Available online at www.sciencedirect.com
SCIENCE DIRECT®
Data & Knowledge Engineering 49 (2004) 287–309
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

Real-time update of access control policies

Indrakshi Ray *

Department of Computer Science, Colorado State University, 601 S Howes Street, Fort Collins, CO 80523-1873, USA

Received 1 March 2003; received in revised form 8 September 2003; accepted 9 September 2003
Available online 9 December 2003

Abstract

Access control policies are security policies that govern access to resources. The need for real-time update of such policies while they are in effect and enforcing the changes immediately, arise in many scenarios. Consider, for example, a military environment responding to an international crisis, such as a war. In such situations, countries change strategies necessitating a change of policies. Moreover, the changes to policies must take place in real-time while the policies are in effect. In this paper we address the problem of real-time update of access control policies in the context of a database system. Access control policies, governing access to the data objects, are specified in the form of policy objects. The data objects and policy objects are accessed and modified through transactions. We consider an environment in which different kinds of transactions execute concurrently some of which may be policy update transactions. We propose algorithms for the concurrent and real-time update of security policies. The algorithms differ on the basis of the concurrency provided and the semantic knowledge used.

Keywords: Security policies; Concurrency control; Transaction management

1. Introduction

Like any other software artifacts in an enterprise, a security policy is subject to adaptive, preventive, and corrective maintenance. Since such policies are extremely critical for an enterprise, it is important to control the manner in which policies are updated. Updating policy in an adhoc manner may result in inconsistencies and problems with the policy specification; this, in turn, may create other problems, such as, security breaches, unavailability of resources, etc. In other words,

* Tel.: +1-970-491-7986; fax: +1-970-491-2466.
E-mail address: iray@cs.colostate.edu (I. Ray).

0169-023X/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2003.09.003

policy update should not be effected through adhoc operations but performed through well-defined *transactions* that have been previously analyzed.

An important issue that must be kept in mind when designing policy update transactions is that some policies may require *real-time update*. We use the term real-time update of a policy to mean that a policy is changed while it is in effect and this change needs to be enforced immediately. An example will help motivate the need for real-time update of policies. Suppose user *John*, by virtue of some policy P_i, has the privilege to execute a long-duration transaction that prints a large volume of sensitive financial information of a company kept in file I. While *John* is executing this transaction, an insider threat is suspected and the policy P_i is changed such that *John* no longer has the privilege of executing this transaction. Since existing access control mechanisms check *John*'s privileges *before John* initiates the transaction and never *during* the execution of the transaction, the updated policy P_i will not be correctly enforced causing financial loss to the company. In this case, the policy was updated correctly but not enforced immediately resulting in a security breach. Such real-time update of policies is also important for environments 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 update of policies.

In this paper we consider real-time policy updates in the context of a database system. We limit ourselves to access control policies. In our model a database consists of a set of objects. These objects are accessed and modified through transactions. Access control policies determine who has access to which objects. Access control policies are specified in the form of *policy objects*. The update of policy objects is done through well-defined transactions. The environment that we address is one in which different kinds of transactions execute concurrently some of which are policy update transactions. In other words, a policy object may be updated while transactions are executing by virtue of this policy. We propose different algorithms that allow for concurrent, real-time update of access control policies. The algorithms differ with respect to the level of semantic-knowledge that is used and the degree of concurrency achieved.

The rest of the paper is organized as follows. Section 2 describes our model. Section 3 describes a simple concurrency control algorithm for policy update. Section 4 shows how the semantics of the policy update operation can be used to increase concurrency. Section 5 illustrates how semantics of the transactions can be exploited to get even more concurrency. Section 6 highlights the related work. Section 7 concludes our paper with some pointers to future directions.

2. Our model

A *database* is specified as a collection of objects, along with some *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 can be initiated by a user, a group, a role or

another process. A transaction inherits the access rights of the entity initiating it. A transaction can access a database object only if it has the privilege to access it.

Before a transaction can access or modify an object, there must exist a policy that will allow the transaction to access the object. In this paper, we focus our attention to systems that support positive authorization policies only and consider simple kinds of authorization policies that are specified by *subject*, *target*, and *rights*. A subject can perform only those operations on the target that are specified by the rights. A subject can be a user, a group, a role or a process [17]. A target, in our model, is a data object, a group of data objects, or an object class. Next, we introduce the notion of a policy object.

Definition 1 (*Policy object*). A *policy object* is a triple $\langle SS_i, TS_i, RS_i \rangle$ where SS_i, TS_i, RS_i denote the subject set, the target set, and the set of access rights of the policy respectively. Elements in the subject set SS_i can perform only those operations on elements of the target set TS_i that are specified in RS_i.

Example 1. Let $P_i = \langle \{John, Joe\}, \{FileF, FileG\}, \{r, w, x\} \rangle$ be a policy object. This policy object gives subject *John* and *Joe* the privilege to Read, Write, and Execute *FileF* and *FileG*.

A policy object is one that stores information about a policy; we use the term policy object to distinguish them from the other data objects. Policy objects, like data objects, can be read and written. However, unlike ordinary data objects, policy objects can also be *deployed*.

Definition 2 (*Deploy*). A policy object P_j is said to be *deployed* if there exists a subject which is currently accessing a target by virtue of the privileges given by policy object P_j.

Example 2. The policy object P_i allows subject S_j to Read object O_k. Subject S_j initiates a transaction T_l that Reads O_k. While the transaction T_l Reads O_k, we say that the policy object P_i is deployed.

The environment we consider is one in which multiple subjects access and modify data and policy objects, while the policy objects are in effect. To deal with this scenario, we need some concurrency control mechanism. The objectives of our concurrency control mechanism are the following:

- Allow concurrent access to data objects and policy objects.
- 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. We begin by giving some definitions.

Definition 3 (*Conflicting operations*). Two operations are said to *conflict* if both operate on the same object and one of them is a Write operation.

The Write operation conflicts with a Read, Write, or a Deploy operation on the same object.

Definition 4 (*Transaction*). A *transaction* T_i is a partial order with ordering relation $<_i$ where

1. $T_i \subseteq \{r_i[x], w_i[x] | x \text{ is a data or policy object}\} \cup \{d_i[x] | x \text{ is a policy object}\} \cup \{a_i, c_i\}$;
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 $p \in T_i$, $p_i <_i t$; and
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 $d_i[x]$, $w_i[x] \in T_i$, then either $d_i[x] <_i w_i[x]$ or $w_i[x] <_i d_i[x]$.

