Applying Semantic Knowledge to Real-Time Update of Access Control Policies

Abstract

Real-time update of access control policies, that is, updating policies while they are in effect and enforcing the changes immediately, is necessary for many security-critical applications. In this paper, we consider real-time update of access control policies in a database system. Updating policies while they are in effect can lead to potential security problems, such as, access to database objects by unauthorized users. In this paper, we propose several algorithms that not only prevent such security breaches but also ensure the correctness of execution. The algorithms differ from each other in the degree of concurrency provided and the semantic knowledge used.

Of the algorithms presented, the most concurrency is achieved when transactions are decomposed into atomic steps. Once transactions are decomposed, the atomicity, consistency, and isolation properties no longer hold. Since the traditional transaction processing model can no longer be used to ensure the correctness of the execution, we use an alternate semantic-based transaction processing model. To ensure correct behavior, our model requires an application to satisfy a set of necessary properties, namely, semantic atomicity, consistent execution, sensitive transaction isolation, and policy-compliant. We show how one can verify an application statically to check for the existence of these properties.

1 Introduction

Security policy of an application is often subject to change. Some of these applications may require *real-time update* of security policies. We use the term real-time update of a policy to mean that the policy is changed while it is in effect and this change needs to be enforced immediately. Examples of applications requiring real-time policy updates are those that are responding to international crisis, such as relief or war efforts. Often times in such scenarios, system resources need reconfiguration or operational modes require change; this, in turn, necessitates policy changes. Real-time policy updates are also needed in pervasive computing applications where the change of environment requires update of policy controlling the access and immediate enforcement of the modified policies.

We use a simple example to motivate the need for real-time updates of policies. Suppose user $John$, by virtue of some policy P, can execute a long-duration transaction that prints a large volume of sensitive financial information stored in file I. While $John$ is executing this transaction, an insider threat is suspected and the policy P is changed such that $John$ no longer has the privilege to execute this transaction. Since existing access control mechanisms check $John$'s privileges *before*

John initiates the transaction and not *during* the execution of the transaction, the updated policy P will not be correctly enforced. In this case, the policy was updated correctly but not enforced immediately thereby resulting in a security breach and causing financial loss to the company.

In this paper we consider real-time policy updates in the context of a database system. A database consists of a set of objects that are accessed through transactions. Transactions performing operations on database objects must have the privileges to execute those operations. Such privileges are specified by access control policies; access control policies are stored in the form of *policy objects*. Transactions executing by virtue of the privileges given by a policy object is said to *deploy* the policy object. In addition to being deployed, a policy object can also be read and modified by transactions. Since we allow real-time updates of policy, a policy object can be modified while it is deployed. Modification to deployed policies pose new problems. For example, transactions executing by virtue of the privileges given by the deployed policy may no longer enjoy those privileges. In such scenarios, allowing the transaction to execute may result in a security breach. In this work we provide algorithms that ensure such problems do not arise.

The first algorithm is a very simple one – any time a deployed policy is modified, transactions executing by virtue of that policy are aborted. This algorithm, although easy to implement, results in unnecessary aborts. For example, the update of policy may not be restricting privileges or may be removing privileges that does not affect the transaction that deploys this policy. Since we are dealing with critical transactions, aborting them unnecessarily is not desirable.

The second algorithm uses the semantics of the policy update transactions to minimize such aborts. In this approach, the policy update transactions are categorized as policy relaxations or policy restrictions. Policy relaxations do not decrease the access control privileges of any subject. A policy update that is not a policy relaxation is treated as a policy restriction. Policy relaxation, unlike restriction, does not require abort of transactions that are executing by virtue of the policy.

Often times, a policy restriction may not be problematic for a transaction that is deploying this policy. This situation occurs when the restriction does not decrease the privileges required to execute the transaction. The third algorithm exploits the semantics of the transactions to further minimize aborts. The transactions in an application are classified into different types. Static analysis is performed to check if the policy update transaction type X affects transaction type Y. If not, then transactions of type Y need not be aborted because of the execution of transactions of type

X. This approach reduces, but does not eliminate, unnecessary abort of transactions. [1]

Finally, to reduce the effects of unnecessary aborts, we propose to decompose a transaction into steps. Decomposition improves performance and reduces unnecessary aborts, but introduces additional problems. Specifically, the isolation, atomicity, and consistency properties of a transaction are violated and we no longer have a basis for arguing about the correctness of the execution. To solve this problem, we propose to use an alternate semantic-based transaction processing model. This model, extends the one proposed in [3], has a set of replacement properties which applications must satisfy to ensure the correctness of the execution. We show how the applications can be analyzed to check for the satisfaction of these properties. We use the specification language Z [36] to perform our analysis.

The rest of the paper is organized as follows. Section 2 describes our model. Section 3 presents our first algorithm. Section 4 proposes an algorithm that uses the semantics of the policy update transaction. Section 5 shows how unnecessary aborts can be reduced by doing a semantic analysis of the application. Section 6 illustrates how the effect of useless aborts can be minimized by decomposing the transactions. Section 7 discusses some related work in this area. Section 8 concludes the paper. Appendix A gives the relevant Z notations. Appendix B demonstrates the proofs of some theorems. Appendix C illustrates how an example application can be analyzed to check for the satisfaction of properties and a pictorial description of the entire process.

2 Our Model

A *database* is specified as a collection of objects together with a set of *integrity constraints* on these objects. At any given time, the *state* of the database is determined by the values of the objects in the database. A change in the value of a database object changes the state. Integrity constraints are predicates defined over the state. A database state is said to be *consistent* if the values of the objects satisfy the given integrity constraints.

A *transaction* is an operation that transforms the database from one consistent state to another. To prevent the database from becoming inconsistent, transactions are the only means by which data objects are accessed and modified. A transaction is executed in a session that is initiated by some

[1]To completely eliminate unnecessary aborts, we would need to perform a dynamic analysis which is associated with a high execution time overhead. We do not take this approach.

user. Each session is associated with a single user. A transaction inherits the access privileges of the session in which it is initiated. A transaction can execute an operation on a database object only if it has the privilege to perform it. Such privileges are specified by access control policies.

In this paper, we consider only one kind of access control policies, namely, authorization policies[2]. An authorization policy specifies what operations an entity can perform on another entity. To keep our model simple, we consider systems that support positive authorization policies only. This means that the policies only specify what operations an entity is *allowed* to perform on another entity. The absence of an authorization policy authorizing an entity A to perform some operation O on another entity B, is interpreted as A not being allowed to perform operation O on entity B. For simplifying our presentation, we consider only simple kinds of authorization policies that are specified by *subjects*, *objects*, and *rights*. A subject can be a user, a group of users, a role or a process [35]. An object can be a database object, a group of such objects, or an object class. Subjects can perform only those operations on objects that are specified in the rights.

Definition 1 [Policy] A *policy* is a function that maps a set of subjects and a set of objects to a set of access rights. We formally denote this as follows: $P : \mathbb{P}(S) \times \mathbb{P}(O) \rightarrow \mathbb{P}(R)$ where P, $\mathbb{P}(S)$, $\mathbb{P}(O)$, and $\mathbb{P}(R)$ represent the policy function, the power set of subjects, the power set of objects, and the power set of access rights respectively.

In a database, policies are stored in the form of policy objects. A policy object is one that stores information about a policy. Henceforth, we use the term data objects to refer to the database objects that store data and the term policy object to those storing information about policies. Policy objects, like data objects, can be read and written. However, unlike ordinary data objects, policy objects can also be *deployed*.

Definition 2 [Policy Object] A policy object P_i is a triple $< S_i, O_i, R_i >$ where S_i, O_i, R_i denote the set of subjects, the set of objects, and the set of access rights of the policy respectively. A subject in the set S_i can perform only those operations on an object in O_i that are specified in R_i.

For example, $P =< \{John, Joe\}, \{FileF, FileG\}, \{r, w, x\} >$ is a policy object that gives subjects *John* and *Joe* the privilege to Read, Write, and Execute *FileF* and *FileG*. When *John* Reads *FileF* by virtue of policy P, we say P is deployed. We capture this by providing an explicit *deploy*

[2]Henceforth, we use the term policy or access control policy to mean authorization policy.

operation on policy object which is defined below. Note that, in order to deploy a policy object, it must be read. In other words, the operation deploy on a policy object assumes an implicit read operation on the policy object.

Definition 3 [Deploy] A policy object P_j is said to be *deployed* if there exists at least one subject in P_j that is currently accessing an object in P_j by virtue of the access rights specified in P_j.

We consider an environment in which multiple subjects read and modify data and policy objects, while the corresponding policies are in effect. To deal with this scenario, we need concurrency control mechanisms that (i) allow concurrent access to data objects and policy objects, and (ii) prevent security violations arising due to policy updates.

3 A Simple Algorithm for Policy Update

In this section we present a simple solution to the problem of policy updates. We assume that each data object is associated with two operations: Read and Write. A policy object is associated with three operations: Read, Write and Deploy. To distinguish between data objects and policy objects, we use English letters (x, y) to represent data objects and Greek letters (α, β) to denote policy objects. We begin by giving some definitions.

Definition 4 [Conflicting Operations] Two operations are said to *conflict* if both operate on the same data (policy) object and one of them is a Write operation.

Definition 5 [Transaction] A *transaction* T_i is a partial order with ordering relation $<_i$ where

1. $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a data object}\} \cup \{r_i[\alpha], w_i[\alpha], d_i[\alpha] \mid \alpha \text{ is a policy object}\} \cup \{a_i, c_i\}$;
2. $a_i \in T_i$ iff $c_i \notin T_i$;
3. if t is c_i or a_i, for any other operation $p_i \in T_i$, $p_i <_i t$;
4. if $r_i[x]$, $w_i[x] \in T_i$, then either $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$;
5. if $r_i[\alpha]$, $w_i[\alpha] \in T_i$, then either $r_i[\alpha] <_i w_i[\alpha]$ or $w_i[\alpha] <_i r_i[\alpha]$; and
6. if $d_i[\alpha]$, $w_i[\alpha] \in T_i$, then either $d_i[\alpha] <_i w_i[\alpha]$ or $w_i[\alpha] <_i d_i[\alpha]$.

Condition 1 defines the different kinds of operations in the transactions ($r_i[x]$, $w_i[x]$, $r_i[\alpha]$, $w_i[\alpha]$, $d_i[\alpha]$, a_i, c_i denote Read operation on data object x, Write operation on x, Read operation on policy object α, Write operation on α, Deploy operation on α, Abort or Commit operation in transaction T_i respectively). Condition 2 states that this set contains an Abort or a Commit operation but not both. Condition 3 states that Abort or Commit operation must follow all other operations of the transaction. Condition 4 (5) requires that the partial order $<_i$ specify the order of execution of Read and Write operations on a common data (policy) object. Condition 6 requires that the partial order $<_i$ specify the order of execution of Deploy and Write operations on a common policy object.

