Towards Formal Security Analysis of GTRBAC using Timed Automata

Samrat Mondal
School of IT
IIT Kharagpur, India
samratm@sit.iitkgp.ernet.in

Shamik Sural
School of IT
IIT Kharagpur, India
shamik@cse.iitkgp.ernet.in

Vijayalakshmi Atluri
MSIS Department and CIMIC
Rutgers University, USA
atluri@rutgers.edu

ABSTRACT

An access control system is often viewed as a state transition system. Given a set of access control policies, a general safety requirement in such a system is to determine whether a desirable property is satisfied in all the reachable states. Such an analysis calls for formal verification. While formal analysis on traditional RBAC has been done to some extent, the extensions of RBAC lack such an analysis. In this paper, we propose a formal technique to perform security analysis on the Generalized Temporal RBAC (GTRBAC) model which can be used to express a wide range of temporal constraints on different RBAC components like role, user and permission. In the proposed approach, at first the GTRBAC system is mapped to a state transition system built using timed automata. Characteristics of each role, user and permission are captured with the help of timed automata. A single global clock is used to express the various temporal constraints supported in a GTRBAC model. Next, a set of safety and liveness properties is specified using computation tree logic (CTL). Model checking based formal verification is then done to verify the properties against the model to determine if the system is secure with respect to a given set of access control policies. Both time and space analysis has been done for studying the performance of the approach under different configurations.

Categories and Subject Descriptors

H.2.0 [**Information Systems**]: Security, integrity, and protection; D.4.6 [**Security and Protection**]: Access Controls

General Terms

Security, Verification, Algorithms, Performance

Keywords

GTRBAC, Timed Automata, Security Analysis, Model Checking, CTL

SACMAT'09, June 3–5, 2009, Stresa, Italy.

1. INTRODUCTION

In a multiuser environment, access control is required for controlled sharing and protection of resources. Over the past few decades, many access control models have been proposed to specify the different access control policies. These primarily include Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Role Based Access Control (RBAC). Among these, RBAC has gained considerable attention due to its flexibility, ease of administration and intuitiveness. RBAC can be viewed as a state transition system where state changes occur via administrative operations. Roles, users and permissions are the three primary components of RBAC.

For some applications, it is often required that access to a particular resource is given on the basis of the current time or location of the subject making the request. Due to such growing requirements, the RBAC model has been extended in temporal, spatial and spatio-temporal domains. These extensions offer flexibility in specifying a variety of security policies. Bertino et al. [1] introduced the idea of TRBAC, which is an extension of RBAC in the temporal domain. Using this model, a user may activate a role only during certain time intervals. Atluri and Gal proposed a Temporal Data Authorization Model (TDAM). It is capable of expressing access control policies based on the temporal characteristics of data, such as valid and transaction time [2]. Recently, two other models have been proposed for spatio-temporal access control [3][4]. Both time and location of access request are considered for allowing access in these models. Despite the availability of a number of access control models, not many formal security analysis techniques have been developed.

The fundamental question of safety is to answer the following: "Given the current authorization state and the policy specification, will a user (subject) ever gain access to a specific resource (object)?" Thus, the basic aim of security analysis is to check whether the system maintains desirable security properties in all the states. The security properties are formulated from a set of access control policies. A formal verification technique then can be used to ensure the consistency of an access control specification. Security analysis provides answers to the queries whether an undesirable state is reachable or all the reachable states satisfy the desirable properties. In the context of an RBAC model, a typical safety query would be "Whether a role remains enabled forever" or "Whenever a role is enabled, is another role also enabled"? These queries can be classified primarily into two types, namely safety and liveness. With safety property, we try to ensure that "something bad should never

happen". With liveness property we try to capture that "finally something good should happen". When the analysis techniques consider only safety properties, then we call it as safety analysis. Security analysis generalizes the notion of safety analysis in the sense that it considers not only safety properties but also other types of properties.

Recently, researchers have initiated some work on formal analysis of temporal and spatio-temporal extensions of access control models. Our goal is to develop an approach to answer the safety and liveness queries for temporal extensions of RBAC. In TRBAC, temporal constraints on role activation and temporal dependencies among roles are only considered [1]. As a result, only a limited number of temporal policies can be specified using this model. Joshi et al. later proposed a more general model known as the Generalized Temporal RBAC (GTRBAC) [5]. In this model it is possible to specify a wider range of constraints, which include temporal constraints on role enabling, role activation, role hierarchy, Separation of Duty (SoD) and role trigger. While GTRBAC is capable of expressing many temporal policies, it makes its formal analysis much more challenging. In this paper, we propose a method for formal analysis of the GTRBAC model. As a first step, a given GTRBAC model is mapped to a state transition system using timed automata. Next a set of security properties is specified in computation tree logic (CTL), which are finally verified against the model.

The rest of the paper is organized as follows. Section 2 summarizes prior work done in this field. Section 3 describes the method of mapping from a GTRBAC model to a timed automata based model. Section 4 deals with the correctness of the proposed mapping mechanism. A set of security properties in the context of GTRBAC is specified in Section 5. Verification results are discussed in Section 6. Section 7 concludes this paper and provides future direction for work.

2. RELATED WORK

Harrison et al. [6] first formalized the problem of safety analysis in the context of access control matrix model (commonly known as the HRU model). They showed that in general, safety analysis problem for access control matrix model is undecidable. Jones et al. proposed Take-Grant Protection Model (TGPM) [7] which represents a system as a directed graph where vertices are either subjects or objects. They proved that it is possible to decide on the safety of a system even when the number of subjects and objects is very large, or unbounded. Thus, in a general scheme like HRU, safety analysis is undecidable but in a scheme enforced with reasonable restrictions such as TGPM, safety analysis becomes decidable. So a balance is required between the ability to perform an efficient safety analysis and the generalization ability of a model. For this purpose, Sandhu introduced the idea of Schematic Protection Model (SPM) [8]. It provides a "high-level" structure compared to the "low-level" structure of the access matrix. This makes policy specification more convenient in SPM.

