CERIAS Tech Report 2006-42

VERIFICATION OF DATABASE TRANSACTION LOCK MANAGEMENT IN THE PRESENCE OF ROLE BASED ACCESS CONTROL POLICY

by Arjmand Samuel, Arif Ghafoor

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Verification of Database Transaction Lock Management in the Presence of Role Based Access Control Policy

Arjmand Samuel and Arif Ghafoor
School of Electrical and Computer Engineering
Purdue University, West Lafayette, USA
{amsamuel, ghafoor}@purdue.edu

Abstract

In a computing environment where access to system resources is controlled by an access control policy and execution of database transactions is dictated by database locking policy, interaction between the two policies can result in constraints restricting execution of transactions. We present a methodology for the verification of database transaction requirements in a Role Based Access Control (RBAC) environment. Specifically, we propose a step by step approach for the extraction of implicit requirements of a database transaction, and present a mechanism whereby these requirements can be verified against an RBAC policy representation. Based on the requirements of database transaction, we define feasible states of the access control policy which allow the transaction to be executed. We also illustrate the interaction of multiple database transactions executing in a single security environment. Finally, we define conditions in an access control policy, which allow the execution of a database transaction without relying on the underlying database locking policy for serializability and deadlock avoidance.

1. Introduction

A typical computing environment is controlled by a number of subsystem policies. Examples of these policies are the access control policy, CPU scheduling policy, database record locking policy, semaphore locking policy, memory management policy, etc. Each of the subsystem policy controls its relevant component independently with no knowledge of the rest of the subsystem policies; contrary to the fact that the subsystems themselves are interrelated and mutually dependent. The indirect interaction between the subsystem policies at times gives rise to drastic consequences. The actions of the security subsystem are controlled by an access control policy, which dictates circumstances in which object permissions are granted to authenticated users. The database locking policy allows more than one transaction to access a single object. In a computing system where access control policy and database locking policy co-exist, interaction between the two creates an interesting interdependence, which we address in our discussion here.

In this paper we consider the RBAC model, which has emerged as a de-facto model for specifying security requirements of large organizations. Its strength lies in the definition of user roles more akin to the functional responsibilities of users in the organization and abstracting object permissions as roles [10, 11]. RBAC provides us with expressiveness to define object permissions and system states under which these permissions can be granted to users. In this paper we model RBAC security policy in the form of a Finite State Machine (FSM). This model allows us to explicitly represent security system properties and use them to verify different system constraints dictated by the database transactions and locking policies.

A database system in its most general form consists of a set of objects and a set of transactions [8, 9]. An important issue in database design is the ability of the database engine to run multiple transactions simultaneously [2], while ensuring consistency of the underlying data, also called serializability [5, 16, 15]. A common approach to ensure serializability is to have a locking policy, which guarantees that all objects accessed by the transaction are locked. There are a number of well known locking policies which not only ensure serializability but also guarantee freedom from deadlock. In this paper, we will consider the two-phase locking policy and the rooted tree locking policy.

Our contributions in this paper are: 1) Verification of database transaction requirements against an RBAC policy, resulting in security states which permit execution of the verified transaction. 2) Verification of a database transaction, resulting in a set of security states of RBAC policy, which allow transaction to execute without relying on database locking policy. In order to illustrate the above

contributions, we present an example based approach and consider typical RBAC policies and database transactions which conform to the 2-phase locking protocol and the rooted tree locking protocol [8].

The remainder of this paper is organized as follows. Section 2 gives preliminary definitions for RBAC policies, its representation as an FSM, and framework for representation of the database transaction. Section 3 states three database transaction properties and provides the verification methodology. Section 4 discusses situations in which the verification methodology can be used effectively. It also lists some of the limitations of the approach. Section 5 presents related work and Section 6 concludes the paper.

2. Preliminary definitions

As mentioned above, we will use RBAC as the security policy definition environment. For this paper we will define RBAC policy *P* as a 5-tuple *(U, R, P, URA, PR)*, where

- *U* and *R* are the finite set of users and roles
- *P* is the set of permissions
- $URA \subseteq U \times R$ is a set of allowable user to role activations
- $PR \subseteq P \times R$ is a set of allowable permission-to-role assignments. Permission to access an object $O1$ is granted to one role at a time only e.g. (P_1, R_1) and (P_1, R_2) are not allowed in the same set.

Any user activating a role is assumed to be assigned to that particular role. The above definition is adequate to illustrate the verification of database transaction requirements in the context of RBAC security policy.