Condition 1 defines the different kinds of operations in the transactions ($r_i[x]$, $w_i[x]$, $d_i[x]$, a_i, c_i denote Read operation on object x, Write operation on x, Deploy operation on x, Abort or Commit operation 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 every other operation of the transaction. Condition 4 requires that the partial order $<_i$ specify the order of execution of Read and Write operations on a common data or policy object. Condition 5 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 protocol [5]. Each data object O_i in our model is associated with two locks: read lock (denoted by $RL(O_i)$) and write lock (denoted by $WL(O_i)$). The locking rules for data objects are the same as the standard two-phase locking protocol [5]. A policy object P_j is associated with three locks: read lock (denoted by $RL(P_j)$), write lock (denoted by $WL(P_j)$) and deploy lock (denoted by $DL(P_j)$).

The locking rules for the policy objects are given in Table 1. *Yes* entry in the lock table indicates that the lock request is granted. *No* entry indicates that the lock request is denied. *Signal* entry in the lock table indicates that the lock request is granted, but only after the transaction currently holding the lock is aborted and the lock is released. The first row corresponds to the case where some transaction has a read lock on a policy object. The first column in the first row is a *Yes*—another transaction requesting a read lock on the same object is given the lock. The second column in the first row is a *No*—another transaction requesting a write lock on the same object is not given the lock. The entry in the first row third column is a *Yes*—this means, that if a

Table 1
Locking rules for policy objects

Has	Wants		
	RL	WL	DL
RL	Yes	No	Yes
WL	No	No	No
DL	Yes	Signal	Yes

transaction has a read lock on a policy object, and another transaction requests a deploy lock on the same object, then this lock request is granted. The second row corresponds to the case where a transaction has a write lock on a policy object. This row has all *No* entries; this means that no other transaction will be given any other locks to the object. Now consider the third row; this row corresponds to a transaction holding a deploy lock on a policy object. The entry in the first and third columns of the third row is a *Yes* signifying that if another transaction wants a read lock or a deploy lock on the object, it is granted. The entry in the second column of the third row is *Signal*. Signal means that the lock request is granted after the transaction currently holding the lock is aborted. For example, suppose some transaction T_i holds a deploy lock DL on a policy object, and another transaction T_j wishes to get the write lock WL and update the policy object. In such a case a signal is generated to abort T_i, after which T_i releases the DL lock and T_j is granted the WL lock. Note that, a transaction updating a policy (T_j) has a higher priority than a transaction (T_i) that deploys this policy; hence, we generate a Signal to abort the transaction deploying the policy (T_i).

Next we define what it means for a transaction in our model to be well-formed.

Definition 5 (*Well-formed transaction*). A transaction is *well-formed* if it satisfies the following conditions.

1. A transaction before reading or writing a data or policy object must deploy the policy object that authorizes the transaction to perform the operation.
2. A transaction before deploying, reading, or writing a policy object must acquire the appropriate lock.
3. A transaction before reading or writing a data object must acquire the appropriate lock.
4. A transaction cannot acquire a lock on a policy or data object if another transaction has locked the object in a conflicting mode.
5. All locks acquired by the transaction are eventually released.

Definition 6 (*Well-formed two-phase transaction*). A well-formed transaction T_i is *two-phase* if all its lock operations precede any of its unlock operations.

Example 3. Consider a transaction T_i that reads object O_j (denoted by $r_i(O_j)$) and then writes object O_k (denoted by $w_i(O_k)$). Policies P_m and P_n authorize the subject initiating transaction T_i the privilege to read object O_j and the privilege to write object O_k respectively. An example well-formed and two-phase execution of T_i consists of the following sequence of operations: $\langle DL_i(P_m), RL_i(O_j), d_i(P_m), r_i(O_j), DL_i(P_n), WL_i(O_k), d_i(P_n), w_i(O_k), UL_i(P_m), UL_i(P_n), UL_i(O_j), \ UL_i(O_k)\rangle$, 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 7 (*Policy-secure transaction*). A transaction is *policy-secure* if for every read or write operation that a transaction performs, there exists a policy that authorizes the transaction to perform the operation for the entire duration of the operation.

Note that, all transactions may not be policy-secure. For instance, suppose entity A can execute a long-duration transaction T_i by virtue of policy P_x. While A is executing T_i, P_x changes and no longer allows A to execute T_i. In such a case, if transaction T_i is allowed to continue after P_x has changed, then T_i will not be a policy-secure transaction.

We adopt the definitions of *history*, *conflict equivalence*, *committed projection of a history*, *serial history* and *conflict serializable history* from Bernstein et al. [5]. For the sake of completeness, we state them below.

Definition 8 (*History*). A *history* H defined over a set of transactions $\mathbf{T} = \{T_1, \ldots, T_m\}$, is a partial order of operations with ordering relation $<_H$ where:

1. $H = \bigcup_{i=1}^{m} T_i$;
2. $<_H \supseteq <_i$; and
3. for any two conflicting operations $p, q \in H$, either $p <_H q$ or $q <_H p$.

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

Definition 9 (*Conflict equivalent*). Two histories are *conflict equivalent* if (i) they are defined over the same set of transactions and have the same operations and (ii) they order conflicting operations of nonaborted transactions in the same way.

Definition 10 (*Serial history*). A history is *serial* if, for every pair of transactions, all of the operations of one transaction execute before any of the operations of the other.

Definition 11 (*Committed projection of a history*). The *committed projection of a history* is the history obtained by considering all operations of transactions that are committed in the history.

Definition 12 (*Conflict serializable history*). A history is *conflict serializable* if it's committed projection is conflict equivalent to some serial history.

Definition 13 (*Serialization graph of a history*). The *serialization graph* of a history H is a directed graph whose nodes are the transactions that are committed in H and whose edges are all $T_i \rightarrow T_j$ $(i \neq j)$ such that one of T_i's operations precedes and conflicts with T_j's operations in H.

Theorem 1. *A history is serializable iff its serialization graph is acyclic.*

Proof. This proof is identical to the proof of the same theorem that is given in Bernstein et al. [5].

Next we define what we mean by a *policy-secure history*.

Definition 14 (*Policy-secure history*). A history is *policy-secure* if all the transactions in its committed projection are policy-secure transactions.

Theorem 2. *A history H consisting of a set of well-formed and two-phase transactions is serializable.*

Proof. We prove this by contradiction. Assume that the history H, produced by transactions $\{T_1, T_2, \ldots, T_n\}$, is not serializable. Then the graph produced from this history contains a cycle. Without loss of generality, assume that this cycle is $T_1 \rightarrow T_2 \rightarrow T_3 \cdots T_n \rightarrow T_1$. 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 serializable is wrong. Therefore, the history H is serializable. □

Theorem 3. *A history H consisting of a set of well-formed and two-phase transactions is policy-secure.*

Proof. Assume that the history H is not policy-secure. This means that one or more transactions in the history H are not policy-secure. 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 the policy object. In other words, before T_i can access O_r, it has to obtain the deploy lock for a policy P_m that authorizes T_i to update O_r. Thus, when T_i initially accessed O_r, there was a policy P_m that allowed T_i to update O_r. So, the only possibility is that while T_i was updating O_r, the policy P_m got deleted or modified. But according to well-formed rules this is not possible. Any transaction T_j modifying the policy P_m has to obtain a write lock (*WL*) on P_m. Before the write lock on P_m can be granted, the transaction T_i holding the deploy lock (*DL*) has to be aborted and the deploy lock released (Table 1). 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-secure. Our assumption, that the history H is not policy-secure, is wrong. □

Although the lock based concurrency control approach provides policy-secure and serializable schedule, it is overly restrictive. A change of policy may not change the specific subject's access rights or may result in increased access rights; in such cases terminating valid access will result in poor performance. For this reason, we look at alternate semantic-based approach to policy modifications.