Later, Koch et al. [9] showed that with reasonable restrictions on the rules, the safety analysis problem becomes decidable. They used graph transformations as a general formalism to specify access control policies based on roles. Ahmed and Tripathi [10] proposed a model checking mechanism for verification of security requirements in role based Computer Supported Cooperative Work (CSCW) systems. Li and Tripunitara [11] were the first to analyze RBAC. They performed security analysis on two restricted versions of administrative RBAC. These are known as AATU (Assignment And Trusted Users) and AAR (Assignment And Revocation). They proposed two reduction algorithms and studied complexity results for various analysis problems such as safety, availability and containment. Stoller et al. [12] considered negative preconditions and static mutually exclusive role constraints in the policy analysis of administrative RBAC. Zhang and Joshi [13] addressed constraints like hybrid hierarchy and dynamic separation of duty in handling the user authorization process. In [14], Jha et al. made a comparison between the use of model checking and first order logic programming for the security analysis of RBAC. It was concluded that model checking is a promising approach in this context.

Formal analysis of GTRBAC needs to be done differently due to the presence of temporal constraints in the system. Joshi et al. [15] presented an analysis of the expressiveness of the constructs provided by GTRBAC model and showed that its constraints-set is not minimal. Bertino et al. [1] provided a polynomial time algorithm to verify whether specifications in TRBAC are free from ambiguities. In [16], Shafiq et al. studied a Petri-Net based framework for verifying the correctness of event-driven RBAC policies in real time systems. It considers only the cardinality and separation of duty constraints. But it does not handle temporal RBAC policies. A few other systems where time is considered to be a critical factor have also been analyzed. Alur et al. [17] have used model checking for the analysis of real time systems. Furfaro and Nigro [18] have used timed automata for temporal verification of real time state machines.

Recently, Mondal and Sural have suggested two approaches for the security analysis of RBAC with limited temporal constraints using timed automata [19][20]. Though the first one [19] performs better, it addresses a limited number of constraints. Role trigger is not considered in the second approach [20]. Both the approaches assume that a user can activate only a single role at a particular instance of time. Security analysis performance degrades with growing number of users, roles and permissions. In this paper, our analysis scheme addresses all the temporal constraints of GTRBAC including triggers and it also allows users to activate multiple roles at the same time as well as at different points in time.

3. FROM GTRBAC TO TIMED AUTOMATA BASED MODEL

The first step of security analysis is to design a formal model using timed automata from a given GTRBAC model. Each component, namely role, user and permission, is represented using a timed automaton. Different temporal constraints on role enabling, disabling, role activation, deactivation, permission assignment, user assignment, role hierarchy and SoD are considered during this mapping process. Before going into the details of the mapping mechanism, the GTRBAC model is introduced first followed by a brief overview of timed automaton.

3.1 GTRBAC - Preliminaries

Joshi et al. introduced the GTRBAC model [5] to specify a comprehensive set of temporal constraints. It is an exten-

sion of TRBAC model proposed in [1]. Several important aspects that can be handled in GTRBAC are given below-

- GTRBAC addresses temporal constraints on user-role assignments, role-permission assignments, role hierarchy, SoD, role trigger. This provides a large number of options in specifying a wide range of temporal access control policies.
- This model distinguishes between role enabling and role activation. A role is in the *enabled* state if it is ready for user assignment and the role is in the *disabled* state if the users cannot activate the role. A role is said to be *active* if there is at least one user who has assumed that role. Once a role is in active state, reactivation of the role does not change the state of the role. When all the users deactivate the role then role moves to the *enabled* state.
- The model can handle periodic as well as duration constraints. The periodicity constraints are used to specify the exact intervals during which a role can be enabled or disabled, and during which a user-role assignment or a role-permission assignment is valid. Duration constraints are used to specify a duration for which enabling or assignment of a role is valid. Depending upon the organizational requirements, these constraints can be applied on various components of RBAC.

3.2 Brief Overview of Timed Automaton

Alur and Dill [21] introduced the idea of timed automaton. It is a finite state machine equipped with a set of clocks. All the clocks are synchronized, i.e., they advance at the same pace. The clocks can take any non-negative real number in $\mathbb{R}$. If C is a finite set of clock variables, then a clock valuation v over C is a function $v : C \rightarrow \mathbb{R}$ which associates with every clock c, its value $v(c) \in \mathbb{R}$. The clock valuations over C are represented by $\mathbb{R}^C$. If d is a delay such that $d \in \mathbb{R}$ then $v(c) + d$ represents the clock valuation associated with clock c and delay d. $B(C)$ represents a set of clock constraints over C. The clock constraints can be of the form $c \bowtie k$ where $c \in C$, $\bowtie \in \{\leq, <, ==, >, \geq\}$ and $k \in \mathbb{N}$. When a valuation v satisfies a constraint g, it is written as $v \models g$. A timed automaton is a 6-tuple (L, l_0, C, A, E, I) where:

- L is a finite set of control states, also called locations.
- $l_0 \in L$ is the initial location.
- C is a finite set of clocks.
- A is a set of actions.
- $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a finite set of transitions. An element $e \in E$ can be expressed as $e = (l_1, a, g, r, l_2)$ which represents a transition from l_1 to l_2, g is a guard which is a conjunction of boolean expressions involving clocks or some other variables, r is the set of clocks that is reset by e, and a is the action of e. The transition can also be represented by $l_1 \xrightarrow{g,a,r} l_2$.
- $I : L \rightarrow B(C)$ assigns invariants to locations.

Several timed automata can interact with each other by the use of channel synchronization. The intuition is that two automata can synchronize on enabled edges annotated with complementary synchronization labels, i.e., two edges in different automata can synchronize if the guards of both edges are satisfied, and they have the synchronization labels *action*? and *action*! respectively.

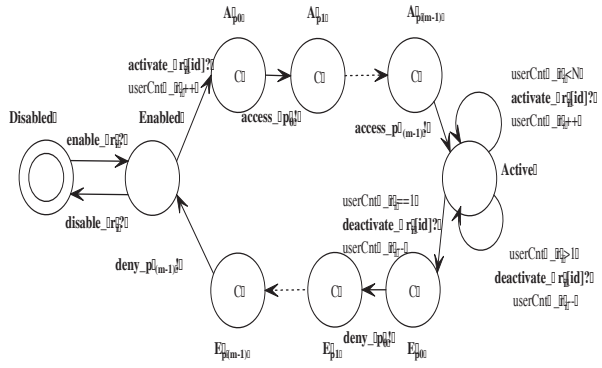

Figure 1: Role timed automaton

3.3 Construction of Timed Automata for GTRBAC Components

Role, user and permission - these are the three primary components of an RBAC scheme. The behavior of these components is represented using TA. Like a directed graph, a timed automaton has many nodes, known as control states, which are connected via directed edges labeled by temporal constraints and actions. The target here is to identify the different states of each GTRBAC component and represent them by one or more control states. As mentioned before, in GTRBAC, a role can be in *disabled*, *enabled* or *active* state. For representing enabling and disabling behaviors, two control states - labeled as "Disabled" and "Enabled" - are created in the automaton where "Disabled" is considered to be the initial state. Transitions from "Disabled" to "Enabled" and "Enabled" to "Disabled" are annotated using guards or some synchronization actions. Thus, with these two locations, the role enabling and disabling behavior can be represented as shown in Figure 1.

