Generic Algorithms for Consistency Checking of Mutual-Exclusion and Binding Constraints in a Business Process Context

Mark Strembeck[1] and Jan Mendling[2]

[1] Vienna University of Economics and Business (WU Vienna), Austria
mark.strembeck@wu.ac.at
[2] Humboldt-Universität zu Berlin, Germany
jan.mendling@wiwi.hu-berlin.de

Abstract In this paper, we present generic algorithms to ensure the consistency of mutual-exclusion and binding constraints in a business process context. We repeatedly identified the need for such generic algorithms in our real-world projects. Thus, the algorithms are a result of the experiences we gained in analyzing, designing, and implementing a number of corresponding software systems and tools. In particular, these algorithms check corresponding consistency requirements to prevent constraint conflicts and to ensure the design-time and runtime compliance of a process-related role-based access control (RBAC) model.

1 Introduction

Security properties such as mutual-exclusion and binding-of-duty play an increasingly important role in process-aware information systems [11]. In the context of business process management, *mutual exclusion* and *binding constraints* are an important means to assist the specification of business processes and to control the execution of workflows. In particular, they are used to enforce process-related separation of duty (SOD) and binding of duty (BOD) policies with respect to a corresponding role-based access control (RBAC) model (see, e.g., [1,3,4,17,18]). A number of approaches exist that allow for the formal specification and analysis of process-related access control policies and constraints (see, e.g., [2,8,17]). However, when building a software system we have to "translate" such formal approaches for the specification of access control policies and constraints to the (programming) language that is used to implement the respective system. With respect to the rapidly increasing importance of process-aware information systems, the correct implementation of corresponding consistency checks in these systems is an important issue.

In this paper, we present a set of algorithms that check and ensure the consistency of mutual-exclusion and binding constraints in a business process context. The definition of these algorithms was inspired by our real-world RBAC and role engineering projects, where we repeatedly identified the need for such generic (i.e. programming language independent) consistency checks. In particular, the algorithms result from the experiences we gained in the analysis, design, and implementation of corresponding software systems and tools (see, e.g., [6,9,10,13,14,15,16]).

The remainder of this paper is structured as follows. Section 2 gives an overview of mutual-exclusion and binding constraints. Next, Section 3 defines the essential elements of process-related RBAC models and specifies requirements for design-time and runtime consistency of these models. Sections 4 and 5 present our algorithms for ensuring the consistency of mutual-exclusion and binding constraints in a process-related RBAC model. Section 6 discusses related work and Section 7 concludes the paper.

2 Mutual Exclusion and Binding Constraints

Separation of duty (SOD) constraints enforce conflict of interest policies [1,5,7]. Conflict of interest arises as a result of the simultaneous assignment of two mutual exclusive tasks or roles to the same subject. Thus, the definition of mutual exclusive artifacts is a well-known mechanism to enforce separation of duty. *Mutual exclusive* roles or tasks result from the division of powerful rights or responsibilities to prevent fraud and abuse. An example is the common practice to separate the "controller" role and the "chief buyer" role in medium-sized and large companies. In this context, a *task-based SOD constraint* is a constraint that considers task order and task history in a particular process instance to decide if a certain subject or role is allowed to perform a certain task [3,17,19]. Task-based SOD constraints can be static or dynamic. A *static task-based SOD constraint* can be enforced by defining that two statically mutual exclusive (SME) tasks must never be assigned to the same role and must never be performed by the same subject. This constraint is global with respect to *all process instances* in the corresponding information system. In contrast, a *dynamic task-based SOD constraint* refers to individual process instances and can be enforced by defining that two dynamically mutual exclusive (DME) tasks must never be performed by the same subject in the *same process instance*. In other words: two DME tasks can be assigned to the same role. However, to complete a process instance which includes two DME tasks, one needs at least two different subjects. This means, although a subject might possess a role which includes all permissions to perform two DME tasks, a DME constraint enforces that the same subject does not perform both tasks in the same process instance.

Binding of Duty (BOD) constraints [4,17,18] define a connection between two (or more) tasks so that a subject (or role) who performed one of these tasks must also perform the corresponding related task(s). In other words, in a given process instance two "bound tasks" must always be performed by the same subject/role, e.g. because of specific knowledge the subject/role acquires while performing the first of two bound tasks, for reasons of organization-internal processing standards, or to simplify interaction with other process stakeholders. Moreover, BOD can be subdivided in subject-based and role-based constraints. A *subject-based BOD constraint* then defines that the *same individual* who performed the first task must also perform the bound task(s). In contrast to that, 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* and *role-binding* as synonyms for subject-based BOD constraints and role-based BOD constraints respectively.

3 Basic Definitions for Process-Related RBAC Models

The context of a workflow system is given through process instances and corresponding task instances. In this paper, we therefore focus on mutual-exclusion and binding constraints defined on the task level. Definition 1 specifies the essential elements of process-related RBAC models and their basic interrelations.

