Computer Science and Information Systems 15(1):1–30 https://doi.org/10.2298/CSIS160628037M

# Context-sensitive Constraints for Access Control of Business Processes

Gordana Milosavljević, Goran Sladić *, Branko Milosavljević, Miroslav Zarić, Stevan Gostojić, and Jelena Slivka

Faculty of Technical Sciences, University of Novi Sad, Trg D. Obradovića 6
21000 Novi Sad, Serbia
{grist, sladicg, mbranko, miroslavzaric, gostojic, slivkaje}@uns.ac.rs

**Abstract.** Workflow management systems (WfMS) are used to automate and facilitate business processes of an enterprise. To simplify the administration, it is a common practice in many WfMS solutions to allocate a role to perform each activity of the process and then assign one or more users to each role. Typically, access control for WfMS is role-based with a support of constraints on users and roles. However, merely using role and constraints concepts can hardly satisfy modern access control requirements of a contemporary enterprise. Permissions should not solely depend on common static and dynamic principles, but they must be influenced by the context in which the access is requested. In this paper, we focus on the definition and enforcement of the context-sensitive constraints for workflow systems. We extended the common role-based constraints listed in literature with context-sensitive information and workflow specific components. Also, we propose a mechanism for enforcing such constraints within WfMS.

**Keywords:** constraints, separation of duty, access control, context-sensetive, business process.

## 1. Introduction

A business process is defined as a set of one or more linked activities that collectively realize a business objective within the context of organizational structure that defines functional roles and relationships [11] [16]. Workflow management aims at modeling and controlling the execution of business processes involving a combination of manual and automated activities in an organization. A workflow management system (WfMS) is a system that provides process specification, enactment, monitoring, coordination and administration of workflow process through the execution of software, whose order of execution is based on the workflow logic [11].

To ensure that the activities (tasks) of a business process (workflow) are executed by authorized users, a proper authorization mechanisms must be applied. To simplify security administration, it is common practice in many WfMS to express access control in terms of the roles within the organization rather than individuals. The Role-based Access Control (RBAC) model [19] and its derivates with different constraints had been widely applied to both commercial and research workflow systems to meet workflow access control requirements. Usually, a particular role is assigned to each task in a workflow. Thus, users can

* Goran Sladić is the corresponding author.

perform tasks based on the privileges possessed by their role or roles they inherit. However, there still exist many problems in the aspect of describing complex workflow access control authorization and constraints. Especially, those models unable to express business character and phase authorization constraints in carrying out the task between roles and users. Traditional access control models, such as RBAC are passive access control. They do not take into account contextual information, such as processed data, location, time, history, etc. for making access decisions. Consequently, these models are inadequate for specifying the access control needs of many complex real-world workflows. As context data gets involved, the access decision no longer depends on user credentials only; it also depends on the state of the system's environment and the system itself.

In our previous research, we propose *Context-sensitive access control model for business processes* (COBAC) [40]. We separate our model in two components: *Core* (contains core model entities) and *Constraint* (contains constraints defined over the core entities). In the previous paper [40], the *Core* component of the COBAC model is described including the process of the access control execution. In this paper, we present the COBACs *Constraint* component. Also, the process of access control execution with the constraints enforcement is given. Rather than defining a new type of constraints, the constraints defined by the COBAC model are based on the standard RBAC constraints that can be found in the literature that are extended to support an influence of the context and which are adapted for the use in a workflow system. The COBAC constraint model identifies and extends two common types of constraints: *static* and *dynamic*. The static constraints are enforced during the administration phase while the dynamic constraints are invoked when users are actively executing a business process. Our model is independent of the notation/model used for a business process modeling as long as in a used approach it is possible to identify activities and order of their execution.

The rest of the paper is structured as follows. Section 2 reviews the related work. Clarification of the core COBAC concepts, which are necessary for understanding the proposed constraint model are described in Section 3. Section 4 presents the static constraints defined by the COBAC model while the dynamic constraints are given in Section 5. The constraints enforcement is given in Section 6. Section 7 concludes the paper and outlines further research directions.

## 2. Related Work

In this section, two fields of research are reviewed. The first field deals with role-based models adopted for use in workflow systems. The notion of the context, in this case, is usually limited to the workflow environment, and access control models are extended with some workflow specific features. The second research field gives an overview of the latest results on context-sensitive access control, where the notion of the context is more general and covers a wider range of meaning.

One approach to align RBAC with access control requirements for workflow is to extend the standard RBAC with the task. The paper [50] introduces the TRBAC (Task-Role-Based Access Control) model that is based on RBAC and TBAC (Task-Based Access Control) [44] models. The central idea of this research is that the user has a relationship with permission through role and task. Permissions are assigned to tasks and tasks are assigned to roles. A similar model is presented by Oh and Park [32]. Their model is based on

the classification of job functions. Three different types of tasks are identified: *workflow task* used for workflow oriented job functions, *non workflow task* used for non workflow oriented functions, and *supervision task* used for supervision job functions. Identically as the previous model, permissions to the user are assigned through roles and task. Yao et al. [49] are also using tasks to connect roles and permissions. The unit of the task becomes the permission granularity. Authors propose constraints on user, role, task, and session so that an access control configuration will not result in the leakage of a right to an unauthorized principal. An RBAC based workflow access control model in which tasks and permissions are assigned to the roles is presented in [9].

Since business processes execution may be spread over different organizational units, it is noticed that workflow access control models should be organizational aware. Wainer et al. [47] propose a workflow access control model based on RBAC model extended with *case* and *organization unit* entities and appropriate relations. The entity *case* is added to be able to refer to an instance of a process. Within organizations, and thus in workflow applications, the concept of a hierarchy of people/organizations is prevalent. While workflow systems, as a rule, include some form of organizational modeling capabilities, RBAC by itself does not have such a hierarchy modeled. Therefore, the RBAC model is extended with the *organization unit* entity. Constraints can be established over any of the relationships of the model and can be broadly classified into static and dynamic. Authors consider that some constraints are more important than others. In certain situations, it may be acceptable to override the less important constraints. Therefore, a priority of constraints is introduced.

The standard RBAC model [19] supports a limited number of different types of authorization constraints, which cannot fulfill requirements that have emerged in modern organization's business processes. A typical constraint, which is very relevant, well-known and probably the most used in the security area is *Separation of Duty* (SoD). Although there are many variations, SoD is fundamentally a requirement that critical operations are divided among two or more people so that no single individual can compromise security. The standard RBAC model defines two types of the SoD constraints: *static* and *dynamic*. Static SoD is enforced during the administration phase while dynamic SoD is invoked when users are actively using the system. Researchers have proposed a great variety of SoD models, only some of which are implemented in real systems. Simon and Zurko [39] examines how SoD is used in role-based environments. They characterize role-based environments with an emphasis on those concepts needed to define SoD variations. Also, authors outline different kinds of SoD variations and discuss the mechanisms required to implement those policies. The paper [29] summarize and categorize SoDs of the RBAC model. It gives the formal description of seven typical SoDs and analyzes their state transitions.

Botha and Eloff [10] analyze potential access control conflicts between common workflow entities. They identified different kinds of conflicts that exist when considering separation of duty requirements in workflow environments. Based on the identified conflicts, authors propose an approach for separation of duties in workflow environment that is based on *conflicting entities*: *conflicting privileges*, *conflicting users*, *conflicting roles* and *conflicting tasks*. Bertino et al. [6] present a language for defining constraints on role assignment and user assignment to tasks in a workflow. By using this language, it is possible to express conditions constraining the users and roles that can execute a given

task. Such constraint language supports both static and dynamic separation of duties. Authors show how such constraints can be a formally expressed as clauses in a logic program so that it is possible to exploit all the results available in logic programming and deductive database areas. Also, the authors classified three types of constraints: static, dynamic, and hybrid-based on the time at which they can be evaluated. While static constraints can be evaluated before the execution of the workflow, dynamic constraints can only be evaluated during execution of the workflow. Hybrid constraints can be partially evaluated before execution. Perelson et al. [33] demonstrate the "conflicting entities" (conflicting permissions, conflicting roles, conflicting users and conflicting tasks) paradigm as a way of specifying SoD requirements for workflow systems. This paradigm uses the task abstraction to define an intuitive separation of duty requirements that involve a sequence of operations. It was shown that both Static and Dynamic SoD requirements could be formulated according to the "conflicting entities" paradigm. Crampton [13] describe a model, independent of any underlying access control paradigm, for specifying authorization constraints such as separation of duty, binding of duty and cardinality constraints in workflow systems. Warner and Atluri [48] considered that SoD constraints that span multiple instances of a workflow also need to be analyzed to mitigate the security fraud. They extend the notion of SoD to include constraints that span multiple executing instances of a workflow and constraints that also take into consideration the history of completed workflow instances.

Various definitions of context have been proposed in literature [1] [17] [21] [36] [45]. Broadly, the notion of context relates to the characterization of environment conditions that are relevant for performing appropriate actions in the computing domain. Probably the most widely accepted definition has been provided in [1] [17]:

"*Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and application, including the user and applications themselves*".

