Building a Stateful Reference Monitor with Coloured Petri Nets

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

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

Abstract

The need for collaboration and information sharing has been recently growing dramatically with the convergence of outsourcing and offshoring, the increasing need to cut costs through cooperative agreements between partners as well as competitors, and the rise in the demand for a high-quality healthcare from different healthcare actors. 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 collaboration and secure information sharing has been dramatically growing with the broadened spread of the social networks, the convergence of outsourcing and offshoring, the increasing need to cut costs through cooperative agreements between partners as well as competitors, and the rise in the demand for a high-quality healthcare from different healthcare actors [8]. 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 [2]. Outsourcing entails a delegation of (a part of) business function to an external company and share the needed resources, which requires tight collaboration and strict control. In order to increase healthcare quality and efficiency, different actors and institutions have to collaborate and share resource, EHR (Electronic Health Record) at the heart of them.

Problem Statement: In these modern collaborative and distributed environments new security requirements arises. (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 [16]. 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 [23] 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 functional behavior in proposed policy models and enforcement mechanisms.

Plenty of work have been dedicated to formalize and analyze security policies. However, fewer have paid 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 [12], [7], 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. [21], and lately by Pretschner et al. [18] and Janicke et al. [12]. However concurrency seems to have taken a back seat. Plenty of work have proposed enforcement architectures/models, for example, Zhang et al. [23] 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 [3]. 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-net or CPN) and security rules or properties are defined based on the CPN'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 CPN. 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. An overview of Coloured Petri Nets is presented in Section 2 and the motivating example is shown in Section 3. Basic concepts and main notations used in our stateful object-centric reference monitor are discussed in Section 4. In Section 5 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 6 by considering the motivating example. In Section 7 we discuss the related work and conclude the paper and discuss our future work in Section 8.

2. Colored Petri Nets Overview

A Colored Petri Net (CPN) [13] can be defined as a tuple *CPN = (Σ,P,T,A,N,C,G,E,I,SC)*, where Σ 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*. $A(x1, x2)$ is the set of arcs from x_1 to x_2, and the expression of (x_1, x_2) is $E(x_1, x_2) = \Sigma_{a \in A(x1,x2)} E(a)$.

$Var(t)$ 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 $(t, < v_1 = c_1, v_2 = c_2, ..., v_n = c_n >)$, where $v_1, v_2, ..., v_n \in Var(t)$ are the variables of t and $c_1, c_2, ..., c_n$ 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 $M_1[(t, b)\rangle$, then (t, b) may *occur* in M_1 yielding some marking M_2. This is written as $M_1[(t, b)\rangle M_2$. Accordingly, a *finite occurrence sequence* is a sequence consisting of a marking M_i an binding elements (t_i, b_i) denoted $M_1[(t_1, b_1)\rangle M_2 ... M_{n-1}[(t_{n-1}, b_{n-1})\rangle M_n$ and satisfying $M_i[(t_i, b_i)\rangle M_{i+1}$ 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. $[M_0\rangle$ denotes the set of reachable markings. Finally, The sets of all markings and steps is denoted by $\mathbb{M}$ and $\mathbb{Y}$, 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 from: $\sigma = M_1[Y_1\rangle M_2 \dots M_n[Y_n\rangle M_{n+1}$. 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 σ, respectively.

Finally, we introduce timed CP-net that 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.

3. Motivating Example

The following example illustrates the concepts of our reference monitor. We consider a collaborative modeling application between two automaker companies: $Comp_1$ and $Comp_2$. According to a cooperation agreement and in order to cut costs, the two competitors agree on purchasing together some of car components. The cooperation on purchasing components requires integrating some of financial and purchasing business processes by specifying an inter-organizational work flow between the financial and purchasing departments in $Comp_1$ and $Comp_2$. They decide to use a collaborative modeling application, similar to the one proposed by Rittgen in [19], [20], for the modeling procedure.

