Detecting and Resolving Conflicts of Mutual-Exclusion and Binding Constraints in a Business Process Context

Sigrid Schefer[1], Mark Strembeck[1], Jan Mendling[2], and Anne Baumgrass[1]

[1] Institute for Information Systems, New Media Lab
Vienna University of Economics and Business (WU Vienna), Austria
{firstname.lastname}@wu.ac.at
[2] Institute for Information Business
Vienna University of Economics and Business (WU Vienna), Austria
jan.mendling@.wu.ac.at

Abstract. Mutual exclusion and binding constraints are important means to define which combinations of subjects and roles can be assigned to the tasks that are included in a business process. Due to the combinatorial complexity of potential role-to-subject and task-to-role assignments, there is a strong need to systematically check the consistency of a given set of constraints. In this paper, we discuss the detection of consistency conflicts and provide resolution strategies for the corresponding conflicts.

Keywords: business processes, information systems, mutual exclusion, separation of duty, binding of duty

1 Introduction

In recent years, business processes are increasingly designed with security and compliance considerations in mind (see, e.g., [3, 16, 19]). For example, the definition of process-related security properties is important if a conflict of interest could arise from the simultaneous assignment of decision and control tasks to the same subject. In this context, process-related access control mechanisms are typically used to specify authorization constraints, such as *separation of duty (SOD)* and *binding of duty (BOD)*, to regulate which subject is allowed (or obliged) to execute a particular task (see, e.g., [4, 5, 14–17, 19]).

In a workflow environment, SOD constraints enforce conflict of interest policies by defining that two or more tasks must be performed by different individuals. Conflict of interest arises as a result of the simultaneous assignment of two mutual exclusive entities (e.g. permissions or tasks) to the same subject. Tasks can be defined as statically mutual exclusive (on the process type level) or dynamically mutual exclusive (on the process instance level). Thus, a *static mutual exclusion (SME)* constraint is global with respect to all process instances in an information system. Therefore, two SME tasks can never be assigned to the same subject or role. On the other hand, two *dynamically mutual exclusive*

(DME) tasks can be assigned to the same subject but must not be executed by the same subject in the same process instance.

In contrast, BOD constraints specify that *bound tasks* must always be performed by the same subject or role (see, e.g., [14–17]). BOD can be subdivided into subject-based and role-based constraints (see, e.g., [14, 15]). A *subject-based BOD constraint* defines that the same individual who performed the first task must also perform the bound task(s). On the other hand, a *role-based BOD constraint* defines that bound tasks must be performed by members of the same role, but not necessarily by the same individual. Throughout the paper, we will use the terms *subject-binding (SB)* and *role-binding (RB)* as synonyms for subject-based BOD constraints and role-based BOD constraints, respectively.

In recent years, role-based access control (RBAC) [7, 11] has developed into a de facto standard for access control. A specific problem in the area of process-related RBAC is the immanent complexity of interrelated mutual-exclusion and binding constraints. Thus, when defining process-related mutual-exclusion or binding constraints, design-time and runtime checks need to ensure the consistency of the corresponding RBAC model. In particular, at design-time conflicts may result from inconsistent constraints or assignment relations. At runtime conflicts may result from invalid task-to-subject allocations (see also [14]).

In this paper, we take the conflicts identified in [14] as a starting point. We adapt the algorithms from [14] to detect and name corresponding conflicts, and discuss resolution strategies for these conflicts. In particular, we consider conflicts at the level of design-time constraint definition, design-time assignment relations, and runtime task allocation.

The remainder of this paper is structured as follows. Section 2 gives an overview of process-related RBAC models and the requirements for design-time and runtime consistency of these models. Sections 3, 4, and 5 present algorithms to detect potential conflicts of mutual-exclusion and binding constraints. Furthermore, we provide resolution strategies that exemplary show how these conflicts can be resolved to ensure the consistency of a process-related RBAC model. Subsequently, Section 6 discusses related work and Section 7 concludes the paper.

2 Process-Related RBAC Models

The algorithms and resolution strategies presented in Section 3, 4, and 5 are based on the formal definitions for process-related RBAC models from [14, 15]. However, due to the page restrictions we cannot repeat the complete list of definitions in this paper. Therefore, we now give an overview of the definitions we use below – for further details please consult [14, 15].

Definition 1 (Process-related RBAC Model). *A Process-related RBAC Model* $PRM = (E, Q, D)$ *where* $E = S \cup R \cup P_T \cup P_I \cup T_T \cup T_I$ *refers to pairwise disjoint sets of the model,* $Q = rh \cup rsa \cup tra \cup es \cup er \cup ar \cup pi \cup ti$ *to mappings that establish relationships, and* $D = sb \cup rb \cup sme \cup dme$ *to binding and mutual-exclusion constraints.*

An element of S is called *Subject*. An element of R is called *Role*. An element of P_T is called *Process Type*. An element of P_I is called *Process Instance*. An element of T_T is called *Task Type*. An element of T_I is called *Task Instance*.

We allow the definition of subject-binding (sb), role-binding (rb), static mutual exclusion (sme), and dynamic mutual exclusion (dme) constraints on task types. Roles can be arranged in a role-hierarchy (rh), where more powerful senior-roles inherit the permissions from their junior-roles. The task-to-role assignment relation (tra) defines which tasks can be performed by the members of a certain role. Thereby, tra specifies the permissions of a role. The task-ownership mapping ($town$) allows to determine which tasks are assigned to a particular role – including the tasks inherited from junior-roles. The inverse mapping ($town^{-1}$) returns the set of roles a task is assigned to. The role-to-subject assignment relation (rsa) defines which roles are assigned to particular users. The role-ownership mapping ($rown$) returns all roles assigned to a certain subject (including roles that are inherited via a role-hierarchy). The inverse mapping ($rown^{-1}$) allows to determine all subjects assigned to a particular role. Each subject can activate the roles that are assigned to this subject, and the active-role mapping (ar) returns the role that is currently activated. For each task instance we have an executing-subject (es) and an executing-role (er).
Definition 2 provides rules for the static correctness of process-related RBAC models to ensure the design-time consistency of the included elements and relationships.