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, such that:*

- *For the sets of the meta model:*
  - *An element of* S *is called* Subject. $S \neq \emptyset$.
  - *An element of* R *is called* Role. $R \neq \emptyset$.
  - *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* T_I *is called* Task Instance.
- *For the partial mappings of the meta model (*$\mathcal{P}$ *refers to the power set):*
  1. *The mapping* $rh : R \mapsto \mathcal{P}(R)$ *is called* role hierarchy. *For* $rh(r_s) = R_j$ *we call* r_s senior role *and* R_j *the set of direct* junior roles. *The transitive closure* rh^* *defines the inheritance in the role-hierarchy such that* $rh^*(r_s) = R_{j^*}$ *includes all direct and transitive junior-roles that the senior-role* r_s *inherits from. The role-hierarchy is cycle-free, i.e. for each* $r \in R : rh^*(r) \cap \{r\} = \emptyset$.
  2. *The mapping* $rsa : S \mapsto \mathcal{P}(R)$ *is called* role-to-subject assignment. *For* $rsa(s) = R_s$ *we call* s subject *and* $R_s \subseteq R$ *the set of* roles assigned to this subject *(the set of roles owned by s). The mapping* $rsa^{-1} : R \mapsto \mathcal{P}(S)$ *returns all subjects assigned to a role (the set of subjects owning a role).*
  This assignment implies a mapping role ownership $rown : S \mapsto \mathcal{P}(R)$*, such that for each subject s all direct and inherited roles are included, i.e.* $rown(s) = \bigcup_{r \in rsa(s)} rh^*(r) \cup rsa(s)$. *The mapping* $rown^{-1} : R \mapsto \mathcal{P}(S)$ *returns all subjects assigned to a role (directy or transitively via a role-hierarchy).*
  3. *The mapping* $es : T_I \mapsto S$ *is called* executing-subject mapping. *For* $es(t) = s$ *we call* s *the* executing subject *and* t *is called* executed task instance.
  4. *The mapping* $er : T_I \mapsto R$ *is called* executing-role mapping. *For* $er(t) = r$ *we call* r *the* executing role *and* t *is called* executed task instance.
  5. *The mapping* $tra : R \mapsto \mathcal{P}(T_T)$ *is called* task-to-role assignment. *For* $tra(r) = T_r$ *we call* r role *and* $T_r \subseteq T_T$ *is called the set of* tasks assigned to r. *The mapping* $tra^{-1} : T_T \mapsto \mathcal{P}(R)$ *returns the set of roles a task is assigned to (the set of roles owning a task).*
  This assignment implies a mapping task ownership $town : R \mapsto \mathcal{P}(T_T)$*, such that for each role* r *the tasks inherited from its junior-roles are included, i.e.* $town(r) = \bigcup_{r_{inh} \in rh^*(r)} tra(r_{inh}) \cup tra(r)$. *The mapping* $town^{-1} : T_T \mapsto \mathcal{P}(R)$ *returns the set of roles a task is assigned to (directly or transitively via a role-hierarchy).*

6. *The mapping* $ti : (T_T \times P_I) \mapsto \mathcal{P}(T_I)$ *is called* task instantiation. *For* $ti(t_T, p_I) = T_i$ *we call* $T_i \subseteq T_I$ *set of* task instances, $t_T \in T_T$ *is called* task type *and* $p_I \in P_I$ *is called* process instance.
7. *The mapping* $pi : P_T \mapsto \mathcal{P}(P_I)$ *is called* process instantiation. *For* $pi(p_T) = P_i$ *we call* p_T *process type and* $P_i \subseteq P_I$ *the set of process instances instantiated from process type* p_T.
8. *The mapping* $ar : S \mapsto R$ *is called* active role mapping. *For* $ar(s) = r$ *we call* s *the subject and* r *the active-role of* s[1].
9. *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.
10. *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.
11. *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.
12. *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.

For process-related RBAC Models there are two types of correctness. *Static correctness* refers to the design-time consistency of the elements and relationships in the Process-related RBAC Model. In particular, it refers to process types and task types. *Dynamic correctness* relates to the compliance of runtime process instances with the mutual-exclusion and binding constraints. Definition 2 provides the rules for static correctness.

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:*
   $\forall t_2 \in sme(t_1) : town^{-1}(t_2) \cap town^{-1}(t_1) = \emptyset$

[1] We assume that each subject can (at the subject's discretion) activate the roles that are directly assigned to this subject as well as the junior-roles of its directly assigned roles (see, e.g., [5,12])

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 2.6 states that it is *not* possible to have a SME constraint *and* a binding constraint between the same task types t_1 and t_2. In other words: *SME constraints conflict with* all types of *binding constraints* (subject-binding and role-binding). This is because a binding constraint defines that (in the context of the same process instance) the instances of two bound task types *must* be performed by the same subject respectively the same role, while a SME constraint defines that the instances of two statically mutual exclusive task types *must not* be performed by the same subject respectively the same role. Obviously, it is impossible to fulfill both constraints at the same time.

Furthermore, Definition 2.7 states that it is *not* possible to specify a DME constraint *and* a subject-binding constraint between the same two task types t_1 and t_2. This means: *DME constraints and subject-binding constraints conflict*. This is because a subject-binding constraint defines that (in the context of the same process instance) the instances of two bound task types *must* be performed by the same subject (the same individual). In contrast, a DME constraint defines that (in the context of the same process instance) the instances of two task types *must not* be performed by the same subject. Again, it is obvious that we cannot fulfill both constraints at the same time. Note that it *is* possible, however, to simultaneously define a role-binding constraint *and* a DME constraint on two tasks. This is because a DME constraint defines that (in the context of the same process instance) a subject *must not* own the instances of two dynamically mutual exclusive task types (see above). A role-binding constraint yet only defines that (in the context of the same process instance) the instances of two bound task types *must* be performed by the same *role*, not by the same subject/individual. This can be interpreted as a peer review (different subjects owning the same role). Therefore, DME constraints and role-binding constraints do *not* conflict. Definition 2.8 specifies that no role can own two SME tasks, neither directly nor via a role-hierarchy (see also Def. 1.1 and Def. 1.5). Finally, Definition 2.9 specifies that no subject can own two roles that are associated with SME tasks.

