

Using Petri Nets to Detect Access Control Violations in a System of Systems

A. Walvekar, M. Smith, M. Kelkar, R. Gamble

Software Engineering and Architecture Team

University of Tulsa, 800 S. Tucker Dr., Tulsa, OK 74104

Technical Report SEAT-UTULSA-09-12

Abstract

System of systems are comprised of multiple, interacting components that may have independent access control policies. Each component in the system of systems has its own access control domain. Certain interactions among the software components within the system of systems imply the need to configure of inter-domain mappings of access privileges. These mappings require formal, scalable scrutiny to uncover potential violations of confidentiality and availability within a single component. In this paper, we introduce the Conflict Petri Net (ConPN) to analyze inter-domain access control mappings for security violations in system of systems. We show formally that ConPN finds all potential conflicts related to role inheritance, Separation of Duty (SoD), cardinality, and temporal restrictions in Role-Based Access Control (RBAC). We generate the ConPN graphically, analyze the policy in motion, and present the conflict outcomes visually and textually.

Introduction

Due to constantly changing demands on today's systems, using component-based software systems is a popular development strategy for large applications. Legacy and stovepipe systems, commercial-off-the-shelf (COTS) products, and other third-party software created independently to interact with one another, are now being forced to work together reliably and securely as *components* of a system of systems.

Many integrated applications operate within secure environments where access control must be efficient, complete, and correct. Access control policies are defined as a set of individual rules (functions providing privileges) applied to requests from subjects (users) to perform certain actions on a particular set of objects that require a particular access right [1], [2]. Access is granted if the rule evaluation provides privileges for the access requested by the subject. Each component system has its individual access control policy described in terms of hierarchical, separation of duty (SoD), cardinality, and/or time assignments and constraints. The role hierarchy defines seniority among the roles, while SoD constraints restrict access to mutually exclusive operations. Cardinality constraints add numerical restrictions to allowable accesses to a system, and timing, or temporal, constraints define access over a given time interval.

Vulnerabilities related to access control have been defined and organized into different categories that facilitate their detection and resolution [1]. We assume secure components have a domain in which there are no conflicts among their policies. However, when components become part of a larger System of systems, individual component security is not enough to determine if the whole system is secure. Security vulnerabilities present themselves as policy conflicts or violations that occur due to *inter-domain mappings*, the access control mappings between local component system domains. Inter-domain access is more specific to the integrated systems where subjects from one component domain try to access objects from another component domain. Inter-domain

mappings can also define further restrictions on inter-domain access such as user-role assignments, SoD, role hierarchy, cardinality, and time [3],[4],[5]. With the inter-domain mapping in place, it is possible for access to remain undecided from the integration and for new access decisions to disagree with locally defined policies.

In this paper, we define the Conflict Petri Net (ConPN) to analyze inter-domain access control mappings for System of systems applications and evaluate their potential to introduce violations to individual component security policies. We show formally how ConPN denotes inter-domain policy violations for Role-Based Access Control (RBAC) systems. Specifically, ConPN examines role inheritance, SoD, cardinality, and temporal policy constraints for compliance. We indicate the violations using their formal definition within ConPN. Our solution allows system integrators to manage access control policies among communicating components to ensure that violations will not emerge due to policy constraints once the system is in production.

Background

This section discusses the foundation for the ConPN, describing security policy conflict manifestations and Role Based Access Control. Additionally, we discuss the foundations of Petri Nets and the Colored Petri Net extension on which the Conflict Petri Net definition is based.

Security Policy Conflict Manifestations

For an integrated system composed of multiple components, security characterization must happen at three different levels [7]:

1. Within a single isolated component – its domain policy
2. Between locally interacting components – the inter-domain policy
3. Between component interactions and the global System of systems policy – large-scale compliance policies

Security is violated if access control to each component is considered separately, ignoring interrelationships among components [3],[8].

To detect inter-domain conflicts that cause security vulnerabilities, our previous research describes a process called DIPC [9]. DIPC accumulates component-related access control properties from established sources and then categorizes the properties toward developing a security profile for individual component policies. The profile is then used to assess conflicts generated due to inter-domain property mismatches. Three main categories form a security profile: authorization data, access control and access policy combining. Security properties are listed for each category and have distinct values that may contradict each other leading to interoperability, security, or discrepancy types of conflicts.

Comparing components based on how they are built (property definitions) is no longer sufficient as they should be evaluated based on how they act together (property interactions). The more predominant cause for security conflicts is not in the value-mismatches for a property but in the inaccurate mappings between components that contradict local property definitions. Thus, conflict evaluation is more of a “runtime” or dynamic process than static property comparison.

Role Based Access Control

Role-Based Access Control (RBAC) is commonly used to define access parameters within components and combinations thereof. This is done by creating rule-sets of permissions, assigning the rules to roles, and then assigning roles to users [6]. The robust, low-maintenance, and efficient nature of RBAC systems allow for simple modeling of many constraints including hierarchical, SoD, cardinality, and temporal [10]. RBAC systems have noted limitations that

should be addressed in integrated system security analysis [11], [3]. Because each user of a system takes on an assigned role, roles should be defined based on how the organization works. Inheritance may be ambiguous when it does not correspond to an organization's hierarchy [11]. Context is not included in role assignment constraints, which can be eased if traceable origins of inherited access are maintained across domains of the integrated system.

A role hierarchy is a partial order relationship established among roles, through which access is granted. SoD constraints define mutually exclusive relations between two entities. Each individual is authorized or not authorized to have access based on the role he or she has been assigned, allowing a system to restrict access to authorized users and manage those permissions associated with groups of users easily by mapping users to roles. Other constraints can be used to restrict access to no more than a specified number of users (cardinality constraints) or to be granted during specific times (temporal constraints).

We graphically represent RBAC systems using the convention where users and roles are nodes in the graph and the connections between them are directed edges, called mappings. The arrow connecting a user to a role represents assignment and the arrow connecting a role to a role represents a hierarchy. Figure 1 illustrates this concept. Role r_1 inherits the permissions given to r_2 . Due to the temporal constraint on the mapping between r_1 and r_2 , r_1 only inherits from r_2 on Monday, Tuesday, and Wednesday. Because r_2 inherits privileges from r_3 , r_1 also indirectly inherits permissions from r_3 . Role r_3 does not inherit from any other roles. User 1 is given all permissions defined for roles r_2 and r_3 because of the user-role assignment to r_2 and the role hierarchy between r_2 and r_3 . The cardinality constraint on r_3 indicates that only one user is allowed to have r_3 's privileges at a time. This means that if another user is assigned to any of the other roles, due to inheritance, this constraint would be violated. The mapping between roles r_3 and r_4 is considered an inter-domain mapping because it spans from one domain to another.

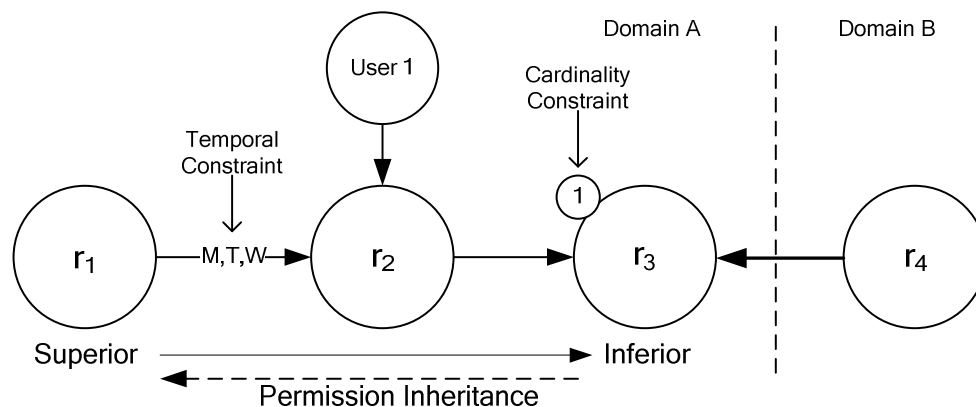


Figure 1. Role Hierarchy and Permission Inheritance

Most RBAC technologies only consider single-domain systems. Few focus their efforts on the inter-domain mappings. Many software approaches to single-domain policy control exist, but none seem to detect inter-domain conflicts at the design phase [12]. Some approaches using XML or UML can detect conflicts, but the process of conflict detection is not automated [13, 14]. In this case, reoccurrence of a conflict template in the UML system representation denotes conflict. So, the UML architecture diagram needs to be manually scanned to detect the conflict [15]. The most similar research to our effort lies within [3]. The administrative model shows different conflicts and examples of cross-domain problems. Though the conflict resolution technique is automated using formal language implementation, conflict detection is a manual process and is done by visually scanning directed sub-graphs. Directed graphs have limitations in modeling dynamic systems. Hence, conflicts occurring during the execution of systems are harder to detect. A visual scan of directed sub-graphs is not guaranteed to find all of the conflicts.

