Building a Stateful Reference Monitor with Coloured Petri Nets

Basel Katt, Ruth Breu and Michael Hafner
University of Innsbruck
Innsbruck, Austria
{basel.katt, ruth.breu, m.hafner}@uibk.ac.at

Xinwen Zhang
Samsung Information Systems, America
San Jose, CA, USA
xinwen.z@samsung.com

Abstract

The need for secure information sharing has been recently growing dramatically with the broadened spread of social networks, the convergence of outsourcing and offshoring, and the increasing need for collaborations between different partners/competitors. New access control requirements have emerged in these modern collaborative and distributed environments, such as continuous control of resource usage considering temporal and cardinal rules, execution of additional tasks to compensate violation of security policies or enforce obliged actions, and constraints for concurrent access and usage of shared resources. These new requirements stipulate the need for new policy models and advanced enforcement mechanisms. Towards these we aim at developing a formal framework based on Coloured Petri Nets theory for the specification of enforcement mechanisms of a resource-centric reference monitor.

1. Introduction

The need for secure information sharing has been dramatically growing with the broadened spread of the social networks, the convergence of outsourcing and offshoring, and the increasing need for collaborations. In order to achieve their goals, especially after the recent financial crises, companies tend to merge or collaborate with partners and competitors to cut costs. Outsourcing requires the transfer of resources to a third party company for processing. Healthcare records of citizens stored in distributed healthcare network have to be shared and modified by different stakeholders in healthcare domains. Social networks, on the other hand, enable users to share some personal information and document that can be accessed by different users.

In these modern collaborative and distributed environments new security requirements arise. (i) Usually, it is not enough to make authorization decisions based on the static rules of each operation, since the behavior of applications and the temporal interaction of security related operations and actions play critical roles in security [12]. For example, only users having already joined a discussion room can have access to a shared document. (ii) Beside authorization decisions, it is important to be able to execute some actions to compensate violations of security rules and enforce some obligatory tasks. E.g., updating attributes of users and resources [19] or sending a notification to the clinic in case of a privacy violation by a healthcare actor. (iii) Concurrency is a very important aspect in collaborative environments when resources are shared. These new requirements stipulate the need for new policy models and advanced enforcement mechanisms. They also show the importance considering application functional behavior in proposed policy models and enforcement mechanisms.

Towards these requirements we aim to build stateful security policy model and enforcement mechanism, instead of traditional static and stateless approaches. We identify three views of a system (or an application or program): (i) the functional view which defines the behavioral rules or the functional behavior of the system, (ii) the security view which defines the security rules (policy) that must be applied for each action, and (iii) the enforcement view which defines the secure behavior that has to be enforced or monitored by the reference monitor of the system. In other words, the enforcement view uses both functional and security views to define the controlled behavior of the reference monitor, which ensures that the behavior is safe and the security rules are fulfilled. In this paper, our focus is on the enforcement view.

Problem Statement: Plenty of work has been dedicated to formalize and analyze security policies (For example, recently, Krishnen et al. [12] have identified a rich space of security policies in Group-centric information sharing systems.). However, fewer have payed the same attention to enforcement mechanisms. Developing new policy languages without considering their enforcement mechanisms increases the gap between the policies expressed and the actual enforcement [8], [4], thus the correctness of the policy analysis is under question when a policy is to be enforced. This is true especially when the policy is dynamic and stateful. Hence developing a formal framework for enforcement mechanisms is very desired to correctly derive necessary and safe mechanism from policies and thus enable
analyzing the policies and mechanisms at the same level of abstraction.

Monitors for program execution have been investigated by a stream of work started by Schneider et al. [17], and lately by Pretschner et al. [14] and Janicke et al. [8]. However concurrency seems to have taken a back seat. Plenty of work has proposed enforcement architectures/models, for example, Zhang et al. [19] in the context of collaborative computing systems, however, their approaches are implementation oriented and lack a formal foundation. Other industrial standards deal with reference monitor based enforcement mechanism such as XACML architecture [2], however it does not fit well for stateful policies, where the behavior and the state of the reference monitor play an important role in making decisions.

