International Journal of Cooperative Information Systems
© World Scientific Publishing Company

OBJECT-SPECIFIC ROLE-BASED ACCESS CONTROL

NICOLAS MUNDBROD AND MANFRED REICHERT

Institute of Databases and Information Systems, Ulm University, James-Franck-Ring, 89073 Ulm, Germany

Received (Day Month Year)
Revised (Day Month Year)

The proper management of privacy and security constraints in information systems in general and access control in particular constitute a tremendous, but still prevalent challenge. Role-based access control (RBAC) and its variations can be considered as the widely adopted approach to realize authorization in information systems. However, RBAC lacks a proper object-specific support, which disallows establishing the fine-grained access control required in many domains. By comparison, attribute-based access control (ABAC) enables a fine-grained access control based on policies and rules evaluating attributes. As a drawback, ABAC lacks the abstraction of roles. Moreover, it is challenging to engineer and to audit the granted privileges encoded in rule-based policies. This paper presents the generic approach of object-specific role-based access control (ORAC). On one hand, ORAC enables information system engineers, administrators and users to utilize the well-known principle of roles. On the other, ORAC allows realizing the access to objects in a fine-grained way where required. The approach was systematically established according to well-elicited key requirements for fine-grained access control in information systems. For the purpose of evaluation, the approach was applied to real-world scenarios and implemented in a proof-of-concept prototype demonstrating its feasibility and applicability.

Keywords: object-specific role-based access control, access control, authorization, role-based access control, instance-specific access control

1. Introduction

Information systems are today's key technology to automate business processes as well as to offer related products and services to customers, employees, and other stakeholders. For enterprise information systems, sophisticated access control to business processes, business functions, and business objects is indispensable to ensure information integrity, privacy and availability. Access control has therefore always been a key concern for information system engineers and researchers[27,26], and numerous approaches with specific pros and cons have been proposed in the literature[26]. Especially, role-based access control (RBAC)[25,7] and attribute-based access control (ABAC) [13,10] as well as numerous variants of these two approaches[9,2,4,1,8,15,11,13,30] have been in the focus of both practitioners and researchers for a long time.

1.1. *Problem Statement*

As of today, RBAC is still the most widely used access control paradigm whose ease of use and manageability are appreciated by information system engineers and administrators[5]. In particular, the resources and application functions, which may be accessed by users filling a specific role, can be easily determined at both design and run time. As a drawback, contemporary RBAC approaches do not allow for fine-grained, object-specific access control that considers, for example, the *ownership* of existing data objects in an information system as well as the relationship between these objects.

Example 1.1. (Patient Treatment Processes) In a hospital information system, digital patient records with their various subordinated objects (e.g., medical findings, therapy plans) are managed digitally. A subject with role *physician* should be only allowed to access those patient records (including the connected objects) he or she is responsible for in order to prevent security issues and to meet privacy requirements.

Based on traditional RBAC, role *physician* would be granted access to *all* patient records. To cope with scenarios as the one presented in Example 1.1, roles are typically specialized (e.g., *physician working in women hospital*). On one hand, this allows for a more fine-grained access control. On the other, the number of roles rapidly grows, making role maintenance hardly manageable over time and, thus, hampering scalability. Additionally, RBAC does not allow specifying that a role owner may only access a specific object with all its subordinated objects (e.g., a medical record with findings and therapy plans). Moreover, as the assignments of users to specific roles are often integrated with the source code of an information system, the introduction of new roles might require a new information system release. Finally, to properly support scenarios like the one from Example 1.1, a specific role would have to be created for every single patient (e.g., *physician of John Doe*).

By contrast, ABAC does not rely on any specified roles with linked privileges[13]. Instead, access privileges are dynamically acquired by virtue of attributes shared by the subject, who wants to gain access, and the requested resource. To realize access control, usually, ABAC-based information systems rely on a large set of policy rules, which are jointly set up and maintained by domain experts and security engineers. At run time, these rules are continuously evaluated to properly handle access requests. In comparison to RBAC, however, ABAC suffers from its fine granularity. The definition of policy rules constitutes a sophisticated task due to the complex data structures comprising many attributes on one hand and the evolution of these structures over time on the other. Due to the complex structure of the policies, in addition, it is challenging to derive the actual set of *privileges* a certain user or group of users (i.e., role) shall obtain[15,5]. Finally, with ABAC, it is by far not trivial to define rules enabling a hierarchical access to interconnected objects as discussed in the context of Example 1.1 (e.g., a medical record with findings).

Overall, both RBAC and ABAC do not always provide the necessary support required to realize object-specific access control. In particular, there exists no fine-grained, role-based access control approach that properly integrates the given data structure and concurrently allows determining risk exposure adequately.

1.2. *Contribution*

This paper introduces **object-specific role-based access control** (ORAC), which enables information system engineers and administrators to jointly establish and manage fine-grained, role-based access control as required in many application domains. In particular, subjects may only perform object-specific actions they are allowed to access, based on object-specific roles and corresponding role assignments. For example, a physician may only read and update the records of those patients he is responsible for. In addition, ORAC allows expressing hierarchical access on the child objects of an object. For example, if a physician has access to a patient record, he may access all therapy plans linked to that patient record as well.