The algorithm that we propose is an extension of the strict two-phase locking (2PL) protocol [7]. The locking rules for data objects are the same as those of strict 2PL. A policy object α is associated with three locks: read lock (denoted by $RL(\alpha)$), write lock (denoted by $WL(\alpha)$) and deploy lock (denoted by $DL(\alpha)$). The locking rules for the policy objects are given in Table 1. *Yes* (*No*) entry in the lock table indicates that the lock request is granted (denied). *Signal* entry indicates that the lock request is granted, but only after doing some additional tasks. For the algorithm presented in this section, *Signal* indicates that the lock request is granted, but only after the transaction currently holding the lock is aborted and the lock is released. We use an example to illustrate this. Suppose some transaction T_k holds a deploy lock DL on a policy object α, and another transaction T_j wishes to get the write lock WL and update α. In such a case, a signal is generated to abort T_k, after which T_k releases the DL lock and T_j is granted the WL lock. We assume that a transaction updating a policy (T_j) has a higher priority than a transaction (T_k) that deploys this policy; hence, we generate a signal to abort the transaction deploying the policy (T_k). Suppose the policy update transaction T_j executes in session a and affected transaction T_k executes in session b. The two sessions may be different. In this case, the session a must have the privilege of terminating transactions in session b.

Current	*New Lock Request*		
Lock Status	*RL*	*WL*	*DL*
RL	*Yes*	*No*	*Yes*
WL	*No*	*No*	*No*
DL	*Yes*	*Signal*	*Yes*

Table 1: Locking Rules for Policy Objects

Transaction T_i acquires a read or a deploy lock on a policy object if no other transaction T_j has a write lock on the same object. Below we specify when a write lock is acquired by T_i.

Algorithm 1 Algorithm for Write

```
Procedure Process-Write ($w_i(\alpha)$)
begin
    if a transaction $T_j$ holds a write/read lock on $\alpha$
        exit;        /* Lock unavailable - $T_i$ can retry later */
    for each $T_j$ holding a deploy lock on $\alpha$
        abort $T_j$
    give write lock to $\alpha$
    accept ($w_i(\alpha)$)        /* Perform the Write Operation */
end
```

Definition 6 [Well-formed Transaction] A transaction T_i is *well-formed* if it satisfies the following conditions:

1. Before T_i reads or writes a data or policy object, it must deploy the policy object that authorizes T_i to perform the operation.

2. T_i must acquire the appropriate lock before deploying, reading, or writing a policy object.

3. T_i must acquire the appropriate lock before reading or writing a data object.

4. T_i cannot acquire a lock on an object if another transaction has locked the object in a conflicting mode.

5. All locks acquired by T_i are eventually released.

Definition 7 [Well-formed Two-phase Transaction] A transaction T_i is *well-formed two-phase* if it is well-formed and all its lock operations precede any of its unlock operations.

Consider a transaction T_i that reads object x (denoted by $r_i(x)$) and then writes object y (denoted by $w_i(y)$). Policy objects α and β authorize the subject initiating transaction T_i the privilege to read object x and the privilege to write object y respectively. An example well-formed two-phase execution of T_i consists of the following sequence of operations: $<DL_i[\alpha]$; $RL_i[x]$; $d_i[\alpha]$; $r_i[x]$; $DL_i[\beta]$; $WL_i[y]$; $d_i[\beta]$; $w_i[y]$; $UL_i[x]$; $UL_i[y]$; $UL_i[\alpha]$; $UL_i[\beta]$; $>$, where DL_i, RL_i, WL_i, d_i, r_i, w_i, UL_i denote the operations of acquiring deploy lock, acquiring read lock, acquiring write lock, deploy, read, write, lock release, respectively, performed by transaction T_i.

Definition 8 [Policy-Compliant Transaction] A transaction is *policy-compliant* if for every read or write operation that the transaction performs, there exists a policy that authorizes the transaction to perform the operation for the entire duration of the operation.

Note that, the long duration sensitive transaction given in Section 1 is not policy-compliant. This resulted in a security breach. Such security breaches are prevented by the locking rules for the deploy and write operations (Algorithm 1).

To argue about correct or incorrect executions we need the notion of histories. We adopt the definitions of *history*, *serial history*, *conflict equivalence of histories*, and *conflict serializable history* from Bernstein et al. [7]. We give some additional definitions which we use in this paper.

Definition 9 [Semantic Equivalence of Histories] Two histories H and H' are semantically equivalent if the following conditions hold:

1. H and H' are defined over the same set of transactions.

2. The state produced by executing H on some initial state s is the same as that produced by executing H' on the same initial state s.

3. The output produced by any transaction T_i in each of the histories is identical.

Note that our notion of semantic equivalence is weaker than conflict equivalence [7] as illustrated by the following theorem.

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

Definition 10 [Serializable History] A history H defined over a set of transactions $\{T_1, T_2, \ldots, T_n\}$ is serializable if it is semantic equivalent to some serial execution of the transactions in H.

Definition 11 [Policy-Compliant History] A history H defined over a set of transactions $\{T_1, T_2, \ldots, T_n\}$ is *policy-compliant* if all the transactions in the history are policy-compliant transactions.

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

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

4 Algorithm using Semantics of Policy Update Transactions

In this section we show how to use the semantics of policy update transactions to increase concurrency. The basic idea is to classify a policy update transaction either as a *policy relaxation* or as a *policy restriction*. Policy relaxation does not decrease the access rights of any subject; transactions executing by virtue of a policy need not be aborted when the policy is being relaxed. For policy restriction, we still need to abort transactions that are executing by virtue of the privileges given by the original policy.

Definition 12 [Policy Update Transaction] A *policy update transaction* modifies a policy object $P_i = < S_i, O_i, R_i >$ to P'_i where P'_i is obtained by transforming either one or more of the following: S_i to S'_i, O_i to O'_i, or R_i to R'_i. S'_i, O'_i, R'_i denote the modified subject, object and access rights respectively.

Definition 13 [Policy Relaxation Transaction] A *policy relaxation transaction* is a policy update transaction that does not decrease the access rights of any subject.

Definition 14 [Policy Restriction Transaction] A *policy restriction transaction* is a policy update transaction that is not a policy relaxation.

Consider the policy object $P_i =< \{John, Joe\}, \{FileF, FileG\}, \{r, w, x\} >$. Suppose P_i is changed to $P'_i = < \{John,\ Joe\},\ \{FileF, FileG, FileH\},\ \{r, w, x\} >$. This is an example of policy relaxation because the access rights of subjects *John* and *Joe* have not decreased. P_i being changed to $P''_i = < \{John\}, \{FileF,\ FileG,\ FileH\},\ \{r, w, x\} >$ is an example of policy restriction because *Joe*'s access rights are being curtailed.

In this paper, we take a semantic-based approach to classify a policy update transaction as a policy relaxation or restriction. We use the specification language Z [36] to express the different types of policy update transactions. Z is based on set theory, first order predicate logic, and a schema calculus to organize large specifications. Table 2 in Appendix A briefly explains the Z notations used in our examples. Other specification and analysis conventions peculiar to Z are explained as the need arises. We illustrate our approach by using the example of a simple hotel database. A partial specification of the hotel database appears in Figure 1. The hotel database has a set of objects, integrity constraints on these objects, and different types of transactions, which

$[GUEST, ROOM, USER, OBJECT, RIGHT]$

$ROLE ::= Manager \mid Supervisor \mid Clerk$

$POLICY == \mathbb{P}(ROLE) \times \mathbb{P}(OBJECT) \times \mathbb{P}(RIGHT)$

$STATUS ::= Available \mid Taken$

Hotel

$$\begin{array}{|l}
Access : \mathbb{P}(POLICY) \\
Role : POLICY \rightarrow \mathbb{P}(ROLE) \\
Object : POLICY \rightarrow \mathbb{P}(OBJECT) \\
Right : POLICY \rightarrow \mathbb{P}(RIGHT) \\
UserRole : USER \rightarrow \mathbb{P}(ROLE) \\
Status : ROOM \nrightarrow STATUS \\
Assign : ROOM \nrightarrow GUEST \\
\hline
\mathrm{dom}(Status \rhd \{Taken\}) = \mathrm{dom}\, Assign
\end{array}$$

DeletePolicy

$$\begin{array}{|l}
p? : \mathbb{P}(POLICY);\ \Delta Hotel \\
\hline
p? \in Access;\ Access' = Access \setminus \{p?\} \\
Object'(p?) = \varnothing;\ Role'(p?) = \varnothing;\ Right'(p?) = \varnothing \\
UserRole' = UserRole \\
Status' = Status;\ Assign' = Assign
\end{array}$$

AssignUserToRole

$$\begin{array}{|l}
\Delta Hotel;\ i? \in USER;\ r? \in ROLE \\
\hline
r? \notin UserRole(i?);\ Assign' = Assign;\ Role' = Role \\
UserRole' = UserRole \oplus \{i \mapsto UserRole(i) \cup \{r?\}\} \\
Access' = Access;\ Object' = Object \\
Right' = Right;\ Status' = Status
\end{array}$$

RemoveUserFromRole

$$\begin{array}{|l}
\Delta Hotel;\ i? \in USER;\ r? \in ROLE \\
\hline
r? \in UserRole(i?);\ Status' = Status \\
UserRole' = UserRole \oplus \{i \mapsto UserRole(i) - \{r?\}\} \\
Access' = Access;\ Role' = Role;\ Object' = Object \\
Right' = Right;\ Assign' = Assign
\end{array}$$

AddPolicy

$$\begin{array}{|l}
p? : \mathbb{P}(POLICY);\ s? : \mathbb{P}(ROLE) \\
t? : \mathbb{P}(OBJECT);\ r? : \mathbb{P}(RIGHTS);\ \Delta Hotel \\
\hline
p? \notin Access;\ Access' = Access \cup \{p?\} \\
Role' = Role \cup \{p? \mapsto s?\};\ UserRole' = UserRole \\
Object' = Object \cup \{p? \mapsto t?\};\ Assign' = Assign \\
Right' = Right \cup \{p? \mapsto r?\};\ Status' = Status
\end{array}$$

AddRoleToPolicy

$$\begin{array}{|l}
p? : \mathbb{P}(POLICY) \\
s? : ROLE;\ \Delta Hotel \\
\hline
p? \in Access;\ s? \notin Role(p?) \\
Role' = Role \oplus \\
\quad \{p? \mapsto Role(p?) \cup \{s?\}\} \\
Object' = Object \\
UserRole' = UserRole \\
Status' = Status;\ Assign' = Assign \\
Right' = Right;\ Access' = Access
\end{array}$$

RemoveRoleFromPolicy

$$\begin{array}{|l}
p? : \mathbb{P}(POLICY) \\
s? : ROLE;\ \Delta Hotel \\
\hline
p? \in Access;\ s? \in Role(p?) \\
Role' = Role \oplus \\
\quad \{p? \mapsto Role(p?) - \{s?\}\} \\
Access' = Access;\ Object' = Object \\
Assign' = Assign \\
UserRole' = UserRole \\
Right' = Right;\ Status' = Status
\end{array}$$

RemoveObject

$$\begin{array}{|l}
p? : \mathbb{P}(POLICY) \\
t? : OBJECT;\ \Delta Hotel \\
\hline
p? \in Access;\ t? \subseteq Object(p?) \\
Object' = Object \oplus \\
\quad \{p? \mapsto Object(p?) - \{t?\}\} \\
Assign' = Assign \\
Role' = Role;\ Status' = Status \\
UserRole' = UserRole \\
Right' = Right;\ Access' = Access
\end{array}$$

AddObject

$$\begin{array}{|l}
p? : \mathbb{P}(POLICY) \\
t? : OBJECT;\ \Delta Hotel \\
\hline
p? \in Access;\ t? \notin Object(p?) \\
Object'(p?) = Object(p?) \cup \{t?\} \\
Role' = Role;\ Access' = Access \\
Assign' = Assign \\
UserRole' = UserRole \\
Right' = Right;\ Status' = Status
\end{array}$$