The collaborative modeling tool provides the following two features [1]: *proposal* and *score*. A proposal is a suggestion by a user for a revision of the current version of shared inter-organizational work flow model, which 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 this collaborative application: (1) *propose* to propose a proposal, (2) *request* for a user to request evaluation of a proposed proposal, (3) *commit* to commit a proposal to the model repository, (4) *update* to get the latest version of a model, (5) *score* to give a score for a proposal, and finally (6) *join* and *leave* a modeling (collaborative) session.

1. These features are inspired from the COMA tool: http://www.coma.nu/html/introduction.html

Requirements: Due to the sensitivity of the information contained in these models, some behavioral and security requirements must be enforced.
`BR1`: Committing is only allowed after a user proposes a proposal and requests evaluations.
`BR2`: After requesting evaluation, a user can withdraw a proposal.
`SR1`: Only users with role "Designer" are allowed to propose proposals.
`SR2`: A maximum number of ten proposals are allowed for the 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.

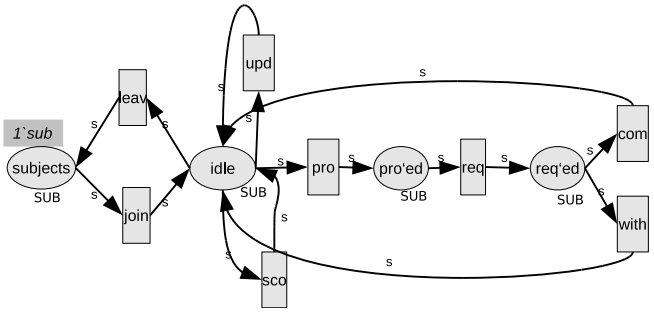

Figure 1: CPN of a functional behavior for collaborative modeling application (SUB is color set representing subjects, s is a free variable of type SUB, and sub is the initial marking of the place subjects).

However, authorization is not supported by the current version of the collaborative tool. Therefore, $Comp_1$ and $Comp_2$ decide to implement a security solution to enforce the needed requirements. For this reason, they need to utilize a policy model/specification that is able to cover the mentioned security requirements (this is out of scope of this paper, but is planned for future work) and an enforcement mechanism that is able to monitor and enforce these policy rules. Furthermore, they want to be able to analyze the mechanisms and ensure that these mechanisms enforce the defined policies. In Section 6 we show how we can build abstract enforcement mechanisms and analyze them.

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 subjects. The figure shows that *com* action is only allowed after the *req*, and *withdraw* action

is available after requesting evaluations.

4. A Stateful 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 CPN notion–a finite occurrence sequence OSF. Furthermore, a system contains a set of subjects *S* represented by token colour *SUB*, and set of objects *O* represented by token colour *OBJ*. Finally, we define a finite set of security related conditions *Cond* [2]. We distinguish three views of an access control system: the functional view, the policy view, and the enforcement view. Functional view represents the behavior of the system without any security consideration. Behavioral, security and concurrency requirements will be presented in the policy view. The enforcement mechanisms (enforcement semantics) of the access control policy is shown in the enforcement view.

Each action in the system, $act \in ACT$, is modeled as one transition, $t \in T$ in the policy view and is mapped into a sequence of Transition-Place-Transition, $(t, p, t) \in T \times P \times T$, 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. Thus, we define the following mapping functions:

• $actP : ACT \rightarrow T$: maps each action in the system to a transition in the policy view.

• $actRM : T \rightarrow T \times P \times T$: constructs the enforcement view of the action transition t. Each system action is represented as a set of two transitions and one place in the enforcement/reference monitor view. We call them $tRe, tEx,$ and tEn, representing the request transition, executing place, and the end transition, respectively. This constructor fulfills the following:

- $C(tEx) = SUB$: The color set of the execution place is of type SUB.
- $[\forall p \in \bullet t \Rightarrow p \in \bullet tRe] \wedge [\forall p \in t\bullet \Rightarrow p \in tEn\bullet] \wedge [\bullet tEx = \{tRe\}] \wedge [tEx\bullet = \{tEn\}]$

Figure 2 shows an action representation in the different views of the system. When the reference monitor intercepts an action $act \in Act$, which is mapped to the transition $t = actP(act)$ in the policy view, that a subject $s \in S$ is trying to execute, this request for execution is represented as a binding $(t1, b1)$, where the transition $t1 = actRM(t).reqT$

2. Please note that we do not restrict any specific model or language for the conditions to keep our model general. For example, constraint specifications languages [6], [4] can be used for this purpose.