Solution: In this paper we establish a formal framework to specify and reason about access control enforcement mechanisms for an object-centric reference monitor, which means that objects and resources are central elements of our reference monitor. Our reference monitor is an abstract state machine based on the Petri Nets formalism. It intercepts security (resource) related sequence of system actions and ensures that the system behaves securely with respect to security conditions, concurrency rules, and behavioral restrictions. In general, the reference monitor has the following advanced enforcement mechanisms: (1) terminating the usage of a resource by any user who tries to violate any security condition without affecting the usage of other users; (2) executing additional actions to compensate for any violation or to enforce obligatory tasks; (3) applying temporal restriction on action execution; (4) applying cardinal restrictions on the execution of actions; and (5) applying concurrency rules when multiple users are executing sensitive actions on resources. Concurrency is one of those aspects that are dealt with in our treatise of reference monitor, in which different users accessing some resources can be modeled.

Mechanisms are modeled with (parts of) Coloured Petri Nets (CP-nets) and security rules or properties are defined based on the CP-net’s formalism. We do not claim that these mechanisms are complete – any access control policy can be enforced by one or more mechanisms of our reference monitor. However, we try to establish an extensible framework for defining enforcement mechanisms based on identified requirements of access control systems. On the other hand, we show that the mechanisms are sound. We specify a security property that is enforced by each mechanism. In this case, a mechanism that enforces a property is sound if the property holds for the mechanism’s CP-net. In other words, we say that a set of enforcement mechanisms of our reference monitor enforces a set of security properties or rules if these rules hold for these mechanisms’ CPN.

Outline: The rest of this paper is organized as follows.

In next section we discuss the related work. An overview of Coloured Petri Nets is presented in Section 3. Basic concepts and main notations used in our stateful object-centric reference monitor are discussed in Section 5. In Section 6 enforcement mechanisms are specified and their correctness is shown with respect to security properties. The combination of different mechanisms and analysis are discussed in Section 7 by considering a simple example. We conclude the paper and discuss our future work in Section 8.

2. Related Work

Major work in this field is the stream of papers started by introducing the concept of execution monitor as security automata by Schneider et al. in [17], [18], followed by [7], [5], [6], and finally the edited automata developed by Ligatti et al. [13]. In this line of work, a reference monitor basically aims at monitoring a program’s behavior to avoid untrusted and non-safe code to be executed. The authors distinguish two implementations of a reference (execution) monitor: one can be positioned between system service entry points and the code providing the services, the other is injected into client program at load time. That is, the main purpose of these reference monitors is to ensure that the code executed by a program is safe, i.e., unacceptable behavior from untrusted programs is prevented. That is why the reference monitor is instrumented into the code of the suspected program or wraps it. We call these monitors code or application centric monitors. Comparing to these approaches, our reference monitor aims at protecting shared resources as well as monitoring system behavior. Systems in our case can be composed by different applications and users, with respect to operations related to the protected resources. Pretschner et al. [14] and Janicke et al. [8] have recently investigated reference monitors for usage control and history based policies, respectively. Concurrency tends to be overlooked by all these approaches, which is one of the issues we are tackling in this contribution. By showing that what is enforced by edit automata is not exactly what is claimed in the policy, authors in [4] show the importance of further investigating enforcement mechanisms for stateful policies.

Due to the fact that petri nets is used in our approach, another direction of related work that can be argued is in the workflow community. There is a stream of work that applies the formalism of PN to authorization in workflows, most notably [3], [11], and others. Concurrency and execution mechanisms, despite the fact that petri nets were used, are tend to be overlooked. Furthermore, the notion of reference monitor and distinguishing between policies and mechanisms is not considered.
3. Colored Petri Nets Overview

A Colored Petri Net (CP-net) [9] can be defined as a tuple $CPN = (\Sigma, P, T, A, N, C, G, E, I, SC)$, where $\Sigma$ is a set of color sets, $P$, $T$ and $A$ are sets of places/states, transitions and arcs, respectively. A transition has incoming and outgoing arc(s). Incoming arcs indicate that a transition may remove one or more tokens from the corresponding input places while outgoing arcs indicate that the transition may add tokens to the output places. The exact number of tokens and their values are determined by the arc expression, defined by the function $E$. $N$ is a node function that determines the source and destination of an arc. $C$ is a color function that associates a color set $C(p)$ or a type with each place $p$. $G$ is a guard function that maps each transition $t$ to a boolean expression $G(t)$. For a transition to be enabled, a binding of the variables that appear in the arc expressions must be found, and for this “binding element” the guard function must evaluate to true. This binding makes the arc expression of each input arc evaluates to a multi-set of token colors. $I$ is an initialization function that maps each place to a multiset $I(p)$. The last element is $SC$, which is the segmentation code function of a transition, mapping a transition to a set of actions that are executed when the transition occurs.