RemoveRight

$$\begin{array}{|l}
p? : \mathbb{P}(POLICY) \\
r? : RIGHT;\ \Delta Hotel \\
\hline
p? \in Access;\ r? \in Right(p?) \\
Right' = Right \oplus \\
\quad \{p? \mapsto Right(p?) - \{r?\}\} \\
Role' = Role;\ Assign' = Assign \\
Object' = Object \\
UserRole' = UserRole \\
Access' = Access;\ Status' = Status
\end{array}$$

AddRight

$$\begin{array}{|l}
p? : \mathbb{P}(POLICY) \\
r? : RIGHT;\ \Delta Hotel \\
\hline
p? \in Access;\ r? \notin Right(p?) \\
Right' = Right \oplus \\
\quad \{p? \mapsto Right(p?) \cup \{r?\}\} \\
Role' = Role;\ Access' = Access \\
Assign' = Assign \\
Object' = Object;\ Status' = Status \\
UserRole' = UserRole
\end{array}$$

Figure 1: Different Types of Policy Update Transactions

we identify and explain below. The specification assumes the given types *GUEST* and *ROOM*, which enumerate all possible guests and rooms, respectively. The enumerated type $STATUS$ gives the possible status of each room which can be either $Available$ or $Taken$. The hotel database uses a Role-Based Access Control (RBAC) policy. In RBAC users are assigned to roles (job function), roles are associated with access rights and users acquire access rights by being members of roles. To support RBAC, the specification assumes the types *USER*, *OBJECT* and *RIGHT* which enumerate all possible users, objects and access rights, respectively. The enumerated type *ROLE* lists the possible roles associated with the application which in this case can be $Manager$, $Supervisor$ or $Clerk$. We define *POLICY* as the cartesian product of the set of roles, the set of objects and the set of rights.

In Z, states, as well as operations, are described with a two-dimensional notation called a $schema$. The declarations for the objects appear in the top part of the schema and constraints on the objects appear in the bottom part. The objects in the hotel database are listed in the schema $Hotel$, which defines the state of the hotel. The object $Access$ stores the set of policies pertaining to the hotel database. The objects *Role*, *Object*, and *Right* are functions that record the set of roles, set of objects, and set of access rights, respectively, that are associated with a policy. The object *UserRole* is a function that gives the set of roles that are associated with a user. The object $Status$ is a partial function that records the status of each room which may be $Available$ or $Taken$. The object $Assign$ is a partial function that relates the rooms with status *Taken* and the guests to which these rooms are assigned. Note that the partial functions representing the objects $Status$ and $Assign$ implicitly capture the following integrity constraints: (i) each room can be either available or taken, and (ii) each room can be assigned to at most one guest. An additional integrity constraint on the objects in hotel database appears in the bottom part of schema $Hotel$: $\mathrm{dom}(Status \rhd \{Taken\}) = \mathrm{dom}\, Assign$ – this states that the set of rooms with status taken is exactly the set of rooms assigned to guests.

For this specific application, we have different types of transactions, namely, *AddRoleToPolicy*, *RemoveRoleFromPolicy*, *AddObject*, *RemoveObject*, *AddRight*, and *RemoveRight*, that update policies. *AddRoleToPolicy* takes as input a policy object $p?$, and the role $s?$ that is to be added to the policy. *AddRoleToPolicy* has two preconditions: (i) the policy $p?$ that is to be modified must be an existing policy and (ii) the role $s?$ should not already be in the role set of policy $p?$. The postcondition of *AddRoleToPolicy* modifies $Role$ to include the new role $s?$ in the role set of policy $p?$. The other objects in the hotel database, namely, $Status$, $Assign$, $Access$, $Object$, $Right$ and $UserRole$ remain unchanged. The other policy update transactions are specified in a similar manner. We also have two types of transactions, namely, $AddPolicy$ and $DeletePolicy$, that add a new policy and delete an existing policy, respectively. In RBAC, users can be assigned to roles or removed from existing roles. This is handled by the transactions $AssignUserToRole$ which assigns a new role to an user and $RemoveUserFromRole$ which deletes a previously assigned role from the user.

Before we proceed further, we make a distinction between types of transactions and instances of transaction. The specifications described above represent the different types of transactions in the hotel database. Histories, on the other hand, refer to instances of transactions. We denote instances of transactions using the notation T_i, T_j. We denote the type of an instance of a transaction T_i as $ty(T_i)$. The total number of types of transactions in any given application are finite. However, infinite instances of transactions may be generated from these finite types of transactions.

From the specifications we can infer that the transactions of type *AssignUserToRole*, *AddPolicy*, *AddRoleToPolicy*, *AddObject*, and *AddRight* are policy relaxations; *RemoveUserFromRole*, *DeletePolicy*, *RemoveRoleFromPolicy*, *RemoveObject*, and *RemoveRight* are policy restrictions.

4.1 Concurrency Control Based on Knowledge of Policy Change

We now give a concurrency control algorithm that uses the knowledge of the kind of policy change. The algorithm is a modification to that presented in Section 3. The only difference is in the tasks performed when a $Signal$ entry is encountered. Recall that a $Signal$ entry is encountered when a transaction (say, T_i) wants to acquire a write lock on some policy object while another transaction (say, T_k) holds a deploy lock on the same policy object. In the previous approach (Section 3), the transaction T_k gets aborted and subsequently T_i gets the write lock. In this approach, we first check if T_i is a policy restriction transaction. If so, we abort all the transactions that are deploying the policy (such as, T_k), and then grant the write lock to T_i. If T_i is not a policy restriction operation, then no transaction deploying this policy needs to be aborted.

Note that this algorithm no longer guarantees that transactions will be well-formed (Definition 6). All conditions, except Condition 4, of well-formed transactions still hold. The algorithm relaxes Condition 4 in Definition 6. It allows a transaction to acquire a write lock on a policy object even though other transactions may hold deploy locks on the same policy object. This does not cause any problems as indicated by the Theorems given below.

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

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

Theorem 6 This mechanism provides more concurrency than the one given in Section 3.

5 Algorithms using Semantics of Application

The algorithm presented in the previous section helps eliminate aborts of transactions that are executing by virtue of policies that are being relaxed. However, there are still some unnecessary aborts of transactions resulting from policy restrictions. For example, consider a transaction T_i that restricts policy P_k but does not impact a transaction T_j that deploys this policy P_k. In such situations with our earlier algorithms, the transaction T_j gets aborted. Such useless aborts can be avoided if we analyze the transaction deploying the policy and the policy restriction transaction. In this section we show how to perform such analysis using the semantic knowledge of the transactions.

To illustrate our technique, we specify some other types transactions that are relevant to the hotel database. These are $\mathit{Reserve}$, Cancel and Report as shown in figure 2. Transactions of these types can be executed provided that the user initiating the transaction has the privilege to perform the operations specified in the transaction. Moreover, checking whether a user has the privilege to execute a transaction T_i and executing the transaction T_i should take place atomically. Otherwise a policy update transaction T_j can execute in between and change the privilege in such a manner that the user no longer can execute the transaction T_i; this will result in T_i not being policy-compliant. Thus, checking the privilege can be thought of as a precondition of the transaction.

The $\mathit{Reserve}$ transaction takes as input a room $r?$, a guest $t?$, and a user $i?$ initiating the transaction. $\mathit{Reserve}$ has preconditions that check whether user $i?$ has the privilege to execute the transaction. These include that the user $i?$ must be assigned to the $\mathit{Supervisor}$ role and there must exist policies $P1$ and $P2$ that give the $\mathit{Supervisor}$ the privilege to read and write the objects Status and Assign. $\mathit{Reserve}$ has another precondition that the room $r?$ must be $\mathit{Available}$. $\mathit{Reserve}$ has a postcondition that the status of $r?$ is changed to $Taken$, and the ordered pair $r? \mapsto t?$ is added to the function $Assign$. The other objects remain unchanged.

$Cancel$ cancels an assignment of a room to a guest. $Cancel$ has the precondition that $r?$ must be assigned to some guest (that is, $r?$ must be in the domain of $Assign$). Like $Reserve$, $Cancel$ has preconditions that ensure only authorized users can execute the transaction. The postcondition of $Cancel$ is that $r?$ is removed from the domain of the function $Assign$, and status of $r?$ is changed to $Available$.

$Report$ prints the objects $Status$ and $Assign$ as outputs. The precondition of $Report$ ensures that only $Clerk$ can execute this transaction, and $Clerk$ has the privileges necessary to execute the operations in this transaction.

$$
\begin{array}{|l}
\hline
\textit{Reserve} \\
\Delta Hotel;\ t? : GUEST \\
r? : ROOM;\ i? : USER \\
\hline
Supervisor \in UserRole(i?) \\
\exists P1 : POLICY \bullet P1 \in Access \\
\quad \wedge\ Supervisor \in Role(P1) \\
\quad \wedge\ Status \in Object(P1) \\
\quad \wedge\ \{r, w\} \subseteq Right(P1) \\
\exists P2 : Policy \bullet P2 \in Access \\
\quad \wedge\ Supervisor \in Role(P2) \\
\quad \wedge\ Assign \in Object(P2) \\
\quad \wedge\ \{r, w\} \subseteq Right(P2) \\
Status(r?) = Available \\
Status' = Status \oplus \{r? \mapsto Taken\} \\
Assign' = Assign \cup \{r? \mapsto t?\} \\
UserRole' = UserRole \\
Access' = Access;\ Role' = Role \\
Object' = Object;\ Right' = Right \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Cancel} \\
\Delta Hotel;\ r? : ROOM \\
i? : USER \\
\hline
Supervisor \in UserRole(i?) \\
\exists P1 : POLICY \bullet P1 \in Access \\
\quad \wedge\ Supervisor \in Role(P1) \\
\quad \wedge\ Status \in Object(P1) \\
\quad \wedge\ \{r, w\} \subseteq Right(P1) \\
\exists P2 : Policy \bullet P2 \in Access \\
\quad \wedge\ Supervisor \in Role(P2) \\
\quad \wedge\ Assign \in Object(P2) \\
\quad \wedge\ \{r, w\} \subseteq Right(P2) \\
r? \in \operatorname{dom} Assign \\
Role' = Role \\
UserRole' = UserRole \\
Assign' = \{r?\} \mathbin{\lhd} Assign \\
Object' = Object;\ Right' = Right \\
Status' = Status \oplus \{r? \mapsto Available\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Report} \\
\Xi Hotel \\
i? : USER \\
opstatus! : ROOM \nrightarrow STATUS \\
opassign! : ROOM \nrightarrow GUEST \\
\hline
Clerk \in UserRole(i?) \\
\exists P1 : POLICY \bullet P1 \in Access \\
\quad \wedge\ Clerk \in Role(P1)\ \wedge \\
\quad Status \in Object(P1) \\
\quad \wedge\ r \in Right(P1) \\
\exists P2 : Policy \bullet P2 \in Access \\
\quad \wedge\ Clerk \in Role(P2) \\
\quad \wedge\ Assign \in Object(P2) \\
\quad \wedge\ r \in Right(P2) \\
opstatus! = Status \\
opassignments! = Assign \\
\hline
\end{array}
$$