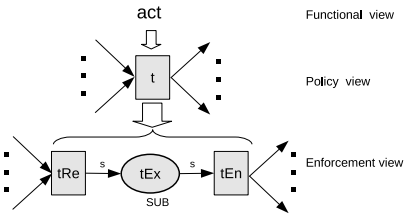

Figure 2: Action representation in the functional and the enforcement views.

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 CP-net reference monitor. The occurrence of this binding means that the action is allowed and it moves the subject token $sub = s$ to the executing place $actRM(t).exeP$. Finally, the occurrence of the binding $(t2, b2)$ in a marking $M2$ represents the end of the execution of the action act, where $t2 = actRM(t).endT$ represents the end transition of the action act in the enforcement view, and $b2$ is an enabled binding having the subject $s = b2(sub)$ as its subject element. From this discussion we can define the access request for executing an action as follows:

Access Request: the request transition of an action is enabled with a binding b that contains a token of the subject sub iff (i) the subject sub has requested that action, which is recognized by the defined bahavioral CPN, and (ii) the guard of the request transition evaluates to true for this binding b. Requests for single action is presented as binding elements in which the requesting subject is contained in the binding and the transition is the request transition of requested action.

Our reference monitor is stateful and is a recognizer of the correct behavior of the application, which is executed on behalf of users. That means the control of the reference monitor is continuous during the access session of resources. We define the access session as follows:

Definition 1. *An access session of a resource or a set of resources is defined as a finite occurrence sequence* $\sigma = M_0[Y1...Yn\rangle M_n$*, where* M_0 *is the initial marking of the access and the* M_n *is the finial marking where the access is finished.*

For example, the access session in the motivating example represents the sequence of actions that all participants perform on the shared resources (models) during the meeting period. It starts by initiating a collaborative session by an administrator and ends by closing this session. During the session periods, users can join and leave the session.

From the discussion above we can conclude that our reference monitor can be seen as (1) a recognizer of the

functional behavior of a 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, while the details are illustrated in next section.

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. [21], [22].
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.

5. 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. The development of a configurable reference monitor is a subject of our future work. 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 $Mech = (act, o, PAR)$, where $act \in Act$ is the triggering system action, $t = actP(act)$ is the transition representing act in the policy view, $o \in O$ is the protected resource, and 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.

5.1. $Mech_1$: Halting Mechanism

A halting mechanism can be defined as a 3-tuple $Mech_1 = (act, o, co) \in HMech$, where $co \in Cond$ is a condition that must be fulfilled instantly and $HMech$ 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 $co \in Cond$ must be fulfilled before a subject, represented by the token $sub \in SUB$, execute an action $act \in Act$ on a resource, represented by the token $obj \in OBJ$ as follows (where $t_1 = actRM(t).req$ and $sub = b(s)$):

$$\forall (t_1, b) \in BE.\ \forall M \in \mathbb{M} : (cond = false) \wedge sub = b(s) \Rightarrow \nexists M' \in \mathbb{M} : M[(t_1, b)\rangle M' \quad (1)$$

This rule is very intuitive and simply says: in any state of the reference monitor with marking M, if a condition $cond$ 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 act. 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.

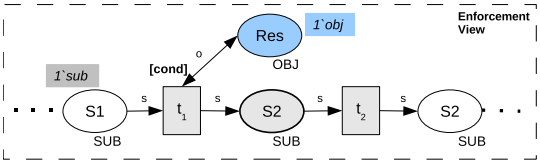

Figure 3: Halting mechanism.