A token element is a pair $(p, c)$ such that $p \in P$ and $c \in C(p)$. For a color set $s \in \Sigma$, the base color sets of $s$ are the color sets from which $s$ was constructed using some structuring mechanisms such as cartesian products, records, or unions. The set of all token elements is denoted by $TE$.

For $x, x_1, x_2 \in P \cup T$, $Out(x) = \{y \in P \cup T \mid \exists a \in A : N(a) = (x, y)\}$ is the postset of $x$; and $In(x) = \{y \in P \cup T \mid \exists a \in A : N(a) = (y, x)\}$ is the preset of $x$. $A(x_1, x_2)$ is the set of arcs from $x_1$ to $x_2$, and the expression of $(x_1, x_2)$ is $E(x_1, x_2) = \exists a \in \mathcal{A}(x_1, x_2) E(a)$.

$Var(t)$ is the set of variables of a transition $t$. $Type(v) \in \Sigma$ denotes the type of the variable $v$. A binding element $(t, b)$ is a pair consisting of a transition $t$ and a binding $b$ of data values to its variables such that $G(t) < b >$ evaluates to true. $expr < b >$ in general denotes the value obtained by evaluating the expression $expr$ in the binding $b$. By $B(t)$ we denote the set of all bindings for a transition $t$. The Binding element is written in the form $(t, c_1 = v_1, c_2 = v_2, \ldots, c_n = v_n)$, where $v_1, v_2, \ldots, v_n \in Var(t)$ are the variables of $t$ and $c_1, c_2, \ldots, c_n$ are the data values such that $c_i \in Type(v_i)$ for $1 \leq i \leq n$. For a binding element $(t, b)$ and a variable $v$ of $t$, $b(v)$ denotes the value assigned to $v$ in the binding $b$. $B(t)$ denotes the set of all binding elements is denoted $BE$.

$M(p)$ denotes the marking of a place $p$ in the marking $M$. $M_0$ is the initial marking. If a binding element $(t, b)$ is enabled in a marking $M_1$, denoted $M_1[(t, b)]$, then $(t, b)$ may occur in $M_1$ yielding some marking $M_2$. This is written as $M_1[(t, b)] \rightarrow M_2$. Accordingly, a finite occurrence sequence is a sequence consisting of a marking $M_i$ an binding elements $(t_i, b_i)$ denoted $M_1[(t_1, b_1)]M_2[\ldots]M_{n-1}[(t_{n-1}, b_{n-1})]M_n$ and satisfying $M_i[(t_i, b_i)]M_{i+1}$ for $1 \leq i < n$. $M_1$ is called start marking, $M_{n+1}$ is called end marking, and $n$ is called the length of the occurrence sequence. If the length is infinite we call the occurrence sequence infinite occurrence sequence. The set of all finite occurrence sequences is denoted by $OSF$, while the set of all infinite occurrence sequences is denoted by $OSI$, and finally $OS = OSF \cup OSI$ is the set of all occurrence sequences. A reachable marking is a marking which can be obtained by an occurrence sequence starting in the initial marking. $[M_0]$ denotes the set of reachable markings. Finally, The sets of all markings and steps is denoted by $M$ and $Y$, respectively.

Let $X \subseteq BE$ be a set of binding elements and $\sigma \in OSF$ is a finite occurrence sequence, we can also consider an infinite one, of the form: $\sigma = M_1[Y_1]M_2 \ldots M_n[Y_n]M_{n+1}$. For each $i \in N_+$, $EN_{X,i}(\sigma)$ is the number of elements from $X$ which are enabled in the marking $M_i$ and $OC_{X,i}(\sigma)$ is the number of elements from $X$ which occur in the step $Y_i$. Furthermore, $EN_X(\sigma) = \sum_{i \in N_+} EN_{X,i}(\sigma)$ and $OC_X(\sigma) = \sum_{i \in N_+} OC_{X,i}(\sigma)$ are the total number of enabling and occurrences in $\sigma$, respectively.

4. Motivating Example

The following example illustrates the concepts of our reference monitor. We consider a collaborative Modeling applications between two automaker Companies: $Cop_1$ and $Cop_2$. According to a cooperation agreement and in order to cut costs, the two competitors agree to co-source some of the car components. This cooperation on purchasing components requires integrating some of the financial and purchasing business processes by specifying an inter-organizational workflow between the financial and purchasing departments in the two companies. They decided to use a collaborative modeling application, similar to the one proposed by Rittgen in [15], [16], to enhance the modeling procedure.