2.1. Modeling of the Role Based Access Control Policy

We use the state based representation of the role based access control policy. This representation allows us to define implicit properties of the policy and analyze them in an explicit manner. In the following, we describe a FSM based model of an access control policy. The FSM model *M* is defined as *M*=$\{S, I, E\}$, where *S* is the set of states, *I* is the set of inputs and *E* is the set of edges which connects one state to another. A binary state variable defines each of the states in set S. The number of bits of the state variable, S_b=number of authenticated users in the system + number of objects being accessed by the users. The set $I = \{P_O^{AR}, P_O^{DR}, U_R^A, U_R^D\}$ is a finite set of allowable inputs, where P_O^{AR} refers to members of role R being granted permission to access an object O. P_O^{DR} is the permission revocation for members of role R on object O. Each permission in this set refers to a either a read or a write permission on a physical database object, although in our discussion we do not differentiate between the two. U_R^A is the user role activation of user U in role R and U_R^D is the user role deactivation of user U from role R. E, the set of edges, consists of directed arrows which define the direction of the transition. In order to differentiate between the invocation of a permission and revocation of permission, we use solid edges for the first and dashed edges for the later. Similarly, activation of a user in a role is depicted by a solid edge and de-activation of a user from a role is represented by a dashed edges. De-activation of users and revocation of permissions are shown in the FSM only when required.

An example RBAC access control policy K1 is given below, which we will use as a running example in this paper.

$U = \{U1, U2\}$
$R = \{R1\}$
$P = \{P1, P2\}$
$URA = \{(U1, R1), (U2, R2)\}$
$PR = \{(P1, R1), (P2, R1)\}$

K1 has two users $U1$ and $U2$. Each user can activate a role $R1$, as represented by the set URA above. There are two objects, $O1$ and $O2$. Permissions to access these objects are represented by $P1$ and $P2$, respectively, which is granted to any user activating role $R1$, represented by the set PR above. The state variable is of the form $(x_1x_2x_3x_4)$, which is a binary representation of states. Bits (x_1) and (x_2) represent the state of users $U1$ and $U2$, respectively. If user $U1$ has activated the role $R1$, bit (x_1) is set to 1, and 0 otherwise. Similarly, if user $U2$ has activated role $R1$, the bit (x_2), is set to 1, and 0 otherwise. The bits (x_3) and (x_4) represent the invocation of permissions $P1$ and $P2$ to role $R1$. If $P2$ is granted to role $R1$, bit (x_4) will be set to 1, and 0 otherwise. Complete list of the states is given in Figure 1(a), along with details of the semantics of the state. As pointed out earlier, in order to keep the FSM simple, transitions like de-activation of users and revocation of permissions to objects are not shown but exist in the policy. We represent K1 as an FSM in Figure 1(b).

2.2. Representation of the Database Transaction

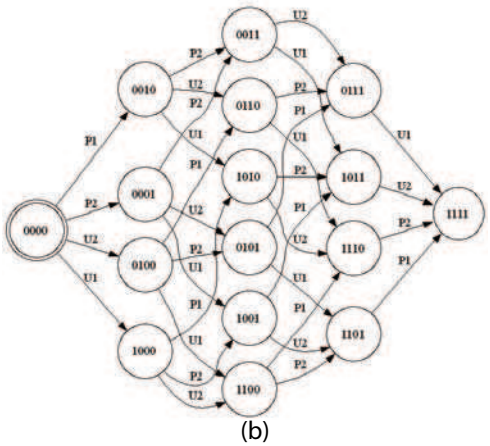
Next, we present definitions which will be used to represent and analyze transaction constraints and requirements effectively.

Definition 1 (Transaction State Machine): A *Transaction State Machine* (TSM) is a state based representation (FSM) of a database transaction, represented by a tuple $TSM = \{TS, TE\}$, where,

TS is a set of states, each state representing the user executing the transaction and the objects being acted upon. The

State Variable	Detail
0000	U1 not active in role R1, U2 not active in Role R1, P1 not granted to R1, P2 not granted to R1
0001	U1 not active, U2 not active, P1 not granted, P2 not granted
0010	U1 not active, U2 not active, P1 granted, P2 not granted
0011	U1 not active, U2 not active, P1 granted, P2 granted
0100	U1 not active, U2 active, P1 not granted, P2 not granted
0101	U1 not active, U2 active, P1 not granted, P2 granted
0110	U1 not active, U2 active, P1 granted, P2 not granted
0111	U1 not active, U2 active, P1 granted, P2 granted
1000	U1 active, U2 not active, P1 not granted, P2 not granted
1001	U1 active, U2 not active, P1 not granted, P2 granted
1010	U1 active, U2 not active, P1 granted, P2 not granted
1011	U1 active, U2 not active, P1 granted, P2 granted
1100	U1 active, U2 active, P1 not granted, P2 not granted
1101	U1 active, U2 active, P1 not granted, P2 granted
1110	U1 active, U2 active, P1 granted, P2 not granted
1111	U1 active, U2 active, P1 granted, P2 granted

(a)



(b)

Figure 1. (a) States of the Access Control Policy K1. (b) FSM of Access Control Policy K1

state variable is a binary number in which the number of bits NTS=Total number of authenticated users + Total number of database objects in the system. The initial state in TSM represents the user executing the transaction. All object bits in the initial state are set to don't care (x). A database object bit is set to 1 if the transaction has requested a lock on the object. On the other hand, it is set to (x) if the transaction is not holding a lock on it. TE is the set of directed edges which connect two consecutive states.

