An Approach for Consistent Delegation in Process-Aware Information Systems

Sigrid Schefer-Wenzl, Mark Strembeck, and Anne Baumgrass

Institute for Information Systems and New Media
Vienna University of Economics and Business (WU Vienna), Austria
{firstname.lastname}@wu.ac.at

Abstract. Delegation is an important concept to increase flexibility in authorization and obligation management. Due to the complexity of potential delegation relations, there is a strong need to systematically check the consistency of all delegation assignments. In this paper, we discuss the detection of delegation conflicts based on the formal definitions of a model that supports the delegation of roles, tasks, and duties in a business process context.

Key words: Access Control; Business Processes; Delegation; RBAC;

1 Introduction

A business process includes a set of tasks which are performed to reach certain corporate goals. To support the secure execution of a business process, subjects participating in a particular process instance must own the permissions that are needed to execute the corresponding tasks (see, e.g., [17]). In recent years, Role-Based Access Control (RBAC) [7, 9] has developed into a de facto standard for access control. In RBAC, roles are used to model different job positions and responsibilities within an organization and/or information system. Permissions are assigned to roles according to the tasks each role has to accomplish. The roles are then assigned to human users according to their respective work profile [15]. Roles are also used as an abstract concept for delegation [5, 18] or for the assignment of duties defined via obligations [11, 21].

Authorization policies define a subject's permissions, while obligation policies define a subject's duties (see, e.g., [3]). Delegation provides a mechanism to increase flexibility in authorization and obligation management. In essence, a subject can delegate tasks, duties, or roles to another subject [11]. Subsequently, the subject receiving the delegation (the delegatee) will act on behalf of the delegating subject (the delegator). While delegation authorizes subjects to perform tasks they usually are not allowed to perform, authorization constraints, such as mutual-exclusion (ME) and binding constraints, restrict which subject is allowed to execute a particular task (see, e.g., [16, 17, 19]). In process-aware information systems, ME constraints enforce conflict of interest policies. Conflict of interest arises as a result of the simultaneous assignment of two mutually exclusive tasks or roles to the same subject. In contrast to ME constraints, binding constraints define that bound tasks must be executed by the same subject or role. The immanent complexity of delegations is a central problem in process-aware information systems (see, e.g., [4, 10]). Thus, when delegating tasks, roles, or duties, design-time and run-time checks need to ensure the consistency of the corresponding RBAC model including mutual-exclusion and binding constraints. In [13, 16], we provide a set of algorithms that check and ensure the consistency of process-related RBAC models without addressing delegation aspects.

The main contribution of this paper is the consideration of delegations when checking and ensuring the consistency of process-related RBAC models. In particular, we integrate the formal definitions of our delegation model into process-related RBAC models [17]. These definitions are based on several existing, well-known delegation models and are the basis for the algorithms presented in this paper. The algorithms systematically detect potential conflicts when delegating roles, tasks, and duties at design- and run-time. For this purpose, we take the conflicts identified in [13, 16] as a starting point.

The remainder of this paper is structured as follows. In Section 2, we introduce relevant terms and present the formal definitions of a process-related RBAC delegation model. Section 3 provides algorithms to detect potential delegation conflicts to ensure the consistency of a process-related RBAC delegation model. Section 4 discusses related work and Section 5 concludes the paper.

2 Process-related RBAC Delegation Models

In our process-related RBAC delegation model, roles, tasks, and associated duties are delegatable. Each *task* in an IT-supported workflow (such as negotiating a contract) is typically associated with certain access operations (e.g., to sign the contract). Thus, a subject participating in a workflow must be authorized to perform the tasks needed to complete the process (see, e.g., [17]). In organizational contexts, tasks can be associated with duties. Each *duty* defines an action that must be performed by a certain subject in order to comply with legal or organizational regulations (see, e.g., [3, 12]). A *subject* may either be a human user or a software-based system. In RBAC, a *role* is a subject abstraction containing the tasks and duties of a certain subject-type.

In the context of RBAC, several delegation approaches use the concept of *delegation roles* (see, e.g., [8, 14, 20]). In our delegation model, a delegation role is created by the *delegator* and comprises a *set of delegated tasks and duties* (similar to [20]). Hereby, each duty is associated with a certain task [12]. A delegator can delegate all or a subset of his/her delegatable tasks, duties, or roles by assigning them to a delegation role. Subsequently, delegation roles are assigned to delegatees and can either be defined for temporary or for permanent delegation (see, e.g., [2, 20]). By default, delegation roles are permanent which means they authorize the delegatee to perform the delegated tasks and duties in all instances of a process. In contrast, a temporary delegation role authorizes the delegatee to perform the delegated tasks and duties only in particular process instances.