Petri Nets

Petri Net design and analysis have been extensively researched and many various algorithms, techniques, and tools exist to aid this process [16]. Petri Nets describe systems at various levels of abstraction, and when combined with the ability to represent hierarchies, modeling complex systems becomes much easier [17]. Petri Nets are well suited for modeling concurrent, distributed, asynchronous, non-deterministic, and parallel systems [6]. Petri Nets are bipartite directed graphs, making it easy for modeling and formal verification. The theory behind Petri Nets allows flexibility to extend existing models once they conform to the basic Petri Net constraints. Moreover, they can capture both static and dynamic aspects of a system, which is not possible in other techniques like graph-based models.

A Petri Net is a graph, $G_{PN} = (V, E)$, where the set, V , of vertices is comprised of *places* and *transitions* and E is the set of edges, or *arcs* between them. A place is never connected to another place directly, and transitions are never connected to another transition directly. Places are static entities. Transitions represent dynamic entities because the transition firing rules can change the contents of the tokens that flow through the Petri Net. A Petri Net is a 3-tuple:

$PN = (P, T, F)$ such that

P : Set of *places*, $P \subseteq V$

T : Set of *transitions*, $T \subseteq V$, $P \cap T = \emptyset$

F : Flow relation for *arcs*, $F = (P \times T) \cup (T \times P)$, $F \subseteq E$

We define the following specific Petri Net entities

- *Input Arc*: Flow f represents an input arc for transition tr when $f = (p, tr)$ such that $f \in F$, $p \in P$, and $tr \in T$ because it flows from a place into a transition.
- *Output Arc*: Flow f represents an output arc for transition tr when $f = (tr, p)$ such that $f \in F$, $p \in P$, and $tr \in T$ because it flows from a transition to a place.
- *Input Place*: $p \in P$ is an input place when it is connected to a transition $tr \in T$ through an input arc.
- *Output Place*: $p \in P$ is an output place when it is connected to a transition $tr \in T$ through an output arc.

There can be multiple input and output places for a given transition, which then forms a set of input places and a set of output places. In the most basic transition, tokens flow from all input places into all output places, even if the number of input and output places differs. Graphically, a place is represented by a circle; a transition by a rectangle or a bar; and an arc with an arrow (Figure 2).

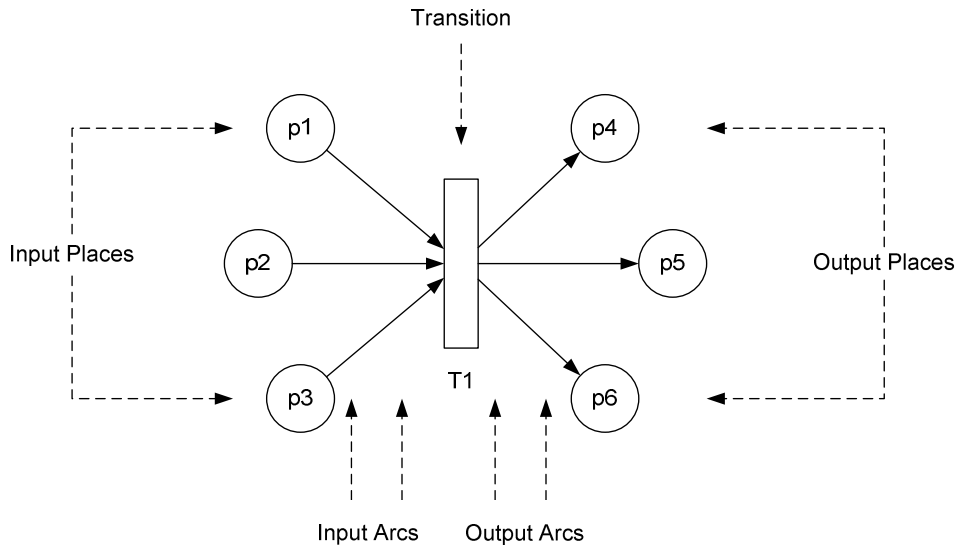


Figure 2. Basic Petri Net

- *Token*: A token is the entity that flows within an executing Petri Net, represented by small dots inside a place.
- *Enabled Transition*: A transition becomes enabled when all its input places have at least one token.
- *Fired Transition*: An enabled transition is fired (Figure 3) upon removing the tokens from the input place and placing them into the output places according to the firing rules.

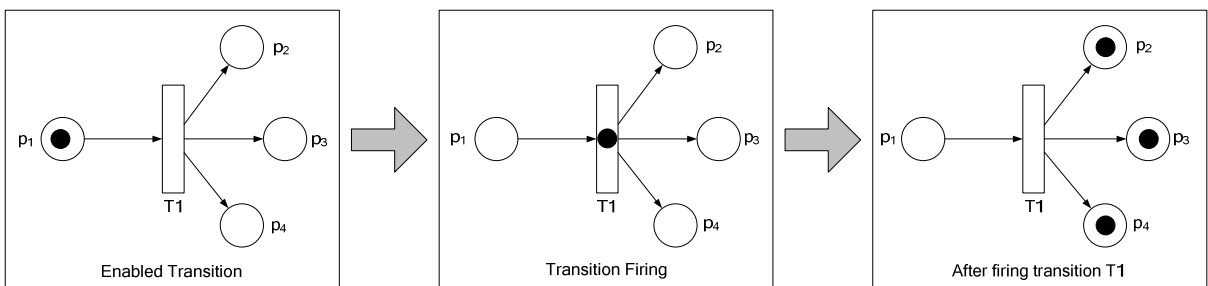


Figure 3. Transition Firing

Executing a Petri Net is moving a set of tokens through the graph via transition firing rules. A Petri Net *executes* while it has enabled transitions that can be fired. This is known as a *liveness* property of a Petri Net. When a Petri Net reaches a state where no transition can be fired, it is known to be *dead*.

The basic Petri Net has been extended in various ways including Colored Petri Nets [6], Stochastic Petri Nets [18], Timed Petri Nets [19], and Hierarchical Petri Nets [16]. These modifications include tokens with information, arcs carrying expressions, time-driven firing, and other higher-level abstractions. The ConPN is built on the foundation of Colored Petri Nets.

Colored Petri Nets allow distinguishable tokens by assigning a particular color to a token [6]. They also introduce the concept of arc-expressions, binding variables, and guards. These improvements dramatically increase the overall expressive power of Petri Nets by bringing them

closer to programming languages. Various industrial applications, theoretical problems, and dynamic processes can be modeled using Colored Petri Nets [6].

In Figure 4 we can see how a *color set* is defined. In our ConPN, we use this concept to place necessary information within tokens so that conflicts can be found. Here, place P1 can hold red or green tokens. The arc between P1 and T1 allows only green tokens and the arc between P1 and T2 allows only red tokens. In this case, the token color represents a data type, and each place, transition, or arc can place requirements as to what color/type of token can exist on it. Place P2 does not put any restriction on what color the tokens need to be, and the variable X can be either red or green. The transition T3 has a guard expression and is known as a *guard transition*. Depending upon the guard evaluation, the transition will fire or not fire. This means that if X is red, then T3 will fire, putting the token into place P3. Similarly, arcs also can have arc expressions whose functionality is the same as guard expressions.