To overcome various shortcomings of the RBAC model, many context-sensitive extensions of the RBAC model have been proposed. Kumar et al. [30] noticed that the RBAC model does not specify whether the permission applies to a target object or all instances of a class of objects. They propose the Context-sensitive RBAC that define a permission as the authority to perform a specific operation on a class of objects. A user may be granted membership of a role, but the role membership is only valid within a certain role context. The role context limits the applicability of the role's permissions to a subset of the instances. To define the role context, they define a user context (captures all users security-relevant information) and an object context (captures all objects security-relevant information). The role context is composed of these contexts by specifying a logical constraint expression. Fadhel et al. [18] propose the GemRBAC model (Generalized Model for RBAC) that includes all the conceptual entities required to define most important constraints they identified in the literature. These constraints are specified using UML and OCL. Contextual information (time and location) can be assigned to users, roles, and/or permissions. The paper [43] presents an approach that uses special purpose RBAC constraints to base certain access control decisions on context information. In this approach, a context constraint is defined as a dynamic RBAC constraint that checks the actual values of one or more contextual attributes for predefined conditions. If these conditions are satisfied, the corresponding access request can be permitted. Accordingly, a

conditional permission is an RBAC permission that is constrained by one or more context constraints. Schefer-Wenzl and Strembeck [34] [35] address RBAC context constraints from a business process modeling perspective. The proposed solution integrates context constraints into process related RBAC models to support context-dependent task execution. A process-related context constraint is associated with a task, meaning that certain contextual attributes must meet certain predefined conditions to permit the task execution. Authors formally embed RBAC context constraints into a business process context. Based on the formal model, a corresponding extension for UML 2 activity diagrams is defined. The paper [26] describes a context-sensitive access control model that consists of the context model, context-sensitive policy model, and context-sensitive request model. In the paper [3], the Conditional Role Based Access Control (C-RBAC) model is presented. This model relies on the RBAC model and extends the notion of the role by incorporating attributes, and is based on the notion of system context. Covington et al. [12] introduce the notion of the environmental role and provides a uniform access control framework that can be used to secure context-sensitive applications. Georgiadis et al. [23] discuss the integration of contextual information with the team and role-based access control. In the paper [20] authors showed the use of context information and its quality indicators to grant access permissions to resources. Many authors [7] [25] [28] [38] propose a context-based access control model for web services. Their approach grants and adapts permissions to users based on a set of contextual information collected from environments of the system. Usage of the context-sensitive access control for controlling XML documents is presented in the papers [8] [41] [42]. Influence of temporal constraints to access control is probably the most detailed analyzed in [31] [4] while the influence of geospatial constraints is presented in [5] [14]. The importance of the time factor in workflow access control models is noticed in [22] [27] [37].

To the best of our knowledge, there are not many publications covering the research on context-sensitive constraints for workflows systems. Existing models only partially support context-sensitive constraints, usually by defining a new constraint type - *context constraint*, but to not facilitate influence of context on existing, well-known, access control constraints. The COBAC model enables context influence on business processes access control definition and enforcement by extending the notion of the role and the assignment relations with the context. The COBAC's constraint model we propose in this paper is based on the well-known RBAC constraints that are extended with context condition to support context-aware constraints.

## 3. The Overview of the COBAC Model

In this section, we give the overview of the COBAC's core concepts necessary for understanding the COBAC's constraints. Detail explanation of the core COBAC model can be found in [40]. Application of the COBACs prototype implementation for judicial processes access control is described in [24]. The core COBAC model is based on the standard RBAC model which is extended with following concepts: *business process*, *activity*, *context* and *resource category* (Figure 1). The reason for introducing *business process* and *activity* is to achieve efficient usage of the RBAC model in business processes because certain access control aspects need to be defined for a concrete process or its activities. There are cases when a business process can be fully executed within a user's session, but there

are also cases where the process execution is distributed through more users' sessions. Therefore, it is necessary that an access control mechanism supports both of those cases. By using *business process* and *activity* concepts, it is possible to bind certain access control segments with activities instead of binding them with a current session. This ensures the independence of these segments from the number of sessions in which activities are executed. In the COBAC model activities are assigned to roles that are authorized to execute them. During activity execution, it may be necessary to access to some resources. To execute an activity, permissions required for the resources being accessed are assigned to that activity. Since access control may be influenced by some other factors from a system and an environment, the given model is also extended by the notion of a *context*. The *categorization* of resources provides a definition of policies for the whole category of resources, and thus the number of necessary policies may be reduced.

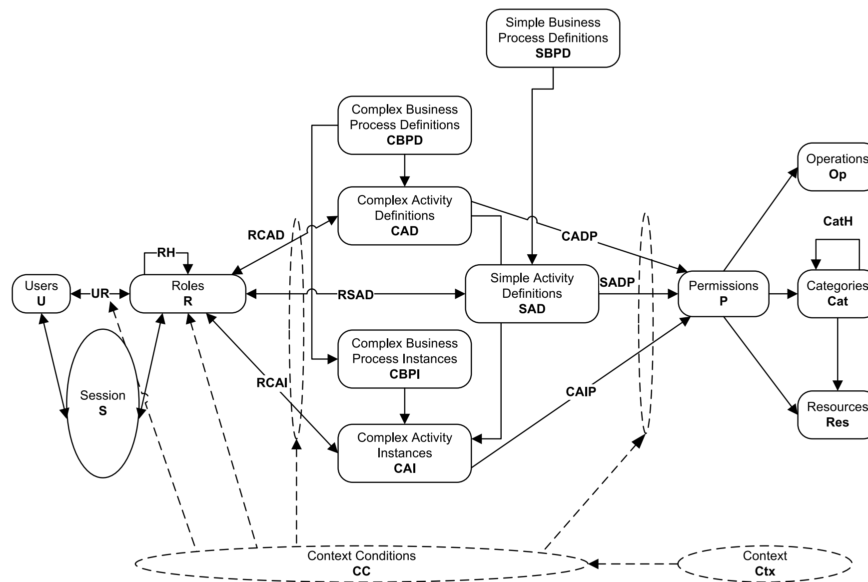


**Fig. 1.** COBAC model

The basic terms used to define the COBAC constraint model are: $U$ - users set, $R$ - roles set, $S$ - users sessions set, $CBPD$ - complex business processes definitions set, $CAD$ - complex activities definitions set, $CBPI$ - complex business processes instances set, $CAI$ - complex activities instances set, $SBPD$ - simple business processes definitions set, $SAD$ - simple activities definitions set, $P$ - permissions set, $Op$ - operations set, $Res$ - resources set, $Cat$ - categories set, $Ctx$ - context, and $CC$ - context conditions set. Relation of the COBACs static and dynamic constraint with basic elements are shown in the following sections.

### 3.1. Context

Basic context's ontology entities are presented in Figure 2. This model defines *ontology* for a context in business systems. This ontology models only basic concepts and relations between them. When the COBAC model is used in the concrete case, the ontology should be extended with new concepts and relations that are specific for the case. Two basic classes of the context model (see Figure 2) are *ContextFact* and *ContextExpression*. The class *ContextFact* represents the base context fact, while the class *ContextExpression* represents the context expression. Different context facts can be classified through the *ContextFact* specializations. The *Actor* class is used for representing different participants of events. Different activities are modeled by the *Action* class, and resources are modeled by the *Resource* class. The location is described by the *Location* class. The *Time* class is used for modeling time factor in the model. Different purposes in this model are represented by *Purpose* class, while means are described by the *Means* class.

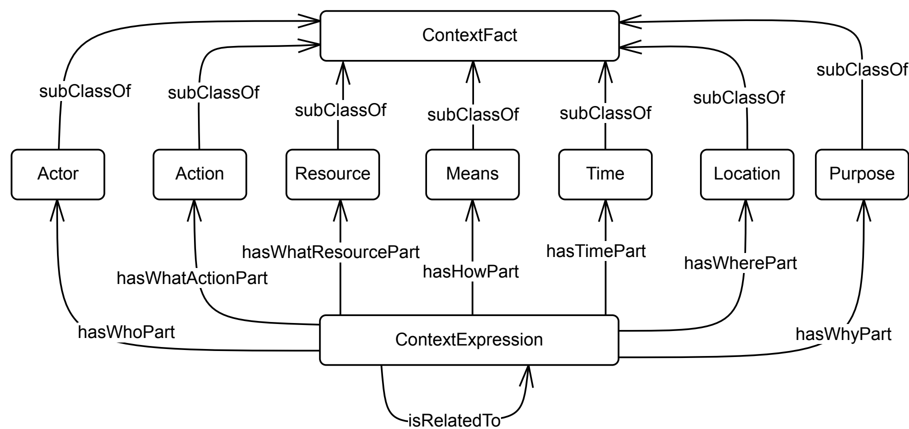


**Fig. 2.** COBAC's context entities and relations

The context expression (class *ContextExpression*) represents semantic binding of previous concepts, and it is based on the seven semantic dimensions (relations). Each of the afore-mentioned context facts creates one semantic dimension. Actually, the context expression describes events that took place and the conditions under which these events occurred. We extend five semantically dimensions defined in [2] [46] (*"who"*, *"what"*, *"where"*, *"when"* and *"how"*) with the notion *"why"* which defines purpose and with the notion *"related"* which defines relation between different context expressions. We also define two specializations of the *"what"* concept. The specialization *"what action"* is used for defining the *"what"* relation with an action, while the specialization *"what resource"* is used for defining the *"what"* relation with a resource. In our definition, the context expression must contain at least one *"who"* and *"what"* relation. We added this restriction because it is necessary that the context expression contains the information who/what did something, and what he/it did in order to describe an event.

We define the context condition as a logical expression which may consist of queries for searching context ontology (like SPARQL), context functions, logical operators and relational operators. The context functions are used to retrieve some current information from a system, like *who is current user*, or *what activity is being currently executed*. The more information about context condition can be found in [40].

The function $evalCond : CC \rightarrow \{\top, \bot\}$ verifies if the given context condition is satisfied. $CC$ contains two specific conditions: $\sigma$ ($\sigma = TRUE$) and $\bar{\sigma}$ ($\bar{\sigma} = FALSE$). The condition $\sigma$ is the condition which is always satisfied, i.e. $evalCond(\sigma) = \top$. The condition $\bar{\sigma}$ is the condition opposite to $\sigma$, i.e. it is never satisfied, $evalCond(\bar{\sigma}) = \bot$.

### 3.2. Business Process

The COBAC model defines two types of business process: *complex business process* and *simple business process*. A complex business process is defined as proposed in [11] [16]. It consists of a set of activities between which there is a proper order relation. Because there are many relatively simple tasks that do not require a workflow, we introduced a simple business process (consisting of only one activity) to represent them (e.g. create a notification). Access control policies in the COBAC model can be defined for definitions and instances of business processes. If a policy is defined for a process definition, it will be applied to all instances of that process while policies defined for instance will be applied only to the instance. In the case of complex business processes, COBAC supports the access control policies at process definition and the process instance level, while, in the case of simple business processes, only policies on the process definition level are supported, since their simplicity does not require special permissions for each instance.