Next, the *active* state of a role is considered. An *enabled* role becomes *active* when a user acquires the permissions associated with it. So, with the first user assignment, an *enabled* role goes to the *active* state. A user assignment is performed by an activation operation invoked by the user. Any subsequent activation operation does not change the state of the role. When all the users are unassigned from the role then the role goes back to the *enabled* state. From *enabled* or *active* state, a role can go back to the *disabled* state. In case of role hierarchy, the users of a senior role can directly activate a junior role provided the junior role is already in *enabled* state. To capture this behavior in the timed automaton, a control location labeled as "Active" is added. Now the transition from "Enabled" to "Active" is labeled by a role activation operation initiated by a user. This action is represented as $activate\_r_i[id]$ where r_i is the role under consideration and id is an integer variable representing the user who is trying to activate the role. To keep track of the number of users assigned to a role, a counter $userCnt\_r_i$ is used. The value of the counter is adjusted with each acti-

vation and deactivation operation. A self-loop labeled with activation action is required at the "Active" location to reflect the fact that when a role is in "Active" state, it remains in that state with further such activation operations. Similar to role activation, two transitions – one from "Active" to "Active" and another from "Active" to "Enabled" are needed. These transitions are labeled by an action $deactivate\_r_i[id]$. A complete role timed automaton is shown in Figure 1.

Algorithm 1 Construction of Role Timed Automaton

for each $r \in R$ **do**
  $i = 0$; $j = 0$;
  for each $(< u, r' > \in UA)$ AND $(r' == r)$ **do**
    $i++$;
  end for
  $n = i$; {/* n users can activate role r*/}
  for each $(< r', p > \in PA)$ AND $(r' == r)$ **do**
    $j++$;
  end for
  $m = j$;{/* r is associated with m no. of permissions */}
  construct a timed automaton $TA_r = < L, L_0, C, A, E, I >$ such that
  L=$\{Disabled, Enabled, Active, A_{p0}, \ldots, A_{p(m-1)}, E_{p0}, \ldots, E_{p(m-1)}\}$;
  $commit(\{A_{p0}, \ldots, A_{p(m-1)}, E_{p0}, \ldots, E_{p(m-1)}\})$;{/* commit() function marks a set of locations as "Committed" */}
  $L_0 = \{Disabled\}$; $C = \emptyset$; $I = \emptyset$;
  A=$\{enable\_r, disable\_r, activate\_r[n], deactivate\_r[n], access\_p_0, \ldots, access\_p_{m-1}, deny\_p_0, \ldots, deny\_p_{m-1}\}$;
  $E = \{Disabled \xrightarrow{enable\_r?} Enabled, Enabled$
  $Enabled, Enabled \xrightarrow{activate\_r[id]?} A_{p0}, \ldots, A_{p(m-1)}$
  $A_{p0}, \ldots, A_{p(m-1)} \xrightarrow{access\_p_{m-1}!} Active, Active$
  $Active, Active \xrightarrow{activate\_r[id]?} Active, Active$
  $Active, Active \xrightarrow{deactivate\_r[id]?} Active, Active$
  $Active, Active \xrightarrow{deactivate\_r[id]?} E_{p0}, \ldots, E_{p(m-1)}$
  $E_{p0}, \ldots, E_{p(m-1)} \xrightarrow{deny\_p_{m-1}!} Enabled, Enabled \xrightarrow{disable\_r?} Disabled\}$;
end for

Temporal constraints are also applicable on the permission assignment relation. With this constraint, when a role is active, the permissions associated with it are made available on satisfying the given temporal constraints. But in most of the situations it is desirable that when the role is active then all the permissions associated with it are also available to the users of the role. With the first role activation operation, the permissions associated with the role can be accessible by the users of that role, and when all the users deactivate the role, permissions are also no longer accessible. This is done in our proposed approach by introducing a "Committed" location for each permission in the role timed automaton.

Definition 1. Committed Location: *A committed location in a timed automaton is a location which freezes time. Thus in a committed location, the value of the clock variables does not progress.*

These locations are labeled as $A_{p0}, A_{p1}, \ldots, A_{p(m-1)}$ in Figure 1. From each such location, a permission access request is sent using synchronization action $access\_p_i$ where $i = 0, \ldots, (m-1)$. Similarly, from the committed locations $E_{p0}, E_{p1}, \ldots, E_{p(m-1)}$, another synchronization action $deny\_p_i$ is used. The complete process of constructing role timed automata is given in Algorithm 1.

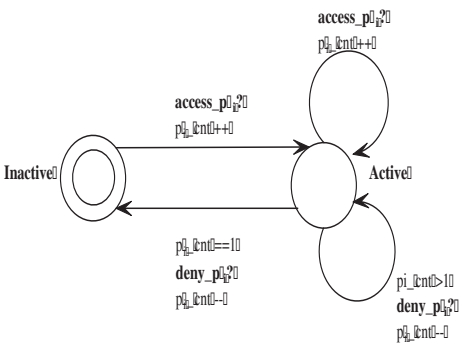

Figure 2: Permission timed automaton

A permission timed automaton as shown in Figure 2 is created using two locations - "Inactive" and "Active". Transition from "Inactive" to "Active" is labeled with the action $access\_p_i$ and transition from "Active" to "Inactive" is labeled with $deny\_p_i$. So with the first access request, the permission role timed automaton is in the "Active" state. Two self-loop transitions are also added in the "Active" location. One is labeled with $access\_p_i$ action which keeps the automaton in "Active" location as long as the access request (in the form of $access\_p_i$) is coming from the role timed automaton. Another transition is labeled with $deny\_p_i$ action. This indicates that the role remains in "Active" location unless all the deny requests (in the form of $deny\_p_i$) are served. The algorithm for the construction of permission timed automata is given in Algorithm 2.