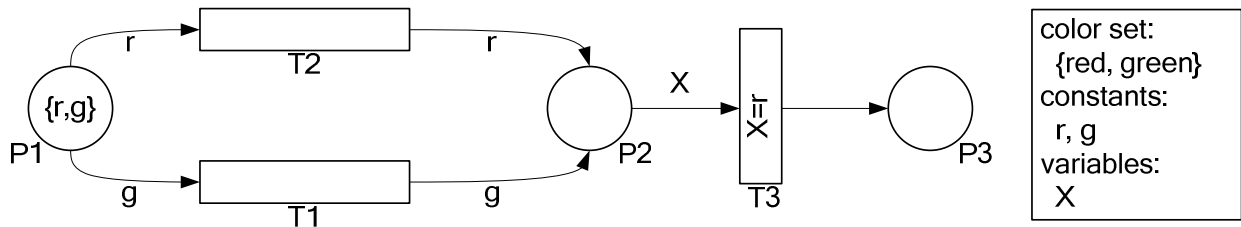


Figure 4. Colored Petri Net

One modified Colored Petri Net model represents multimedia documents and its related security requirements [20]. This approach proposes a time augmented Colored Petri Net that allows specification of a multilevel security policy for multimedia documents. Multiple security policies and protection schemes are also supported by this Petri Net model. Their Colored-GOCPN model shows how Petri Nets can be modified to represent locked and unlocked tokens, weighted arcs, special transition firing rules, and special states. Our modifications follow a similar approach to codify the requirements for conflict representation in the ConPN.

Petri Nets have appeared in various security-related research areas. Access control, security models, secure software architectures, and overall system security can be modeled and studied using Petri Nets. Mandatory access control policies are modeled using Colored Petri Nets, supporting the Military Security model [21]. A dynamic workflow access control mechanism is implemented using Petri Nets in [22].

The most recent works in using Petri Nets to model access control policies can be found in [23], [24], and [25]. To model the Biba Strict Integrity policy in a Petri Net, Zhang et al., modeled objects as places and subjects as transition [23]. An input arc meant a subject could read an object, while an output arc meant an object could be written by a subject. Overlaying the integrity levels of subjects and objects onto the Petri Net resulted in its basic organization. Examining execution of the Petri Net would only yield failure if tokens could not get from a start place to an end place. A secondary representation, called a coverability graph that denoted the access values of each place within a single tuple and their allowed associations from the integrity levels to show where access control broke down. The actual execution of the Petri Net then became secondary to the examination of coverability. The same coverability graph is used to demonstrate Chinese Wall conflicts using a Colored Petri Net. Similar read/write traversal rules are used. But it is the coverability graph that indicates whether a path exists through the Petri Net or not. Again, the Petri Net would simply not allow the proper tokens to reach an end state [24].

Self-authentication RBAC is modeled using a colored Petri Net in [25]. A different approach is used in which the users and roles are tokens that pass through assignment and de-assignment

transitions. Places are equipped with decision-based functionality. For example, the Self-Authentication Decision making place stores the token of color Self-Authentication. “The token $\langle u, r, s, a \rangle$ represents: If user u , which is authorized with role r , wants to make the ACCESS a in session s , the self authentication mechanism will be activated to generate a decision result: allowing or refusing.” Transition activity is a complex manipulation of the users and their roles. The goal is to guarantee consistency in each state of the Petri Net that is reachable from the initial state pointing to the necessity of an occurrence graph for analysis, which is not shown because of the exceedingly large state space from even a small policy.

Inheritance Policy and Conflicts

As described earlier, we focus on policies in which access is defined in terms of user-role mappings that specify a set of security constraints to restrict role access for a set of users. Two broad categories of violations exist: inheritance and conflict of interest. We express a security policy based on privilege inheritance in RBAC systems. Often a security policy document is expressed in XML using the RBAC standard [26]. However, for this paper, we take a format-neutral approach so that converting the policy into the ConPN is doable.

We define an inheritance policy as a 4-tuple, $InheritancePolicy(U, R, SoD, M)$ such that

- U : Finite set of Users
- R : Finite set of Roles
- SoD : Finite set of role based SoD requirements expressed as a triple (u, r_1, r_2) where $u \in U$ and $\{r_1, r_2\} \subseteq R$
- M : Finite set of user-to-role and role-to-role mappings (or assignments) with cardinality constraints expressed as triples (u, r_1, n) or (r_1, r_2, n) where $u \in U$, $\{r_1, r_2\} \subseteq R$, and $n \in \mathbb{N}$.

A role in R can have a restricted cardinality or may allow infinitely many users in U to have access. A SoD requires at least two roles in R to indicate that the same user in U cannot be involved in both at the same time. If a mapping in M has a temporal constraint, then the inheritance of a role is restricted to a certain time interval as expressed by a natural number. If there is no temporal constraint, then $n = \infty$.

Let IP_A and IP_B be the inheritance policies of domains A and B, respectively, which may represent independent components. Let IP_{join} be the inter-domain mappings that tie the access policies together.

Let $IP = IP_A \cup IP_B \cup IP_{join}$ be an inheritance policy. An *inheritance conflict* exists when a role inherits permissions it should not be allowed to have. This generally occurs with an incorrect role hierarchy between superior and inferior roles

Conflicts of interest exist when entities (users or roles) should not be instantiated at the same time and are either directly or indirectly allowed to do so because of faulty inter-domain mappings. Conflicts of interest are often forcibly prevented by including a SoD constraint on roles. If SoD lines are incorrectly mapped, when two domains are combined into one system, a role may be accessed by two conflicting users at the same time [12].

Cardinality and temporal constraints extend the conflict of interest constraints with time and assignment number restrictions. Cardinality constraints assign an upper limit to the number of users assigned to a role at one time. Temporal constraints assign certain time-periods for which a mapping is valid. A temporal conflict arises if a particular user can be assigned to a specific role that does not have equivalent temporal units. If this is the case, then either the user can access the role at a time that is not permitted or the role is incorrectly unavailable to the user.

Motivating Example

We have augmented an example initially published in [3] to demonstrate the conflicts defined above and how they are depicted in ConPN. **Figure** depicts access control policies of Domain A, Domain B, and the inter-domain mapping between A and B.

Thus, $IP = IP_A \cup IP_B \cup IP_{join}$ where

$IP_A = (U_A, R_A, SoD_A, M_A)$, such that

$U_A = \{u_1, u_2, u_3\}$,

$R_A = \{r_{1A}, r_{2A}, r_{3A}, r_{4A}\}$,

$SoD_A = \{(u_1, r_{1A}, r_{2A})\}$,

$M_A = \{(u_1, r_{1A}, \infty), (u_1, r_{2A}, \infty), (u_2, r_{2A}, \infty), (u_3, r_{3A}, \infty), (r_{1A}, r_{4A}, \infty), (r_{4A}, r_{3A}, \infty)\}$

$IP_B = (U_B, R_B, SoD_B, M_B)$, such that

$U_B = \{u_4, u_5\}$,

$R_B = \{r_{1B}, r_{2B}\}$,

$SoD_B = \{\}$,

$M_B = \{(u_4, r_{1B}, \infty), (u_5, r_{2B}, \infty), (r_{1B}, r_{2B}, M-Th)\}$

$IP_{join} = (U_{join}, R_{join}, SoD_{join}, M_{join})$, such that

$U_{join} = \{\}$,

$R_{join} = \{\}$,

$SoD_{join} = \{\}$,

$M_{join} = \{(r_{1B}, r_{2A}, \infty), (r_{1A}, r_{1B}, \infty), (r_{1B}, r_{4A}, F), (r_{2B}, r_{4A}, W-Th), (r_{3A}, r_{2B}, \infty)\}$

The inter-domain mapping, denoted by M_{join} , causes all five types of conflicts to occur. We show how ConPN can detect the potential for these conflicts. Using a Petri Net allows us to separate concerns among conflict types and form our model atop a commonly accepted formal technique. This process is an improvement over manually scanning graphs or XML documents to find conflicts, such as in [3] where only conflict resolution is automated.

