Information and Software Technology 56 (2014) 1289–1308

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Model-driven specification and enforcement of RBAC break-glass policies for process-aware information systems

Sigrid Schefer-Wenzl [a,b,*], Mark Strembeck [b,*]

[a] Competence Center for IT-Security, University of Applied Sciences Campus Vienna, Austria
[b] Institute for Information Systems and New Media, WU Vienna, Austria

ARTICLE INFO

Article history:
Received 11 April 2013
Received in revised form 10 February 2014
Accepted 7 April 2014
Available online 18 April 2014

Keywords:
Access control
Business process modeling
Model-driven development
UML

ABSTRACT

Context: In many organizational environments critical tasks exist which – in exceptional cases such as an emergency – must be performed by a subject although he/she is usually not authorized to perform these tasks. Break-glass policies have been introduced as a sophisticated exception handling mechanism to resolve such situations. They enable certain subjects to break or override the standard access control policies of an information system in a controlled manner.
Objective: In the context of business process modeling a number of approaches exist that allow for the formal specification and modeling of process-related access control concepts. However, corresponding support for break-glass policies is still missing. In this paper, we aim at specifying a break-glass extension for process-related role-based access control (RBAC) models.
Method: We use model-driven development (MDD) techniques to provide an integrated, tool-supported approach for the definition and enforcement of break-glass policies in process-aware information systems. In particular, we provide modeling support on the computation independent model (CIM) layer as well as on the platform independent model (PIM) and platform specific model (PSM) layers.
Results: Our approach is generic in the sense that it can be used to extend process-aware information systems or process modeling languages with support for process-related RBAC and corresponding break-glass policies. Based on the formal CIM layer metamodel, we present a UML extension on the PIM layer that allows for the integrated modeling of processes and process-related break-glass policies via extended UML Activity diagrams. We evaluated our approach in a case study on real-world processes. Moreover, we implemented our approach at the PSM layer as an extension to the BusinessActivity library and runtime engine.
Conclusion: Our integrated modeling approach for process-related break-glass policies allows for specifying break-glass rules in 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., [65]). Corresponding access control policies specify which subjects are authorized to perform the tasks that are included in the business processes (see, e.g., [58,60,64,67]). While this 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., [45,69]). Exception Handling refers to actions that are executed when deviations appear between what is planned and what is actually happening (see, e.g., [15,16,40,65]).

1.1. Motivation

In recent years, role-based access control (RBAC) [22,38] 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 [58]. Access permissions are assigned to roles according to the tasks this role has to accomplish, and subjects (e.g., human users) are assigned to roles according to their work

* Corresponding author at: Competence Center for IT-Security, University of Applied Sciences Campus Vienna, Austria. Tel.: +43 1 606 68 77 2134.
E-mail addresses: sigrid.schefer-wenzl@fh-campuswien.ac.at (S. Schefer-Wenzl), mark.strembeck@wu.ac.at (M. Strembeck).

http://dx.doi.org/10.1016/j.infsof.2014.04.010
0950-5849/

profiles. Thereby, 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., [25,32,60,64,67]).

In many organizational environments some critical tasks exist which – in exceptional cases – must be performed by a subject although he/she is usually not authorized 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. They supplement ordinary access control policies in order to allow the controlled overriding of access rights (see, e.g., [4,12,13,24,28,41]). *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 triggered, 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.

To effectively enforce processes and related break-glass definitions in PAIS, we need to integrate the different concepts into a consolidated modeling approach. Although a number of sophisticated approaches exist that allow for the formal specification and analysis of process-related access control policies and constraints (see, e.g., [8,42,71]), corresponding modeling support for process-related break-glass policies is largely missing. In this paper, we integrate the notion of break-glass policies into a business process context and thereby support the exception handling of access control policies in PAIS.

1.2. Approach synopsis

We apply model-driven development (MDD) techniques (see, e.g., [54–56]) to support the integrated modeling and execution of break-glass policies and business processes. In MDD, domain-specific languages (DSLs) can be defined that provide domain abstractions as first-class language elements (see, e.g., [29,62]). A DSL can either be a standalone language or it is embedded into a host language to extend it with domain-specific language abstractions. The aim of this paper is to provide domain-specific modeling support for the model-driven specification of process-related break-glass policies at the business process modeling-level (see Fig. 1).

In the MDD context, a computation-independent model (CIM) defines a certain domain (or subdomain) at a generic level. The CIM is independent of a particular modeling language or technology. A CIM can be used to build a platform-independent model (PIM) of the corresponding domain. While it is independent of any platform, and thereby neutral from an implementation point of view, the PIM is typically specified in a particular modeling language (for example via MOF-based languages such as BPMN or UML [33,35,36]) and describes the structure of a system, the elements/results that are produced by a system, or the control and object flow in a system. Finally, a platform-specific model (PSM) describes the realization/implementation of a software system via platform-specific technologies and tools. The intention for choosing a model-driven approach is to separate the business/application logic from the underlying platform technology (see, e.g., [54–56]).

The main contribution of this paper is an integrated approach for the specification and enforcement of break-glass policies in process-related RBAC models. Our integrated modeling approach for process-related break-glass policies and corresponding business processes serves as an enabler to document and communicate which break-glass policies can be applied when executing a certain process in case of emergency. To achieve this, we extend our previous contributions from [50,52]: Our approach is based on a metamodel which formally integrates the core elements of process models and break-glass policies at the CIM layer. This metamodel was presented in [52]. In addition, we introduced modeling support at the PIM layer for process-related break-glass RBAC models in [50] via *extended UML2 Activity diagrams*. In this paper, we consolidate the results from [50,52] and present the following novel contributions:

- We define the dynamic semantics of our metamodel at the CIM layer via a set of generic algorithms and procedures, which check and ensure the runtime consistency of our extended process models. These algorithms are independent of a particular programming language and/or software platform and can be used to implement consistency checks for our formal metamodel.
- We implemented a break-glass extension to the Business Activity Library and Runtime engine (see [59,60]) to provide support for platform specific models (PSM). All concepts introduced at the CIM and PIM layers can be seamlessly mapped to the

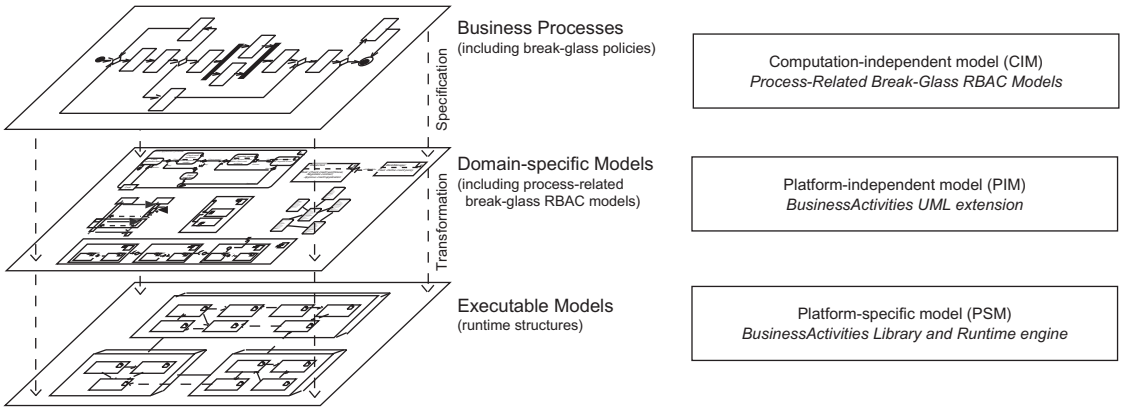

Fig. 1. Model-driven development approach.

corresponding runtime models that are managed with our software platform. The source code of the Business Activity Library and Runtime Engine is available for download from [1].
- Moreover, we report on the findings of a case study that we conducted to evaluate the applicability of our approach on real-world processes.

The remainder of this paper is structured as follows. Section 2 gives an overview of process-related RBAC models and introduces a motivating example for integrating break-glass policies into business processes. Section 3 presents our formal CIM layer metamodel for break-glass RBAC models in business processes. Subsequently, Section 4 introduces our extension for modeling break-glass models at the PIM layer via extended UML2 Activity diagrams. Moreover, we formally define the semantics of our newly introduced modeling elements via OCL constraints. Section 5 presents the results from a case study demonstrating the practical applicability of our approach for real-world business processes. Furthermore, Section 6 gives an overview of our extended software platform to manage process-related break-glass RBAC models at the PSM layer. Section 7 discusses related work and illustrates how our approach can be used to specify tailored break-glass policies for certain application domains. Section 8 concludes the paper.

2. Background

In this Section, we will first give an overview of process-related RBAC models introduced in [60]. Subsequently, we will present a motivating example for integrating break-glass policies into the generic metamodel from [60].

2.1. Process-related RBAC models

Each *task* in a process (e.g., to edit a patient record) is typically associated with certain access permissions (e.g., to read and write the patient record). Therefore, *subjects* participating in a workflow, i.e. human users or software-agents, must be authorized to perform the tasks needed to complete the process (see, e.g., [25,32]). In RBAC, a *role* is an abstraction containing the tasks of a certain subject-type (see, e.g., [57,67]).

In addition, RBAC supports the definition of different types of entailment constraints. A *task-based entailment constraint* places some restriction on the subjects who can perform a $task_x$ given that a certain subject s_1 has performed $task_y$. Thus, task-based entailment constraints have an impact on the combination of subjects and roles who are allowed to execute particular tasks (see, e.g., [19,44,49,59,60,63,68,72]). Examples of entailment constraints include static mutual exclusion (SME), dynamic mutual exclusion (DME), subject-binding (SB), and role-binding (RB) constraints. A SME constraint defines that two statically mutual exclusive tasks must never be *assigned* to the same subject. In turn, DME tasks can be assigned to the same role, but within the *same process instance* they must be *executed* by *different* subjects. A SB constraint defines that two bound tasks must be performed by the *same individual* within the same process instance. A RB constraint defines that bound tasks must be performed by *members of the same role*, but not necessarily by the same individual.

Moreover, in an IT-supported workflow, *context constraints* can be defined as a means to consider context information in access control decisions (see, e.g. [7,61]). Typical examples for context constraints in organizational settings regard the temporal or spatial context of task execution, user-specifc attributes, or the task execution history of a user (see, e.g., [20]). RBAC supports the definition of context constraints on various parts of an RBAC model (see, e.g., [25,61,68]). In this paper, a context constraint is a modeling level concept to support conditional task execution. In particular, context constraints define that certain contextual attributes must meet certain predefined conditions to permit the execution of a specific task (see [61]).

In [60], we present the BusinessActivities framework which presents an approach for the model-driven specification of process-related RBAC models. Several extensions to this framework exist, considering, for example, context constraints [51], process-related duties/obligations [46], secure object flows [26], or the delegation of roles, tasks, and duties [47,53]. The approach presented in this paper further extends the BusinessActivities framework with support for process-related break-glass policies.

2.2. A motivating example

For visualizing process-related RBAC models including mutual-exclusion, binding, and context constraints we use the BusinessActivities UML extension presented in (see [51,60]). Fig. 2 shows a simplified medical examination process (modeled as a BusinessActivity) which will serve as a running example in this paper. The process from Fig. 2 starts when a patient arrives at the hospital. Subsequently, the "Medical examination" task (t_1) is conducted to reach a medical diagnosis. Next, the "Determine treatment options" task (t_2) is executed to devise an appropriate treatment plan. This treatment plan has to be confirmed by a second physician (t_3). In case the treatment plan includes errors or is incomplete, it must be revised before it is resubmitted for confirmation. Finally, the "Medical treatment" task (t_4) is performed.

In the example, we define a subject-binding between the tasks t_1 and t_2 to ensure that the same physician who performed the examination in the "Medical examination" task also evaluates appropriate medical treatment options. This subject-binding is indicated via *SBind* entries in the corresponding task symbols (see Fig. 2). Furthermore, we define a dynamic mutual exclusion (*DME*) constraint on the tasks t_2 and t_3 to enforce the four-eyes-principle on medical treatment decisions. DME tasks can be assigned to the same role but must *not* be allocated to the same individual in the same process instance (see, e.g., [9,59,63]). Thus, for each medical examination the "Determine treatment options" and the "Confirm treatment" tasks must always be conducted by two different individuals. This is an essential quality and safety measure in hospitals to guard against mistakes and malpractice. Moreover, a context constraint (*CC*) is defined on t_3 which specifies

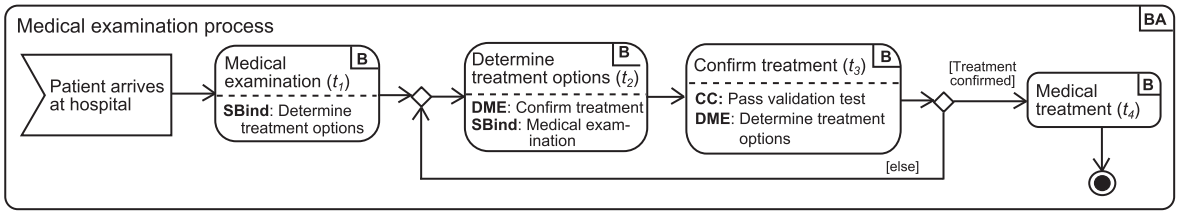

Fig. 2. Simplified medical examination process.