Algorithm 2 Construction of Permission Timed Automaton

for each $p \in P$ **do**
  construct a timed automaton $TA_p = < L, L_0, C, A, E, I >$ such that
  L=$\{Inactive, Active\}$;
  $L_0 = \{Inactive\}$; $C = \emptyset$; $I = \emptyset$;
  A=$\{access\_p, deny\_p\}$;
  $E = \{Inactive \xrightarrow{access\_p?} Active, Active \xrightarrow{access\_p?} Active, Active \xrightarrow{deny\_p?} Active, Active \xrightarrow{deny\_p?} Inactive\}$;
end for

It can be observed that the basic structure of role timed automaton is the same for all roles except for the action names. This is also true for permission timed automata. But to represent various types of users behavior, the structure of the user timed automaton may not remain the same. For example, one user may be allowed to activate only a single role, whereas another user may be allowed to activate multiple roles at the same time, or multiple roles but not at the same time. So, depending upon such behavioral characteristics, four different types of user interactions are

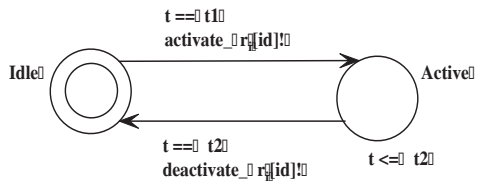

Figure 3: User timed automaton for Type-1 interaction

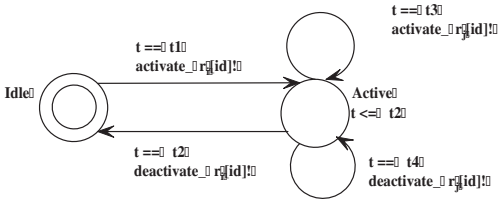

Figure 4: User timed automaton for Type-2 interaction

identified. They are termed as Type-1, Type-2, Type-3 and Type-4 interactions.

- **Type-1 interaction**: A user participating in this type of interaction can activate only a single role at a time. To represent it using a timed automaton, two locations - "Idle" and "Active" are needed. The transition from "Idle" to "Active" is labeled with a temporal constraint and an action for activating the role. Similarly, another transition from "Active" to "Idle" is needed. In Figure 3, user timed automaton for Type-1 interaction is shown. Here the user sends an activation action to $r1$ when $t = t1$ and at $t = t2$, the user sends a deactivation action. The user can remain in the "Active" location till $t \leq t2$, so an invariant $t \leq t2$ is used for "Active" location.

- **Type-2 interaction**: In this type, a user can activate multiple roles at different time instances. With one activation request, the user goes to the "Active" state and it remains in that state if it wants to activate any other role. When the user has sent all the deactivation requests, it goes back to the "Idle" state. The basic structure of user automaton for Type-2 interaction is same as that of Type-1, but in this case "Active" location has an extra pair of self-loops for each additional role activation and deactivation operation, which are labeled with some temporal constraints. This is illustrated in Figure 4.

- **Type-3 interaction**: In this type of interaction, a user can activate multiple roles at a particular instance of time. This can be done using the "Committed" location where time variable does not proceed. So to activate n roles at a single time instance, "Committed" locations $A_0,\ A_1,\ \ldots,\ A_{n-2}$ are used in the automaton as shown in Figure 5. Using the first activation operation, the automaton goes to the A_0 location and then time does not proceed further as long as they pass through the "Committed" locations. The next transitions are labeled with other activation actions. Thus when the "Active" location is reached, a total of n activation requests have been sent by the user without change of time. Similarly, the user can send multiple deactivation requests at a particular time instance using the "Committed" locations $I_0,\ I_1, \ldots,\ I_{n-2}$.

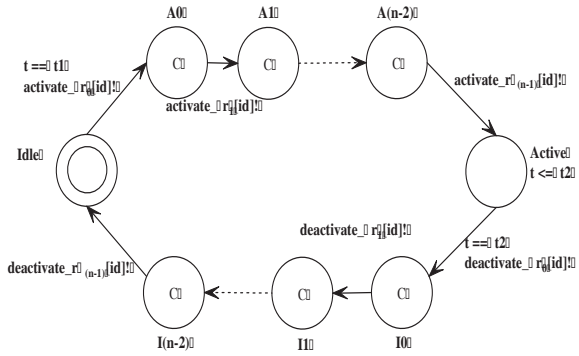

Figure 5: User timed automaton for Type-3 interaction

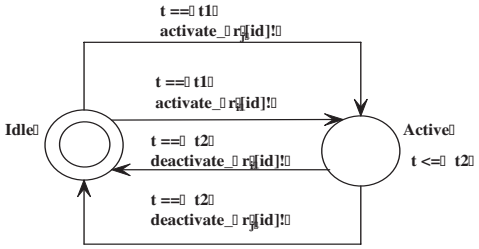

Figure 6: User timed automaton for Type-4 interaction

- **Type-4 interaction**: This type of interaction satisfies the constraint Separation of Duty (SoD). Users can activate multiple roles but not at the same time. To implement such type of interaction, parallel transitions are needed between two locations. Both the transitions are labeled with the same temporal constraints but with different actions. At a given time, either of the two transitions is selected non-deterministically. However, when one is selected the other one cannot be selected. So if there are two conflicting roles, then it will introduce two parallel transitions from "Idle" to "Active" for activation operations and two more parallel transitions from "Active" to "Idle" for deactivation operation. In Figure 6, a user timed automaton having Type-4 interaction is shown with two conflicting roles r_i and r_j.

A separate algorithm has been developed for the construction of user automaton for each type of interaction. Due to page restriction we only show user automaton construction for Type-4 interaction in Algorithm 3.