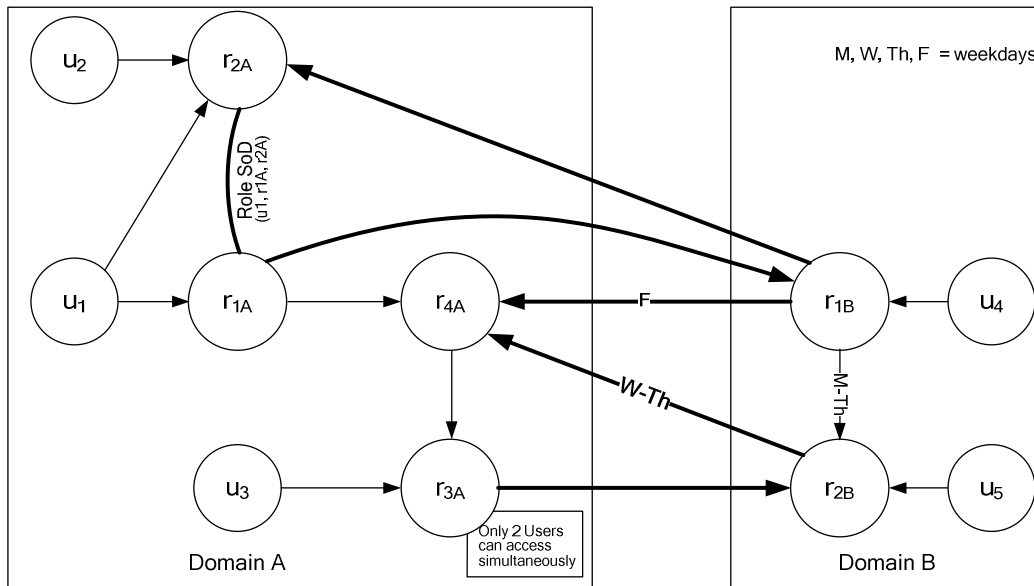


Figure 5. Motivating Example

Access control conflicts can allow the most unqualified user access to the most sensitive information if even one mapping is incorrectly specified. Depending on the sensitivity of the system, this could allow information leaks that affect a company's survival against competition or it could threaten national security by allowing attackers into sensitive government systems. The

goal of ConPN is to guarantee that all policy conflicts are found, which is the only way to definitively say that a system's access control is secure. Thus, the approach will err on the side of false positives, rather than missing any potential conflict. Realizing the consequences of improper access, the operation of ConPN is designed to monitor a user's access path to indicate what roles a user has been given access to. This eases conflict resolution as it helps isolate where in the Inheritance Policy faulty mappings have occurred.

Constructing the Conflict Petri Net

We build upon the foundation of Colored Petri Nets to create the Conflict Petri Net, *ConPN*. ConPN requires extensions to the definitions of places, tokens, and arcs without interfering with the basic rules of execution and analysis of the Colored Petri Net. We formally define the transition rules that underlying the perspective-based, conflict detection mechanism. The structure of ConPN represents role-based access control policies easily and completely.

ConPN is a graph, $D = (P, Arc)$, where vertices are comprised of start places, role places, and choice places and edges are comprised of input and output arcs. Tokens show the policy in motion by flowing from place to place using transitions and firing rules to traverse the arcs. ConPN retains meta-data based on token movement and the roles visited to determine if a state can be reached that indicates a policy violation.

Places

A place is defined as follows:

Place = (*ID*, *CurrTk*, *TkLog*) such that
ID: Unique place identifier
CurrTk: Current token set at place
TkLog: Bag of tokens that have visited the place, can be null

The set of all places, P , in ConPN is partitioned into Normal and Choice places. Normal places are further partitioned into Start places (SP) and Role places (RP). The set of Choice places (CP) simulate the controlled access to certain roles for the specified user as dictated by SoD requirements. Choice places differ from start and role places because they embody these essential policy constraints. Thus,

$$SP \subseteq P \wedge RP \subseteq P \wedge CP \subseteq P \wedge (SP \cap RP \cap CP = \emptyset)$$

All users have a 1:1 mapping with Start places in ConPN, signifying user access to the system. All roles in IP have a 1:1 mapping with Role places. SoD constraint triples have a 1:1 mapping with Choice places. For example, the inheritance policy represented by Figure 5 translates into five Start places with IDs $\{u_1, u_2, u_3, u_4, u_5\}$, seven Role places with IDs $\{r_{1A}, r_{2A}, r_{3A}, r_{4A}, r_{1B}, r_{2B}\}$, and one Choice place, $\{rsod_1\}$.

The cardinality of each place in P is determined by the function $C: P \rightarrow \mathbb{N}$, which maps a place to a natural number indicating the maximum number of tokens allowed to flow to that place. If no cardinality restriction for some place $p \in P$ exists, then $C(p) = \infty$. In our example, only one role, r_{3A} , in Figure 5 has a cardinality restriction, such that $C(r_{3A}) = 2$.

Figure 6 shows the organization of the places in ConPN that correspond to the example in Figure 5.

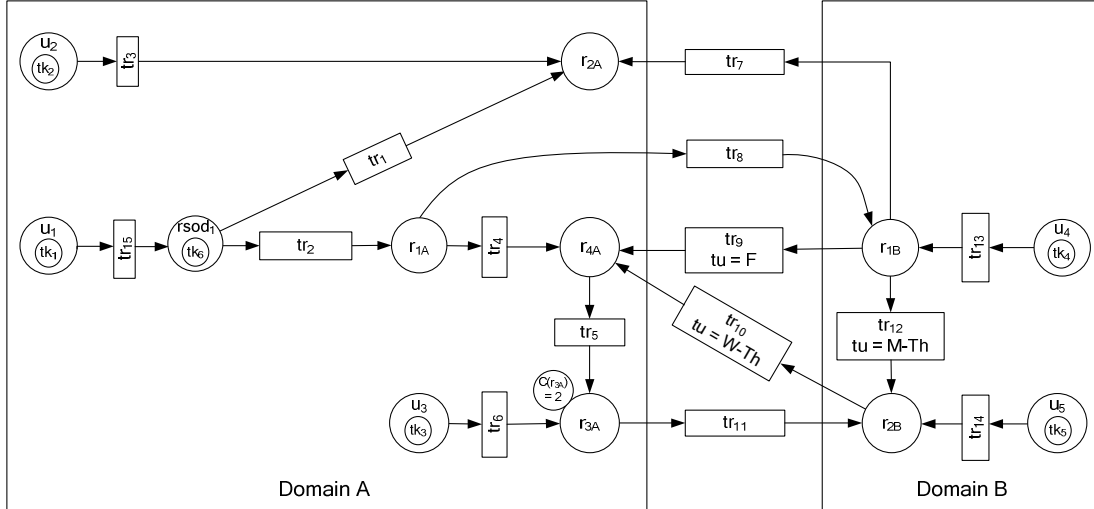


Figure 6. Example ConPN

Tokens

In ConPN, tokens represent the execution semantics of the inheritance policy. Tokens facilitate snapshot and post execution analyses that identify inter-domain policy conflicts. Their values can be updated and evaluated throughout the ConPN execution. Similar to tokens in a Colored Petri Net, tokens in ConPN are distinguishable from one another. Let TK be the set of all possible tokens allowed in the ConPN. We define a *Token* as follows.