The collaborative modeling tool provides the following two features $3$: proposal and score. proposal is a suggestion by a user for a revision of the current version of the model. This proposal is to be evaluated by other users participating in the modeling session. Each user gives a score for the proposal and according to some threshold, the proposal can be accepted or rejected. If accepted the proposal can be committed to the shared model repository. We can summarize the following functionalities required from the collaborative application: (1) propose to propose a proposal, (2) request for a user to request evaluation of the proposed proposal, (3) commit to commit the proposal to the model repository, (4) update to get the latest version of the model, (5) score to give a score for a proposal, and finally (6) join and leave the modeling session.

3. These features are inspired from the COMA tool.
**Requirement**: Due to the sensitivity of the information contained in the models, some behavioral and security requirements must be enforced.

**BR1**: Committing is only allowed after the user proposes a proposal and requests evaluations.

**BR2**: After requesting evaluation, users can withdraw the proposal.

**SR1**: Only users with role “Designer” are allowed to propose proposals.

**SR2**: A maximum number of ten proposals are allowed for each model.

**SR3**: Requesting must not last more than 5 minutes for each proposal.

**SR4**: Every time a new version is committed, a notification message must be sent to all participants.

**SR5**: Only one user is allowed to commit at a time.

As aforementioned, we distinguish three views on an access control system: the functional view, the security view, and the enforcement view. While the security view represents security and concurrency related rules that must be applied when each action in the system is invoked, the other two views represent system and reference monitor behaviors, respectively. Functional behavior is defined as the required behavior of the system when a subject tries to use resources. The functional behavior is modeled as a Coloured Petri Net. In general it can be represented by a transition system or an automata. The enforcement view represents the expected behavior of the reference monitor after taking the functional behavior and the security rules into account.

Each action in the system is modeled as one transition in the functional view and is mapped into a sequence of Transition-Place-Transition in the enforcement view. The first transition represents the action request and the start of executing the action, the place represents the execution state of the resource by the subject, and the second transition represents the end of the action execution. In this context we define the following help functions:

- \( actS \in [ACT \rightarrow T] \): maps each action in the system to a transition in the functional behavior.
- \( actRM \in [ACT \rightarrow T \times P \times T] \): maps each system action to a set of two transitions and one place. We call them \( reqT, exeP, \) and \( endT \), representing the request transition, executing place, and the end transition, respectively. For example \( t1 = actRM(act).reqT, t2 = actRM(act).exeP, \) and \( p = actRM(act).exeP \) represent the request transition, end transition, and execution place of the action \( act \in Act \), respectively.
- \( time \in [OSF \rightarrow N] \): defines the duration of a finite occurrence sequence \( \sigma \in OSF \).

Figure 1 shows that required behavior modeled as CPN. The transitions represent system actions discussed previously, each abbreviated by the first 3-4 letters. *subjects* place represents a pool of available subject. The figure shows that *com* action is only allowed after the *req*, and *withdraw* action is available after requesting evaluations.

**5. Stateful Object-centric Reference Monitor**

In this section we establish the formal framework and basic notations for modeling and analyzing reference monitors and security rules they enforce. We specify a system at a high level of abstraction as a non-empty set of actions \( Act \) which is modeled in our CPN notions as a set of transitions \( T \). An execution is a finite sequence of actions, or -in the CPN notion- a finite occurrence sequence \( OSF \). Furthermore, the system contains a set of subjects \( S \) represented by the token colour \( SUB \), and set of objects \( O \) represented by the token colour \( OBJ \). Finally, we define a finite set of security related conditions \( Cond \).

![Diagram](image.png)

Figure 1: Example of CP-net representing the functional behavior for using a socket

Figure 2 shows the mapping between the functional and the enforcement representation of actions. When the reference monitor intercepts an action \( act \in Act \) that a subject \( s \in S \) is trying to execute, this request for execution is represented as a binding \((t1, b1)\), where the transition \( t1 = actRM(act).reqT \) represents the request transition of the action \( act \) in the enforcement view and \( s = b1(sub) \), i.e., the subject element of the binding \( b1 \) equals the subject \( s \) that tries to execute the action. If the binding element is enabled in a marking \( M1 \), the action is recognized by the CPN reference monitor. The occurrence of this binding

4. Please note that we do not restrict any specific model or language for the conditions to keep our model general. For example, it can represent RBAC constraints, attributes based constraints, or others.
means that the action is allowed and it moves the subject token \( sub = s \) to the executing place \( actRM(\text{act}).exeP \). Beside allowing the execution by the reference monitor the assigned control mechanisms for this action is executed (checked). Those mechanisms are explained in details in the next section. The \( exeP \) place represents the execution state of an action by subjects whose tokens are placed in this place. Finally, the occurrence of the binding \((t_2, b_2)\) in a marking \( M_2 \) represents the end of execution of the action \( act \), where \( t_2 = actRM(\text{act}).\text{endT} \) represents the end transition of the action \( act \) in the enforcement view, and \( b_2 \) is an enabled binding having the subject \( s = b_2(sub) \) as its subject element. We call the finite occurrence sequence between the markings \( M_1 \) and \( M_2 \), the execution sequence \( ES \) which is a part of the access session.

