Generic Support for RBAC Break-Glass Policies in Process-Aware Information Systems

Sigrid Schefer-Wenzl
Institute for Information Systems and New Media
WU Vienna, Austria
sigrid.schefer-wenzl@wu.ac.at

Mark Strembeck
Institute for Information Systems and New Media
WU Vienna, Austria
mark.strembeck@wu.ac.at

ABSTRACT

We present a break-glass extension for process-related role-based access control (RBAC) models. Our extension ensures the static (design-time) and dynamic (runtime) consistency of corresponding break-glass models. The extension is generic in the sense that it can, in principle, be used to extend arbitrary process-aware information systems or process modeling languages with support for process-related RBAC and corresponding break-glass policies. We implemented a library and runtime engine that provides full platform support for all properties of our approach.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access Controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security; Business Processes; Management;

Keywords

Role-Based Access Control; Break-Glass Policies; Process-aware Information Systems;

1. INTRODUCTION

Process-aware information systems (PAIS) can be configured via process models that define all expected execution paths for each business process (see, e.g., [29]). Corresponding access control models specify which subjects are authorized to perform the tasks that are included in the business processes (see, e.g., [26, 30]). While such an approach is well suited for process instances that conform to one of the expected execution scenarios, we encounter problems when dealing with exceptional situations, e.g., when no authorized subject is available to execute a particular task in case of emergency (see, e.g., [19, 31]). Exception Handling refers to actions that are executed when deviations appear between what is planned and what is actually happening (see, e.g., [4, 5, 17, 29]).

In recent years, role-based access control (RBAC) [7, 15] has developed into a de facto standard for access control in both, research and industry. In RBAC, roles correspond to different job-positions and scopes of duty within a particular organization or information system [24]. Access permissions are assigned to roles according to the tasks this role has to accomplish. Via its role memberships, each subject acquires all permissions that are necessary to fulfill its duties. Several extensions for RBAC exist for different application domains. In a business process context, RBAC has been extended to consider access permissions for tasks included in a business process (see, e.g., [10, 13, 28, 30]).

In many organizational environments some critical tasks exist which – in exceptional cases – must be performed by a subject although he/she is usually not permitted to perform these tasks. For example, if a deadline is about to expire and the senior lawyer is not available, a junior lawyer may be authorized to submit a written objection to the court in order to avoid damage to the company. In case of emergency, machine operators are authorized to switch production machines into an emergency state to ensure safety for personnel and machinery. In a hospital context, a junior physician shall be able to perform certain tasks of a senior physician in case of emergency. Accordingly, a PAIS has to provide mechanisms that help to coordinate exception handling activities.

Break-glass policies have been introduced as a sophisticated exception-handling mechanism for access control policies. They supplement ordinary access control policies in order to allow the controlled overriding of access rights (see, e.g., [3, 9, 11, 18]). Break-glass (the term is a metaphor relating to the act of breaking the glass to pull a fire alarm) refers to the possibility for a subject, who is not authorized to execute a task, to gain authorization for this task in exceptional cases. Therefore, subjects should only make use of break-glass policies if a regular task execution is not possible (e.g., no authorized subject is available). If a break-glass policy is activated, the resulting task executions must be carefully recorded for later audit and review. Typically, a special review process is triggered to monitor such break-glass executions.

In this paper, we provide a generic metamodel to formally embed the break-glass policy concept into process-related RBAC models [26]. This metamodel can be used to extend arbitrary process modeling languages or process

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'13 March 18-22, 2013, Coimbra, Portugal.
Copyright 2013 ACM 978-1-4503-1656-9/13/03 ...$10.00.

engines. To demonstrate our approach, we defined a UML extension based on this generic metamodel that allows for modeling break-glass policies via extended UML Activity diagrams (see [22]). Moreover, we also implemented a break-glass extension to the BusinessActivity library and runtime engine (see [25, 26]). The source code of our implementation is available for download [1].