4. Towards a semantic-based approach for policy update

In this section we show how we can use semantics of the policy update operation to increase concurrency. The basic idea is to classify a policy update operation either as a *policy relaxation* or

as a *policy restriction* operation. 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. Policy restriction may decrease the access rights of one or more subjects; transactions executing by virtue of the privileges given by the original policy must be aborted. Before going into the details, we first give definitions of policy update operation, policy relaxation and policy restriction.

Definition 15 (*Policy update operation*). A *policy update operation* modifies a policy object $P_i = \langle SS_i, TS_i, RS_i \rangle$ to P_i' where P_i' is obtained by transforming either one or more of the following: SS_i to SS_i', TS_i to TS_i', or RS_i to RS_i'. SS_i', TS_i', RS_i' denote the modified subject set, target set and access rights set respectively.

Definition 16 (*Policy relaxation operation*). A *policy relaxation operation* is a policy update operation that does not decrease the access rights of any subject.

Example 4. The policy object $P_i = \langle \{John, Joe\}, \{FileF, FileG\}, \{r, w, x\} \rangle$ is changed to $P_i' = \langle \{John, Joe\}, \{FileF, FileG, FileH\}, \{r, w, x\} \rangle$. This is an example of policy relaxation because the access rights of subjects John and Joe have not decreased.

Definition 17 (*Policy restriction operation*). A *policy restriction operation* is a policy update operation that is not a policy relaxation operation.

Example 5. The policy object $P_i = \langle \{John, Joe\}, \{FileF, FileG\}, \{r, w, x\} \rangle$ is changed to $P_i' = \langle \{John\}, \{FileF, FileG\}, \{r, w, x\} \rangle$. This is not a policy relaxation because access rights of Joe are being curtailed. Therefore we treat this as policy restriction. A second example of policy restriction is P_i being changed to P_i'' where $P_i'' = \langle \{John\}, \{FileF, FileG, FileH\}, \{r, w, x\} \rangle$. In this case John's access rights are increased and Joe's rights are decreased. This is not a policy relaxation because Joe's access rights are being curtailed. Hence, this is classified as a policy restriction operation.

We now show how to classify a policy update operation as policy restriction or policy relaxation. Recall from Definition 1 that a policy P_i is expressed as a tuple $P_i = \langle SS_i, TS_i, RS_i \rangle$ where SS_i, TS_i, and RS_i denote the set of subjects, the set of targets, and the set of access rights of the policies. The policy P_i can be changed by modifying one or more of the following: SS_i, TS_i, RS_i. Note that, classifying a complex policy change operation as policy relaxation or restriction is not trivial. For this reason, we first consider only *atomic* changes to the policy, and later show how our approach scales up to *composite* changes.

Definition 18 (*Atomic change*). The transformation of policy object $P_i = \langle SS_i, TS_i, RS_i \rangle$ to P_i' is an *atomic change* A_j if only one of the following conditions is satisfied:

1. $P_i' = \langle SS_i', TS_i, RS_i \rangle$ and A_j is the transformation from SS_i to SS_i' by the application of a single operation on SS_i.
2. $P_i' = \langle SS_i, TS_i', RS_i \rangle$ and A_j is the transformation from TS_i to TS_i' by the application of a single operation on TS_i.

3. $P_i' = \langle SS_i, TS_i, RS_i' \rangle$ and A_j is the transformation from RS_i to RS_i' by the application of a single operation on RS_i.

In other words, an atomic change of policy P_i is a policy update in which only one of SS_i, TS_i, RS_i is changed by the application of one operation.

The different operations that can be applied depend on the specific policy language being used. For our case, the following is the set of operations.

Change of subject: The set of subjects can be changed by either the operation of (1) set union: a new set can be added to the existing set, or (2) set difference: a new set can be subtracted from the existing set.

Change of target: The set can be changed by either the operation of (1) set union: a new set can be added to the existing set, or (2) set difference: a new set can be subtracted from the existing set.

Change of rights: The set of access rights can be changed either by (1) set union: a new set of access rights can be added to the existing set or (2) set difference: a set of access rights can be removed from the existing set of rights.

Example 6. The policy object $P_i = \langle \{John, Joe\}, \{FileF, FileG\}, \{r, w, x\} \rangle$ is changed to $P_i' = \langle \{John\}, \{FileF, FileG\}, \{r, w, x\} \rangle$. The transformation of P_i to P_i' involved only one operation: the application of set difference operation on the subject set; hence, this is an atomic change. Suppose the above policy object P_i is changed to P_i'' where $P_i'' = \langle \{John\}, \{FileF, FileG, FileH\}, \{r, w, x\} \rangle$. This is not an atomic change because two operations are applied on P_i to change it to P_i'': (i) set difference operation is applied on the subject set and (ii) set union is applied on the target set.

Changing the subject set: Let $P_i = \langle SS_i, TS_i, RS_i \rangle$ be the original and $P_i' = \langle SS_i', TS_i, RS_i \rangle$ be the modified policy object. The modification is caused by the subject set being changed from SS_i to SS_i'. The subject set can be changed by any of the operations given below.

Set union ($SS_i' = SS_i \cup Add$). In this case the set of subjects in *Add* gets added to the existing set SS_i. Since the access rights of subjects in the set *Add* increase, this is classified as a policy relaxation operation.

Example 7. The policy object $P_i = \langle \{John, Joe\}, \{FileF, FileG\}, \{r,w,x\} \rangle$ is changed to $P_i' = \langle \{John, Joe, Denny, George\}, \{FileF, FileG\}, \{r, w, x\} \rangle$. This update operation occurred as a result of performing a set union operation on the subject set. The access rights of the new subjects *Denny* and *George* increased and the access rights of no other subjects decreased. Hence, this is a policy relaxation operation.

Set difference ($SS_i' = SS_i - Remove$). In this case the set *Remove* gets removed from the subject set SS_i. Since the access rights of the subjects in the set *Remove* diminishes, this operation is classified as a policy restriction operation.

Example 8. The policy object $P_i = \langle \{John, Joe\}, \{FileF, FileG\}, \{r, w, x\} \rangle$ is changed to $P_i' = \langle \{John\}, \{FileF, FileG\}, \{r, w, x\} \rangle$. The policy update was caused by performing a set difference operation on the subject set.