The $i$-th complex business process ($cbpd_i$) is defined as: $cbpd_i = (CAD_i, F_{BPD_i})$, where:

- $CAD_i$ - The set of complex activity definitions of $cbpd_i$, i.e. $CAD_i = \{cad_{i1}, cad_{i2}, \cdots, cad_{in}\}$, $cad_{ij}$ is the definition of the $j$-th complex activity of the $i$-th complex business process.
- $F_{BPD_i}$ - The flow control relation which defines execution order of activities from the set $CAD_i$.

The given model specifies three types of complex activity:

- *start* - The first (starting) activity of a complex business process. We define starting activity as the first activity performed in a process instance. This activity can be bounded to a starting event of a process instance. Each complex business process definition must have exactly one activity of the start type.
- *end* - The last (ending) activity of a complex business process. Completing this activity takes the process to its end state (end event).
- *regular* - The regular activity that represents a part of a business logic implemented through a business process.

The function $typeOf : CAD_i \rightarrow \{start, regular, end\}$ determines the type of the complex activity, while the function $instanceOf_{BP} : CBPI \rightarrow CBPD$ determines the definition of the complex business process instance. The similar function is used to determine the definition of the complex activity instance: $instanceOf_A : CAI \rightarrow CAD$.

The function which determines to which complex business process definition belongs the complex activity definition is defined as:

$$activityDefOf : CAD \rightarrow CBPD$$
$$activityDefOf(cad_i) = cbpd_i \mid$$
$$cbpd_i = (CAD_i, F_{BPD_i}) \wedge cad_i \in CAD_i$$

Similarly as the $activityDefOf$ function, the $activityInstOf$ function is defined. This function determines to which complex business process instance belongs the complex activity instance.

As noted before, the $i$-th simple business process ($spd_i$) consists of a simple activity ($sad_i$), e.g. $spd_i = (sad_i)$.

### 3.3. Roles

Similar as in [4] [31], roles can be in one of three following states:

- *disabled* - A user can not activate roles in this state in her/his current session. Roles from this state can go to the *enabled* state.
- *enabled* - A user can activate roles in this state in her/his current session. From this state a role can go to the *disabled* or *active* state.
- *active* - This state applies to each user individually. If a user activates a role in her/his session, then that role goes from *enabled* to *active*, but only for the user. By deactivating a role, it goes from the *active* state to the *enabled* state only if the role was active in **exactly one** session of the user, otherwise the role will continue to be in the *active* state. From the *active* state, a role can also go to the *disabled* state.

Whether a role will be in the *enabled* or *disabled* state depends on the current context state. Therefore, we define role as tuple : $(rn, sc, sct)$ where:

- $rn$ - The role name,
- $sc$ - The context condition for enabling/disabling the role (*state condition*), and
- $sct$ - The context condition type (*state condition type*). Possible values are from the set $\{dc, ec\}$. It defines on what the condition applies to: the condition for disabling role ($dc$) or the condition for enabling role ($ec$).

If the condition $sc$ is satisfied the role will be in the *disabled* state if $sct = dc$, or in the *enabled* state if $sct = ec$. If the condition is not satisfied the role will be in the opposite state then defined by $sct$.

In the particular case, the $sc$ condition is defined as the context condition that is always satisfied: $(rn, \sigma, ec)$. This means that there is no context influence to disabling/enabling a role. A role is always *enabled*.

To determine the role state we introduce the following predicates:

- $disabled(r)$ - Verify if the role is disabled,
- $enabled(r)$ - Verify if the role is enabled, and
- $active(r, u)$ - Verify if the role $r$ is active for the user $u$.

The role hierarchy ($RH$) is defined as partial order over the roles set $R$ ($RH \subseteq R \times R$), i.e. as inheritance relation, marked as $\succeq$, where if the role $r_1$ inherits the role $r_2$ then stands $r_1 \succeq r_2$.

The predicate $canObtainRole(u, r)$ verifies if the role $r$ can be assigned to the user $u$ in the presence of role hierarchy:

$$
\begin{aligned}
canObtainRole(u, r_j) \Leftarrow & isRoleAssigned(u, r_j) \vee \\
& ((r_i \succeq r_j) \wedge canObtainRole(u, r_i))
\end{aligned}
$$

where the predicate $isRoleAssigned(u, r)$ verifies if the role $r$ is directly assigned to the user $u$.

### 3.4. Permissions

To improve the COBAC administration, resources can be organized into the categories, where a resource can be classified into more categories. The relation $assignCat(res, cat)$ classifies the resource $res \in Res$ into the $cat \in Cat$ category. Like roles, resource categories can also be hierarchically organized. The resource hierarchy is defined as partial order over the categories set $Cat$ ($CatH \subseteq Cat \times Cat$), i.e. as inheritance relation, marked as $\succeq_{cat}$, where if the category $cat_i$ inherits the category $cat_j$ then stands $cat_i \succeq_{cat} cat_j$. All resources that belong to the category $cat_i$ also belong to the category $cat_j$. The predicate $belongsToCat$ verifies if the resource belongs to the category.

$$
\begin{aligned}
belongsToCat(res, cat_j) \Leftarrow & assignCat(res, cat_j) \vee \\
& (cat_i \succeq_{cat} cat_j \wedge belongsToCat(res, cat_i))
\end{aligned}
$$

A permission that allows executing a certain operation can be defined for resources and categories. The permission defined for a category applies to all category's resources and subcategories.

The resource permission is defined as the tuple: $rp = (res, op), res \in Res, op \in Op$, while the category permission is defined as: $cp = (cat, op), cat \in Cat, op \in Op$. Let $RP$ be the set of all permissions defined for resources and $CP$ be the set of all permissions defined for categories then $P = RP \cup CP$.

The activity-permission assignment is specified with the following relations:

- $cadPermissionAssign(cad, p, cc)$ - the permission $p \in P$ is assigned to the complex activity definition $cad \in CAD$ if the context condition $cc \in CC$ is fulfilled,
- $caiPermissionAssign(cai, p, cc)$ - the permission $p \in P$ is assigned to the complex activity instance $cai \in CAI$ if the context condition $cc \in CC$ is fulfilled, and
- $sadPermissionAssign(sad, p, cc)$ - the permission $p \in P$ is assigned to the simple activity definition $sad \in SAD$ if the context condition $cc \in CC$ is fulfilled.

The functions that determine the privileges, defined for the given operation, associated to a complex activity definition or instance or simple activity definition are defined as follows:

$$
\begin{aligned}
& cadPermission : CAD \times Op \rightarrow 2^P \\
& cadPermission(cad, op) = \{p \mid p.op = op \wedge cadPermissionAssign(cad, p, cc)\}
\end{aligned}
$$

$$caiPermission : CAI \times Op \rightarrow 2^P$$
$$caiPermission(cai, op) = \{p \mid p.op = op \wedge caiPermissionAssign(cai, p, cc)\}$$

$$sadPermission : SAD \times Op \rightarrow 2^P$$
$$sadPermission(sad, op) = \{p \mid p.op = op \wedge sadPermissionAssign(sad, p, cc)\}$$

The permission $p_1$ is *contained* into permission $p_2$ ($p_1 \sqsubseteq p_2$) if (a) $p_1$ and $p_2$ are defined for the same operation and resource ($p_1 = p_2$), or (b) $p_1$ and $p_2$ are defined for the same operation and the $p_1$ category is a subcategory of the $p_2$ category, or (c) $p_1$ and $p_2$ are defined for the same operation and the $p_1$ resource belongs to $p_2$ category. This formally can be noted as follows:

$$\begin{aligned} p_1 \sqsubseteq p_2 \Rightarrow & (p_1 \in RP \wedge p_2 \in RP \wedge p_1.op = p_2.op \wedge p_1.res = p_2.res) \vee \\ & (p_1 \in CP \wedge p_2 \in CP \wedge p_1.op = p_2.op \wedge p_1.cat \succeq_{cat} p_2.cat) \vee \\ & (p_1 \in RP \wedge p_2 \in CP \wedge p_1.op = p_2.op \wedge belongsToCat(p_1.res, p_2.cat)) \end{aligned}$$

If an activity possess the permission $p_2$, and if that activity needs the permission $p_1$ to execute certain operation on a resource, then activity will be able to execute the operation only if $p_1 \sqsubseteq p_2$.

## 4. Static Constraints

Since the static constraints are enforced during the administration phase, they will prevent prohibited relations (assignments). In the paper [39] it is noted that they could be very restricted for business rules, especially for small organizations. However, they can be useful when certain business rules should be applied in a whole organization. The COBAC model identifies following static constraints (Figure 3):

- Static Separation of Duties (SSoD),
- Users-based Static Separation of Duties (USSoD), and
- Activity-based Static Separation of Duties (ASSoD).

### 4.1. Static Separation of Duties

Static Separation of Duties prevents a user from being a member of two conflicting roles. This type of constraint is rigorous and in real-life may be mapped to users current occupation. A simple example of this constraint may be found in judiciary procedures. If a user is a prosecutor, she/he may not act as a judge. Similarly, if a user is a judge than in no circumstances that the same user may appear as a prosecutor or a lawyer in any proceedings. The user's involvement is statically determined by its job position (occupation) and may be changed only if his occupation is changed and that change is reflected through administration of the user profile/rights. As soon as its current occupation is changed the new restrictions will apply, and that user is still prevented to act in conflicting activities (although now in changed capacities).