Definition 2. *Let $PRM = (E, Q, D)$ be a Process-related RBAC Model. PRM is said to be statically correct if the following requirements hold:*

1. *Tasks cannot be mutual exclusive to themselves:*
   $\forall t_2 \in sme(t_1) : t_1 \neq t_2$ *and* $\forall t_2 \in dme(t_1) : t_1 \neq t_2$
2. *Mutuality of mutual exclusion constraints:*
   $\forall t_2 \in sme(t_1) : t_1 \in sme(t_2)$ *and* $\forall t_2 \in dme(t_1) : t_1 \in dme(t_2)$
3. *Tasks cannot be bound to themselves:*
   $\forall t_2 \in sb(t_1) : t_1 \neq t_2$ *and* $\forall t_2 \in rb(t_1) : t_1 \neq t_2$
4. *Mutuality of binding constraints:*
   $\forall t_2 \in sb(t_1) : t_1 \in sb(t_2)$ *and* $\forall t_2 \in rb(t_1) : t_1 \in rb(t_2)$
5. *Tasks are either statically or dynamically mutual exclusive:*
   $\forall t_2 \in sme(t_1) : t_2 \notin dme(t_1)$
6. *Either SME constraint or binding constraint:*
   $\forall t_2 \in sme(t_1) : t_2 \notin sb(t_1) \wedge t_2 \notin rb(t_1)$
7. *Either DME constraint or subject-binding constraint:*
   $\forall t_2 \in dme(t_1) : t_2 \notin sb(t_1)$
8. *Consistency of task-ownership and SME:*
   $\forall t_2 \in sme(t_1) : town^{-1}(t_2) \cap town^{-1}(t_1) = \emptyset$
9. *Consistency of role-ownership and SME:* $\forall t_2 \in sme(t_1), r_2 \in town^{-1}(t_2), r_1 \in town^{-1}(t_1) : rown^{-1}(r_2) \cap rown^{-1}(r_1) = \emptyset$

Definition 3 provides the rules for dynamic correctness of a process-related RBAC model, i.e. the rules that can only be checked in the context of runtime process *instances*.

Definition 3. *Let $PRM = (E, Q, D)$ be a Process-related RBAC Model and P_I its set of process instances. PRM is said to be dynamically correct if the following requirements hold:*

1. *In the same process instance, the executing subjects of SME tasks must be different:*
   $\forall t_2 \in sme(t_1), p_i \in P_I : \forall t_x \in ti(t_2, p_i), t_y \in ti(t_1, p_i) : es(t_x) \cap es(t_y) = \emptyset$
2. *In the same process instance, the executing subjects of DME tasks must be different:*
   $\forall t_2 \in dme(t_1), p_i \in P_I : \forall t_x \in ti(t_2, p_i), t_y \in ti(t_1, p_i) : es(t_x) \cap es(t_y) = \emptyset$
3. *In the same process instance, role-bound tasks must have the same executing-role:* $\forall t_2 \in rb(t_1), p_i \in P_I : \forall t_x \in ti(t_2, p_i), t_y \in ti(t_1, p_i) : er(t_x) = er(t_y)$
4. *In the same process instance, subject-bound tasks must have the same executing-subject:* $\forall t_2 \in sb(t_1), p_i \in P_I : \forall t_x \in ti(t_2, p_i), t_y \in ti(t_1, p_i) : es(t_x) = es(t_y)$

3 Constraint Definition Conflicts

When defining SME, DME, RB, or SB constraints at design-time, a number of conflicts may occur that would lead to inconsistencies in the corresponding process-related RBAC model. Below we first present algorithms to detect these constraint definition conflicts. If a conflict is detected, the algorithms return the name of the respective conflict. In the following subsections, we provide descriptions for each conflict type and present different resolution strategies.

3.1 Algorithms for Detecting Constraint Definition Conflicts

Algorithm 1 *Check if the definition of a new SME constraint is allowed.*

Name: $isSMEConstraintAllowed$
Input: $task_1, task_2 \in T_T$
if $task_1 == task_2$ then return selfConstraintConflict
if $task_1 \in dme(task_2)$ then return directDMEConflict
if $task_1 \in allRoleBindings(task_2)$ then return RBConflict
if $task_1 \in allSubjectBindings(task_2)$ then return SBConflict
if $\exists\ r \in R \mid r \in town^{-1}(task_1) \wedge r \in town^{-1}(task_2)$
    then return taskOwnershipConflict
if $\exists\ s \in S \mid r_1 \in rown(s) \wedge r_2 \in rown(s) \wedge$
    $r_1 \in town^{-1}(task_1) \wedge r_2 \in town^{-1}(task_2)$
    then return roleOwnershipConflict
return true

Algorithm 2 *Check if the definition of a new DME constraint is allowed.*

Name: $isDMEConstraintAllowed$
Input: $task_1, task_2 \in T_T$
if $task_1 == task_2$ then return selfConstraintConflict

if $task_1 \in sme(task_2)$ **then return** *directSMEConflict*
if $task_1 \in allSubjectBindings(task_2)$ **then return** *SBConflict*
return *true*

Algorithm 3 *Check if the definition of a new RB constraint is allowed.*

Name: $isRBConstraintAllowed$
Input: $task_1, task_2 \in T_T$
if $task_1 == task_2$ **then return** *selfConstraintConflict*
if $task_1 \in sme(task_2)$ **then return** *directSMEConflict*
if $\exists task_x \in sme(task_1) \mid task_x \in allRoleBindings(task_2)$
  then return *transitiveSMEConflict*
if $\exists task_x \in sme(task_2) \mid task_x \in allRoleBindings(task_1)$
  then return *transitiveSMEConflict*
return *true*