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:* $\forall t_2 \in sme(t_1), pi \in P_I : \forall t_x \in ti(t_2, pi), t_y \in ti(t_1, pi) : es(t_x) \cap es(t_y) = \emptyset$
   Please note that we include this rule for the sake of completeness only, as the rule must always hold due to the consistency rule for role-ownership and SME (see Def. 2.9).
2. *In the same process instance, the executing subjects of DME tasks must be different:* $\forall t_2 \in dme(t_1), pi \in P_I : \forall t_x \in ti(t_2, pi), t_y \in ti(t_1, pi) : es(t_x) \cap es(t_y) = \emptyset$
3. *In the same process instance, role-bound tasks must have the same executing-role:* $\forall t_2 \in rb(t_1), pi \in P_I : \forall t_x \in ti(t_2, pi), t_y \in ti(t_1, pi) : er(t_x) = er(t_y)$
4. *In the same process instance, subject-bound tasks must have the same executing-subject:* $\forall t_2 \in sb(t_1), pi \in P_I : \forall t_x \in ti(t_2, pi), t_y \in ti(t_1, pi) : es(t_x) = es(t_y)$

4 Algorithms for Design-Time Consistency

The algorithms defined in this section check the design-time consistency of a process-related RBAC model. Therefore, these algorithms operate on task *types* defined in the context of a process-related RBAC model (see Section 3). For the purposes of this paper, we distinguish algorithms and procedures. Here, an *algorithm* performs certain checks based on the current configuration of a process-related RBAC model. Algorithms either return true or false. A *procedure* operates on the current configuration of a process-related RBAC model and may include side-effects (i.e. change model elements, relations, or variables). Procedures either return a set or do not return anything (void).

4.1 Checks for Constraint Definition

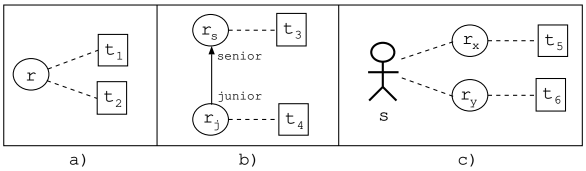


Figure 1. Examples for Algorithm 1

Algorithm 1 *Check if it is allowed to define a (new) SME constraint on two task types.*

Input: $task_1, task_2 \in T_T$
1: **if** $task_1 == task_2$ **then return** false
2: **if** $task_1 \in dme(task_2)$ **then return** false
3: **if** $task_1 \in rbt(task_2)$ **then return** false
4: **if** $task_1 \in sbt(task_2)$ **then return** false
5: **if** $\exists\, r \in R \mid r \in town(task_1) \wedge r \in town(task_2)$
6:     **then return** false
7: **if** $\exists\, s \in S \mid r_1 \in rown(s) \wedge r_2 \in rown(s) \wedge$
8:     $r_1 \in town(task_1) \wedge r_2 \in town(task_2)$
9:     **then return** false
10: **return** true

A task type must not be mutual exclusive to itself (see Def. 2.1). Thus, line 1 of Algortihm 1 returns false if this consistency requirement is not fulfilled. Next, lines 2-4 check the consistency requirements specified in Def. 2.5 and Def. 2.6. Subsequently, lines 5-6 check if a role exists which already owns the two task types. In case a corresponding role is found, the algorithm returns false because defining a SME constraint on two task types that are owned by the same role would violate the consistency requirement specified in Def. 2.8. For example, the definition of a new SME constraint on the tasks t_1 and t_2 in Figure 1a) must be forbidden. Otherwise, r would subsequently own two SME tasks. Similarily, the definition of a new SME constraint on tasks t_3 and t_4 in Figure 1b) must be forbidden. Otherwise, the senior-role r_s would subsequently own two SME tasks (t_3 is directly assigned to r_s and t_4 is inherited from its junior-role r_j). Afterwards, lines 7-9 check if a subject exists that (via its roles) already owns the two task types. In case a corresponding subject is found, the algorithm returns false because defining a SME constraint on two task types that are owned by the same subject would violate the consistency requirements specified in Def. 2.9. Figure 1c) shows an example where the definition of a new SME constraint on the tasks t_5 and t_6 must be forbidden. Otherwise, subject s would subsequently own the right to perform two SME tasks (via its roles r_x and r_y). If none of the above checks returns false, the algorithm finally reaches line 10 and returns true – meaning that it is allowed to define a new SME constraint on the respective task types.

Algorithm 2 *Check if it is allowed to define a (new) DME constraint on two task types.*

Input: $task_1, task_2 \in T_T$
1: **if** $task_1 == task_2$ **then return** false
2: **if** $task_1 \in sme(task_2)$ **then return** false
3: **if** $task_1 \in sbt(task_2)$ **then return** false
4: **return** true

Because it requires less consistency checks, Algorithm 2 is much more simple compared to Algorithm 1. In Algorithm 2, line 1 first ensures the consistency requirement specified in Def. 2.1. Next, lines 2-3 check if the new DME constraint would violate the consistency requirements specified in Def. 2.5 and Def. 2.7. In case none of the above checks returns false, the algorithm finally reaches line 4 and returns true – meaning that it is allowed to define a new DME constraint on the respective task types.

Procedure 1 *Compile the set of all task types that have a direct or a transitive subject-binding relation to a particular* $task_a$.

Name: *allSubjectBindings*
Input: $task_a \in T_T$
$task_a$ **set** *visited* = true
create_empty_set *directbindings*
create_empty_set *transitivebindings*
for each $task_b \in sbt(task_a)$
    if ! $task_b$ $visited$ **then**
        add $task_b$ to *directbindings*
        add *allSubjectBindings*($task_b$) to *transitivebindings*
return *directbindings* $\cup$ *transitivebindings*

Procedure 1 traverses a graph consisting of task types (forming the graph's nodes) and subject-binding relations (forming the graph's edges) that are defined on these task types. In particular, the procedure receives a certain task type ($task_a$) as input parameter and compiles the list of all task types that have a direct or a transitive subject-binding relation to $task_a$. In accordance with standard graph traversal algorithms, each node processed by the algorithm is marked as "visited" in order to have a stop criterion (i.e. all

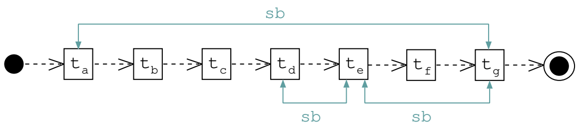


Figure 2. Example for Procedure 1

reachable nodes have been visited). To find all transitive nodes, the algorithm includes a recursion (see line 7). After all reachable nodes have been visited, the algorithm returns the set of all task types having a (direct or transitive) subject-binding to $task_a$. In case no subject-binding for $task_a$ exists, the procedure returns an empty set. The example from Figure 2 shows a process that includes three subject-binding relations between t_a and t_g, t_g and t_e, as well as t_e and t_d respectively. In this example, t_a thus has a direct subject-binding to t_g and transitive subject-bindings to t_e and t_d.