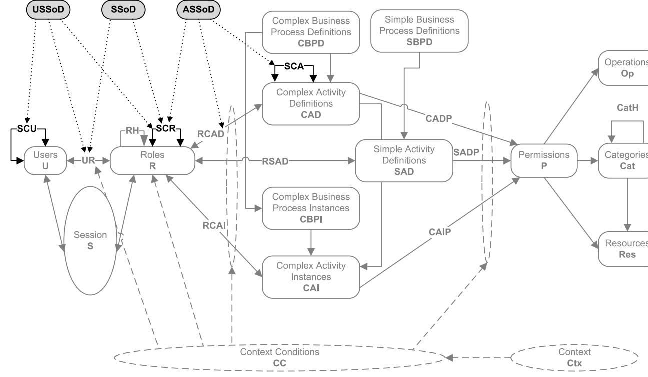


**Fig. 3.** COBAC's static constraints

The set of statically conflicting roles is defined as $SCR \subseteq R \times R$. Verification if two roles are mutually conflicting, in presence of role hierarchy, is defined with the predicate $sRolesInConflict$:

$$\begin{aligned} sRolesInConflict(r_i, r_j) \Leftarrow & \exists(r_i, r_j) \in SCR \vee \exists(r_j, r_i) \in SCR \vee \\ & r_i \succeq r_m \wedge sRolesInConflict(r_m, r_j) \vee \\ & r_j \succeq r_n \wedge sRolesInConflict(r_i, r_n) \end{aligned}$$

Now, the SSoD constraint can be formally defined as follows:

$$r_i, r_j \in R \wedge u \in U \wedge canCFObtainRole(u, r_i) \wedge canCFObtainRole(u, r_j) \Rightarrow \\ \neg sRolesInConflict(r_i, r_j)$$

where the predicate $canCFObtainRole(u, r)$ verifies if the role $r$ is assigned to the user $u$ without influence of the context. A static constraint is possible to define only for assignment relations without context influence because a context condition is evaluated during runtime while static constraints are enforced during administration phase. Therefore, the static constraint on relations with the context condition different than $\sigma$ can not be defined.

### 4.2. Users-based Static Separation of Duties

Users-based Static Separation of Duties represents an extension of the previous constraint. There are cases when it is insufficient to prevent a user from being a member of two or more conflicting roles, but it is necessary to prevent that any user from a certain group of mutually related users (*conflicting users*) be a member of conflict roles. This constraint prevents that mutually conflicting roles are assigned to any user from the set of conflicting users. USSoD may easily be found in a public services. Usual requirement is that close relatives of elected public officers may not occupy any post controlled or depicted by that elected official. Hence, based on the user membership to a particular group, she/he is prevented from taking part in any activity that may cause a conflict of interest for any other group member.

Let $SCU \subseteq U \times U$ be the set of related users:

$$(u_i, u_j) \in SCU \wedge r_i, r_j \in R \wedge canCFObtainRole(u_i, r_i) \wedge \\ canCFObtainRole(u_j, r_j) \Rightarrow \neg sRolesInConflict(r_i, r_j)$$

### 4.3. Activity-based Static Separation of Duties

Activity-based Static Separation of Duties prevents that two conflicting activities are executed by a user or conflicting users. The conflicting activities can belong to the same business process or different business processes. This constraint allows only conflicting roles to execute conflicting activities. Since it is the static constraint, it can be defined only for complex activity definitions because activity instances do not exist during enforcement. ASSoD is applicable wherever there is a requirement for any of two (or more phases) decision/approval process. For example, if expenditure approval requires co-signing, this requirement will prevent the same user to execute its right of signing more than once. Therefore, additional signing is forbidden activity for the same user.

Let $SCA \subseteq CAD \times CAD$ be the set of conflicting activity definitions:

$$((a_i, a_j) \in SCA \wedge r_i, r_j \in R) \wedge \\ (rcdActivityAssign(r_i, a_i, cc_i) \wedge cc_i = \sigma) \wedge \\ (rcdActivityAssign(r_j, a_j, cc_j) \wedge cc_j = \sigma) \Rightarrow sRolesInConflict(r_i, r_j)$$

where $rcdActivityAssign(r, cad, cc)$ defines relation which allows the role $r \in R$ to execute the complex activity definition $cad \in CAD$ if the context condition $cc \in CC$ is fulfilled.

## 5. Dynamic Constraints

Dynamic constraints are enforced during the execution of business processes. The COBAC model supports the definition of the context condition for each dynamic constraint. If a constraint has a context condition, then the constraint will be applied only if the condition is satisfied. The given model defines following dynamic constraints (Figure 4):

- Dynamic Separation of Duties (DSoD),
- Users-based Dynamic Separation of Duties (UDSoD),
- Activity-based Dynamic Separation of Duties (ADSoD),
- Dynamic Binding of Duties (DBoD), and
- Dynamic Execution Constraint (DEC).

### 5.1. Dynamic Separation of Duties

Dynamic separation of duties prevents that a user in her/his session activates two or more conflicting roles. An example may be found in online auction sites where users are required to register before participating in any auction. In each auction, user may act as a bidder (buyer) or a seller. But any user should be prevented to assume both roles in any session. The user may choose to participate in auction created by some other user. In this

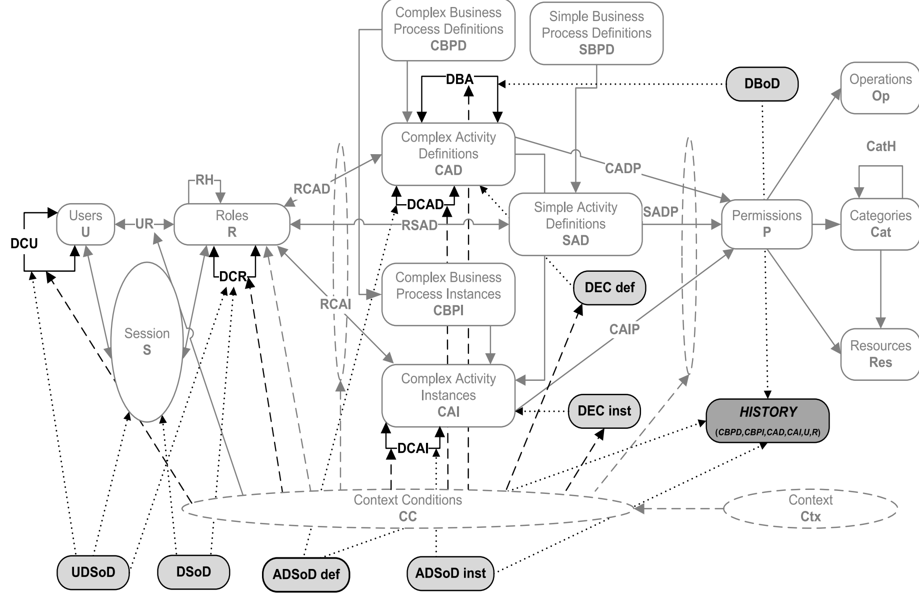


**Fig. 4.** COBAC's dynamic constraints

case, the user acts as a bidder and may place the bid for auctioned item. On second occasion the same user may be the one to put an item to auction. He creates an auction, which is then bound to this user. In this case, although he is a registered user he must be dynamically assigned only the role of the seller, preventing him to place the bid for its own auctioned item.

Let $DCR \subseteq R \times R \times CC$ be a set of dynamically conflicting roles. For each element $(r_i, r_j, cc) \in DCR$, roles $r_i$ and $r_j$ are in conflict if the condition $cc$ is satisfied. Verification if two roles are in conflict while role hierarchy is present, is defined with the predicate $dRolesInConflict$:

$$\begin{aligned} dRolesInConflict(r_i, r_j) \Leftarrow &(\exists (r_i, r_j, cc) \in DCR \wedge evalCond(cc)) \vee \\ &(\exists (r_j, r_i, cc) \in DCR \wedge evalCond(cc)) \vee \\ &r_i \succeq r_m \wedge dRolesInConflict(r_m, r_j) \vee \\ &r_j \succeq r_n \wedge dRolesInConflict(r_i, r_n) \end{aligned}$$

Now, DSoD constraint can be formally defined as follows:

$$\begin{aligned} &r_i, r_j \in R \wedge u \in U \wedge s \in S \wedge usersSession(u, s) \wedge \\ &activeInSession(s, r_i) \wedge activeInSession(s, r_j) \Rightarrow \\ &\neg dRolesInConflict(r_i, r_j) \end{aligned}$$

where the predicate $activeInSession(s, r)$ verifies if the role $r$ is active in the session $s$.

Algorithm 1 shows an example of the DSoD constraint enforcement. If there are two DSoD conflicts roles in the $ARS$ set, one of them is removed. The presented algorithm

will always remove the second role, but it is possible to insert some additional logic to determine which role will be removed.

**Algorithm 1** DSoD enforcement

```
NAME: EnforceDSoD
INPUT: ARS ⊆ R - set of user's roles in the session
OUTPUT: ARS ⊆ R - input set after DSoD enforcement

for each r_i, r_j ∈ ARS ∧ r_i ≠ r_j do
  if dRolesInConflict(r_i, r_j) then
    removeFromSet(r_j, ARS)
```

### 5.2. Users-based Dynamic Separation of Duties

This constraint extends the previous one to a set of users, by preventing that two mutually conflicting users in their session activate conflicting roles. It is useful because sometime it is much easier to commit fraud when two persons are involved, and each person has a different conflicting role. As this constraint is an extension of previous one, it can also be easily explained on an extended version of auction example. We may assume that items are put to the auction by the auction house, which may have several representatives, and that bidders may also be auction house representatives. In this case, it is necessary to prevent the representatives of the same auction house to act as bidders for the item that is placed for auction by the other representatives of the same auction house. If this would be allowed than auction would be easily manipulated to achieve higher bids.

