Concurrent and Real-Time Update of Access Control Policies*

Indrakshi Ray and Tai Xin

Department of Computer Science
Colorado State University
Email: {iray,xin}@cs.colostate.edu

Abstract. Access control policies are security policies that govern access to resources. *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. We consider an environment in which different kinds of transactions execute concurrently some of which are policy update transactions. Updating policy objects while they are deployed can lead to potential security problems. We propose two algorithms that not only prevent such security problems, but also ensure serializable execution of transactions. The algorithms differ on the degree of concurrency provided.

1 Introduction

An enterprise security policy is subject to adaptive, preventive and corrective maintenance. Since security 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, policy updates should not be through adhoc operations but done through well-defined *transactions* that have been previously analyzed.

An important issue that must be kept in mind about policy update transactions is that some policies may require *real-time updates*. We use the term real-time update of a policy to mean that the policy will be changed while it is in effect and this change will be enforced immediately. An example will help motivate the need for real-time updates of policies. Suppose the user $John$, by virtue of some policy P, has the privilege to execute a long-duration transaction that prints a large volume of sensitive financial information kept 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 of executing 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 causing financial loss to the company. In this case, the policy was updated correctly but not enforced immediately resulting in

* This work was done in part while the author was working as a Visiting Faculty at Air Force Research Laboratory, Rome, NY in Summer 2002.

a security breach. 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 policy updates.

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 and modified through transactions. Transactions performing operations on database objects must have the privilege 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 accessed and modified by transactions. We are considering an environment in which different kinds of transactions execute concurrently some of which are policy update transactions. In other words, a policy may be updated while transactions are executing by virtue of this policy. We propose two different algorithms that allow for concurrent and real-time updates of policies. The algorithms differ with respect to the degree of concurrency achieved.

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

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 can be initiated by a user, a group, or another process. A transaction inherits the access privileges of the entity initiating it. 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: authorization policies[1]. An authorization policy specifies what operations an entity can perform on another entity. We focus our attention to 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. There is no explicit policy that specifies what operations an entity is *not allowed* to perform on another entity. The absence of an

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

explicit 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.

We consider simple kinds of authorization policies that are specified by *subject*, *object*, and *rights*. A subject can be a user, a group of users or a process. An object, in our model, is a data object, a group of data objects, or an object class. A subject can perform only those operations on the object that are specified in the rights.

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

In a database, policies are stored in the form of policy objects.

Definition 2. *A* policy object P_i *consists of the triple* $< S_i, O_i, R_i >$ *where* S_i*,* O_i*,* R_i *denote the subject, the object, and the access rights of the policy respectively. Subject* S_i *can perform only those operations on the object* O_i *that are specified in* R_i*.*

Example 1. Let $P =< John, FileF, \{r, w, x\} >$ be a policy object. This policy object gives subject John the privilege to Read, Write, and Execute $FileF$.

Before proceeding further, we discuss how to represent the access rights. The motivation for this representation will be clear in Section 4.

Definition 3. *Let* $\mathbf{O}_i = \{o_1, o_2, \ldots, o_n\}$ *be the set of all the possible operations that are specified on Object* O_i*. The set of operations in* $\mathbf{O_i}$ *are ordered in the form of a sequence* $< o_1, o_2, \ldots, o_n >$*. We* represent an access right on the object O_i *as an* n*-element vector* $[i_1 i_2 \ldots i_n]$*. If* $i_k = 0$ *in some access right* R_j*, then* R_j *does not allow the operation* o_k *to be performed on the object* O_i*.* $i_k = 1$ *signifies that the access right* R_j *allows operation* o_k *to be performed on the object* O_i*. The total number of access rights that can be associated with object* O_i *equals* 2^n*.*

Example 2. Let $< r,w,x >$ be the operations allowed on a file F. The access right $R_1 = [001]$ signifies that r, w operations are not allowed on the file F but the operation x is permitted on File F. The access right $R_2 = [101]$ allows r and x operations on the file F but does not allow the w operation.

Definition 4. *The set of all access rights associated with a object* O_i *having* n *operations forms a* partial order *with the ordering relation* $\geq_{O_i}$*. The ordering relation is defined as follows: Let* $R_j[i_k]$ *denote the* i_k*-th element of access right* R_j*. Then* $R_p \geq_{O_i} R_q$*, if* $R_p[i_k] = R_q[i_k]$ *or* $R_p[i_k] > R_q[i_k]$*, for all* $k = 1 \ldots n$*.*

Definition 5. *Given two access rights* R_p *and* R_q *associated with an object* O_i *having* n *operations, the* least upper bound *of* R_p *and* R_q*, denoted as* $lub(R_p, R_q)$ *is computed as follows. For* $k = 1 \ldots n$*, we compute the* i_k*-th element of the least upper bound of* R_p *and* R_q*:* $lub(R_p, R_q)[i_k] = R_p[i_k] \vee R_q[i_k]$*. The* n*-bit vector obtained from the above computation will give us the least upper bound of* R_p *and* R_q*.*