Changing the target (*rights*) *set*: Similar arguments can be made for changes in the target (rights) set. If the target (rights) set is increased by set union operation, then it is a policy relaxation operation. If the target (rights) set is decreased by set difference operation, then it is a policy restriction.

In short, if the atomic change A_j involved a set union, then it is a policy relaxation operation; otherwise, it is a policy restriction operation.

Example 9. The policy object $P_i = \langle\{John, Joe\}, \{FileF, FileG\}, \{r, w, x\}\rangle$ is changed to $P_i' = \langle\{John\}, \{FileF, FileG\}, \{r, w, x\}\rangle$. In this case the atomic change A_j applied the set difference operation on the subject set $\{John, Joe\}$ to transform it $\{John\}$. Thus, this is a policy restriction operation.

Having discussed about atomic changes, we are now in a position to discuss composite change.

Definition 19 (*Composite change*). The transformation of policy object P_i to P_i' is a *composite change* if it involves one or more atomic changes. A composite change C_k can be represented as a sequence of atomic changes $A_1, A_2, \ldots, A_n$; that is, $C_k = \langle A_1, A_2, \ldots, A_n\rangle$.

Example 10. The policy object $P_i = \langle\{John, Joe\}, \{FileF, FileG\}, \{r, w\}\rangle$ is changed to $P_i'' = \langle\{John\}, \{FileF, FileG, FileH\}, \{r, w\}\rangle$. This is a composite change because two atomic changes A_1, A_2 are applied to P_i to change it to P_i'': (i) A_1—set difference operation applied on the subject set $\{John, Joe\}$ to transform it to $\{John\}$, and (ii) A_2—the set union operation applied on the target set $\{FileF, FileG\}$ to transform it to $\{FileF, FileG, FileH\}$.

Example 11. Another example of composite change will be the policy object $P_i = \langle\{John, Joe\}, \{FileF, FileG\}, \{r, w\}\rangle$ changed to $P_i' = \langle\{John, Joe\}, \{FileF, FileG, FileH\}, \{r, w, x\}\rangle$. The following two atomic changes A_1 and A_2 were applied in this case: (i) A_1—set union operation on the target set $\{FileF, FileG\}$ to transform it to $\{FileF, FileG, FileH\}$, and (ii) A_2—set union operation on the access rights set $\{r, w\}$ to transform it to $\{r, w, x\}$.

Consider the composite change $C_k = \langle A_1, A_2, \ldots, A_n\rangle$. We classify whether this composite change C_k is a policy relaxation or a policy restriction in the following way: If all the atomic changes $(A_1, A_2, \ldots, A_n)$ making up the composite change are policy relaxation operations, then the composite change C_k is a policy relaxation operation; otherwise, C_k is a policy restriction operation.

Example 12. The composite change given in Example 10 classifies it as a policy restriction operation whereas the composite change in Example 11 classifies it as a policy relaxation operation.

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 operations specified on data objects are Read and Write. A policy object is now

associated with four operations: Read, Deploy, WriteRelax, WriteRestrict. The Read and Deploy operations are similar to those specified in Section 3. The Write operations on policy object are classified as WriteRelax or WriteRestrict. A WriteRelax operation is one in which the policy gets relaxed. All other write operations on the policy object are treated as WriteRestrict. Since the operations are different than those discussed in Section 3, we modify the definition of transaction given in Definition 4 to the following:

Definition 20 (*Transaction*). A *transaction* T_i is a partial order with ordering relation $<_i$ where

1. $T_i \subseteq \{r_i[x], w_i[x] | x \text{ is a data object}\} \cup \{d_i[x], r_i[x], ws_i[x], wx_i[x] | x \text{ is a policy object}\} \cup \{a_i, c_i\}$;
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 $p \in T_i$, $p_i <_i t$; and
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 $d_i[x]$, $ws_i[x] \in T_i$, then either $d_i[x] <_i ws_i[x]$ or $ws_i[x] <_i d_i[x]$.
6. if $d_i[x]$, $wx_i[x] \in T_i$, then either $d_i[x] <_i wx_i[x]$ or $wx_i[x] <_i d_i[x]$.

Condition 1 is changed from that in Definition 4 to reflect that the operations allowed on data objects are Read and Write and the operations allowed on policy objects are Read, Deploy, WriteRelax (denoted by *wx*), and WriteRestrict (denoted by *ws*). Conditions 2–4 are the same as given in Definition 4. Condition 5 given in Definition 4 is no longer applicable as there is no simple Write operation on policy objects; this condition is replaced by two conditions (Conditions 5 and 6 in Definition 20). Condition 5 specifies that if there is a Deploy operation on a policy object and a WriteRestrict operation on the same object, then the ordering relation $<_i$ must specify the order of the operations. Condition 6 specifies a similar condition for Deploy and WriteRelax operation.

Now we give the details of the locking rules. Each data object O_i is associated with two locks: read lock (denoted by $RL(O_i)$) and write lock (denoted by $WL(O_i)$). The locking rules for data objects are the same as the strict two-phase locking protocol [5]. Corresponding to the four operations on the policy object, we have four kinds of locks associated with policy objects: read locks (*RL*), deploy locks (*DL*), relax locks (WXL) and restrict locks (WSL). The locking rules are given in the Table 2.

The column with heading *Has* specifies the locks currently held by some transaction; the row with heading *Wants* specifies the locks requested by some transaction. The first row corresponds to the case where some transaction has a read lock on a policy object. The first column in the first row is a *Yes*—another transaction requesting a read lock on the same object is given the lock. The

Table 2
Locking rules for policy objects

Has	Wants			
	RL	WXL	WSL	DL
RL	Yes	No	No	Yes
WL	No	No	No	No
DL	Yes	Yes	Signal	Yes

second and third column in the first row is a *No*—another transaction requesting a relax or restrict lock on the same object is not given the lock. The entry in the first row fourth column is a *Yes*—this means, that if a transaction has a read lock on a policy object, and another transaction requests a deploy lock on the same object, then this lock request is granted. The second row corresponds to the case where a transaction has a relax lock on a policy object. This row has all *No* entries; this means that no other transaction will be given any lock to the object. The third row corresponds to the case where a transaction has a restrict lock on a policy object. This row has all *No* entries; this means that no other transaction will be given any lock to the object. Now consider the fourth row; this row corresponds to a transaction holding a deploy lock on a policy object. The entry in the first and fourth columns of the fourth row is a *Yes* signifying that if another transaction wants a read lock or a deploy lock on the object, it is granted. The entry in the second column of the fourth row is a *Yes*—if a transaction has a deploy lock on a policy object and another transaction wants a relax lock on the same object, then the lock request is granted. The entry in the third column of the fourth row is *Signal*. This is the case of some transaction T_i holding a deploy lock *DL* on a policy object, and another transaction T_j wanting to perform an update causing policy restriction. In this scenario, a signal is generated to abort T_i, after which T_i releases the *DL* lock and T_j is granted the WSL lock.