Let $DCU \subseteq U \times U \times CC$ be the set of dynamically conflicting users. Then, similarly as for previous constraint, for each element $(u_i, u_j, cc) \in DCU$, the users $u_i$ and $u_j$ will be in conflict only if the condition $cc$ is satisfied. Verification if two users are in conflict is defined with the $usersInConflict$ predicate:

$$usersInConflict(u_i, u_j) \Leftarrow (\exists(u_i, u_j, cc) \in DCU \land evalCond(cc)) \lor (\exists(u_j, u_i, cc) \in DCU \land evalCond(cc))$$

UDSoD constraint can formally be defined as follows:

$$u_i, u_j \in U \land r_i, r_j \in R \land s_i, s_j \in S \land usersInConflict(u_i, u_j) \land usersSession(u_i, s_i) \land activeInSession(s_i, r_i) \land usersSession(u_j, s_j) \land activeInSession(s_j, r_j) \Rightarrow \neg dRolesInConflict(r_i, r_j)$$

where the predicate $usersSession(u, s)$ verifies if the session $s$ belongs to the user $u$ and the predicate $activeInSession(s, r)$ verifies if the role $r$ is active in the session $s$.

The function that determine conflict users $dConflictUsers : U \rightarrow 2^U$ is defined as follows:

$$dConflictUsers(u_i) = \{u_j \mid usersInConflict(u_i, u_j)\}$$

UDSoD enforcement is described by Algorithm 2. For each user who conflicts with the current user, it is verified if she/he has already activated a role that conflicts with any

role from the $ARS$ set. If such role exists, then the current user cannot activate her/his conflicting role, and therefore that role is removed from the active roles set $ARS$.

**Algorithm 2** UDSoD enforcement

```
NAME: EnforceUDSoD
INPUT: ARS ⊆ R - set of user's roles in the session
u ∈ U - user
OUTPUT: ARS ⊆ R - input set after UDSoD enforcement

CUS := dConflictUsers(u)
for each cu ∈ CUS do
   CURS := activeRoles(cu)
   for each ur ∈ ARS do
      for each cur ∈ CURS do
         if dRolesInConflict(ur, cur) then
            removeFromSet(ur, ARS)
```

### 5.3. Activity-based Dynamic Separation of Duties

Activity-based Dynamic Separation of Duties prevents that a user or conflicting users execute dynamically conflicting activities. An example of this type of constraint may be found in loan processing. We may assume that a bank employee may work in a loan approval department. The bank employee may request a loan, probably under the preferential terms from its bank. However, under no circumstances, she/he may be included in the process of loan approval, even though it is her/his regular duty. In this case, hers/his first activity in the process - requesting the loan, determines further constraints for his engagement with the process. Furthermore, she/he must be excluded from decision-making tasks in any subsequent process to refinance, or change the terms of the given loan. Therefore, this constraint must hold over the one or more processes as long as she/he is dynamically determined to be the loan user.

The conflicting activities can belong to the same process or different processes, i.e. the model supports intra-process and inter-process ADSoD constraints. This constraint can be defined for the conflicting complex activity instances and the conflicting complex activity definitions. A constraint defined for instances will apply only to those instances. If a constraint is defined for an activity definition, then there are two types of this constraint:

- *inter (inter-process, inter-instance)* - constraint will be applied on all definition's instances. If two definitions are in conflict, then all theirs instances are in conflict too.
- *intra (intra-process, intra-instance)* - applies only to the intra-process constraints. This constraint is applied only to activity instances that belong to the **same** process instance.

To define this type of dynamic constraint, we introduce *business process execution history*. It is defined as the tuple $(HISTORY, \succeq_H)$, where $HISTORY$ represents the set of previously executed activity instances and $\succeq_H$ is the partially order relation over the $HISTORY$ set. If $h_i \succeq_H h_j \wedge h_i, h_j \in HISTORY$ then the activity represented with $h_i$ is executed before the activity represented with the $h_j$ element. The $HISTORY$ set consists of elements defined as the tuple: $(cbpd, cbpi, cad, cai, u, r)$ where:

- $cbpd$ - complex business process definition ($cbpd \in CBPD$),

- *cbpi* - complex business process instance ($cbpi \in CBPI \wedge instanceOf_{BP}(cbpi) = cbpd$),
- *cad* - complex activity definition ($activityDefOf(cad) = cbpd$),
- *cai* - complex activity instance ($activityInstOf(cai) = cbpi$),
- $u$ - the user who executed $cai$ ($u \in U$), and
- $r$ - the role which allows the user to execute $cai$ ($r \in R$).

The predicate $wasExecutedInst(cai, u)$ verifies if the user $u$ has executed the activity instance $cai$, the predicate $wasExecutedDefInter(cad, u)$ verifies if the user $u$ has executes any instance of the activity definition $cad$ (the inter type), while the predicate $wasExecutedDefIntra(cbpi, cad, u)$ verifies if the user $u$ has executed instance of the activity definition $cad$ in the process instance $cbpi$ (the intra type).

$$wasExecutedInst(cai, u) \Leftarrow \exists(cbpd, cbpi, cad, cai, u, r) \in HISTORY$$
$$wasExecutedDefInter(cad, u) \Leftarrow \exists(cbpd, cbpi, cad, cai, u, r) \in HISTORY$$
$$wasExecutedDefIntra(cbpi, cad, u) \Leftarrow \exists(cbpd, cbpi, cad, cai, u, r) \in HISTORY$$

Let $DCAD \subseteq CAD \times CAD \times CC$ be the set of dynamically conflicting activity definitions for the inter type, wherein for each element $(cad_i, cad_j, cc) \in DCAD$, the activities $cad_i$ and $cad_j$ are in conflict if the context condition $cc$ is satisfied. Verification if two activity definitions are in conflict is defined with the $actDefInConflictrInter$ predicate:

$$\begin{aligned} actDefInConflictInter(cad_i, cad_j) \Leftarrow & (\exists(cad_i, cad_j, cc) \in DCAD \wedge evalCond(cc)) \vee \\ & (\exists(cad_j, cad_i, cc) \in DCAD \wedge evalCond(cc)) \end{aligned}$$

Now, ADSoD constraint (the inter type) in the case of conflict activity definitions can be formally defined as follows:

$$\begin{aligned} & cad_i, cad_j \in CAD \wedge actDefInConflictInter(cad_i, cad_j) \wedge \\ & (\exists h_i \in HISTORY \wedge h_i.cad = cad_i) \wedge (\exists h_j \in HISTORY \wedge h_j.cad = cad_j) \\ & \Rightarrow h_i.u \neq h_j.u \wedge \neg usersInConflict(h_i.u, h_j.u) \end{aligned}$$

Let the set of dynamically conflicting activity definitions of the intra type be defined as $DCADI \subseteq CAD \times CAD \times CC$, wherein for each element $(cad_i, cad_j, cc) \in DCADI$, the activities $cad_i$ and $cad_j$ are in conflict if the context condition $cc$ is satisfied. Verification if two activity definitions are in conflict is defined with $actDefInConflictIntra$:

$$\begin{aligned} actDefInConflictIntra(cad_i, cad_j) \Leftarrow & (\exists(cad_i, cad_j, cc) \in DCADI \wedge \\ & evalCond(cc)) \vee \\ & (\exists(cad_j, cad_i, cc) \in DCADI \wedge \\ & evalCond(cc)) \end{aligned}$$

The ADSoD constraint in the case of conflicting activity definitions for the intra process type can be formally defined as follows:

$$\begin{aligned} & cad_i, cad_j \in CAD \wedge actDefInConflictIntra(cad_i, cad_j) \wedge \\ & (\exists h_i \in HISTORY \wedge h_i.cad = cad_i) \wedge (\exists h_j \in HISTORY \wedge h_j.cad = cad_j) \wedge \\ & h_i.cbpi = h_j.cbpi \Rightarrow h_i.u \neq h_j.u \wedge \neg usersInConflict(h_i.u, h_j.u) \end{aligned}$$

In the similar manner, the ADSoD constraint for complex activity instances is defined. Let $DCAI \subseteq CAI \times CAI \times CC$ be the set of dynamically conflicting activity instances. The predicate $actInstInConflict$ verifies if two activity instances are in conflict:

$$\begin{aligned} actInstInConflict(cai_i, cai_j) \Leftarrow &(\exists(cai_i, cai_j, cc) \in DCAI \wedge evalCond(cc)) \vee \\ &(\exists(cai_j, cai_i, cc) \in DCAI \wedge evalCond(cc)) \end{aligned}$$

ADSoD constraint in the case of conflicting activity instances can formally be defined as follows:

$$\begin{aligned} &cai_i, cai_j \in CAI \wedge actInstInConflict(cai_i, cai_j) \wedge \\ &(\exists h_i \in HISTORY \wedge h_i.cai = cai_i) \wedge (\exists h_j \in HISTORY \wedge h_j.cai = cai_j) \\ &\Rightarrow h_i.u \neq h_j.u \wedge \neg usersInConflict(h_i.u, h_j.u) \end{aligned}$$

The functions for determining conflict definitions (inter and intra) and instances of complex activities are defined as follows:

$$\begin{aligned} &dConflictDefActivityInter : CAD \rightarrow 2^{CAD} \\ &dConflictDefActivityInter(cad_i) = \{cad_j \mid actDefInConflictInter(cad_i, cad_j)\} \\ &dConflictDefActivityIntra : CAD \rightarrow 2^{CAD} \\ &dConflictDefActivityIntra(cad_i) = \{cad_j \mid actDefInConflictIntra(cad_i, cad_j)\} \\ &dConflictInstActivity : CAI \rightarrow 2^{CAI} \\ &dConflictInstActivity(cai_i) = \{cai_j \mid actInstInConflict(cai_i, cai_j)\} \end{aligned}$$

Enforcement of ADSoD for activity instances is shown in Algorithm 3, while the enforcement of the Inter-ADSoD constraint is presented in Algorithm 4. As noted before, on activity instances are applied both: constraints defined for these instances and constraints defined for these instances definitions (inter and intra). The result of both algorithms is the input set from which are removed activities that violate the ADSoD constraint for the user.