To evaluate the feasibility and applicability of ORAC, the approach was applied in two case studies with specific application scenarios. Furthermore, an advanced proof-of-concept prototype was developed. Using the latter, we were able to show that the definition of privileges can be intuitively integrated into the information system development process through (method) annotations, increasing the transparency and explicitness of access privileges at design time.

The remainder of this paper is structured as follows: Section 2 introduces a real-world scenario and defines key terms needed for understanding this work. Section 3 then discusses fundamental use cases and key requirements for ORAC. In Section 4, an overview of the ORAC approach is given and its key components are presented. Section 5 addresses the process of establishing and enforcing object-specific role-based access control. In Section 6, the proof-of-concept prototype as well as two case studies are presented to evaluate the applicability of ORAC. Related work is discussed in Section 7. Finally, Section 8 concludes the paper and gives an outlook.

2. Backgrounds

This section introduces key terms required for understanding this work. To illustrate issues that emerge when coping with access control on complex data structures, a real-world scenario, which we elaborated in prior work[17,18], is introduced first. To ease the understanding of ORAC, the scenario is used as a *running example*.

Example 2.1. (Recruiting Processes) In a *recruitment process*, *applicants* may apply for a job offer defined by a *manager* of a functional division. Once an *application* has been submitted by an applicant, the responsible *recruiter* in the human resource (HR) department is notified. The overall process goal is to decide which applicant shall get the job. If an application is ineligible (e.g., formal requirements are not met), the applicant will be immediately rejected. By contrast, the recruiter may request one or more internal *reviews* for each applicant. Based on the reviews, the recruiter decides on the applications and will initiate further steps, e.g., *interviews* with selected applicants. To document the interviews, *minutes* are created and linked to the corresponding applications. At the end of this process, the recruiter may create a *contract offer* for an applicant. If the latter agrees with the offer, subsequent processes will be triggered afterwards. Alternatively, a contract may be offered to other applicants, or the entire recruitment process may have to be started once more.

2.1. *Objects and Models*

The scenario introduced in Example 2.1 confirms that *subjects* with specific *roles* (e.g., recruiter) may deal with a variety of interconnected *objects* of different *object types*[28]. In the scope of the *recruitment process* object, for example, dozens of objects of type *application* may be submitted (and, thereby, created) by applicants individually. Moreover, every *application* object may be linked to several objects of type *review*. As a consequence, fine-grained access control needs to be integrated with the given *data model* and be adopted to it. In particular, access control should take into account that, at run time, the subjects interacting with the information system dynamically create *complex graphs of objects* based on the given data model.

To properly design, enact and implement an object-specific role-based access control in information systems, first of all, four relevant terms are sketched in the following: *object types*, *objects*, *object models*, and *object model instances*. An object type corresponds to a data structure that determines the structure of an object by defining a set of attributes. The latter, in turn, are defined by a data type (e.g., String) and a name. Every object exactly references one object type and features values for the attributes of the object type. Figure 1 illustrates a sample application object type (a) with corresponding application objects (b).

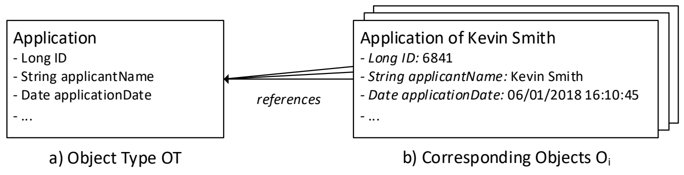
Application
- Long ID
- String applicantName
- Date applicationDate
- ...
references
Application of Kevin Smith
- Long ID: 6841
- String applicantName: Kevin Smith
- Date applicationDate: 06/01/2018 16:10:45
- ...
a) Object Type OT
b) Corresponding Objects O_i

Fig. 1: Sample Object Type with corresponding Objects

To consider the relationship between object types and objects, *object models* and *object model instances* need to be discussed next: An object model consists of finite sets of object types and corresponding *relationship types*. Each relationship type connects a source object type with a target object type, and further specifies a minimum and a maximum cardinality. At run time, an object model may be used to create object model instances. An object model instance, in turn, consists of two finite sets of objects and object relations, of which each either refers to an object type or a relation type of the underlying object model. Fig. 2 illustrates an object model with a corresponding object model instance related to Example 2.1 (for the sake of readability, possible attributes of the object types are omitted).