**Definition 1.** An access session of a resource or a set of resources is defined as a finite occurrence sequence \( \sigma = M_0[Y_1\ldots Y_n]M_n \), where \( M_0 \) is the initial marking of the access and the \( M_n \) is the final marking where the access is finished.

For example, in an online meeting or conference the access session represents the sequence of actions that all participants perform on the shared resource during the meeting period. It starts by initiating the conference session by the conference administrator and ends by closing the conference session by the administrator.

From the discussion above we can conclude that our reference monitor can be seen as (1) a recognizer of the functional behavior of the system, with respect to the protected resources, and (2) a controller of the actions that a subject is executing on resources. The following control mechanisms are supported by our reference monitor:

1) **Halting Mechanism:** stops the execution by a subject. It prevents the violation of security rules. This can be mapped to the mechanism of security automata introduced by Schneider et al. [17], [18].

2) **Temporal Mechanism:** allows the execution of an action by a subject for a specific time period.

3) **Cardinal Mechanism:** allows the execution of an action by a subject for a maximum number of times.

4) **Execution Mechanism:** allows the execution of additional actions/tasks in order to compensate a violation of security rules or enforce some needed tasks. These actions can be either executed instantly or within a specific time frame.

5) **Concurrency Mechanism:** this mechanism controls concurrent executions of an action by multiple subjects.

In the next section we tackle these mechanisms in more details.

### 6. Abstract Enforcement Mechanisms

Using Coloured Petri Nets for specifying enforcement mechanisms has two advantages. First, it provides a mathematical framework to reason about and analyze the enforcement mechanisms. Hence, it enables proofing the correctness of the enforcement with respect to security rules to be enforced. Second, it can be used for automatic configuration of an enforcement engine based on CPN-based engines, thus closing the gap between a policy specification and the reference monitor implementation. This gap can lead to insecure systems as the overall security depends on the correctness of the mechanisms used to enforce sound policies.

Each mechanism is triggered or associated with one system action. We define a general abstract enforcement mechanisms as 3-tuple \( \text{Mech} = (\text{act}, o, \text{PAR}) \), where \( \text{act} \in \text{Act} \) is the triggering system action, \( o \in O \) is the protected resource, and \( \text{PAR} \) is a set of parameters that depend on each individual mechanism. For each of the defined enforcement mechanisms we specify the security property it enforces and show the correctness of the enforcement.

#### 6.1. \text{Mech}_1: Halting Mechanism

A halting mechanism can be defined as a 3-tuple \( \text{Mech}_1 = (\text{act}, o, co) \in \text{HMech} \), where \( co \in \text{Cond} \) is a condition that must be fulfilled instantly and \( \text{HMech} \) is the set of all halting mechanisms. In other words, it must fulfill an Instant property. Instant properties represent security rules that must be checked before a subject gets access to a resource. It can be mapped to traditional access control policies. Formally, we define the instant property that the condition \( co \in \text{Cond} \) must be fulfilled before a subject \( s \in \text{SUB} \) execute an action \( \text{act} \in \text{Act} \) on a resource \( o \in \text{OBJ} \) as follows (where \( t_1 = actRM(\text{act}).\text{req} \) and \( s = b(sub) \)):

\[
\forall(t_1, b) \in BE. \forall M \in M : (\text{cond} = \text{true}) \land M[(t_1, b)] \land s = b(sub) \Rightarrow \exists M' \in M : M[(t_1, b)]M' \quad (1)
\]

This rule is very intuitive and simply says: in any state of the reference monitor with marking \( M \), if a condition \( \text{cond} \) is true and the transition \( t_1 \) is enabled, then the transition \( t_1 \) must occur, i.e., the action must be allowed. \( t_1 \) is enabled means that beside the condition is true, the subject \( s \) has requested the action \( \text{act} \). Generally, the request transition of an action is enabled with a binding \( b \) that contains a token of the subject \( s \) iff (1) the subject \( s \) has requested that action, which is recognized by the defined functional behavior, and (2) the guard of the request transition evaluates to true for...
this binding b. The occurrence of the binding transforms the state of the reference monitor to the marking M'. In other words, the equation 1 indicates that if the subject stated in the binding element tries to execute the action act and the condition cond is not true then his request must be rejected and his access is halted.