Moreover, we support single- and multi-step delegation (see, e.g., [2, 18]). In single-step delegation, a delegated task, duty, or role cannot be delegated further by the delegatee. Multi-step delegation allows a delegatee to further delegate the delegated tasks, duties, and roles. In general, delegation roles and all assignments to delegation roles are managed by the delegating subject. All other roles are called *regular roles* and are usually managed by the organization's security officer. Fig. 1 shows a class diagramm that depicts the elements of the RBAC delegation model (see Definition 1).

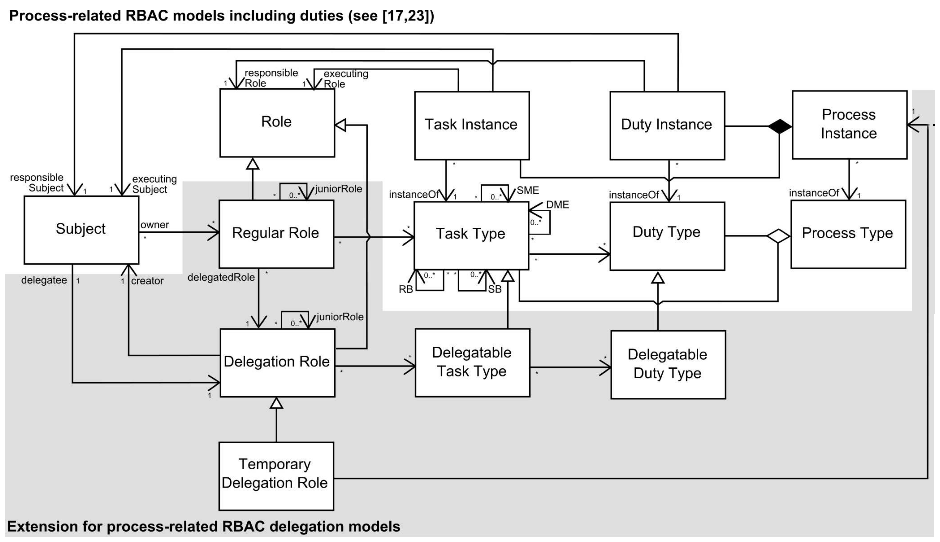

Fig. 1. Main elements of process-related RBAC delegation models

Furthermore, different kinds of authorization constraints can be defined to restrict which subjects are allowed to execute a particular task or duty (see, e.g., [17, 19]). In this paper, we focus on static mutual exclusion (SME), dynamic mutual exclusion (DME), subject-binding (SB), and role-binding (RB) constraints. A SME constraint defines that two statically mutual exclusive tasks must never be *assigned* to the same subject. In turn, a DME constraint defines that two dynamically mutual exclusive tasks must never be *executed* by the same subject in the *same process instance*. A SB constraint defines that two bound tasks must be performed by the same individual within the same process instance. A RB constraint defines that bound tasks must be performed by members of the same role, but not necessarily by the same individual. To ensure proper delegation, authorization constraints must be considered when delegating tasks, duties, and roles (see Section 3). For example, a delegation assignment must not authorize the delegatee to perform two SME tasks.

Definition 1 formally specifies the essential elements and their basic interrelations in a metamodel for process-related RBAC delegation models (see Fig. 1).

Definition 1. ***(Process-Related RBAC Delegation Model).*** *Let PRDM = (E,Q,D,DL) be a Process-Related RBAC Delegation Model, where E refers to the* *pairwise disjoint sets of the metamodel, Q to mappings that establish relationships, D to binding and mutual-exclusion constraints, and DL to mappings for delegation policies.*

The sets E of the Process-Related RBAC Delegation Model are:

– *An element of S is called* Subject*.* $S \neq \emptyset$.
– *An element of R is called* Role*.* $R \neq \emptyset$.
– *An element of RR is called* Regular Role*.* $RR \subseteq R$.
– *An element of DR is called* Delegation Role*.* $DR \subseteq R$
– *An element of DRT is called* Temporary Delegation Role*.* $DRT \subseteq DR$.
– *An element of P_T is called* Process Type*.* $P_T \neq \emptyset$.
– *An element of P_I is called* Process Instance*.* $P_I \neq \emptyset$.
– *An element of T_T is called* Task Type*.* $T_T \neq \emptyset$.
– *An element of DT_T is called* Delegatable Task Type*.* $DT_T \subseteq T_T$.
– *An element of T_I is called* Task Instance*.*
– *An element of DU_T is called* Duty Type*.*
– *An element of DDU_T is called* Delegatable Duty Type*.* $DDU_T \subseteq DU_T$.
– *An element of DU_I is called* Duty Instance*.*