The remainder of this paper is structured as follows. Section 2 presents our formal metamodel for break-glass RBAC models in business processes. Section 3 discusses related work and Section 4 concludes the paper.

2. GENERIC METAMODEL FOR PROCESS-RELATED BREAK-GLASS POLICIES

To support the definition of break-glass policies in a business process context, we formally embed them into our generic metamodel for Process-Related RBAC models (see [26]). In particular, we specify that certain *breakable tasks* can be performed by subjects who are usually not allowed to execute these tasks. For this purpose, override rules regulate that members of a certain role are permitted to perform a certain task in case of emergency (*breakable-by-role* override). In addition to role-based break-glass rules, our approach enables the definition of subject-specific break-glass rules, i.e. only a certain subject is authorized to execute a task in case of emergency (*breakable-by-subject* override). Breakable-by-subject override rules are used in cases where only certain members of a role have all necessary competencies to perform the breakable task. Each break-glass execution will be recorded and monitored via a corresponding review process.

The subsequent definitions provide a generic framework for integrating break-glass policies into a business process context. Based on the extended metamodel presented below, we defined a UML extension [22] and implemented a break-glass extension to the BusinessActivity library and runtime engine [1] to demonstrate our approach. For the purposes of this paper, Definition 1 repeats some of the definitions for Process-Related RBAC models (for details see [23, 26]). New definitions for *Process-Related Break-Glass RBAC model*s are introduced in Definitions 2, 3, 4, and 5.

DEFINITION 1. ***(Process-Related RBAC Model)*** *Let S be a set of subjects, R a set of roles, P_T a set of process types, P_I a set of process instances, T_T a set of task types, T_I a set of task instances, and CC a set of context constraints. 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, $Q = rsa \cup tra \cup pi \cup ti \cup es \cup er$ to mappings that establish relationships, and $D = sme \cup dme \cup sb \cup rb \cup linked_{CC} \cup fulfilled_{CC}$ to mutual exclusion, binding, and context constraints. For the partial mappings of the metamodel ($\mathcal{P}$ 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 $s \in S$* subject *and $R_s \subseteq R$ the* set of roles assigned to this subject *(the set of roles owned by s). This assignment implies a mapping* **role ownership** *$rown : S \mapsto \mathcal{P}(R)$ that returns all direct and inherited roles for a subject. The mapping $rown^{-1} : R \mapsto \mathcal{P}(S)$ determines all subjects owning a role, directly or transitively via the role-hierarchy.*

3. *The mapping $tra : R \mapsto \mathcal{P}(T_T)$ is called* **task-to-role assignment**. *For $tra(r) = T_r$, we call $r \in R$* role *and $T_r \subseteq T_T$ is called the* set of tasks assigned to r. *This assignment implies a mapping* **task ownership** *$town : R \mapsto \mathcal{P}(T_T)$ which returns all tasks for a role. The mapping $town^{-1} : T_T \mapsto \mathcal{P}(R)$ determines the set of roles a task is assigned to, directly or transitively via the role-hierarchy.*

4. *The mapping $ptd : P_T \mapsto \mathcal{P}(T_T)$ is called* **process type definition**. *For $ptd(p_T) = T_{p_T}$, we call $p_T \in P_T$* process type *and $T_{p_T} \subseteq T_T$ the* set of task types associated with p_T.

5. *The mapping $pi : P_T \mapsto \mathcal{P}(P_I)$ is called* **process instantiation**. *For $pi(p_T) = P_i$, we call $p_T \in P_T$* process type *and $P_i \subseteq P_I$ the* set of process instances instantiated from process type p_T.

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 $es : T_I \mapsto S$ is called* **executing-subject** *mapping. For $es(t) = s$, we call $s \in S$ the* executing-subject *and $t \in T_I$ is called the* executed task instance.

8. *The mapping $er : T_I \mapsto R$ is called* **executing-role** *mapping. For $er(t) = r$, we call $r \in R$ the* executing-role *and $t \in T_I$ is called the* executed task instance.