Procedure 2 *Compile the set of all task types that have a direct or a transitive role-binding relation to a particular $task_a$.*

Name: *allRoleBindings*
Input: $task_a \in T_T$
$task_a$ **set** *visited* $=$ `true`
`create_empty_set` *directbindings*
`create_empty_set` *transitivebindings*
for each $task_b \in rbt(task_a)$
  if $!\ task_b\ visited$ **then**
    `add` $task_b$ `to` *directbindings*
    `add` *allRoleBindings*($task_b$) `to` *transitivebindings*
return *directbindings* $\cup$ *transitivebindings*

Procedure 2 is similar to Procedure 1, only that it compiles and returns the set of all task types that have a (direct or transitive) role-binding to a certain task type $task_a$.

Algorithm 3 *Check if it is allowed to define a (new) subject-binding constraint on two task types $task_1$ and $task_2$.*

Input: $task_1, task_2 \in T_T$
if $task_1 == task_2$ **then return** `false`
if $task_1 \in dme(task_2)$ **then return** `false`
if $task_1 \in sme(task_2)$ **then return** `false`
if $\exists\ task_x \in sme(task_1) \mid task_x \in$ *allSubjectBindings*($task_2$)
  then return `false`
if $\exists\ task_x \in dme(task_1) \mid task_x \in$ *allSubjectBindings*($task_2$)
  then return `false`
if $\exists\ task_x \in sme(task_2) \mid task_x \in$ *allSubjectBindings*($task_1$)
  then return `false`
if $\exists\ task_x \in dme(task_2) \mid task_x \in$ *allSubjectBindings*($task_1$)
  then return `false`
return `true`

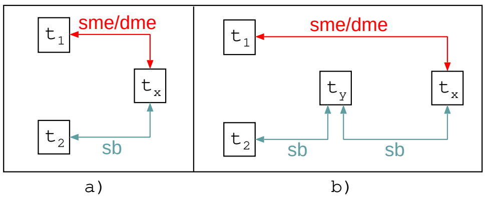


Figure 3. Examples for Algorithm 3

In Algorithm 3, lines 1-3 ensure that the consistency requirements specified in Def. 2.3, 2.6, and 2.7 hold. Next, lines 4-5 check if some $task_x$ exists that is already defined as SME to $task_1$ while having a subject-binding relation to $task_2$ at the same time. In case such a $task_x$ exists, Algorithm 3 returns false because the definition of a new (direct) subject-binding relation between $task_1$ and $task_2$ would also define a (transitive) subject-binding between $task_1$ and $task_x$. In other words, because SME constraints and binding constraints conflict, such a configuration would violate the consistency requirement specified in Def. 2.6. Lines 6-7 perform a similar check for DME constraints to ensure that the consistency requirement specified in Def. 2.7 holds. Figure 3a) shows an example, where the definition of a new subject-binding between the tasks t_1 and t_2 must be forbidden because t_1 already has a (static or dynamic) mutual-exclusion relation to a third task t_x which, at the same time, has a subject-binding relation to t_2. Figure 3b) shows another example, where a new subject-binding between t_1 and t_2 must be forbidden because t_2 has a transitive subject-binding relation to a task t_x which also has a (static or dynamic) mutual-exclusion relation to t_1 (see also Procedure 1). Note that it is necessary to perform the checks from the perspective of $task_1$ (lines 4-7) *and* from the perspective of $task_2$ (lines 8-11) . In case none of the above checks returns false, the Algorithm finally reaches line 12 and returns true – meaning that it is allowed to define a new subject-binding constraint on the respective task types.

Algorithm 4 *Check if it is allowed to define a (new) role-binding constraint on two task types*

Input: $task_1, task_2 \in T_T$
1: **if** $task_1 == task_2$ **then return** false
2: **if** $task_1 \in sme(task_2)$ **then return** false
3: **if** $\exists\ task_x \in sme(task_1) \mid task_x \in$ *allRoleBindings*($task_2$)
4: **then return** false
5: **if** $\exists\ task_x \in sme(task_2) \mid task_x \in$ *allRoleBindings*($task_1$)
6: **then return** false
7: **return** true

In principle, the checks in Algorithm 4 are similar to the checks performed by Algorithm 3. However, because DME constraints do not conflict with role-binding constraints (see Section 3), Algorithm 4 only has to ensure that the consistency requirements specified in Def. 2.3 (line 1) and Def. 2.6 (lines 2-6) hold. If none of the above checks returns false, Algorithm 4 finally reaches line 7 and returns true – meaning that it is allowed to define a new role-binding constraint on the corresponding task types.

4.2 Checks for new Assignment Relations

Procedure 3 *Compile the set of all direct and transitive senior-roles of a* $role_a$*.*

Name: *allSeniorRoles*
Input: $role_a \in R$
1: create_empty_set *transitiveseniorroles*
2: **for each** $role_x \in directSeniorRoles(role_a)$
3: add *allSeniorRoles*($role_x$) to *transitiveseniorroles*
4: **return** *transitiveseniorroles* $\cup$ *directSeniorRoles*($role_a$)

First, we define the procedure $allSeniorRoles$ because this procedure is needed for the definition of the subsequent algorithms. Procedure 3 traverses the role-hierarchy to compile the set of all (direct and transitive) senior-roles of a particular role.To find all transitive senior-roles the procedure includes a recursion (see line 3).