Figure 2: Specification of Transactions of the Hotel Database

Let us consider the problem of policy updates. Consider the execution of a $Reserve$ transaction. The $Reserve$ transaction deploys policy $P1$ and $P2$. Suppose a $RemoveRoleFromPolicy$ transaction executes and updates policy $P1$, then the $Reserve$ transaction will be aborted. The $RemoveRoleFromPolicy$ transaction, specified in Figure 1, does not contain much semantic information. This transaction can be executed to remove any role from any given policy. Rather than

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{RemoveRoleManager}} \\
p? : \mathbb{P}(\mathit{POLICY});\ \Delta \mathit{Hotel} \\
\hline
p? \in \mathit{Access} \\
\{\mathit{Manager}\} \subseteq \mathit{Role}(p?) \\
\mathit{Role}' = \mathit{Role} \oplus \\
\quad \{p? \mapsto \mathit{Role}(p?) - \{\mathit{Manager}\}\} \\
\mathit{Status}' = \mathit{Status} \\
\mathit{Access}' = \mathit{Access} \\
\mathit{Assign}' = \mathit{Assign} \\
\mathit{UserRole}' = \mathit{UserRole} \\
\mathit{Object}' = \mathit{Object};\ \mathit{Right}' = \mathit{Right} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{RemoveRoleSuper}} \\
p? : \mathbb{P}(\mathit{POLICY});\ \Delta \mathit{Hotel} \\
\hline
p? \in \mathit{Access} \\
\{\mathit{Supervisor}\} \subseteq \mathit{Role}(p1) \\
\mathit{Role}' = \mathit{Role} \oplus \\
\quad \{p? \mapsto \mathit{Role}(p?) - \{\mathit{Supervisor}\}\} \\
\mathit{Access}' = \mathit{Access} \\
\mathit{Status}' = \mathit{Status} \\
\mathit{Assign}' = \mathit{Assign} \\
\mathit{UserRole}' = \mathit{UserRole} \\
\mathit{Object}' = \mathit{Object};\ \mathit{Right}' = \mathit{Right} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{RemoveRoleClerk}} \\
p? : \mathbb{P}(\mathit{POLICY});\ \Delta \mathit{Hotel} \\
\hline
p? \in \mathit{Access} \\
\{\mathit{Clerk}\} \subseteq \mathit{Role}(p?) \\
\mathit{Role}' = \mathit{Role} \oplus \\
\quad \{p? \mapsto \mathit{Role}(p?) - \{\mathit{Clerk}\} \\
\mathit{Access}' = \mathit{Access} \\
\mathit{Status}' = \mathit{Status} \\
\mathit{Assign}' = \mathit{Assign} \\
\mathit{UserRole}' = \mathit{UserRole} \\
\mathit{Object}' = \mathit{Object};\ \mathit{Right}' = \mathit{Right} \\
\hline
\end{array}$$

Figure 3: Further Classification of Policy Update Transactions

having this *RemoveRoleFromPolicy* transaction, we propose to have more specialized versions, such as, *RemoveRoleManager*, *RemoveRoleSuper*, and *RemoveRoleClerk*. The specifications of these transactions are given in Figure 3. The *RemoveRoleManager* takes as input a policy $p?$ and removes *Manager* from the role set of this policy. The *RemoveRoleSuper* and *RemoveRoleClerk* are specified in a similar manner.

Now, let us consider the previous scenario of a *Reserve* transaction executing by virtue of the policies $P1$ and $P2$. Suppose $P1$ is updated by a transaction of type *RemoveRoleClerk*. The *Reserve* transaction is not affected by this change to $P1$ and need not be aborted. Suppose a *Report* transaction was also executing by virtue of the privilege given by policy $P1$. The *Report* transaction, in this case, will be affected and must be aborted. To reason about which transactions are affected by which policy update transactions, we propose the notion of commutativity of transactions. Our notion is weaker than the syntactic notion of commutativity [7].

Definition 15 [Commutativity of Types of Transactions] Two transaction types X and Y commute if the following holds: the final state and the outputs produced by executing any transaction T_i of type X followed by executing a transaction T_j of type Y on some initial state S are the same as that obtained by executing T_j followed by T_i on the same initial state. Stated formally, transaction types X and Y commute if they satisfy the following property: $X \mathbin{⨾} Y \Leftrightarrow Y \mathbin{⨾} X$, where $X \mathbin{⨾} Y$ formally defines the state resulting from the execution of an instance of X followed by the execution of an instance of Y.

Definition 16 [Commute Set of $ty(T_i)$] The set of types of transactions that commute with transactions of type $ty(T_i)$.

The schema composition operation in Z can be used to evaluate whether two transaction types commute or not. Theorem 7 proves that $Reserve$ and $RemoveRoleManager$ commute. Similarly, *Commute Set of RemoveRoleManager* = $\{$*Reserve, Cancel, Report*$\}$, *Commute Set of RemoveRoleSuper* = $\{$*Report*$\}$, and *Commute Set of RemoveRoleClerk* = $\{$*Reserve, Cancel*$\}$. The commute set of any policy relaxation operation contains all types of transactions.

Theorem 7 $Reserve \fatsemi RemoveRoleManager \Leftrightarrow RemoveRoleManager \fatsemi Reserve$

5.1 Concurrency Control Mechanism

The concurrency control mechanism requires that for each type of policy update transaction, we evaluate the commute set of that type of transaction. Note that this task is performed only once for every application. However, if the application changes, we need to re-evaluate the commute sets of the policy update transactions.

The mechanism is similar to that presented in Section 4.1. The only difference is in obtaining a write lock on a policy object. For a transaction T_i to obtain a write lock on a policy object α, we first check whether some other transaction T_j already has a read or write lock on α. If so, the lock request is denied. Otherwise, we proceed to check whether some transaction T_j has a deploy lock on this object. If so, we check whether $ty(T_j)$ is in the commute set of $ty(T_i)$. If not, T_j is aborted and the write lock is granted to T_i. If $ty(T_j)$ is in commute set of $ty(T_i)$, then T_j need not be aborted to grant T_i the write lock.

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

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

6 Minimizing the Effects of Unnecessary Aborts

The approach given in the previous section can minimize but not eliminate aborts. This is because transactions are classified into a set of finite types. The types of transactions are analyzed statically to check whether they commute with each other. Whenever there is a potential for a policy restriction transaction to affect some transaction that deploys the policy, the transaction deploying the policy must be aborted. Such unnecessary aborts can be eliminated by doing a dynamic analysis; we do not adopt this approach because of the high execution time penalty. In this section, we propose an approach that helps to mitigate the effects of these unnecessary aborts.

Our approach is to decompose transactions into steps. Each step of a transaction is now executed atomically. Decomposition of transactions into steps helps increase the performance in two ways. First, the data objects and policy objects are locked for a shorter duration of time (the duration of a step). This helps to achieve more concurrency. Second, policy restrictions no longer cause useless transaction aborts; they cause steps to abort. Since the work done by a step is less than that done by the original transaction, less amount of work is wasted.

Decomposition of a transaction into steps introduces new problems. Specifically, the *atomicity*, *isolation*, and *consistency* properties are no longer satisfied by transactions. Consider, for example, the *Reserve* transaction that is decomposed into two steps: *Res1* and *Res2*. Step *Res1* changes the status of a room and step *Res2* assigns this room to a guest. We can no longer guarantee the atomicity of the *Reserve* transaction. *Res1* may successfully execute, but the corresponding *Res2* may not be able to execute. This may happen if a step of some other transaction executes after *Res1* whose postcondition prevents the execution of *Res2*. Since *Res1* has committed, it cannot be undone. In such cases, we need compensating steps that will undo the effects of *Res1*. Decomposition of a transaction may also violate database consistency. For example, execution of *Res1* violates the integrity constraint: the rooms with status taken are those assigned to guests. Moreover, if the *Report* transaction is executed between these two steps of the *Reserve* transaction, inconsistencies will be displayed to the user. This is because transactions no longer have the isolation property and the *Report* transaction can see the partial effects of the other transactions. In short, the traditional transaction processing model [7] can no longer be used to ensure the correctness of the execution. To solve this problem, we propose using an alternate semantic-based transaction processing model.

6.1 Decomposing Transactions into Steps

In our semantic-based transaction processing model we decompose transactions into steps. To simplify our presentation, we propose decomposing only the *Reserve* and the *Cancel* transactions. The other transactions are not decomposed; these can be thought of as single step transactions.

$$\begin{array}{l}
\underline{\mathit{Res1}} \\
\Delta \mathit{HotelD};\ r? : \mathit{ROOM} \\
i? : \mathit{USER} \\
\hline
\mathit{Supervisor} \in \mathit{UserRole}(i?) \\
\exists P1 : \mathit{POLICY} \bullet P1 \in \mathit{Access} \\
\quad \wedge\ \mathit{Supervisor} \in \mathit{Role}(P1) \\
\quad \wedge\ \mathit{Status} \in \mathit{Object}(P1) \\
\quad \wedge\ \{r, w\} \subseteq \mathit{Right}(P1) \\
\mathit{Status}(r?) = \mathit{Available} \\
\mathit{Status}' = \mathit{Status} \oplus \{r? \mapsto \mathit{Taken}\} \\
\mathit{acquired}' = \mathit{acquired} \cup \{r?\} \\
\mathit{Assign}' = \mathit{Assign} \\
\mathit{UserRole}' = \mathit{UserRole} \\
\mathit{Access}' = \mathit{Access};\ \mathit{Role}' = \mathit{Role} \\
\mathit{Object}' = \mathit{Object};\ \mathit{Right}' = \mathit{Right}
\end{array}$$

$$\begin{array}{l}
\underline{\mathit{Res2}} \\
\Delta \mathit{HotelD};\ t? : \mathit{GUEST} \\
r? : \mathit{ROOM};\ i? : \mathit{USER} \\
\hline
\mathit{Supervisor} \in \mathit{UserRole}(i?) \\
\exists P2 : \mathit{Policy} \bullet P2 \in \mathit{Access} \\
\quad \wedge\ \mathit{Supervisor} \in \mathit{Role}(P2) \\
\quad \wedge\ \mathit{Assign} \in \mathit{Object}(P2) \\
\quad \wedge\ \{r, w\} \subseteq \mathit{Right}(P2) \\
\mathit{Assign}' = \mathit{Assign} \cup \{r? \mapsto t?\} \\
\mathit{acquired}' = \mathit{acquired} \setminus \{r?\} \\
\mathit{Status}' = \mathit{Status} \\
\mathit{Object}' = \mathit{Object} \\
\mathit{Right}' = \mathit{Right} \\
\mathit{UserRole}' = \mathit{UserRole} \\
\mathit{Access}' = \mathit{Access};\ \mathit{Role}' = \mathit{Role}
\end{array}$$

$$\begin{array}{l}
\underline{\mathit{HotelD}} \\
\mathit{Access} : \mathbb{P}(\mathit{POLICY}) \\
\mathit{UserRole} : \mathit{USER} \rightarrow \mathbb{P}(\mathit{ROLE}) \\
\mathit{Role} : \mathit{POLICY} \rightarrow \mathbb{P}(\mathit{ROLE}) \\
\mathit{Object} : \mathit{POLICY} \rightarrow \mathbb{P}(\mathit{OBJECT}) \\
\mathit{Right} : \mathit{POLICY} \rightarrow \mathbb{P}(\mathit{RIGHT}) \\
\mathit{Assign} : \mathit{ROOM} \nrightarrow \mathit{GUEST} \\
\mathit{Status} : \mathit{ROOM} \nrightarrow \mathit{STATUS} \\
\mathit{acquired} : \mathbb{P}(\mathit{ROOM}) \\
\mathit{released} : \mathbb{P}(\mathit{ROOM}) \\
\hline
\mathrm{dom}(\mathit{Status} \rhd \{\mathit{Taken}\}) \cup \\
\quad \mathit{released} = \mathrm{dom}\, \mathit{Assign} \cup \mathit{acquired}
\end{array}$$