For the mappings of the Process-Related RBAC Model *(Q,D) see [17]. Below, we define additional mappings for delegation:* $DL = rrh \cup drh \cup creator \cup drpi \cup trra \cup trdel \cup dta \cup rrsa \cup rsdel \cup dui \cup res \cup rer$ *($\mathcal{P}$ refers to the power set):*

1. Roles R are partitioned into regular roles and delegation roles. In RBAC, roles can be arranged in a role-hierarchy, where senior-roles inherit the permissions from their junior-roles. To avoid invalid permission inheritance, the regular role-hierarchy consists of regular roles only. If a model uses process-related RBAC delegation, this mapping replaces the role-hierarchy mapping rh in [17]: *The mapping $rrh : RR \mapsto \mathcal{P}(RR)$ is called* ***regular role-hierarchy****. For $rrh(r_s) = RR_j$, we call $r_s \in RR$* senior regular role *and $RR_j \subseteq RR$ the* set of direct junior regular roles*. The transitive closure rrh^* defines the inheritance in the role-hierarchy such that $rrh^*(r_s) = RR_{j*}$ includes all direct and transitive regular junior-roles that the senior-role r_s inherits from. The regular role-hierarchy is cycle-free, i.e. for each $r \in RR : rrh^*(r) \cap r = \emptyset$.*
2. Delegation roles can be arranged in a delegation role-hierarchy via role-to-role delegation. Note that each delegation role may have junior regular roles or junior delegation roles (see, e.g., [20]). However, delegation roles must not have senior regular roles to avoid invalid permission inheritance in the regular role hierarchy: *The mapping $drh : DR \mapsto \mathcal{P}(R)$ is called* ***delegation role-hierarchy****. For $drh(dr_s) = R_j$, we call $dr_s \in DR$* senior delegation role *and $R_j \subseteq R$ the* set of direct junior-roles*. The transitive closure drh^* defines the inheritance in the role-hierarchy such that $drh^*(dr_s) = R_{j*}$ includes all direct and transitive junior-roles that the senior-role dr_s inherits from. The delegation role-hierarchy is cycle-free, i.e. for each $r \in R : drh^*(r) \cap r = \emptyset$.*
3. Each subject can create an arbitrary number of delegation roles. Subsequently, the creator will act as the delegator of its delegation roles: *The*

mapping $creator(dr) : DR \mapsto S$ *is called* ***delegation role creator****. For* $creator(dr) = s$*, we call* $dr \in DR$ delegation role *and* $s \in S$ *the* creator of this delegation role*.*

4. Each delegation role can be specified either for permanent or for temporary delegation. By default, a delegation role is permanent and is valid for all process types. In case of temporary delegation, a temporary delegation role is only valid for particular process instances: *The mapping* $drpi : DRT \mapsto \mathcal{P}(P_I)$ *is called* ***delegation role-to-process assignment****. For* $drpi(drt) = P_{drt}$*, we call* $drt \in DRT$ temporary delegation role*, and* $P_{drt} \subseteq P_I$ *the* set of process instances*.*

5. Task types are assigned to regular roles to define the permissions of the corresponding role. If a model uses process-related RBAC delegation, this mapping replaces the task-to-role assignment mapping tra in [17]: *The mapping* $trra : RR \mapsto \mathcal{P}(T_T)$ *is called* ***task-to-regular role assignment****. For* $trra(r) = T_r$*, we call* $r \in RR$ regular role *and* $T_r \subseteq T_T$ *is called the* set of tasks assigned to r*. The mapping* $trra^{-1} : T_T \mapsto \mathcal{P}(RR)$ *returns the set of regular roles a particular task is assigned to.*

6. Task types can be defined as being delegatable. Only delegatable tasks can be assigned to delegation roles. Thus, a subject can delegate a task by assigning this task to a delegation role: *The mapping* $trdel : DR \mapsto \mathcal{P}(DT_T)$ *is called* ***task-to-role delegation****. For* $trdel(dr) = DT_{dr}$*, we call* $dr \in DR$ delegation role *and* $DT_{dr} \subseteq DT_T$ *is called the* set of delegated tasks assigned to dr*. The mapping* $trdel^{-1} : DT_T \mapsto \mathcal{P}(DR)$ *returns the set of delegation roles a particular delegatable task is assigned to.*

7. Further, $trra$ and $trdel$ imply a mapping ***task ownership*** $town : R \mapsto \mathcal{P}(T_T)$ to determine all tasks that are assigned to a particular role. If a model uses process-related RBAC delegation, this mapping replaces the *town*-mapping from [17]: *For each* $r \in R$*, the tasks inherited from its junior roles are included, i.e.* $town(r) = town_{rrh}(r) \cup town_{drh}(r)$*,* where $town_{rrh}(r) = \bigcup_{r_{inh} \in rrh^*(r)} trra(r_{inh}) \cup trra(r)$ and $town_{drh}(r) = \bigcup_{r_{inh} \in drh^*(r)} trdel(r_{inh}) \cup trdel(r)$*.*