Token = (*Type*, *Origin*, *Time*, *RLog*, *ConflictID*) such that

- Type*: normal or choice
- Origin*: Origin of this token, start or choice place
- Time*: Temporal Unit, initially empty
- RLog*: Bag of roles, initially empty
- ConflictID*: Set of place IDs

There are two types of tokens, normal and choice. The origin of the token indicates where the token starts, i.e., a start or choice place. Every start place has exactly one normal token that originates there (recall the 1:1 mapping of user to start place). Every choice place has exactly one choice token that originates there (recall the 1:1 mapping of SoD requirements to choice places). Figure 6 shows the initial token placements for the example. The token time is the current temporal unit held by the token as seen in Table 1 and is initialized to the empty set. As normal tokens move through the ConPN, they collect the IDs of the roles visited in *RLog*. This is a bag because normal tokens can visit the same role multiple times and each visit is recorded. *RLog* is initially empty. Choice tokens work to model SoD conflicts. Since these token types are restricted in their movements, they use *ConflictID* to initialize the role places that cannot be jointly accessed. *ConflictID* is always empty for normal tokens.

Tokens	Places
$tk_1 = ("normal", u_1, \emptyset, \emptyset, \emptyset)$	u_1
$tk_2 = ("normal", u_2, \emptyset, \emptyset, \emptyset)$	u_2
$tk_3 = ("normal", u_3, \emptyset, \emptyset, \emptyset)$	u_3
$tk_4 = ("normal", u_4, \emptyset, \emptyset, \emptyset)$	u_4
$tk_5 = ("normal", u_5, \emptyset, \emptyset, \emptyset)$	u_5
$tk_6 = ("choice", rsod_1, \emptyset, \emptyset, \{r_{1A}, r_{2A}\})$	$rsod_1$

Table 1: Initial Tokens and Places

Since start places and choice places in ConPN are each initialized with a single independent token of the proper type. When a start or role place has multiple output arcs, it replicates its token for each output arc. An example of this is role r_{1B} in Figure 6 in which a token from r_{1a} or u_4 will be replicated. In contrast, a choice place does not replicate tokens.

Transitions

As defined for the basic Petri Net, places are not directly connected to other places. Tokens must flow through a transition in the set T , from an input place to an output place according to transition firing rules. In ConPN, we extend the concept of a transition to include a function Temporal: $T \rightarrow TU$, which maps a transition to a set of temporal units from the power set of all such units, TU . These temporal units indicate the constraints on the time at which the transition's input place can pass a token to its output place.

Each role assignment in the inheritance policy (visualized by a directional arrow in Figure 6) has a corresponding transition, directly mapping to 14 transitions. We add to the set those transitions supporting the Role SoD. For the each Role SoD triple in the inheritance policy (u_1, r_{1A}, r_{2A}) , we include a new transition to each choice place that branches to those transitions associated with the competing role assignments. Thus, from our example, we introduce transition tr_{15} between r_{1A} and $rsod_1$. The final set of ConPN transitions generated by the inheritance policy in Figure 6 is $T = \{tr_1, \dots, tr_{15}\}$.

Arcs

Recall that an input arc connects a place in P to a transition in T and an output arc is directed from transition to place. Let Arc be the set of all arcs in the ConPN. To create the notion of SoD that disallows access to a role because access has been granted to a competing role, we provide each arc with a status by defining $Status: Arc \rightarrow \{\text{"active"}, \text{"blocked"}\}$ to indicate if an arc is valid to transition a token from one place to the next.

Figure 6 shows the complete ConPN generated from the same inheritance policy as the example in Figure 5.

Transition Firing Rules

As the ConPN executes, a *transition firing rule* analyzes its places, tokens, and arcs to dictate the flow of the tokens throughout the net. Tokens can flow in parallel. We tailor the transition rules so that the movements of tokens through the ConPN imitate the access granted through IP_A , IP_B , and IP_{Join} . The transition firing rules reflect the Inheritance, SoD, Cardinality, and Temporal constraints.

We use a precondition/postcondition format to express the transition firing rules. Notationally, “:=” is pseudo code for *gets*, “/” represents *set delete*, and “ Θ ” represents *bag union*. Only changes to entities are detailed in the postconditions, whose statement order is meaningful.

Transition firing rule **TR1** moves a normal token (tk) from a normal or start input place (p_1) to the transition's (tr) output place (p_2) that may be of any type. Token and place meta-data ($CurrTK$, $RLog$, and $TkLog$) are updated. If the transition has defined temporal restrictions, the temporal units of the normal token ($Time$) take on the intersection of the time units associated with the transition as dictated by the security principle [1]. Arc status is unaffected by **TR1**. We formally define the conditions under which **TR1** fires as follows.

For $tr \in T$; $tk \in TK$; $arc_1, arc_2 \in Arc$; $p_1 \in SP \cup RP$; $p_2 \in P$

Preconditions:

$arc_1 = (p_1, tr) \wedge arc_2 = (tr, p_2) \wedge Status(arc_1) = Status(arc_2) = \text{"active"}$
 $tk \in p_1.CurrTk \wedge tk.Type = \text{"normal"}$

Postconditions:

$p_1.CurrTk := p_1.CurrTk / \{tk\}$
 $p_2.CurrTk := p_2.CurrTk \oplus \{tk\}$
 $p_2.TkLog := p_2.TkLog \oplus \{tk\}$
 $tk.Time := tk.time \cap Temporal(tr)$
 $tk.RLog := tk.RLog \oplus \{p_2\}$

This transition firing rule, **TR1**, executes when there are active arcs connecting two places with a normal token being moved between them. The place (p_1) containing the token is either a start place (SP) or a role place (RP). The place where the token moves (p_2) can be any place within the ConPN. After the transition fires, the p_1 deletes the token from its set of current tokens ($p_1.CurrTk$). The token's time constraint ($tk.Time$) is updated to match the temporal units of the transition being fired. The token also updates its log of visited roles ($tk.RLog$) to include p_2 and the token is added to p_2 's set of current tokens ($p_2.CurrTk$). Finally, the token log for p_2 is updated to include the token ($p_2.TkLog$). Figure 7 depicts the resulting changes in the ConPN when **TR1** is applied to transition tr_9 .

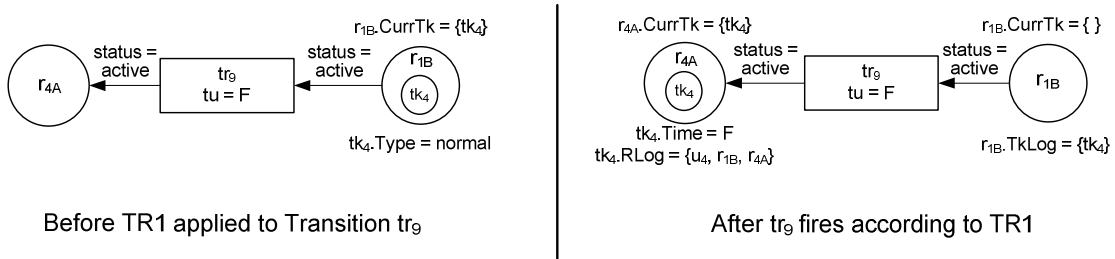


Figure 7. Transition Firing Rule TR1 as Applied to Transition tr_9

Transition firing rules **TR2** and **TR3** both rely on a choice place as their input place to enforce the restriction implicit in a role-based SoD requirement. **TR2** dictates the firing of the transition to move a normal token (tk) to a chosen role (r) when it resides at the choice place (c) where a choice token (ctk) also resides.

The choice token must be forced to move to the alternate role where it stays for the duration of the execution. The enforcement occurs because **TR2** changes the state of output arc (arc_1) that the normal token uses to "blocked." The choice token then has only one active arc to leave the choice place. **TR3** dictates the firing of the choice token (ctk), relying on the firing of **TR2** as indicated by the presence of a blocked arc (arc_1). For a conflict to exist there must be a secondary path to the non-chosen role. Hence, if the same normal token arrives at the place where the choice token newly resides, then the SoD requirement is violated.

TR2 fires under the following conditions.