$$\begin{array}{l}
\underline{\mathit{Can1}} \\
\Delta \mathit{HotelD};\ r? : \mathit{ROOM} \\
i? : \mathit{USER} \\
\hline
\mathit{Supervisor} \in \mathit{UserRole}(i?) \\
\exists P1 : \mathit{POLICY} \bullet P1 \in \mathit{Access} \\
\quad \wedge\ \mathit{Supervisor} \in \mathit{Role}(P1) \\
\quad \wedge\ \mathit{Status} \in \mathit{Object}(P1) \\
\quad \wedge\ \{r, w\} \subseteq \mathit{Right}(P1) \\
r? \in \mathrm{dom}\, \mathit{Assign} \\
\mathit{Role}' = \mathit{Role};\ \mathit{Assign}' = \mathit{Assign} \\
\mathit{Object}' = \mathit{Object};\ \mathit{Right}' = \mathit{Right} \\
\mathit{UserRole}' = \mathit{UserRole} \\
\mathit{released}' = \mathit{released} \cup \{r?\} \\
\mathit{Status}' = \mathit{Status} \oplus \{r? \mapsto \mathit{Available}\}
\end{array}$$

$$\begin{array}{l}
\underline{\mathit{Can2}} \\
\Delta \mathit{HotelD};\ r? : \mathit{ROOM} \\
i? : \mathit{USER} \\
\hline
\mathit{Supervisor} \in \mathit{UserRole}(i?) \\
\exists P2 : \mathit{Policy} \bullet P2 \in \mathit{Access} \\
\quad \wedge\ \mathit{Supervisor} \in \mathit{Role}(P2) \\
\quad \wedge\ \mathit{Assign} \in \mathit{Object}(P2) \\
\quad \wedge\ \{r, w\} \subseteq \mathit{Right}(P2) \\
\mathit{Assign}' = \{r?\} \lhd \mathit{Assign} \\
\mathit{released}' = \mathit{released} \setminus \{r?\} \\
\mathit{Status}' = \mathit{Status};\ \mathit{Role}' = \mathit{Role} \\
\mathit{UserRole}' = \mathit{UserRole} \\
\mathit{Object}' = \mathit{Object} \\
\mathit{Right}' = \mathit{Right}
\end{array}$$

$$\begin{array}{l}
\underline{\mathit{ReportD}} \\
\Xi \mathit{HotelD} \\
i? : \mathit{USER} \\
\mathit{opstatus}! : \mathit{ROOM} \nrightarrow \mathit{STATUS} \\
\mathit{opassign}! : \mathit{ROOM} \nrightarrow \mathit{GUEST} \\
\hline
\mathit{Clerk} \in \mathit{UserRole}(i?) \\
\mathit{acquired} = \varnothing;\ \mathit{released} = \varnothing \\
\exists P1 : \mathit{POLICY} \bullet P1 \in \mathit{Access} \\
\quad \wedge\ \mathit{Clerk} \in \mathit{Role}(P1)\ \wedge \\
\quad \mathit{Status} \in \mathit{Object}(P1) \\
\quad \wedge\ r \in \mathit{Right}(P1) \\
\exists P2 : \mathit{Policy} \bullet P2 \in \mathit{Access} \\
\quad \wedge\ \mathit{Clerk} \in \mathit{Role}(P2) \\
\quad \wedge\ \mathit{Assign} \in \mathit{Object}(P2) \\
\quad \wedge\ r \in \mathit{Right}(P2) \\
\mathit{opstatus}! = \mathit{Status} \\
\mathit{opassignments}! = \mathit{Assign}
\end{array}$$

Figure 4: Specification of the Decomposed Hotel Database

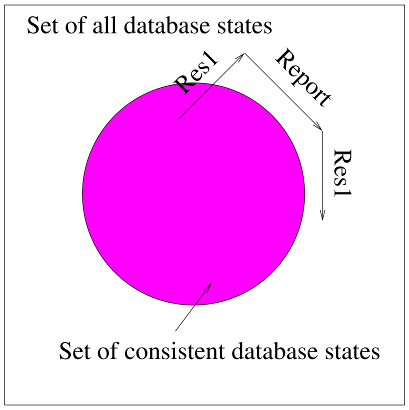
We decompose the *Reserve* transaction such that $Res1$ changes the status of the room, and $Res2$ assigns this room to the guest. The preconditions of *Reserve* are also divided appropriately. For instance, the *Reserve* transaction has a precondition that ensures that it is executed only by the role *Supervisor*. Thus, each of the steps $Res1$ and $Res2$ must include preconditions that ensure each get executed by the *Supervisor*. The *Reserve* transaction also requires the existence of some policy $P1$ that gives the *Supervisor* read and write privilege on the object *Status*. This precondition is now a part of $Res1$ as the object *Status* is read and written by $Res1$ only. The other precondition that requires the existence of policy $P2$ that allows the *Supervisor* to read and write the object *Assign* is now a part of $Res2$. The precondition that checks whether the status of room $r?$ is available becomes the precondition for $Res1$. The detailed specification for $Res1$ and $Res2$ appears in Figure 4. Similarly, the *Cancel* transaction is decomposed into *Can1* and *Can2*.

6.2 Generalizing the Integrity Constraints

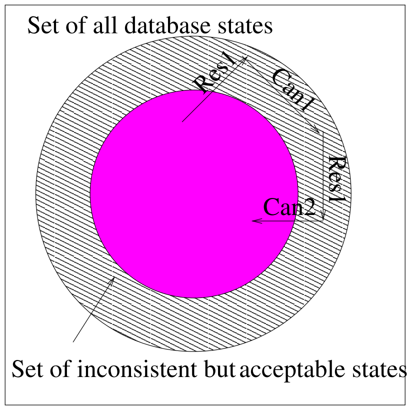
When transactions are decomposed into steps, the execution of a step may no longer satisfy the integrity constraints. Consider the $Reserve$ transaction in the hotel database decomposed into two steps $Res1$ and $Res2$, where $Res1$ changes the status of a room $r?$ from $Available$ to $Taken$, and $Res2$ assigns the room $r?$ whose status was changed by $Res1$ to the guest $t?$. Execution of atomic steps, such as $Res1$, violates the integrity constraint: the rooms with status taken are those assigned to guests $((\mathrm{dom}\, Status \rhd \{Taken\}) = \mathrm{dom}\, Assign)$). The arrow labeled $Res1$ in figure 5(a) illustrates this possibility. Once the integrity constraints are violated, we have no basis for assessing the correctness of the execution.

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

Figure 5(b) illustrates a model that allows inconsistent states – as defined by the integrity constraints – that are nonetheless acceptable. The temporary inconsistency introduced by $Res1$ is allowed, and steps of some other transactions, for example $Can1$, can tolerate the inconsistency introduced by $Res1$, and so are allowed to proceed. The general approach is to generalize the orig-



(a) Standard classification of database states


Set of all database states
Res1
Can1
Res1
Can2
Set of inconsistent but acceptable states

(b) Database states as classified in our model

Figure 5: Classification of Database States

inal integrity constraints and decompose transactions such that each step satisfies the new integrity constraints.

The inner circle in figure 5(b) denotes the set consisting of all consistent database states; that is, states that satisfy the original integrity constraints. The outer circle denotes the set consisting of states that satisfy the generalized integrity constraints. The ring describes the states that satisfy the generalized integrity constraints but not the original integrity constraints. The important part about figure 5(b) is that the set of inconsistent but acceptable states is formally identified and distinguished from the states that are unacceptable. The advantage is that formal analysis can be used to investigate activities in states satisfying the generalized integrity constraints.

In the hotel database, the original integrity constraint $\mathrm{dom}(\mathit{Status} \rhd \{\mathit{Taken}\}) = \mathrm{dom}\,\mathit{Assign}$ is generalized by adding the auxiliary variables $\mathit{acquired}$ and $\mathit{released}$. The auxiliary variable $\mathit{acquired}$, which is a set of room, denotes the rooms whose status is taken but which have not yet been assigned to guests. The auxiliary variable $\mathit{released}$ is also a set of rooms; it denotes the rooms whose status is available but which are still assigned to some guests. The generalized integrity constraint is specified as follows $(\mathrm{dom}(\mathit{Status} \rhd \{\mathit{Taken}\}) \setminus \mathit{acquired}) \cup \mathit{released} = \mathrm{dom}\,\mathit{Assign}$. The left hand side signifies the union of rooms with status taken that are assigned to some guests and rooms with status available that are assigned to some guests. The generalized integrity constraint can be rewritten as $\mathrm{dom}(\mathit{Status} \rhd \{\mathit{Taken}\}) \setminus \mathit{acquired} = \mathrm{dom}\,\mathit{Assign} \setminus \mathit{released}$. The new state of the hotel database, given by $HotelD$, appears in Figure 4.

6.3 Sensitive Transaction Isolation Property

After generalizing the integrity constraints, we are now in a position to argue about the correctness of the decomposition. We do this by proposing a set of necessary properties that must be satisfied by the decomposed application. One such property is the sensitive transaction isolation property.

In our model, we allow transactions to access database states that do not satisfy the original integrity constraints. But we may wish to keep some transactions from viewing any inconsistency with respect to the original integrity constraints. For example, some transactions may output data to users; these transactions are called *sensitive* transactions [16]. We require sensitive transactions to appear to have generated outputs from a consistent state.

Definition 17 [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 (a state satisfying the original integrity constraints) even though T_i may be running on an inconsistent database state.

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 original integrity constraints relevant to the calculation of any outputs. This subset of the integrity constraints must be implied by the preconditions of the step or steps that generate the output. Second, as pointed out by Rastogi, Korth and Silberschatz [37], if outputs are generated by multiple steps, interleaving between these steps must be controlled to ensure that outputs from later steps are consistent with outputs from earlier steps.