The definition of well-formed transaction is changed as follows:

Definition 21 (*Well-formed transaction*). A transaction is *well-formed* if it satisfies the following conditions.

1. A transaction before reading or writing a data object must deploy the policy object that authorizes the transaction to perform the operation.
2. A transaction before reading, write relaxing or write restricting a policy object must deploy the policy object that authorizes the transaction to perform the operation.
3. A transaction before reading or writing a data object must acquire the appropriate lock.
4. A transaction before deploying, reading, write relaxing, or write restricting a policy object must acquire the appropriate lock.
5. A transaction cannot acquire a lock on a policy or data object if another transaction has locked the object in a conflicting mode.
6. All locks acquired by the transaction are eventually released.

To ensure serializable and policy-secure histories, we require each transaction to be well-formed (Definition 21) and two-phase (Definition 6).

Theorem 4. *The locking rules provided in Table* 2 *provide more concurrency than those given in Table* 1.

Proof. Let **H1** and **H2** be the set of all possible histories generated by the locking rules given in Tables 1 and 2 respectively. We need to prove that **H1** is a proper subset of **H2**, that is, $\mathbf{H1} \subset \mathbf{H2}$. The proof will proceed in two parts: (i) First, we will prove that for any history H_1, if $H_1 \in \mathbf{H1}$, then $H_1 \in \mathbf{H2}$. (ii) Next, we will prove that for some history H_2 where $H_2 \in \mathbf{H2}$, $H_2 \notin \mathbf{H1}$. In the following paragraphs we outline the two proofs.

Proof of (*i*): Let $H_1 \in \mathbf{H1}$. We need to prove that $H_1 \in \mathbf{H2}$. Assume that $H_1 \notin \mathbf{H2}$. This is possible only if there is some operation in H_1 that cannot be scheduled by locking rules of Table 2. In other words, the locking rules of Table 2 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 operation is a Read, Write or Deploy operation on a policy object. Suppose this is a Read operation. The locking rule prevents read locks only when some other prior transaction has already acquired some kind of write lock on the object. But in this case the locking rule of Table 1 would have also disallowed the read lock and hence the Read operation in H_1. Thus, the operation is not a Read operation. Similar arguments can be made for Deploy or Write operations. Hence any operation in H_1 will also be permitted by the locking rules of Table 2. In other words $H_1 \in \mathbf{H2}$.

Proof of (*ii*): We need to show that for some $H_2 \in \mathbf{H2}$, $H_2 \notin \mathbf{H1}$. Let $H_2 = \langle d_i(P_x), r_i(O_a), d_j(P_y), wx_j(P_x), w_i(O_b), c_i, c_j \rangle$. H_2 is a history generated by interleaving the operations of two transactions T_i and T_j, where $T_i = \langle d_i(P_x), r_i(O_u), w_i(O_v), c_i \rangle$ and $T_j = \langle d_j(P_y), wx_j(P_x), c_j \rangle$. T_i Reads and Writes the data objects O_u and O_v respectively. T_i executes by virtue of policy P_x. T_j updates policy P_x by relaxing it. It performs a policy relaxation operation (indicated by *wx*). T_j executes due to the privileges given by policy P_y. Since the history H_2 can be generated by the locking rules given in Table 2, $H_2 \in \mathbf{H2}$. However, H_2 cannot be generated by the locking rules given in Table 1. This is because before the operation wx_j can take place, the locking protocol must obtain the *WL* on P_x. This will necessitate generating a Signal lock that will abort transaction T_i. Therefore, $H_2 \notin \mathbf{H1}$. □

5. Semantics-based concurrency control algorithm

Most cases of policy update will not be policy relaxations. Hence, to get improved concurrency it is essential to analyze the policy update transaction in greater details and study the interactions of this policy update with the other transactions. Note that, what impact a specific policy update will have on a transaction cannot be determined without analyzing the transaction and the policy change.

An example will help motivate the need for analyzing the interaction of policy update transactions with the regular transactions. To illustrate our point, we use the example of the hotel database. The hotel database has a set of objects, an integrity constraint on these objects, and different types of transactions, which we identify and explain below. The hotel database has different types of transactions, such as, *Reserve*, *Cancel* and *Report*. The *Reserve* transaction reserves a room for a guest. The *Reserve* transaction executes by virtue of the privileges given by a policy P_i. The policy P_i gives *Supervisors* and *Clerks* permission to Read and Write the objects associated with the *Reserve* Transaction. Another transaction *RemoveClerkRights* updates policy P_i; it eliminates the *Clerks* access rights to some objects. Now, if the *Reserve* and the *RemoveClerkRights* are executed concurrently, the *Reserve* transaction will be aborted because the rights in the policy P_i are being curtailed. [1] However, we can perform a detailed analysis to study

[1] Note that, *RemoveClerkRights* is a policy restriction operation as per Definition 17.

whether the interactions are problematic or not. Such analysis should not be adhoc but done in a formal manner.

We adopt the Z specification language [18] for performing our analysis. Z is based on set theory, predicate calculus, and a schema calculus to organize large specifications. Z is a model-oriented specification language and is suitable for modeling database transactions, database objects and integrity constraints [1,3,20]. The symbols used to explain the example are given in Table 3.

The partial specification of the hotel database appears in Fig. 1. For lack of space we do not show the specification of the *Cancel* and *Report* transactions in Fig. 1. Our specification assumes some given types, *Room* and *Guest*, which enumerate all possible rooms and all possible guests respectively.

States, as well as operations, are described in Z 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 schema *Hotel*, which defines the state of the database, lists the objects in the hotel. The partial injection *RM* relates guests to rooms. Our particular example does not allow guests to register multiple times, which is reflected by the fact that *RM* is an injective function. The example could be modified easily with different constraints. The partial function *ST* records the status of each room. Additional integrity constraints on the objects in the hotel database appear in the bottom part of *Hotel*. There is one such constraint: $\mathrm{dom}(ST \rhd$

Table 3
Relevant *Z* notation

$\mathbb{P}A$	Powerset of set A
$\setminus$	Set difference (also schema 'hiding')
$A \circ B$	Forward composition of A with B
$A \nrightarrow B$	Partial function from A to B
$A \rightarrowtail B$	Partial injective function from A to B
$A \rhd B$	Relation A with range restricted to set B
$\mathrm{dom}A$	Domain of relation A
$\mathrm{ran}A$	Range of relation A
$A \oplus B$	Function A overridden with function B
$x?$	Variable $x?$ is an input
$x!$	Variable $x!$ is an output
x	State variable x before an operation
x'	State variable x' after an operation
ΔA	Before and after state of schema A

$[Guest, Room]$

$Status ::= Available \mid Taken$

Hotel

$ST : Room \nrightarrow Status$

$RM : Guest \rightarrowtail Room$

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

Reserve

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

$g? \notin \mathrm{dom}\, RM$

$ST(r?) = Available$

$ST' = ST \oplus \{r? \mapsto Taken\}$

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

Fig. 1. Partial specification of the hotel database.