**Algorithm 3** ADSoD enforcement for instances

```
NAME: EnforceADSoDInst
INPUT: AIS ⊆ CAI - activity instance set
: u ∈ U - user
OUTPUT: AIS ⊆ CAI - input set after ADSoD enforcement for instances

US := {u} ∪ dConflictUsers(u)
{Enforce for instances}
for each cai ∈ AIS do
   CAIS := dConflictInstActivity(cai)
   for each confAI ∈ CAIS do
      for each cu ∈ US do
         if wasExecutedInst(confAI, cu) then
            removeFromSet(cai, AIS)
{Enforce for definitions - inter type}
for each cai ∈ AIS do
   CADS := dConflictDefActivityInter(instanceOf_A(cai))
   for each confAD ∈ CADS do
      for each cu ∈ US do
         if wasExecutedDefInter(confAD, cu) then
            removeFromSet(cai, AIS)
{Enforce for definitions - intra type}
for each cai ∈ AIS do
   CADS := dConflictDefActivityIntra(instanceOf_A(cai))
   cbpi := activityInstOf(cai)
   for each confAD ∈ CADS do
      for each cu ∈ US do
         if wasExecutedDefIntra(cbpi, confAD, cu) then
            removeFromSet(cai, AIS)
```

**Algorithm 4** Inter-ADSoD enforcement for definitions

```
NAME: EnforceADSoDDefInter
INPUT: ADS ⊆ CAD - activity definition set
: u ∈ U - user
OUTPUT: ADS ⊆ CAD - input set after ADSoD enforcement for definitions

US := {u} ∪ dConflictUsers(u)
for each cad ∈ ADS do
   CAS := dConflictDefActivityInter(cad)
   for each confAD ∈ CAS do
      for each cu ∈ US do
         if wasExecutedDef(confAD, cu) then
            removeFromSet(cad, ADS)
```

### 5.4. Dynamic Binding of Duties

Dynamic Binding of Duties is opposite to the previous constraints. The user who executes an activity is obliged to execute some other bound activity. Such a binding enforces what is commonly known as case-based activity assignment model in business processes, where the prior knowledge of the activities in the process is beneficial for process execution. For example, an administrative procedure where a citizen is filing a certain request. There may be a large number of desk clerks that are capable of performing certain tasks. We may assume that all of them have been assigned the role of clerk. Any initial task to be performed by the role clerk may be assigned to any clerk randomly. However, it is far more efficient to assign any subsequent tasks aimed at the role clerk to the same

person, as she/he already has a prior knowledge of the case, and may fulfill this duty more expediently.

The set of dynamically bounded activities is defined as $DBA = \{(cad_i, cad_j, cc) \mid cad_i, cad_j \in CAD \wedge activityDefOf(cad_i) = activityDefOf(cad_j) \wedge cc \in CC\}$. This means that the COBAC constraint model allows binding only the activities from the same business process. This constraint type requires that a user who executes a bound activity in a business process instance must execute all other activities (for that instance) bound with the executed one. Similarly, as for separation of duties, the activities $cad_i$ and $cad_j$ will be bound if the context condition $cc$ is satisfied. The predicate $activityInBind$ verifies if the activities are bound:

$$\begin{aligned} activityInBind(cad_i, cad_j) \Leftarrow & (\exists (cad_i, cad_j, cc) \in DBA \wedge evalCond(cc)) \vee \\ & (\exists (cad_j, cad_i, cc) \in DBA \wedge evalCond(cc)) \end{aligned}$$

Formally, the DBoD constraint is represented as follows:

$$\begin{aligned} & cad_i, cad_j \in CAD \wedge activityInBind(cad_i, cad_j) \wedge \\ & (\exists h_i \in HISTORY \wedge h_i.cad = cad_i) \wedge (\exists h_j \in HISTORY \wedge h_j.cad = cad_j) \wedge \\ & (h_i.cbpi = h_j.cbpi) \Rightarrow h_i.u = h_j.u \end{aligned}$$

The function $dBindActivity : CAD \rightarrow 2^{CAD}$ determines binded activities and can be defined as follows:

$$dBindActivity(cad_i) = \{cad_j \mid activityInBind(cad_i, cad_j)\}$$

Algorithm 5 shows DBoD enforcement. The output of the algorithm is the pruned input set $AIS$. From this set are removed all activity instances that should be executed by another user according to the DBoD constraint.

**Algorithm 5** DBoD enforcement

```
NAME: EnforceDBoD
INPUT: AIS ⊆ CAI - activity instance set
u ∈ U - user
OUTPUT: AIS ⊆ CAI - input set after DBoD enforcement

for each cai ∈ AIS do
  cbpi := activityInstOf(cai)
  BAS := dBindActivity(instanceOf_A(cai))
  for each bind ∈ BAS do
    if ¬wasExecutedDefIntra(cbpi, bind, u) then
      removeFromSet(cai, AIS)
```

## 5.5. Dynamic Execution Constraint

This constraint represents an additional condition that needs to be satisfied to allow execution of the required activity. Dynamic Execution Constraint prevents a user to execute activity although she/he has the privilege to execute it with a role that is assigned to her/him. An example of this constraint may be found in a control process within power

distribution. Certain critical activities of control process (i.e. turn on/off major switches) are allowed only to personnel with the role operator, but for security reasons, this activity usually can be conducted only from specific workstations. For such activities, the dynamic execution constraint can define an additional condition that allows the activity execution only from permitted workstations.

Since this constraint can be defined for activity definitions and instances it is represented as the pair $(cad, cc), cad \in CAD, cc \in CC$ when it is defined for definitions, and as the pair $(cai, cc), cai \in CAI, cc \in CC$ when it is defined for instances. $DECD \subseteq CAD \times CC$ is the set of the dynamic execution constraints defined for activity definitions, and $DECI \subseteq CAI \times CC$ is the set of the dynamic execution constraints defined for instances. The execution of an activity instance, from the point of satisfaction of the execution constraints, is possible only if all execution constraints defined for that activity instance and its definition are satisfied.

The $dExecConstDefSatisf$ predicate verifies DEC defined for activity definitions, and same verification for activity instances is done with the $dExecConstInstSatisf$ predicate:

$$dExecConstDefSatisf(cad) \Leftarrow \forall(cad, cc) \in DECD \wedge evalCond(cc)$$
$$dExecConstInstSatisf(cai) \Leftarrow \forall(cai, cc) \in DECI \wedge evalCond(cc)$$

Algorithm 6 shows DEC enforcement for activity definitions. From the input set are removed all activities that not satisfy constraint. Similar process is used for activity instances (Algorithm 7), only in this case constraints defined for activity instances and for their definitions are applied.

**Algorithm 6** DEC enforcement for definitions

```
NAME: EnforceDECDef
INPUT: ADS ⊆ CAD - activity definition set
OUTPUT: ADS ⊆ CAD - input set after DEC enforcement

for each cad ∈ ADS do
  if ¬ dExecConstDefSatisf(cad) then
    removeFromSet(cad, ADS)
```

**Algorithm 7** DEC enforcement for instances

```
NAME: EnforceDECInst
INPUT: AIS ⊆ CAI - activity instance set
OUTPUT: AIS ⊆ CAI - input set after DEC enforcement

for each cai ∈ AIS do
  if ¬dExecConstInstSatisf(cai) ∨ ¬dExecConstDefSatisf(instanceOf_A(cai)) then
    removeFromSet(cai, AIS)
```

## 6. Dynamic Constraints Enforcement

While the static constraints are defined and enforced in the policies definition phase, when policies are created, the dynamic constraints are verified during runtime phase, as a part

of the access control enforcement activity. In this section, we described how the COBACs dynamic constraints validations are embedded within access control execution process.

The process of the access control enforcement defined by the COBAC core model is conducted through the following activities [40]:

– role activation,
– creation of task (activity) list containing activities that a user can execute,
– verification if the user is allowed to execute the activity at the moment when she/he initiate its execution, and
– verification if the user is allowed to access the requested resources during the activity execution.

Constraints enforcement is performed within first three phases, while the last one does not require any constraint verification. For completeness of the paper, all four phases are described.

### 6.1. Constraints and Variability of Context Condition

The context we propose is a dynamic category, meaning that context information can change over time and thus cause different results of context condition influenced by that information. To efficiently define and execute access control, some context conditions must have a period of time when the condition's result will not be changed. The length of this period depends on the type of constraint (or relation) where the condition is used. Depending on when is safe that context condition is changed we identified three types of conditions [40]:

– *slowly varying* (*session safe*) context conditions are those conditions that are not changed during a user session.
– *frequently varying* (*activity safe*) context conditions can be changed during a user session, but they are unchangeable during execution of each activity.
– *continuously varying* context conditions are those conditions can be changed any time, during a user session or an activity execution.

To define access control enforcement in a consistent way we assume that the context conditions specified in the user-role assignment relations and the context conditions used for enabling/disabling roles are *slowly varying*, while the conditions in the role-activity relations and the activity-permission relations are *frequently varying*. Based on the previous assumptions, roles assigned to a user in her/his session are **not changed** during the session duration, but it is possible that privileges (to execute certain activities) assigned to those roles are **changed**.

Since different constraints are verified in different moments (different phases of access control execution) then context conditions in the DSoD constraint and UDSoD are *slowly varying*, while context conditions in ADSoD, DBoD and DEC are *frequently varying*.

### 6.2. Roles Activation

The role activation process starts by creating a set of all roles assigned to the user (Algorithm 8). For each role from this set, the role condition is verified, and all roles in *disabled*

state (see Section 3.3) are removed from the set. The function *activateRolesForSession* uses this pruned set and activates some or all roles from it. Which roles are to be activated solely depends on business logic and requirements of an application. Then, the DSoD and UDSoD constraints are enforced on the *active* roles. The functions that perform the constraint enforcement return the input role set from which the roles that violate the given constraint are removed.