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.

13. *The mapping $linked_{CC} : T_T \mapsto \mathcal{P}(CC)$ is called* **context constraint to task linkage**. *For $linked_{CC}(t) = CC_T$, we call $t \in T_T$* constrained task *and $CC_T \subseteq CC$ the set of* context constraints linked to this task.

14. *The mapping* $fulfilled_{CC} : CC \mapsto BOOLEAN$ *is called* **context constraint fulfillment**. *For* $fulfilled_{CC}(cc) = boolean$, *we call* $cc \in CC$ context constraint. *The mapping follows a two-valued logic returning exactly one truth value (true or false). Thus, the* $fulfilled_{CC}$ *mapping returns* true *iff all conditions linked to the context constraint are true.*

The following definitions provide an extension to the metamodel for Process-related RBAC models defined in [26]. Definition 2 first specifies the new elements for Process-Related Break-Glass RBAC models.

Definition 2. ***(Process-Related Break-Glass RBAC Model).*** *Let PRBGM = (E,Q,D,BG) be a Process-Related Break-Glass RBAC Model, where E refers to the pairwise disjoint sets of the metamodel, Q to mappings that establish relationships, D to binding, mutual exclusion, and context constraints, and BG to mappings for break-glass policies. Below, we define the additional mappings for break-glass policies BG (*$\mathcal{P}$ *refers to the power set):*

1. To define which role is authorized to perform a certain task, task types are assigned to roles via the task-to-role assignment mapping *tra* (see Definition 1.3). In addition, *breakable* tasks that are assigned to roles can be executed in a break-glass scenario:
   The mapping $bbr : R \mapsto \mathcal{P}(T_T)$ *is called* **breakable-by-role override**. *For* $bbr(r) = T_b$, *we call* $r \in R$ role *and* $T_b \subseteq T_T$ *is called the* set of breakable tasks assigned to r. *The mapping* $bbr^{-1} : T_T \mapsto \mathcal{P}(R)$ *returns all roles a particular task is assigned to via the bbr-mapping.*

2. The breakable-by-role override mapping implies a mapping to determine all breakable tasks that are assigned to a particular role. Note that in a role-hierarchy, each role owns the tasks that are directly assigned to this role, as well as the tasks inherited from its junior-roles (see Definition 1.1):
   The mapping $btown : R \mapsto \mathcal{P}(T_T)$ *is called* **break-glass task ownership**. *For each* $r \in R$, *the tasks inherited from its junior-roles are included, i.e.* $btown(r) = \bigcup_{r_{inh} \in rh^*(r)} bbr(r_{inh}) \cup bbr(r)$.
   The mapping $btown^{-1} : T_T \mapsto \mathcal{P}(R)$ *determines the set of roles a task is assigned to via a break-glass override assignment (directly or transitively via a role hierarchy). The btown mapping complements the task ownership mapping (town) from Definition 1.3.*

3. Breakable tasks can also be directly assigned to subjects:
   The mapping $bbs : S \mapsto \mathcal{P}(T_T)$ *is called* **breakable-by-subject override**. *For* $bbs(s) = T_b$, *we call* $s \in S$ subject *and* $T_b \in T_T$ *is the* set of breakable tasks assigned to s. *The mapping* $bbs^{-1} : T_T \mapsto \mathcal{P}(S)$ *returns all subjects assigned to a particular breakable task via the bbs-mapping.*

4. A certain task instance is said to be "broken" if it is executed by a subject via a break-glass override assignment:
   The mapping $broken_{T_I} : T_I \mapsto BOOLEAN$ *is called* **broken task instance** *mapping. For* $broken_{T_I}(t_b) = boolean$, *we call* $t_b \in T_I$ task instance *with* $t_b \in ti(t_T, p_x)$. *The mapping follows a two-valued logic returning exactly one truth value (true or false):* $broken(t_b) = true$ *if* $es(t_b) \in bbs^{-1}(t_T) \vee es(t_b) \in rown^{-1}(r)$ *with* $t_t \in bbr(r)$.