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

Input: $task_x \in T_T, role_y \in R$
1: **if** $\exists\, task_y \in town(role_y) \mid task_y \in sme(task_x)$
2: **then return** false
3: **if** $\exists\, role_z \in allSeniorRoles(role_y) \mid$
4: $task_z \in town(role_z) \wedge task_z \in sme(task_x)$
5: **then return** false
6: **if** $\exists\, s \in S \mid role_y \in rown(s) \wedge role_z \in rown(s) \wedge$
7: $task_z \in town(role_z) \wedge task_z \in sme(task_x)$
8: **then return** false
9: **return** true

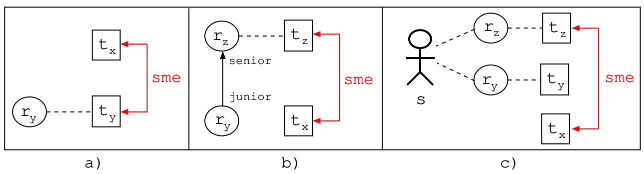


Figure 4. Examples for Algorithm 5

In Algorithm 5, lines 1-2 check if the role already owns some $task_y$ which has a SME constraint to $task_x$. If such a $task_y$ exists, the algorithm returns false to ensure the consistency requirement specified in Def. 2.8. Figure 4a) shows a corresponding example, where task t_x must not be assigned to role r_y because r_y already owns t_y which is defined as SME to t_x. Next, lines 3-5 check if $role_y$ has a (direct or tranisitive) senior-role $role_z$ which again owns a $task_z$ that has a SME constraint to $task_x$. In case such a $role_z$ exists, the algorithm returns false to ensure the consistency requirement specified in Def. 2.8. Figure 4b) shows an example where a senior-role r_z owns a task t_z which is defined as SME to task t_x. Therefore, t_x must not be assigned to r_y (or any other junior-role of r_z). This is because assigning t_x to r_y would also mean to transitively assign t_x to r_z (and to any other senior-role of r_y). Thus, r_z would inherit t_x from its junior-role r_y and thereby own two SME tasks (see also Def. 1.1 and Def. 1.5). Subsequently, lines 6-8 check if one of the subjects owning $role_y$ does also own another $role_z$ which again has a $task_z$ that is defined as SME to $task_x$. In case such a subject exists, the algorithm returns false to ensure the consistency requirement specified in Def. 2.9. In the example from Figure 4c), subject s owns the right to perform the tasks t_y and t_z (via its roles r_y and r_z). Moreover, t_z has an SME constraint on t_x. Therefore, t_x must not be assigned to r_y. This means, although r_y does not own a task which has a SME constraint on t_x (neither directly nor transitively), the assignment of t_x to r_y must still be forbidden because subject s simultaneously owns r_y and r_z. In case none of the above checks returns false, Algorithm 5 finally reaches line 9 and returns true – meaning that it is allowed to assign $task_x$ to $role_y$.

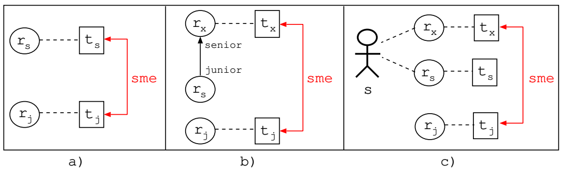


Figure 5. Examples for Algorithm 6

Algorithm 6 *Check if it is allowed to define a (new) junior-role relation between two roles. In particular, check if it is allowed to define a role $junior$ as junior-role of another role $senior$ (also called role-to-role assignment).*

Input: $junior, senior \in R$

1: **if** $junior == senior$ **then return** $false$
2: **if** $\exists\, task_j \in town(junior) \wedge task_s \in town(senior) \mid$
3: $\quad task_j \in sme(task_s)$
4: $\quad$ **then return** $false$
5: **if** $\exists\, role_x \in allSeniorRoles(senior) \mid$
6: $\quad task_x \in town(role_x) \wedge task_j \in town(junior) \wedge$
7: $\quad task_x \in sme(task_j)$
8: $\quad$ **then return** $false$
9: **if** $\exists\, s \in S \mid senior \in rown(s) \wedge role_x \in rown(s) \wedge$
10: $\quad task_x \in town(role_x) \wedge task_j \in rown(junior) \wedge$
11: $\quad task_x \in sme(task_j)$
12: $\quad$ **then return** $false$
13: **return** $true$

Because a role cannot be its own junior-role, line 1 of Algorithm 6 first checks this consistency requirement (see also Def. 1.1). Next, lines 2-4 check if the designated

$junior$ role owns a $task_j$ that has a SME constraint to another $task_s$ which is owned by the designated $senior$ role. In case such two a $task_j$ and $task_s$ exist, the algorithm returns false to ensure the consistency requirement specified in Def. 2.8. Figure 5a) shows a corresponding example, where the definition of a new junior-role relation between r_s and r_j must be forbidden, because t_s and t_j are SME tasks. Next, lines 5-8 check if the designated $senior$ role already has a (direct or transitive) senior-role $role_x$ that owns a $task_x$ and has a SME constraint to another $task_j$ that is assigned to the designated $junior$ role. In case such a $role_x$ exists, the algorithm returns false to ensure the consistency requirement specified in Def. 2.8. For example, in Figure 5b) role r_j must not be defined as junior-role of r_s. Otherwise, r_x (senior-role of r_s) would be able to perform the two SME tasks t_x and t_j. Subsequently, lines 9-12 check if one of the subjects already owning the designated $senior$ role, does also own another $role_x$ that grants the right to perform a $task_x$ which has a SME constraint to another $task_j$ assigned to the designated $junior$ role. In case such a $role_x$ exists the algorithm returns false to ensure the consistency requirement specified in Def. 2.9. Figure 5c) shows a corresponding example where role r_j must not be defined as junior-role of r_s. This means, although r_s and r_j do not own two SME tasks, the definition of a new junor-role relation between r_s and r_j must still be forbidden because subject s simultaneously owns r_x and r_s. Otherwise, s would acquire the right to perform two SME tasks (t_x and t_j). In case none of the above checks returns false, Algorithm 6 finally reaches line 13 and returns true – meaning that it is allowed to define a new junior-role/senior-role relation between the corresponding roles. Note, that because role-hierarchies are directed acyclic graphs, it is in fact also necessary to check if a new junior-role relation (a new role-to-role assignment) would create a cycle in the role-hierarchy (see also Def. 1.1). However, because this issue is generic to each DAG and is not related to mutual-exclusion or binding constraints, we decided to omit the cycle check in Algorithm 6.