6.4 Compensating Steps

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

Suppose a $Reserve$ transaction aborts after step $Res1$ commits. Some mechanism must undo the effects of $Res1$. Nullifying the effects of $Res1$ using the backwards recovery method of traditional undo [7], where the state that existed before $Res1$ is physically restored, is not possible

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{CompRes1}} \\
\Delta \mathit{HotelD};\ r? : \mathit{ROOM};\ i? : \mathit{USER} \\
\hline
\mathit{Supervisor} \in \mathit{UserRole}(i?) \\
\exists P1 : \mathit{POLICY} \bullet P1 \in \mathit{Access} \\
\quad \wedge\ \mathit{Supervisor} \in \mathit{Role}(P1) \\
\quad \wedge\ \mathit{Status} \in \mathit{Object}(P1) \\
\quad \wedge\ \{r, w\} \subseteq \mathit{Right}(P1) \\
\mathit{Status}' = \mathit{Status} \oplus \{r? \mapsto \mathit{Available}\} \\
\mathit{acquired}' = \mathit{acquired} \setminus \{r?\} \\
\mathit{Assign}' = \mathit{Assign};\ \mathit{Access}' = \mathit{Access} \\
\mathit{Subject}' = \mathit{Subject} \\
\mathit{Object}' = \mathit{Object};\ \mathit{Right}' = \mathit{Right} \\
\mathit{UserRole}' = \mathit{UserRole} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{CompCan1}} \\
\Delta \mathit{HotelD};\ r? : \mathit{ROOM} \\
\hline
\mathit{Supervisor} \in \mathit{UserRole}(i?) \\
\exists P1 : \mathit{POLICY} \bullet P1 \in \mathit{Access} \\
\quad \wedge\ \mathit{Supervisor} \in \mathit{Role}(P1) \\
\quad \wedge\ \mathit{Status} \in \mathit{Object}(P1) \\
\quad \wedge\ \{r, w\} \subseteq \mathit{Right}(P1) \\
\mathit{Status}' = \mathit{Status} \oplus \{r? \mapsto \mathit{Taken}\} \\
\mathit{released}' = \mathit{released} \setminus \{r?\} \\
\mathit{Assign}' = \mathit{Assign};\ \mathit{Access}' = \mathit{Access} \\
\mathit{Subject}' = \mathit{Subject} \\
\mathit{Object}' = \mathit{Object};\ \mathit{Right}' = \mathit{Right} \\
\mathit{UserRole}' = \mathit{UserRole} \\
\hline
\end{array}$$

Figure 6: Compensating Steps for *Reserve* and *Cancel* transactions

because steps of other transactions may have read the updates of *Res*1. Instead we adopt the forward recovery solution of a compensating step [16] which semantically undoes the effects of the committed step *Res*1 and does not disturb the transactions that read from *Res*1. Like other steps, compensating steps are executed atomically. However, the role of a compensating step differs from that of other steps. A compensating step is not considered part of normal processing of a transaction; it is initiated only to semantically undo a transaction.

In the hotel database, we specify compensating steps for transactions *Reserve* and *Cancel*. *Reserve* has a compensating step, *CompRes*1, that undoes the actions of step *Res*1. Similarly, the compensating step *CompCan*1 undoes the action of step *Can*1 of the *Cancel* transaction. The specifications of the compensating steps are shown in Figure 6.

6.5 Semantic Histories

In the previous subsections we show how *Reserve* and *Cancel* transactions can be decomposed into steps and how we can specify compensating steps for these transactions. To argue about the correctness of executions, we need the notion of histories.

Histories reflect actual executions of transactions and consists of instances of steps and compensating steps. In cases where it is 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. In our model, each step or compensating step of a transaction is executed atomically. Since transactions are not executed atomically, the notion of *serial* histories in the traditional transaction processing model [7] is replaced by the notion of stepwise serial history.

Definition 18 [Stepwise Serial History] A *stepwise serial history H* over a set of transactions **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 a compensating step $C_{ij} \in H$, then $S_{ij} \in H$ and S_{ij} precedes C_{ij} in H, and
4. if S_{ij} precedes S_{im} in H and $C_{ij} \in H$, then C_{im} precedes C_{ij} in H.

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 a compensating step appears after the corresponding step. Condition 4 ensures that if the steps of a transaction are executed in a particular order, the corresponding compensating steps must be executed in the reverse order. Before describing what it means for a history to be complete, we introduce the notion of complete execution of transactions.

Definition 19 [Complete Execution] Consider a transaction T_i decomposed into steps $S_{i1}, \ldots, S_{in}$ with compensating steps C_{i1}, …, $C_{i(n-1)}$. 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 steps C_{ij}, …, C_{i1}, where $j < n$.

Unlike *serial histories* in traditional databases, not all stepwise serial histories are correct. To reason about correct and incorrect executions we need state information in the histories. This motivates us to propose the notion of *semantic history*.

Definition 20 [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 is incomplete, but from a formal perspective, partial semantic histories are just semantic histories. Complete semantic histories are a special case of semantic histories:

Definition 21 [Complete Semantic History] A semantic history H over a set of transactions **T** is a *complete* semantic history if the execution of each T_i in **T** is complete.

Having defined semantic histories, we now give some properties of semantic histories that are needed for correct execution. These properties are explained in the subsequent sections.

6.6 Semantic Atomicity Property

We begin by motivating the need for the semantic atomicity property. When transactions have been decomposed into steps, it may not be possible to complete a partially executed transaction. This may happen if the precondition of some later step is not satisfied, and the earlier steps cannot be compensated for. The semantic atomicity property formalizes this idea. If an application has the semantic atomicity property, then all partially executed transactions will complete. If the application does not have this property, then the application must be revised.

Definition 22 [Semantic Atomicity Property] Every semantic history H_p defined over a set of transactions **T** is a prefix of some complete semantic history H over **T**.

6.7 Consistent Execution Property

In our model steps may be executed in an inconsistent database state. However, after all the transactions complete, the database should be restored to a consistent state. The consistent execution property formalizes this idea. If an application has consistent execution property, then any complete semantic history executed on a consistent state will result in a consistent state.

Definition 23 [Consistent Execution Property] If we execute a complete semantic history H on an initial state (i.e., state prior to the execution of any step in H) that satisfies the original integrity constraints, then the final state (i.e., state after the execution of the last step in H) also satisfies the original integrity constraints.

6.8 Policy-Compliant Property

To ensure that execution of transactions do not cause security breaches, we propose the policy-compliant property of semantic histories. But first, we define policy-compliant steps:

Definition 24 [Policy-Compliant Step] A step T_{ij} of transaction T_i is *policy-compliant* if for every read or write operation that the step performs, there is a policy that authorizes the step to perform the operation for the entire duration of the operation.

Definition 25 [Policy-Compliant History] A semantic history H defined over a set of transactions $\mathbf{T} = \{T_1, T_2, \ldots, T_m\}$ is *policy-compliant* if every step T_{ij} in H is policy-compliant.

Definition 26 [Policy-Compliant Property] Every semantic history is a policy-compliant history.

A discussion of the necessary properties of the hotel database appear in Appendix C. The entire process of policy update using transaction decomposition approach is also illustrated in this section.

6.9 Acceptable Histories

Next, we define which histories are acceptable and which are not. Recall that, the decomposition of transaction into steps violates the atomicity, consistency, and isolation properties of the transaction. We have proposed three replacement properties, namely, semantic atomicity, consistent execution, and sensitive transaction isolation, that must be satisfied by histories to ensure correct execution in our model. In addition, we propose the policy-compliant property which is necessary to prevent security breaches. We now give the formal definition of a correct semantic history.

Definition 27 [Correct Semantic History] A semantic history satisfying all the four properties, that is, the semantic atomicity property, the consistent execution property, the sensitive transaction isolation property, and the policy-compliant property is defined to be a *correct semantic history*.

In each correct semantic history, the steps of a transaction are executed serially. Serial execution of steps of transactions results in poor performance. To improve the performance, we propose that the steps of the transactions be executed concurrently. When steps are executed concurrently, we need to reason about the correctness of the execution. To do this, we propose another definition of history by using the notion of steps and conflicts.

Definition 28 [History] A history H defined over a set of transactions **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 **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. We now give the definition of correct concurrent histories.

Definition 29 [Correct Stepwise Serializable History] A history that is semantic equivalent to a correct semantic history is a correct stepwise serializable history.

6.10 Concurrency Control Mechanism

In this section we describe a mechanism that generates correct stepwise serializable histories. The concurrency control mechanism is similar to that given in Section 5.1. The only difference is that steps and not transactions are executed atomically. This means that steps request locks instead of transactions and lock release takes place when steps terminate. The algorithm for obtaining the write lock on a policy object makes use of the notion of commutativity of types of steps. Each type of step of a policy update transaction has an associated commute set that contains the types of steps of transactions that commute with it. Suppose step T_{ij} requests a write lock on policy object α. The algorithm first checks to see if some other step, say T_{pq}, has a read or write lock on α. If so, the lock request is denied and T_{ij} has to try later. Otherwise the algorithm finds the set of steps that have a deploy lock on α. For each such step T_{mn}, the algorithm checks if $ty(T_{mn})$ is in the commute set of $ty(T_{ij})$. If $ty(T_{mn})$ is not in this commute set, then T_{mn} is aborted and the locks associated with T_{mn} are released and T_{ij} gets the write lock. Otherwise T_{mn} need not be aborted to grant T_{ij} the write lock.

Theorem 10 The committed projection of a history H generated by our mechanism produces policy-secure histories.

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.

7 Related Work

Related Work on Policy Updates

A large body of work appears in the area of access control. Researchers have proposed access control models [6, 8, 44, 45], policy specification languages [5, 9, 12, 19, 21, 22, 34, 42], and techniques for identifying conflicts in policies [18, 28, 32, 50, 51]. Policy updates have received relatively little attention. Ray and Xin [41] proposed algorithms for concurrent and real-time update of access control policies. The algorithms proposed are syntax-based and do not use any semantic knowledge of transactions. The idea that semantic knowledge can be used for real-time update of access control policies was proposed by Ray [38]. In this work the author considered only simple kinds of authorization policies $P_i =< SS_i, TS_i, RS_i >$ where SS_i, TS_i, RS_i represents the set of subjects, the set of objects, and the set of access rights respectively. A policy can be changed by performing a series of set union or set difference operations. Depending on what kinds of operations were performed, the policy update is classified as policy restriction or relaxation. The author proposed algorithms that uses this knowledge to generate conflict serializable histories. The author also introduced the idea of how the application context can be used to check whether a policy update transaction interferes with a transaction that deploys the policy. However, a formal treatment of this idea is missing from the work. The current work provides a more in-depth treatment, together with theorems and proofs. In addition, in this work we show how transactions can be decomposed to minimize the effects of useless aborts.

Some work has been done in identifying interesting adaptive policies and formalization of these policies [14, 47]. A separate work [46] illustrates the feasibility of implementing adaptive security policies. The above works pertain to multilevel security policies encountered in military environments; the focus is in protecting confidentiality of data and preventing covert channels. We consider a more general problem and our results will be useful to both the commercial and military sector.

Automated management of security policies for large scale enterprise has been proposed by

Damianou [11]. This work uses the PONDER specification language to specify policies. The simplest kinds of access control policies in PONDER are specified using a *subject-domain*, *object-domain* and *access-list*. The subject-domain specifies the set of subjects that can perform the operations specified in the access-list on the objects in the object-domain. New subjects can be added to the subject-domain or subjects can be removed from the subject-domain. The object-domain can also be changed in a similar manner. But this work does not allow the policy specification itself to change. An example will help illustrate this point. Suppose we have a policy in PONDER that is implementing Role-Based Access Control: *subject-domain = Manager*, *object-domain = /usr/local*, *access-list = read, write*. This policy allows all *Managers* to *read/write* all the files stored in the directory */usr/local*. The supporting toolkit allows adding/removing users from the domain *Manager*, adding/deleting files in the domain */usr/local*. However, it will not allow the policy specification to be changed. For example, the subject-domain cannot be changed to *Supervisors*. Our work, focuses on the problem of updating the policy specification itself and complements the above mentioned work.

Related Work on Semantic-Based Transaction Processing

Concurrency control in database systems is a well researched topic. Some of the important pioneering works have been described by Bernstein et al. [7]. Thomasian [52] provides a more recent survey of concurrency control methods and their performance. The use of semantics for increasing concurrency has also been proposed by various researchers [3, 4, 15, 16, 17, 20, 24, 29]. The use of semantic knowledge for solving other problems, such as, ensuring atomicity of secure multilevel transactions [2, 39], and ensuring autonomy of local databases [37, 40], have also been investigated by researchers.