Definition 6. *Given two access rights R_p and R_q associated with an object O_i having n operations, the* greatest lower bound *of R_p and R_q, denoted as $glb(R_p, R_q)$ is computed as follows. For $k = 1 \ldots n$, we compute the i_k-th element of the greatest lower bound of R_p and R_q: $glb(R_p, R_q)[i_k] = R_p[i_k] \wedge R_q[i_k]$. The n-bit vector obtained from the above computation will give the greatest lower bound of R_p and R_q.*

Since each pair of access rights associated with an object have a unique least upper bound and a unique greatest lower bound, the access rights of an object can be represented as a lattice.

Definition 7. *The set of all possible access rights on an object O_i can be represented as a lattice which we term the* access rights lattice of object O_i. *The notation $ARL(O_i)$ denotes the set of all nodes in the access rights lattice of object O_i.*

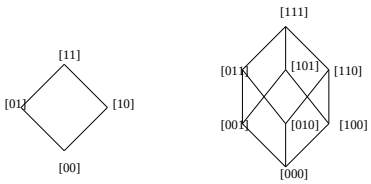

Fig. 1. Representing Possible Access Control Rights of Objects

Figure 2(a) shows the possible access rights associated with a file having only two operations: Read and Write. The most significant bit denotes the Read operation and the least significant bit denotes the Write operation. The lower bound labeled as Node 00 signifies the absence of Read and Write privilege. The Node 01 signifies that the subject has Write privilege but does not have Read privileges. The Node 10 signifies that the subject has Read privilege but no Write privilege. The Node 11 indicates that the subject has both Read and Write privileges. Figure 2(b) shows the possible access rights associated with a object having three operations.

Next we define a policy in terms of the access rights lattice.

Definition 8. *A* policy *P_i maps a subject S_i's access privilege to a Node j in the access rights lattice of the object O_i. Formally, $P : S \rightarrow (ARL(O))$.*

Definition 9. *A* policy update *is an operation that changes some policy object $P_i =< S_i, O_i, R_i >$ to $P_i' =< S_i, O_i, R_i' >$ where P_i' is obtained by transforming R_i to R_i'. Let R_i, R_i' be mapped to Node j, Node k of $ARL(O_i)$ respectively. The update of policy object P_i changes the mapping of the subject S_i's access privilege from Node j to Node k in the access rights lattice of object O_i.*

Having given some background on the policies, we are now in a position to discuss policy objects. Recall from Definition 2 that policies are stored in the database in the form of policy objects. Next we describe the operations associated with the policy objects. Policy objects, like data objects, can be read and written. However, unlike ordinary data objects, policy objects can also be *deployed*.

Definition 10. *A policy object P_j is said to be* deployed *if there exists a subject in P_j that is currently accessing an object in P_j by virtue of the privileges given by policy object P_j.*

Example 3. Suppose 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 are considering is one in which multiple users will be accessing and modifying data and policy objects, while the policy objects are deployed. To deal with this scenario, we need some concurrency control mechanism. The objectives of our concurrency control mechanism are the following: (1) Allow concurrent access to data objects and policy objects. (2) Prevent security violations arising due to policy updates.

3 A Simple Algorithm for Policy Updates

In our model, each data object is associated with Read and Write operations. A policy object is associated with three operations: Read, Write and Deploy. We now give some definitions.

Definition 11. *Two operations are said to* conflict *if both operate on the same data object and one of them is a Write operation.*

Definition 12. *A* transaction *T_i is a partial order with ordering relation $<_i$ where*

1. *$T_i \subseteq \{r_i[x], w_i[x] \mid x$ is a data or policy object $\} \cup \{d_i[x] \mid x$ 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.

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

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

(a) Syntax-Based Algorithm

Has \ *Wants*	RL	WXL	WSL	DL
RL	Yes	No	No	Yes
WXL	No	No	No	No
WSL	No	No	No	No
DL	Yes	Yes	Signal	Yes

(b) Semantics-Based Algorithm

Table 1. Locking Rules for Policy Objects

same as the standard two-phase locking protocol [3]. 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(a). *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.

Definition 13. *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 14. *A well-formed transaction T_i is* two-phase *if all its lock operations precede any of its unlock operations.*

Example 4. 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 of a well-formed and two-phase execution of T_i consists of the following sequence of operations: $< DL_i(P_m),\ RL_i(O_j),\ d_i(P_m),\ r_i(O_j),\ DL_i(P_n),\ WL_i(O_k),\ d_i(P_n),\ w_i(O_k),\ UL_i(P_m),\ UL_i(P_n),\ UL_i(O_j),\ UL_i(O_k) >$, 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 15. *A transaction is* policy-secure *if for every operation that a transaction performs, there exists a policy that authorizes the transaction to perform 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 borrow the definitions of *history*, and *serializable history* from Bernstein et al. [3]. Next we define what we mean by a *policy-secure* history.

