurrency control algorithm in Section 4 will be able to complete this history. Since the history  $H_2$  can be generated by the locking rules given in Section 4,  $H_2 \in \mathbf{H2}$ . However,  $H_2$  cannot be generated by the locking rules given in Section 3. This is because before the operation  $w_j[\alpha]$  can take place, the locking protocol must obtain the write lock on  $\alpha$ . Before the write lock can be obtained, transaction  $T_i$  must be aborted. Therefore,  $H_2 \notin \mathbf{H1}$ .

## Proofs of Theorems in Section 5

**Theorem 7**  $Reserve \circ RemoveRoleManager \Leftrightarrow RemoveRoleManager \circ Reserve$

**Proof 7** L.H.S. on simplification yields the following schema

$$\Delta Hotel; t? : GUEST; r? : ROOM; i? : USER; p? : \mathbb{P}(POLICY)$$

$$Supervisor \in UserRole(i?)$$

$$\exists P1 : POLICY \bullet P1 \in Access \wedge Supervisor \in Role(P1) \\ \wedge Status \in Object(P1) \wedge \{r, w\} \subseteq Right(P1)$$

$$\exists P2 : Policy \bullet P2 \in Access \wedge Supervisor \in Role(P2) \\ \wedge Assign \in Object(P2) \wedge \{r, w\} \subseteq Right(P2)$$

$$Status(r?) = Available; p? \in Access; Manager \subseteq Role(p?)$$

$$Role' = Role \oplus \{p? \mapsto Role(p?) - \{Manager\}\}$$

$$Status' = Status \oplus \{r? \mapsto Taken\}; Assign' = Assign \cup \{r? \mapsto t?\}$$

$$Access' = Access; Object' = Object; Right' = Right; UserRole' = UserRole$$

The R.H.S. on simplification also yields the same schema.

**Theorem 8** The committed projection of history  $H$  generated by this mechanism is policy-compliant.

**Proof 8** This can be proved in a manner similar to Theorem 4.

**Theorem 9** The committed projection of history  $H$  generated by this mechanism is serializable.

**Proof 9** The change from the algorithm given in Section 3 is that we allow a transaction to acquire a write lock on a policy object while other transactions hold deploy locks on the same policy object. Note that, this is allowed only when the other transactions holding the deploy locks can commute with the transaction updating the policy. In other words, we allow conflicting locks to be held simultaneously only when the operations do not conflict. In such cases the policy update transaction being interleaved between other transactions that are deploying the policy has the same effect as executing the policy update transaction serially after the transactions that have deployed the policy.

## Proofs of Theorems from Section 6

**Theorem 10** The committed projection of history  $H$  generated by this mechanism are policy-compliant.

**Proof 10** This can be proved in a manner similar to Theorem 4.

**Theorem 11** If the application has been decomposed such that it produces correct semantic histories, then the concurrency control mechanism generates correct stepwise serializable histories.

**Proof 11** The application has been decomposed such that it produces correct semantic histories. We need to prove that our mechanism produces correct stepwise serializable histories. First, let us consider the case when the commute set of each step of any policy update transaction is empty. In such cases, our mechanism produces histories that are conflict equivalent to correct semantic histories. Next, suppose that the commute sets are not empty. In such cases, the mechanism may produce histories that are not conflict equivalent to correct semantic histories. This happens because it allows a step of policy update transaction to acquire a write lock while steps of other transactions may hold deploy locks. Because of this operations of a step of policy update transaction can interleave with operations of steps of transactions that are deploying this policy. This interleaving is allowed only when the steps of policy update transaction and steps of transactions commute and do not conflict. In such cases, the effect of the interleaving is the same as that of executing the step of the policy update transaction after the step of the transaction deploying this policy. Thus, all histories produced are semantic equivalent to correct semantic histories.

## Appendix C – Properties for the Example

### The Consistent Execution Property

The consistent execution property requires that if we execute a complete semantic history  $H$  on an initial state that satisfies the original integrity constraints, then the final state also satisfies the original integrity constraints.

For the hotel database, the original integrity constraints are satisfied when  $acquired = \emptyset$  and  $released = \emptyset$ . Thus we have to prove that if the initial state of a complete semantic history satisfies  $acquired = \emptyset$  and  $released = \emptyset$ , the final state of the history will also satisfy  $acquired = \emptyset$  and  $released = \emptyset$ .