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 t_1, 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. *$Mech_1$ enforces instant rules.*

Proof: The proof is straightforward consequence of the definition of step enabling and step occurrence of CP-net [13]. □

5.2. $Mech_2$: Temporal Mechanism

Temporal mechanism can be defined as a 3-tuple $Mech_2 = (act, o, d) \in TMech$, 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 $sub \in SUB$ is allowed to executed an action $act \in Act$, where t_1, t_2, and p are the request transition, end transition and executing place of the action act respectively, for the time period $d \in N$. Formally we define this temporal rule as follows (where $s = b(sub)$):

$$\forall (t_1, b) \in BE \ \ \forall M, M' \in \mathbb{M} : M[(t, b)\rangle M' \wedge sub \in b(s) \\ \Rightarrow \exists M'' \in \mathbb{M} \ \ \exists \sigma \in OSF : M'[\sigma\rangle M'' \wedge sub \notin M''(p) \quad (2)$$

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 $time(\sigma) \leq d$, then we get the definition

$$\forall (t_1, b) \in BE \ \ \forall M, M' \in \mathbb{M} : M[(t, b)\rangle M' \wedge s \in b(sub) \\ \Rightarrow \exists M'' \in \mathbb{M} \ \ \exists \sigma \in OSF : M'[\sigma\rangle M'' \wedge s \notin M''(p) \\ \wedge time(\sigma) \leq d \quad (3)$$

The end of the execution is denoted by the condition $sub \notin M''(p)$, 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 [14]. 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.

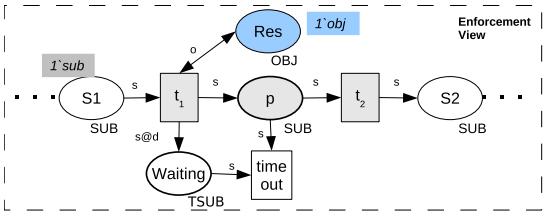

Figure 4: Timing mechanism.

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 state, indicating that the subject is executing the action. Furthermore, a subject token with a time stamp $@d$ is moved to the *waiting* place. 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. *$Mech_2$ enforces temporal rules.*

Proof: we want to proof that the temporal property for the action act holds for the $Mech_2$'s CP-net. We assume that a binding element (b,t) is enabled in a marking M where $b(s) = sub$ and $sub \in M(S1)$. The occurrence of this step gives us a new marking M' where the subject token is moved to the executing place p, i.e., $sub \in M'(p)$. Furthermore, firing the transition t_1 moves also a subject token to the place *Waiting* with a timestamp d, i.e., $sub@d \in M'(Waiting)$. From this marking M' we assume that there exists a finite occurrence sequence $\sigma \in OSF$. 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 t_2. 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 act within the duration d. In this case, after d time units the *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 $sub \notin M''(p)$. 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. □

5.3. $Mech_3$: Cardinal Mechanism

We define a cardinal mechanism as a 3-tuple $Mech_3 = (act, o, n) \in CMech$, where n is the number of times an action act can be used within one access session and $CMech$ is the set of all cardinal mechanisms. For this mechanism we define the cardinal property, which indicates that an action act can be used for only n times during each access session. Assuming an access session $\sigma = M_0[Y_1 Y_2 \dots Y_n\rangle M_n$, and the request transition of the action act is t_1, we define the cardinal property formally as follows:

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

This definition indicates that within one access session the request transition of the action 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 *UNIT* type containing n unit elements and connect this place to the request transition of the action act with one headed arc as shown in Fig. 5. The main element of the this CP-net is the *Card* place that contains a multi-set of UNIT color. The coefficient indicates how many times the action, whose request transition is t_1, is allowed to be executed.