Next, we present a procedure for forming the TSM of a database transaction.

MakeTSM(Transaction T)

```
1 Start with initial state of size NTS, with all bits set to don't care
2 Set the corresponding bit for the user executing the transaction
3 Do for each line in the transaction
4 If (transaction is locking an object)
5 Set the corresponding bit for permission required to access an object in the state variable
6 Else
7 Reset the corresponding bit for permission required to access an object in the state variable
8 Create a state for the state variable
9 Add a transition from current state to next state
10 Repeat till end of transaction
11 Output the TSM
```

Definition 2 (Transaction State Vector): A *Transaction State Vector* (TSV) is the state from the set TSM which has the maximum number of object permissions assigned. In other words TSV is the state from amongst all TSM states with most number of 1s. It represents the point in the transaction which corresponds to the transaction accessing the maximum number of objects.

Definition 3 (Feasible State and transition): A TSV is *feasible* for an access control policy, if we have states in the FSM of the access control policy which can be mapped to the values of the TSV. Collection of such states is called a *feasible region*. A transition connecting two feasible states is called a *feasible transition*.

Definition 4 (Entry State): The entry state of the TSM is the first state at which the transaction starts executing. It represents the state in the FSM of an access control policy in which a user has sufficient privileges to invoke a transaction.

In order to illustrate extraction of specific properties from a database transaction we consider a simple transaction T1, given below. T1 conforms to the Two Phase Locking Protocol [8].

Transaction T1

$Lock(O1)$
$Lock(O2)$
$Unlock(O1)$
$Unlock(O2)$

T1 acts on database objects $O1$ and $O2$. The transaction is being executed by user $U1$. In order for $U1$ to have permission to act on $O1$, it needs to activate role $R1$ and permission $P1$ has to be granted to role $R1$. Next, we create the TSM of transaction T1 using the procedure given in Definition 1. The resulting TSM is depicted in Figure 2. The (x) in the state variable symbolize don't care condition. The ovals in Figure 2 are the corresponding lines of code that have resulted in a certain state. As stated in Definition 2, the Transaction State Vector for T1 is $(1x11)$. $(1xxx)$ is the entry state of this TSM.

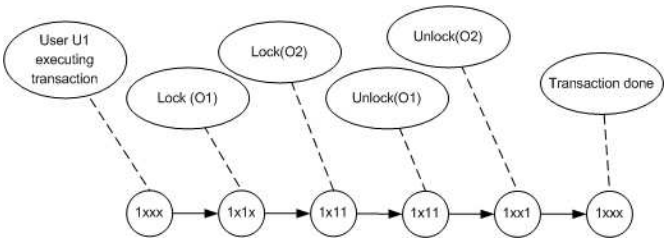


Figure 2. Transaction State Machine (TSM) of T1

3. Verification Methodology

The correct execution of a database transaction is possible only because of the underlying safeguards provided by the locking mechanism [16]. As discussed earlier, locking is an integral part of transaction execution providing conflict serializability and deadlock avoidance [15]. A locking policy does not take into account the system security and privileges granted to users, which is the domain of the access control policy. However, the two policies interact with each other, complimenting or making each others actions and definitions redundant. Hence we ask the following questions.

- *Given an access control policy, can we run an arbitrary database transaction?*
- *What are states of the security system in which a particular database transaction can execute?*
- *How do multiple database transactions interact with each other under the influence of a single access control policy?*
- *Given a specifically designed access control policy, is database locking policy required?*

In order to answer the above questions, we state the following properties and in the ensuing discussion, verify them, given the FSM of the access control policy and the TSM of the transaction.

- *Property 1: A database transaction can execute only in the feasible region of an access control policy.*
- *Property 2: A database transaction can execute without any locking safeguards, in access control policy states which lie in its feasible region and which are not feasible for entry states of any other transaction.*
- *Property 3: A database transaction can be executed in the absence of any locking mechanism, if the access control policy has locking safeguards built into it.*

3.1. Verification of properties for a Database Transaction

Property 1: A database transaction can execute only in the feasible region of an access control policy.

In order to arrive at a set of states in which T1 can be executed, we search the state space of the access control policy for feasible states of the TSV. This search can be done using any of the known graph discovery algorithms such as Depth First Search or Breath First Search. For the FSM given in Figure 1, we conclude that states (1011) and (1111) are the only states in which this transaction can be performed or the TSV is feasible. The transition, $U2$ (User $U2$ activating role $R1$) is the only transition allowed or feasible. The two permissible states, together with the one permissible transition are outlined in dark oval in Figure 3. Any other transition which may lead the system to leave the feasible state, is prohibited. These transitions could be the de-activation of users from a role or revocation of permissions held by a role. The prohibited transitions in our current example are depicted in short arrows leaving the feasible states of Figure 3.