**Algorithm 8** An example of role activation

```
NAME: ActivateRoles
INPUT: u ∈ U - user
s ∈ S - session of u
OUTPUT: ARS - activated roles
URS := usersRoles(u)
for each r ∈ URS do
  if disabled(r) then
    removeFromSet(r, URS)
ARS := activateRolesForSession(URS, s)
ARS := EnforceDSoD(ARS)
ARS := EnforceUDSoD(ARS, u)
```

### 6.3. Activity List Creation

A common functionality of many workflow-based systems is a task (activity) list. This list is created runtime for each users session (or request) and contains all tasks currently assigned to the user that are ready to be executed. We define this activity list as the tuple $(SCPS, ECAS, ESAS)$ where:

- $SCPS \subseteq CAD \wedge \forall cad \in SCPS, typeOf(cad) = start$ (*Start Complex Process Set*) is a set of the complex activity definitions of "start" type that can be execute, i.e. the user can create a new instance of the complex business process.
- $ECAS \subseteq CAI$ (*Execute Complex Activity Set*) is the complex activity instances set that the user can execute.
- $ESAS \subseteq SAD$ (*Execute Simple Activity Set*) is the simple activities set that the user can execute.

All complex activity definitions of the *start* type which are assigned to the user's active roles comprise the $SCPS$ set. This set is created through three steps (Algorithm 9). In the first step, that is repeated for each user's role, the $assignedCompProcToStart(r)$ function determines all activities of the *start* type which are assigned to the user's role. Thus created set is then filtered by verifying the ADSoD and DEC constraints defined for activity definitions. In the case of the ADSoD constraints only constraints of the **inter** type are verified, while constraints of the **intra** type are not enforced. Since there is still no executed activities in this process instance because the *start* activity initiate the creation of a new process instance, the ADSoD intra type of conflict cannot exist. For the same reason, ADSoD and DEC constraints defined for instances are not necessary to enforce because still there is no process instance. As the DBoD constraint applies on the same instance of the business process and the *start* activity type is the first activity for any complex business process, for the $SCPS$ set it is not necessary to enforce the DBoD constraint. The functions that perform the constraint enforcement return the input activity set from which activities that violate the given constraint are removed.

**Algorithm 9** Activity list creation

```
NAME: CreateActivityList
INPUT: u ∈ U - user
ARS ⊆ R - active roles of user in session
OUTPUT: (SCPS, ECAS, ESAS) - user's actitivty list

{Create Start Complex Process Set (SCPS)}
SCPS := ⋃_{r∈ARS} assignedCompProcToStart(r)
SCPS := EnforceADSoDDefInter(SCPS, u)
SCPS := EnforceDECDef(SCPS)

{Create Execute Complex Activity Set (ECAS)}
ECAS := ⋃_{r∈ARS} assignedCompActToExec(r)
ECAS := EnforceADSoDInst(ECAS, u)
ECAS := EnforceDBoD(ECAS, u)
ECAS := EnforceDECInst(ECAS)

{Create Execute Simple Activity Set (ESAS)}
ESAS := ⋃_{r∈ARS} assignedSimpActToExec(r)
```

The $ECAS$ set contains all activity instances assigned to the user's active roles, and that should be executed next. The initial version of this set is created as the union of complex activity instances which all users active roles can execute (the union of results of $assignedCompActToExec$). Thus, created set is then pruned by enforcing the both types of ADSoD, DBoD, and DEC constraints.

The $SCPS$ set consists of simple activities that are assigned to the user's active roles. It is created by combining the results of the function $assignedSimpActToExec$ called for each users active role. Since the COBAC constraints cannot be defined for the simple business processes, verification of dynamic constraints for this set is not performed.

### 6.4. Action Execution

When a user selects an activity to execute (from her/his activity list) it is necessary, once more, to verify if the user can execute that activity. The reason for this additional verification is the existence of the *frequently varying* context conditions and the possibility that in the meantime a conflicting user executed the activity that onflicts with the activity that the user wants to execute.

Verification if it is allowed to execute a complex activity instance is presented in Algorithm 10. The user can execute activity instance if she/he has an active role with the privilege to execute that activity (the function $canExecCompActInst$) and if none of the assigned ADSoD, DBoD and DEC constraints are violated. The functions that perform the constraint enforcement return the input activity instance set from which activity instances that violate the given constraint are removed. Verification if any constraints are violated is reduced to verification if the $ECAS$ set is empty because, before constraint verification, this set contains only the activity instance that the user wants to execute.

The similar algorithm is used to verify if the execution of simple activities or creation of new process instance is allowed.

**Algorithm 10** Verification if it is allowed to execute an activity instance

```
NAME: ExecOfComplexActivityAllowed
INPUT: cai ∈ CAI - complex activity instance to execute
u ∈ U - user
ARS ⊆ R - active roles of user in session
OUTPUT: result - result, can user execute taks

result := false
for each r ∈ ARS do
  if canExecCompActInst(r, cai) then
    CAIS := {cai}
    CAIS := EnforceADSoDInst(CAIS, u)
    if CAIS ≠ ∅ then
      CAIS := EnforceDBoD(CAIS, u)
    if CAIS ≠ ∅ then
      CAIS := EnforceDECInst(CAIS)
    {If cai not removed from CAIS constraints allow execution of cai}
    if CAIS ≠ ∅ then
      result := true
```

### 6.5. Resource Access

Execution of an activity may require access to some resources. In this case, it is necessary to verify if the activity has permission to perform requested operation on the accessed resource.

**Algorithm 11** Permission verification

```
NAME: CanExecOperation
INPUT: cai ∈ CAI - complex activity instance
op ∈ Op - operation
res ∈ Res - resource
OUTPUT: result - result, can operation be executed on resource

result := false
p := (res, op)
RPS := caiPermission(cai, p.op) ∪ cadPermission(instanceOf_A(cai), p.op)
for each rp ∈ RPS do
  if p ⊑ rp ∧ evalCond(rp.cc) then
    result := true
```

Verification if the complex activity instance $cai$ can execute the operation $op$ on the resource $res$ is described by Algorithm 11 [40]. $cai$ will be allowed access to the resource if the permission that allows execution of $op$ on $res$ is **contained** in any of the permission assigned to the instance ($caiPermission$) or its definition ($cadPermission$) (see Section 3.4). Since the COBAC model does not provide constraints on resources access it is not necessary to verify them during this phase.

## 7. Conclusion

Access control mechanism that includes constraints for business processes may depend on different factors, which can vary from process to process. A possible solution for this problem is to use context-sensitive access control. In this paper, we present the context-sensitive constraints defined by the COBAC model. The presented constraints are based

on the common RBAC constraints listed in the literature which are extended with context-sensitive information and adopted for use in workflow systems.

Existing models for context dependant constraints for workflow access control only partially support context-sensitive constraints, usually by introducing new (context-based) constraint type. The most notable contributions of our solution is the extension of the existing constraints with the context condition and proposal how workflow access control can be enforced with such constraints. The presented constraints can be adopted for use in different workflow environments and use-cases. The main advantage of the proposed model is that it does not introduce new *context* constraints, but it extends existing access control constraints with the context. This can facilitate the process of access control policies definition since we are relying on well-known constraints. Security experts together with domain experts can extend those constraints with context when fine-grained access control is required. This process is probably simpler than using new constraints types and trying to achieve fine-granularity with the existing constraints and new ones.

Probably, the main obstacle in applying the COBAC model and its constraints is the proper extension of the context model for a specific purpose. We plan to investigate is it possible to provide (semi-)automatic generation of context parts from user requirements, data model, and business process model. Also, a proper tool that will integrate business process definition with constraints definition (including context) can facilitate this process.

Simulating a new business process is important activity since it minimizes the risk of an inappropriate business process being implemented. This process is usually performed by simulating the newly developed business process under various initial conditions and what-if scenarios using proper simulation tools [15]. So far we did not consider how the COBCACs access control policies can be verified using different business process simulation tools.

The proposed context-sensitive constraints model does not consider the quality of the context information, like accuracy, reliability, etc. This information can be very important for access control in certain cases. In future, we plan to examine how usage of the context quality influences constraint enforcement. So far, we do not detect if a context condition belongs to slowly, frequently or continuously varying context conditions type. Detection of the condition type can be crucial for usage of the model in systems when large contexts and a large number of context conditions are used. We plan to extend our context model to automate the detection of the context condition type.

## References


1. Abowd, G.D., Dey, A.K., Brown, P.J., Davies, N., Smith, M., Steggles, P.: Towards a better understanding of context and context-awareness. In: HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing. pp. 304–307. Springer-Verlag (1999)
2. Abowd, G.D., Mynatt, E.D., Rodden, T.: The human experience. IEEE Pervasive Computing 1(1), 48–57 (2002)
3. Bao, Y., Song, J., Wang, D., Shen, D., Yu, G.: A role and context based access control model with UML. In: International Conference for Young Computer Scientists. vol. 0, pp. 1175–1180. IEEE Computer Society (2008)