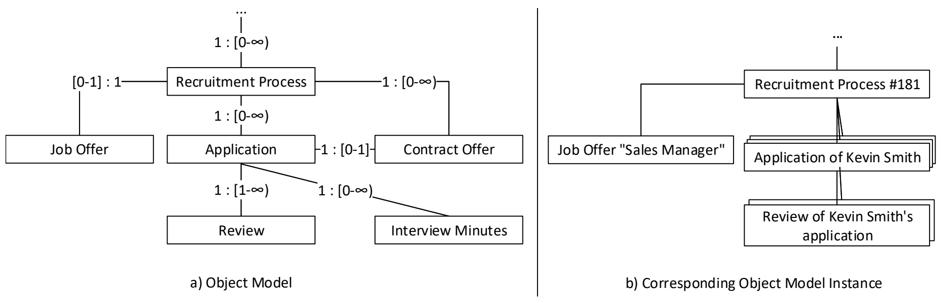

Fig. 2: Sample Object Model with corresponding Object Model Instance

Example 2.1 and Fig. 2 highlight that objects may depend on other objects, e.g., a *recruitment process* object type may be in a *parental relationship* to the *job offer*, *application*, and *contract offer* object types. Consequently, *application* objects should be automatically deleted when removing its *parental object* with type *recruitment process* (i.e., cascading delete). Note that such knowledge about parental relationships between object types is crucial for enabling an object-specific access control.

2.2. *Subjects, Actions and Privileges*

Subjects, *actions*, and *privileges* are fundamental concepts of any access control approach. Subjects are considered as users or technical agents (e.g., client application) interacting with implemented business processes, business functions, and data objects of an information system. In turn, users–as subjects–may belong to *organizational units* (e.g., *HR department*) and obtain *organizational roles* (e.g., *recruiter*). Note that this organizational context becomes crucial as soon as access shall be granted to a set of subjects (cf. Section 4).

When ensuring access control at run time, it needs to be checked whether subjects may perform specific *actions* (cf. Fig. 3). For example, an action may create a new object, change its attributes, or remove an object from its parental object. Hence, actions are often performed with respect to specific objects. Note that in object-oriented programming languages, actions are usually represented as class methods, e.g., manipulating the attributes of an object or calculating a desired result based on several objects.

Finally, a *privilege* denotes the right to perform one or more specific actions. Typically, a privilege is granted to a subject directly or indirectly through, for example, an organizational role assigned to the subject. An action, in turn, may require several privileges granted to the subject. The other way round, a certain privilege may be connected to several actions (i.e., n:m relationship). To enforce access control at run time, dedicated algorithms determine whether a subject obtains one or more privileges required to perform an action.

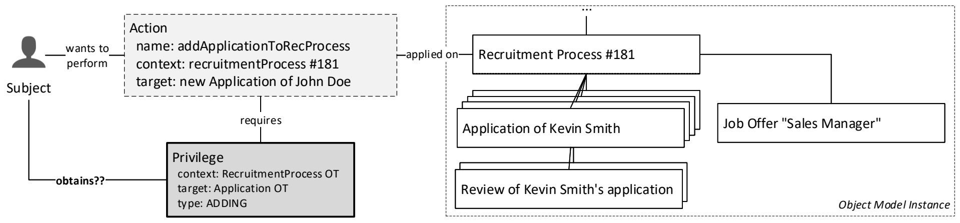
Subject
wants to perform
Action
name: addApplicationToRecProcess
context: recruitmentProcess #181
target: new Application of John Doe
applied on
requires
obtains??
Privilege
context: RecruitmentProcess OT
target: Application OT
type: ADDING
...
Recruitment Process #181
Application of Kevin Smith
Job Offer "Sales Manager"
Review of Kevin Smith's application
Object Model Instance

Fig. 3: Relationship between Subjects, Actions and Privileges

3. Requirements

To systematically develop object-specific role-based access control, we thoroughly analyzed access control scenarios in various application domains, including healthcare[21,24], human resource management[19,16], and automotive engineering[22,29]. In case studies conducted in these domains, we encountered various challenges that had resulted from the lack of an object-specific role-based access control.

To elaborate the requirements, first of all, we analyzed the different types of actions performed on objects in the considered scenarios. Based on this analysis, we derived and defined *action use cases* (cf. Section 3.1), providing the necessary basis for specifying the different privileges required for a fine-grained access control (cf. Fig. 3). Furthermore, the action use cases foster the general understanding of the dynamic interactions between subjects and the data model of the information system. In this context, we further analyzed in which sequence the different user groups had performed the actions on the objects, what results of the performed actions had been generated, and what restrictions had been put in place (e.g., users of group A may only access specific objects they *own*). Finally, these insights were aggregated in a set of key requirements regarding the development of an object-specific role-based access control.

3.1. *Action Use Cases*

The following *action use cases* (AUCs) highlight different *action types* that may be performed on objects in an object model instance. The proper understanding of the action types is important: The privileges required to perform actions shall reflect the semantics of the actions in order to enable sound access control decisions. While Table 1 depicts essential action use cases for objects (i.e., to create, read, update, or delete objects), Table 2 introduces more sophisticated action use cases.