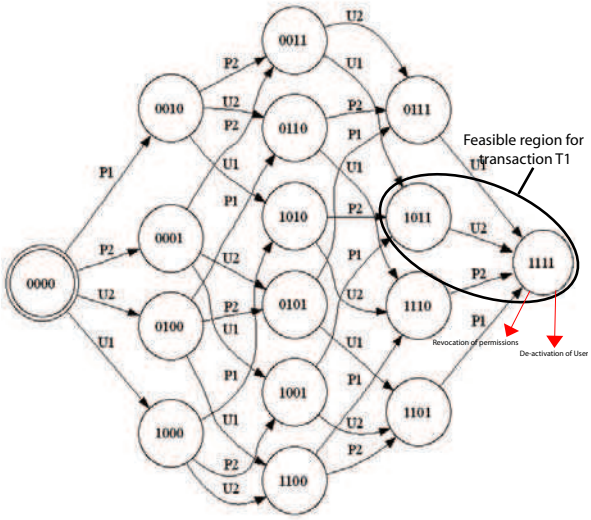


Figure 3. Feasible TSV for the Access control Policy

Given the FSM in Figure 3, we observe the following. There are specific security states in which execution of T1 is possible, and there are states in which the same transaction cannot be executed. In order to execute T1, when the security system is not in the feasible region of a transaction, we find a path from the current state to the feasible region. If a path exists, then we can execute the transaction, and if not, then the transaction cannot be executed. When

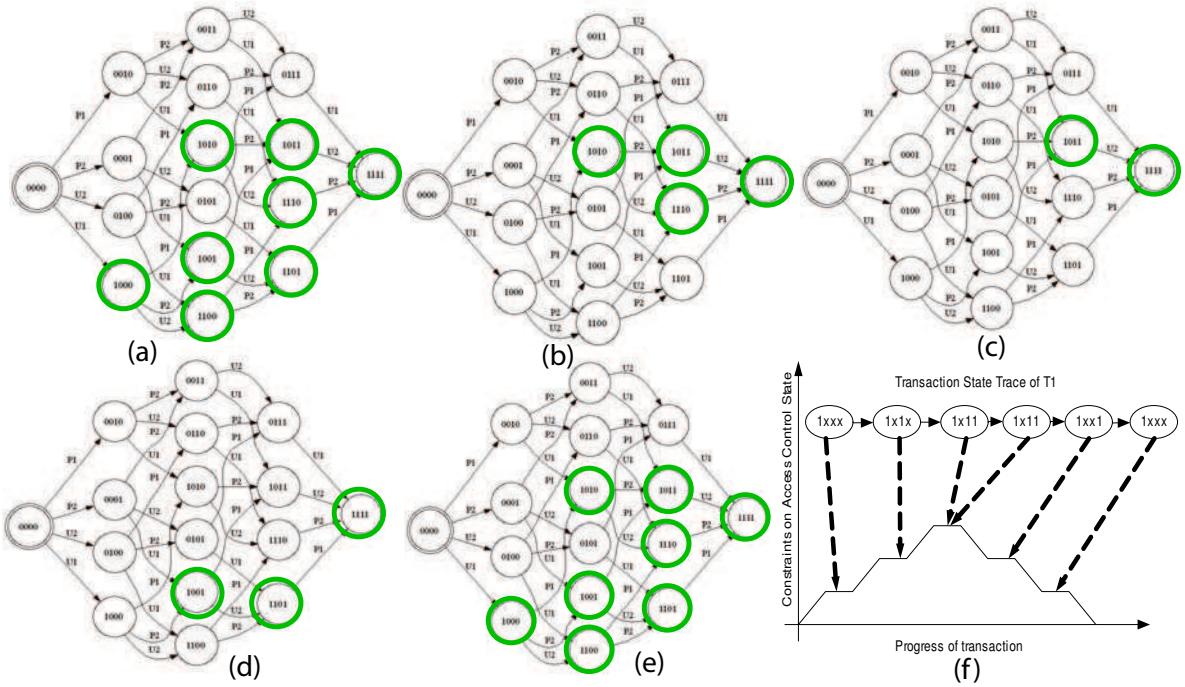


Figure 4. (a), (b), (c), (d), (e) Feasible states for TSM in access control policy. (f) Progress of transaction T1

the transaction is executing, the only permissible transition in the security policy is the feasible transition and all other paths in the access control FSM are prohibited.

It may be noted here that the two states (1111 and 1011) feasible for the transaction are the access control policy states which have maximum constraints, in terms of permissions on objects, defined on them. The TSM of T1 shows that during the course of the transaction, the requirements of access and permissions vary. This variation of security requirements can be seen by finding feasible access control states for each TSM state. Figure 4a, b, c d, and e, depict each of the feasible states for corresponding TSM.