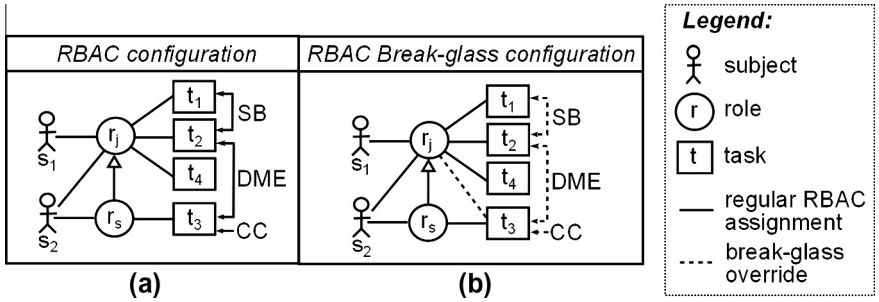

Fig. 3. Example RBAC and break-glass assignments.

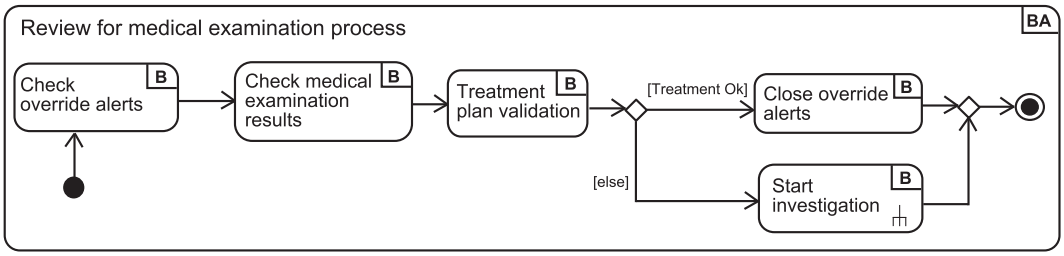

Fig. 4. Example review process.

several conditions that must be met in order to successfully confirm a treatment plan.

Fig. 3a shows the roles and subjects assigned to the tasks of the medical examination process from Fig. 2. Members of the junior physician role r_j are permitted to perform the tasks t_1 ("Medical examination"), t_2 ("Determine treatment options"), and t_4 ("Medical treatment"). Task t_3 ("Confirm treatment") can only be performed by subjects assigned to the senior physician role r_s. This RBAC configuration also is consistent with all entailment constraints defined on the tasks in the medical examination process.

Let us consider three potential *emergency scenarios* for the medical examination process. Without introducing certain exception handling mechanisms, they would lead to a critical delay during process execution.

(1) In a case of medical emergency, no senior-physician is available. However, because only members of r_s are allowed to perform t_3, the start of the medical treatment task (t_4) is delayed.
(2) In an emergency situation, only a senior-physician is available. However, the two DME tasks t_2 and t_3 must be executed by different subjects. Again, the start of t_3 and of all subsequent tasks is delayed until a (second) authorized subject is available to perform t_3.
(3) The context constraint defined on task t_3 cannot be fulfilled in an emergency case. Yet, if the physician assesses the treatment plan as appropriate for this particular emergency case, he needs to be able to override this context constraint in order to proceed with the next task.

Break-glass policies can be used to resolve the above emergency scenarios:

(1) To be able to start the medical treatment (t_4), we can define a break-glass policy for members of r_j which authorizes junior-physicians to perform t_3 in a case of medical emergency (see Fig. 3b).
(2) A break-glass policy authorizes physicians to override DME constraints in the event of an emergency.
(3) A break-glass policy authorizes physicians to override context constraints in emergency situations.

Note that all override actions need to be recorded for later reviews. A *review process* is triggered each time after a break-glass override is registered. An example review process for the medical examination process is shown in Fig. 4. In particular, a physician (who was not involved in the medical examination process) is appointed to perform the following tasks: After checking the override alerts for a particular process, the physician checks the medical examination results and validates the medical treatment plan. If the treatment plan is successfully validated, the override alerts are closed. Otherwise, an investigation process is started (see Fig. 4).

3. 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 CIM layer metamodel for process-related RBAC models [60]. In particular, we specify that certain *breakable tasks* can be performed by subjects who are usually not allowed to execute these tasks. Note that specifying explicit DENY policies in our model is not possible. Thus, tasks have to be defined as being *breakable* in order to be able to apply a break-glass override rule. For this purpose, override rules regulate that members of a certain role are permitted to perform a certain task in an exceptional case (*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 an exceptional case (*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. The most prominent example of an exceptional case that requires the use of a break-glass privilege is probably an emergency that demands an immediate action in order to prevent harm to people or to an organization. However, in our approach the use of a break-glass privilege is not restricted by some external state (such as an

"emergency"). Every subject who owns a break-glass privilege is free to use this privilege whenever it seems suitable. To prevent misuse of break-glass privileges, each break-glass execution will be recorded and subsequently be monitored via a corresponding review process.

We also implemented the extended metamodel presented in Section 3 as well as the corresponding consistency checks and algorithms provided in Appendix A as a break-glass extension to the BusinessActivity library and runtime engine (available for download at [1]).

The subsequent definitions provide a generic framework for integrating break-glass policies into a business process context. For the purposes of this paper, Definition 1 repeats some of the definitions for process-related RBAC models (for details see [51,60]). New definitions for *process-related break-glass RBAC models* are introduced in Definitions 2–5.

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, $Q = rsa \cup tra \cup ptd \cup pi \cup ti \cup es \cup er \cup ar$ 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 meta-model ($\mathcal{P}$ refers to the power set):

1. The mapping $rh : R \mapsto \mathcal{P}(R)$ is called **role hierarchy**. For $rh(r_s) = R_j$, we call r_s senior role and R_j the set of direct junior roles. The transitive closure rh^* defines the inheritance in the role-hierarchy such that $rh^*(r_s) = R_{j^*}$ includes all direct and transitive junior-roles that the senior-role r_s inherits from. The role-hierarchy is cycle-free, i.e. for each $r \in R : rh^*(r) \cap \{r\} = \emptyset$.
2. The mapping $rsa : S \mapsto \mathcal{P}(R)$ is called **role-to-subject assignment**. For $rsa(s) = R_s$, we call $s \in S$ subject and $R_s \subseteq R$ the set of roles assigned to this subject (the set of roles owned by s).
3. The mapping $rown : S \mapsto \mathcal{P}(R)$ is called **role ownership** and returns all direct and inherited roles for a subject. The mapping $rown^{-1} : R \mapsto \mathcal{P}(S)$ determines all subjects owning a particular role, directly or transitively via the role-hierarchy.
4. 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.
5. The mapping $town : R \mapsto \mathcal{P}(T_T)$ is called **task ownership** and returns all tasks for a role. The mapping $town^{-1} : T_T \mapsto \mathcal{P}(R)$ determines the set of roles a particular task is assigned to, directly or transitively via the role-hierarchy.
6. 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.
7. The mapping $pi : P_T \mapsto \mathcal{P}(P_I)$ is called **process instantiation**. For $pi(p_T) = P_i$, we call $p_T \in P_T$ process type and $P_i \subseteq P_I$ the set of process instances instantiated from process type p_T.
8. 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.
9. 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.
10. 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.
11. The mapping $ar : S \mapsto R$ is called **active role mapping**. For $ar(s) = r$, we call s the subject and r the active-role of s.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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 subsequent definitions provide an extension to the meta-model for process-related RBAC models defined in [60]. 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 as specified in Definition 1. 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 task-to-role assignments (see Definition 1.4). In addition, breakable tasks that are assigned to roles can be executed in a break-glass scenario. For example, in the medical examination process from Fig. 2, task t_3 can usually only be performed by members of role r_s. A break-glass policy can extend this configuration by defining that in case of emergency, members of role r_j are also authorized to execute t_3 (see Fig. 3b):
   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 demands a mapping to determine all breakable tasks that are assigned to a particular role. Thus, for a particular role r_s this mapping not only returns all breakable tasks which are directly assigned to r_s but also those breakable tasks which are assigned to junior roles of r_s (and thus are also breakable by members of r_s):
   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 1.5.

3. Breakable tasks can also be directly assigned to subjects. For example, instead of defining the break-glass override on t_3 for all members of r_j in the medical examination process (see Figs. 2 and 3) we can define that only s_1 is allowed to execute t_3 in a break-glass scenario. This might be reasonable if only s_1 has all necessary skills to perform t_3:
   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. Thus, if task t_3 from Figs. 2 and 3 is executed by a member of the junior physician role r_j in a particular process, this instance of t_3 is marked as broken:
   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). For example, if subject s_1 executes t_3 using a break-glass override (see Figs. 2 and 3), the corresponding process instance is said to be broken:
   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. For example, for each broken instance of the medical examination process (see Section 2.2), a senior physician has to check the validation results of t_3 as soon as he/she is on duty again (see Fig. 4):
   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 must be clearly separated (see also Appendix A). In particular, this means that a subject cannot accidentally perform a break-glass task (see Algorithm 1 in Appendix A). 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 mutual exclusion and binding constraints.

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., [44,49,59,63,68,72]). Thus, if we want to define that a certain subject 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., [44,49,59,63,68,72]). 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). 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 (see Appendix A), 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. [19,48]). 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 [59]). 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 (see Appendix A) 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 (see Appendix A), 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 [60].

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 [51,60].

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. Definitions 4.2–4.5 supersede the corresponding Definitions from [51,60] in break-glass scenarios. In particular, they define that:

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. This Definition supersedes Def. 3.1 from [60] specifying that the executing subject of SME subject need 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. This Definition supersedes Def. 3.2 from [60] specifying that the executing subject of DME subject need 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. This Definition supersedes Def. 3.3 from [60] specifying that role-bound tasks must have the same executing role:

$if\ \exists p_b \in P_I$ *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. This Definition supersedes Def. 3.4 from [60] specifying that subject-bound tasks must have the same executing subject:

$if\ \exists p_b \in P_I$ *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. This Definition supersedes the corresponding Definitions from [51] specifying that the context constraints associated to a task have to be fulfilled:

$if\ \exists p_b \in P_I$ *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 extends the definition for execution histories from [60]. The execution history $h(p)$ of a process-related break-glass RBAC model 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 $h : P_I \mapsto \mathcal{P}(\{(t_x, t_t, r, s) | t_x \in T_I, t_t \in T_T, r \in R, s \in S\})$, which maps $h(p)$ to a set of execution events $exec(p)$ (for further details, see [60]).

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 $h_b : P_I \mapsto \mathcal{P}(\{(t_b, t_t, r_b, s_b) | t_b \in T_I, t_t \in T_T, broken(t_b) = true, s_b = es(t_b), r_b = er(t_b)\})$ with $h_b \subseteq h$.

4. A UML extension for process-related break-glass policies

Based on the formal definitions presented in Section 3 domain-specific modeling support can be provided via models that are specified independent from a particular implementation platform (see Section 1). These models are typically defined in a particular modeling language, e.g., BPMN or UML (platform independent models (PIM)). Subsequently, PIMs can be mapped to platform specific models (PSM).

The UML offers a comprehensive and well-defined modeling framework and is the de facto standard for modeling and specifying information systems. The UML's main intention is to capture modeling artifacts throughout the whole development lifecycle with the same modeling language (see [35]). A uniform framework for all of these heterogeneous diagram types and the relationships between them is defined via a common metamodel. This metamodel builds upon the OMG Meta Object Facility (MOF [36]) and formally defines the abstract syntax of all UML diagram types. Modeling support for break-glass policies via a standard notation can help to bridge the communication gap between software engineers, security experts, experts of the application domain, and other stakeholders (see, e.g., [30]). Our domain-specific modeling extension for break-glass policies serves as an enabler to document and communicate how certain emergency scenarios can be handled in a business process.