8. A duty defines an action that must be performed by a certain subject. In a business process context, each duty is associated with a task [12]: *The mapping* $dta : T_T \mapsto \mathcal{P}(DU_T)$ *is called* ***duty-to-task assignment****. For* $dta(t) = DU_x$*, we call* $t \in T_T$ task type *and* $DU_x \subseteq DU_T$ *is called the* set of duties assigned to this task type*.*

9. Delegatable tasks can only be delegated, if all associated duties are also delegatable: $\forall t_x \in trdel(dr) : \forall du \in dta(t_x) : du \in DDU_T$

10. Regular roles are assigned to subjects. Thereby, subjects acquire the rights to execute the corresponding tasks and duties. If a model uses process-related RBAC delegation, this mapping replaces the role-to-subject assignment mapping rsa in [17]: *The mapping* $rrsa : S \mapsto \mathcal{P}(RR)$ *is called* ***regular role-to-subject assignment****. For* $rrsa(s) = RR_s$*, we call* $s \in S$ subject *and* $RR_s \in RR$ the set of regular roles owned by s*. The mapping* $rrsa^{-1} : RR \mapsto \mathcal{P}(S)$ *returns all subjects assigned to a regular role.*

11. Delegation roles are assigned to delegatees who are subsequently authorized and responsible to perform the corresponding delegated tasks and duties: *The mapping $rsdel : S \mapsto \mathcal{P}(DR)$ is called **role-to-subject delegation**. For $rsdel(s) = DR_s$, we call $s \in S$* delegatee *and $DR_s \in DR$* the set of delegation roles owned by s*. The mapping $rsdel^{-1} : DR \mapsto \mathcal{P}(S)$ returns all delegatees assigned to a delegation role.*

12. Further, $rrsa$ and $rsdel$ imply a mapping ***role ownership*** $rown : S \mapsto \mathcal{P}(R)$ to determine all roles that are assigned to a particular subject. If a model uses process-related RBAC delegation, this mapping replaces the $rown$-mapping from [17]: *For each $s \in S$, all inherited roles are included, i.e. $rown(s) = rown_{rrh}(s) \cup rown_{drh}(s)$,* where $rown_{rrh}(s) = \bigcup_{r \in rrsa(s)} rrh^*(r) \cup rrsa(s)$ and $rown_{drh}(s) = \bigcup_{r \in rsdel(s)} drh^*(r) \ \cup \ rsdel(s)$.

13. For each task type, we can create an arbitrary number of respective task instances via the ***task instantiation*** mapping ti [17]. Similarly, each duty type is instantiated by a number of duty instances: *The mapping $dui : (DU_T \times P_I) \mapsto \mathcal{P}(DU_I)$ is called **duty instantiation**. For $dui(du_T, p_I) = DU_i$, we call $DU_i \subseteq DU_I$ set of* duty instances*, $du_T \in DU_T$ is called* duty type *and $p_I \in P_I$ is called* process instance*.*

14. The ***executing-subject*** mapping es returns the subject executing a particular task instance [17]. The subject responsible for discharging a duty is called the responsible subject of this duty instance: *The mapping $res : DU_I \mapsto S$ is called **responsible-subject** mapping. For $res(du) = s$, we call $s \in S$ the* responsible subject *and $du \in DU_I$ is called* duty instance*.*

15. Within the same process instance, a subject executing a task is also responsible for discharging all associated duties: $\forall du \in dta(t_1), p_i \in P_I : \forall t_x \in ti(t_1, p_i), du_x \in dui(du, p_i) : es(t_x) = res(du_x)$

16. The ***executing-role*** mapping er returns the role executing a particular task instance [17]. The ***active-role*** mapping ar returns the role a subject has currently activated [16]. The role being responsible for actually discharging a certain duty instance is called the responsible-role: *The mapping $rer : DU_I \mapsto R$ is called **responsible-role** mapping. For $rer(du) = r$, we call $r \in R$ the* responsible role *and $du \in DU_I$ is called* discharged duty instance*.*