Algorithm 4 *Check if the definition of a new SB constraint is allowed.*

Name: $isSBConstraintAllowed$
Input: $task_1, task_2 \in T_T$
if $task_1 == task_2$ **then return** *selfConstraintConflict*
if $task_1 \in dme(task_2)$ **then return** *directDMEConflict*
if $task_1 \in sme(task_2)$ **then return** *directSMEConflict*
if $\exists task_x \in sme(task_1) \mid task_x \in allSubjectBindings(task_2)$
  then return *transitiveSMEConflict*
if $\exists task_x \in dme(task_1) \mid task_x \in allSubjectBindings(task_2)$
  then return *transitiveDMEConflict*
if $\exists task_x \in sme(task_2) \mid task_x \in allSubjectBindings(task_1)$
  then return *transitiveSMEConflict*
if $\exists task_x \in dme(task_2) \mid task_x \in allSubjectBindings(task_1)$
  then return *transitiveDMEConflict*
return *true*

3.2 Resolving Constraint Definition Conflicts

Self-constraint conflict: A *self-constraint conflict* occurs if we try to define tasks as mutual exclusive or bound to themselves (see Figure 1a and Algorithms 1-4). However, because mutual exclusion as well as binding constraints must be defined on two different task types, such a "self-exclusion" or "self-binding" would violate the consistency requirements defined in Def. 2.1 and Def 2.3.

Resolution to self-constraint conflicts: In order to prevent inconsistencies resulting from a *self-constraint conflict*, mutual exclusion and binding constraints need always be defined on two different task types (see Resolution 1 and Figure 1a).

Direct SME conflict: A *direct SME conflict* occurs if one tries to define a new DME, RB, or SB constraint on two task types which are already defined as

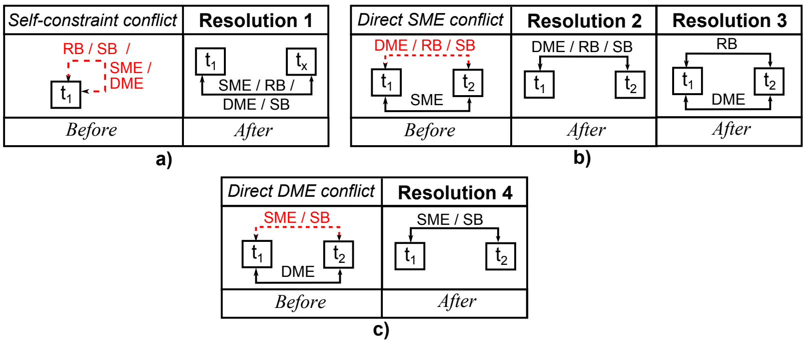

Fig. 1: Resolving self-constraint (a), SME (b), or DME (c) conflicts

being statically mutual exclusive (see Figure 1b). However, as defined in Def. 2.5, two tasks can either be statically or dynamically mutual exclusive (see also [14, 15]). Furthermore, if two tasks are defined as statically mutual exclusive, it is not possible to define a binding constraint between the same tasks (see Def. 2.6).

Resolutions to direct SME conflicts: Figure 1b shows two resolutions to prevent *direct SME conflicts*. In particular, this type of conflict can be avoided by removing the conflicting SME constraint before defining the new DME or binding constraint (see Resolution 2). If a direct SME conflict occurs when defining a RB constraint, it can also be resolved by changing the SME into a DME constraint (see Resolution 3), because DME constraints do not conflict with RB constraints (see [14, 15]).

Direct DME conflict: A *direct DME conflict* occurs if one tries to define a new SME or SB constraint on two task types which are already defined as being dynamically mutual exclusive (see Figure 1c). However, as defined in Def. 2.5, two tasks can either be statically or dynamically mutual exclusive. Moreover, DME and SB constraints conflict (see Def. 2.7, Def. 3.2, and Def. 3.4).

Resolution to direct DME conflicts: A *direct DME conflict* can be prevented by removing the conflicting DME constraint before defining the new SME or SB constraint (see Resolution 4 and Figure 1c).

RB conflict: A *RB conflict* arises if one tries to define a new SME constraint on two role-bound task types (see Figure 2a). However, because one cannot define a SME constraint and a RB constraint on the same task types (see Def. 2.6), such a configuration would result in a RB conflict.

Resolution to RB conflicts: A *RB conflict* can be prevented by removing the conflicting RB constraint before defining the new SME constraint (see Resolution 5 and Figure 2a).

SB conflict: A *SB conflict* arises if one tries to define a SME or a DME constraint between two subject-bound tasks (see Figure 2b). However, because we cannot define a mutual exclusion constraint and a SB constraint on the same task types (see Def. 2.6 and Def. 2.7), such a configuration would result in a SB conflict.

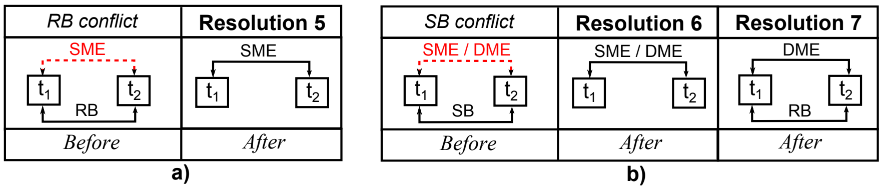

Fig. 2: Resolving RB conflicts (a) or SB conflicts (b)

Resolutions to SB conflicts: A *SB conflict* can be prevented by removing the conflicting SB constraint before defining the new mutual exclusion constraint (see Resolution 6 and Figure 2b). If a SB conflict occurs when defining a DME constraint, it can also be avoided by changing the conflicting SB constraint into a RB constraint (see Resolution 7), because DME and RB do not conflict (see [14, 15]).

Resolution Strategies for Constraint Definition Conflicts

The following resolution strategies define the conflict resolutions described above with respect to the formal definitions of process-related RBAC models (see Section 2 and [14, 15]).