UML2 Activity models offer a process modeling language that allows to model the control and object flows between different actions. The main element of an Activity diagram is an Activity.

Its behavior is defined by a decomposition into different Actions. A UML2 Activity thus models a process while the Actions that are included in the Activity can be used to model tasks (for details on UML2 Activity models, see [35]). However, sometimes UML diagrams cannot provide all relevant aspects of a specification. Therefore, there is a need to define additional constraints about the modeling elements. The Object Constraint Language (OCL) provides a formal language that enables the definition of constraints on UML models [34]. We apply the OCL to define additional break-glass specific constraints for our UML extension. In particular, the OCL invariants defined in Section 4.2 ensure the consistency and correctness of UML models using our new modeling elements.

The UML standard basically provides two options to adapt its metamodel to a specific area of application [35]: (a) defining a UML profile specification using stereotypes, tag definitions, and constraints. A UML profile must not change the UML metamodel but can only extend existing UML meta-classes for special domains. Thus, UML profiles are not a first-class extension mechanism (see [35, page 660]); and (b) extending the UML metamodel, which allows for the definition of new elements with customized semantics.

In this paper, we apply the second option (extending the UML metamodel) because the newly defined modeling elements for break-glass policies require new semantics which are not available in the UML metamodel. Thus, we introduce the *BreakGlassBusinessActivities* extension for the UML metamodel which is designed for modeling process-related break-glass policies (see Section 3). In particular, we extend the *BusinessActivities* package [60], which provides UML modeling support for process-related RBAC models. We also implemented the extended metamodel presented in Section 3 as well as the corresponding constraints provided in Section 4.2 as a break-glass extension to the BusinessActivity library and runtime engine (see Section 6).

4.1. Metamodel overview

A *BusinessActivity* [60] is a specialized UML Activity (see Fig. 5). A *BusinessAction* corresponds to a task and comprises all permissions to perform the task. *Roles* and *Subjects* are linked to BusinessActions. The metaclasses *BusinessAction*, *Role* and *Subject* are defined as subclasses of the UML Classifier metaclass, which yields a number of advantages. For example, it allows modelers to define specialized subtypes of BusinessAction with own specialized behavioral or structural features. Moreover, BusinessActions can be instantiated and, in contrast to ordinary UML Actions, the same instance can be used/executed multiple times in an activity (see also [35]). Among other things, this means that each instance of a BusinessAction can have its own state and history, for example including attributes to capture how often the action has been

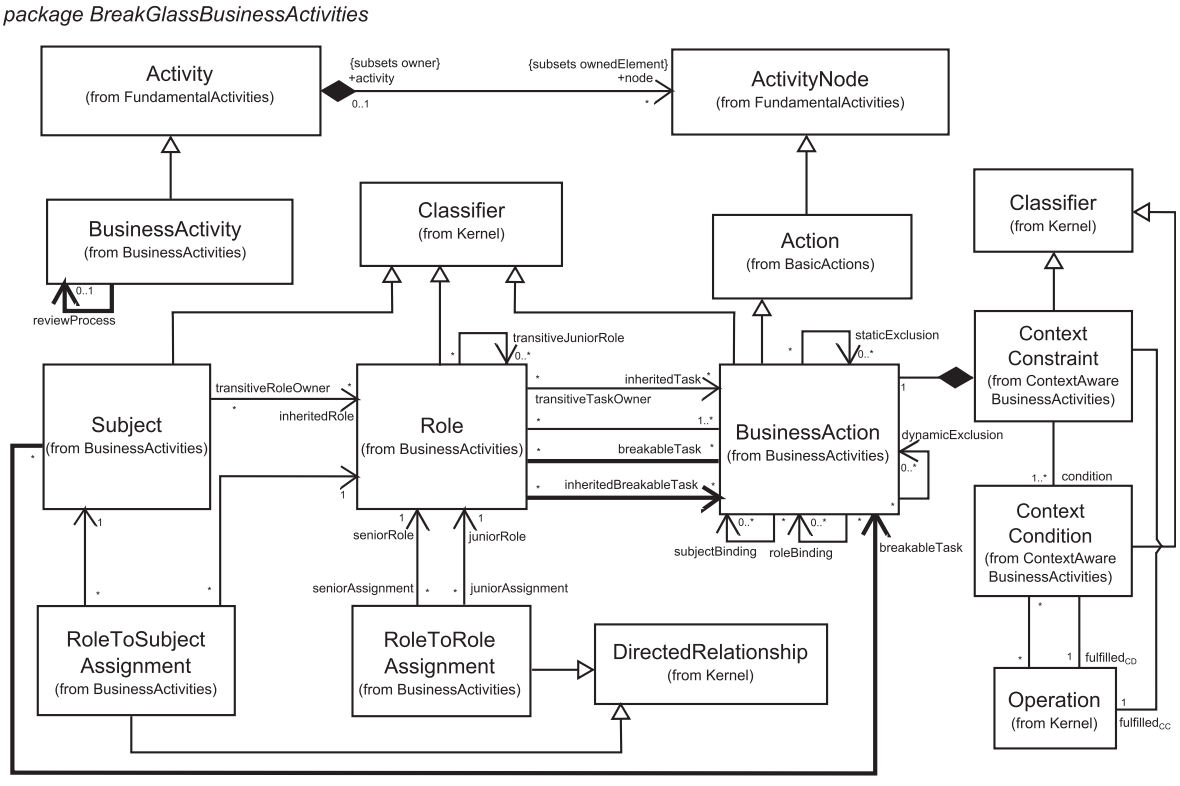

Fig. 5. UML metamodel extension for process-related break-glass RBAC models.

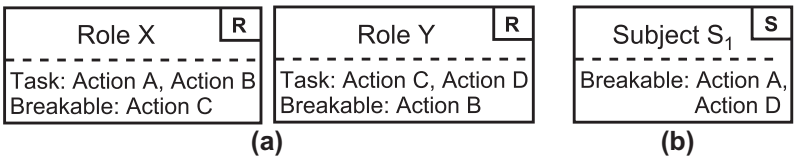

Fig. 6. Visualizing (a) breakable-by-role and (b) breakable-by-subject override relations.

executed, which subjects and roles executed the action, etc. For a detailed discussion on how mutual exclusion, binding, and context constraints are integrated into the BusinessActivities extension, see [60,51].

For integrating break-glass policies into the UML metamodel, we introduce the following new relations: Each Role can include *breakableTasks* and *inheritedBreakableTasks*, which are inherited from its junior-roles (see Definitions 1 and 2 in Section 3). These two relationships can be used to visualize that members of a certain role are authorized to perform the assigned tasks only in case of emergency. Similarly, each Subject can be assigned to *breakableTasks* to show that a particular subject is allowed to perform the assigned tasks in case of emergency (see Definition 3). Fig. 6 illustrates presentation options to visualize the breakable-by-role and breakable-by-subject override relations via "Breakable" entries. Note that these relations are formally defined through our UML metamodel extension and therefore exist independent of their actual graphical representation. Moreover, each BusinessActivity is related to a *reviewProcess* (see below).

Each instance of a BusinessAction and the corresponding BusinessActivity instance are marked as *broken* if the BusinessAction has been executed by a subject via a break-glass override assignment (see Definitions 4 and 5 in Section 3) and Constraints 1 and 2 in Section 4.2). For each broken BusinessActivity, there has to exist a corresponding reviewProcess (see Fig. 5 and Constraint 3). Roles and subjects can own a task either regularly or via a break-glass override assignment (see Constraints 4 and 5). Moreover, in a break-glass scenario, all task-based entailment constraints do not have to be fulfilled (see Constraints 10 and 11 as well as Constraints 6–9).

4.2. OCL constraints

Often a structural UML model cannot capture all types of domain-specific constraints which are relevant for describing a target domain. Thus, additional constraints can be defined, for example, by using a constraint expression language, such as the OCL [34]. In this paper, we use OCL invariants to define the semantics by encoding break-glass specific constraints. For the sake of readability, this Section only shows three example OCL invariants. The complete list of OCL invariants for the Break-Glass Business Activity extension is found in Appendix B.

Constraint 1. Each BusinessAction defines an attribute called "broken" stating if a certain BusinessAction instance is executed via a break-glass override assignment:

```
context BusinessAction inv:
self.instanceSpecification→forAll (i |
  i.slot→exists (b |
    b.definingFeature.name = broken
    and
    b.definingFeature.type.name = Boolean))
```

Constraint 3. For each broken BusinessActivity instance, there has to exist a corresponding reviewProcess:

```
context BusinessActivity inv:
self.instanceSpecification→forAll (i |
  if i.slot→exists (b |
    b.definingFeature.name = broken and
    b.value = true)
    then self.reviewProcess→notEmpty ()
  else true endif
```

Constraint 7. For all broken BusinessAction instances, the executing subjects of DME tasks do not have to be different:

```
context BusinessAction inv:
self.instanceSpecification→forAll (b |
  b.slot→select (s |
    s.definingFeature.name = broken
    if (s.value = true) then
      self.dynamicExclusion→forAll (dme |
        dme.instanceSpecification→forAll (i |
          b.slot→forAll (bs |
            i.slot→forAll (is |
              if
  bs.definingFeature.name = executingSubject
                and
  is.definingFeature.name = executingSubject
              then (bs.value = is.value) or
  not (bs.value = is.value)
              else true endif)))))
    else true endif))
```

Each of the CIM-layer definitions from Section 3 is mapped to our PIM-layer UML extension. In particular, structural properties of the CIM-layer metamodel are defined as additional elements in the UML metamodel. New semantic properties are specified via OCL constraints. Each of the above OCL constraints uses the UML InstanceSpecification element to refer to instances of the newly added BusinessAction and BusinessActivity classes (see also Fig. 5). The UML standard allows for a very flexible use of the InstanceSpecification element (see [35]). In general, an instance specification represents an instance in a modeled system. Each instance specification can be associated with an arbitrary number of classifiers (i.e. direct or indirect sub-classes of the UML Classifier metaclass, such as BusinessAction or BusinessActivity for example). Moreover, each instance specification may include an arbitrary number of slots. A UML Slot specifies that an instance specification has a value for the so called "defining feature" of the respective slot. In this way, the UML InstanceSpecification element allows to define (constraints on) the values of runtime instances. For a detailed specification of the UML InstanceSpecification and Slot metaclasses please see [35].

Table 1 gives an overview of how each of the definitions from Section 3 is mapped to our UML extension for Break-Glass Business Activities (see also Appendix B). For example, Definition 2.4 specifies that we can determine if a task instance is broken. This is mapped to Constraint 1 which specifies that each BusinessAction instance includes a corresponding slot that carries a boolean value to indicate if this particular instance is broken (see above and Appendix B). Definition 4.1 specifies that for each broken process instance a corresponding review process has to exist. This is mapped to Constraint 3 which specifies that for each broken BusinessActivity instance (i.e. instances where the value of the slot including the defining feature "broken" is set to "true") a corresponding review process has to exist (see above and Appendix B).

4.3. Example for UML break-glass models

We suggest to use three complementary perspectives to model process-related break-glass RBAC models. This is because capturing all aspects within the process model will presumably overload it. Fig. 7a shows the *process perspective* of the medical examination process (see Section 2.2).

The *Break-Glass RBAC perspective* is exemplified in Fig. 7b illustrating task-to-role, role-to-subject, and role-to-role assignments. For example, subject s_1 is assigned to the Junior Physician role. Corresponding notation symbols are described in detail in [60].

Table 1
Consistency between generic metamodel and UML extension.