Table 1: Basic Action Use Cases

ID and Name	**Effects on the Information System Level**
AUC01 *Add object*	A subject adds a new object to a parental object or to the root-level of the object model instance. For example, an applicant may create/submit an application, which is then added to a specific recruitment process.
AUC02 *Read object*	A subject may read an entire object including its attributes. For example, an applicant should be able to read his application.
AUC03 *Read object attribute(s)*	A subject may read a subset of the object attributes. For example, an employee of a functional division may only view a partial set of the attributes of an application.
AUC04 *Read objects*	A subject may retrieve a set of child objects of a given parental object. For example, a recruiter may want to retrieve all applications of a specific recruitment process.
AUC05 *Read objects with subset of their attributes*	In addition to AUC04, a subject may retrieve a set of objects with only a subset of attributes. For example, an employee of a functional division may view the applications of a recruitment process, but the applications must not be displayed in all (personal) details.
AUC06 *Update object*	A subject wants to update an object, i.e., its attributes. Thereby, the access shall not be restricted to a subset of object attributes (compare AUC07); instead, all attributes may be updated. For example, a recruiter may want to update a contract offer over time.
AUC07 *Update object attributes*	A subject may only update a specified subset of the object's attributes. For example, an applicant may only update certain parts of his application after having submitted it initially.
AUC08 *Remove object*	A subject may remove an object from its parental object(s). For example, a recruiter may delete a review (from the recruitment process) not being accurate enough.

Regarding privilege-based access control, one of the key challenges is to determine the appropriate granularity for defining privileges as well as to establish a meaningful privilege classification to ease the allocation of privileges in general. If the classification is too fine-grained and privileges become too detailed, one can hardly allocate privileges to roles and, consequently, to subjects. If privileges and their classification are too coarse-grained, in turn, the expressiveness of the access control approach will be limited in general.

We took the action use cases depicted in Tables 1 and 2 as basis for deriving a handy set of 12 different types of privileges (cf. Section 4.2). Though the proposed set of action types might be not complete for covering access control in all kind of information systems, they can be considered as an important foundation or even be used as templates to derive action types and, finally, types of privileges for a specific information system currently being under development.

Table 2: Advanced Action Use Cases

ID and Name	Effects on the Information System Level
AUC09 *Link object to another object*	A subject may create a non-parental relationship between two objects $o_x, o_y \in O_{all}$ with $o_x \neq o_y$ and o_y not being a subordinated object of o_x. For example, a recruiter may link a contract offer to a specific application.
AUC10 *Unlink object from another object*	A subject may remove a non-parental relationship between an object $o_x \in O_{all}$ and another object $o_y \in O_{all}$, which was established previously by applying AUC09. For example, a recruiter may remove a link between a contract offer and an application.
AUC11 *Search objects*	A subject may search for objects with object type $ot_x \in OT_{all}$ matching the given search parameters. In this context, one needs to consider AUC02 and AUC03, as subjects may only retrieve objects (and read corresponding attributes) for which access has been granted to them. For example, an employee may look for an application with certain properties. However, she may only retrieve selected applications and, in addition, access only a subset of the attributes of the selected applications.
AUC12 *Perform complex action*	A subject may perform a complex action, which cannot be classified according to the action use cases AUC01-AUC11. The complex action is performed in respect to a given object $o_x \in O_{all}$ and its subordinated objects $o_{x+1}, o_{x+2}, \ldots, o_{x+n}$. For example, a recruiter may execute an analysis function providing performance indicators related to the recruitment process (e.g., process time).

3.2. *Key Requirements*

To establish a solid base for object-specific role-based access control, we elicited key requirements based on the insights from prior case studies conducted in the aforementioned application areas.

R01: ***User Sovereignty***—The users of an information system, i.e. its subjects, shall be enabled to manage the key aspects of the access control approach themselves. Amongst others, this includes the allocation of privileges related to object(s) to other users. This allocation, in turn, needs to be accomplished in a safe and controlled way (e.g., through predefined roles "bundling" privileges). Finally, to ensure that a user may only grant those privileges to other users he or she obtains, an approval process is required.

R02: ***Definition of Privileges***—The definition, integration and management of a large set of privileges in information systems constitutes a costly and error-prone task. Hence, it is highly desirable to ease this process and to homogenize the set of created privileges (cf. Section 3.1). In particular, the defined privileges should correspond to the business functions of an information system (i.e., actions), and be definable by developers in a controlled and intuitive manner.

R03: ***System-specific Roles***—Organizations usually use *roles* to describe functions performed by several individuals. For example, a subject $s_{Michael}$ heading the HR department may obtain the *organizational role* of a *director*. As discussed, however, organizational roles are often not fine-grained enough to grant access to an information system. For example, subject $s_{Michael}$ may obtain several *system-specific roles* in an information system: he may act as *recruiter*, as an *applicant* (as he might internally apply for a job as well), and as a *manager* in the context of different *recruitment processes* or *applications*. Thus, system-specific roles are required to additionally enhance and refine existing organizational roles (cf. Section 2.2).

R04: ***Object-specific Role Assignments***—System-specific roles need to be assignable to concrete objects of an object model instance. In particular, a system-specific role is only valid if it references existing objects. Hence, a subject may obtain a system-specific role in relation to a set of objects the role shall provide privileges for. Consequently, a subject *may obtain various system-specific roles, which are object-specifically assigned* in an information system employing object-specific role-based access control.