Proposition 3. *$Mech_3$ 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

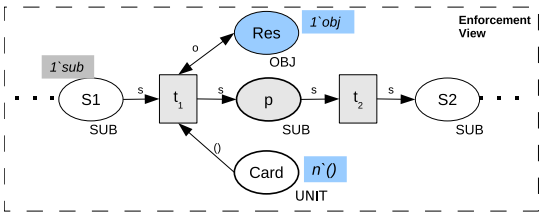

Figure 5: Cardinal mechanism.

access session. After n occurrences of the binding (t_1, b) the place *Card* contains no elements and thus the binding cannot be enabled. □

5.4. $Mech_4$: Execution Mechanism

An execution mechanism is a mechanism that executes additional actions/tasks when a subject uses a resource. To distinguish these actions that must be executed from the actions that a subject can execute on resources, we call the additionally executed actions *tasks*. They are of type ACT and are represented as part of the segmentation code. 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: $executed \in [Act \rightarrow Bool]$, 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 act, and the task $tsk \in ACT$ must be executed when the act is executed by a subject sub, then we define the three types of execution rules as follows:

$$\forall (t_1, b) \in EB\ \forall M, M' \in \mathbb{M} : sub \in b(s) \wedge M[(t, b)\rangle M' \Rightarrow executed(tsk) \quad (5)$$

$$\forall (t_2, b) \in EB\ \forall M, M' \in \mathbb{M} : s \in b(sub) \wedge M[(t_2, b)\rangle M' \Rightarrow executed(tsk) \quad (6)$$

$$\forall (t_1, b) \in EB\ \forall M, M' \in \mathbb{M} : s \in b(sub) \wedge M[(t, b)\rangle M' \Rightarrow \exists M'' \in \mathbb{M}\ \ \exists \sigma \in OSF : M'[\sigma\rangle M'' \wedge executed(tsk) \wedge time(\sigma) \leq d \quad (7)$$

Execution mechanisms can be defined as a tuple $Mech_4 = (act, o, type, task, d) \in EMech$, where $task$ is the task that must be executed , $type = After|Before$ indicates whether the task must be executed before or after the execution of the system action, d indicates the period of time after which the task must be executed, and finally $EMech$ is the set of all execution mechanisms. This mechanism can be constructed by adding the required task(s) in the segmentation code of the request transition or end transition of an action to execute the task before or after this resource action is executed. Or a timer can be set and the task can be executed when the deadline occurs. Figure 6

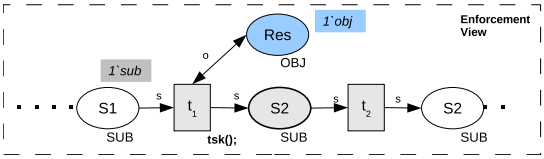

Figure 6: Enforcement action execution mechanism.

shows the CP-net representation of this mechanism for the first type and Firgure 7 for the third type.

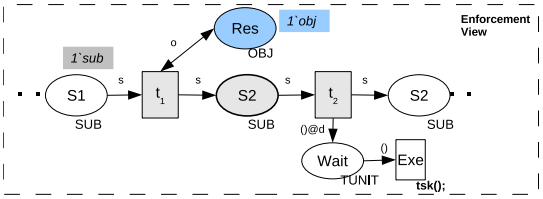

Figure 7: Enforcement action execution mechanism with delay.

Proposition 4. *$Mech_4$ enforces the first type of execution rules.*

Proof: Straightforward from the definition of binding element, enabling, occurrence and the code segmentation of CP-nets. □

5.5. $Mech_5$: Concurrency Mechanism

A basic concurrency mechanism can be defined as a 3-tuple $Mech_5 = (act, o, n) \in COMech$, where n is the number of subjects that are allowed to execute the action act in a truly concurrent way, and $COMech$ is the set of all concurrency mechanisms. More advanced and complex concurrency rules can be defined by specifying the set of users who are allowed to execute the action concurrently and another set of users whose access must be interleaved. Due to space limit in this paper we present the simplest way concurrency can be supported by CPN. The property that a concurrency mechanism enforces is called *concurrency property*. Assuming that the executing place of an action act is p and the initial marking of the access session is M_0, the concurrency rule can be defined as:

$$\forall M \in [M_0\rangle : |M(p)| \leq n \quad (8)$$

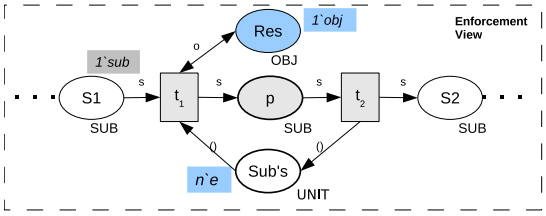

Figure 8: Concurrency mechanism.