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

The transactions are specified by preconditions and postconditions. A transaction can execute only when its preconditions are satisfied. After the execution of the transaction, the postconditions are satisfied. Consider the *Reserve* transaction that reserves a room $r?$ for a guest $g?$ The *Reserve* transaction has preconditions (i) checks that the room asked for is available (indicated by $ST(r?) = Available$), and (ii) the guest must not be an existing guest (indicated by $g? \notin \text{dom}RM$ in the specification). The postcondition of *Reserve* is that the chosen room's status is made unavailable (indicated by $ST' = ST \oplus \{r? \mapsto Taken\}$), and the room is assigned to the guest (indicated by $RM' = RM \cup \{g? \mapsto r?\}$).

We assume that the hotel database has a role-based access control policy. To specify the policy, we have to extend the specification given in Fig. 1. In addition to the given types *Guest* and *Room* shown in Fig. 1, we introduce new given types *User*, *Target* and *Rights*. The given type *User* enumerates all possible users. *Targets* and *Rights* give all the possible objects that must be protected and all the possible rights that can be associated with these targets. There is also an enumerated type called *Role* that enumerates all possible roles.

The schema *Hotel*, which defines the state of the database, must now be changed to include information about access control policies. The new schema, which we call *RBACHotel*, has an additional object called AL that stores the the set of access control rights associated with the hotel database.

A transaction can be executed, provided that the user initiating the transaction has the privilege to perform the operations specified in the transaction. For example, the *Reserve* transaction can be executed by users assigned the role of *Supervisors*. Moreover, *Supervisors* should have the privilege to read and write the objects ST and RM.

Checking whether a user has the privilege to execute some transaction T_i and executing transaction T_i should take place atomically. Otherwise a policy update transaction T_j can execute in between and change the privilege that enables the user to execute the transaction T_i; this will result in T_i not being policy-secure. Thus, we can model the checking of a privilege as a part of the transaction. We change the specification of *Reserve* transaction to include extra preconditions that check for the satisfaction of privileges. We call this new specification *RBACReserve*. As shown in Fig. 2 the following preconditions are added: (i) $i? \in Supervisor$—checks if $i?$ is a *Supervisor*, (ii) $(Supervisor, ST, w) \in AL$—checks if *Supervisor* can write ST (iii) $(Supervisor, RM, w) \in AL$—checks if *Supervisor* can write RM). [2]

In addition, the hotel database has transactions that change the policy. *RemoveClerkRights* is one such transaction that removes the access right $r?$ of clerks on some specified target $t?$ The precondition for this transaction is that the clerk must be a subject of the policy that is being changed (indicated by $(Clerk, t?, r?) \in AL$). Additional preconditions check that the user has the privilege of executing this transaction (these are indicated by $u? \in Manager$ and $(Manager, AL, w) \in AL$). The postcondition is that the clerks are removed from the policy (indicated by $AL' = AL \setminus \{(Clerk, t?, r?)\}$). The other objects remain unchanged (indicated by $ST' = ST$ and $RM' = RM$).

[2] For lack of space we have not shown the checking of *Supervisor*'s *Read* privileges.

$[Guest, Room, Users, Target, Rights]$
$Roles ::= Manager \mid Supervisor \mid Clerk$
$Status ::= Available \mid Taken$

RBACReserve

$\Delta RBACHotel$
$g? : Guest$
$r? : Room$
$i? : User$

$i? \in Supervisor$
$(Supervisor, ST, w) \in AL$
$(Supervisor, RM, w) \in AL$
$g? \notin \mathrm{dom}\, RM$
$ST(r?) = Available$
$ST' = ST \oplus \{r? \mapsto Taken\}$
$RM' = RM \cup \{g? \mapsto r?\}$
$AL' = AL$

RBACHotel

$ST : Room \nrightarrow Status$
$RM : Guest \rightarrowtail Room$
$AL : \mathbb{P}(Roles \times Target \times Rights)$

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

RemoveClerkRights

$\Delta RBACHotel;\ t? : Target$
$r? : Rights;\ u? : User$

$u? \in Manager$
$(Manager, AL, w) \in AL$
$(Clerk, t?, r?) \in AL$
$AL' = AL \setminus \{(Clerk, t?, r?)\}$
$ST' = ST$
$RM' = RM$

Fig. 2. Partial specification of the RBAC hotel database.

Formally specifying the transactions will allow us to evaluate whether two transactions commute. Note that this semantic notion of commutativity is more general than the syntactic notion of commutativity [5].

Definition 22 (*Commutativity of transactions*). Two transactions T_i and T_j commute if the following holds: the final state and the output produced by the executing transaction T_i followed by T_j on some state S is the same as that obtained by executing T_j followed by T_i. Stated formally, transactions T_i and T_j commute if they satisfy the following property:

$$T_i \circ T_j \Longleftrightarrow T_j \circ T_i$$

$T_i \circ T_j$ formally defines the state resulting from the execution of T_i followed by T_j.

Before we proceed further, we make a distinction between types of transactions and instances of transactions. *Reserve*, *RemoveClerkRights* 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 number of instances can be generated from the finite number of transaction types. Next we define the commute sets of types of transactions.

Definition 23 (*Commute set of* $ty(T_i)$). The set of types of transactions that commute with transactions of type $ty(T_i)$.

For our specific example, we can prove that *Reserve* is in the commute set of *RemoveClerk-Rights* (the proof is shown in the Appendix A). Formal languages thus allow us to specify and automatically check (in some cases) the existence of properties such as the one stated above.

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 operations allowed on the data objects are Read and Write. Data objects are associated with two kinds of locks: read locks and write locks. The data objects follow the rules of the strict two-phase locking protocol [5]. The operations allowed on the policy objects are Read, Write and Deploy. Consequently, policy objects are associated with three kinds of locks: read, write and deploy locks. The algorithms for Read, Deploy, and Write operations for the policy objects are given below. These algorithms are executed atomically. Note that, the mechanisms for obtaining read locks and deploy locks are similar to the locking rules given in Table 1. The mechanism for obtaining write locks is, however, different.