Figure 3: Halting mechanism.

The halting mechanism for enforcing instant rules can be simply constructed by placing the condition as a guard on the request transition of the action act. Figure 3 shows this mechanism for restricting access to the action act, whose request transition in the enforcement view is t₁, by a subject presented as the token in the S1 place. If the condition is not fulfilled the execution is halted, otherwise the request transition occurs and the subject starts executing the action.

**Proposition 1.** Mech₁ enforces instant rules.

*Proof:* The proof is straightforward consequence of the definition of step enabling and step occurrence of CP-net [9].

### 6.2. Mech₂: Temporal Mechanism

Temporal mechanism can be defined as a 3-tuple Mech₂ = (act, o, d) ∈ TMech, where d is the duration within which the execution of the action act is granted to a subject and TMech is the set of all temporal mechanisms. We call the property that the temporal mechanism enforces a temporal rule. A temporal property indicates that a subject s ∈ SUB is allowed to executed an action act ∈ Act, where t₁, t₂, and p are the request transition, end transition and executing place of the action act respectively, for the time period d ∈ N. Formally we define this temporal rule as follows (where s = b(sub)):

\[\forall (t₁, b) ∈ BE \forall M, M' ∈ M : M[(t₁, b)]M' ∧ s ∈ b(sub) \Rightarrow \exists M'' ∈ M \exists σ ∈ OSF : M''(σ)M'' ∧ s \not∈ M''(p) \]  

(2)

In this definition we do not specify the allowed duration of execution, but we require that eventually the subject ends his execution. However, it is possible to define the duration of the allowed execution by adding the condition \(time(σ) ≤ d\), then we get the definition

\[\forall (t₁, b) ∈ BE \forall M, M' ∈ M : M[(t₁, b)]M' ∧ s ∈ b(sub) \Rightarrow \exists M'' ∈ M \exists σ ∈ OSF : M''(σ)M'' ∧ s \not∈ M''(p) \land time(σ) ≤ d \]  

(3)

The end of the execution is denoted by the condition \(b \not∈ M''(p)\), which means that the token element of the subject sub leaves the executing place p in the marking M''.

The temporal mechanism for enforcing temporal rules requires timer functionality. For this purpose timed CP-nets can be used [10]. Originally, timed CP-net is used for investigating the performance of systems, for example the maximum time for execution and average waiting times for certain requests. We leverage timed CP-net to provide the mechanism to integrate time aspects in our enforcement mechanism. Specifically, some tokens are allowed to carry a time stamp which indicates when a token is ready to be used by a transition. Using timed CP-net, the temporal mechanism can be constructed by waiting \(d\) time units, and then check if the subject is still executing the action, i.e., the subject token is still in the executing place.

Figure 4 shows the CP-net representation of this mechanism (please note that the token color of the waiting place is a timed subject). When a subject starts executing the action act, the request transition occurs. The occurrence of this transition moves a subject token to the executing place, indicating that the subject is executing the action. Furthermore, a subject token with a time stamp @d moves to the waiting state. Thus, after \(d\) time units the token in the waiting state is ready and the transition timeout occurs if the subject token is still in the executing place p. In this case the subject is moved from the executing place and the execution is forced to be ended.

**Proposition 2.** Mech₂ enforces temporal rules.

*Proof:* we want to proof that the temporal property for the action act holds for the Mech₂'s CP-net. We assume that a binding element (b₁) is enabled in a marking M where \(b(sub) = s\) and \(sub ∈ M(S₁)\). The occurrence of this step gives us a new marking \(M'\) where the subject token is moved to the executing place p, i.e., \(sub ∈ M'(S₁)\). Furthermore, firing the transition \(t₁\) moves also a subject token to the place Waiting with a timestamp \(d, i.e., sub ∈ M'(Waiting)\). From this marking \(M'\) we assume that there exists a finite occurrence sequence \(σ ∈ OSF\). We can distinguish two possible cases:

**Case 1:** The subject ends the execution of the action act within the duration \(d\) by firing the end transition \(t₂\). The
occurrence of \( t_2 \) removes the subject token from the place \( p \).

**Case 2:** The subject does not end the execution of the action \( \text{act} \) within the duration \( d \). In this case, after \( d \) time units the \( \text{timeout} \) transition occurs and the subject token is extracted from the place \( p \), i.e., the execution of the action by the subject is revoked.