17. Further, we allow the definition of subject-binding, role-binding, static mutual exclusion, and dynamic mutual exclusion constraints on task types. Related consistency requirements are specified in [17]: *The mapping $sb : T_T \mapsto \mathcal{P}(T_T)$ is called **subject-binding**. For $sb(t_1) = T_{sb}$, we call t_1 the* subject binding task *and $T_{sb} \subseteq T_T$ the set of* subject-bound tasks*. The mapping $rb : T_T \mapsto \mathcal{P}(T_T)$ is called **role-binding**. For $rb(t_1) = T_{rb}$, we call t_1 the* role binding task *and $T_{rb} \subseteq T_T$ the set of* role-bound tasks*. The mapping $sme : T_T \mapsto \mathcal{P}(T_T)$ is called **static mutual exclusion**. For $sme(t_1) = T_{sme}$ with $T_{sme} \subseteq T_T$, we call each pair t_1 and $t_x \in T_{sme}$* statically mutual exclusive tasks*. The mapping $dme : T_T \mapsto \mathcal{P}(T_T)$ is called **dynamic mutual exclusion**. For $dme(t_1) = T_{dme}$ with $T_{dme} \subseteq T_T$, we call each pair t_1 and $t_x \in T_{dme}$* dynamically mutual exclusive tasks*.*

3 Detecting Delegation Conflicts

When delegating tasks, duties, or roles several conflicts may occur. In [13, 16], we detect conflicts of process-related RBAC models at design-time and run-time. In this paper, we provide additional algorithms to detect delegation conflicts. Algorithms 1–3 check the design-time consistency of a process-related RBAC delegation model when defining a task-to-role, role-to-role, or role-to-subject delegation relation. Algorithm 4 checks the consistency of a process-related RBAC delegation model at run-time. All other conflicts that can potentially occur at design- or run-time are addressed by the algorithms presented in [13, 16].

Algorithm 1 *Check if it is allowed to delegate a task type to a delegation role.*

```
Input: $task_x \in T_T, drole_y \in DR, delegator \in S$
1: if $delegator \neq creator(drole_y)$ then return false
2: if $task_x \notin DT_T$ then return false
3: if $\exists\ duty_x \in dta(task_x) \mid duty_x \notin DDU_T$ then return false
4: if $\nexists\ r \in rown(delegator) \mid task_x \in town(r) \ \wedge\ r \in RR$ then return false
5: if $\exists\ task_y \in town(drole_y) \mid task_y \in sme(task_x)$ then return false
6: if $\exists\ role_z \in allSeniorRoles(drole_y) \mid task_z \in town(role_z) \wedge$
7:     $task_z \in sme(task_x)$ then return false
8: if $\exists\ s \in S \mid drole_y \in rown(s) \ \wedge\ role_z \in rown(s) \ \wedge$
9:     $task_z \in town(role_z) \ \wedge\ task_z \in sme(task_x)$ then return false
10: if $\exists\ task_y \in sb(task_x) \mid task_y \notin DT_T$ then return false
11: if $\exists\ task_y \in rb(task_x) \mid task_y \notin DT_T$ then return false
12: if $\exists\ task_y \in sb(task_x) \mid duty_y \in dta(task_y) \wedge duty_y \notin DDU_T$ then return false
13: if $\exists\ task_y \in rb(task_x) \mid duty_y \in dta(task_y) \wedge duty_y \notin DDU_T$ then return false
14: return true
```

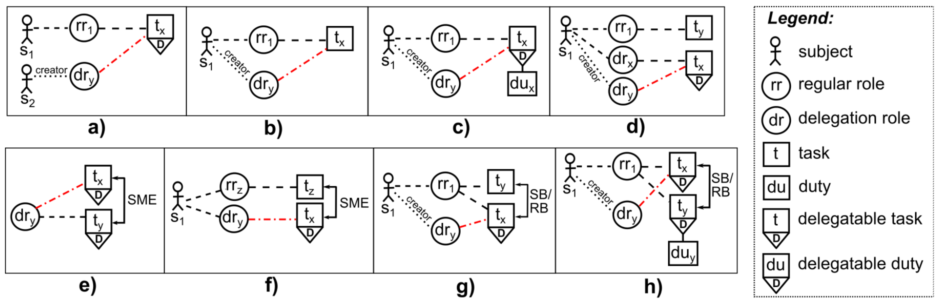

Fig. 2. Delegation conflicts

Only the creator of a delegation role can delegate to it and assign delegatees. Thus, Algorithm 1, line 1 returns false if a subject tries to delegate to a delegation role which he/she has not created. For example, in Fig. 2a, subject s_1 tries to delegate task t_x to delegation role dr_y. Task t_x is delegatable which is visualized in Fig. 2 by a triangle attached to the task-symbol including the letter D. However, s_1 is not the creator of dr_y and thus s_1 cannot delegate to it.

Next, Algorithm 1, line 2 checks if a subject tries to delegate a task which is not delegatable. In Fig. 2b, task t_x cannot be delegated to delegation role

dr_y, because t_x is not delegatable. Afterwards, line 3 checks if a subject tries to delegate a task which is associated with a non-delegatable duty. Duties always need to be discharged by the subject executing the corresponding task. Thus, if a task is delegated, the corresponding duty also needs to be delegatable. In Fig. 2c, task t_x can not be delegated to delegation role dr_y, because the duty du_x associated to t_x is not delegatable.