R05: ***Hierarchical Privileges***—In general, a particular object may always have several child objects (cf. Section 2.1). For example, a *recruitment process* object may have a set of dynamically added *application* objects, which shall be always accessible by subjects owning the object-specific role *recruiter*. As a consequence, subjects may require hierarchical privileges for all child objects of a specific object. This way, it is ensured that newly created child objects may be properly accessed (without additional need to reference them explicitly after their creation).

R06: ***Customizable Roles***—In response to domain-specific requirements, system-specific roles shall be customizable for privileged users at run time as well. For example, an administrator of an HR system may want to remove a privilege (e.g., update of the application) granted to applicants. Another use case concerns the implementation of new features in an information system. In the latter context, roles need to be created and adjusted in order to incorporate the privileges coming along with the new features.

R07: ***Semi-Automatic Role Assignments***—System-specific roles need to be assignable as convenient as possible. As a characteristic use case consider the creation of an object. Often, the creator of an object should automatically perceive a set of privileges (object-ware role) related to this object. For example, if a user of an HR system creates a recruitment process object (and thereby starts the corresponding recruitment process), she should be automatically assigned to the role of a recruiter in relation to this object in order to be able to manage the process properly. Furthermore, in case a subject wants to assign a role to another one, she shall only retrieve the roles assignable in this situation.

4. Object-specific Role-based Access Control

To address the key requirements for a fine-grained, role-based access control (cf. Section 3) and to overcome the drawbacks of existing approaches (cf. Sections 1 and 7), we developed the approach of object-specific role-based access control (ORAC). In particular, the latter enables the fine-grained integration of role-based access control with a given object model. To ease the understanding of the ORAC approach, first of all, we provide an overview of the overall approach and, especially, discuss the interplay of the key ORAC components (cf. Fig. 4). Based on these insights, the ORAC components are presented in detail in the following in order to establish a deep understanding of the approach and its benefits.

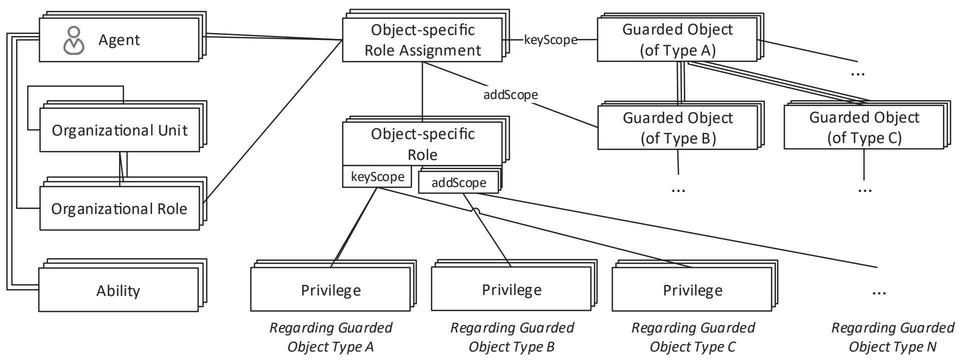
Agent
Organizational Unit
Organizational Role
Ability
Object-specific Role Assignment
keyScope
addScope
Guarded Object (of Type A)
Guarded Object (of Type B)
Guarded Object (of Type C)
...
Object-specific Role
keyScope
addScope
Privilege
Regarding Guarded Object Type A
Privilege
Regarding Guarded Object Type B
Privilege
Regarding Guarded Object Type C
...
Regarding Guarded Object Type N

Fig. 4: Object-Specific Role-Based Access Control in a Nutshell

In a nutshell, the ORAC approach provides *guarded objects*, *privileges*, *object-specific roles* (cf. Requirements R03 and R06), *organizational entities*, *agents*, and *object-specific role assignments* (cf. Requirement R04) as key components.

Objects, whose access is controlled by ORAC, are denoted as *guarded objects*. Thus, an action manipulating a guarded object is only accessible for agents who have been granted access to this action before. Interacting with an ORAC-based information system, *agents* may either be humans (i.e., a user) or robots (e.g., another system). In turn, *organizational entities* allow modeling the organizational context of agents. More precisely, there exist *organizational units* (e.g., *HR department*), *organizational roles* (e.g., *director*), and *abilities* (e.g., *office skills*).

In general, organizational units may be hierarchically organized to cover common organizational structures as they can be found in enterprises. In addition, every organizational unit (e.g., *HR department*) may further entail several organization roles (e.g., *head of department*). Each organizational role, in turn, may be linked to a set of agents acting in this organizational role. Finally, abilities (e.g., *office skills*) allow grouping and selecting agents across different organizational units based on their skills. In summary, organizational entities may be used to increase the number of options for assigning agents to object-specific roles.