5. A certain process instance is said to be "broken" if it includes at least one broken task instance (see Definition 2.4):
   The mapping $broken_{P_I} : P_I \mapsto BOOLEAN$ *is called* **broken process instance** *mapping. For* $broken_{P_I}(p_b) = boolean$, *we call* $p_b \in P_I$ process instance. *The mapping follows a two-valued logic returning exactly one truth value (true or false):* $broken(p_b) = true$ *if* $\exists t_b \in ti(t_T, p_b)$ *with* $broken(t_b) = true$.

6. To determine if the use of a break-glass policy was justified, the execution of broken task instances needs to be monitored and reviewed. Thus, if a certain process instance is broken, a corresponding review process is triggered. In particular, a review process has to check all broken task instances included in the broken process instance:
   The mapping $review : P_T \mapsto P_T$ *is called* **review process definition**. *For* $review(p_b) = p_r$, *we call* $p_b \in P_T$ process type *and* $p_r \in P_T$ review process type.

As defined above, break-glass overrides enable certain subjects or roles to perform certain tasks in emergency situations only. Therefore, the runtime allocation of ordinary tasks on the one hand and tasks that are allocated via a break-glass override on the other hand must be clearly separated. In particular, this means that a subject cannot accidentally perform a break-glass task. Instead, it must actively and explicitly choose to use a break-glass override. In this context, it is important to discuss the different implications of entailment constraints, such as mutual exclusion and binding constraints, defined in RBAC models.

SME constraints define that two statically mutual exclusive tasks must never be assigned to the same role and must never be performed by the same subject. This type of constraint is global with respect to all process instances in the corresponding information system. Therefore, SME constraints do not only affect runtime task execution, they already affect the task-to-role and role-to-subject assignment relations at design-time (see, e.g., [21, 25, 27, 32]). Thus, if we want to define that certain subjects or members of a certain role are allowed to perform two SME tasks in exceptional (emergency) situations, we must explicitly define a corresponding break-glass override via the *bbr* or *bbs* mappings (see Definition 2).

In contrast, DME constraints define that two dynamically mutual exclusive 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 (see, e.g., [21, 25, 27, 32]). In a break-glass scenario, DME tasks are different from SME tasks because one (or more) subjects may legally own two DME tasks (and are thereby competent and empowered to perform both tasks–see, e.g., [11]). Thus, in case a subject already owns two DME tasks via the *tra* and

rsa mappings (see Definition 1) we do not need to define an additional *bbr* or *bbs* override assignment for the same tasks. Instead, we "only" need to allow that these subjects are permitted to break the DME constraint (of tasks they already own) in emergency situations. An abuse of this option is prevented because a break-glass allocation is always conducted on purpose and cannot be performed accidentally, and because each broken process instance is reviewed (see Definition 2.6).

In contrast to mutual exclusion constraints, binding constraints define that the *same* role or subject who performed a $task_x$ must also perform a bound $task_y$. Therefore, bound tasks *must* be assigned to the same subject or role in order to ensure the satisfiability of the corresponding business processes (see, e.g. [6, 20]). However, in a break-glass scenario it may be necessary to break a binding constraint and perform a break-glass reallocation for tasks that have already been allocated due to the transitivity of binding constraints (see also [25]). For example, such a situation may arise if the subject who is allocated to a $task_y$ because of a binding constraint has an accident and therefore cannot perform $task_y$. In such a situation, we can perform a break-glass reallocation if the delay of $task_y$ would result in an emergency. Again, an abuse of this option is prevented because a break-glass allocation is always conducted on purpose and cannot be performed accidentally, and because each broken process instance is reviewed (see Definition 2.6).