Other researchers have also generalized the notion of serializability and proposed new correctness criteria for transaction executions [1, 13, 15, 16, 24, 25, 27, 54]. For example, in [13], the authors develop *quasi-serializability* as a correctness criterion for transactions executing on heterogeneous distributed database systems. The authors in [1] propose three correctness criteria: *consistency*, *orderability* and *strong orderability* for non-serializable executions. Garcia-Molina [16] proposed the idea of semantically consistent schedule and sensitive transactions. A semantically consistent schedule is one which maintains database consistency and sensitive transactions view consistent data. In some of these works, researchers have decomposed transactions into steps

[1, 15, 16, 24, 48, 49] and developed semantic based correctness criteria [1, 15, 16]. A formal treatment of compensating transactions is presented in [26] in which the authors propose a generalized model that allows the definition of various types of compensating transactions.

Related Work on Dynamic and Adaptive Workflow

Researchers [10, 23, 30, 31, 33, 43, 53] have motivated the need for workflow systems that have the ability to adapt to the changing environments. Kammer et al. [23] have identified different kinds of exception handling capabilities for adaptive workflows. They also describe how some of these capabilities have been incorporated in the Endeavors workflow support system. Van der Aalst et al.[53] have mentioned that when a workflow is changed there are three ways to manage workflow instances that were active during the change: (a) restart, (b) proceed, and (c) transfer. Restart causes abort of all the tasks that were being executed under the original model. Proceed requires continuation of the workflow instance under the original model. Transfer requires the instance to be changed dynamically such that it complies with the new model.

Sadiq [43] recognizes that changes to workflow model or workflow instances can lead to serious inconsistencies and errors. He points out that the problem becomes more critical when active instances are involved while the workflow model is being changed. The authors approach this problem by first formalizing how workflow can be modified, and then checking whether the instance initiated under the original workflow complies with the updated workflow. The instances generated from the old workflow may continue to follow the old model or they may be provided with a revised schedule to comply with the new model. The generation of revised schedule may need manual intervention.

Mathews [31] has also addressed the problem of changing workflow policies while active workflow instances are in progress. He agrees that aborting active workflow instances prior to changing a workflow policies is inefficient and ineffective. To deal with this situation, he proposes SWAP, an agent-based architecture, that deals with changes in B2B workflow policies. In this architecture, the different agents keep track of the active workflow instances and their state of execution and are responsible for transitioning the workflow instance so that it complies with the new policy.

Muller and Rahm[33] suggested that dynamic workflow modification is needed for many applications, such as, medical domain. In such domains the large number of exceptions cannot be handled by the domain except. They have developed a rule-based approach for detecting semantic exceptions that occur during instance execution. The control-flow of other instances affected by this exception needs to be modified dynamically. For this purpose, the authors have proposed two algorithms that carry out the dynamic change of control-flow.

8 Conclusion

Real-time update of policy is an important problem for both the commercial and the military sector. Towards this end, we propose different kinds of concurrency control algorithms that will allow real-time and concurrent policy updates in a database system. The algorithms differ with respect to the level of semantic knowledge used and the degree of concurrency provided. Of the algorithms proposed, the most concurrency is obtained from the one in which transactions are not executed as atomic units but are decomposed into steps. Once transactions are decomposed into steps, the properties of atomicity, consistency and isolation are violated and the traditional transaction processing model can no longer be used. To solve this problem, we propose an alternate semantic-based transaction processing that ensures the correctness of execution by requiring the application to have a set of necessary properties.

Checking whether the application satisfies the necessary properties comes with a cost. The cost involves formally specifying and analyzing the application. However, such analysis is done off-line and incurs no execution time penalty. Moreover, the analysis needs to be done only when the application is developed or modified. One can nevertheless argue that checking for the satisfaction of properties manually may be a tedious and error-prone process for real-world applications. Towards this end, we propose the use of model-checkers for automatically verifying the properties. In earlier works [39, 40] we have shown how properties for multi-level secure and multidatabase applications can be verified using state-of-the-art model checkers. Similar techniques can be used for verification of properties presented in this paper. Sometimes applications may not have the necessary properties. In such cases, the applications must be revised. Note that sometimes it may not be possible to change the application without modifying its semantics. For these situations our model is not suitable and an alternate approach must be used.

In this work we have shown how formal methods can be used in semantics-based concurrency control. Using formal methods has additional benefits. In future, we plan to investigate how formal methods can be used to detect inconsistencies arising due to policy updates. Examples of inconsistencies include conflicts in the policy specification and loss of functionality due to errors in the policy specification. We plan to extend our approach by specifying more complex policies and providing more detailed analysis of inconsistencies that can arise due to policy updates.

References

[1] D. Agrawal, A. El Abbadi, and A. K. Singh. Consistency and Orderability: Semantics-Based Correctness Criteria for Databases. *ACM Transactions on Database Systems*, 18(3):460–486, September 1993.

[2] P. Ammann, S. Jajodia, and I. Ray. Ensuring Atomicity of Multilevel Transactions. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 74–84, Oakland, CA, May 1996.

[3] P. Ammann, S. Jajodia, and I. Ray. Applying Formal Methods to Semantic-Based Decomposition of Transactions. *ACM Transactions on Database Systems*, 22(2):215–254, June 1997.

[4] B.R. Badrinath and K. Ramamritham. Semantics-Based Concurrency Control: Beyond Commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, March 1992.

[5] S. Barker. Security Policy Specification in Logic. In *Proceedings of the International Conference on Artificial Intelligence*, pages 143–148, Las Vegas, NV, 2000.

[6] D. E. Bell and L. J. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report MTR-2997, MITRE Corporation, Bedford, MA, July 1975.

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

[8] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning. *ACM Transactions on Database Systems*, 23(3):231–285, 1998.

[9] F. Chen and R. Sandhu. Constraints for Role-Based Access Control. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, Gaithersburg, MD, 1995.

[10] E. C. Cheng. OMM: An Organization Modeling and Management System. In *Proceedings of Towards Adaptive Workflow Systems Workshop co-located with CSCW 98*, Seattle, WA, November 1998.

[11] N. Damianou, T. Tonouchi, N. Dulay, E. Lupu, and M. Sloman. Tools for Domain-based Policy Management of Distributed Systems. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium*, Florence, Italy, April 2002.

[12] N. C. Damianou. *A Policy Framework for Management of Distributed Systems*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, London, U.K., 2002.

[13] W. Du and A.K. Elmagarmid. Quasi Serializability: A Correctness Criterion for Global Concurrency Control in Interbase. In *Proceedings of the International Conference on Very Large Databases*, pages 347–355, Amsterdam, Netherlands, 1989.

[14] J. Thomas Haigh et al. Assured Service Concepts and Models: Security in Distributed Systems. Technical Report RL-TR-92-9, Rome Laboratory, Air Force Material Command, Rome, NY, January 1992.

[15] A. A. Farrag and M. T. Özsu. Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

[16] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[17] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 249–259, San Francisco, CA, 1987.

[18] N. Griffeth and H. Velthuijsen. Reasoning About Goals to Resolve Conflicts. In *Proceedings of the International Conference on Intelligent Cooperative Information Systems*, pages 197–204, Los Alamitos, California, 1993.

[19] R. J. Hayton, J. M. Bacon, and K. Moody. Access Control in Open Distributed Environment. In *IEEE Symposium on Security and Privacy*, pages 3–14, Oakland, CA, May 1998.

[20] M. P. Herlihy and W. E. Weihl. Hybrid Concurrency Control for Abstract Data Types. *Journal of Computer and System Sciences*, 43(1):25–61, August 1991.

[21] M. Hitchens and V. Varadarajan. Tower: A Language for Role-Based Access Control. In *Proceedings of the Policy Workshop*, Bristol, U.K., 2001.

[22] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A Logical Language for Expressing Authorizations. In *IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, May 1997.

[23] P. Kammer, G. Bolcer, R. Taylor, and M. Bergman. Techniques for Supporting Dynamic and Adaptive Workflow. *Computer Supported Cooperative Work*, 9(3-4):269–292, 2000.

[24] 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.

[25] 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.

[26] H.F. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the International Conference on Very Large Databases*, pages 95–106, Brisbane, Australia, 1990.

[27] E. Levy, H. F. Korth, and A. Silberschatz. A Theory of Relaxed Atomicity. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 95–109, August 1991.

[28] E. Lupu and M. Sloman. Conflict Analysis for Management Policies. In *Proceedings of the 5th IFIP/IEEE International Symposium on Integrated Network Management*, pages 430–443, San Diego, California, May 1997. Chapman & Hall.

[29] Nancy A. Lynch. Multilevel Atomicity—A New Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[30] D. Manolescu and R. Johnson. Dynamic Object Model and Adaptive Workflow. In *Proceedings of Metadata and Active Object-Model Pattern Mining Workshop co-located with OOPSLA 99*, Denver, CO, November 1999.

[31] M. G. Mathews. Supporting Dynamic Change in B2B Coordination. Technical report, The MITRE Corporation, June 2001.

[32] N. Minsky, V. Ungureanu, W. Wang, and J. Zhang. Building Reconfiguration Primitives into the Law of a System. In *Proceedings of the International Conference on Configurable Distributed Systems*, pages 89–97, Annapolis, MD, May 1996.

[33] R. Muller and E. Rahm. Rule-Based Dynamic Modification of Workflows in a Medical Domain. In A.P. Buchmann, editor, *Proceedings of BTW*, pages 429–448, Freiburg in Breisgau, Germany, March 1999. Springer.

[34] R. Ortalo. A Flexible Method for Information Systems Security Policy Specification. In *Proceedings of the 5th European Symposium on Research in Computer Security*, Louvain-la-Neuve, Belgium, 1998. Springer-Verlag.

[35] J. Park and R. Sandhu. Towards Usage Control Models: Beyond Traditional Access Controls. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies*, pages 57–64, Monterey, California, June 2002.

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

[37] R. Rastogi, H. F. Korth, and A. Silberchatz. Exploiting Transaction Semantics in Multidatabase Systems. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 101–109, Vancouver, Canada, June 1995.

[38] I. Ray. Real-Time Update of Access Control Policies. *Data and Knowledge Engineering*, 49(3):287–309, June 2004.

[39] I. Ray, P. Ammann, and S. Jajodia. A Semantic Model for Multilevel Transactions. *Journal of Computer Security*, 6(3):181–217, 1998.

[40] I. Ray, P. Ammann, and S. Jajodia. Using Semantic Correctness in Multidatabases to Achieve Local Autonomy, Distribute Coordination, and Maintain Global Integrity. *Information Sciences*, 129(1-4):155–195, November 2000.

[41] I. Ray and T. Xin. Concurrent and Real-Time Update of Access Control Policies. In *Proceedings of the 14th International Conference on Database and Expert Systems*, pages 330–339, Prague, Czech Republic, September 2003.

[42] C. Ribeiro, A. Zuquete, and P. Ferreira. SPL: An Access Control Language for Security Policies with Complex Constraints. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2001.

[43] S. Sadiq. Workflows in Dynamic Environment – Can they be managed? In *Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*, Wollongong, Australia, March 1999.

[44] P. Samarati and S. Vimercati. Access Control: Policies, Models and Mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design (Tutorial Lectures)*, pages 137–196, September 2000.