Resolution 1 *Select two different tasks*

`Input:` $task_i \in T_T$
`select` $task_x \in T \mid task_i \neq task_x \wedge task_x \notin sme(task_i) \wedge task_x \notin dme(task_i) \wedge$
$\quad task_x \notin allRoleBindings(task_i) \wedge task_x \notin allSubjectBindings(task_i)$

Resolution 2 *Remove SME constraint*

`Input:` $task_1, task_2 \in T_T$
`remove` $task_1$ `from` $sme(task_2)$ `so that` $task_1 \notin sme(task_2)$

Resolution 3 *Change SME constraint into DME constraint*

`Input:` $task_1, task_2 \in T_T$
`remove` $task_1$ `from` $sme(task_2)$ `so that` $task_1 \notin sme(task_2)$
`and add` $task_1$ `to` $dme(task_2)$ `so that` $task_1 \in dme(task_2)$

Resolution 4 *Remove DME constraint*

`Input:` $task_1, task_2 \in T_T$
`remove` $task_1$ `from` $dme(task_2)$ `so that` $task_1 \notin dme(task_2)$

Resolution 5 *Remove RB constraint*

`Input:` $task_1, task_2 \in T_T$
`remove` $task_1$ `from` $rb(task_2)$ `so that` $task_1 \notin rb(task_2)$

Resolution 6 *Remove SB constraint*

`Input:` $task_1, task_2 \in T_T$
1: `remove` $task_1$ `from` $sb(task_2)$ `so that` $task_1 \notin sb(task_2)$

Resolution 7 *Change SB constraint into RB constraint*

`Input:`$task_1, task_2 \in T_T$
1: `remove` $task_1$ `from` $sb(task_2)$ `so that` $task_1 \notin sb(task_2)$
2: `and add` $task_1$ `to` $rb(task_2)$ `so that` $task_1 \in rb(task_2)$

3.3 Resolving Ownership Conflicts

Task-ownership conflict: A *task-ownership conflict* occurs if one tries to define a SME constraint between two task types that are already assigned to the same role (see Figure 3a). Because two SME tasks must never be assigned to the same role (neither directly nor transitively) such a configuration would result in a task-ownership conflict (see Def. 2.8).

Resolutions to task-ownership conflicts: Figure 3a shows two resolutions to prevent *task-ownership conflicts*. A task-ownership conflict can be avoided by revoking one of the tasks from the corresponding role before defining the new SME constraint (see Resolution 8), or by deleting the conflicting role before defining the new SME constraint (see Resolution 9). Note that Resolution 9 will rarely be applicable in real-world scenarios and is thus only presented for the sake of completeness.

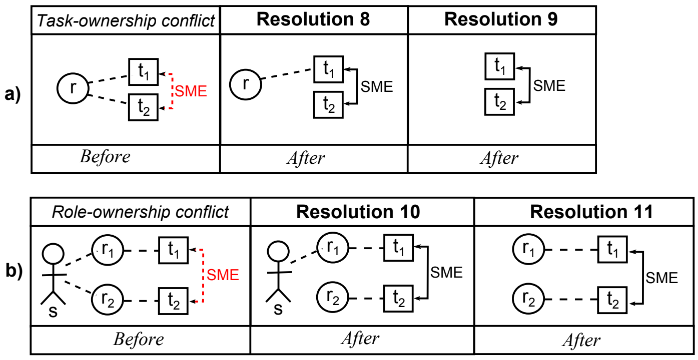

Fig. 3: Resolving task-ownership (a) and role-ownership (b) conflicts

Role-ownership conflict: A *role-ownership conflict* occurs if one tries to define a SME constraint on two task types which are (via the subject's roles) already assigned to the same subject (see Figure 3b). Because two SME tasks must never be assigned to the same subject (see Def. 2.9) such a configuration would result in a role-ownership conflict.

Resolutions to role-ownership conflicts: A *role-ownership conflict* as shown in Figure 3b can be prevented by revoking one of the conflicting task-to-role assignments before defining the new SME constraint (see Resolution 8), or by revoking one of the corresponding roles from the subject before defining the new SME constraint (see Resolution 10). Alternatively, it can be avoided by removing role r_1 or r_2 (see Resolution 9) or by removing the subject which owns the conflicting roles (see Resolution 11). Again, Resolutions 9 and 11 will rarely be applicable in real-world scenarios and are only presented for the sake of completeness.

Resolution Strategies for Ownership Conflicts

The following resolution strategies define the conflict resolutions described above with respect to the formal definitions of process-related RBAC models (see Section 2 and [14, 15]).

Resolution 8 *Remove task-to-role assignment*

`Input:` $role \in R, task \in T_T$
1: `remove` $role$ `from` $town^{-1}(task)$ `so that` $role \notin town^{-1}(task)$

Resolution 9 *Remove role*

`Input:` $role \in R$
1: `remove` $role$ `from` R `so that` $role \notin R$

Resolution 10 *Remove role-to-subject assignment*

`Input:` $subject \in S, role \in R$
1: `remove` $role$ `from` $rown(subject)$ `so that` $role \notin rown(subject)$

Resolution 11 *Remove subject*

`Input:` $subject \in S$
1: `remove` $subject$ `from` S `so that` $subject \notin S$

Resolution 12 *Remove task*

`Input:` $task \in T_T$
1: `remove` $task$ `from` T_T `so that` $task \notin T_T$

3.4 Resolving Transitive Constraint Conflicts

Transitive SME or DME conflicts arise because of the transitivity of binding constraints (see Def. 3.3, Def. 3.4, and [14, 15]). Therefore, a conflict may arise when defining a RB or SB constraint on two tasks t_1 and t_2 because of pre-existing mutual exclusion constraints between on one of the tasks t_1 or t_2 and some third task t_3.

Transitive SME conflict: Figure 4a shows a *transitive SME conflict* that occurs if one tries to define a new role- or subject-binding constraint between two tasks (t_1 and t_2 in Figure 4a) that would result in a transitive binding of a third task (t_x in Figure 4a) which is already defined as statically mutual exclusive to one of the other tasks (see SME constraint between t_1 and t_x in Figure 4a). However, because binding constraints define that two task instances must be executed by the same subject/role (see Def. 3.3 and Def. 3.4), while SME tasks must *not* be executed by the same subject (see Def. 3.1) such a configuration would result in a transitive SME conflict between t_1 and t_x (see also Def. 2.6).