For $tr \in T$; $tk, ctk \in TK$; $arc_1, arc_2 \in Arc$; $c \in RSoD$; $r \in RP$

Preconditions:

$arc_1 = (c, tr) \wedge arc_2 = (tr, r) \wedge Status(arc_1) = Status(arc_2) = \text{"active"}$
 $\{tk, ctk\} \subseteq c.CurrTk$
 $tk.Type = \text{"normal"} \wedge ctk.Type = \text{"choice"} \wedge ctk.Origin = c$

Postconditions:

$c.CurrTk := c.CurrTk / \{tk\}$
 $status(arc_1) := \text{"blocked"}$

$$\begin{aligned}
r.CurrTk &:= r.CurrTk \ominus \{tk\} \\
r.TkLog &:= r.TkLog \ominus \{tk\} \\
tk.Time &:= tk.Time \wedge \text{Temporal}(tr) \\
tk.RLog &:= tk.RLog \ominus \{r\}
\end{aligned}$$

Transition Firing Rule **TR2** applies when there are two active arcs (arc_1, arc_2) connecting a choice place (c) to a role place (r), where the choice place contains both a normal token (tk) and a choice token (ctk). The choice token originates ($ctk.Origin$) at the choice place where it is residing. After **TR2** fires, the choice place removes the normal token (tk) from its set of current tokens ($c.CurrTk$). The token adopts the temporal constraints of the transition ($tk.Time$), the token's log of visited roles ($tk.RLog$) now includes the role place (r), and the token is added to the role place's set of current tokens ($r.CurrTk$) and its log of tokens ($r.TkLog$). This rule additionally sets the status of the arc between the choice place and the transition (arc_1) as *blocked*. The choice token's existence is necessary to determining if **TR2** is enabled, but the choice token is moved separately using **TR3**, described next. **TR3** fires under the following conditions.

For $tr_1, tr_2 \in T$; $ctk \in TK$; $arc_1, arc_2, arc_3 \in Arc$; $c \in CP$; $r \in RP$

Preconditions:

$$\begin{aligned}
arc_1 &= (c, tr_1) \wedge arc_2 = (c, tr_2) \wedge arc_3 = (tr_2, r) \\
\text{Status}(arc_1) &= \text{"blocked"} \\
\text{Status}(arc_2) &= \text{Status}(arc_3) = \text{"active"} \\
ctk &\in c.CurrTk \wedge ctk.Type = \text{"choice"} \wedge ctk.Origin = c
\end{aligned}$$

Postconditions:

$$\begin{aligned}
c.CurrTk &:= c.CurrTk / \{ctk\} \\
ctk.ConflictID &:= ctk.ConflictID / \{r\} \\
r.CurrTk &:= r.CurrTk \ominus \{ctk\} \\
r.TkLog &:= r.TkLog \ominus \{ctk\}
\end{aligned}$$

The choice token is moved by **TR3**, which by definition can only be applied after **TR2** is fired. This is because of the precondition that an output arc (arc_1) from the choice place (c) is blocked. The choice token (ctk) must both originate from and currently reside in the choice place. Once **TR3** fires, the choice token is removed from the choice place ($c.CurrTk$) and the set of conflicting id's ($ctk.ConflictID$) is changed to omit the role place (r) where the choice token now resides. Only the alternate role is left in the *ConflictID* set. There are no transition firing rules to move the choice token. Thus, if a normal token joins this choice token and the normal token has visited the role in the choice token's *ConflictID* set, then the SoD requirement has been violated because the normal token has visited both roles places and should not have been able to. The role place (r) then adds the choice token to its set of current tokens ($r.CurrTk$) and its token log ($r.TkLog$).

Since choice tokens are not transitioned further by any firing rules and are not included in inheritance, cardinality, or temporal conflict assessment, no further postcondition changes are warranted than those presented above. It is important to note that the finding of a single conflict denotes a problem with this SoD constraint. Therefore, it is not important to the conflict detection that the arc blocks further flow from the RSoD place to the chosen role.

Figure 8 illustrates these transition firing rules on the ConPN example shown in Figure 5.

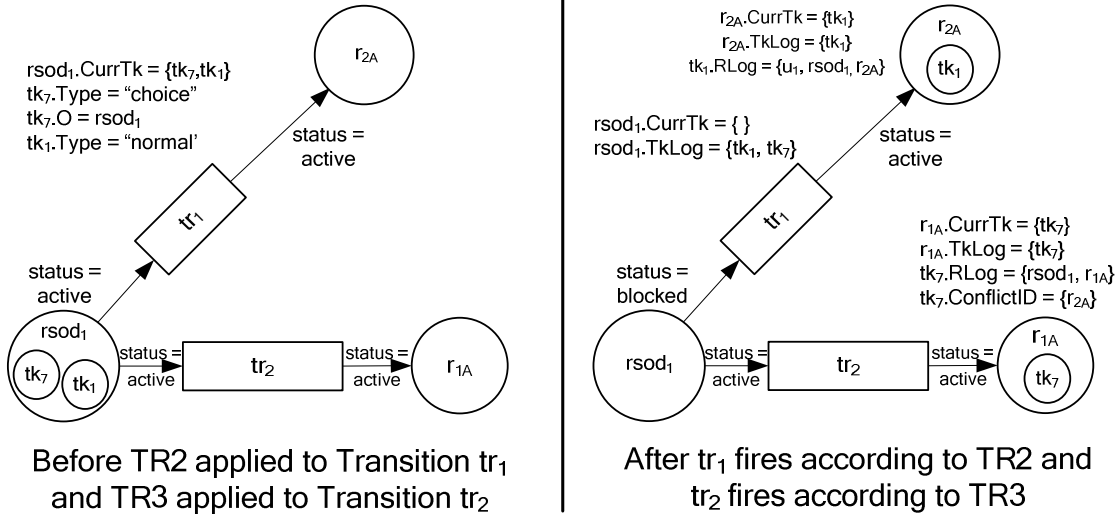


Figure 8. Transition Firing Rules TR2 and TR3

These transition firing rules provide the means to detect conflicts in the ConPN. By moving tokens through the Petri net, different states of the system are found. The ConPN executes in a step-wise fashion where enabled transitions are fired until a quiescent state is reached. This organized evaluation is easily automated using ConPN-specific software [24]. The next section shows how inheritance policy violations are found from the ConPN execution.

Finding Conflicts with ConPN

The ConPN can detect inheritance, role-based SoD, cardinality, and temporal compliance violations when representing an inheritance policy, IP . We discuss these findings below.

Inheritance Conflict

If a token visits the same role multiple times, it means that a role in IP can be visited by a cyclical firing of transitions in the ConPN. This state is evident in the $RLog$ of each token that retains the roles visited by the token. Thus, the cycle is visible if any place, other than a choice place, appears in the $RLog$ of a normal token more than once. Formally,

$$\exists tk \in Token; r \in SP \cup RP \text{ such that } tk.RLog(r) > 1 \Leftrightarrow \text{InheritanceConflict}(r)$$

We use a formal definition of the bag, $RLog$, as a function from places to the set of natural numbers indicating the number of times a role appears in the bag. Thus, $tk.RLog(r)$ returns the number of times r appears in $tk.RLog$.

Figure 9 shows a potential inheritance conflict in the ConPN using the example from Figure 5. Since $tk_3.RLog(r_{3A}) > 1$, it is clear that a violation exists. This violation occurs because in $IP_{join} \{(r_{2B}, r_{4A}, W-Th), (r_{3A}, r_{2B}, \infty)\} \subseteq M_{join}$. This mapping translates into an execution of the ConPN in which the normal token, tk_3 , inside role r_{3A} is shown as having progressed through the sequence of transitions $\langle tr_6, tr_{11}, tr_{10}, tr_3 \rangle$ to progress from u_3 to r_{3A} , r_{3A} to r_{2B} , r_{2B} to r_{4A} , and back to r_{3A} . As shown in bold text in Figure 9, the $tk_3.RLog$ contains role r_{3A} twice indicating an inheritance conflict. Because r_{3A} is both superior and inferior to r_{4A} , the RBAC system cannot determine which role is genuinely superior.