Object-specific role assignments, in turn, constitute the key component of ORAC, tying together agents (directly or indirectly through organizational entities), object-specific roles, and guarded objects. In particular, the purpose of an object-specific role assignment is to specify that one or more agents obtain an object-specific role in relation to one or multiple guarded objects. In this respect, every *object-specific role* entails a key scope and, optionally, multiple additional scopes. These scopes are used to specify that certain privileges are only granted in relation to guarded objects of pre-defined guarded object types. Furthermore, the scopes determine the way object-specific role assignments are created: at run time, an object-specific role may be assigned to one or several guarded objects matching the pre-defined scopes of the object-specific role. The privileges, which are linked to the respective scope, are then evaluated only in relation to the selected guarded object and, if specified, its child objects.

Example 4.1. (Object-specific Role Recruiter) An object-specific role *recruiter* shall only be assigned to guarded objects of type *recruitment process*. Hence, the key scope of role *recruiter* refers to guarded object type *recruitment process*. For a particular agent, e.g. a_{Lisa}, who is supposed to obtain the role of a *recruiter* with respect to a *recruitment process* object, an object-specific role assignment is then created. This role assignment interconnects a_{Lisa}, the object-specific role *recruiter* (via its key scope), and a specific guarded object of type *recruitment process*.

Based on object-specific roles and role assignments, the set of actions an agent may perform on a guarded object can be accurately granted and restricted, respectively. To facilitate the assignment of privileges to object-specific roles as well as to establish an effective object-specific access control, each privilege is classified by an action type (cf. Section 3.1), which obtains a pre-specified scope as well. In addition, to allow for the hierarchical application of privileges (cf. Section 3, Requirement R05), an object-specific role may reference a set of object-related privileges as well as a set of hierarchical privileges for every scope separately.

Example 4.2. (Privileges of the Object-specific Role Recruiter) The object-specific role *recruiter* has a key scope targeting the guarded object type *recruitment process*. In turn, the key scope is linked to privileges for updating a recruitment process as well as to various hierarchical privileges for managing child (guarded) objects of type *application*. Hence, an agent assigned to the object-specific role *recruiter* and a *recruitment process* guarded object may then manage all guarded child *application* objects of the recruitment process.

To finally enforce access control with respect to a certain action to be performed on a guarded object, first of all, the object-specific role assignments referring to the guarded object are considered. It is checked whether the given role assignments are linked to the respective agent and to an object-specific role the required privilege has been assigned to. If this applies, access will be granted immediately. Alternatively, the object-specific role assignments, referencing the parental guarded objects of the initially considered one (cf. Section 4.1), and the current agent are considered stepwise to check whether access can be granted anyhow.

To illustrate the following discussion, Fig. 5 depicts the scenario introduced in Section 2 with corresponding ORAC components. Sections 4.1–4.4 present the fundamental concepts of *guarded objects*, *privileges*, *object-specific roles*, and *object-specific role assignments* in detail. Section 5 then discusses the algorithms necessary to enforce ORAC at run time, highlighting the interplay of the ORAC components.

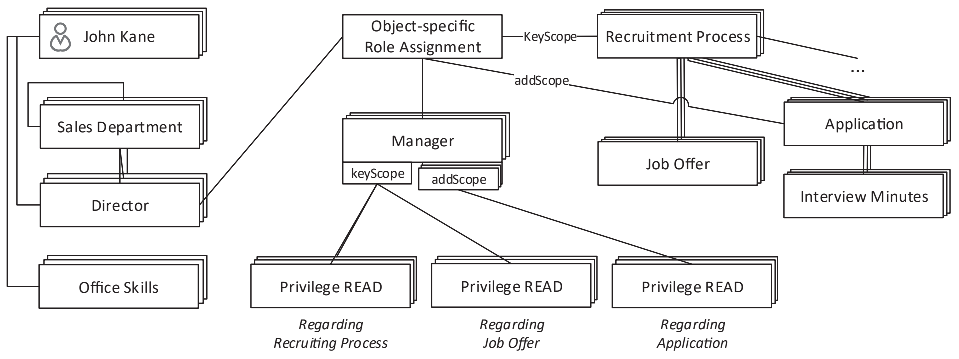

Fig. 5: Object-specific Role-based Access Control applied to Application Scenario

4.1. *Guarded Objects*

To establish object-specific role-based access control, it is essential to differentiate between objects protected by access control and those that may be accessed without any restrictions. Therefore, *guarded objects* shall denote objects whose access is protected based on ORAC. Furthermore, to support the hierarchical application of privileges (cf. Requirement R05), the relationship between guarded objects has to be taken into account as well. To meet Requirement R05, it becomes crucial to determine the parental guarded objects of a particular guarded object at run time to properly enforce ORAC.

In this context, a peculiarity of object relationships in object model instances need to be discussed: an object of a specific object type may have multiple parental objects. Example 4.3 and Fig. 6 illustrate this issue.

Example 4.3. (Multiple Parental Objects) Objects of object type *document* are subordinated to objects of types *application* and *job offer*. Additionally, a particular object of type *document*, e.g., a document containing regulations, may be child of different *contract offer* objects. Hence, *document* objects may have two or more parental objects.