4. Bertino, E., Bonatti, P.A., Ferrari, E.: TRBAC: A temporal role-based access control model. ACM Trans. Inf. Syst. Secur. 4(3), 191–233 (2001)
5. Bertino, E., Catania, B., Damiani, M.L., Perlasca, P.: GEO-RBAC: a spatially aware RBAC. In: SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies. pp. 29–37. ACM (2005)
6. Bertino, E., Ferrari, E., Atluri, V.: The specification and enforcement of authorization constraints in workflow management systems. ACM Trans. Inf. Syst. Secur. 2(1), 65–104 (1999)
7. Bhatti, R., Bertino, E., Ghafoor, A.: A trust-based context-aware access control model for web-services. Distributed and Parallel Databases 18(1), 83–105 (2005)
8. Bhatti, R., Bertino, E., Ghafoor, A., Joshi, J.B.: XML-based specification for web services document security. Computer 37(4), 41–49 (2004)
9. Botha, R.A., Eloff, J.H.P.: Access control in document-centric workflow systems – an agent-based approach. Computers & Security 20(6), 525 – 532 (2001), http://www.sciencedirect.com/science/article/B6V8G-44416WY-F/2/d0d1f2c9bbd12fa52bc648371a9e5a39
10. Botha, R.A., Eloff, J.H.P.: Separation of duties for access control enforcement in workflow environments. IBM Systems Journal 40(3), 666–682 (2001)
11. Coalition, W.M.: Workflow management coalition the workflow reference model. TCOO- 1003 (1994)
12. Covington, M.J., Long, W., Srinivasan, S., Dev, A.K., Ahamad, M., Abowd, G.D.: Securing context-aware applications using environment roles. In: Proceedings of the 6th ACM Symposium on Access Control Models and Technologies (SACMAT). pp. 10–20. ACM (2001)
13. Crampton, J.: A reference monitor for workflow systems with constrained task execution. In: SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies. pp. 38–47. ACM (2005)
14. Damiani, M.L., Bertino, E., Catania, B., Perlasca, P.: GEO-RBAC: A spatially aware RBAC. ACM Trans. Inf. Syst. Secur. 10(1), 2 (2007)
15. Damij, N., Boškoski, P., Bohanec, M., Boshkoska, B.M.: Ranking of business process simulation software tools with dex/qq hierarchical decision model. PloS one 11(2), e0148391 (2016)
16. Davenport, T.H., Short, J.E.: The new industrial engineering: Information technology and business process redesign. Sloan Management Review 31(4), 11–27 (1990)
17. Dey, A.K.: Understanding and using context. Personal Ubiquitous Comput. 5(1), 4–7 (2001)
18. Fadhel, A.B., Bianculli, D., Briand, L.: A comprehensive modeling framework for role-based access control policies. Journal of Systems and Software 107(0), 110 – 126 (2015)
19. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Transactions on Information and System Security (TISSEC) 4(3), 224–274 (2001)
20. Filho, J.B., Martin, H.: Using context quality indicators for improving context-based access control in pervasive environments. In: EUC '08: Proceedings of the 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing. pp. 285–290. IEEE Computer Society (2008)
21. de Freitas Bulcao Neto, R., da Graca Campos Pimentel, M.: Toward a domain-independent semantic model for context-aware computing. In: Proceedings of the 3rd Latin American Web Congress (LA-WEB). pp. 61–70. IEEE Computer Society (2005)
22. Gao, L., Zhang, L., Xu, L.: Access control scheme for workflow. In: ICCET '09: Proceedings of the 2009 International Conference on Computer Engineering and Technology. pp. 215–217. IEEE Computer Society (2009)
23. Georgiadis, C.K., Mavridis, I., Pangalos, G., Thomas, R.K.: Flexible team-based access control using contexts. In: SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies. pp. 21–27. ACM (2001)

24. Gostojić, S., Sladić, G., Milosavljević, B., Konjović, Z.: Context-sensitive access control model for government services. Journal of Organizational Computing and Electronic Commerce 22(2), 184–213 (2012)
25. Haibo, S., Fan, H.: A context-aware role-based access control model for web services. Proceedings of the IEEE International Conference on e-Business Engineering (ICEBE) 0, 220–223 (2005)
26. Han, W., Zhang, J., Yao, X.: Context-sensitive access control model and implementation. Proceedings of the 5th International Conference on Computer and Information Technology (CIT) 0, 757–763 (2005)
27. Irwin, K., Yu, T., Winsborough, W.H.: Enforcing security properties in task-based systems. In: SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies. pp. 41–50. ACM (2008)
28. Kapsalisa, V., Hadellisb, L., Karelisb, D., Koubiasc, S.: A dynamic context-aware access control architecture for e-services. Computers & Security 25(7), 507–521 (2006)
29. Kong, G., Li, J.: Research on rbac-based separation of duty constraints. Journal of Information and Computing Science 2(3), 235–240 (2007)
30. Kumar, A., Karnik, N., Chafle, G.: Context sensitivity in role-based access control. SIGOPS Oper. Syst. Rev. 36(3), 53–66 (2002)
31. Latif, U., Joshi, J.B.D., Bertino, E., Ghafoor, A.: A generalized temporal role-based access control model. IEEE Trans. on Knowl. and Data Eng. 17(1), 4–23 (2005)
32. Oh, S., Park, S.: Task-role-based access control model. Information Systems 28(6), 533–562 (2003)
33. Perelson, S., Botha, R., Eloff, J.: Separation of duty administration. SACJ/SART - South African Computer Journal 1(27), 64–69 (2001)
34. Schefer-Wenzl, S., Strembeck, M.: Modeling context-aware rbac models for business processes in ubiquitous computing environments. In: Mobile, Ubiquitous, and Intelligent Computing (MUSIC), 2012 Third FTRA International Conference on. pp. 126–131. IEEE (2012)
35. Schefer-Wenzl, S., Strembeck, M.: Modelling context-aware rbac models for mobile business processes. International Journal of Wireless and Mobile Computing 3 6(5), 448–462 (2013)
36. Schilit, B., Adams, N., Want, R.: Context-aware computing applications. In: Proc of IEEE Workshop on Mobile Computing Systems and Applications. pp. 85–91. IEEE Computer Society (1994)
37. Shafiq, B., Samuel, A., Ghafoor, H.: A GTRBAC based system for dynamic workflow composition and management. In: Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC). pp. 284–290. IEEE Computer Society (2005)
38. Shang, C., Yang, Z., Liu, Q., Zhao, C.: A context based dynamic access control model for web service. In: International Conference on Embedded and Ubiquitous Computing, IEEE/IFIP. vol. 2, pp. 339–343. IEEE Computer Society (2008)
39. Simon, R., Zurko, M.E.: Separation of duty in role-based environments. In: CSFW '97: Proceedings of the 10th IEEE workshop on Computer Security Foundations. p. 183. IEEE Computer Society (1997)
40. Sladić, G., Milosavljević, B., Konjović, Z.: Context-sensitive access control model for business processes. Computer Science and Information Systems (ComSIS) 10(3), 939–972 (2013)
41. Sladić, G., Milosavljević, B., Konjović, Z., Vidaković, M.: Access control framework for XML document collections. Computer Science and Information Systems (ComSIS) 8(3), 591–609 (2011)
42. Sladić, G., Milosavljević, B., Surla, D., Konjović, Z.: Flexible access control framework for MARC records. The Electronic Library 30(5), 25 (2012)
43. Strembeck, M., Neumann, G.: An integrated approach to engineer and enforce context constraints in rbac environments. ACM Trans. Inf. Syst. Secur. 7(3), 392–427 (2004)

44. Thomas, R.K., Sandhu, R.S.: Task-based authorization controls (TBAC): A family of models for active and enterprise-oriented autorization management. In: Proceedings of the IFIP TC11 WG11.3 11th International Conference on Database Securty XI. pp. 166–181. Chapman & Hall, Ltd. (1998)
45. Tripathi, A.R., Kulkarni, D., Ahmed, T.: A specification model for context-based collaborative applications. Pervasive Mob. Comput. 1(1), 21–42 (2005)
46. Truong, K.N., Abowd, G.D., , Brotherton, J.A.: Who, what, when, where, how: Design issues of capture & access applications. In: Ubicomp 2001: Ubiquitous Computing. pp. 209–224. Springer-Verlag (2001)
47. Wainer, J., Barthelmess, P., Kumar, A.: W-RBAC - a workflow security model incorporating controlled overriding of constraints. International Journal of Cooperative Information Systems 12(4), 455–485 (2003)
48. Warner, J., Atluri, V.: Inter-instance authorization constraints for secure workflow management. In: SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies. pp. 190–199. ACM (2006)
49. Yao, L., Kong, X., Xu, Z.: A task-role based access control model with multi-constraints. In: NCM '08: Proceedings of the 2008 Fourth International Conference on Networked Computing and Advanced Information Management. pp. 137–143. IEEE Computer Society (2008)
50. Zhang, L., Luo, L., Zhang, L., Geng, T., Yue, Z.: Task-role-based access control in application on MIS. In: Proceedings of the 2006 IEEE Asia-Pacific Conference on Services Computing (APSCC). pp. 153–159. IEEE Computer Society (2006)

**Gordana Milosavljević** is holding the associate professor position at University of Novi Sad, Faculty of Technical Sciences. She teaches courses in Business Information Systems and Model Driven Software Development. Her research interests focus on software engineering methodologies, rapid development tools and enterprise information systems design.

**Goran Sladić** is holding the associate professor position at the Faculty of Technical Sciences, University of Novi Sad since 2016. Mr. Sladi received his PhD degree at the University of Novi Sad in 2011. His research interests include information security and privacy, document management systems, context-aware computing, software engineering and workflow systems. He is the corresponding author.

**Branko Milosavljević** received a PhD degree at the University of Novi Sad in 2003. He is holding the position of full professor at the University of Novi Sad, Faculty of Technical Sciences. He has authored or co-authored more than 80 scientific papers. Most of these publications are related to software engineering, net-centric computing, document management, digital libraries and access control.

**Miroslav Zarić** received his PhD in Electrical engineering and computing from University of Novi Sad, Serbia. He is currently an assistant professor at University of Novi Sad, Faculty of Technical Sciences. His research interest include business process management and process automation, service oriented architectures. He is a member of ACM.

**Stevan Gostojić** is assistant professor of applied computer science and informatics at Faculty of Technical Sciences, University of Novi Sad. He has a Ph.D. in electrical engineering and computer science from the same university. His research interests are: legal informatics, e-government and information society services; knowledge engineering,

knowledge based systems, linked data and semantic web; document engineering, document management and business process management; and open data, open knowledge and open government.

**Jelena Slivka** is holding the assistant professor position at the Faculty of Technical Sciences, Novi Sad, Serbia since 2015. Ms. Slivka received her Master degree (2008) and PhD degree (2014) all in Computer Science from the University of Novi Sad, Faculty of Technical Sciences. Since 2009 she is with the Faculty of Technical Science in Novi Sad. Her main research interests are data mining and machine learning.

*Received: June 28, 2016; Accepted: July 29, 2017.*