Resolutions to transitive SME conflicts: Figure 4a shows conflict resolutions for *transitive SME conflicts*. Such a conflict can be avoided by removing the SME constraint before defining the new binding constraint (see Resolution 2). If the conflict arises when defining a RB constraint, it can also be prevented by changing the SME into a DME constraint before defining the new RB constraint (see Resolution 3). Moreover, the conflict can be resolved by removing the pre-existing binding constraint between t_2 and t_x before defining the new binding constraint on t_1 and t_2 (see Resolution 5 for removing RB constraints and Resolution 6 for removing SB constraints). Alternatively, a transitive SME conflict can be avoided by removing the task that causes the transitive SME conflict (see Resolution 12). However, Resolution 12 will rarely be applicable in practice.

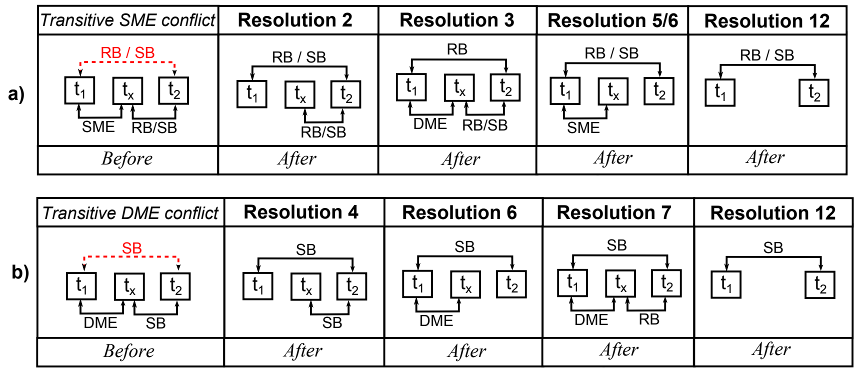

Fig. 4: Resolving transitive SME (a) and DME (b) conflicts

Transitive DME conflict: A *transitive DME conflict* arises because of the transitivity of SB constraints. Figure 4b shows a transitive DME conflict that occurs if one tries to define a new subject-binding between two tasks (t_1 and t_2 in Figure 4b) that would result in a transitive subject-binding of a third task (t_x in Figure 4b) which is already defined as dynamically mutual exclusive to one of the other tasks (see DME constraint between t_1 and t_x in Figure 4b). However, SB constraints define that two task instances must be executed by the same subject (see Def. 3.4), while DME constraints define that the corresponding task instance must *not* be executed by the same subject (see Def. 3.2). Therefore, such a configuration would result in a transitive DME conflict between t_1 and t_x (see also Def. 2.7).

Resolutions to transitive DME conflicts: Figure 4b shows resolutions for *transitive DME conflicts*. Such a conflict can be prevented by removing the DME constraint before defining the new SB constraint (see Resolution 4), or by removing the pre-existing SB constraint between t_2 and t_x before defining the new SB constraint (see Resolution 6). It can also be avoided by changing the existing SB constraint into a RB constraint before defining the new SB constraint (see Resolution 7), or by removing the conflicting task t_x (see Resolution 12).

4 Detecting and Resolving Assignment Conflicts

Assignment conflicts arise at design-time when defining new assignment relations between roles, subjects, and tasks. The algorithms defined below check the design-time consistency of a process-related RBAC model when defining a task-to-role, role-to-role, or role-to-subject assignment relation. If an assignment conflict is detected, the algorithms return the name of the respective conflict (see also [14]).

4.1 Algorithms for Detecting Assignment Conflicts

Algorithm 5 *Check if it is allowed to assign a particular task type to a particular role (task-to-role assignment).*

Name: $isT2RAssignmentAllowed$
Input: $task_x \in T_T, role_y \in R$
if $\exists\ task_y \in town(role_y) \mid task_y \in sme(task_x)$ **then return** taskAssignmentConflict
if $\exists\ role_z \in allSeniorRoles(role_y) \mid task_z \in town(role_z) \wedge$
    $task_z \in sme(task_x)$ **then return** taskAssignmentConflict
if $\exists\ s \in S \mid role_y \in rown(s) \wedge role_z \in rown(s) \wedge$
    $task_z \in town(role_z) \wedge task_z \in sme(task_x)$ **then return** roleAssignmentConflict
return true

Algorithm 6 *Check if it is allowed to define a (new) junior-role relation between two roles (role-to-role assignment).*

Name: $isR2RAssignmentAllowed$
Input: $junior, senior \in R$
if $junior == senior$ **then return** selfInheritanceConflict
if $senior \in rh^*(junior)$ **then return** cyclicInheritanceConflict
if $\exists\ task_j \in town(junior) \wedge task_s \in town(senior) \mid$
    $task_j \in sme(task_s)$ **then return** taskAssignmentConflict
if $\exists\ role_x \in allSeniorRoles(senior) \mid task_x \in town(role_x) \wedge$
    $task_j \in town(junior) \wedge task_x \in sme(task_j)$
    then return taskAssignmentConflict

if $\exists\ s \in S \mid senior \in rown(s) \wedge role_x \in rown(s)\ \wedge$
    $task_x \in town(role_x) \wedge task_j \in town(junior) \wedge task_x \in sme(task_j)$
    then return *roleAssignmentConflict*
return *true*

Algorithm 7 *Check if it is allowed to assign a particular role to a particular subject.*

Name: *isR2SAssignmentAllowed*
Input: $role_x \in R, subject \in S$
if $\exists\ role_y \in rown(subject) \mid task_y \in town(role_y)\ \wedge$
    $task_x \in town(role_x) \wedge task_y \in sme(task_x)$ **then return** *roleAssignmentConflict*
return *true*

4.2 Resolving Assignment Conflicts

Self inheritance conflict: A *self inheritance conflict* may arise when defining a new inheritance relation between roles. In particular, a role cannot be its own junior-role (see Figure 5a and [14, 15]).