3.4 Controller Timed Automaton

In Subsection 3.3, role, permission and users are mapped to timed automata. To design a complete GTRBAC sys-

Algorithm 3 Construction of User Timed Automaton of Type-4 Interaction

1: **for** each $u \in U$ **do**
2: construct a timed automaton $TA_u =< L, L_0, C, A, E, I >$ such that {/* u can activate one of the conflicting roles at a time */}
3: i=0;
4: **for** each $(< u', r > \in UA)$ AND $(u' == u)$ **do**
5: $A = A \cup \{activate\_r[id], deactivate\_r[id]\}$; i++;
6: **end for**
7: n=i;
8: L=$\{Idle, Active\}$; $L_0 = \{Idle\}$; $C = t$;
9: $E = \{Idle \xrightarrow{g, activate\_r[id]!} Active, \ldots, Idle \xrightarrow{g', activate\_r'[id]!} Active, Active \xrightarrow{g, deactivate\_r[id]!} Idle, Active \xrightarrow{g', deactivate\_r'[id]!} Idle\}$;
10: $I = Active \rightarrow frameInv(g')$;{/* frameInv() constructs the invariant */}
11: **end for**

tem, several timed automata are required. In a timed automaton, temporal constraints are specified using clock variables which are to be used carefully since they directly affect the verification process. More the number of clock variables, higher is the complexity of the model checking process [22][23]. To express the temporal constraints, a global clock can be used by all the timed automata. Although these automata can use the global clock, they are not allowed to update the clock value. So a separate automaton is created to keep control over the global clock variable. It is responsible for enabling and disabling of roles and updating of the clock variable. This automaton is termed as the *Controller Automaton*. In this automaton, at the first transition, the clock value is initiated and at the last transition, the clock value is reset. Each transition is labeled with a temporal constraint using the global clock and a corresponding role enabling and disabling action. If two roles have to be enabled or disabled at the same time instance, then a "Committed" location can be used. In Figure 7, a Controller Timed Automaton is shown. It enables three roles among which two are enabled at the same time instance. Also they are disabled at the same time instance. The reset function is used to update the value of the clock variable. The details of construction of controller automaton is given in Algorithm 4.

4. CORRECTNESS OF THE MAPPING PROCESS

To show the correctness of the proposed mapping scheme, it is to be shown that each of the constraints of GTRBAC model is properly mapped to timed automaton. In GTRBAC, time is represented by a periodic expression [5]. A periodic time instance can be expressed as a tuple <[begin, end],P> where P is a periodic expression denoting an infinite set of time instants and [begin, end] is a time interval denoted by a lower and an upper bound that are imposed on instants in P.

The periodicity part of a time instance is represented in the timed automaton as $m * i + t_c$, where m is a constant and i is an integer variable. If *hour* is taken as the lowest

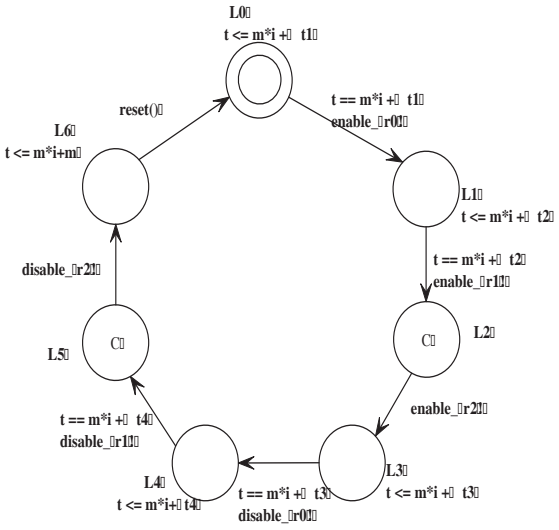

Figure 7: Controller timed automaton

Algorithm 4 Construction of Controller Timed Automaton

1: i=0;
2: **for** each $(r \in R)$ **do**
3: **if** $enable(r, t)$ **then**
4: $list[i{+}{+}] =< r, t, 0 >$;{/* 0 is used for role enabling action */}
5: **end if**
6: **if** $disable(r, t)$ **then**
7: $list[i{+}{+}] =< r, t, 1 >$;{/* 1 is used for role disabling action */}
8: **end if**
9: **end for**
10: n=i;
11: sortAsc(list);{/* sort list in ascending order on the basis of t */}
12: construct a timed automaton $TA_c =< L, L_0, C, A, E, I >$ such that
13: $L = \{L_0\}$; $A = \emptyset$; $E = \emptyset$; $L_0 = L_0$; $C = t$
14: $prev\_time = 0$;
15: **for** $i = 0$ to $(n - 1)$ **do**
16: $L = L \cup \{L_{i+1}\}$;
17: $curr\_time = getTime(list[i])$;
18: **if** $(curr\_time = prev\_time)$ **then**
19: $commit(\{L_i\})$;{/* commit() function marks a set of locations as "Committed" */}
20: **end if**
21: g = frameGuard(list[i]);{/* frameGuard() constructs the temporal expression */}
22: act = getAction(list[i]); {/* getAction() determines whether the function is $enable\_r_i$ or $disable\_r_i$ */}
23: $I = L_{i+1} \rightarrow frameInv(g)$;{/* frameInv() constructs the invariant */}
24: $A = A \cup \{act\}$; $E = E \cup \{L_i \xrightarrow{g, act!} L_{i+1}\}$;
25: $prev\_time = curr\_time$;
26: **end for**
27: $E = E \cup \{L_{i+1} \xrightarrow{reset()} L_0\}$;

time unit, then to represent each day's behavior for a week, m can be assumed to be 24 and the value of i may range from $0, 1, \ldots, 6$. Durability constraint can also be applied here. Suppose a role remains in the *enabled* state for n hours. With one transition, say $t == 24 * i + t1$, the role gets enabled and with another transition, say $t == 24*i+t2$, the role is disabled. Here $t2$ should be defined in such a way that $t2 = t1+n$. Thus, the duration is the difference between the time instances of outgoing and incoming transitions of a location. Also, if *day* is considered to be the lowest time unit, then m becomes 7 to represent weekly behavior of the role and i can range from $0, 1, \ldots, 3$ to represent the four weeks. In the controller automaton the last transition resets the clock.