Based on the discussion above, we define two types of correctness for Process-Related Break-Glass RBAC models. *Static correctness* refers to the design-time consistency of the elements and relationships in the Break-Glass RBAC Model. *Dynamic correctness* refers to the compliance of process instances with the break-glass definition as well as with entailment and context constraints at runtime. Definition 3 provides static correctness rules that must hold in addition to the rules for Process-Related RBAC models presented in [26].

DEFINITION 3. ***(Static Correctness)***. *Let PRBGM = (E,Q,D,CX,BG) be a Process-Related Break-Glass RBAC Model. PRBGM is said to be statically correct if the following requirements hold:*

1. Each role is allowed to own a task either regularly or via a break-glass override assignment. To separate regular task ownerships from break-glass task ownerships, we need to ensure that no task is assigned to a certain role via both mappings:
   $\forall t_T \in T_T : town^{-1}(t_T) \cap btown^{-1}(t_T) = \emptyset$

2. Each subject is allowed to own a task either regularly (via its role memberships) or via a *breakable-by-role* override assignment. To separate regular task ownerships from breakable task ownerships, we need to ensure that no task is assigned to a certain subject via both mappings:
   $\forall t_T \in T_T, r_1, r_2 \in R$ `with` $t_T \in btown(r_1)$ `and` $t_T \in town(r_2) : rown^{-1}(r_1) \cap rown^{-1}(r_2) = \emptyset$

3. Each subject is allowed to own a task either regularly (via its role memberships) or via a *breakable-by-subject* override assignment. To separate regular task ownerships from breakable task ownerships, we need to ensure that no task is assigned to a certain subject via both mappings:
   $\forall t_T \in T_T, r \in R$ `with` $t_T \in town(r) : rown^{-1}(r) \cap bbs^{-1}(t_T) = \emptyset$

Definition 4 specifies rules for the dynamic correctness of Process-Related Break-Glass RBAC models. These rules need to be fulfilled at runtime, i.e. when executing a certain process instance. They extend the dynamic correctness rules for Process-Related RBAC models specified in [26].

DEFINITION 4. ***(Dynamic Correctness)***. *Let PRBGM = (E,Q,D,CX,BG) be a Process-Related Break-Glass RBAC Model and P_I its set of process instances. PRBGM is said to be dynamically correct if the following requirements hold:*

1. For each broken process instance, there has to exist a corresponding review process:
   $\forall p_b \in pi(p_T)$ `with` $broken(p_b) = true : \exists p_r \in review(p_T)$

2. For all broken task instances within the same process instance, the executing-subjects of SME tasks do *not* have to be different:
   `if` $\exists p_b \in P_I$ `with` $broken(p_b) = true$ `then`
   $\forall t_x \in ti(t_1, p_b), \forall t_y \in ti(t_2, p_b)$ `with` $t_2 \in sme(t_1)$ `and` $broken(t_x) = true$:
   $(es(t_x) \neq es(t_y) \ \vee \ es(t_x) = es(t_y))$

3. For all broken task instances within the same process instance, the executing-subjects of DME tasks do *not* have to be different:
   `if` $\exists p_b \in P_I$ `with` $broken(p_b) = true$ `then`
   $\forall t_x \in ti(t_1, p_b), \forall t_y \in ti(t_2, p_b)$ `with` $t_2 \in dme(t_1)$ `and` $broken(t_x) = true$:
   $(es(t_x) \neq es(t_y) \ \vee \ es(t_x) = es(t_y))$

4. For all broken task instances within the same process instance, the executing-role of role-bound tasks does *not* have to be the same:
   `if` $\exists p_b \in P_I$ `with` $broken(p_b) = true$ `then`
   $\forall t_x \in ti(t_1, p_b), \forall t_y \in ti(t_2, p_b)$ `with` $t_2 \in rb(t_1)$ `and` $broken(t_x) = true$:
   $(er(t_x) \neq er(t_y) \ \vee \ er(t_x) = er(t_y))$