Algorithm 1 (*Algorithm for Read*).

```
Procedure Process-Read (Ri(x))
begin
   if a transaction Tj holds a write lock on x
      exit;   /* Lock unavailable—Ti can retry later */
   give read lock to x
   accept (Ri(x))
end
```

Algorithm 2 (*Algorithm for Deploy*).

```
Procedure Process-deploy (Di(x))
begin
   if a transaction Tj holds a write lock on x
      exit;   /* Lock unavailable—Ti can retry later */
   give deploy lock to x
   accept (Di(x))
end
```

Algorithm 3 (*Algorithm for Write*).

```
Procedure Process-write (Wi(x))
begin
   if a transaction Tj holds a write lock on x
      exit;   /* Lock unavailable—Ti can retry later */
```

```
    if a transaction T_j holds a read lock on x
       exit;   /* Lock unavailable—T_i can retry later */
    for each T_j holding a deploy lock on x
       if ty(T_j) ∉ Commute-set(ty(T_i))
          abort T_j
    give write lock to x
    accept (W_i(x))
end
```

The algorithm for the Write operation on a policy object x initiated by transaction T_i is as follows. If another transaction T_j has a read or write lock on this policy object, then the request to obtain the write lock is denied. However, if another transaction T_j holds a deploy lock on x, then a check is made to ascertain whether $ty(T_j)$ is in the commute set of $ty(T_i)$. If $ty(T_j)$ is not in the commute set of $ty(T_i)$, then T_j is aborted and the write lock is given to T_i. On the other hand if $ty(T_j)$ is in the commute set of $ty(T_i)$, then T_j need not be aborted to give T_i the write lock.

Theorem 5. *The locking rules of the semantics-based approach presented in Section* 5 *provides more concurrency than the locking rules presented in Section* 4 (*given in Table* 2).

Proof. To prove this we must show that (i) any history produced by the locking rules of Table 2 can also be produced by the semantic approach and (ii) some history produced by the semantics-based approach of Section 5 cannot be produced by the locking rules of Table 2.

Proof of (*i*): The locking rules for data objects are the same in both the approaches. The rules for obtaining read locks and deploy locks on policy objects are the same. The two approaches differ with respect to the transactions that must be aborted when a transaction obtains a write lock on a policy object. In the locking rules given in Table 2 a policy relaxation operation does not abort transactions executing by virtue of the policy. Note that, these transactions are not aborted in the semantics-based approach of Section 5 either. This is because the commute set of a policy relaxation operation will include all transactions. Thus, any history produced by the locking rules given in Table 2 can also be produced by the semantics-based approach.

Proof of (*ii*): Consider the case where a *RBACReserve* transaction executes by virtue of the policy object *AL*. The transaction *RemoveClerkRights* modifies the policy object *AL* by removing the privileges of *Clerk*. Clearly, this is a policy restriction operation. The locking rules given in Table 2 would cause the abort of the *RBACReserve* transaction. However, the semantics-based approach allows *RBACReserve* to continue because it is in the commute set of *RemoveClerk-Rights*. □

6. Related work

Although a lot of work appears in the area of security policies (please refer to Damianou's thesis [8] for a survey), policy update has received relatively little attention. Some work has been

done in identifying interesting adaptive policies and formalization of these policies [9,23]. A separate work [22] 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 [7]. 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*, *target-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 target-domain. This work describes the implementation of a basic toolkit. The toolkit has a high-level language editor for specifying policies, a compiler for translating policies into enforcement components targeted to different platforms, a browser to view and manipulate the domains of subjects and objects to which policies apply. Thus, 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*, *target-domain* = */usr/local*, *access-list* = *read*, *write*. This policy allows all *Managers* to *read/write* all the files stored in the directory */usr/local*. Now the toolkit will allow 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.

Concurrency control in database systems is a well researched topic. Some of the important pioneering works have been described by Bernstein et al. [5]. Thomasian [25] 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 [4,10–16,24,26–28]. The use of semantic knowledge for solving other problems, such as, ensuring atomicity of secure multilevel transactions [2,20], and ensuring autonomy of local databases [19,21], have also been investigated by researchers.

7. Conclusion and future work

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 secure, real-time, concurrent policy update. The algorithms differ on the basis of the semantic knowledge that is used and in the degree of concurrency achieved.

A lot of work still remains to be done. In this work we considered very simple kinds of authorization policies. For example, we assume there exists exactly one policy by virtue of which any transaction has access privilege to some object. In a real world scenario, there may be multiple policies P_a, P_b, P_c, giving transaction T_i Read access to the object O_w. Suppose policy P_a is modified removing Read privilege from subject T_i. In such cases we would want transaction T_i to continue

to have access to the object O_w. Our algorithms should take into account these possibilities. In some cases precedence relationship may exist among these multiple policies. Policy update must take into account these precedence relationships. In future we plan to extend our approach to handle more complex kinds of authorization policies, such as, support for negative authorization policies, incorporating conditions in authorization policies, and support for specifying priorities in policies. Specifically, we plan to investigate how policies specified in the PONDER specification language [6] can be updated.

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 update. Examples of inconsistencies include conflicts in the policy specification and loss of functionality due to errors in the policy specification.

Acknowledgements

The author would like to thank the anonymous reviewers for their comments and suggestions. This work was partially supported by National Science Foundation under grant number IIS 0242258.

Appendix A

Theorem 6. $RBACReserve \circ RemoveClerkRights \iff RemoveClerkRights \circ RBACReserve$

Proof. LHS on simplification yields the following schema

$$
\begin{array}{|l}
\hline
RBACReserveCompRemoveClerk \\
\Delta Hotel;\ g? : Guest;\ r? : Room \\
i? : User;\ t? : Target;\ r? : Rights;\ u? : User \\
\hline
i? \in Supervisor \\
(Supervisor, ST, w) \in AL \\
(Supervisor, RM, w) \in AL \\
u? \in Manager \\
(Manager, AL, w) \in AL \\
(Clerk, t?, r?) \in AL \\
g? \notin \operatorname{dom} RM \\
ST(r?) = Available \\
ST' = ST \oplus \{r? \mapsto Taken\} \\
RM' = RM \cup \{g? \mapsto r?\} \\
AL' = AL \setminus \{Clerk, t?, r?\} \\
\hline
\end{array}
$$

The RHS on simplification yields the following schema:

$$
\begin{array}{|l}
\hline
RBACRemoveClerkCompReserve \\
\Delta Hotel;\ g? : Guest;\ r? : Room \\
i? : User;\ t? : Target;\ r? : Rights;\ u? : User \\
\hline
i? \in Supervisor \\
(Supervisor, ST, w) \in AL \setminus \{(Clerk, t?, r?)\} \\
(Supervisor, RM, w) \in AL \setminus \{(Clerk, t?, r?)\} \\
u? \in Manager \\
(Manager, AL, w) \in AL \\
(Clerk, t?, r?) \in AL \\
g? \notin \mathrm{dom}\, RM \\
ST(r?) = Available \\
ST' = ST \oplus \{r? \mapsto Taken\} \\
RM' = RM \cup \{g? \mapsto r?\} \\
AL' = AL \setminus \{(Clerk, t?, r?)\} \\
\hline
\end{array}
$$

Note that the two schemas have identical state change. The only way that the schemas differ is in two of their preconditions. If we can show that these two sets of preconditions are equivalent, then we can conclude that the schemas are also equivalent. In other words, to show the equivalence of two schemas we need to prove the following:

(i) $(Supervisor, ST, w) \in AL \setminus \{(Clerk, t?, r?)\} \iff (Supervisor, ST, w) \in AL.$
(ii) $(Supervisor, RM, w) \in AL \setminus \{(Clerk, t?, r?)\} \iff (Supervisor, RM, w) \in AL.$

We prove (i) only. (ii) can be proved using similar arguments.