Generic definition	Covered through
Definition 1: Process-Related RBAC Model	Defined via the BusinessActivities extension and the OCL constraints presented in [60]
Definition 2.1: $bbr : R \mapsto \mathcal{P}(T_T)$	Metamodel extension: breakableTasks Association between Role and BusinessAction (see Fig. 5)
Definition 2.2: $btown : R \mapsto \mathcal{P}(T_T)$	Metamodel extension: breakableTasks and inheritedBreakableTasks Associations between Role and BusinessAction (see Fig. 5)
Definition 2.3: $bbs : S \mapsto \mathcal{P}(T_T)$	Metamodel extension: breakableTasks Association between Subject and BusinessAction (see Fig. 5)
Definition 2.4: $broken_{T_I} : T_I \mapsto BOOLEAN$	Constraint 1
Definition 2.5: $broken_{P_I} : P_I \mapsto BOOLEAN$	Constraint 2
Definition 2.6: $review : P_T \mapsto P_T$	Metamodel extension: recursive review Association on BusinessActivity (see Fig. 5); Constraint 3
Definition 3.1: $\forall t_T \in T_T : town^{-1}(t_T) \cap btown^{-1}(t_T) = \emptyset$	Constraint 4
Definition 3.2: $\forall t_T \in T_T, r_1, r_2 \in R$ with $t_T \in btown(r_1)$ and $t_T \in town(r_2) : rown^{-1}(r_1) \cap rown^{-1}(r_2) = \emptyset$	Constraint 5
Definition 3.3: $\forall t_T \in T_T, r \in R$ with $t_T \in town(r) : rown^{-1}(r) \cap bbs^{-1}(t_T) = \emptyset$	Constraint 5
Definition 4.1: $\forall p_b \in pi(p_T)$ with $broken(p_b) = true : \exists p_r \in review(p_T)$	Constraint 3
Definition 4.2: $\forall t_x \in ti(t_1, p_b), \forall t_y \in ti(t_2, p_b)$ with $t_2 \in sme(t_1)$ and $broken(t_x) = true : (es(t_x) \neq es(t_y) \vee es(t_x) = es(t_y))$	Constraint 6
Definition 4.3: $\forall t_x \in ti(t_1, p_b), \forall t_y \in ti(t_2, p_b)$ with $t_2 \in dme(t_1)$ and $broken(t_x) = true : (es(t_x) \neq es(t_y) \vee es(t_x) = es(t_y))$	Constraint 7
Definition 4.4: $\forall t_x \in ti(t_1, p_b), \forall t_y \in ti(t_2, p_b)$ with $t_2 \in rb(t_1)$ and $broken(t_x) = true : (es(t_x) \neq es(t_y) \vee es(t_x) = es(t_y))$	Constraint 8
Definition 4.5: $\forall t_x \in ti(t_1, p_b), \forall t_y \in ti(t_2, p_b)$ with $t_2 \in sb(t_1)$ and $broken(t_x) = true : (es(t_x) \neq es(t_y) \vee es(t_x) = es(t_y))$	Constraint 9
Definition 4.6: $\forall t_x \in ti(t_T, p_b)$ with $broken(t_x) = true$ `if` $cc_x \in linked_{CC}(t_T)$ `then` $(fulfilled_{CC}(cc_x) = true \vee fulfilled_{CC}(cc_x) = false)$	Constraints 10 and 11
Definition 5: $h_b : P_I \mapsto \mathcal{P}(\{(t_b, t_t, r_b, s_b) \mid t_b \in T_I, t_t \in T_T, broken(t_b) = true, s_b = es(t_b), r_b = er(t_b)\})$	Implicitly defined via our metamodel extension and the specification of UML Activity models (see Fig. 5 and [35])

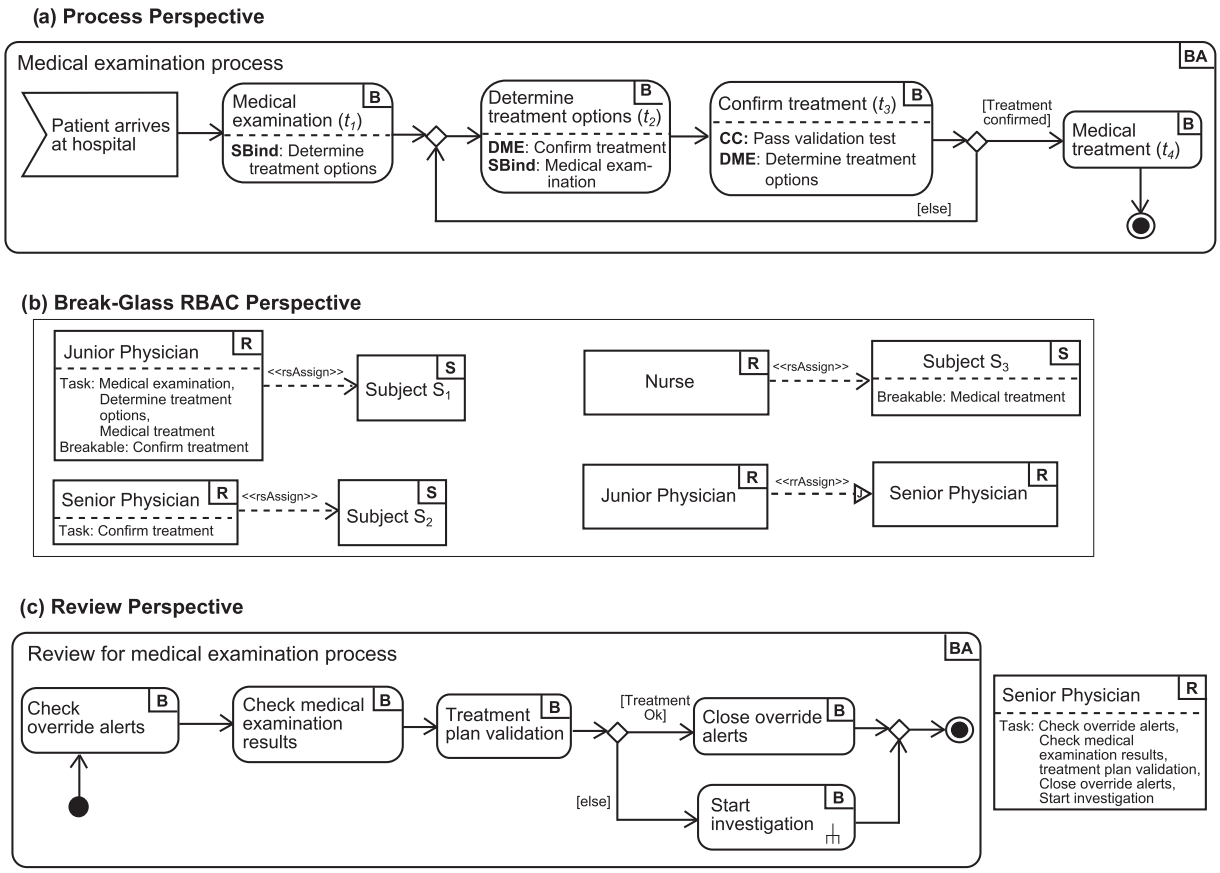

Fig. 7. Example for process-related break-glass RBAC models.

Moreover, this perspective provides a detailed view on which role or subject is allowed to perform a break-glass override. For example, we define a breakable-by-role override relation between the Junior Physician role and the "Confirm treatment" task in Fig. 7b. Thus, in a break-glass scenario, members of the junior-physician role are able to perform the "Confirm treatment" task. Moreover, a breakable-by-subject override is defined on subject s_3 and the "Medical treatment" task, because nurse s_3 has all necessary skills to perform the medical treatment in an emergency case.

Finally, the *review perspective* illustrates the review process which is triggered each time after a break-glass override is executed (Constraint 3). An example review process for the medical examination process is shown in Fig. 7c. In particular, a physician (who was not involved in the medical examination process) is appointed to perform the following tasks: After checking the override alerts for a particular process, the physician checks the medical examination results and validates the medical treatment plan. If the treatment plan is successfully validated, the override alerts are closed. Otherwise, an investigation process is started.

5. Case study on modeling process-related break-glass RBAC models

To evaluate the approach presented in this paper with regard to its practical applicability, we conducted a case study applying our UML extension on real-world processes. The case study presented in this Section is based on a collection of real-world process models we retrieved from a large Austrian school center. The selection consists of about 30 organizational processes, which were collected by members of the school center during a process management

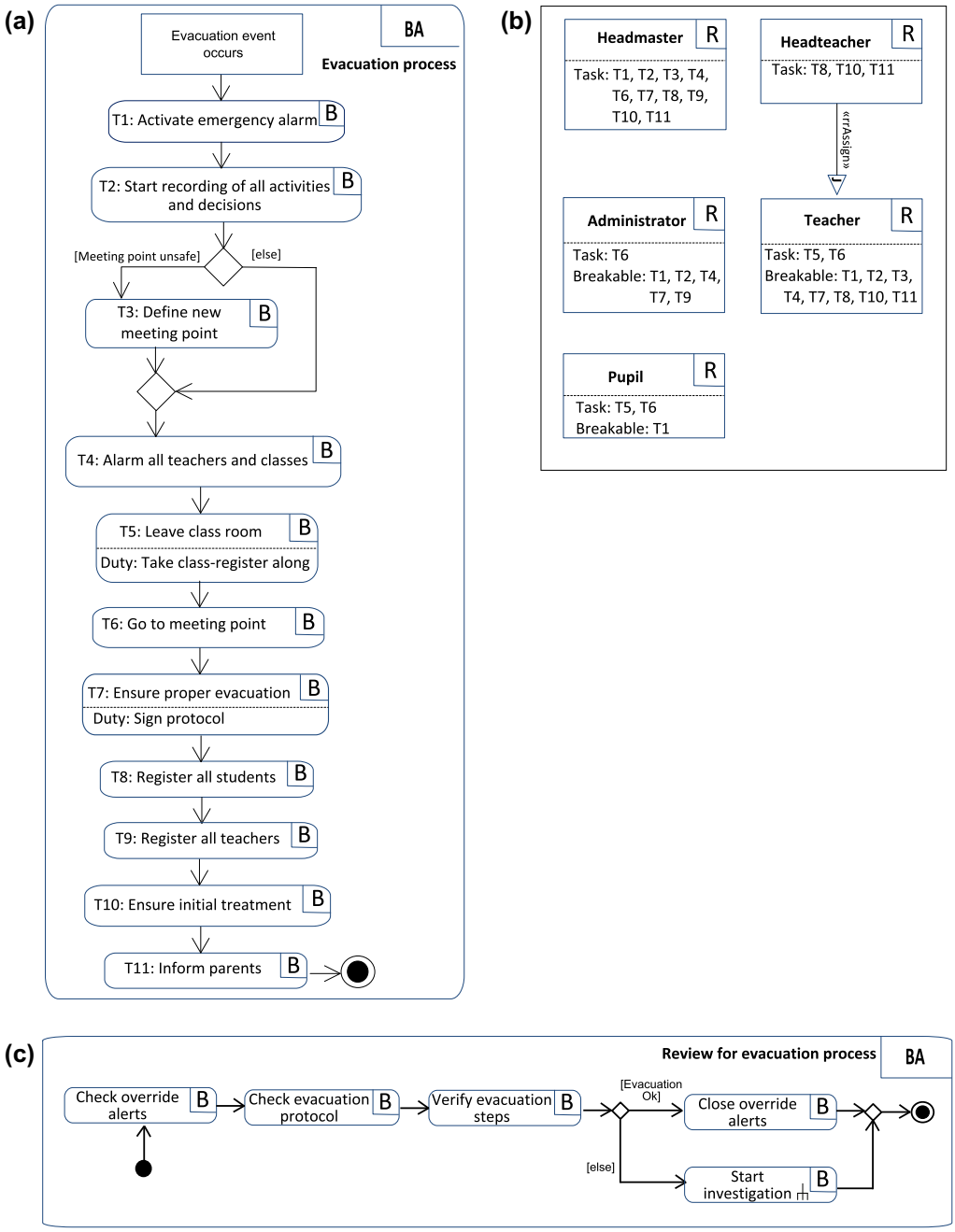

Fig. 8. Evacuation process in an Austrian school.

initiative. The control flow of some processes was graphically visualized depicting the sequence of tasks as well as of corresponding authorized and responsible individuals. However, these processes were visualized using an ad hoc (non-standard) graphical notation. Furthermore, most of the processes were described in a detailed textual/tabular listing of activities with varying level of granularity. The process descriptions included references to legal requirements (e.g., certain paragraphs in the Austrian law concerning teaching in schools) and other internal or external regulations.

In our case study, we identified processes including information on emergency scenarios and remodeled them via our UML extension for process-related break-glass policies (see Section 4). In Fig. 8, an example process from our case study is presented which illustrates the school's evacuation process. The BusinessActivity in Fig. 8a consists of eleven BusinessActions carried out in case an evacuation event occurs. Fig. 8b shows the roles associated with tasks in the evacuation process. For each role, the regular task assignments as well as the break-glass task assignments were derived from the process descriptions. Most of the tasks from the school's evacuation process are usually performed by the headmaster of the school. However, in case this process is executed in the event of an actual emergency (i.e., not for practice purposes), teachers and pupils are also authorized to perform these tasks (see Fig. 8b). For example, the headmaster is authorized to perform task T1 (Activate emergency alarm). In case of emergency, the administrator as well as all teachers and pupils are also authorized to execute T1. Furthermore, Fig. 8c shows the review process for the evacuation process. In this review process, all break-glass alerts recorded in the evacuation protocol are checked by an external auditor. If the evacuation protocol is successfully verified, the override alerts are closed. Otherwise, an investigation process is triggered.