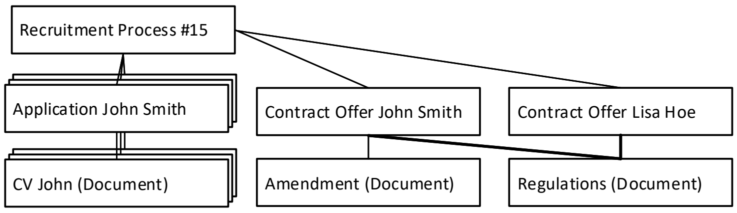

Fig. 6: Example of an Object with two Parents

In general, each guarded object may have an arbitrary number of child and parental guarded objects. Accordingly, Definition 4.1 formally introduces *guarded objects types* and *guarded objects* including the notion of parental guarded objects.

Definition 4.1. (Guarded Object Type and Guarded Object) Let OT_{all} be the set of all object types and O_{all} be the set of all objects.

(a) A **guarded object type** *got* is an object type protected by ORAC. With GOT_{all} denoting the set of all guarded object types, a guarded object type is defined as a tuple $got = (ot, parentGOT) \in GOT_{all}$, where

- $ot \in OT_{all}$ is a object type,
- $parentGOT : GOT_{all} \rightarrow \mathcal{P}(GOT_{all})$ is a function returning the parental guarded object types of *got*.

(b) A **guarded object** *go* is an object protected by ORAC. With GO_{all} denoting the set of all guarded objects, a guarded object is defined as a tuple $go = (got, o, parentGO) \in GO_{all}$, where

- $got = (ot, parentGOT)$ is the guarded object type of *go*,
- $o \in O_{all}$ is a object of type ot embedded in *go*,
- $parentGO : GO_{all} \rightarrow \mathcal{P}(GO_{all})$ is a function returning the parental guarded objects of *go*.

Note that ORAC excludes the creation of cycles when defining the parental relationships in a object model[a]. Furthermore, agents and all kinds of organizational entities may be represented as guarded object types as well. In this context, use cases like updating personal user data or establishing an organizational model can be supported as well.

[a]A cycle of parental relationships would rule out the hierarchical application of privileges

4.2. *Privileges*

Privileges denote the right to perform one or multiple actions on a guarded object, and are required to safely check access at run time. When a subject obtains a privilege in relation to a guarded object based on an object-specific role (cf. Section 4.3) and a corresponding object-specific role assignment (cf. Section 4.4), access is granted and the action may be performed. To make privileges as accurate and meaningful as required and to properly allocate them to the scope of an object-specific role, every privilege has an action name, an action type (cf. Section 3.1) denoting the purpose of the privilege, and a scope. Shaped together by a target and context guarded object type, the privilege scope specifies the objects the respective privilege may be applied to. Thereby, the context guarded object type is a possible parental guarded object type of the target object type (see $parentGOT$ of Definition 4.1). Definition 4.2 formally introduces privileges.

Definition 4.2. (Privilege) A **privilege** $p \in P_{all}$ is defined as a tuple $p = (actionName, actionType, targetGOT, contextGOT)$, where

- $actionName$ is the name of the action, p is granting,
- $actionType \in \{READ, READ\_ATTRIBUTE, ADD, ADD\_LINK$ $UPDATE, UPDATE\_ATTRIBUTE, REMOVE, REMOVE\_LINK,$ $LISTING, SEARCH, ADVANCED\}$ is the action type of p,
- $targetGOT \in GOT_{all}$ is the target guarded object type of p,
- $contextGOT \in GOT_{all}$ is the context guarded object type of p,

P_{all} denotes the set of all privileges.

To illustrate Definition 4.2, Table 3[b] provides examples of privileges whose definitions refer to the action use cases (presented in Section 3.1) as well as the application scenario introduced in Section 2.

Table 3: Exemplary Privileges

ActionName	ActionType	TargetGOT	ContextGOT
addRecruitmentProcess	ADD	*RecProcess*	*HR System*
...	...	...	...
getJobOffer	READ	*Job Offer*	*Job Offer*
getTitle	READ_ATTRIBUTE	*Job Offer*	*Job Offer*
setTitle	UPDATE_ATTRIBUTE	*Job Offer*	*Job Offer*
removeJobOffer	REMOVE	*Job Offer*	*RecProcess*
...	...	...	...
addApplication	ADD	*Application*	*RecProcess*
LinkApplication-ToContractOffer	ADD_LINK	*Application*	*ContractOffer*
...	...	...	...

[b]Due to space limitations, RecProcess abbreviates the *recruitment process* guarded object type

4.3. *Object-specific Roles*

To ensure the proper assignment of privileges to agents and guarded objects as well as to provide system-specific roles (cf. Requirement R03), we propose the use of *object-specific roles* bundling privileges in relation to specified scopes. As object-specific roles can be safely specified, the assignment of privileges to meaningful object-specific roles is well conductible by eligible subjects (e.g., administrators; cf. Requirement R06). By using scopes in object-specific roles, ORAC enables the hierarchical access of child guarded objects (cf. Requirement R05) as well as the coverage of more sophisticated access control use cases (cf. Example 4.4).