- **Temporal constraints on role enabling**: A role can be enabled or disabled depending upon satisfying some temporal constraints. In our proposed mapping scheme, the controller timed automaton is used for triggering the enabling and disabling actions. In Subsection 3.4, it has already been explained how the controller automaton works.
- **Temporal constraints on role activation**: Temporal constraints on role activation determine when a user will get a role. This temporal constraint is used on the user timed automaton. The "Idle" to "Active" transition is labeled with a temporal constraint. An activation request is sent during the transition to a role it wants to activate. Thus the role gets active at some particular time instance.
- **Temporal constraints on role hierarchy**: Role hierarchical relation is implemented using user timed automaton with Type-2 interaction. Role hierarchy means the users of an active role (senior role) can activate another role (junior role). In Type-2 interaction shown in Figure 4, the transition from "Idle" to "Active" activates the senior role using some temporal constraints. The transitions from "Active" to "Active" activates a junior role with another temporal constraint. Thus, unless the senior role is active the junior role cannot be active.
- **SoD**: This constraint is incorporated using Type-4 interaction. In SoD, whenever a user has activated a role, it cannot activate another conflicting role at the same time interval. In Type-4 interaction shown in Figure 6, two transitions from "Idle" to "Active" are used for activation of two conflicting roles. Any one of the transition is enabled at a time, and with this enabled transition, one of the roles is active. Unless the user has deactivated the role, it cannot activate the other role.
- **Trigger**: In GTRBAC, an event can trigger another event, which is also captured in the proposed scheme. When a role is enabled, it may enable another role. For example, when a role r_j is enabled, it can enable another role, say r_k. This is achieved by adding a self loop at the "Enabled" location of the r_j role timed automaton. This self loop transition is labeled with an enabling action. Role r_k is enabled after receiving this action. In the controller timed automaton there is no transition for r_k. So unless r_i is enabled, r_k will not be enabled. In Figure 8 the role timed automaton is shown which triggers another role to move to the enabled state.

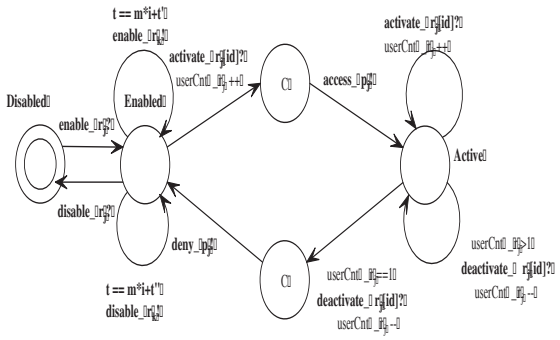

Figure 8: An enabled role triggers another role to enable

Thus it is seen that all the features of GTRBAC can be appropriately represented using timed automata.

5. SECURITY QUERIES FOR GTRBAC

In this section, we propose a desirable set of security queries that can be used for the security analysis of GTRBAC. The queries are specified using temporal logic, which expresses time as a sequence of states. Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) are two variants of temporal logic. In LTL, the future is considered as a single path while CTL models time as a tree like structure where the *future* may consist of different paths. Two common operators used to express state formulae are G, which means "globally" or "in all states", and F, which means "finally" or "there exists some state". In our proposed scheme, we use CTL as it has the feature to express quantifiers over paths of reachable states. Two path quantifiers - A for "all the paths" and E for "there exists a path", are frequently used in specifying the queries. A set of constraints applicable to GTRBAC is listed in Table 1.

The predicate $enable(r, t)$ asks whether a role r can be enabled at a particular time instance t. To ensure safety, a query can be asked whether a role remains enabled at some *unfavorable condition.*

DEFINITION 2. Unfavorable Condition: *An unfavorable condition w.r.t an event is defined as a time instance, or some condition like enabling or activating of a role, which must not hold for the event to be true.*

An *unfavorable condition* is represented here by κ which is a time instance (possibly a periodic time instance) or some event unfavorable for role r. Thus κ w.r.t a role activation event $r.active$ may be defined as $t = t_k$ or $r'.active$ where r and r' are two conflicting roles. So, in general, a safety query related to role enabling constraint is "A role r should never be enabled at some unfavorable condition". In our proposed scheme this can be expressed in CTL as follows - $AG(r.enabled \rightarrow \neg\kappa)$. Here AG means "in all the paths and in all states". Similar reasoning holds for predicate $active(r, t)$. In CTL, the safety property related to role activation can be expressed as $AG(r.active \rightarrow \neg\kappa)$.

Table 1: GTRBAC Constraints and Semantics

Constraints	Representation	Semantics
`Role Enabling`	$enable(r,t)$	whether r is enabled at time t
`User Role Activation`	$activate(u,r,t)$	whether u can activate r at time t
`Role Activation`	$active(r,t)$	whether r is active at time t
`Role Hierarchy`	$r\_active(r1,r2,t)$	whether users of r1 can activate r2 at time t
`SoD`	$(\pi, SSoD_s(r1,r2,\ldots,rn,u))$	in the interval π u should not activate more than one role from $r1,r2,\ldots,rn$
`Trigger`	$e1 \rightarrow e2$	event e1 triggers event e2

Another predicate related to user role activation constraint is $activate(u,r,t)$, i.e., it asks whether a user u can activate role r at time t. Here a general safety query can be stated as "A user should never activate a role at some unfavorable condition". This can be expressed in CTL as $AG((u.active \rightarrow r.active) \rightarrow \neg\kappa)$.

Next we consider the role hierarchy constraint. Predicate $r\_active(r1,r2,t)$ represents a role hierarchical relation. Here $r1$ is a senior role and $r2$ is a junior role. The relation indicates that the users of role $r1$ can activate role $r2$ at time instance t. So in this context, a safety query is "Whether a user of a senior role $r1$ activates a junior role $r2$ at some unfavorable condition". This is an example of activation-time hierarchy constraint, i.e., a user can activate the junior role only if it has already activated the senior role. Other types of temporal hierarchies can also be addressed but for that the necessary adjustments are to be made in the role timed automaton. In CTL the property can be expressed as $AG(((u.active \rightarrow r1.active) \rightarrow (u.active \rightarrow r2.active)) \rightarrow \neg\kappa)$

Separation of Duty (SoD) is an important constraint in GTRBAC. During a particular time interval if a user has activated a role then he should not be allowed to activate any other conflicting role. Here a strong form of interval constraint on Static SoD is considered. From this constraint, another important security property can be obtained- "A user should never activate more than one conflicting roles within a particular time interval". In CTL, this can be expressed as $AG(((u.active \rightarrow r1.active) \rightarrow \neg(u.active \rightarrow r2.active)) \vee ((u.active \rightarrow r2.active) \rightarrow \neg(u.active \rightarrow r1.active)))$

Another constraint in GTRBAC is through the use of triggers. An event $e1$ may trigger another event $e2$ which is expressed as $e1 \rightarrow e2$. For example, when a role is enabled, it can enable another role. Here one is the cause for the other. So a safety requirement may be "An event $e1$ should always trigger another event $e2$". This can be expressed as $AG(e1 \rightarrow e2)$.