After remodeling the processes via our UML extension, we evaluated the remodeled process diagrams of the case study by performing semi-structured interviews with three members of the school, including the head master, one teacher, and one member of the administrative staff. This approach was chosen because interviews are one of the most important methods in case study research [43]. Moreover, for qualitative case studies it is recommended to choose subjects from different parts of the organization to involve different roles in the interviews [18].

The interview was carefully designed using the guidelines from [27]. It consisted of five main open-ended questions. Each interview varied between 20 and 25 min in length. The answers were recorded by using field notes which were then subsequently analyzed by the interviewer. Table 2 details the main questions asked in the interviews.

Table 2
Questions from semi-structured interviews.

Q1	Do the process models provide added value for the school? If yes, in how far can the school/members of the school benefit from the extended process models?
Q2	How will the extended process models potentially be used in the school?
Q3	What do you think about our approach of integrating process models and related security aspects? Advantages/Disadvantages?
Q4	Do you have difficulties in understanding different parts of the processes? If yes, which parts are easy to understand and which parts are difficult or not comprehensible?
Q5	Do you have any suggestions on how the graphical representation of the processes can be improved?

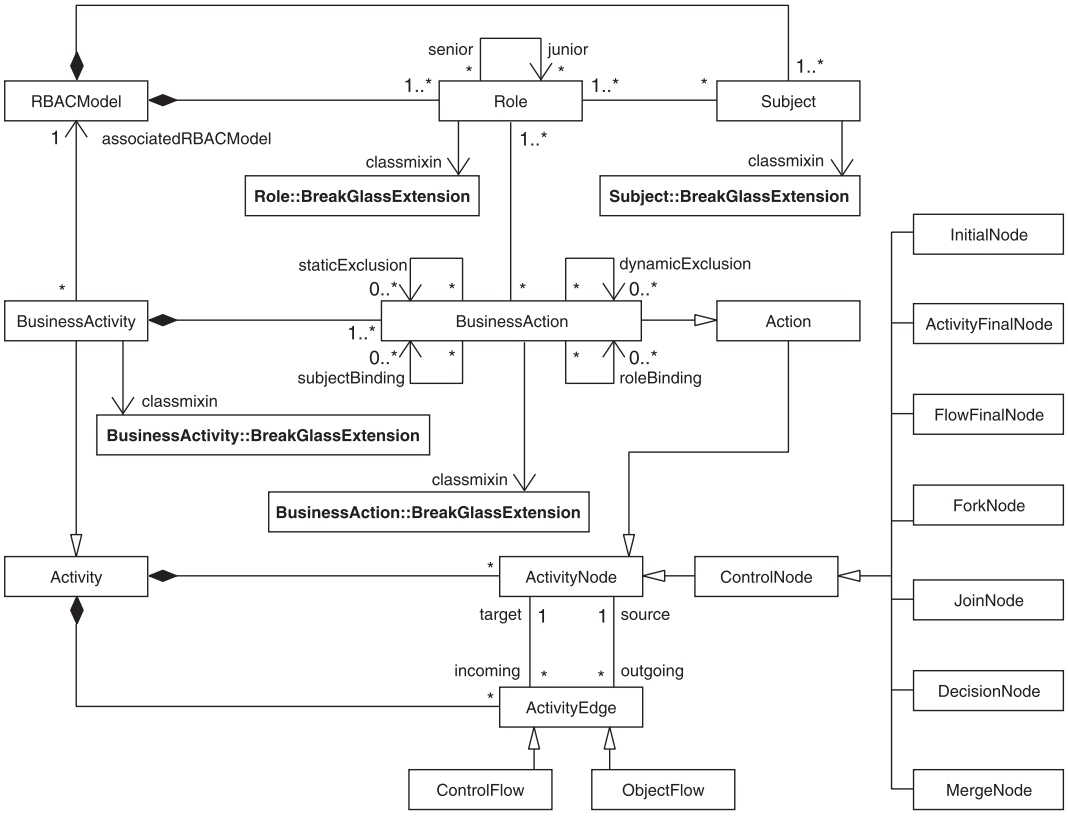

Fig. 9. Class model of the extended Business Activity library and runtime engine (excerpt).

In the interviews, two advantages of the visually modeled processes were communicated: First, the headmaster emphasized that new employees who are not familiar with school procedures would now have a comprehensive and easy-to-understand, diagram-based documentation of key processes and related break-glass concerns at hand. This would have the potential of facilitating work tasks and communication with other school members during the first weeks after joining the school. This opinion may also support the frequently cited conjecture that models employing a process flow metaphor are suitable communication instruments for non-technical domain experts (see, e.g., [21]). In addition, before the case study was performed, only a few processes were depicted using an ad hoc (i.e., non-standard) visual notation. Most processes were described via textual documents in varying degrees of detail. The state of the organization's process descriptions was therefore inconsistent and inhomogeneous. Moreover, the interview partners noted that the security-aware process models would improve the general awareness among the school members of how closely security requirements are related to key organizational processes. All three members of the school stated that the process models are easy to comprehend in their essence (e.g., task and role labels, basic sequencing of tasks, relations between duties and tasks).

6. Platform support

In order to enforce break-glass policies in actual software systems (i.e., at the PSM layer), we implemented an extension to the BusinessActivity library and runtime engine (see [60]). Our implementation provides platform support for all modeling-level elements and automatically enforces all invariants defined via OCL constraints. Moreover, it implements the algorithms for runtime consistency of break-glass RBAC models (see Sections 3 and 4). The source code of our implementation is available for download at [1].

Fig. 9 shows the essential class relations of our library and runtime engine. Classes that were added to implement the break-glass extension are highlighted via bold font. From a software technical point of view we used *mixin classes* as a primary extension mechanism. Simplified, a mixin is a class that can be dynamically

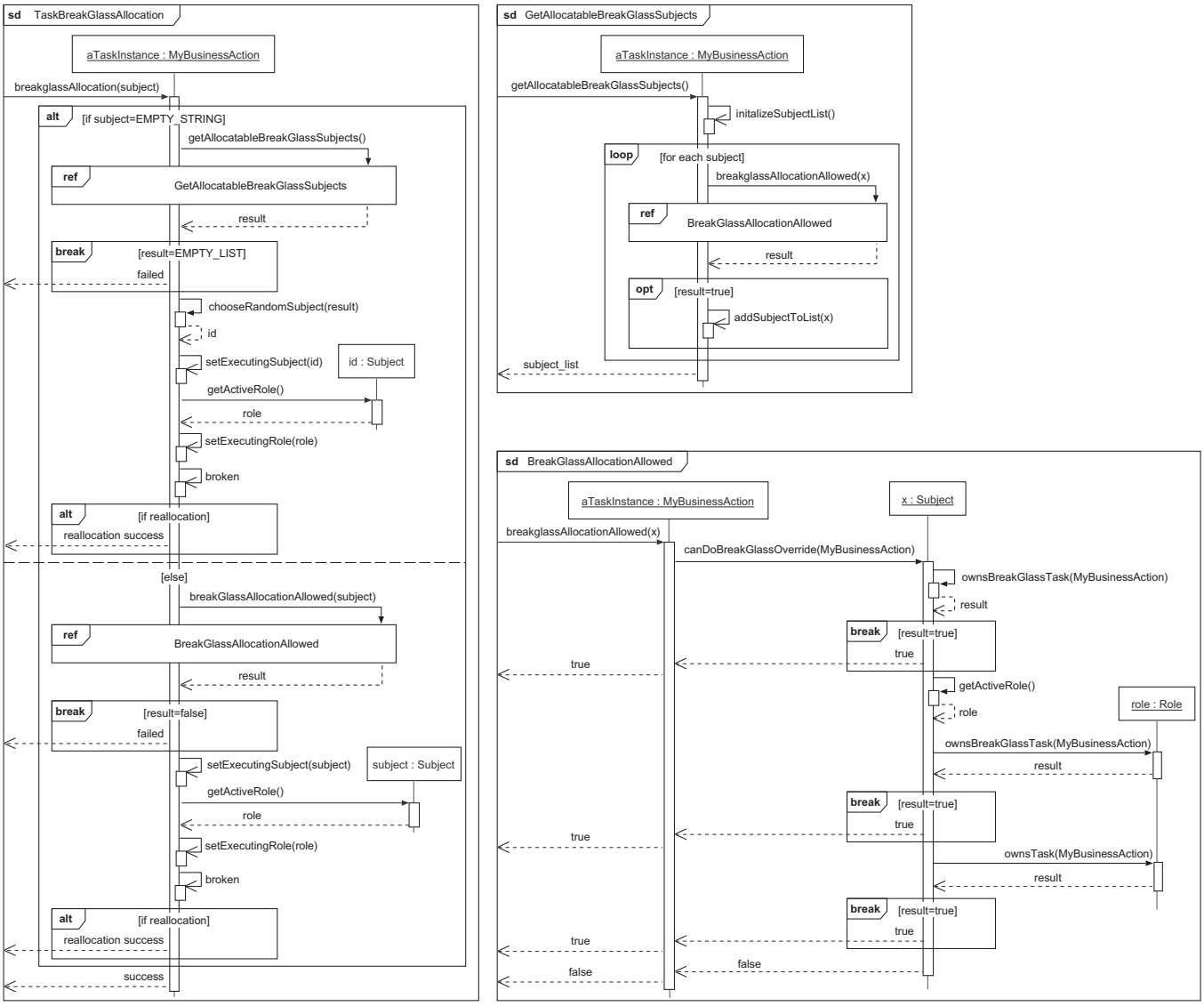

Fig. 10. Break-glass task allocation.

registered as an extension to another class. Mixins are a flexible extension mechanism that can be applied if an extension of the class-hierarchy via (multiple) inheritance is not desirable (see, e.g., [10,11,70,73]). Mixins can be added or removed (activated or deactivated) at runtime and thereby provide a means to individually tailor the behavior of the extended entity. In general, we distinguish class mixins (also: per-class mixins or type mixins) and instance mixins (also: per-object mixins) [73]. *Class mixins* are classes that are applied as mixins for a class. They are types for all direct and indirect instances of this class. *Instance mixins* are classes that are applied as mixins for an individual object, i.e., for an instance of a class. They extend the types of this particular object with (one or more) instance mixins.

The four extension classes shown in Fig. 9 are defined as class mixins. The `Role::BreakGlassExtension` class extends the `Role` class and adds the capabilities to define breakable-by-role overrides. The `Subject::BreakGlassExtension` extends the `Subject` class and adds the capabilities to define breakable-by-subject overrides (see Section 3). The `BusinessAction::BreakGlassExtension` and `BusinessActivity::BreakGlassExtension` classes extend the `BusinessAction` and `BusinessActivity` classes respectively. They provide functions for break-glass task allocation and add runtime consistency checks that conform to the algorithms defined in Appendix A. In addition to the extension classes shown in Fig. 9, our implementation includes three extension classes that are defined as instance mixins. In contrast to the class mixins discussed above, these instance mixins are dynamically added to corresponding task objects or process objects (i.e., instances of the `BusinessAction` and `BusinessActivity` classes) in case a break-glass override is actually used and a particular task instance is "broken". Thus, these instance mixins enforce the changed behavior of broken task and process instances as defined in Sections 3 and 4.

Fig. 10 shows UML interaction models that describe our implementation of the break-glass task allocation procedure in detail. When a `breakglassAllocation` call is invoked on a particular task instance (i.e., an instance of the `BusinessAction` class, see Fig. 9) the respective task instance first checks if the subject parameter is empty or if it includes a subject name. If the subject parameter is empty (i.e., it includes the empty string), it calls the `getAllocatableBreakGlassSubjects` method to fetch the list of all subjects who can potentially perform a breakable-by-subject override (see Section 3) for this task. In order to compile this list, the task instance calls the `breakglassAllocationAllowed` method for each subject. Subsequently, the respective subject checks if it can (potentially) perform a break-glass-override for the corresponding task and, depending on the result of these checks, returns either "true" or "false". All subjects that are capable of performing a break-glass-override are added to the respective list. After all subjects are visited, the `getAllocatableBreakGlassSubjects` method returns the subject list back to the task instance. If this list is empty (i.e., no subject can perform a break-glass override for this task instance) the `breakglassAllocation` method returns a "failed" result. However, if the subject list is not empty, the task instance randomly chooses a subject from the list, performs a break-glass allocation and sets the "broken" flag to announce that this task instance is now broken and to trigger a corresponding review process (see Section 3).