Let  $n_1, n_2, n_3, n_4, cn_1, cn_3$  be the number of steps of type  $Res1, Res2, Can1, Can2, CompRes1$  and  $CompCan1$  respectively present in any complete history. The variable  $acquired$  is modified by steps of type  $Res1, Res2$  and  $CompRes1$ .  $Res1$  adds a room to the set  $acquired$ . This room is taken out in step  $Res2$  or in compensating step  $CompRes1$ . Thus,  $|acquired| = n_1 - (n_2 + cn_1) \dots (1)$ . In a complete history all reserve transactions have completed execution. Corresponding to each step of type  $Res1$  in the history, there is either a step of type  $Res2$  or a compensating step of type  $CompRes1$ . Thus,  $n_1 = n_2 + cn_1 \dots (2)$ . From (1) and (2),  $|acquired| = 0$ . In other words  $acquired = \emptyset$ . Using similar arguments we can prove that,  $released = \emptyset$ . Thus, in a complete history the final state satisfies  $acquired = \emptyset$  and  $released = \emptyset$ .

## The Sensitive Transaction Isolation Property

The sensitive transaction isolation property requires that all output data produced by sensitive transactions should have the appearance that the sensitive transactions are executed on a consistent database state even though they may be running on an inconsistent database state.

In the hotel database, we have only one sensitive transaction *Report*. The sensitive transaction isolation property is achieved by construction. We compute the subset of the integrity constraints that must hold for the precondition for *Report*. We obtain these from *Hotel* by hiding the state variables not involved in producing the output of *Report*. Since there are no such state variables, the schema obtained is the same as *Hotel*. In other words the original integrity constraint must be satisfied for producing consistent output in *Report*:

$$\text{dom}(\text{Status} \triangleright \{ \text{Taken} \}) = \text{dom}(\text{Assign})$$

This constraint is implied by the generalized integrity constraint:

$$\text{dom}(\text{Status} \triangleright \{ \text{Taken} \}) \cup \text{released} = \text{dom}(\text{Assign}) \cup \text{acquired}$$

when  $\text{acquired} = \text{released}$ . Since the sets *acquired* and *released* are disjoint, we include preconditions  $\text{acquired} = \emptyset$  and  $\text{released} = \emptyset$  as preconditions for *Report*.

## The Semantic Atomicity Property

The semantic atomicity property requires that all partially executed transactions will complete.

In the hotel database, we need to prove that any partial semantic history  $H_p$  defined over one or more incomplete transactions of type *Reserve*, *Cancel* is a prefix of a complete history defined over the same transactions.

Consider a partial semantic history  $H_p$  containing an incomplete *Reserve* transaction. In other words *Res1* has been executed but not *Res2* or *CompRes1*. First, we check whether *Res2* can execute and complete the execution of *Reserve*. *Res2* has preconditions that check whether a user executing this transaction is a *Supervisor* and whether there exists a policy that allows the *Supervisor* to read and write the object *Assign*. These preconditions will be violated when the policy authorizing the supervisor to execute this transaction is deleted or modified. Now let us see whether this transaction can be completed by executing a *CompRes1*. *CompRes1* has preconditions that check whether a user executing this transaction is a *Supervisor* and whether there exists a policy that allows the *Supervisor* to read and write the object *Status*. Here again, the execution of a policy update transaction, say *RemoveRoleSuper*, may violate the precondition of *CompRes1*.

<i>ModCompRes1</i>	<i>ModCompCan1</i>
$\Delta HotelD; r? : ROOM$	$\Delta HotelD; r? : ROOM$
$i? \in Supervisor \cup SecurityOfficer$	$i? \in Supervisor \cup SecurityOfficer$
$\exists P1 : POLICY \bullet P1 \in Access$	$\exists P1 : POLICY \bullet P1 \in Access$
$\wedge (Supervisor \in Subject(P1)$	$\wedge (Supervisor \in Subject(P1)$
$\vee SecurityOfficer \in Subject(P1))$	$\vee SecurityOfficer \in Subject(P1))$
$\wedge Status \in Object(P1)$	$\wedge Status \in Object(P1)$
$\wedge \{r, w\} \subseteq Right(P1)$	$\wedge \{r, w\} \subseteq Right(P1)$
$Status' = Status \oplus \{r? \mapsto Available\}$	$Status' = Status \oplus \{r? \mapsto Taken\}$
$acquired' = acquired \setminus \{r?\}$	$released' = released \setminus \{r?\}$
$Subject' = Subject$	$Subject' = Subject$
$Assign' = Assign; Access' = Access$	$Access' = Access; Assign' = Assign$
$Object' = Object; Right' = Right$	$Object' = Object; Right' = Right$

Figure 7: Modified Compensating Steps to Ensure Semantic Atomicity

In other words, the *Reserve* transaction cannot be completed in some scenarios and the application does not have semantic atomicity property.