The mapping of state $(1xxx)$ is shown in Figure 4(a), mapping of $(1x1x)$ is shown in Figure 4(b) and so on. It is observed from Figure 4 that the feasible states become more and more constrained (have more object permissions defined), with each step of the transaction. It may be noted here that each step in the transaction changes the feasible states of the access control policy. The behavior of the interaction between the access control policy and the transaction is shown in Figure 4(f). On the x-axis, we have the progress of the transaction and on the y-axis, we have the acceleration of security constraints. State $(1x11)$ is most constrained state and states $(1xxx)$ are the least constrained.

Property 2: A database transaction can execute without any locking safeguards, in access control policy states which lie in its feasible region and which are not feasible for entry states of any other transaction.

In order to verify this property, we expand our previous example and consider an access control policy K2 with two users $U1$ and $U2$, one role $R1$ and four objects $O1$, $O2$, $O3$ and $O4$. The permissions associated with these objects are $P1$, $P2$, $P3$ and $P4$. The FSM of this access control policy is depicted in Figure 10.

$U = \{U1, U2\}$
$R = \{R1\}$
$P = \{P1, P2, P3, P4\}$
$PR = \{(P1, R1), (P2, R1), (P3, R1), (P4, R1)\}$
$URA = \{(U1, R1), (U2, R2)\}$

We now define two transactions T2 and T3 (depicted in Figure 5(a) and 5(b), respectively) which will be executed under the access control policy K2, discussed above.

Next, we form the TSM for both T2 and T3. User $U1$ is executing T2 and user $U2$ is executing T3. The two TSMs are depicted in Figure 6(a) and 6(b).

The feasible TSVs in this example are $(1x111x)$ for T2 and $(x11x11)$ for T3. These TVSs are mapped onto the relevant parts of the access control policy in Figure 7.

We observe the following from Figure 6 and 7. The entry state of TSM for T2 $(1xxxxx)$ can be mapped to state (111011) in the feasible region of T3. Hence T2 can be started in the feasible state of T3, although the two TSVs

Transaction T2:	Transaction T3:
Lock *O1*	Lock *O4*
Lock *O2*	Lock *O3*
Lock *O3*	Lock *O1*
Unlock *O1*	Unlock *O4*
Unlock *O3*	Unlock *O3*
Unlock *O2*	Unlock *O1*

Figure 5. (a) Transaction T2. (b) Transaction T3

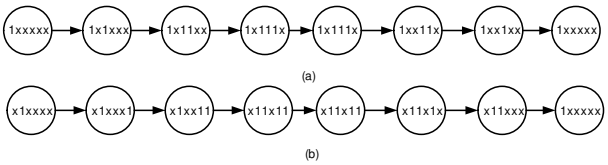


Figure 6. (a) TSM of T2. (b) TSM of T3

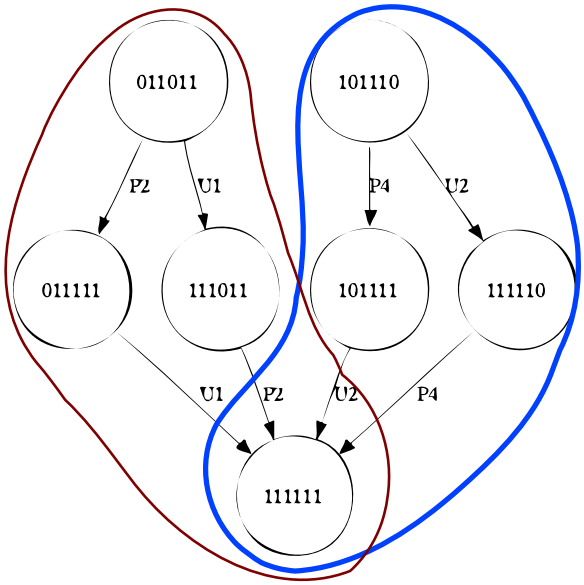


Figure 7. Feasible states for T2 and T3

do not overlap. Figure 8(a) and (b) show the feasible entry states of T2 and T3, respectively. The feasible entry states are the double circle states in each case. In case of T2, states (101110) and (101111), are the states in which T3 cannot be executed at all, similarly, in case of T3, T2 can not be executed in states (011011) and (011111). We therefore can execute T2 in states (101110) and (101111) without any database locking safeguards. Similarly, we can execute T3 in states (011011) and (011111) in the absence of any database locking safeguards.