The purpose of this mechanism is to control the subjects that are allowed to be located in the executing place of the protected action. Figure 8 shows how the reference monitor restricts subjects that are in the executing place p, i.e., subjects that are executing the action t concurrently, based on the coefficient of the tokens e or () stated in the *Sub's* place.

Proposition 5. *$Mech_5$ enforces concurrency rules.*

Proof: We want to proof that at any reachable marking the number of tokens in the executing place p is less or equal n. It can be noticed that $\bullet p = \{t_1\}$ and $p\bullet = \{t_2\}$, which means that tokens can only be added to p by the occurrence of the transition t_1 and tokens are withdraws from p by the occurrence of t_2. Furthermore, $\bullet Sub's = \{t_2\}$ and $Sub's\bullet = \{t_1\}$, which means that tokens are added to the place *Sub's* by the occurrence of the transition t_2 and are withdrawn by the occurrence of the transition t_1. Based on this observation we conclude that $\forall M \in [M_0\rangle : |M(p)| + |M(Sub's)| = Const$, where $Const$ is the number of token in both places at the initial marking. However, $|M_0(p)| = 0$ and $|M_0(Sub's)| = n$, thus $\forall M \in [M_0\rangle : |M(p)| \leq n$. □

5.6. Combined Mechanisms

After considering each mechanism separately, the correctness must be preserved when different mechanism are combined for a specific action. It can be noted that execution mechanisms has no effect on the access decisions, i.e., the flow of subject tokens, and does not cause any violation problems if it co-exists with other mechanism. Thus we will consider the case of an action that applies all, but execution, mechanisms and check the affect of applying more than one mechanism to the correctness of each.

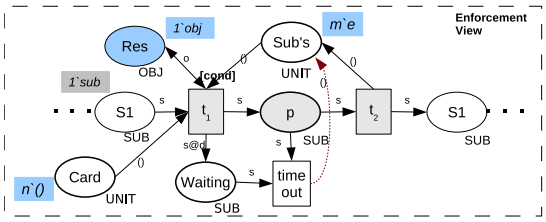

Figure 9: Combination of Mechanisms.

Figure 9 shows the combination of the four discussed enforcement mechanisms applied on an action act, whose enforcement view is the tuple (t_1, p, t_2). First, the request transition can not occur if the condition of the halting property (Eq. 1) is false. Thus it can be easily concluded that the this property holds for the combined mechanisms. Second, it can be seen that the request transition can only occur for a maximum number of n times during any occurrence sequence, i.e., access session, and lead to the cardinal property, Eq. 4. Furthermore, the temporal property can also be proved similar to the proof of proposition 2. Concerning the concurrency property, its proof assumes that $p\bullet = \{t_2\}$, however, the temporal mechanism adds an arc from the place p to the new $timeout$ transition. This violate the assumption, thus the concurrency property does not hold. Each occurrence of the action $timeout$ will cause the $UNIT$ tokens of the $Sub's$ place to be decreased by one. To solve this problem we add one arc from the $timeout$ transition to the $Sub's$ place (the dashed shadowed arc in Figure 9). With this new arc the concurrency property can be proved similar to the proof of proposition 5.

From the discussion above we conclude that when temporal and concurrency mechanism co-exists for an action, an arc between the $timeout$ transition, of temporal mechanism, and the $Sub's$ place, of concurrency mechanism, must be added.