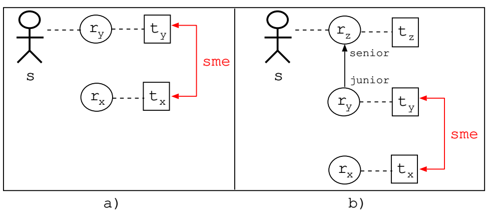


Figure 6. Examples for Algorithm 7

Algorithm 7 *Check if it is allowed to assign a particular role to a particular subject (role-to-subject assignment).*

Input: $role_x \in R, subject \in S$

if $\exists\, role_y \in rown(subject) \mid task_y \in town(role_y) \wedge$

$\quad task_x \in town(role_x) \wedge task_y \in sme(task_x)$

$\quad$ ***then return*** *false*

return *true*

In Algorithm 7, lines 1-3 check if the respective subject already owns a $role_y$ which grants the right to perform a $task_y$ that has a SME constraint to one of the tasks assigned to $role_x$. In case such a $role_y$ exists, the algorithm returns false to ensure the consistency requirement specified in Def. 2.9. Figure 6a) shows a respective example, where role r_x must not be assigned to subject s, because s already owns role r_y. Otherwise, s would acquire the right to perform two SME tasks (t_x and t_y). In Figure 6b) we see another example, where r_x must not be assigned to s because s already owns r_z and can thereby (via the role-hierarchy) perform t_y which has an SME constraint on t_x.

5 Algorithms for Runtime Consistency

Runtime consistency refers to the fact that the constraints defined in a process-related RBAC model must not only be enforced at design-time but also when executing actual process instances (see Section 3). In particular, mutual-exclusion and binding constraints directly impact the allocation of tasks to subjects. First, we define the procedure $executableTasks$ because it is needed for the definition of the subsequent algorithm.

Procedure 4 *Compile the set of all tasks a particular subject could currently execute (based on the roles currently assigned to this subject).*

Name: *executableTasks*
Input: $s \in S$
create_empty_set *executable*
for each $role \in rown(s)$
    add $town(role)$ to *executable*
return *executable*

Procedure 4 visits all roles assigned to a particular subject to compile the set of all tasks that can (potentially) be executed by this particular subject, i.e. all tasks that are directly or transitively assigned to the respective subject (see also Def. 1.2 and Def. 1.5).

Algorithm 8 *Check if particular task instance (which is part of a particular process instance) can be allocated to a particular subject.*

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** false
if $es(task_{instance}) \neq \emptyset$ **then return** false
if $er(task_{instance}) \neq \emptyset \land er(task_{instance}) \neq ar(subject)$
    then return false
if $\exists\, type_x \in allSubjectBindings(task_{type}) \mid$
    $type_x \notin executableTasks(subject)$
    then return false
if $\exists\, instance_y \in ti(type_y, process_{instance}) \mid$
    $type_y \in dme(task_{type}) \land es(instance_y) == subject$
    then return false
return true

Algorithm 8 checks if the instance of a certain task type can be allocated to a certain subject. First, line 1 checks if the corresponding $subject$ is allowed to execute the $tasks_{type}$ the corresponding $task_{instance}$ was instantiated from (see also Procedure 4). If the $subject$ is not allowed to execute this particular $task_{type}$ the respective instance must not be allocated to this subject and therefore the algorithm returns false. Next, line 2 checks if the corresponding task instance has already been allocated, i.e. if this task instance already has an executing-subject (see also Def. 1.3). In case the respective task instance is already allocated to another subject, the algorithm returns false. In particular, this means that the $subject$ cannot be allocated to this very $task_{instance}$ but it can still be allocated to other instances of the corresponding $task_{type}$, of course. Afterwards, lines 3-4 check if the $task_{instance}$ already has an executing-role, and if so whether this executing-role is also the currently active role of the respective $subject$. If this is not the case, the algorithm returns false (note that the executing-role of a task instance can be allocated before allocating an executing-subject to this task instance, see also discussion concerning Procedure 5 below).Subsequently, lines 5-7 check if a $type_x$ exists that has a subject-binding relation to $task_{type}$ but cannot be executed by the $subject$. In case such a $type_x$ exists, the algorithm returns false to ensure the consistency requirement specified in Def. 3.4. In other words, a $task_{instance}$ must only be allocated to a certain $subject$ if this subject owns the right to perform the corresponding $task_{type}$ as well as all subject-bound tasks. Next, lines 8-10 check if the $subject$ is already allocated to the $instance_y$ of a $task_y$ which has a DME constraint to $task_{type}$. If the $subject$ already is the executing-subject of such an $instance_y$, the algorithm returns false to ensure the consistency requirement specified in Def. 3.2. In other words, in the same process instance a task instance must not be allocated to a particular subject if this very subject is already allocated to the instance of a DME task type. If none of the above checks returns false, Algorithm 8 finally reaches line 11 and returns true – meaning that the corresponding $subject$ may actually be allocated to the corresponding $task_{instance}$. Algorithm 8 may also be used to compile the set of all subjects who are potentially allocatable to a certain task instance and then randomly allocate the corresponding task instance to one of these subjects (see also [11]). Note that we do not need to check static mutual-exclusion constraints when allocating a task, because in conformance with algorithms 1 to 7 no subject can ever be assigned to two SME tasks.