If the subject parameter of the `breakglassAllocation` method was not empty, we want to perform a breakable-by-subject override with a particular subject (i.e., not some randomly chosen subject). In this case, we also need to call the `breakGlassAllocationAllowed` method to check if this very subject is allowed to perform a breakable-by-subject override. If the subject is not allowed to perform this override, the `breakglassAllocation` returns a "failed" result. Otherwise, the task instance performs a break-glass allocation and sets the "broken" flag (see Fig. 10).

7. Discussion and related work

In recent years, there has been much work on various aspects of break-glass policy concepts. Break-glass features are also implemented in major business software, e. g., in the SAP Virsa Firefighter [3] or in Oracles Role Manager [2]. In this Section, we first compare our approach introduced in this paper with related work and then exemplarily show how our approach can be integrated with other approaches.

7.1. Related work

In general, we distinguish two types of related work for this paper. First, we have approaches that primarily aim to integrate break-glass policies into (role-based) access control models. Second, a number of different approaches exist that integrate break-glass related information into a business process/workflow environment. Many of the access control- and business process-related approaches are complementary to our work and are well-suited to be combined with our process-related break-glass RBAC models.

Table 3 shows an overview of related work on modeling break-glass policies in an access control or business process context. With respect to the concepts and artifacts specified in Sections 3, 4, and 6, we use a $\surd$ if a related approach provides similar and/or comparable support for a certain concept, and a Δ if a related approach provides at least partial support for a particular aspect.

Several approaches exist to integrate break-glass policies into access control models. For example, the optimistic security principle [37] aims to handle exceptional cases. In the approach from [37], any access is legitimate and is thus granted. Monitoring and recording functions are provided to guarantee traceability. These functions are implemented using the Clark–Wilson model rules [17]. A similar approach is presented by Ardagna et al. [5]. They introduce a break-glass approach where an action can be performed by finding a corresponding emergency policy. Alternatively, a break-glass override can be granted if the system is in an emergency state and a supervisor can be notified about the override. In comparison to our work, the enforcement of security policies in these approaches is retrospective. They rely on administrators to detect unreasonable accesses and subsequently take steps to compensate for undesired behavior. This approach, however, causes an immense burden on administrators. Moreover, these approaches do not consider entailment constraints and do not provide corresponding tool support.

The break-the-glass RBAC (BTG-RBAC) model [23] specifies for each permission-to-role assignment if a break-glass override is allowed. Moreover, obligations can be defined for permissions to define arbitrary actions that must be performed in case of a break-glass override. Again, this break-glass model does not consider entailment constraints. Corresponding tool support is also not provided. In [12], a break-glass extension for SecureUML is provided. The resulting SecureUML break-glass policies can then be transformed into XACML. In contrast to our work, this approach does not consider break-glass decisions in connection with dynamic mutual exclusion constraints or binding constraints. Furthermore, the model does not allow for the definition of subject-specific override rules, if only certain members of a role have all necessary competencies to perform the breakable task. Another approach for discretionary overriding of access control in XACML policies is introduced in [4] by using XACML obligations. Hereby, a break-glass policy is specified as an override-obligation, which

logs the activity, prompts the user for confirmation, and notifies the authority. This approach does not offer role-based break-glass policies and does not consider entailment constraints. In [41], a certificate-based approach based on the Privilege Calculus Framework is used to implement a break-glass mechanism. The Secure information sharing break-glass model introduced in [14] uses the Core Event Specification Language (CESL) for visualizing logical definitions and sequences. This language provides stream, event and pattern operators to express queries. In comparison to other approaches, emergency policies are only valid temporarily and cannot be triggered by a user but only by the system. Moreover, contextual information is taken into account in access control decisions.

Note that most of the break-glass approaches presented above are rather generic and can thus be adapted for a PAIS context with reasonable customization effort. Only few contributions exist which explicitly integrate the concept of break-glass policies into a business process context though. In [31], a survey on flexibility criteria for business process management systems is presented. Amongst others, clearly defined responsibilities for tasks via roles and sophisticated exception handling mechanisms are identified as important flexibility requirements for process-aware information systems. In [67] Wainer et al. present an RBAC model for workflow systems, called W-RBAC. 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. Compared to our approach, subject-specific break-glass policies are not supported in the W-RBAC model. Moreover, corresponding modeling support for the visualization of business processes and corresponding break-glass policies is not provided. von Stackelberg et al. [66] present a break-glass approach for business processes and, in contrast to other work, explicitly consider contextual information in their break-glass assignments (such as start and end time of task execution). They specify an annotation language which allows to define actors involved and context-specific constraints for process-related break-glass scenarios. However, graphical symbols for break-glass concepts as corresponding tool support is not provided.

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, [39] 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 [69], 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 [40]. In particular, several correctness criteria and their application to specific process meta models are discussed. Thus, this approach handles exceptional process executions by dynamically adapting the process flow. In comparison to our work, all the approaches that integrate break-glass policies into business processes have in common that processes must be changed in order to handle exceptional situations. The main goal of our approach, however, is to maintain the designed process flow, while ensuring that only authorized subjects are allowed to participate in a workflow. Moreover, none of these approaches considers the implications of task-based entailment constraints in exceptional situations. They also do not offer modeling support for business processes and related break-glass policies. Corresponding tool support is only provided in [39].

7.2. Integration with other approaches

The metamodel presented in Section 3 provides a generic approach for integrating break-glass policies into process-related RBAC models. It 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. Based on these definitions, e.g., a process engine or a process modeling language can be extended with break-glass functionalities. For example, as mentioned above, we implemented a break-glass extension to the BusinessActivity library and runtime engine and defined a break-glass extension to the UML (see Sections 4 and 6). However, as the metamodel for process-related break-glass RBAC models provides a generic framework, it does not include domain-specific or application-specific definitions. Thus, when applying the metamodel to a certain domain (e.g., for a hospital information system), it may be necessary to tailor it accordingly.

In principle, any break-glass model compliant to our metamodel can be used to define tailored domain-specific break-glass policies. For the purposes of this paper, we exemplarily show how *Rumpole* [28] can be used to define domain-specific break-glass

Table 3
Comparison of related work.

	Business processes	Subject-specific BG policies	Role-based BG policies	Review mechanism	Entailment constraints	Formal metamodel	Modeling support	Consistency checks	Tool support
Approaches to integrate break-glass policies into (role-based) access control									
Optimistic security [37]		Δ		Δ		Δ			
Policy spaces [5]		Δ		Δ		Δ			
BTG-RBAC [23]			√	Δ		√			
Break-Glass SecureUML [12]			√	Δ	Δ	Δ	Δ		Δ
Access control in XACML [4]		√		Δ				Δ	Δ
Extended privilege calculus [41]		√		Δ	Δ	Δ			
Secure information sharing [14]		Δ	Δ	Δ		Δ		Δ	Δ
Approaches to integrate break-glass policies into business processes									
W-RBAC [67]	√		Δ	Δ	Δ	√			Δ
Context-aware BTG [66]	√	√	Δ	Δ	Δ	Δ	Δ		
ADEPT$_{flex}$ [39]	√					Δ		Δ	Δ
Change support in PAIS [69]	√		√	Δ		Δ			
Flexibility in PAIS [40]	√			Δ		Δ		Δ	
Process-related Break-Glass RBAC models (our approach)	√	√	√	√	√	√	√	√	√

policies that are based on our metamodel. Compared to other break-glass models, Rumpole provides a more detailed feedback on why an access was denied by using a logic programming language defined over Belnap's four-valued logic [6]. Most other break-glass models do not consider the reasons for issuing a denial, but have a fixed procedure to determine if an override is permitted. Using Rumpole, a policy writer is able to constrain a break-glass override permission in a fine-grained way. Below, we give an overview how Rumpole can be used to tailor our generic break-glass model (see Section 3) to express domain-specific break-glass policies.

In particular, Rumpole distinguishes between a subject's competences and empowerments to gain more insight into the causes for an access denial. The notion of *competence* captures if a subject has all necessary permissions to perform a certain action. The notion of *empowerment* captures if all integrity constraints, e.g., context constraints or entailment constraints, are fulfilled. The concepts of competence and empowerment are then used in break-glass override rules to determine whether a subject is permitted or denied to override an access denial. In [28], rules defining competences and empowerments are formulated using the following predicates:

[competent/empowered](Sub, Tar, Act) = true: Subject *Sub* is competent/empowered to perform a certain action *Act* on target *Tar*.

For the purposes of this paper, we slightly modify this predicate by embedding it into the context of process-related RBAC models (see Section 3). Rules defining competences and empowerments in business processes are thus formulated via the following predicates:

[competent/empowered](Sub, Process, Task) = true: Subject *Sub* is competent/empowered to perform a certain *Task* within a particular *Process*.

We can specify *break-glass override rules* defining if a subject is permitted or denied to override an access control denial via the following predicates (slightly modified from [28]):

[permit/deny](Sub, Process, Task) = true: Subject *Sub* is permitted/denied to perform a certain *Task* within a particular *Process* in case of emergency.

Moreover, we can specify *obligations* which must be fulfilled in order to allow the break-glass override. For example, the *agreedObl*-predicate is used to denote whether the subject has agreed to perform the requested obligatory action. Other predicates regarding obligations are described in [28].

via this mechanism, we define the following tailored break-glass policies to handle the emergency scenarios outlined for our example process in Section 2.2.

(1) Physicians that are directly permitted to perform the "Confirm treatment" task (t_3) in case of emergency (via the *bbs* mapping) can only execute the task if they are authorized to perform a medical examination (via the regular *rown* and *town* mappings) (see Fig. 3b). However, an override can take place only during night shifts (see [28]):

permit ($s \in bbs^{-1}$(*confirm treatment*), *medical examination process, confirm treatment*) $\Leftarrow$
competent ($s \in rown^{-1}(r) : r \in town$ (medical examination), medical examination process, medical examination)
if *currentTime (T)* $\wedge$ *nightShift (T)*

(2) In a second example, we override entailment and context constraints with tailored Rumpole policies. For example, we can define that in case of emergency the DME constraint defined on the "Determine treatment options" and "Confirm treatment" tasks is ignored only if a certain reason is provided:

permit ($s \in rown^{-1}(r) : r \in town$(*confirm treatment*), *medical examination process,*
dme (determine treatment options)) $\Leftarrow$
competent (*s*, *medical examination process, dme (determine treatment options)*)
if *agreedObl (Sub, log, giveReason)*

Moreover, Rumpole allows to constrain access control overrides by imposing non-contextual condition, such as how much incomplete knowledge is allowed or limit of overrides per subject or process (see [28]).

8. Conclusion

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. In this paper, we presented a break-glass extension for process-related RBAC models. Our approach is based on a generic CIM layer metamodel. It is generic in the sense that it can be used to extend process-aware information systems or process modeling languages with support for break-glass policies. Moreover, we defined a set of generic algorithms to ensure the runtime consistency of our extended process models. Because our CIM layer model for process-related break-glass RBAC provides a generic (i.e., platform and domain independent) framework, it does not include domain-specific or application-specific definitions (e.g. for hospital information systems, or for bank information systems). However, we exemplarily discussed how our approach can be combined with other approaches to define tailored, domain-specific break-glass policies.

At the PIM layer, we provide UML modeling support for the integrated modeling of business processes and corresponding break-glass policies via extended UML Activity diagrams. Moreover, to support the controlled overriding of access rights at the PSM layer we implemented our approach as a break-glass extension for the BusinessActivity library and runtime engine, which is available for download at [1]. We also performed a case study and conducted interviews to evaluate the practical applicability of our integrated modeling approach on real-world processes. In our future work, we plan to conduct further industrial case studies to analyze, for instance, potential issues regarding the complexity and comprehensibility of the graphical syntax of our modeling extension. Moreover, we will investigate how other security-related concepts can be integrated with the break-glass extension. For instance, we intend to integrate our extension with other security extensions, such as the Secure Object Flows (SOF) extension introduced in [26]. Furthermore, we plan to use our generic CIM layer model to extend other process modeling languages (such as BPMN) with a break-glass extension and analyse potential differences between different host languages with respect to these security extensions.

Appendix A. Algorithms for runtime consistency

In this Section, we provide a set of generic algorithms and procedures which can be used independent of a particular programming language and/or software platform in order to implement our formal metamodel for process-related break-glass RBAC

Models presented in Section 3. We implemented these algorithms and corresponding procedures in our BusinessActivities Library and Runtime engine (see [1] and Section 6). For the purposes of this paper, we distinguish algorithms and procedures. Here, an *algorithm* performs certain checks based on the current configuration of a process-related RBAC model. Algorithms either return "true" or "false" and do not have side-effects. A *procedure* operates on the current configuration of a process-related break-glass RBAC model and may include side-effects (i.e., change model elements, relations, or variables). Procedures either return a set or do not return anything (void).