Definition 16. *A history is* policy-secure *if all the transactions in the history are policy-secure transactions.*

The lock based concurrency control approach provides policy-secure and serializable histories.

4 Concurrency Control using the Semantics of Policy Update

The approach presented in Section 3 is overly restrictive. A change of policy may result in increased access privileges; in such cases terminating valid access will result in poor performance. This motivates us to classify a policy update operation either as a *policy relaxation* or as a *policy restriction* operation. Policy relaxation causes increase in subject's access rights; transactions executing by virtue of a policy need not be aborted when the policy is being relaxed. On the other hand, a policy restriction does not increase the access rights of the subject. To ensure policy-secure transactions, we must abort the transactions that are executing by virtue of the policy that is being restricted. Before going into the details, we first give the definitions of policy relaxation and policy restriction.

Definition 17. *A* policy relaxation operation *is a policy update that increases the access rights of the subject. Let the policy object $P_i =< S_i, O_i, R_i >$ be changed to $P_i' =< S_i, O_i, R_i' >$. Let Let R_i, R_i' be mapped to the nodes k, j respectively in $ARL(O_i)$. A policy update operation is a policy relaxation operation if $lub(k, j) = j$.*

Example 5. Let the operations allowed on $FileF$ be $< r, w, x >$. Suppose the policy $P_i =< John, FileF, [001] >$ is changed to $P_i' =< John, FileF, [101] >$. This is an example of policy relaxation because the access rights of subject John has increased. Note that $lub([001], [101]) = [101]$. Thus, this is a policy relaxation.

Definition 18. *A* policy restriction operation *is a policy update operation that is not a policy relaxation operation. Let the policy object $P_i =< S_i, O_i, R_i >$ be changed to $P_i' =< S_i, O_i, R_i' >$. Let Let R_i, R_i' be mapped to the nodes k, j respectively in $ARL(O_i)$. A policy update operation is a policy restriction operation if $lub(k, j) \neq j$.*

Example 6. Let the operations allowed on $FileF$ be $< r, w, x >$. Suppose the policy $P_i =< John, FileF, [001] >$ is changed to $P_i' =< John, FileF, [110] >$. This is an example of policy restriction because the access rights of subject John has not increased. Note that, $lub([001], [110]) = [111]$. Since $lub([001], [110]) \neq [110]$, this is an example of policy restriction.

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. Distinguishing between policy restriction and relaxation will increase concurrency. 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 definitions of transaction and well-formed transaction (Definitions 12 and 13).

Definition 19. *A* transaction T_i *is a partial order with ordering relation* $<_i$ *where*

1. $T_i \subseteq \{r_i[x], w_i[x] \mid x$ *is a data object* $\} \cup \{d_i[x], r_i[x], ws_i[x], wx_i[x] \mid x$ *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 12 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 2,3, and 4 are the same as given in Definition 12. Condition 5 given in Definition 12 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 19). 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. The locking rules for data objects are as given in Section 3. 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 1(b).

Definition 20. *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 (Def. 20) and two-phase (Def. 14).

5 Related Work

Although a lot of work appears in the area of security policies [6], policy updates have received relatively little attention. Some work has been done in identifying interesting adaptive policies and formalization of these policies [7, 13]. A separate work [12] 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 [5]. 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. This work allows new subjects to be added or existing subjects to 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*. 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. [3]. Thomasian [14] 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 [1, 2, 8–11].

6 Conclusion and Future Work

Real-time updates of policy is an important problem for both the commercial and the military sector. In this paper we focus on real-time update of access control policies in a database system. We propose two algorithms for real-time update of access control

policies. The algorithms generate serializable and policy-secure histories and provide different degrees of concurrency.

A lot of work still remains to be done. In this work we assume there exists exactly one policy by virtue of which any subject has access privilege to some object. In a real-world scenario multiple policies may be specified over the same subject and object. The net effect of these multiple policies depend on the semantics of the application. Changing the policies in such situations is non-trivial. 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, support for specifying priorities in policies. Specifically, we plan to investigate how policies specified in the PONDER specification language [4] can be updated.

References

1. 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.
2. B.R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, March 1992.
3. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.
4. N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Proceedings of the Policy Workshop*, Bristol, U.K., January 2001.
5. 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.
6. 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.
7. 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.
8. H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.
9. 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.
10. H. F. Korth and G. Speegle. Formal aspects of concurrency control in long-ouration transaction systems using the NT/PV model. *ACM Transactions on Database Systems*, 19(3):492–535, September 1994.
11. Nancy A. Lynch. Multilevel atomicity—A new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.
12. 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.
13. 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.
14. A. Thomasian. Concurrency Control: Methods, Performance and Analysis. *ACM Computing Surveys*, 30(1):70–119, 1998.