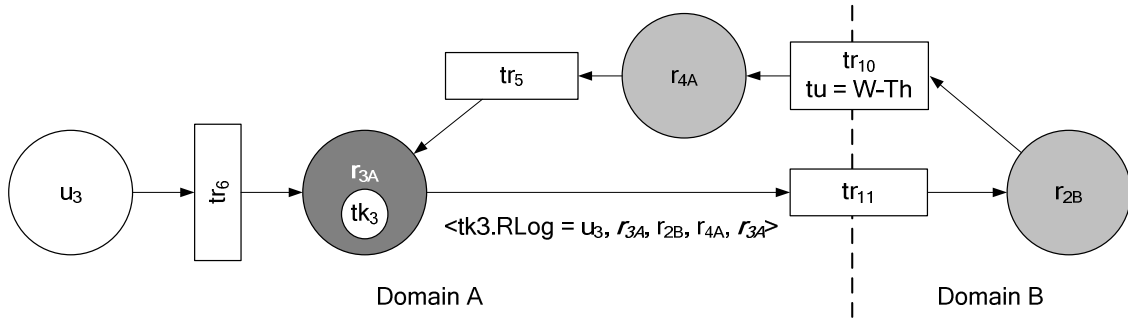


Figure 9. Inheritance Conflict

Role-Based SoD Conflict

A potential role-based SoD conflict is detected if a normal token visits a role place that holds a choice token under the following conditions.

- The choice token is not at its place of origin.
- The choice token holds in its ConflictID set a role that the normal token has visited.

This indicates that the choice token was unable to protect the use of the role, where it currently resides, from being accessed by a token that has already accessed the competing role governed by the RSoD constraint. This is formalized as follows.

$$\begin{aligned}
 &\exists tk, ctk \in Token; r_1, r_2 \in RP \text{ such that} \\
 &\quad tk.type = \text{"normal"} \wedge ctk.type = \text{"choice"} \wedge \\
 &\quad \{tk, ctk\} \subseteq r_1.CurrTk \wedge ctk.Origin \neq r_1 \wedge \\
 &\quad r_2 \in tk.RLog \wedge r_2 \in ctk.ConflictID \\
 &\Leftrightarrow RSoDConflict(r_1)
 \end{aligned}$$

For role-based SoD transitions, the transition firing rules **TR2** and **TR3** are the only ones enabled. Each role-based SoD place holds a choice token, where its set of conflicting IDs (*ConflictID*) is initialized with the competing roles. Upon the first normal token's path being taken from a SoD place, **TR2** is performed. The normal token moves to the next place and the arc it took becomes blocked. **TR3** becomes enabled in this state. When **TR3** fires, it sends the resident choice token down the alternate path from the normal token. Choice tokens will never be moved from a role place since no transition firing rules exist to move them out of a role place.

Figure 10 shows the role-based SoD conflict in the example. The lightly shaded role places (r_{1B} , r_{1A}) indicate the path of the normal token. The more darkly shaded place (r_{2A}) indicates the conflict where the two tokens arrive at the same place. User u_1 initially contains the normal token tk_1 . The choice place, $rsod_1$, initially contains the choice token tk_6 . When these tokens are both contained in $rsod_1$, they are moved according to rules **TR2** and **TR3**. Assume the case in which tk_1 moves through tr_2 to r_{1A} . By **TR3**, tk_6 is moved to r_{2A} through tr_1 . The *ConflictID* set of tk_6 becomes $\{r_{1A}\}$. As tk_1 moves through the ConPN, it is able to access r_{2A} . A violation occurs because its RLog contains r_{1A} . Thus, user u_1 can be assigned roles r_{1A} and r_{2A} simultaneously, bypassing the Role SoD constraint between roles r_{1A} and r_{2A} .

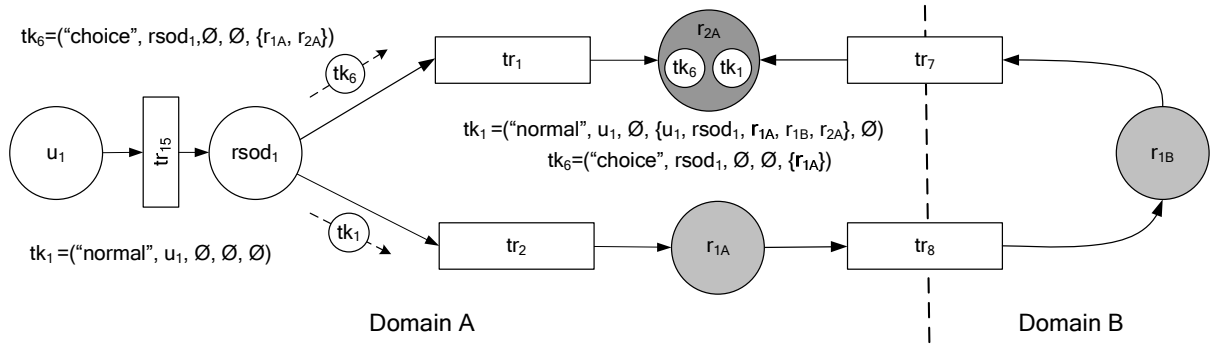


Figure 10. Role-Based SoD Conflict

Cardinality Conflict

A potential cardinality conflict is present in the ConPN when more tokens visit a role place than its cardinality allows. Cardinality conflicts are formally defined as follows, with the symbol ‘#’ meaning ‘size of’.

$$\exists r \in RP \text{ such that } \mathbf{C}(r) < \#r.TkLog \Leftrightarrow \text{CardinalityConflict}(r)$$

Figure 11 shows the cardinality conflict in the ConPN where tokens t_1 , t_3 , and t_4 have reached role place r_{3A} . According to IP_A , u_1 and u_3 can legitimately access r_{3A} . Given that $(r_{1B}, r_{4A}, F) \in M_{join}$, user u_4 can also access role r_{3A} leading to cardinality conflict as the number of tokens reaching role place r_{3A} exceeds the cardinality limit. Though, not shown in Figure 11, u_5 can also access r_{3A} .

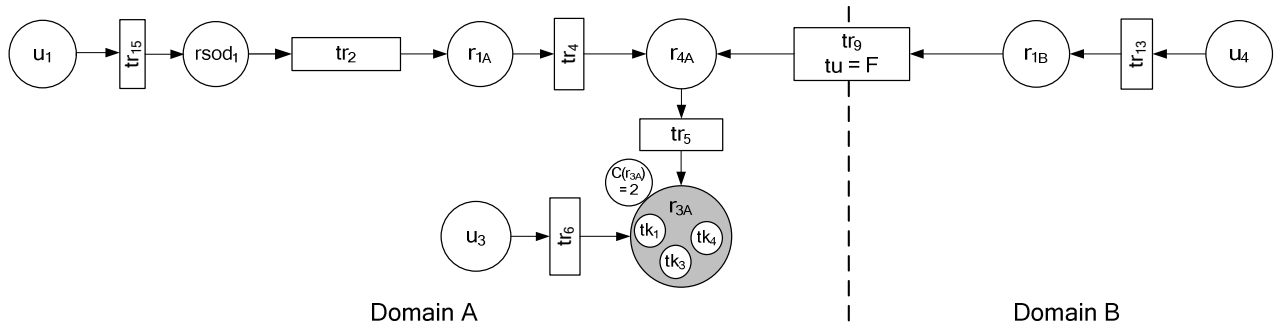


Figure 11. Cardinality Conflict

Temporal Conflict

A potential temporal conflict is present if tokens originating from the same start place can reach the same role place via distinct paths and end up with unequal temporal units. The inequality means that the role has been over or under restricted due to the inter-domain mappings. Both cases are considered improper access. Formally, a potential temporal conflict exists when

$$\exists tk_1, tk_2 \in Token; r_1 \in SP; r_2 \in RP \text{ such that}$$

$$\begin{aligned}
& (tk_1.Origin = r_1 \wedge tk_2.Origin = r_1) \wedge \\
& r_2 \in tk_1.RLog \wedge r_2 \in tk_2.RLog \wedge tk_1.RLog \neq tk_2.RLog) \wedge \\
& (tk_1.Time \neq tk_2.Time) \\
& \Leftrightarrow \text{TemporalConflict}(r_1, r_2)
\end{aligned}$$

The temporal unit of an inter-domain mapping is the smallest unit granularity on which temporal constraints are defined, for example days or hours. Temporal constraints are dictated by a transition when the token's *Time* is assigned the value of the Temporal function for the transition. Figure 12 illustrates how the ConPN identifies this conflict type. User u_4 has access to r_{4A} via two paths: $u_4 \rightarrow r_{1B} \rightarrow r_{4A}$ and $u_4 \rightarrow r_{1B} \rightarrow r_{2B} \rightarrow r_{4A}$. Note that r_{1B} replicates token tk_4 to create tk'_4 . Transitions tr_9 and tr_{10} have different temporal units, which cause confusion as to when u_4 should be allowed r_{4A} 's permissions. Because the two temporal units are different, improper access can occur. This state occurs even each transition reassigns the temporal units to the intersection of the existing Time value and the transition's time constraint.