Procedure 1 compiles the set of tasks that are assigned to a subject via the *bbs* and *bbr* mappings (see Definition 2).

Procedure 1. Compile the set of all tasks that are assigned to a subject via a break-glass assignment.

```
Name: breakableTasks
Input: s ∈ S
1: create_empty_set breakable
2: add bbs(s) to breakable
3: for each role ∈ rown(s)
4:     add btown(role) to breakable
5: return breakable
```

Procedure 2. Compile the set of all tasks a particular subject is assigned to.

```
Name: executable Tasks
Input: s ∈ S
1: create_empty_set executable
2: for each role ∈ rown(s)
3:     add town(role) to executable
4: return executable
```

Algorithm 1 checks if it is allowed to allocate a certain task instance to a certain subject. Algorithm 1, line 1 checks if the corresponding subject s is allowed to execute the task type t_{type} the corresponding $t_{instance}$ was instantiated from via a break-glass override (see Procedure 1). Alternatively, a subject may be assigned to a task type via a regular role membership (see Procedure 2). If the corresponding subject is not allowed to execute this particular t_{type}, the respective instance must not be allocated to this subject. Thus, if none of the checks in line 1 or 2 does return "true", Algorithm 1 finally reaches line 3 and returns "false" – meaning that the corresponding subject cannot be allocated to the corresponding task instance in a break-glass scenario.

Algorithm 1. Check if a particular task instance can be allocated to a particular subject via a break-glass policy.

```
Name: isBreakGlassAllocationAllowed
Input: s ∈ S, t_type ∈ T_T, p_type ∈ P_T,
       p_instance ∈ pi(p_type), t_instance ∈ ti(t_type, p_instance)
1: if t_type ∈ breakableTasks(s) then return true
2: if t_type ∈ executableTasks(s) then return true
3: return false
```

After using Algorithm 1 to check if a certain task instance can be allocated to a particular subject, we can actually allocate the task instance via Procedure 3. First, we define the respective subject as the executing-subject of the task instance $t_{instance}$, and the subject's active role as the executing-role of $t_{instance}$ (see lines 1–2). Next, the status of the current $t_{instance}$ is set to "broken" in line 3. Finally, the respective review process for the corresponding process type is instantiated.

Procedure 3. Allocate a certain broken task instance to a particular subject.

```
Name: BreakGlassAllocation
Input: s ∈ S, t_type ∈ T_T, p_type ∈ P_T,
       p_instance ∈ pi(p_type), t_instance ∈ ti(t_type, p_instance)
1: set es(t_instance) = s
2: set er(t_instance) = ar(s)
3: set broken(t_instance) = true
4: instantiate review(p_type)
```

Procedure 4. Compile the set of all task types that have a direct or a transitive subjectbinding relation to a particular $task_a$.

```
Name: allSubjectBindings
Input: task_a ∈ T_T
1: task_a set visited = true
2: create_empty_set directbindings
3: create_empty_set transitivebindings
4:   for each task_b ∈ sbt(task_a)
5:     if !task_b visited then
6:         add task_b to directbindings
7:         add allSubjectBindings(task_b) to transitivebindings
8: return directbindings ∧ transitivebindings
```

In addition to the algorithm and procedures for break-glass task allocation, we also have to adapt the allocation of ordinary (unbroken) tasks. Procedure 5 redefines the *allocateTask*-Procedure from [59]. It describes the steps that are performed to allocate an unbroken task instance to a certain subject. To consider break-glass policies, we now have to include checks for broken subject-bound or role-bound tasks (see also the discussion from Section 3). First, Procedure 5 defines the respective subject as the executing-subject of the task instance $t_{instance}$, and the subject's active role as the executing-role of $t_{instance}$ (see lines 1–2). Next, line 3 checks if the task has a subject-binding to tasks which already have been broken (see Algorithm 2). If no subject-bound task is broken, lines 4–7 perform a lookup to find all instances of subject-bound tasks (see Procedure 4). In particular, all instances of subject-bound tasks that are *not* broken are allocated to the same subject to ensure that all subject-bound tasks are performed by the same subject. If, however, one of the subject-bound tasks is broken the corresponding binding constraint is disabled (see Definition 4). Subsequently, line 8 checks if the task has a role binding to tasks which already have been broken (see Algorithm 3). If no role-bound task is broken, lines 9–11 perform a look-up to find all role-bound tasks and set the executing-role accordingly. In particular, all instances of role-bound tasks that are not broken are allocated to the same role to ensure that all role-bound tasks are performed by the same role. If, however, one of the role-bound tasks is broken the corresponding binding constraint is disabled (see Definition 4).

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

Name: allocateTask
Input: $s \in S, t_{type} \in T_T, p_{type} \in P_T,$
$p_{instance} \in pi(p_{type}), t_{instance} \in ti(t_{type}, p_{instance})$
1: *set* $es(t_{instance}) = s$
2: *set* $er(t_{instance}) = ar(s)$
3: *if* $isSubjectBoundTaskBroken(t_{type}, p_{instance}) == false$ *then*
4: *for each* $type_x \in allSubjectBindings(t_{type})$
5: *for each* $instance_x \in ti(type_x, p_{instance})$
6: *set* $es(instance_x) = s$
7: *set* $er(instance_x) = ar(s)$
8: *if* $isRoleBoundTaskBroken(t_{type}, p_{instance}) == false$ *then*
9: *for each* $type_y \in allRoleBindings(t_{type})$
10: *for each* $instance_y \in ti(type_y, p_{instance})$
11: *set* $er(instance_y) = ar(s)$

Algorithm 2. Check if a subject-bound task has been broken.

Name: isSubjectBoundTaskBroken
Input: $task_x \in T_T, p_{instance} \in P_I$
1: *for each* $task_y \in allSubjectBindings(task_x)$
2: *for each* $instance_y \in ti(task_y, p_{instance})$
3: *if* $broken(instance_y) == true$ *then return* true
4: *return* false

Algorithm 3. Check if a role-bound task has been broken.

Name: isRoleBoundTaskBroken
Input: $task_x \in T_T, p_{instance} \in P_I$
1: *for each* $task_y \in allRoleBindings(task_x)$
2: *for each* $instance_y \in ti(task_y, p_{instance})$
3: *if* $broken(instance_y) == true$ *then return* true
4: *return* false

The complexity of the algorithms and procedures is as follows. Algorithm 1 as well as Procedures 1 and 2 have a linear worst case complexity of $\mathcal{O}(n)$. The complexity of Procedure 3 is constant $\mathcal{O}(1)$. The complexity of Procedure 4 adds up to $\mathcal{O}(n^2)$, while Procedure 5 as well as Algorithms 2 and 3 have a complexity of $\mathcal{O}(n^4)$.

Appendix B. Invariants for break-glass business activity models

Constraint 1. Each BusinessAction defines an attribute called "broken" stating if a certain BusinessAction instance is executed via a break-glass override assignment. For details on the InstanceSpecification element see [35]

```
context BusinessAction inv:
self.instanceSpecification→forAll (i |
  i.slot→exists (b |
    b.definingFeature.name = broken
    and
    b.definingFeature.type.name = Boolean))
```

Constraint 2. Each BusinessActivity defines an attribute called "broken" stating if a certain BusinessActivity instance includes at least one broken BusinessAction:

```
context BusinessActivity inv:
self.instanceSpecification→forAll (i |
  i.slot→exists (b |
    b.definingFeature.name = broken
    and
    b.definingFeature.type.name = Boolean))
```

Constraint 3. For each broken BusinessActivity instance, there has to exist a corresponding reviewProcess:

```
context BusinessActivity inv:
self.instanceSpecification→forAll (i |
  if i.slot→exists (b |
    b.definingFeature.name = broken and
  b.value = true)
  then self.reviewProcess→notEmpty ()
  else true endif
```

Constraint 4. 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 BusinessAction is assigned to a certain role via both mappings:

```
context Role inv:
  self.businessAction→forAll (b |
  self.breakableTask→select (bbr |
    bbr.name = b.name)→isEmpty ())
inv: self.businessAction→forAll (b |
  self.inheritedBreakableTask→select (bbri |
  bbri.name = b.name)→isEmpty ())
inv: self.inheritedTask→forAll (bi |
  self.breakableTask→select (bbr |
    bbr.name = bi.name)→isEmpty ())
inv: self.inheritedTask→forAll (bi |
  self.inheritedBreakableTask→select (bbri |
    bbri.name = bi.name)→isEmpty ())
```

Constraint 5. Each subject is allowed to own a task either regularly (via its role memberships) or via a break-glass override assignment. To separate regular task ownerships from breakable task ownerships, we need to ensure that no BusinessAction is assigned to a certain subject via both mappings:

```
context Subject inv:
  self.roleToSubjectAssignment→forAll (rsa |
  rsa.role.businessAction→forAll (b |
    self.breakableTask→select (bbs |
      bbs.name = b.name)→isEmpty ())
inv: self.roleToSubjectAssignment→forAll (rsa |
  rsa.role.inheritedTask→forAll (bi |
    self.breakableTask→select (bbs |
    bbs.name = bi.name)→isEmpty ())
inv: self.inheritedRole→forAll (ri |
  ri.role.businessAction→forAll (b |
    self.breakableTask→select (bbs |
      bbs.name = b.name)→isEmpty ())
inv: self.inheritedRole→forAll (ri |
  ri.role.inheritedTask→forAll (bi |
    self.breakableTask→select (bbs |
      bbs.name = bi.name)→isEmpty ())
```

Constraint 6. For all broken BusinessAction instances, the executing subjects of corresponding SME tasks do not have to be different:

```
context BusinessAction inv:
self.instanceSpecification→forAll (b |
  b.slot→select (s |
    s.definingFeature.name = broken
    if (s.value = true) then
      self.staticExclusion→forAll (sme |
       sme.instanceSpecification→forAll (i |
         b.slot→forAll (bs |
           i.slot→forAll (is |
             if bs.definingFeature.name = executing
             Subject
               and is.definingFeature.name = executing
               Subject
               then (bs.value = is.value) or
               not (bs.value = is.value)
             else true endif)))))
else true endif))
```

Constraint 7. For all broken BusinessAction instances, the executing subjects of DME tasks do not have to be different:

```
context BusinessAction inv:
self.instanceSpecification→forAll (b |
  b.slot→select (s |
    s.definingFeature.name = broken
    if (s.value = true) then
      self.dynamicExclusion→forAll (dme |
        dme.instanceSpecification→forAll (i |
          b.slot→forAll (bs |
            i.slot→forAll (is |
              if bs.definingFeature.name = executing
              Subject
                and is.definingFeature.name = executing
                Subject
                then (bs.value = is.value) or
                not (bs.value = is.value)
              else true endif)))))
else true endif))
```

Constraint 8. For all broken BusinessAction instances, the executing role of role-bound tasks does not have to be the same:

```
context BusinessAction inv:
self.instanceSpecification→forAll (b |
  b.slot→select (s |
    s.definingFeature.name = broken
    if (s.value = true) then
      self.roleBinding→forAll (rbt |
        rbt.instanceSpecification→forAll (i |
          b.slot→forAll (bs |
            i.slot→forAll (is |
              if bs.definingFeature.name = executing
              Subject
                and is.definingFeature.name = executing
                Subject
                then (bs.value = is.value) or
                not (bs.value = is.value)
              else true endif)))))
else true endif))
```

Constraint 9. For all broken BusinessAction instances, the executing subject of subject-bound tasks does not have to be the same:

```
context BusinessAction inv:
self.instanceSpecification→forAll (b |
  b.slot→select (s |
    s.definingFeature.name = broken
      if (s.value = true) then
        self.subjectBinding→forAll (sbt |
          sbt.instanceSpecification→forAll (i |
            b.slot→forAll (bs |
              i.slot→forAll (is |
                if bs.definingFeature.name = executing
                Subject
                and is.definingFeature.name = executing
                Subject
                then (bs.value = is.value) or
                not (bs.value = is.value)
              else true endif)))))
else true endif))
```

Moreover, the following two constraints must be satisfied which cannot be expressed in OCL (see [35]):

Constraint 10. For all broken BusinessAction instances, context constraints do not have to be fulfilled. Therefore, the $fulfilled_{CC}$ Operations do not have to evaluate to true.