5. For all broken task instances within the same process instance, the executing-subject of subject-bound tasks does *not* have to be the same:
   `if` $\exists p_b \in P_I$ `with` $broken(p_b) = true$ `then`
   $\forall t_x \in ti(t_1, p_b), \forall t_y \in ti(t_2, p_b)$ `with` $t_2 \in sb(t_1)$ `and` $broken(t_x) = true$:
   $(es(t_x) \neq es(t_y) \ \vee \ es(t_x) = es(t_y))$

6. For all broken task instances within the same process instance, context constraints do *not* have to be fulfilled (remember that all broken process instances are reviewed, see Definitions 2.6 and 4.1):
   `if` $\exists p_b \in P_I$ `with` $broken(p_b) = true$ `then`
   $\forall t_x \in ti(t_T, p_b)$ `with` $broken(t_x) = true$
   `if` $cc_x \in linked_{CC}(t_T)$ `then`
   $(fulfilled_{CC}(cc_x) = true \ \vee$
   $fulfilled_{CC}(cc_x) = false)$

Furthermore, the execution history of a process instance p must reflect which subject has executed which task instance. For this purpose, Definition 5 specifies the execution history $h(p)$ of a Process-Related Break-Glass RBAC model which includes a record of all broken process instances.

DEFINITION 5. ***(Execution History)***. *Let $PRBGM = (E, Q, D, CX, BG)$ be a Process-Related Break-Glass RBAC Model and P_I its set of process instances. For a particular process instance $p \in P_I$, an execution event $exec(p) \in (T_I \times T_T \times R \times S)$ is a record of a particular task execution where T_I refers to the set of task instances, T_T to the set of corresponding task types, R to the set of executing-roles, and S to the set of executing-subjects. The execution history $h(p)$ of a process instance p is defined as a mapping $h : P_I \mapsto \mathcal{P}(\{(t_x, t_t, r, s) \mid t_x \in T_I, t_t \in T_T, r \in R, s \in S\})$, which maps $h(p)$ to a set of execution events $exec(p)$ (for further details, see [26]).*
The execution history includes a record of all broken process instances. For a particular broken process instance, i.e. $broken(p_i) = true$, the broken task instances, corresponding executing-subjects, and executing-roles are documented. The break-glass execution history $h_b(p_b)$ of a process instance p_b is defined as a mapping $h_b : P_I \mapsto \mathcal{P}(\{(t_b, t_t, r_b, s_b) \mid t_b \in T_I, t_t \in T_T, broken(t_b) = true, s_b = es(t_b), r_b = er(t_b)\})$ with $h_b \subseteq h$.

3. RELATED WORK

Several approaches exist to integrate break-glass policies into access control models. For example, the optimistic security principle [14] aims to handle exceptional cases. In [14], any access is legitimate and is thus granted. A similar approach is presented by Ardagna et al. [2]. They introduce a break-glass approach where an action can be performed either by finding a corresponding emergency policy or by granting a break-glass override. In both approaches, the enforcement of security policies is retrospective, and relies on administrators to detect unreasonable accesses and subsequently take steps to compensate undesired behavior. Such approaches, however, cause an immense burden for administrators.

The break-the-glass RBAC (BTG-RBAC) model [8] specifies for each permission-to-role assignment if a break-glass override is allowed or not. Moreover, obligations can be specified to define arbitrary actions that must be performed in case of a break-glass override. In [3], a break-glass extension for SecureUML is provided. The resulting SecureUML break-glass policies can then be transformed into XACML. However, this approach does not consider break-glass decisions in connection with dynamic mutual exclusion or binding constraints.

Only few contributions exist to integrate the concept of break-glass policies into a business process context. In [30] Wainer et al. present an RBAC model for workflow systems. They also extend this model via exception handling functionalities that allow the controlled overriding of entailment constraints in case of emergency. To achieve this, each constraint is associated with a certain level of priority. On the other hand, roles hold override privileges according to their level of responsibility. A comprehensive overview of exception handling patterns – including resource reallocation – is provided in [19].