Algorithm 1, line 4 returns false if a subject tries to delegate a task which he/she is not assigned to via its (regular) role ownership assignments. If single-step delegation is preferred, the subject can only delegate tasks and duties which he/she owns directly or transitively via a regular role. This is because single-step delegation does not allow to further delegate a delegated task. In Fig. 2d, subject s_1 tries to delegate task t_x to its delegation role dr_y. However, none of the *regular* roles owned by s_1 is assigned to t_x. Thus, s_1 cannot delegate t_x to dr_y. In case of multi-step delegation, a subject can delegate all of his/her tasks and duties. For this purpose, we need to change the condition $r \in RR$ in Algorithm 1, line 4 to $r \in R$. Subsequently, a subject can delegate tasks and duties which he/she owns directly or transitively via its regular or delegation role memberships.

Moreover, it is not possible to delegate a task if this delegation would result in the assignment of two SME tasks to the same role or subject (see Algorithm 1, lines 5-9). Fig. 2e depicts an example where a delegation role dr_y owns a task t_y which is defined as SME to another task t_x. Thus, t_x cannot be delegated to dr_y. Otherwise, dr_y would subsequently own two SME tasks. Fig. 2f shows an example, where the delegation of task t_x to the delegation role dr_y is not allowed because s_1 would then be authorized to perform the two SME tasks t_z and t_x.

If a subject tries to delegate a task which has a subject-binding to one or more non-delegatable task(s), Algorithm 1, line 10 returns false. This is because subject-bound tasks always have to be performed by the same subject. Thus, if a task is delegated, all subject-bound tasks also need to be assigned to the same delegation role. Otherwise, the SB constraint cannot be fulfilled. In Fig. 2g, a SB constraint is defined on t_x and t_y. Therefore, the subject performing t_x also has to perform t_y. When delegating t_x to dr_y Algorithm 1 returns false, because t_y is not delegatable. However, to fulfill the SB constraint, both tasks need to be delegated to dr_y. Similarly, Algorithm 1, line 11 returns false if a subject tries to delegate a task which has a role-binding to one or more non-delegatable task(s). Thus, if a task is delegated, all role-bound tasks also need to be assigned to the same delegation role.

Furthermore, a subject cannot delegate a task which has a subject-binding to other tasks, if one of the subject-bound tasks is associated with a non-delegatable duty. In Fig. 2h, a SB constraint is defined on t_x and t_y. Moreover, t_y is associated with a duty du_y. If subject s_1 tries to delegate t_x to dr_y, it also has to delegate all subject-bound tasks and associated duties. In this example, du_y is not delegatable. Thus, Algorithm 1, line 12 returns false. Similarly, if a subject tries to delegate a task which has a role-binding to other tasks, Algorithm 1, line 13 returns false, if one of the role-bound tasks is associated with a non-delegatable duty. If none of the above checks returns false, Algorithm 1 finally reaches line

14 and returns true – meaning that it is allowed to delegate a particular task type to a certain delegation role.

Algorithm 2 *Check if it is allowed to delegate a role to a delegation role.*

```
Input: $junior \in R, senior \in DR, delegator \in S$
1: if $delegator \neq creator(senior)$ then return $false$
2: if $junior \notin rown(delegator)$ then return $false$
3: if $junior == senior$ then return $false$
4: if $\exists\ task_x \in town(junior) \mid task_x \notin DT$ then return $false$
5: if $\exists\ task_x \in town(junior) \mid duty_x \in dta(task_x) \wedge$
6:     $duty_x \notin DDU_T$ then return $false$
7: if $junior \in DR$ then $\exists\ r \in rown(delegator) \mid task_x \in town(junior) \wedge$
8:     $task_x \in town(r) \ \wedge \ r \in RR$ else return $false$
9: if $senior \in drh^*(junior)$ then return $false$
10: if $\exists\ task_j \in town(junior) \mid \ task_s \in town(senior) \wedge$
11:     $task_j \in sme(task_s)$ then return $false$
12: if $\exists\ role_x \in allSeniorRoles(senior) \mid task_x \in town(role_x) \wedge$
13:     $task_j \in town(junior) \ \wedge \ task_x \in sme(task_j)$ then return $false$
14: if $\exists\ s \in S \mid senior \in rown(s) \ \wedge \ role_x \in rown(s) \ \wedge \ task_x \in town(role_x) \wedge$
15:     $task_j \in town(junior) \ \wedge \ task_x \in sme(task_j)$ then return $false$
16: if $\exists\ task_x \in town(junior) \mid task_y \in sb(task_x) \wedge task_y \notin DT_T$ then return $false$
17: if $\exists\ task_x \in town(junior) \mid task_y \in sb(task_x) \wedge duty_y \in dta(task_y) \wedge$
18:     $duty_y \notin DDU_T$ then return $false$
19: return $true$
```