Resolution to self inheritance conflicts: This conflict can be resolved by changing one of the selected roles so that the inheritance relation is defined between two different roles (see Figure 5a and Resolution 13).

Cyclic inheritance conflict: A *cyclic inheritance conflict* results from the definition of a new inheritance relation in a role-hierarchy (also called role-to-role assignment). In particular, a role-hierarchy must not include a cycle because all roles within such a cyclic inheritance relation would own the same permissions which would again render the respective part of the role-hierarchy redundant (see Figure 5b and [14, 15]).

Resolutions to cyclic inheritance conflicts: This conflict can be resolved by defining a new inheritance relation between roles which are not already part of the same role-hierarchy (see Resolution 13). In Figure 5b, Resolution 13 is applied by defining a new inheritance relation between r_x and r_y while keeping the existing inheritance relation between r_y and r_z. Moreover, the existing inheritance relation between r_y and r_z can be removed before defining the inverse inheritance relation with r_z as junior role of r_y (see Resolution 14).

Task-assignment conflict: A *task-assignment conflict* may occur if the definition of a new *tra* or junior-role relation would result in the assignment of two SME tasks to the same role (see Def. 2.8). Figure 6a depicts an example where a role r_y owns a task t_y which is defined as SME to another task t_x. Thus, assigning t_x to r_y would result in a task-assignment conflict.

Resolutions to task-assignment conflicts: To avoid the *task-assignment conflict* in Figure 6a, the conflicting SME constraint between the two task types can be removed or changed into a DME constraint (see Resolutions 2 and 3). Alternatively, task t_y can be revoked from r_y, or the conflicting task t_y can be deleted (see Resolutions 8 and 12).

Role-assignment conflict: A *role-assignment conflict* arises if a new assignment relation would authorize a subject to perform two SME tasks. Figure 6b

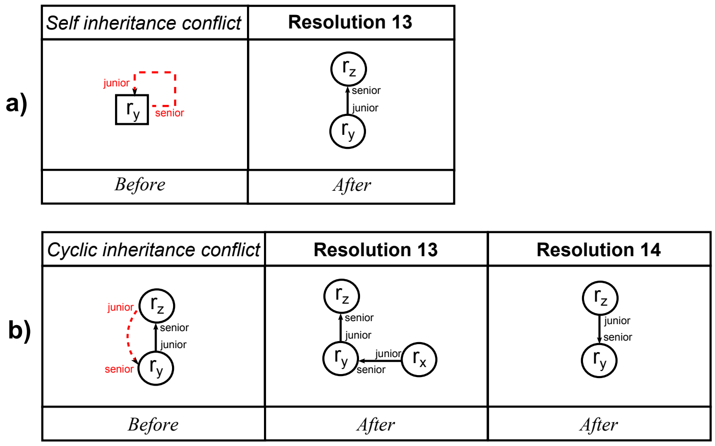

Fig. 5: Resolving self-inheritance (a) and cyclic inheritance (b) conflicts

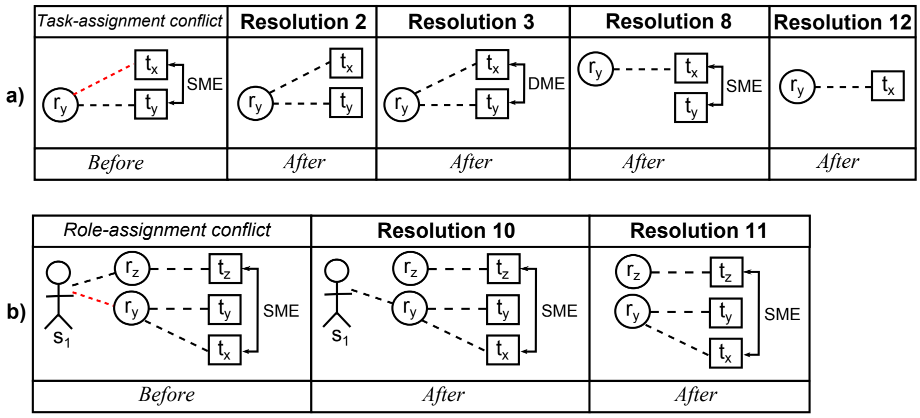

Fig. 6: Resolving task- (a) and role-assignment (b) conflicts

shows an example, where an assignment of role r_y to subject s_1 would result in a role-assignment conflict because subject s_1 would then be authorized to perform the two SME tasks t_z and t_x. Thus, such an assignment would violate the consistency requirement specified in Def. 2.9. Similarly, when defining a new junior-role or *tra* relation, we need to check for role-assignment conflicts.

Resolutions to role-assignment conflicts: To avoid a *role-assignment conflict*, the same resolutions as for task-assignment conflicts can be applied (see Resolutions 2, 3, 8, and 12). In addition, Resolution 10 can be applied by removing the conflicting assignment between r_z and s_1 (see Figure 6b). Moreover, the conflict can (theoretically) be resolved by removing the conflicting subject s_1 which is assigned to the two SME tasks (see Resolution 11).

Resolution Strategies for Assignment Conflicts

The following resolution strategies define the conflict resolutions described above with respect to the formal definitions of process-related RBAC models (see Section 2 and [14, 15]).

Resolution 13 *Select two different roles*

`Input:` $role_i \in R$
`select` $role_x \in R \mid role_i \neq role_x \wedge role_x \notin rh^*(role_i) \wedge role_i \notin rh^*(role_x)$

Resolution 14 *Remove junior-role relation*

`Input:` $role_y, role_z \in R$
`remove` $role_y$ `from` $rh^*(role_z)$ `so that` $role_y \notin rh^*(role_z)$

5 Detecting and Resolving Runtime Conflicts

Conflicts may also occur when executing process instances. Thus, runtime conflicts arise when actually enforcing constraints. In particular, mutual-exclusion and binding constraints directly impact the allocation of tasks to subjects. Below we discuss five potential conflicts when allocating a particular task instance to a certain subject. These conflicts are illustrated in Figures 7a-e, where conflicts arise when we try to allocate subject s_1 to an instance of the a task type t_x (in Figure 7 instances of t_x are labeled as t_{xi}).