[45] R. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[46] E. A. Schneider, W. Kalsow, L. TeWinkel, and M. Carney. Experimentation with Adaptive Security Policies. Technical Report RL-TR-96-82, Rome Laboratory, Air Force Material Command, Rome, NY, June 1996.

[47] E. A. Schneider, D. G. Weber, and T. de Groot. Temporal Properties of Distributed Systems. Technical Report RADC-TR-89-376, Rome Air Development Center, Rome, NY, September 1989.

[48] L. Sha, J. P. Lehoczky, and E.D. Jensen. Modular Concurrency Control and Failure Recovery. *IEEE Transactions on Computers*, 37(2):146–159, February 1988.

[49] D. Shasha, E. Simon, and P. Valduriez. Simple Rational Guidance for Chopping Up Transactions. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 298–307, San Diego, CA, June 1992.

[50] E. Sibley. Experiments in Organizational Policy Representation: Results to Date. In *Proceedings of the IEEE International Conference on Systems Man and Cybernetics*, pages 337–342, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[51] E. Sibley, J. Michael, and R. Wexelblat. Use of an Experimental Policy Workbench: Description and Preliminary Results. In C. Landwehr and S. Jajodia, editors, *Database Security V: Status and Prospects*, pages 47–76. Elsevier Science Publishers, 1992.

[52] A. Thomasian. Concurrency Control: Methods, Performance and Analysis. *ACM Computing Surveys*, 30(1):70–119, 1998.

[53] W.M.P. van der Aalst, T. Basten, H.M.W. Verbeek, P.A.C. Verkoulen, and M. Voorhoeve. Adaptive Workflow. In J.B.L. Filipe, editor, *Enterprise Information Systems*. Kluwer Academic Publishers, 1999.

[54] H. Wachter and A. Reuter. The ConTract model. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219–263. Morgan Kauffman, San Mateo, CA, 1992.

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')
$A \mathbin{⨾} B$	Forward Composition of A with B
$x \mapsto y$	Ordered Pair $(x,\ y)$
$A \mathrel{⇸} B$	Partial Function from A to B
$A \mathrel{⤔} B$	Partial Injective Function from A to B
$B \mathbin{⩤} A$	Relation A with Set B Removed from Domain
$A \rhd B$	Relation A with Range Restricted to Set B
$\operatorname{dom} A$	Domain of Relation A
$\operatorname{ran} A$	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 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 α that authorizes T_i to update O_r. Thus, when T_i initially accessed O_r, there was a policy object α that allowed T_i to update O_r. So, the only possibility is that while T_i was updating O_r, the policy object α got deleted or modified. But according to well-formed rules this is not possible. Any transaction T_j modifying the policy object α has to obtain a write lock (WL) on α. Before the write lock on α

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 $\{T_1, T_2, \ldots, T_n\}$, 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 $T_1 \rightarrow T_2 \rightarrow T_3 \ldots T_n \rightarrow T_1$. 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 α that authorizes T_i to update O_r. Thus, when T_i initially accessed O_r, there was a policy object α that allowed T_i to update O_r. So, the only possibility is that while T_i was updating O_r, the policy object α 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 α has to obtain a write lock (WL) on α. Before the write lock on α can be granted, the transaction T_i holding the deploy lock (DL) has to be aborted and the deploy lock released (because T_j restricts 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, α), 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 $H_1 \in$ **H1**, then $H_1 \in$ **H2**. (ii) Next, we will prove that for some history H_2 where $H_2 \in$ **H2**, $H_2 \notin$ **H1**. In the following paragraphs we outline the two proofs.

Proof of (i): Let $H_1 \in$ **H1**. We need to prove that $H_1 \in$ **H2**. Assume that $H_1 \notin$ **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 $H_1 \in$ **H2**.

Proof of (ii): We need to show that for some $H_2 \in$ **H2**, $H_2 \notin$ **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 α. T_j updates policy α by performing a policy relaxation operation $w_j(\alpha)$. T_j executes due to the privileges given by policy β. Let us see how the concurrency control mechanism in Section 4 executes H_2. First, a deploy lock will be obtained on policy object α 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 β and perform the deploy operation. Then, T_j will acquire a write lock on policy object α 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, $H_2 \in$ **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 α. Before the write lock can be obtained, transaction T_i must be aborted. Therefore, $H_2 \notin$ **H1**.

Proofs of Theorems in Section 5

Theorem 7 $Reserve \mathbin{⨾} RemoveRoleManager \Leftrightarrow RemoveRoleManager \mathbin{⨾} Reserve$

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

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_ ReserveRemoveRoleManager \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta Hotel;\ t? : GUEST;\ r? : ROOM;\ i? : USER;\ p? : \mathbb{P}(POLICY) \\
\hline
Supervisor \in UserRole(i?) \\
\exists P1 : POLICY \bullet P1 \in Access \land Supervisor \in Role(P1) \\
\qquad \land\ Status \in Object(P1) \land \{r, w\} \subseteq Right(P1) \\
\exists P2 : Policy \bullet P2 \in Access \land Supervisor \in Role(P2) \\
\qquad \land\ Assign \in Object(P2) \land \{r, w\} \subseteq Right(P2) \\
Status(r?) = Available;\ p? \in Access;\ Manager \subseteq Role(p?) \\
Role' = Role \oplus \{p? \mapsto Role(p?) - \{Manager\}\} \\
Status' = Status \oplus \{r? \mapsto Taken\};\ Assign' = Assign \cup \{r? \mapsto t?\} \\
Access' = Access;\ Object' = Object;\ Right' = Right;\ UserRole' = UserRole \\
\hline
\end{array}$$

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 $acquired = \varnothing$ and $released = \varnothing$. Thus we have to prove that if the initial state of a complete semantic history satisfies

$acquired = \varnothing$ and $released = \varnothing$, the final state of the history will also satisfy $acquired = \varnothing$ and $released = \varnothing$.

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, $\mid acquired \mid = n_1 - (n_2 + cn_1) \ldots (1)$. 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, $n_1 = n_2 + cn_1 \ldots (2)$. From (1) and (2), $\mid acquired \mid = 0$. In other words $acquired = \varnothing$. Using similar arguments we can prove that, $released = \varnothing$. Thus, in a complete history the final state satisfies $acquired = \varnothing$ and $released = \varnothing$.

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 $Report$:

$$\mathrm{dom}(Status \rhd \{Taken\}) = \mathrm{dom}(Assign)$$

This constraint is implied by the generalized integrity constraint:

$$\mathrm{dom}(Status \rhd \{Taken\}) \cup released = \mathrm{dom}(Assign) \cup acquired$$

when $acquired = released$. Since the sets $acquired$ and $released$ are disjoint, we include preconditions $acquired = \varnothing$ and $released = \varnothing$ 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 $CompRes1$. 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

$$\begin{array}{|l}
\text{\_}\mathit{ModCompRes}1\rule{0pt}{0pt}\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \\
\Delta \mathit{HotelD};\ r? : \mathit{ROOM} \\
\hline
i? \in \mathit{Supervisor} \cup \mathit{SecurityOfficer} \\
\exists P1 : \mathit{POLICY} \bullet P1 \in \mathit{Access} \\
\quad \wedge (\mathit{Supervisor} \in \mathit{Subject}(P1) \\
\qquad \vee\ \mathit{SecurityOfficer} \in \mathit{Subject}(P1)) \\
\quad \wedge\ \mathit{Status} \in \mathit{Object}(P1) \\
\quad \wedge \{r, w\} \subseteq \mathit{Right}(P1) \\
\mathit{Status}' = \mathit{Status} \oplus \{r? \mapsto \mathit{Available}\} \\
\mathit{acquired}' = \mathit{acquired} \setminus \{r?\} \\
\mathit{Subject}' = \mathit{Subject} \\
\mathit{Assign}' = \mathit{Assign};\ \mathit{Access}' = \mathit{Access} \\
\mathit{Object}' = \mathit{Object};\ \mathit{Right}' = \mathit{Right} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\text{\_}\mathit{ModCompCan}1\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \\
\Delta \mathit{HotelD};\ r? : \mathit{ROOM} \\
\hline
i? \in \mathit{Supervisor} \cup \mathit{SecurityOfficer} \\
\exists P1 : \mathit{POLICY} \bullet P1 \in \mathit{Access} \\
\quad \wedge (\mathit{Supervisor} \in \mathit{Subject}(P1) \\
\qquad \vee\ \mathit{SecurityOfficer} \in \mathit{Subject}(P1)) \\
\quad \wedge\ \mathit{Status} \in \mathit{Object}(P1) \\
\quad \wedge \{r, w\} \subseteq \mathit{Right}(P1) \\
\mathit{Status}' = \mathit{Status} \oplus \{r? \mapsto \mathit{Taken}\} \\
\mathit{released}' = \mathit{released} \setminus \{r?\} \\
\mathit{Subject}' = \mathit{Subject} \\
\mathit{Access}' = \mathit{Access};\ \mathit{Assign}' = \mathit{Assign} \\
\mathit{Object}' = \mathit{Object};\ \mathit{Right}' = \mathit{Right} \\
\hline
\end{array}$$

Figure 7: Modified Compensating Steps to Ensure Semantic Atomicity

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 $\mathit{Res}1$ (Figure 4) we include preconditions that ensure that $\mathit{Res}1$ 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-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

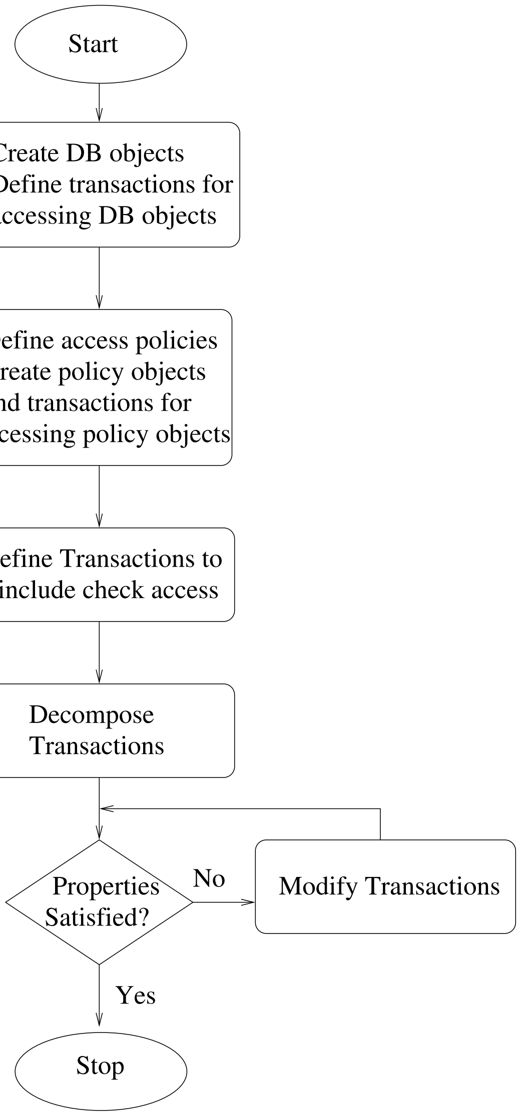

Start
Create DB objects
Define transactions for accessing DB objects
Define access policies
Create policy objects and transactions for accessing policy objects
Refine Transactions to include check access
Decompose Transactions
Properties Satisfied?
No
Modify Transactions
Yes
Stop

Figure 8: Flow Chart of the Process

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.