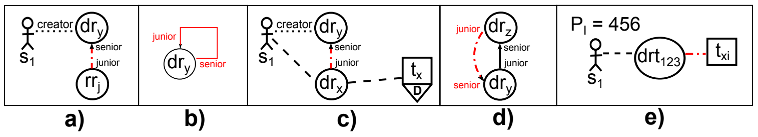

Fig. 3. Delegation conflicts

Algorithm 2 first checks if the delegator of a role is the creator of the corresponding delegation role. Subsequently, line 2 checks if a subject tries to delegate a role which he/she is not assigned to. In Fig. 3a, subject s_1 tries to delegate the regular role rr_j to its delegation role dr_y by assigning rr_j as junior-role of dr_y. However, s_1 is not assigned to rr_j and thus s_1 cannot delegate rr_j.

Next, Algorithm 2, line 3 returns false when delegating a role to itself. In general, a role cannot be its own junior-role (see Fig. 3b and [16, 17]). Algorithm 2, line 4 checks if the role which is to be delegated only contains delegatable tasks. Similarly, lines 5-6 check if all duties associated to the tasks of the corresponding role are delegatable. If either tasks or duties assigned to the role are not delegatable, Algorithm 2 returns false. Algorithm 2, lines 7-8 check if a subject tries to delegate a delegation role owning a task which the delegator is not assigned to via its *regular* role memberships (single-step delegation). Thus, a subject can only delegate tasks and duties which he/she owns directly or transitively via a regular role (see Figure 3c). In case of multi-step delegation, we can omit this check. Moreover, a role-hierarchy must not include a cycle because all roles within such a cyclic inheritance relation would own the same permissions which would render the respective part of the role-hierarchy redundant. Line 9 returns false if a subject tries to delegate a role to a delegation role which is already defined as its senior-role (see Fig. 3d and [16, 17]).

Afterwards, Algorithm 2, lines 10-15 prevent that a role-to-role delegation would result in the assignment of two SME tasks to the same role or subject. In particular, this conflict occurs if the potential senior-role owns a task which is SME to one of the tasks owned by the potential junior-role. Subsequently, if a subject tries to delegate a role owning a task which has a subject-binding to one or more non-delegatable task(s), Algorithm 2, line 16 returns false. This is because subject-bound tasks always have to be performed by the same subject. In case a subject tries to delegate a role owning a task which has a subject-binding to other tasks, Algorithm 2, line 18 returns false, if one of the subject-bound tasks is associated with a non-delegatable duty. If none of the above checks returns false, Algorithm 2 finally reaches line 19 and returns true – meaning that it is allowed to delegate a particular role to a certain delegation role.

Algorithm 3 *Check if it is allowed to assign a particular delegation role to a certain delegatee.*

```
Input: $drole_x \in DR, delegatee, delegator \in S$
1: if $delegator \neq creator(drole_x)$ then return false
2: if $\exists\ role_y \in rown(delegatee) \mid task_y \in town(role_y) \wedge$
3:     $task_x \in town(drole_x) \ \wedge\ task_y \in sme(task_x)$ then return false
4: return true
```

Algorithm 3, line 1 returns false if the subject who wants to assign a delegation role to a particular delegatee is not the creator of this delegation role. Subsequently, we need to check if the delegatee-assignment would result in the assignment of SME tasks to the delegatee (due to other role-memberships of the delegatee). If none of the above checks returns false, Algorithm 3 finally reaches line 4 and returns true – meaning that it is allowed to assign a particular delegatee to a certain delegation role.

Algorithm 4 *Check if a particular task instance that is executed in a certain process instance can be allocated to a specific delegatee.*

```
Input: $drole \in DRT, delegatee \in S, task_{type} \in T_T, process_{type} \in P_T,$
       $process_{instance} \in pi(process_{type}), task_{instance} \in ti(task_{type}, process_{instance})$
1: if $\exists\ instance_y \in ti(type_y, process_{instance}) \mid ar(delegatee) = drole \wedge$
2:     $process_{instance} \notin drpi(drole)$ then return false
3: return true
```

Algorithm 4, line 2 returns false if the selected subject is not allowed to execute a certain task instance because the *temporary delegation role* is not valid for the corresponding process instance. Each temporary delegation role is only valid for particular process instances (see Definition 1.4). In Fig. 3e, subject s_1 is assigned to the temporary delegation role *drt*, and *drt* is only valid for the process instance 123. However, the actual process instance is 456. Thus, s_1 is not allowed to execute the delegated tasks in this process instance. Note that this check is not required for permanent delegation roles.