Along with safety queries it is also important to specify some liveness queries to ensure that the system is doing something good as well. For liveness, a property should eventually be satisfied at some reachable state. So a general liveness query related to role enabling constraint is "A role should eventually be enabled at some favorable condition".

Definition 3. Favorable Condition: *A favorable condition w.r.t an event can be defined as a time instance, or some specific event like enabling or activation of a role, that must be satisfied when the event is true.*

A *favorable condition* is represented by μ. In CTL, the liveness query mentioned above can be specified as $EF(r.enabled \wedge \mu)$. Here EF means "there exists a path in which there exists a state".

Similar to the role enabling constraint, a liveness query can be applied on role activation constraint as well. For example, "a role will eventually be active at some favorable condition." This can be expressed in CTL as $EF(r.active \wedge \mu)$. A liveness query can also be asked on user role activation constraint. This can be put as "a user can eventually activate a role at some desirable situation". In CTL, this is expressed as $EF((u.active \rightarrow r.active) \wedge \mu)$.

Next, role hierarchical relation is considered. Here a general liveness property can be put as "A user of role $r1$ will eventually activate another role $r2$ at some favorable condition". The equivalent CTL expression can be written as $EF(((u.active \rightarrow r1.active) \rightarrow (u.active \rightarrow r2.active)) \wedge \mu)$.

On SoD constraint, we can state a liveness requirement as "A user can eventually activate two conflicting roles at some favorable condition." In CTL this can be expressed as $EF((((u.active \rightarrow r1.active) \rightarrow \neg(u.active \rightarrow r2.active)) \vee ((u.active \rightarrow r2.active) \rightarrow \neg(u.active \rightarrow r1.active))) \wedge \mu)$.

Finally, the trigger constraint is considered. A liveness query on a triggering constraint could be - "An event $e1$ should eventually trigger another event $e2$". In CTL, we can specify the property as $EF(e1 \rightarrow e2)$.

6. VERIFICATION RESULTS

The verification problem for real time systems is PSPACE complete [22]. In this section we consider different configurations of a system by varying the number of roles, users and permissions and study the time and space required for verifying the properties. At first, the following example is considered.

Example 1. *Let us consider an organization for which the role set is represented as $R = \{r0, r1, \ldots, r4\}$, user set as $U = \{u0, u1, \ldots, u16\}$ and permission set as $P = \{p0, p1, \ldots, p4\}$. For each role, a single permission is associated.*

Some of the important access control policies be specified as follows:

- *Role r0 is enabled daily at 10:00 and disabled at 17:00.*
- *Roles r1 and r2 are enabled daily at 11:00 and disabled at 18:00.*
- *Role r3 is enabled daily at 13:00 and disabled at 16:00.*
- *Whenever any role is active, the permissions associated with it are also accessible.*
- *Enabling of r1 triggers enabling of r4.*

- *Users of role r0 can activate role r2 also (so here r0 is a senior role and r2 is a junior role).*
- *User u6 can activate both r0 and r1 simultaneously.*
- *User u12 can activate either role r1 or r3 (This is an SoD constraint, i.e., u12 is not allowed to activate both roles r1 and r3 at the same time).*

With this information, a timed automata based model is constructed. For verification, the following experimental set up is considered - 2 GHz Intel Pentium Processor with 3 GB RAM running Linux OS. A timed automata based verification tool named Uppaal [24] is used with the following options – breadth first search, conservative state space reduction, Difference Bound Matrix (DBM) state space representation, automatic extrapolation and 16 MB hash table size.

At first the controller automaton is built. Transitions are labeled with temporal constraints for role enabling and disabling. Next, the role timed automaton is constructed for each role. Here five such automata are required for five roles. For each role, there is a permission. For each permission, a permission timed automaton is designed. Finally, a user automaton is constructed for each user. Depending upon the user's association with roles, they behave differently. For example, users of role *r0* can activate role *r2* and hence, the users $u0, u1, \ldots, u6$ are represented by Type-2 interaction. Again, since user *u12* can either activate role *r1* or *r3* at a time, it is modeled as a Type-4 interaction.

Next some of the safety and liveness properties are formed using the guidelines described in Section 5. In Table 2, the time and space requirements for the verification of these safety and liveness properties are shown. Here a system configuration is characterized by the $user : role : permission$ ratio. Various configurations are considered by varying the number of roles, permissions and users. It is observed that both verification time and space are dependent on the number of automata, (commonly known as processes in Uppaal) in the system. Also, since clock constraints are used in each user automaton, the state space size is quite sensitive to the number of users. It is observed that in the configuration 17:5:5, if two users are increased, then both time and space required increase drastically.

In [20], clock constraints are used in the role timed automaton. But the current model performs better than that model. For the configuration 16:3:3, the scheme proposed in [20] requires 39.603s time and 294,520 number of states for verification of a safety query. But if the same safety query is verified in the current scheme using the configuration 16:4:4, it takes only 16.879s and 209,949 number of states. So this scheme performs better compared to the other approach. The detailed result for this model is shown in Table 2.

The proposed model suffers from state space explosion if large number of users are considered. Though a single clock variable is used in the model, yet temporal constraints are required to be specified in each user timed automaton. The size of DBM data structure is proportional to the number of temporal constraints. So with the increase in users, such constraints also increase. This affects the verification process. Thus to verify a large system with many users, state space explosion may occur. One possible solution to verify a large system would be to reduce the number of temporal constraints. In [19], an attempt was made to verify the properties in a temporal RBAC model, where temporal constraints on role activation are only considered. But obviously, it is not sufficient in a realistic application.