Procedure 5 *Allocate a certain task instance to a certain subject.*

Name: *allocateTask*
Input: $s \in S, task_{type} \in T_T, process_{instance} \in P_I$
$task_{instance} \in ti(task_{type}, process_{instance})$

set $es(task_{instance})$ to s
set $er(task_{instance})$ to $ar(s)$
for each $type_x \in allSubjectBindings(task_{type})$
  for each $instance_x \in ti(type_x, process_{instance})$
    set $es(instance_x)$ to s
    set $er(instance_x)$ to $ar(s)$
for each $task_y \in allRoleBindings(task_{type})$
  for each $instance_y \in ti(task_y, process_{instance})$
    set $er(instance_y)$ to $ar(s)$

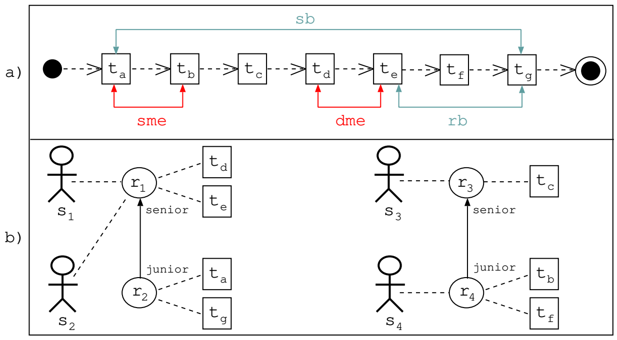
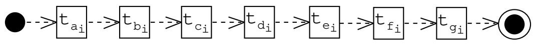


Figure 7. Example for Procedure 5 (Design level)

After we used Algorithm 8 to check if a certain task instance can (potentially) be allocated to a particular subject, we can actually allocate the task instance. Procedure 5 describes the steps that are performed to allocate a task instance. First, we define the respective subject as the executing-subject of the $task_{instance}$, and the subject's active role as the executing role of the $task_{instance}$ (see lines 1-2). Next, lines 3-6 perform a lookup to find all instances of subject-bound tasks (see also Procedure 1) and define the executing-subject and the executing-role for each of the subject-bound tasks accordingly. In particular, all instances of subject-bound tasks are allocated at the same time to ensure the consistency requirements specified in Def. 3.3 and Def. 3.4. Finally, lines 7-9 perform a lookup to find all role-bound tasks and set the executing-role of all role-bound tasks. In particular, the executing-role of all role-bound tasks is allocated at the same time to ensure the consistency requirement specified in Def. 3.3. This means that the executing-role of a certain task instance may be allocated before allocating an executing subject (see example below).

Figure 7a) shows an example process consisting of seven task types t_a to t_g. For the sake of simplicity, we chose a linear example process. However, the task allocation procedure can be applied to arbitrary process definitions, of course. In addition to the process flow, Figure 7a) also indicates different mutual-exclusion and binding relations between some of the tasks. In particular, it shows a SME constraint between t_a and t_b, a DME constraint between t_d and t_e, a subject-binding between t_a and t_g, and a role-binding between t_e and t_g. Moreover, Figure 7b) shows a corresponding process-related RBAC model that includes four roles (r_1 to r_4) as well as four subjects (s_1 to s_4). Role r_1 is assigned to subjects s_1 and s_2, r_3 is assigned to s_3, and r_4 is assigned to s_4.

Now we give an example that demonstrates the allocation of executing-subjects and executing-roles for a particular instance of the process type from Figure 7a) considering the process-related RBAC model from Figure7b). Figure 8 depicts an instance of the respective process type and the successive allocation of the corresponding task instances (in the example an additional index i is used to indicate task instances t_{a_i} to t_{g_i}). First, we have to allocate t_{a_i}. Using Algorithm 8, we find that t_{a_i} is allocatable to subjects s_1 and s_2. In our example, we choose to allocate t_{a_i} to s_1, see Figure 8a). Moreover, because t_a has a subject-binding to t_g (see Figure 7), we allocate t_{g_i} to s_1



a)

	t_{a_i}	t_{b_i}	t_{c_i}	t_{d_i}	t_{e_i}	t_{f_i}	t_{g_i}
executing subject	s_1						s_1
executing role	r_1				r_1		r_1

b)

	t_{a_i}	t_{b_i}	t_{c_i}	t_{d_i}	t_{e_i}	t_{f_i}	t_{g_i}
executing subject	s_1	s_4					s_1
executing role	r_1	r_4			r_1		r_1

c)

	t_{a_i}	t_{b_i}	t_{c_i}	t_{d_i}	t_{e_i}	t_{f_i}	t_{g_i}
executing subject	s_1	s_4	s_3				s_1
executing role	r_1	r_4	r_3		r_1		r_1

d)

	t_{a_i}	t_{b_i}	t_{c_i}	t_{d_i}	t_{e_i}	t_{f_i}	t_{g_i}
executing subject	s_1	s_4	s_3	s_1			s_1
executing role	r_1	r_4	r_3	r_1	r_1		r_1

e)

	t_{a_i}	t_{b_i}	t_{c_i}	t_{d_i}	t_{e_i}	t_{f_i}	t_{g_i}
executing subject	s_1	s_4	s_3	s_1	s_2		s_1
executing role	r_1	r_4	r_3	r_1	r_1		r_1

f)

	t_{a_i}	t_{b_i}	t_{c_i}	t_{d_i}	t_{e_i}	t_{f_i}	t_{g_i}
executing subject	s_1	s_4	s_3	s_1	s_2	s_4	s_1
executing role	r_1	r_4	r_3	r_1	r_1	r_4	r_1

Figure 8. Example for Procedure 5 (Runtime task allocation)

in the same step. In addition, because of the role-binding between t_e and t_g we also set the executing-role of t_{e_i} to r_1, see Figure 8a). Thus, in order to ensure the runtime consistency of the binding relations defined at the design-level (see Figure 7), the allocation of t_{a_i} transitively affects t_{g_i} and t_{e_i} (see also Procedure 5, Def. 3.3, and Def. 3.4). Next, we use Algorithm 8 and Procedure 5 to allocate t_{b_i} to s_4, t_{c_i} to s_3, and t_{d_i} to s_1, see Figure 8b) - d). Subsequently, we have to allocate t_{e_i}. In principle, instances of t_e could be allocated to s_1 or s_2 (see Figure 7). However, because of the DME constraint between t_d and t_e, instances of these tasks must be allocated to different subjects (see also Algorithm 8 and Def. 3.2). In our example, we therefore have to allocate t_{e_i} to s_2 because t_{d_i} was allocated to s_1, see Figure 8e). Finally, we allocate t_{f_i} to s_4 and thereby have allocated all task instances, see Figure 8f).