Several other approaches exist that deal with process adaptations and process evolutions in order to flexibly handle different types of exceptions in process-aware information systems. For example, [16] provides a formal model to support dynamic structural changes of process instances. A set of change operations is defined that can be applied by users in order to modify a process instance execution path, while maintaining its structural correctness and consistency. In [31], change patterns and change support features are identified and several process management systems are evaluated regarding their ability to support process changes. Exception handling via structural adaptations of process models are also considered in [17]. In particular, several correctness criteria and their application to specific process metamodels are discussed. In [12], a survey on flexibility criteria for business process management systems is presented. Amongst others, clearly defined responsibilities for tasks and sophisticated exception handling mechanisms are identified as important flexibility requirements for process-aware information systems. In comparison to our work, these approaches have in common that processes must be changed in order to handle exceptional situations. The main goal of our approach is to maintain the designed process flow, while ensuring that only authorized subjects are allowed to participate in a workflow.

4. CONCLUSION

In this paper, we presented a break-glass extension for process-related RBAC models. In order to handle emergency scenarios in a controlled manner, break-glass policies define which subjects are allowed to execute certain tasks in case of emergency. Our approach is generic in the sense that it can be used to extend arbitrary process-aware information systems or process modeling languages with support for process-related RBAC and corresponding break-glass policies. We used our approach to define a break-glass extension for UML activity diagrams [22] and to implement an extension for the BusinessActivity library and runtime engine[1].

5. REFERENCES

[1] Business Activity Library and Runtime Engine. Available at: `http://wi.wu.ac.at/home/mark/BusinessActivities/library.html`, 2012.

[2] C. A. Ardagna, S. D. C. di Vimercati, S. Foresti, T. W. Grandison, S. Jajodia, and P. Samarati. Access control for smarter healthcare using policy spaces. *Computers & Security*, 29(8):848–858, 2010.

[3] A. D. Brucker and H. Petritsch. Extending Access Control Models with Break-Glass. In *Proceedings of the 14th ACM symposium on Access control models and technologies (SACMAT)*, 2009.

[4] F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Trans. Database Syst.*, 24:405–451, September 1999.

[5] D. K. W. Chiu, Q. Li, and K. Karlapalem. A meta modeling approach to workflow management systems supporting exception handling. *Inf. Syst.*, 24:159–184, April 1999.

[6] J. Crampton and H. Khambhammettu. Delegation and Satisfiability in Workflow Systems. In *Proceedings of the 13th ACM symposium on Access control models and technologies (SACMAT)*, 2008.

[7] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House, second edition edition, 2007.

[8] A. Ferreira, D. Chadwick, P. Farinha, R. Correia, G. Zao, R. Chilro, and L. Antunes. How to Securely Break into RBAC: The BTG-RBAC Model. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, December 2009.

[9] A. Ferreira, R. Cruz-Correia, L. Antunes, P. Farinha, E. Oliveira-Palhares, D. W. Chadwick, and A. Costa-Pereira. How to Break Access Control in a Controlled Manner. In *Proceedings of the 19th IEEE Symposium on Computer-Based Medical Systems*, 2006.

[10] C. K. Georgiadis, I. Mavridis, G. Pangalos, and R. K. Thomas. Flexible Team-Based Access Control Using Contexts. In *Proceedings of the sixth ACM symposium on Access control models and technologies (SACMAT)*, May 2001.

[11] S. Marinovic, R. Craven, J. Ma, and N. Dulay. Rumpole: A Flexible Break-Glass Access Control Model. In *Proceedings of the 16th ACM symposium on Access control models and technologies (SACMAT)*, 2011.

[12] S. Nurcan. A Survey on the Flexibility Requirements Related to Business Processes and Modeling Artifacts. In *Proceedings of the Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS)*, January 2008.

[13] S. Oh and S. Park. Task-Role-Based Access Control Model. *Information Systems*, 28(6), 2003.

[14] D. Povey. Optimistic Security: A New Access Control Paradigm. In *Proceedings of the 1999 workshop on New security paradigms (NSPW)*, 2000.