7. CONCLUSIONS

In this paper a state transition model for GTRBAC has been proposed. We have utilized the expressibility of timed automaton. It is seen that a timed automaton can express any temporal constraint using its clock variables. The proposed model maps the behavior of components such as users, roles, permissions. The diverse behaviors of users are captured by creating automata of different types of interaction. The location in each automaton represents the state of the corresponding component. A desirable set of security properties is constructed from the constraints specified in the GTRBAC model. These properties are then used for verification. To get an efficient result, a single global clock variable is used to express the temporal constraints. In Section 6, it is seen that the verification process is affected by the state space explosion problem. Verifying the result for a large system, thus, still remains a challenge. To perform analysis of a large system, one may have to limit the temporal constraints. So a trade off has to be made on whether to examine a system with large number of temporal constraints or one with large number of users, roles and permissions.

8. ACKNOWLEDGMENTS

This work is partially supported by a research grant from the Department of Science and Technology, Government of India, under Grant No. SR/S3/EECE/ 082/2007 and by the National Science Foundation under Grant No. IIS-0306838.

9. REFERENCES

[1] E. Bertino, P.A. Bonatti, and E. Ferrari. TRBAC: A temporal role based access control model. *ACM Transactions on Information and System Security*, 4(3):191–233, August 2001.

[2] V. Atluri and A. Gal. An authorization model for temporal and derived data: Securing information portals. *ACM Transactions on Information and System Security*, 5(1):62–94, February 2002.

[3] I. Ray and M. Toahchoodee. A spatio temporal role based access control model. In *21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 211–226, Jul 2007.

[4] S. Aich, S. Sural, and A. K. Majumdar. STARBAC: Spatiotemporal role based access control. In *Information Security Conference, LNCS, Springer-Verlag*, pages 1567–1582, November 2007.

[5] J.B.D. Joshi, E. Bertino, U. Latif, and A. Ghafoor. A generalized temporal role-based access control model. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):4–23, January 2005.

[6] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[7] A. K. Jones, R. J. Lipton, and L. Snyder. A linear time algorithm for deciding security. In *Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science*, pages 33–41, October 1976.

Table 2: Time and Space Analysis

User:Role:Perm	Query Type	CTL Representation	Time	States Explored
12 : 4 : 4	`Safety`	$AG(r0.Disabled \rightarrow \neg(t > (24 * i + 10) \wedge t < (24 * i + 17))$	0.954s	19,914
12 : 4 : 4	`Liveness`	$EF(r0.Enabled \wedge (t == 10))$	0.034s	1
16 : 4 : 4	`Safety`	$AG(r0.Disabled \rightarrow \neg(t > (24 * i + 10) \wedge t < (24 * i + 17))$	16.879s	209,949
16 : 4 : 4	`Liveness`	$EF(r0.Enabled \wedge (t == 10))$	0.037s	1
17 : 5 : 5	`Safety`	$AG(r0.Disabled \rightarrow \neg(t > (24 * i + 10) \wedge t < (24 * i + 17))$	1m23.971s	703,871
17 : 5 : 5	`Liveness`	$EF(r0.Enabled \wedge (t == 10))$	0.038s	1
17 : 5 : 5	`Safety`	$AG(u0.Active \rightarrow r1.Active)$	39.290s	703,871
17 : 5 : 5	`Liveness`	$EF(u0.Active \wedge (t == 11))$	0.038s	4
19 : 5 : 5	`Safety`	$AG(r0.Disabled \rightarrow \neg(t > (24 * i + 10) \wedge t < (24 * i + 17))$	36m16.443s	11,051,838
19 : 5 : 5	`Liveness`	$EF(r0.Enabled \wedge (t == 10))$	0.040s	1

[8] R. S. Sandhu. The schematic protection model: Its definition and analysis for acyclic attenuating systems. *Journal of the ACM*, 35(2):404–432, 1988.

[9] M. Koch, L. V. Mancini, and F. Parisi-Presicce. Decidability of safety in graph-based models for access control. In *7th European Symposium on Research in Computer Security*, pages 229–243, 2002.

[10] T. Ahmed and A.R. Tripathi. Static verification of security requirements in role based CSCW systems. In *8th ACM Symposium on Access Control Models and Technologies*, pages 196–203, June 2003.

[11] N. Li and M.V. Tripunitara. Security analysis in role based access control. *ACM Transactions on Information and System Security*, 9(4):391–420, 2006.

[12] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman. Efficient policy analysis for administrative role based access control. In *14th ACM Conference on Computer and Communications Security*, pages 445–455. ACM, October 2007.

[13] Y. Zhang and J. B. D. Joshi. UAQ: A framework for user authorization query processing in RBAC extended with hybrid hierarchy and constraints. In *13th ACM Symposium on Access Control Models and Technologies*, pages 83–92. ACM, June 2008.

[14] S. Jha, N. Li, M. Tripunitara, Q. Wang, and W. Winsborough. Towards formal verification of role based access control policies. *IEEE Transactions on Dependable and Secure Computing*, 5(4):242–255, 2008.

[15] J. B. D. Joshi, E. Bertino, and A. Ghafoor. An analysis of expressiveness and design issues for the generalized temporal role-based access control model. *IEEE Transactions on Dependable and Secure Computing*, 2(2):157–175, April 2005.

[16] B. Shafiq, A. Masood, J. Joshi, and A. Ghafoor. A role based access control policy verification framework for real time systems. In *10th IEEE International Workshop on Object Oriented Real Time Dependable Systems*, pages 13–20, 2005.

[17] R. Alur, C. Courcoubetis, and D. L. Dill. Model checking for real time systems. In *5th Symposium on Logic in Computer Science*, pages 414–425, 1990.

[18] A. Furfaro and L. Nigro. Temporal verification of communicating real time state machines using Uppaal. In *IEEE International Conference on Industrial Technology*, pages 399–404, 2003.

[19] S. Mondal and S. Sural. Security analysis of Temporal RBAC using timed automata. In *4th International Conference on Information Assurance and Security*, pages 37–40, September 2008.

[20] S. Mondal and S. Sural. A verification framework for temporal RBAC with Role Hierarchy (Short Paper). In *4th International Conference on Information and Systems Security*, pages 140–147, December 2008.

[21] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[22] C. Baier and J. P. Katoen. *Principles of Model Checking*. MIT Press, Cambridge, MA, 2008.

[23] F. Laroussinie, N. Markey, and Ph. Schnoebelen. Model checking timed automata with one or two clocks. In *15th International Conference on Concurrency Theory*, pages 387–401, 2004.

[24] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In *4th International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 200–236, 2004.