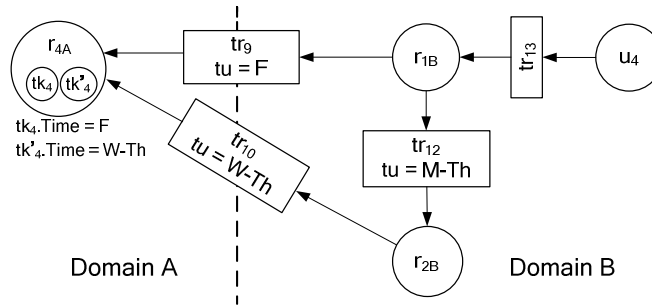


Figure 12. Temporal Conflict

Discussion and Conclusion

Many times different vendors produce components with specific interfaces that require proxies, wrappers, and glue code to mesh incompatible software components. This difficult integration process can introduce unexpected security vulnerabilities in a system of systems, limiting tremendous opportunities for these integrated applications to streamline an organization's operation.

The Conflict Petri Net, ConPN, highlights access control conflicts across inter-domain mappings. By interpreting an access control policy into our InheritancePolicy tuple, the ConPN is constructed and then evaluates the policy. This allows us freedom to represent a policy in any format (a common format is XML) that can be interpreted into our tuple. The ConPN can be proven to find all inter-domain conflicts related to role hierarchy, role-based SoD, cardinality, and temporal constraints in RBAC systems [27]. The simplicity of the mapping between an Inheritance Policy to the ConPN entities makes the equivalence clear. Because we have partitioned the conflict definitions across the individual perspectives of inheritance, SoD, cardinality, and temporal constraints, ConPN finds all potential conflicts, some of which may be eliminated by another perspective. With these assurances, we take advantage of the benefits of using a Petri Net, including the direct automation of token flow and access control conflict detection, as shown in [27].

Acknowledgements. This material is based on research sponsored in part by US Air Force Office of Scientific Research (AFOSR) award FA9550-05-1-0374. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

- [1] M. Bishop, *Computer Security: Art and Science*, Addison-Wesley, 2003.
- [2] S. Goel, C. Clifton, and A. Rosenthal, "Derived access control specification for XML," presented at ACM workshop on XML security Fairfax, Virginia, 2003.
- [3] R. Bhatti, B. Shafiq, E. Bertino, A. Ghafoor, and J. B. Joshi, "X-GTRBAC Admin: A Decentralized Administration Model for Enterprise-Wide Access Control," *ACM Transactions on Information and System Security*, vol. 8, pp. 388-423, 2005.
- [4] I. Ray, R. France, N. Li, and G. Georg, "An Aspect-Based Approach to Modeling Access Control Concerns," *Information and Software Technology*, vol. 40, pp. 557 - 633, 2004.
- [5] B. D. Joshi, B. Shafiq, A. Ghafoor, and B. Bertino, "Dependencies and Separation of Duty Constraints in GTRBAC," presented at eighth ACM symposium on Access control models and technologies, Como, Italy, 2003.
- [6] K. Jensen, *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, vol. 1: Springer-Verlag, 1992.
- [7] K. M. Khan and J. Han, "Deriving Systems Level Security Properties of Component Based Composite Systems," presented at Australian Software Engineering Conference (ASWEC'05), 2005.
- [8] L. Qin and V. Atluri, "Concept-level access control for the Semantic Web," presented at 2003 ACM workshop on XML security, Fairfax, VA, 2003.
- [9] M. A. Kelkar, M. Smith, and R. Gamble, "Interaction Partnering Criteria for COTS Components," presented at International Conference on Software Engineering and Knowledge Engineering San Francisco, CA, 2006.
- [10] B. Shafiq, J. B. Joshi, E. Bertino, and A. Ghafoor, "Secure Interoperation in a Multidomain Environment Employing RBAC Policies," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, pp. 1557-1577, 2005.
- [11] A. A. El Kalam, R. E. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Mieke, C. Saurel, and G. Trouessin, "Organization based access control," presented at Proceedings of the 4th International Workshop on Policies for Distributed Systems and Networks (POLICY '03), Toulouse, France, 2003.
- [12] D. F. Ferraiolo, G. Ahn, R. Chandramouli, and S. I. Gavrila, "The Role Control Center: Features and Case Studies," presented at ACM Symposium on Access Control Models and Technologies Villa Gallia, Como, Italy 2003.
- [13] A. Gummadi, J. P. Yoon, B. Shah, and V. Raghavan, "A Bitmap-Based Access Control For Restricted Views of XML Documents," presented at ACM Workshop on XML Security, Fairfax, Virginia 2003.
- [14] I. Ray, N. Li, R. France, and D. Kim, "Using UML To Visualize Role-Based Access Control Constraints," presented at ACM Symposium on Access Control Models and Technologies, Yorktown Heights, USA, 2004.
- [15] I. Ray, N. Li, R. France, and D. Kim, "Using UML to Visualize Role-Based Access Control Constraints," presented at ninth ACM Symposium on Access Control Models and Technology, York town, New York, USA, 2004.
- [16] C. Girault and R. Valk, *Petri Nets for Systems Engineering A guide to Modeling, Verification and Applications*: Springer Verlag, 2003.
- [17] W. Reisig, *Petri Nets: An Introduction*: Springer-Verlag, 1985.
- [18] M. K. Molloy, "Performance Analysis Using Stochastic Petri Nets," *IEEE Transactions on Computers*, vol. 31, pp. 913-917, 1982.
- [19] T. D. C. Little and A. Ghafoor, "Synchronization and Storage Models for Multimedia Objects," *IEEE Journal on Selected Areas in Communications*, vol. 8, pp. 413-427, 1990.
- [20] J. B. Joshi and A. Ghafoor, "A Petri-net based multilevel security specification model for multimedia documents," presented at IEEE International Conference on Multimedia and Expo, 2000.
- [21] K. Juszczyszyn, "Verifying Enterprises Mandatory Access Control Policies with Coloured Petri Nets," presented at Twelfth International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003.
- [22] K. Knorr, "Dynamic Access Control through Petri Net Workflows," presented at 16th Annual Computer Security Applications Conference New Orleans, Louisiana, USA, 2000.
- [23] Z. Zhang, F. Hong, H. Xiao, "Verification of Strict Integrity Policy via Petri Nets," Proceedings of the International Conference on Systems and Networks Communication, 2006.
- [24] Z. Zhang, F. Hong, J. Liao, "Modeling Chinese Wall using Colored Petri Nets," Proceedings of The Sixth IEEE International Conference on Computer and Information Technology, 2006.

- [25] M. Song & Z. Pang, "Specification of SA-RBAC Policy Based on Colored Petri Net," IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, 2008.
- [26] A. Anderson, "Core and hierarchical role based access control (RBAC) profile of XACML v2.0," presented at Oasis, 2005.
- [27] A. Walvekar, "ConPN: Detecting Conflicts in Inter-domain Mappings," in *M.S. Thesis, Department of Mathematical and Computer Sciences*. Tulsa, OK: University of Tulsa, 2006.