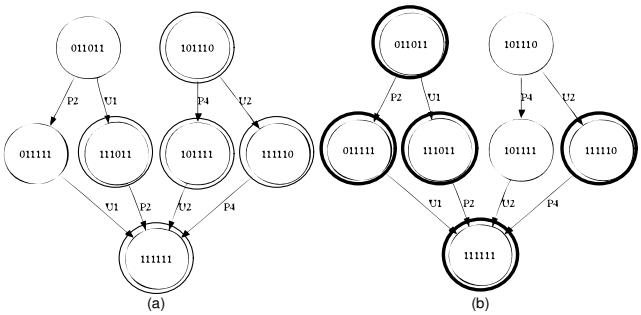


Figure 8. (a) Feasible entry states for T2. (b) Feasible entry states for T3.

Property 3: A database transaction can be executed in the absence of any locking mechanism, if the access control policy has locking safeguards built into it.

In order to closely look at this property, we assume that the database objects have a hierarchal relationship [6] as depicted in Figure 9(a). The locking policy being followed here is the rooted tree locking policy [14]. Object $O1$ is the father of object $O2$, which in turn is the father of objects $O3$ and $O4$. We also define transaction T4 in Figure 9(b). Steps 1 and 2 in T4 may correspond to database actions like add, delete etc and are being performed on object $O2$. Similarly steps 3 and 4 are also database actions like add or delete but may be performed on objects $O2$ or $O4$. Steps 5 and 6 are being performed on object $O2$.

We now present procedure *ModifiedMakeTSM* below. In this procedure we include semantics for capturing the locking constraints of the database transaction exhibiting the rooted tree locking structure.

ModifiedMakeTSM(Transaction)

```
  1 Start with initial state (00000)
(number of bits=Total number of users
in the policy + total number of objects
in the system)
  2 Set the corresponding bit for the
user executing the transaction
  3 Do for each line in the transaction
  4 If (transaction is locking an
object)
```

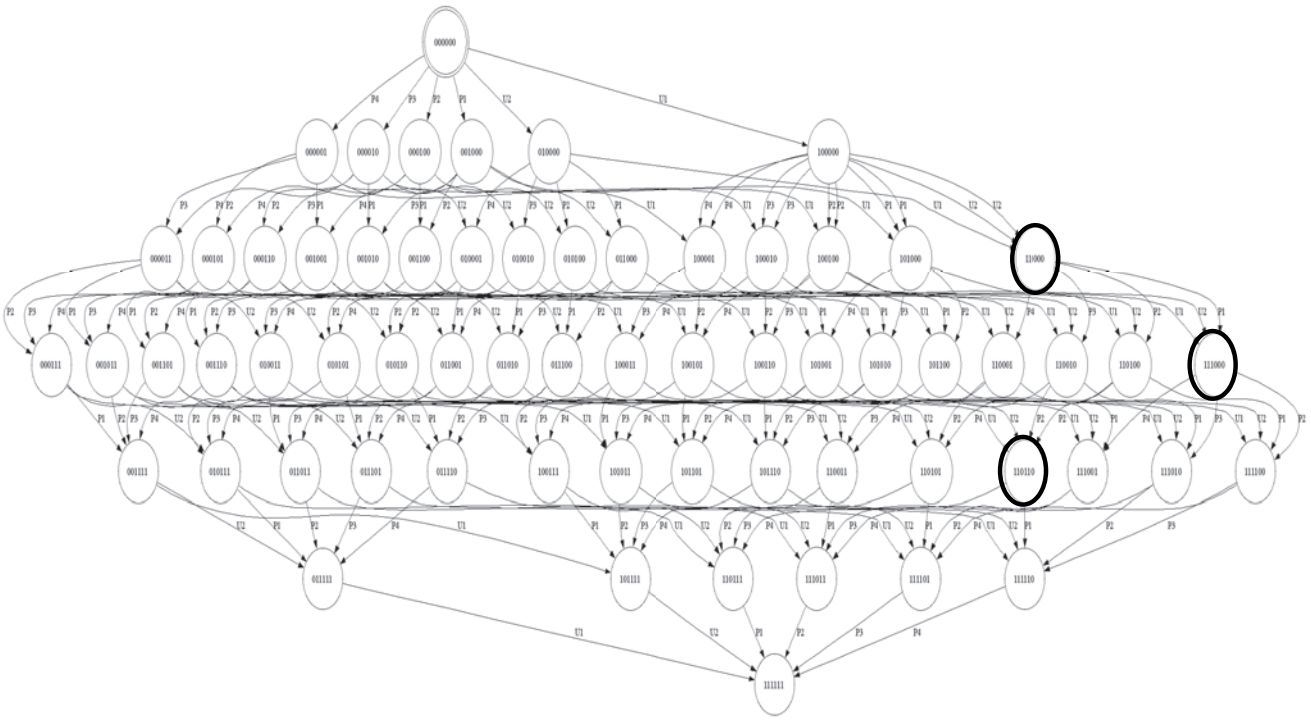
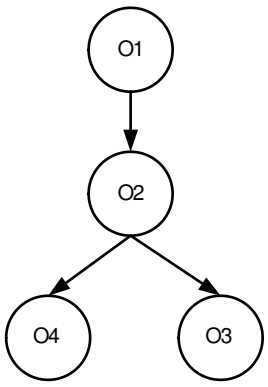