Constraint 11. For all broken BusinessAction instances, context conditions do not have to be fulfilled. Therefore, the $fulfilled_{CD}$ Operations do not have to evaluate to true.

References

[1] Business Activity Library and Runtime Engine, 2012. <http://wi.wu.ac.at/home/mark/BusinessActivities/library.html>.
[2] Oracle Role Manager, 2013. <http://www.oracle.com/us/products/middleware/identity-management/oracle-role-manager/overview/index.html>.
[3] SAP Virsa Firefighter, 2013. <http://sapsecurity.info/virsa-firefighter/>.
[4] J. Alqatawna, E. Rissanen, B. Sadighi, Overriding of access control in XACML, in: Proceedings of the Eighth IEEE International Workshop on Policies for Distributed Systems and Networks, Springer, Washington, DC, USA, 2007, pp. 87–95.
[5] C.A. Ardagna, S.D.C. di Vimercati, S. Foresti, T.W. Grandison, S. Jajodia, P. Samarati, Access control for smarter healthcare using policy spaces, Comput. Secur. 29 (8) (2010) 848–858.
[6] N.D. Belnap, Modern Uses of Multiple-Valued Logics, 1977, pp. 21–32, reidel, (Chapter A useful four-valued logic).
[7] E. Bertino, P.A. Bonatti, E. Ferrari, TRBAC: a temporal role-based access control model, ACM Trans. Inf. Syst. Secur. (TISSEC) 4 (3) (2001).
[8] E. Bertino, E. Ferrari, V. Atluri, The specification and enforcement of authorization constraints in workflow management systems, ACM Trans. Inf. Syst. Secur. (TISSEC) 2 (1) (1999).
[9] R.A. Botha, J.H. Eloff, Separation of duties for access control enforcement in workflow environments, IBM Syst. J. 40 (3) (2001).
[10] G. Bracha, W. Cook, Mixin-based inheritance, in: Proc. of the European Conference on Object-oriented Programming systems, languages and applications (OOPSLA/ECOOP), 1990.
[11] G. Bracha, G. Lindstrom, Modularity meets inheritance, in: Proc. of the IEEE International Conference on Computer Languages, 1992.
[12] A.D. Brucker, H. Petritsch, Extending Access Control Models with Break-Glass, in: Proceedings of the 14th ACM Symposium on Access Control Models and Technologies (SACMAT), 2009.
[13] A.D. Brucker, H. Petritsch, S.G. Weber, Attribute-Based Encryption with Break-glass, in: Proc. of the Workshop In Information Security Theory And Practice (WISTP), 2010.
[14] B. Carminati, E. Ferrari, M. Guglielmi, Secure information sharing on support of emergency management, in: Proc. of the International Conference on Privacy, Security, Risk and Trust, 2011.

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

[16] D.K.W. Chiu, Q. Li, K. Karlapalem, A meta modeling approach to workflow management systems supporting exception handling, Inf. Syst. 24 (1999) 159–184.

[17] D.D. Clark, D.R. Wilson, A comparison of commercial and military security policies, in: IEEE Symposium on Security and Privacy, 1987.

[18] J. Corbin, A. Strauss, Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory, Sage, 2008.

[19] J. Crampton, H. Khambhammettu, Delegation and satisfiability in workflow systems, in: Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT), 2008.

[20] F. Cuppens, N. Cuppens-Boulahia, Modeling contextual security policies, Int. J. Inf. Secur. 7 (4) (2008).

[21] M. Dumas, M.L. Rosa, J. Mendling, R. Maesaku, R. Hajo A, N. Semenenko, Understanding business process models: the costs and benefits of structuredness, in: Prod. of the 24th International Conference on Advanced Information Systems Engineering (CAiSE), 2012.

[22] D.F. Ferraiolo, D.R. Kuhn, R. Chandramouli, Role-Based Access Control, 2nd ed., Artech House, 2007.

[23] A. Ferreira, D. Chadwick, P. Farinha, R. Correia, G. Zao, R. Chilro, L. Antunes, How to securely break into RBAC: the BTG-RBAC model, in: Proceedings of the 2009 Annual Computer Security Applications Conference, December 2009.

[24] A. Ferreira, R. Cruz-Correia, L. Antunes, P. Farinha, E. Oliveira-Palhares, D.W. Chadwick, 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.

[25] C.K. Georgiadis, I. Mavridis, G. Pangalos, 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.

[26] B. Hoisl, S. Sobernig, M. Strembeck, Modeling and enforcing secure object flows in process-driven SOAs: an integrated model-driven approach, Software Syst. Model. (SoSyM) 13 (2) (2014).

[27] S.E. Hove, B. Anda, Experiences from conducting semi-structured interviews in empirical software engineering research, in: Proc. of the 11th IEEE International Software Metrics Symposium (METRICS), 2005.

[28] S. Marinovic, R. Craven, J. Ma, 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.

[29] M. Mernik, J. Heering, A.M. Sloane, When and how to develop domain-specific languages, ACM Comput. Surv. (CSUR) 34 (4) (2005) 316–344.

[30] H. Mouratidis, J. Jürjens, From goal-driven security requirements engineering to secure design, Int. J. Intell. Syst. 25 (8) (2010).

[31] 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 '08, January 2008.

[32] S. Oh, S. Park, Task-role-based access control model, Inf. Syst. 28 (6) (2003).

[33] OMG. OMG Business Process Modeling Notation, <http://www.omg.org/spec/BPMN/1.2/> January 2009, Version 1.2, formal/2009-01-03, The Object Management Group.

[34] OMG. Object Constraint Language Specification, <http://www.omg.org/technology/documents/formal/ocl.htm> February 2010, Version 2.2, formal/2010-02-01, The Object Management Group.

[35] OMG, Unified Modeling Language (OMG UML): Superstructure, <http://www.omg.org/technology/documents/formal/uml.htm> May 2010. Version 2.3, formal/2010-05-03, The Object Management Group.

[36] OMG, Meta Object Facility (MOF) Core Specification – Version 2.4.1, 2011. <http://www.omg.org/spec/MOF>.

[37] D. Povey. Optimistic security: a new access control paradigm, in: Proceedings of the 1999 Workshop on New Security Paradigms, NSPW '99, 2000.

[38] H.F. Ravi Sandhu, Edward Coyne, C. Youman, Role-based access control models, IEEE Comput. 29 (2) (1996).

[39] M. Reichert, P. Dadam, Adept_flex-supporting dynamic changes of workflows without losing control, J. Intell. Inf. Syst. 10 (2) (1998).

[40] M. Reichert, S. Rinderle-Ma, P. Dadam, Flexibility in process-aware information systems, in: K. Jensen, W.M. Aalst (Eds.), Transactions on Petri Nets and Other Models of Concurrency II, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 115–135.

[41] E. Rissanen, B.S. Firozabadi, M. Sergot, Towards a mechanism for discretionary overriding of access control, in: Proceedings of the 12th International Workshop on Security Protocols, 2004.

[42] A. Rodriguez, E. Fernandez-Medina, M. Piattini, Capturing security requirements in business processes through a UML 2.0 activity diagrams profile, in: Workshop Proceedings of Advances in Conceptual Modeling - Theory and Practice (ER Workshops), Lecture Notes in Computer Science (LNCS), vol. 4231, Springer Verlag, 2006.

[43] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Eng. 14 (2) (2009).

[44] N. Russell, A.H.M.T. Hofstede, D. Edmond, Workflow resource patterns: identification, representation and tool support, in: Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05), volume 3520 of Lecture Notes in Computer Science, 2005.

[45] N. Russell, W.M. van der Aalst, A.H.M.T. Hofstede, Exception handling patterns in process-aware information systems, in: International Conference on Advanced Information Systems Engineering (CAiSE), 2006.

[46] S. Schefer, M. Strembeck, Modeling process-related duties with extended UML activity and interaction diagrams, in: Proc. of the International Workshop on Flexible Workflows in Distributed Systems, Workshops der wissenschaftlichen Konferenz Kommunikation in verteilten Systemen (WowKiVS), Electronic Communications of the EASST, vol. 37, March 2011.

[47] S. Schefer, M. Strembeck, Modeling support for delegating roles, tasks, and duties in a process-related RBAC context, in: International Workshop on Information Systems Security Engineering (WISSE), Lecture Notes in Business Information Processing (LNBIP), Springer Verlag, 2011.

[48] S. Schefer, M. Strembeck, J. Mendling, Checking satisfiability aspects of binding constraints in a business process context, in: BPM 2011 Workshops (2), Proc. of the BPM Workshop on Workflow Security Audit and Certification (WfSAC), 2011.

[49] S. Schefer, M. Strembeck, J. Mendling, A. Baumgrass, Detecting and resolving conflicts of mutual-exclusion and binding constraints in a business process context, in: OTM Conferences (1) 2011, Proc. of the 19th International Conference on Cooperative Information Systems (CoopIS), October 2011.

[50] S. Schefer-Wenzl, M. Strembeck, A UML extension for modeling break-glass policies, in: 5th International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA), 2012.

[51] S. Schefer-Wenzl, 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.

[52] S. Schefer-Wenzl, M. Strembeck, Generic support for RBAC break-glass policies in process-aware information systems, in: Proc. of the 28th ACM Symposium on Applied Computing (SAC), 2013.

[53] S. Schefer-Wenzl, M. Strembeck, A. Baumgrass, An approach for consistent delegation in process-aware information systems, in: Proc. of the 15th International Conference on Business Information Systems (BIS), Lecture Notes in Business Information Processing (LNBIP), vol. 117, Springer, 2012.

[54] D.C. Schmidt, Model-driven engineering – guest editorś introduction, IEEE Comput. 39 (2) (2006).

[55] B. Selic, The pragmatics of model-driven development, IEEE Software 20 (5) (2003).

[56] T. Stahl, M. Völter, Model-Driven Software Development, John Wiley & Sons, 2006.

[57] M. Strembeck, Embedding policy rules for software-based systems in a requirements context, in: Proc. of the 6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY), June 2005.

[58] M. Strembeck, Scenario-driven role engineering, IEEE Secur. Privacy 8 (1) (2010).

[59] M. Strembeck, 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), Lecture Notes in Computer Science (LNCS), vol. 6426, Springer Verlag, 2010.

[60] M. Strembeck, J. Mendling, Modeling process-related RBAC models with extended UML activity models, Inf. Software Technol. 53 (5) (2011).

[61] M. Strembeck, G. Neumann, An integrated approach to engineer and enforce context constraints in RBAC environments, ACM Trans. Inf. Syst. Secur. (TISSEC) 7 (3) (2004).

[62] M. Strembeck, U. Zdun, An approach for the systematic development of domain-specific languages, Software: Pract. Exper. (SP&E) 39 (15) (2009).

[63] K. Tan, J. Crampton, 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.

[64] R.K. Thomas, R.S. Sandhu, Task-Based Authorization Controls (TBAC): a family of models for active and enterprise-oriented authorization management, in: Proceedings of the IFIP TC11 WG11.3 Eleventh International Conference on Database Securty XI: Status and Prospects, August 1997.

[65] W.M.P. van der Aalst, M. Rosemann, M. Dumas, Deadline-based escalation in process-aware information systems, Decis. Support Syst. 43 (2007) 492–511.

[66] S. von Stackelberg, K. Böhm, M. Bracht, Embedding 'break the glass' into business process models, in: OTM Conferences (1), 2012.

[67] J. Wainer, P. Barthelmess, A. Kumar, W-RBAC – a workflow security model incorporating controlled overriding of constraints, Int. J. Coop. Inf. Syst. (IJCIS) 12 (4) (2003).

[68] J. Warner, V. Atluri, Inter-instance authorization constraints for secure workflow management, in: Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies (SACMAT), June 2006.

[69] B. Weber, S. Rinderle, M. Reichert, Change patterns and change support features in process-aware information systems, in: International Conference on Advanced Information Systems Engineering (CAiSE), 2007.

[70] D. Wetherall, C.J. Lindblad, Extending Tcl for dynamic object-oriented programming, in: Proc. of the USENIX Tcl/Tk Workshop, 1995.

[71] C. Wolter, A. Schaad, Modeling of task-based authorization constraints in BPMN, in: G. Alonso, P. Dadam, M. Rosemann (Eds.), Business Process Management, Lecture Notes in Computer Science, vol. 4714, Springer, Berlin/Heidelberg, 2007.

[72] C. Wolter, A. Schaad, 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.

[73] U. Zdun, M. Strembeck, G. Neumann, Object-based and class-based composition of transitive mixins, Inf. Software Technol. 49 (8) (2007).