### Modification of the Application

To ensure semantic atomicity, we must modify the application. One possible way is to introduce another role, called *SecurityOfficer*, and change the specification of the compensating steps so that *SecurityOfficer* always can execute the compensating steps. The specification of the modified compensating steps is given in Figure 7. Also, the application must not have a policy update transaction that removes privileges associated with the role *SecurityOfficer*.

### The Policy-Compliant Property

The policy-compliant property requires that every semantic history generated from the application is policy-compliant.

Suppose  $H$  is a semantic history generated from the hotel database. Assume that  $H$  is not policy-compliant. In other words, there is a step  $T_{ij}$  in  $H$  that is not policy-compliant. We show that this situation is not possible. Whenever a step or a compensating step is specified, we impose additional preconditions that check whether the user initiating the step has the privilege for executing the operations in a step. For instance, in the specification for *Res1* (Figure 4) we include preconditions that ensure that *Res1* is executed only by the *Supervisor* and there exists policies that give the *Supervisor* read and write privileges on the object *Status*. Similar preconditions are attached to all steps and compensating steps of the hotel database. Note that a step  $T_{ij}$  can be executed only if all its preconditions are satisfied. So whenever  $T_{ij}$  begins execution, it is policy-

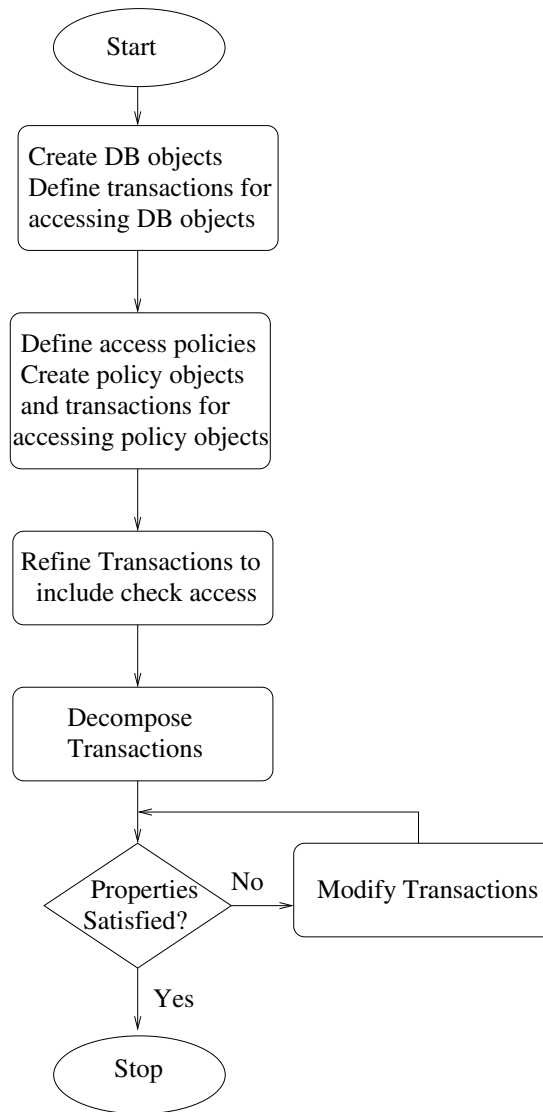


Figure 8: Flow Chart of the Process

compliant. The only possibility is that during the execution of  $T_{ij}$ , the policy authorizing  $T_{ij}$  to perform the operations might be modified. This possibility is avoided because in a semantic history (which is also a stepwise serial history) steps are executed atomically. So if  $T_{ij}$  was policy-compliant when it started execution, it will be so when it completes execution. Thus, step  $T_{ij}$  not being policy-compliant does not arise, which implies that the semantic history  $H$  is also policy-compliant.

## Flow Diagram of the Process

The application development process begins by creating the database objects related to the application and defining transactions for accessing these objects. The next step is to protect the database objects from unauthorized access. To address this issue, we define access control policies for the

application. The access control policies specify who has access to which data and also the kind of access. The access control policies are stored in the form of policy objects. Since policy objects are accessed through transactions, we need transactions for operating on policy objects.

Having defined the database objects, the policy objects, and the transactions for accessing them, the next step is to refine the transactions to include access control checks. These include adding preconditions to the transactions to ensure that only authorized users initiate the transactions and execute them.

In the fourth step, we decompose the transactions to improve the performance. When the transactions are decomposed, the traditional ACID properties are no longer satisfied. To address this, we propose a set of replacement properties. The application must be analyzed to check whether the replacement properties are satisfied. If not, then the application has to be modified and the process of property verification must be repeated. Once the application satisfies the desirable properties, it can be executed by our concurrency control mechanism that generates policy-compliant and correct stepwise serializable histories.