6 Related Work

A number of contributions exist that discuss constraint specification or possible conflicts that may occur when defining SOD or BOD constraints. In [1], Ahn and Sandhu present the RCL 2000 language for the specification of role-based authorization constraints. They also show how separation of duty 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. [2] present a language to express SOD constraints as clauses in logic programs. Moreover, they present corresponding algorithms that check the consistency of such constraints with the users/roles that execute the tasks in a workflow. Thereby they ensure that all tasks within a workflow are performed by predefined users/roles only. In [3], 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. Tan et al. [17] 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 [5] 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.

While most of these works use declarative formalisms, we provide imperative algorithms for implementing SOD and BOD correctness checks. Our work complements previous contributions by providing generic algorithms and procedures to ensure the design-time and runtime consistency of process-related RBAC models. The algorithms result from our experiences in analyzing, designing, and implementing corresponding software systems (see, e.g., [9,10,13,14,15,16]).

7 Conclusion

In this paper, we presented a set of algorithms that ensure the consistency of mutual-exclusion and binding constraints in a business process context. In particular, the algorithms ensure the design-time and runtime consistency of a process-related RBAC model, with respect to the mutual-exclusion and binding constraints that are included in the respective model. The algorithms are defined in a generic fashion that is independent of a certain software platform or programming language. They were inspired through our real-world RBAC and role engineering projects, where we repeatedly identified the need for such generic consistency checks. Thus, our algorithms complement previous work on mutual-exclusion and binding constraints by providing a practical and implementation-oriented perspective on constraint consistency.

In recent years, we see an increasing interest in process-aware information systems in both research and practice. In this context, an increasing number of existing and future systems will have to be extended with respective consistency checks. Among other things, we already implemented the algorithms in an RBAC service, a general-purpose policy framework (supporting authorization, obligation, and delegation policies), a runtime-engine for business processes, as well as in a role-engineering tool.

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. 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), February 1999.

3. R. Botha and J. Eloff. Separation of duties for access control enforcement in workflow environments. *IBM Systems Journal*, 40(3), 2001.
4. F. Casati, S. Castano, and M. Fugini. Managing Workflow Authorization Constraints through Active Database Technology. *Information Systems Frontiers*, 3(3), September 2001.
5. 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.
6. S. Kunz, S. Evdokimov, B. Fabian, B. Stieger, and M. Strembeck. Role-Based Access Control for Information Federations in the Industrial Service Sector. In *Proc. of the 18th European Conference on Information Systems (ECIS)*, June 2010.
7. N. Li, M. Tripunitara, and Z. Bizri. On Mutually Exclusive Roles and Separation-of-Duty. *ACM Transactions on Information and System Security (TISSEC)*, 10(2), May 2007.
8. N. Li and Q. Wang. Beyond separation of duty: An algebra for specifying high-level security policies. *Journal of the ACM (JACM)*, 55(3), July 2008.
9. J. Mendling, K. Ploesser, and M. Strembeck. Specifying Separation of Duty Constraints in BPEL4People Processes. In *Proc. of the 11th International Conference on Business Information Systems (BIS)*, volume 7 of *Lecture Notes in Business Information Processing (LNBIP)*. Springer-Verlag, May 2008.
10. G. Neumann and M. Strembeck. Design and Implementation of a Flexible RBAC-Service in an Object-Oriented Scripting Language. In *Proc. of the 8th ACM Conference on Computer and Communications Security (CCS)*, November 2001.
11. N. Russell, W. van der Aalst, A. ter Hofstede, and D. Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support. In *Proc. of the 17th International Conference on Advanced Information Systems Engineering (CAiSE), Lecture Notes in Computer Science (LNCS), Vol. 3520, Springer Verlag*, June 2005.
12. R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2), February 1996.
13. M. Strembeck. Conflict Checking of Separation of Duty Constraints in RBAC - Implementation Experiences. In *Proc. of the Conference on Software Engineering (SE 2004)*, February 2004.
14. M. Strembeck. A Role Engineering Tool for Role-Based Access Control,. In *Proc. of the 3rd Symposium on Requirements Engineering for Information Security (SREIS)*, August 2005.
15. M. Strembeck. Scenario-Driven Role Engineering. *IEEE Security & Privacy*, 8(1), January/February 2010.
16. M. Strembeck and G. Neumann. An Integrated Approach to Engineer and Enforce Context Constraints in RBAC Environments. *ACM Transactions on Information and System Security (TISSEC)*, 7(3), August 2004.
17. K. Tan, J. Crampton, and C. Gunter. The Consistency of Task-Based Authorization Constraints in Workflow Systems. In *Proc. of the 17th IEEE Workshop on Computer Security Foundations (CSFW)*, June 2004.
18. J. Wainer, P. Barthelmes, and A. Kumar. W-RBAC - A Workflow Security Model Incorporating Controlled Overriding of Constraints. *International Journal of Cooperative Information Systems (IJCIS)*, 12(4), December 2003.
19. J. Warner and V. Atluri. Inter-Instance Authorization Constraints for Secure Workflow Management. In *Proc. of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2006.