(i) $(Supervisor, ST, w) \in AL \setminus \{(Clerk, t?, r?)\} \iff (Supervisor, ST, w) \in AL.$

To prove (i) we must prove the following implications:

(a) $(Supervisor, ST, w) \in AL \setminus \{(Clerk, t?, r?)\} \Rightarrow (Supervisor, ST, w) \in AL,$
(b) $(Supervisor, ST, w) \in AL \Rightarrow (Supervisor, ST, w) \in AL \setminus \{(Clerk, t?, r?)\}.$

Note that, implication (a) is always true. This is because an element x present in a set A is also present in a superset of A. Implication (b) is also true because $(Supervisor, ST, w) \notin \{(Clerk, t?, r?)\}$. □

References

[1] P. Ammann, S. Jajodia, I. Ray. Using formal methods to reason about semantics-based decomposition of transactions, in: Proceedings of the 21st International Conference on Very Large Databases, Zurich, Switzerland, September 1995, pp. 218–227.

[2] P. Ammann, S. Jajodia, I. Ray, Ensuring atomicity of multilevel transactions, in: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 1996, pp. 74–84.
[3] P. Ammann, S. Jajodia, I. Ray, Applying formal methods to semantic-based decomposition of transactions, ACM Transactions on Database Systems 22 (2) (1997) 215–254.
[4] B.R. Badrinath, K. Ramamritham, Semantics-based concurrency control: beyond commutativity, ACM Transactions on Database Systems 17 (1) (1992) 163–199.
[5] P.A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, Reading, MA, 1987.
[6] N. Damianou, N. Dulay, E. Lupu, M. Sloman. The Ponder Policy Specification Language, in: Proceedings of the Policy Workshop, Bristol, UK, January 2001.
[7] N. Damianou, T. Tonouchi, N. Dulay, E. Lupu, 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.
[8] N.C. Damianou, A Policy Framework for Management of Distributed Systems, Ph.D. Thesis, Imperial College of Science, Technology and Medicine, University of London, London, UK, 2002.
[9] 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.
[10] A.A. Farrag, M.T. Özsu, Using semantic knowledge of transactions to increase concurrency, ACM Transactions on Database Systems 14 (4) (1989) 503–525.
[11] H. Garcia-Molina, Using semantic knowledge for transaction processing in a distributed database, ACM Transactions on Database Systems 8 (2) (1983) 186–213.
[12] M. Herlihy, Extending multiversion time-stamping protocols to exploit type information, IEEE Transactions on Computers 36 (4) (1987) 443–448.
[13] M.P. Herlihy, W.E. Weihl, Hybrid concurrency control for abstract data types, Journal of Computer and System Sciences 43 (1) (1991) 25–61.
[14] H.F. Korth, G. Speegle, Formal aspects of concurrency control in long-duration transaction systems using the NT/PV model, ACM Transactions on Database Systems 19 (3) (1994) 492–535.
[15] H.F. Korth, G.D. Speegle, Formal model of correctness without serializability, in: Proceedings of ACM-SIGMOD International Conference on Management of Data, June 1988, pp. 379–386.
[16] Nancy A. Lynch, Multilevel atomicity—a new correctness criterion for database concurrency control, ACM Transactions on Database Systems 8 (4) (1983) 484–502.
[17] J. Park, R. Sandhu, Towards usage control models: beyond traditional access controls, in: Proceedings of the 7th ACM Symposium on Access Control Models and Technologies, Monterey, California, June 2002, pp. 57–64.
[18] B. Potter, J. Sinclair, D. Till, An Introduction to Formal Specification and Z, Prentice-Hall, New York, NY, 1991.
[19] R. Rastogi, H.F. Korth, A. Silberchatz, Exploiting transaction semantics in multidatabase systems, in: Proceedings of the International Conference on Distributed Computing Systems, Vancouver, Canada, June 1995, pp. 101–109.
[20] I. Ray, P. Ammann, S. Jajodia, A semantic model for multilevel transactions, Journal of Computer Security 6 (3) (1998) 181–217.
[21] I. Ray, P. Ammann, S. Jajodia, Using semantic correctness in multi-databases to achieve local autonomy, distribute coordination, and maintain global integrity, Information Sciences 129 (1-4) (2000) 155–195.
[22] E.A. Schneider, W. Kalsow, L. TeWinkel, M. Carney, Experimentation with Adaptive Security Policies. Technical Report RL-TR-96-82, Rome Laboratory, Air Force Material Command, Rome, NY, June 1996.
[23] E.A. Schneider, D.G. Weber, T. de Groot, Temporal Properties of Distributed Systems. Technical Report RADC-TR-89-376, Rome Air Development Center, Rome, NY, September 1989.
[24] L. Sha, J.P. Lehoczky, E.D. Jensen, Modular concurrency control and failure recovery, IEEE Transactions on Computers 37 (2) (1988) 146–159.
[25] A. Thomasian, Concurrency control: methods, performance and analysis, ACM Computing Surveys 30 (1) (1998) 70–119.
[26] H. Wachter, A. Reuter, The ConTract model, in: A.K. Elmagarmid (Ed.), Database Transaction Models for Advanced Applications, Morgan Kauffman, San Mateo, CA, 1992, pp. 219–263.

[27] W.E. Weihl, Specification and Implementation of Atomic Data Types, Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, 1984.
[28] W.E. Weihl, Commutativity-based concurrency control for abstract data types, IEEE Transactions on Computers 37 (12) (1988) 1488–1505.

Indrakshi Ray received her B.E. degree in Computer Science and Technology from the Bengal Engineering College, India in 1988, and the M.E. degree of Computer Science and Engineering from Jadavpur University, India in 1991. She obtained her Ph.D. in Information Technology from George Mason Universtiy in 1997. From 1997 to 2001, she was an Assistant Professor at the University of Michigan-Dearborn. In 2001, she joined the Colorado State University as an Assistant Professor. Her research spans the areas of computer security, e-commerce, database systems and formal methods. She has published over 30 refereed journal and conference papers in these areas. She served as the Program Chair for the IFIP WG 11.3 Conference on Data and Applications Security, 2003. She is a program commitee member for numerous conferences related to computer security, e-commerce, and database systems.