Figure 10. FSM for policy K3 and feasible states for T4



(a)

Transaction T4:
Lock *O2*
Step 1
Step 2
Lock *O4*
Step 3
Step 4
Unlock *O4*
Step 5
Step 6
Unlock *O2*

(b)

Figure 9. (a) Hierarchal relationship between objects. (b) Database transaction T4

```
  5 If (object was not locked before)
  6 Set the corresponding bit for
permission required to access an object
in the state variable
  7 If (first lock of the transaction,
set all object bits on the left of the
current bit to 0)
   8 End if
  9 Else
  10 Reset the corresponding bit for
permission required to access an object
in the state variable
  11 Create a state for the state
variable
  12 Add a transition from current
state to next state
  13 Repeat till end of transaction
  14 Output the TSM
```

Next, we create the TSM for transaction T4 using the *ModifiedMakeTSM* procedure. The resulting TSM is given if Figure 11.

In order to run transaction T4, we consider RBAC access control policy K3 given below.

$U = \{U1\}$
$R = \{R1\}$
$P = \{P1, P2, P3, P4\}$

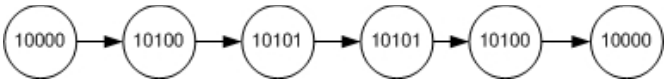


Figure 11. TSM for transaction T4

$PR = \{(P1, R1), (P2, R1), (P3, R1), (P4, R1)\}$
$URA = \{(U1, R1)\}$

The FSM of K3 is given in Figure 10. We map the TSM of T4 onto the FSM and find the feasible states, which are shown as thick circles in the policy FSM.

In order to take a closer look at the feasible region of T4, we expand the affected states and transitions of the corresponding FSM, depicted in Figure 12. In this FSM, the dashed lines show the de-assignment of user $U1$ from the role $R1$ and the revocation of permissions of the various objects. The feasible region for transaction T4 is shown as thick circles. We observe that the system has to be in state (10000) in order to start the transaction T4. In order to execute the first line of the transaction the system has to move to state (10100). The transition $P2$ is the feasible transition and is permitted. The transition $P4$, $P3$, $P1$ and $U1'$ are not permitted at this stage. Once the system is in state (10100), transitions $P3$, $P1$, $P4$ and $U1'$ are not permitted. Likewise, in order to execute each line of the transaction, the system has to move on to the next state which is indicated by the TSM of the transaction. In each feasible state there are transitions which are not permitted and there are transitions which are feasible.

We can see from the above discussion that the rooted tree locking policy is being implemented by the execution state of the security policy. Further, if the RBAC security policy is implemented as per procedure outlined above, the underlying locking policy is not required.