6. Building Reference Monitor Mechanisms

In this section we show the feasibility of our approach by building the mechanisms needed for the motivating example mentioned in Section 3. The example shows the need to protect the usage of a collaborative application in order to facilitate the usability of such applications in the real business scenarios. It can be seen that: in order to enforce requirements `SR1-SR5` we need the following mechanisms: instant, cardinal, temporal, execution, and concurrency mechanisms, respectively. While the behavioral requirements `BR1, BR2` are represented by the behavioral CPN. (This is just a hypothetical example to show the feasibility of the approach.)

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 $propose \in Act$ action is represented by the transition $pro \in T$ in the system view and the $(p_1, pro, p_2) \in T \times P \times T$ 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 [3].

3. This construction can be automatized in case specific access control policy model is considered

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

- $mech1 = (propose, model1, [user.role = \text{“}Designer''])$ $\in HMech$, assuming that the user is trying to propose the model $model1$ and the attribute $user.role$ indicates the role of the user.
- $mech2 = (propose, \text{“}\_'', n) \in CMech$, where $m = 2$ represents the maximum number of users allowed to propose models.
- $mech3 = (request, model1, d) \in TMech$, where $d = 5$ represents the maximum time units (minutes), within which a user has to request scores.
- $mech4 = (commit, model1, \text{“}After'', notify(), 0) \in EMech$, where $notify$ is the task that must be executed after a user has executed $commit$ action.
- $mech5 = (commit, model1, n) \in COMech$, where $n = 1$ represents the number of users allowed to commit to model1 at the same time.

Figure 10 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.

6.1. Analysis

We have used the CPN tools [1] for constructing the mechanisms' CP-net shown in Fig 10. In the initial marking, *Init* place contains five subject tokens and *Models* place contains two models, while *Subs* place contains $n = 2$ unit tokens, and *Card* place contains $m = 10$ unit tokens. This tool allows the simulation of CP-net and the verification of properties. Standard properties like *home, boundness, liveness and fairness* are verified automatically. However, it is also possible to specify additional property, 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 one user can open a commit at the same time, we defined the ML query function: $SearchNodes(EntireGraph, fn\_ => true, noLimit, fn\ n => size(Mark.page1'com\ 1\ n)\ ,\ 0,\ Int.max)$. *EntireGraph* means that the entire state space is to be searched, $fn\ n => size(Mark.page1'com1, n)$ means that the number of tokens in the *com* 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 complete CP-net of the motivating example gives us the expected value of 2.

7. 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 [21], [22], followed by [11], [9], [10], and finally the *edited automata* developed by Ligatti et al. [17]. In this line of work, a reference monitor basically aims at monitoring a program's behavior to avoid entrusted 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 entrusted 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 in distributed and collaborative environments. Systems in our case can be composed by different applications and users, with respect to operations related to the protected resources. Pretschner et al. [18] and Janicke et al. [12] 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 [7] 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 petri nets to authorization in workflows, most notably [5], [15], among 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 are not considered.

8. Conclusions and Future Work

In this paper we establish a framework for building access control enforcement mechanisms for a stateful reference monitor 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 against its enforced property separately or when combined with other mechanisms. Furthermore, CP-net's state space tool can be used for analysis and verification. This framework is the first step to bridge the gap between abstract policies and a reference monitor implementation. To achieve this

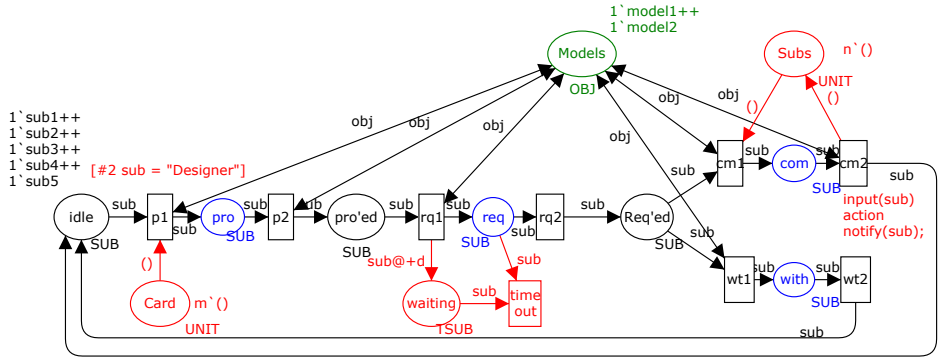

Figure 10: Reference monitor enforcement mechanisms for the motivating example.