It is clear from both cases that from the marking \( M' \) we reach a marking \( M'' \) where \( s \notin M''(p) \). In the first case we reach the marking \( M'' \) in a period of time less than \( d \) and in the second in \( d \) time units.

\[ \Box \]

### 6.3. Mech3: Cardinal Mechanism

We define a cardinal mechanism as a 3-tuple \( \text{Mech}_3 = (\text{act}, o, n) \) \( \in \text{CMech} \), where \( n \) is the number of times an action \( \text{act} \) can be used within one access session and \( \text{CMech} \) is the set of all cardinal mechanisms. For this mechanism we define the cardinal property, which indicates that an action \( \text{act} \) can be used for only \( n \) times during each access session. Assuming an access session \( \sigma = M_0[Y_1Y_2 \ldots Y_n]M_n \), and the request transition of the action \( \text{act} \) is \( t_1 \), we define the cardinal property formally as follows:

\[
\forall Y_i = (t_1, b) \in \sigma : 0 < i < n \Rightarrow OC_{(t_1, b)}(\sigma) \leq n \quad (4)
\]

This definition indicates that within one access session the request transition of the action \( \text{act} \) can occur a maximum number of \( n \) times, i.e., the action can only be allowed for \( n \) times. This cardinal mechanism can be constructed by using a new place of \( \text{UNIT} \) type containing \( n \) unit elements and connect this place to the request transition of the action \( \text{act} \) with one headed arc as shown in Fig. 5.

\[
\forall(t_1, b) \in \text{EB} \forall M, M' \in \mathbb{M} : s \in b(\text{sub}) \land M[(t_1, b)]M' \Rightarrow \text{executed}(tsk) \quad (5)
\]

\[
\forall(t_2, b) \in \text{EB} \forall M, M' \in \mathbb{M} : s \in b(\text{sub}) \land M[(t_2, b)]M' \Rightarrow \text{executed}(tsk) \quad (6)
\]

\[
\forall(t_1, b) \in \text{EB} \forall M, M' \in \mathbb{M} : s \in b(\text{sub}) \land M[(t_1, b)]M' \Rightarrow \exists M'' \in \mathbb{M} \exists \sigma \in \text{OSF} : M'[(t_1, b)]M'' \land \text{executed}(tsk) \land \text{time}(\sigma) \leq d \quad (7)
\]

### 6.4. Mech4: Execution Mechanism

An execution mechanism is a mechanism that executes additional actions/tasks when a subject uses a resource. These tasks must be executed by the system and need not be verified for fulfillment. For example, a security rule requires that whenever a protected file is opened, the system executes logging the name of the user and the accessed resource. For this reason we define the following help function: \( \text{executed} \in [\text{Act} \rightarrow \text{Bool}] \), which maps each task to a boolean value indicating whether the task is executed or not.

Properties that these mechanisms fulfill are called execution properties, of which three types can be identified based on the time instance of executing the task: the task must be executed before the resource action is allowed, after the execution of the resource action, or after a specific period of time. Assuming that \( t_1 \) and \( t_2 \) are the request and end transitions of an action \( \text{act} \), and the task \( tsk \in \text{ACT} \) must be executed when the \( \text{act} \) is executed by a subject \( s \), then we define the three types of execution rules as follows:

**Figure 5: Cardinal mechanism.**

The main element of the this CP-net is the \( \text{Card} \) place that contains a multi-set of \( \text{UNIT} \) color. The coefficient indicates how many times the action, whose request transition is \( t_1 \), is allowed to be executed.

**Proposition 3.** \( \text{Mech}_3 \) enforces cardinal rules.

**Proof:** It is easy to see that the transition \( t_1 \) can occur \( n \) times within any finite occurrence sequence, i.e., for any access session. After \( n \) occurrences of the binding \( (t_1, b) \) the place \( \text{Card} \) contains no elements and thus the binding cannot be enabled.

**Proposition 4.** \( \text{Mech}_4 \) enforces the first type of execution rules.

6. To distinguish these actions that must be executed from the actions that a subject can execute on resources, we call the additionally executed actions \( \text{tasks} \). They are of type \( \text{ACT} \) and are represented as part of the segmentation code.
Proposition 5. \textit{Mech}_5 \textit{enforces concurrency rules.}

7. Combination of Enforcement Mechanisms

In this section we show the feasibility of our approach by building the mechanisms needed for the use case mentioned in Section 4. The example shows the need to protect the usage of the collaborative application in order to facilitate usability of such applications in the real business scenarios. It can be seen that: in order to enforce the requirements \textit{SR1-SR5} we need the following mechanisms: instant, cardinal, temporal, execution, and concurrency mechanisms, respectively. While the behavioral requirements \textit{BR1, BR2} are represented by the CPN. 7.