[15] H. F. Ravi Sandhu, Edward Coyne and C. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2), 1996.

[16] M. Reichert and P. Dadam. Adept_flex-Supporting Dynamic Changes of Workflows Without Losing Control. *J. Intell. Inf. Syst.*, 10(2), 1998.

[17] M. Reichert, S. Rinderle-Ma, and P. Dadam. Flexibility in Process-Aware Information Systems. In *Transactions on Petri Nets and Other Models of Concurrency II*. 2009.

[18] E. Rissanen, B. S. Firozabadi, and M. Sergot. Towards a Mechanism for Discretionary Overriding of Access Control. In *Proceedings of the 12th International Workshop on Security Protocols*, 2004.

[19] N. Russell, W. M. van der Aalst, and A. H. M. T. Hofstede. Exception Handling Patterns in Process-Aware Information Systems. In *International Conference on Advanced Information Systems Engineering (CAiSE)*, 2006.

[20] S. Schefer, M. Strembeck, and J. Mendling. Checking Satisfiability Aspects of Binding Constraints in a Business Process Context. In *Proc. of the BPM Workshop on Workflow Security Audit and Certification (WfSAC)*, 2011.

[21] S. Schefer, M. Strembeck, J. Mendling, and A. Baumgrass. Detecting and Resolving Conflicts of Mutual-Exclusion and Binding Constraints in a Business Process Context. In *Proc. of the 19th International Conference on Cooperative Information Systems (CoopIS)*, October 2011.

[22] S. Schefer-Wenzl and M. Strembeck. A UML Extension for Modeling Break-Glass Policies. In *5th International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA)*, 2012.

[23] S. Schefer-Wenzl and M. Strembeck. Modeling Context-Aware RBAC Models for Business Processes in Ubiquitous Computing Environments. In *Proc. of the 3rd International Conference on Mobile, Ubiquitous and Intelligent Computing (MUSIC)*, June 2012.

[24] M. Strembeck. Scenario-Driven Role Engineering. *IEEE Security & Privacy*, 8(1), 2010.

[25] M. Strembeck and J. Mendling. Generic Algorithms for Consistency Checking of Mutual-Exclusion and Binding Constraints in a Business Process Context. In *Proc. of the 18th International Conference on Cooperative Information Systems (CoopIS)*, 2010.

[26] M. Strembeck and J. Mendling. Modeling Process-related RBAC Models with Extended UML Activity Models. *Information and Software Technology*, 53(5), 2011.

[27] K. Tan, J. Crampton, and C. A. Gunter. The Consistency of Task-Based Authorization Constraints in Workflow Systems. In *Proceedings of the 17th IEEE workshop on Computer Security Foundations*, June 2004.

[28] R. K. Thomas and R. S. Sandhu. Task-Based Authorization Controls (TBAC): A Family of Models for Active and Enterprise-Oriented Autorization Management. In *Proceedings of the IFIP TC11 WG11.3 Eleventh International Conference on Database Securty XI: Status and Prospects*, August 1997.

[29] W. M. P. van der Aalst, M. Rosemann, and M. Dumas. Deadline-based Escalation in Process-Aware Information Systems. *Decision Support Systems*, 43:492–511, March 2007.

[30] J. Wainer, P. Barthelmess, and A. Kumar. W-RBAC - A Workflow Security Model Incorporating Controlled Overriding of Constraints. *International Journal of Cooperative Information Systems (IJCIS)*, 12(4), 2003.

[31] B. Weber, S. Rinderle, and M. Reichert. Change Patterns and Change Support Features in Process-Aware Information Systems. In *International Conference on Advanced Information Systems Engineering (CAiSE)*, 2007.

[32] C. Wolter, A. Schaad, and C. Meinel. Task-Based Entailment Constraints for Basic Workflow Patterns. In *Proceedings of the 13th ACM symposium on Access control models and technologies (SACMAT)*, 2008.