goal our future work is twofold. First we are developing a usage control policy language based on the same theoretical foundation of the reference monitor specification, i.e., based on Coloured Petri Nets. Second, we will explore how our CPN based policies can be transformed automatically to configurable CPN-based reference monitor engine.

References

[1] Cpn tools, http://wiki.daimi.au.dk/cpntools.

[2] http://online.wsj.com/article/bt-co-20090703-703278.html.

[3] Oasis extensible access control markup language (xacml), http://www.oasis-open.org/committees/xacml.

[4] Gail-Joon Ahn and Ravi Sandhu. Role-based authorization constraints specification. *ACM Trans. Inf. Syst. Secur.*, 3(4):207–226, 2000.

[5] Vijayalakshmi Atluri and Wei kuang Huang. An authorization model for workflows. In *In Proceedings of the 4th European Symposium on Research in Computer Security*, pages 44–64. Springer-Verlag, 1996.

[6] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security*, 2(1):65–104, 1999.

[7] Nataliia Bielova and Fabio Massacci. Do you really mean what you actually enforced? edit automata revisited. Technical report, Ingegneria e Scienza dell'Informazione, University of Trento., 2008.

[8] YB Cheung, SB Tan, and KS Khoo. The need for collaboration between clinicians and statisticians: some experience and examples. *Annals of the Academy of Medicine, Singapore*, 30(5):552, 2001.

[9] David E. Evans. *Policy-directed code safety*. PhD thesis, 2000. Supervisor-John V. Guttag.

[10] Philip W. L. Fong. Access control by tracking shallow execution history. *sp*, 00:43, 2004.

[11] Kevin Hamlen. *Security policy enforcement by automated program-rewriting*. PhD thesis, Ithaca, NY, USA, 2006. Adviser-Morrisett,, Greg.

[12] Helge Janicke, Cau Antonio, Siewe Francois, and Zedan Hussein. Deriving enforcement mechanisms from policies. 2007.

[13] K. Jensen. *Coloured Petri Nets*, volume 1. Springer-Verlag, 1992.

[14] K. Jensen. *Coloured Petri Nets*, volume 2. Springer-Verlag, 1997.

[15] K. Knorr. Dynamic access control through petri net workflows. In *ACSAC '00*, page 159, Washington, DC, USA, 2000. IEEE Computer Society.

[16] Ram Krishnan, Ravi Sandhu, Jianwei Niu, and William H. Winsborough. Foundations for group-centric secure information sharing models. In *SACMAT '09*, pages 115–124, New York, NY, USA, 2009. ACM.

[17] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3):1–41, 2009.

[18] A. Pretschner, M. Hilty, D. Basin, C. Schaefer, and T. Walter. Mechanisms for usage control. In *ASIACCS '08*, pages 240–244, New York, NY, USA, 2008. ACM.

[19] Peter Rittgen. Collaborative modeling - a design science approach. *Hawaii International Conference on System Sciences*, 0:1–10, 2009.

[20] P.R08 Rittgen. COMA: A Tool for Collaborative Modeling. In *CAiSE'08 Forum*, page 61.

[21] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[22] Úlfar Erlingsson and Fred B. Schneider. Irm enforcement of java stack inspection. In *S&P '00*, page 246, Washington, DC, USA, 2000. IEEE Computer Society.

[23] Xinwen Zhang, Masayuki Nakae, Michael J. Covington, and Ravi Sandhu. Toward a usage-based security framework for collaborative computing systems. *ACM Trans. Inf. Syst. Secur.*, 11(1):1–36, 2008.