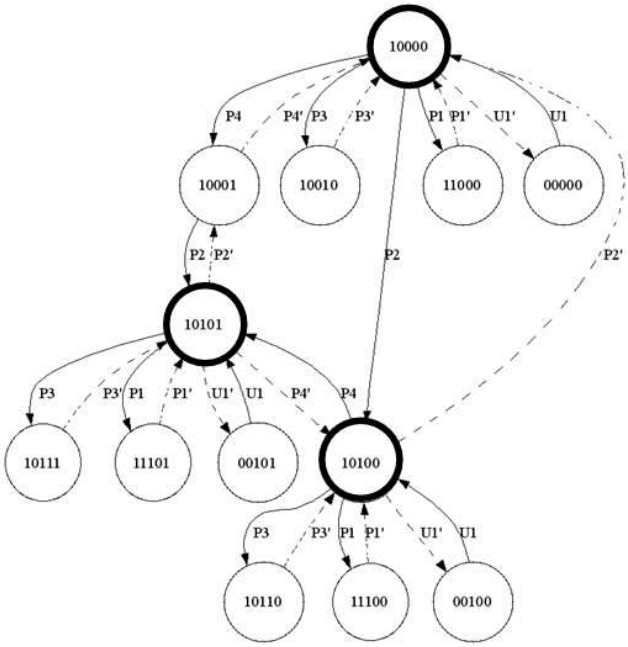


Figure 12. FSM of policy K3 with feasible states of T4

4. Discussion

As illustrated in Section 3, given a database transaction, and an access control policy; we can verify if the transaction can be executed in the context of the access control policy constraints. Property 1 can be verified for transactions exhibiting all possible types of locking policy implementations. The verification process simply points to security states in which the transaction in question can be executed. Further, it also provides a set of steps (transitions) which can be followed in case transaction cannot be executed in the current state, and feasible states are reachable from the current state. The verification process of property 1 also identifies transitions which are prohibited when the transaction is in progress. Property 2 can isolate security states which are feasible for executing a set of transactions ensuring non-interference, and hence donot rely on the underlying locking mechanism. Property 3, adds another dimension to the verification process and takes into account a hierarchal structure of database objects. The TSM of the transaction resulting from running the *ModifiedMakeTSM* procedure also captures the tree locking policy constraints. The resulting TSM isolates feasible states of the security system and provide a set of states and transitions which can be followed so that the transaction can execute without relying on the underlying access control policy. It may be noted here that the procedure *ModifiedMakeTSM* will capture the locking constraints only for the rooted tree locking policy. Also, the non-reliance on locking policy is only possible if the security states can very precisely follow the set of feasible states. If the order of transitions and states is not followed, the underlying locking policy is the only safeguard for conflict serializabilty and deadlock freedom. It is also assumed throughout the verification process that one user will only execute a single transaction at a time.

Verification of the proposed properties can be achieved by applying the various model checking approaches and tools reported in the literature. Notable among them is SPIN [7] which converts the finite state model of the input system and negation of the property in question to Buchi automata. SPIN also provides a counter-example if the property is not valid.

5. Related Work

Model checking helps in automating analysis of complex systems and properties [3]. A technique for the extraction of abstract finite-state machines directly from finite-state programs and satisfaction of a formula by the abstract machine is given in [4]. This methodology is the basis of subsequent model checking approaches. Access control policy analysis and verification has been an active area of research recently. Ahmed and Tripathi [1], have proposed static verification of security requirements for CSCW systems using finite-state techniques. Schaad, Lotz and Sohr [12] have proposed a model-checking approach for automated analysis of delegation and revocation functionalities in the context of a work flow requiring static and dynamic separation of duty constraints. Zhang, Ryan and Guelev [17], have proposed a model-checking algorithm which can be used to evaluate access control policies. Verification of database transactions has been done by a number of authors, notable among them are Sheard and Stemple [13]. They report on a system that performs compile time verification of integrity constraints in a database transaction.

While these approaches provide a valuable insight in verification of access control policies on one hand and verification of integrity constraints associated with database transactions on the other, none has addressed the verification of database transaction against an access control policy. Our approach is unique in the sense that it provides an insight into the interaction between the access control policy and the database locking policy, manifested as database transaction.

6. Conclusions

In this paper, we have presented a methodology for the verification of database transaction requirements executed in the presence of an RBAC security policy. Our contribution is the characterization of three properties, and their illustrative verification steps, which effectively verify database transactions. Property 1 and 2 identify feasible security states and transitions on the one hand and prohibited security states and transitions on the other hand. Property 3, pinpoints security states which allow a database transaction to execute in the absence of the underlying locking policy. The list of properties which effectively verify a database transaction rigorously is in no way complete and remains the subject of our future work. Developing an effective translation mechanism which can translate requirements of database transactions, including locking, is also an area in which we plan to work in the future.

References

[1] T. Ahmed and A. R. Tripathi. Static verification of security requirements in role based cscw systems. In *SACMAT*, pages 196–203, 2003.

[2] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Trans. Software Eng.*, 5(3):203–216, 1979.

[3] E. M. Clarke. Model checking. *Lecture notes in Computer Science*, 1346:54–56, 1997.

[4] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[5] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[6] N. Goodman and O. Shmueli. Syntactic characterization of tree database schemas. *J. ACM*, 30(4):767–786, 1983.

[7] G. J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[8] Z. M. Kedem and A. Silberschatz. Locking protocols: From exclusive to shared locks. *J. ACM*, 30(4):787–804, 1983.

[9] W. H. Kohler. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Comput. Surv.*, 13(2):149–183, 1981.

[10] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[11] R. S. Sandhu, D. F. Ferraiolo, and D. R. Kuhn. The nist model for role-based access control: towards a unified standard. In *ACM Workshop on Role-Based Access Control*, pages 47–63, 2000.

[12] A. Schaad, V. Lotz, and K. Sohr. A model-checking approach to analysing organisational controls in a loan origination process. In *SACMAT*, pages 139–149, 2006.

[13] T. Sheard and D. Stemple. Automatic verification of database transaction safety. *ACM Trans. Database Syst.*, 14(3):322–368, 1989.

[14] A. Silberschatz and Z. M. Kedem. A family of locking protocols for database systems that are modeled by directed graphs. *IEEE Trans. Software Eng.*, 8(6):558–562, 1982.

[15] M. Yannakakis. Serializability by locking. *J. ACM*, 31(2):227–244, 1984.

[16] M. Yannakakis, C. H. Papadimitriou, and H. T. Kung. Locking policies: Safety and freedom from deadlock. In *FOCS*, pages 286–297, 1979.

[17] N. Zhang, M. Ryan, and D. P. Guelev. Evaluating access control policies through model checking. In *ISC*, pages 446–460, 2005.