7.1. Building Combined Mechanisms

To build the enforcement mechanisms needed for these requirements, the following steps are required. First, the functional behavior of the system has to be transformed into the enforcement view. Each system action is represented as one transition in the system view and two transitions and one place in the enforcement’s view. For example, the \textit{propose} \in \textit{Act} action is represented by the transition \textit{prop} \in \textit{T} in the system view and the \((p_1, \textit{prop}, p_2) \in T \times P \times T\) in the enforcement view. Second, after transforming all actions into the enforcement’s view, appropriate mechanisms have to be constructed according to the security requirements 8.

Based on the requirements above mentioned, the following mechanisms are needed:

- \textit{mech}1 = \((\textit{propose}, \textit{model1}, [\textit{user.role} = \textit{“Designer”}]) \in \textit{HMech}, assuming that the user is trying to propose the model \textit{model1} and the attribute \textit{user.role} indicates the role of the user.

- \textit{mech}2 = \((\textit{propose}, \textit{“”, n}) \in \textit{CMech}, where \textit{m} = 2 \text{ represents the maximum number of users allowed to propose models.}

- \textit{mech}3 = \((\textit{request}, \textit{model1}, d) \in \textit{TMech}, where \textit{d} = 5 \text{ represents the maximum time units (minutes), within which a user has to request scores.}

- \textit{mech}4 = \((\textit{commit}, \textit{model1}, \textit{“After”}, \textit{notify}()) \in \textit{EMech}, where \textit{notify} is the task that must be executed after a user has executed \textit{commit} action.

- \textit{mech}5 = \((\textit{commit}, \textit{model1}, n) \in \textit{COMech}, where \textit{n} = 1 \text{ represents the number of users allowed to commit to \textit{model1} at the same time.}

Figure 8 shows the (part of) CP-net model for the enforcement mechanisms of the reference monitor needed to enforce the security and concurrency requirements and to ensure the correct behavior of the system. Finally, the correctness of the

7. Please note that this is just a hypothetical example to show the feasibility of the approach. We are not considering the management of models usage or session issues.

8. This construction can be automatized in case specific access control policy model is considered, however we make it manually as the purpose of this paper is only exploring the enforcement mechanisms.

Proof: Straightforward from the definition of binding element, enabling, occurrence and the code segmentation of CP-nets.
7.2. Analysis of Combined Mechanisms

We have used the CPN tools [1] for constructing the combined mechanisms CP-nets shown in Fig 8. In the initial marking, Init place contains two subject tokens and Models place contains five model tokens. This mechanism allows the simulation of CP-nets and the verification of properties. Standard properties like home, boundedness, liveness and fairness are verified automatically. However, it is also possible to specify additional properties, expressed in ML-like language, and use the state space analysis tools for the verification of these properties. For example, to verify the concurrency property that only two subjects can open a socket at the same time, we defined the ML query function

\[
\text{fn n > size(Mark:page 0 pro 1 n); 0; Int:max(9)}
\]

EntireGraph means that the entire state space is to be searched, fn n > size(Mark:page 0 open 1; n) means that the number of tokens in the open place is to be considered, and finally Int:max means the maximum number is to be returned. Evaluating this query after creating the full occurrence graph of the net gives us the expected value of 2. To verify the temporal property we can add one place between the request and the end transitions of sendpack action to represent the actual execution time for each subject. Then we check that for subject with execution time more than d = 5, timeout transition occurs and the execution is revoked. Due to space limit we omit further details.

8. Conclusions and Future Work

In this paper we establish a framework for building access control enforcement mechanisms based on Coloured Petri Nets. Our reference monitor can be seen as a recognizer of the functional behavior of the system and a controller of the actions that a subject is executing on resources. Five enforcement mechanisms are introduced including halting, temporal, cardinal, execution and concurrency mechanisms. We show also that our approach is sound by checking each mechanism separately against its enforced property and using CP-nets' state space tool for the combined mechanisms. This framework is the first step to bridge the gap between abstract policies and a reference monitor implementation. To achieve this goal our future work is twofold. First we are developing a usage control policy language based on Coloured Petri Nets. Second, we will explore how our CPN-based policies can be transformed to the state machine specification, i.e., based on Coloured Petri Nets.
automatically to configurable CPN-based reference monitor engine.

References