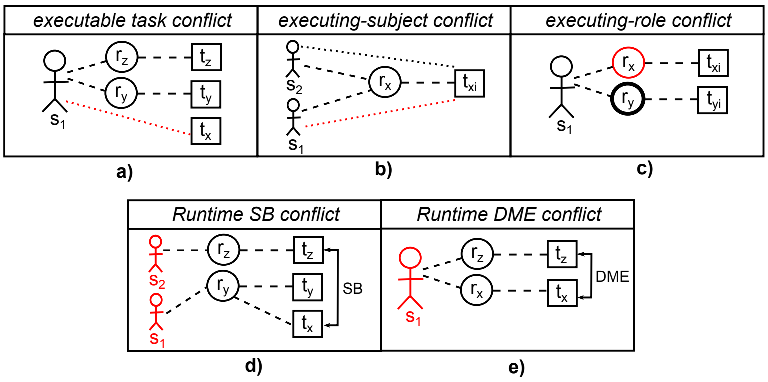

Fig. 7: Runtime conflicts

Algorithm 8 checks the runtime consistency of a process-related RBAC model when allocating a task instance to a particular subject. If one of the runtime conflicts shown in Figures 7a-e is detected, the algorithm returns the name of the respective conflict.

Algorithm 8 *Check if a particular task instance executed during a specific process instance can be allocated to a particular subject.*

`Name:` $isAllocationAllowed$
`Input:` $subject \in S, task_{type} \in T_T, process_{type} \in P_T,$
$process_{instance} \in pi(process_{type}), task_{instance} \in ti(task_{type}, process_{instance})$
if $task_{type} \notin executableTasks(subject)$ **then return** `executableTaskConflict`
if $es(task_{instance}) \neq \emptyset$ **then return** `executingSubjectConflict`
if $er(task_{instance}) \neq \emptyset \wedge er(task_{instance}) \neq ar(subject)$
  then return `executingRoleConflict`
if $\exists\ type_x \in allSubjectBindings(task_{type})\ |$
  $type_x \notin executableTasks(subject)$ **then return** `runtimeSBConflict`
if $\exists\ instance_y \in ti(type_y, process_{instance})\ |$
  $type_y \in dme(task_{type}) \wedge es(instance_y) == subject$
  then return `runtimeDMEConflict`
return `true`

Executable task conflict: An *executable task conflict* arises if the selected subject is not allowed to execute the task type the corresponding task instance was instantiated from. If subject s_1 is not allowed to execute instances of task t_x (see Figure 7a), the respective task instance must not be allocated to s_1.

Resolutions to executable task conflicts: An *executable task conflict* can be resolved by allocating an executing subject that actually owns the permission to perform the respective task (see Resolution 15). Alternatively, one may change the *rsa* or the *tra* relations so that s_1 is allowed to execute t_x.

Executing-subject conflict: An *executing-subject conflict* arises if the allocation is not possible, because the respective task instance already has been allocated to another subject. For example, in Figure 7b the task instance t_{xi} already has an executing subject s_2 and thus cannot be allocated to s_1.

Resolution to executing-subject conflicts: An *executing-subject conflict* can only be resolved by first deallocating the executing-subject before the respective task instance can be reallocated to another subject that is allowed to perform the respective task (see Resolution 16 and Algorithm 8).

Executing-role conflict: An *executing role conflict* visualized in Figure 7c occurs if a task instance already has an executing role, but this executing role is not the active role of the designated executing-subject.

Resolution to executing-role conflicts: An *executing-role conflict* can be resolved by changing the active role of the subject to the executing-role of the respective task instance (see Resolution 17).

Runtime SB conflict: Figure 7d shows an example of a *runtime SB conflict* that occurs when we try to allocate s_1 to an instance of t_x. In particular, we need to check if some task type t_z exists that has a SB relation to t_x but cannot be executed by s_1. Such an allocation violates the consistency requirement specified in Def. 3.4, because subject-bound tasks must have the same executing subject. Thus, a subject can only be allocated if it owns the right to perform the corresponding task type as well as all subject-bound tasks.

Resolutions to runtime SB conflicts: This conflict can be resolved by removing the SB constraint (see Resolution 6). Moreover, the *tra* relation for the subject-bound task or the *rsa* relation for one of the roles owning this task can be changed so that the designated executing-subject is allowed to perform the tasks that are connected via a (transitive) SB constraint. Furthermore, one of the subject-bound tasks can be removed in order to resolve the SB conflict (see Resolution 12), or the executing-subject can be changed (see Resolution 15).

Runtime DME conflict: In the example from Figure 7e, a *runtime DME conflict* would occur if we try to allocate s_1 to an instance of t_z *and* to an instance of t_x in the same process instance. This is because a DME constraint defines that in the same process instance the instances of two DME task types must not be performed by the same subject (see Def. 3.2).

Resolutions to runtime DME conflicts: A *runtine DME conflict* is prevented by either removing the DME constraint, by removing one of the DME tasks, or by changing the executing-subject (see Resolutions 4, 12, 16 and 15).

Resolution Strategies for Runtime Conflicts

The following resolution strategies define the conflict resolutions presented above with respect to the formal definitions of process-related RBAC models.