4 Related Work

In recent years, there has been much work on various aspects of role- and permission-based delegation (see, e.g., [2, 20]). Delegation in a business process/workflow context has also received considerable attention. In [1], the notion of delegation is extended to allow for conditional delegation in workflows. Different types of constraints, such as authorization constraints, are addressed in the context of delegation. The effects of some delegation operations on three workflow execution models are described in [6]. Few contributions exist which consider authorization constraints and related consistency conflicts in the context of delegation. In [14], an extension to PBDM is presented to integrate authorization constraints in permission-based delegation. [14] focuses on static separation of duty constraints and related conflicts and analyzes role-based constraints. In [4], Crampton addresses the satisfiability problem of workflows in the context of constrained delegation and provides an algorithm that determines whether to permit a delegation request. However, the algorithm does not distinguish between different conflict types. In [10], Schaad discusses delegation conflicts. In contrast to our approach, the conflicts are detected after conducting the delegation, while our algorithms detect conflicts before the delegation is performed. Thus, we aim to detect conflicts before causing an inconsistent RBAC configuration.

5 Conclusion

In this paper, we provide a formal metamodel for process-related RBAC delegation models. In addition, we presented generic algorithms to detect conflicts in the context of delegating tasks, duties, and roles. We also discuss the specific problem of mutual-exclusion and binding constraints in an RBAC delegation context. Note that in our approach, conflicts are detected before causing an inconsistent RBAC configuration. Thus, the application of the algorithms presented in this paper helps security engineers to prevent design- and run-time conflicts in access control models and thereby aims to ensure the continuous consistency of corresponding process-related RBAC delegation models.

References

1. V. Atluri and J. Warner. Supporting Conditional Delegation in Secure Workflow Management Systems. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2005.
2. E. Barka and R. Sandhu. Framework for Role-Based Delegation Models. In *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC)*, December 2000.
3. J. Cole, J. Derrick, Z. Milosevic, and K. Raymond. Author Obliged to Submit Paper before 4 July: Policies in an Enterprise Specification. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY)*, January 2001.

4. J. Crampton and H. Khambhammettu. Delegation and Satisfiability in Workflow Systems. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2008.
5. J. Crampton and H. Khambhammettu. Delegation in Role-Based Access Control. *International Journal of Information Security*, 7(2), 2008.
6. J. Crampton and H. Khambhammettu. On Delegation and Workflow Execution Models. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC)*, March 2008.
7. D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House, second edition, 2007.
8. J. B. D. Joshi and E. Bertino. Fine-grained Role-based Delegation in Presence of the Hybrid Role Hierarchy. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2006.
9. R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2), 1996.
10. A. Schaad. Detecting Conflicts in a Role-Based Delegation Model. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC)*, 2001.
11. A. Schaad and J. D. Moffett. Delegation of Obligations. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY)*, June 2002.
12. S. Schefer and M. Strembeck. Modeling Process-Related Duties with Extended UML Activity and Interaction Diagrams. In *Electronic Communications of the EASST, Vol. 37*, March 2011.
13. S. Schefer, M. Strembeck, J. Mendling, and A. Baumgrass. Detecting and Resolving Conflicts of Mutual-Exclusion and Binding Constraints in a Business Process Context. In *Proceedings of the 19th International Conference on Cooperative Information Systems (CoopIS)*, October 2011.
14. Q. Shang and X. Wang. Constraints for Permission-Based Delegations. In *Proceedings of the 8th IEEE International Conference on Computer and Information Technology Workshops (CITWORKSHOPS)*, 2008.
15. M. Strembeck. Scenario-Driven Role Engineering. *IEEE Security & Privacy*, 8(1), 2010.
16. M. Strembeck and J. Mendling. Generic Algorithms for Consistency Checking of Mutual-Exclusion and Binding Constraints in a Business Process Context. In *Proceedings of the 18th International Conference on Cooperative Information Systems (CoopIS)*, October 2010.
17. M. Strembeck and J. Mendling. Modeling Process-related RBAC Models with Extended UML Activity Models. *Information and Software Technology*, 53(5), 2011.
18. J. Wainer, A. Kumar, and P. Barthelmess. DW-RBAC: A formal security model of delegation and revocation in workflow systems. *Information Systems*, 32(3), 2007.
19. J. Warner and V. Atluri. Inter-Instance Authorization Constraints for Secure Workflow Management. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2006.
20. X. Zhang, S. Oh, and R. Sandhu. PBDM: A Flexible Delegation Model in RBAC. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2003.
21. G. Zhao, D. Chadwick, and S. Otenko. Obligations for Role Based Access Control. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW)*, May 2007.