Resolution 15 *Select a subject that is allowed to perform the respective task*

Input: $task \in T_T, role \in R$
1: **select** $subject \in S \mid role \in rown(subject) \wedge task \in town(role)$

Resolution 16 *Deallocate a task instance*

Input: $task_i \in T_I$
1: **set** $es(task_i) = \emptyset$ **and** $er(task_i) = \emptyset$

Resolution 17 *Change the executing-subject's active role to the executing-role of the respective task*

Input: $task_i \in T_I, subject \in S \mid es(task_i) == subject$
1: **if** $er(task_i) \neq ar(subject)$ **then set** $ar(subject) = er(task_i)$

6 Related Work

Sloman and Moffett [9, 10, 13] were among the first to analyze and categorize conflicts between different types of policies. They also presented informal strategies for resolving these conflicts. In [1], Ahn and Sandhu presented the RCL 2000 language for the specification of role-based authorization constraints. They also show how SOD constraints can be expressed in RCL 2000 and discuss different types of conflicts that may result from constraints specified via RCL 2000. Bertino et al. [3] present a language to express SOD constraints as clauses in logic programs. Moreover, they present corresponding algorithms that check the consistency of such constraints. Thereby they ensure that all tasks within a workflow are performed by predefined users/roles only. In [4], Botha and Eloff present an approach called conflicting entities administration paradigm. In particular, they discuss possible conflicts of static and dynamic SOD constraints in a workflow environment and share a number of lessons learned from the implementation of a prototype system. Schaad [12] discusses the detection of conflicts between SOD constraints in a role-based delegation model. Schaad follows a rule-based, declarative approach by using the Prolog language as an executable specification language.

Wang et al. [18] define algorithms for the detection of conflicts between access control policies. Similarly, in [2], an approach for the formalization of policy rules is proposed and algorithms for policy conflict resolutions are derived. Yet, both approaches do not consider conflicts resulting from SOD or BOD constraints. Tan et al. [16] define a model for constrained workflow systems, including SOD and BOD constraints. They discuss different issues concerning the consistency of such constraints and provide a set of formal consistency rules that guarantee the definition of a sound constrained workflow specification. In [6] Ferraiolo et al. present RBAC/Web, a model and implementation for RBAC in Web servers. They also discuss the inheritance and resulting consistency issues of SOD constraints in role-hierarchies. Jaeger et al. [8] present a formal model for constraint conflicts and define properties for resolving these conflicts. They applied metrics for resolving Biba integrity violations in an SELinux example policy.

7 Conclusion

In this paper, we discussed resolution strategies for conflicts of process-related mutual-exclusion and binding constraints. Because of the countless configurations that could cause conflicts, we chose to discuss frequently occurring conflict types which group similar conflicts. In the same way, we described corresponding types of resolution strategies. If a certain resolution strategy is actually applicable to a specific real-world conflict can, however, only be decided by the corresponding process modeler or security engineer.

Note that in our approach, conflicts are detected and resolved before causing an inconsistent RBAC configuration. In other words, the formal consistency requirements for static and dynamic correctness of our process-related RBAC models must hold at any time and therefore prevent the definition of inconsistent RBAC models. The application of the algorithms and resolution strategies described in this paper can help process modelers and security engineers to identify resolution options for design-time and runtime conflicts in process-related RBAC models.

References

1. G. Ahn and R. Sandhu. Role-based Authorization Constraints Specification. *ACM Transactions on Information and System Security (TISSEC)*, 3(4), November 2000.

2. J. Baliosian and J. Serrat. Finite State Transducers for Policy Evaluation and Conflict Resolution. In *Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, June 2004.
3. 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 (TISSEC)*, 2(1), 1999.
4. R. A. Botha and J. H. Eloff. Separation of duties for access control enforcement in workflow environments. *IBM Systems Journal*, 40(3), 2001.
5. F. Casati, S. Castano, and M. Fugini. Managing Workflow Authorization Constraints through Active Database Technology. *Information Systems Frontiers*, 3(3), 2001.
6. D. Ferraiolo, J. Barkley, and D. Kuhn. A Role-Based Access Control Model and Reference Implementation within a Corporate Intranet. *ACM Transactions on Information and System Security (TISSEC)*, 2(1), February 1999.
7. D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control, Second Edition*. Artech House, 2007.
8. T. Jaeger, R. Sailer, and X. Zhang. Resolving constraint conflicts. In *Proc. of the Ninth ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2004.
9. J. D. Moffett and M. S. Sloman. Policy Hierarchies for Distributed Systems Management. *IEEE Journal on Selected Areas in Communications*, 11(9), 1993.
10. J. D. Moffett and M. S. Sloman. Policy Conflict Analysis in Distributed System Management. *Journal of Organizational Computing*, 4(1), 1994.
11. H. F. Ravi Sandhu, Edward Coyne and C. Youman. Role-based access control models. *IEEE Computer*, 29(2), 1996.
12. A. Schaad. Detecting Conflicts in a Role-Based Delegation Model. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC)*, December 2001.
13. M. S. Sloman. Policy Driven Management for Distributed Systems. *Journal of Network and Systems Management*, 2(4), 1994.
14. M. Strembeck and J. Mendling. Generic Algorithms for Consistency Checking of Mutual-Exclusion and Binding Constraints in a Business Process Context. In *Proc. of the 18th International Conference on Cooperative Information Systems (CoopIS)*, volume 6426 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, October 2010.
15. M. Strembeck and J. Mendling. Modeling Process-related RBAC Models with Extended UML Activity Models. *Information and Software Technology*, 53(5), 2011.
16. K. Tan, J. Crampton, and C. A. Gunter. The Consistency of Task-Based Authorization Constraints in Workflow Systems. In *Proceedings of the 17th IEEE workshop on Computer Security Foundations*, June 2004.
17. J. Wainer, P. Barthelmess, P. Barthelmess, and A. Kumar. W-RBAC - A workflow security model incorporating controlled overriding of constraints. *International Journal of Cooperative Information Systems (IJCIS)*, 12(4), 2003.
18. H. Wang, L. Sun, and V. Varadharajan. Purpose-based access control policies and conflicting analysis. In *Security and Privacy - Silver Linings in the Cloud*, volume 330 of *IFIP Advances in Information and Communication Technology*. 2010.
19. J. Warner and V. Atluri. Inter-instance authorization constraints for secure workflow management. In *Proc. of the Eleventh ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2006.