Definition 4.3 introduces *object-specific roles* formally.

Definition 4.3. (Object-specific Role) Let $got_{List} = (GOT_{trace}, order)$ be a tuple denoting a *list of guarded object types*, where

- $GOT_{trace} \subseteq \mathcal{P}(GOT_{all})$ is a set of guarded object types,
- $order : GOT_{trace} \rightarrow \mathbb{N}$ is an ordering function assigning a unique number to every $got \in GOT_{trace}$.

Let further GOT_{List} be the set of all possible lists of guarded object types.

Then: An **object-specific role** $osr \in OSR_{all}$ is defined as a tuple $osr = (P, SC, keySC, AddSC, scp_{sc}, scp_{hi}, scpParams, OSR_{req})$, where

- $P \subseteq P_{all}$ is the set of privileges referenced by osr,
- $SC \subset GOT_{all} \times GOT_{List}$ is the set of scopes comprised by osr. Every $sc \in SC$ corresponds to a tuple $sc = (targetGOT, contextGOT_{List})$, where
  - $targetGOT \in GOT_{all}$ is the target guarded object type,
  - $contextGOT_{List} \in GOT_{List}$ is a list of contextual, parental guarded object types.
- $keySC \in SC$ is the key scope of osr,
- $AddSC \subset SC, \forall addSC \in AddSC : addSC \neq keySC$ is the set of additional scopes; the target guarded object type of $keySC$ must be part of every list of contextual, parental guarded object types of the additional scopes,
- $scp_{sc} : SC \rightarrow \mathcal{P}(P)$ assigns a set of scope-specific privileges $P_{sc} \subseteq P_{osr}$ to each scope $sc \in SC$,
- $scp_{hi} : SC \rightarrow \mathcal{P}(P)$ assigns a set of hierarchical privileges $P_{hi} \subseteq P_{osr}$ to each scope $sc \in SC$,
- $scpParams : SC \rightarrow \mathcal{P}(Params)$ is a function assigning a set of parameters $Params \subseteq \{OnCreationDefault, OnGrantDefault, ScopeManager\}$ to every scope $sc \in SC$,
- $OSR_{req} \subset OSR_{all}$ with $\forall osr_y \in OSR_{req} : osr_y \neq osr$ is a set of required context roles.

Finally, OSR_{all} denotes the set of all object-specific roles.

Definition 4.3 shows that an object-specific role may have different sets of privileges associated with different scopes—the key scope and the additional ones. The key scope is crucial. In order to create an object-specific role assignment (cf. Section 4.4), the key scope must be set up properly and, therefore, a guarded object has to be chosen matching this scope (with its guarded object type).

Example 4.4. (Key Scope) The object-specific role *manager* (cf. Fig. 5) features a key scope referencing the guarded object type *recruitment process*. In order to assign the object-specific role *manager*, therefore, a concrete guarded object of type *recruitment process* has to be chosen as reference for the key scope. Moreover, appropriate guarded objects of type *application* may be selected for the additional scope of the given object-specific role.

To enable agents to access actions on referenced guarded objects, every scope is linked to two sets of privileges: the first one contains privileges directly related to the referenced guarded object (realized by scp_{sc}) in Definition 4.3; e.g., this set may contain privileges to read or update the guarded object. By contrast, the second set contains privileges that are hierarchically applicable to the child guarded objects of the referenced one (realized by scp_{hi}).

Example 4.5. (Hierarchical Privileges and Additional Scope) Agents with object-specific role *manager* shall be able to access the guarded objects of types *recruitment process* and *job offer* (cf. Fig. 5), but may only access selected *application* guarded objects shared by a *recruiter* (cf. Section 2) on purpose. Hence, the object-specific role *manager* features the key scope *recruitment process* with privileges regarding guarded objects of types *recruitment process* and *job offer*. Moreover, *manager* has an additional scope with privileges pertaining guarded objects of type *application*.

Every object-specific role $osr_x \in OSR_{all}$ may feature a set of required contextual object-specific roles ($OSR_{req} \subset OSR_{all}$) to ensure that it can be correctly assigned at run time. During the latter, an object-specific role will only be assigned to an agent and a guarded object $go_x \in GO_{all}$, if the agent is already part of an object-specific role assignment referring to an object-specific role $osr_{req} \in OSR_{req}$ and a parental guarded object of go_x (cf. Section 4.4 for details).

Parameters $OnCreationDefault$, $OnGrantDefault$, and $ScopeManager$ are used to allow for automated creations, controlled recommendations, and proper assignments of object-specific roles to agents and guarded objects during run time. The details of how to use parameter *scpParams* are explained in Section 4.4. Finally, creating, updating and removing object-specific roles can be controlled by considering object-specific roles as guarded objects as well. In this way, ORAC additionally supports the controlled and secured adaptations of its own entities.

4.4. *Object-specific Role Assignments*

Tying together agents, organizational entities, object-specific roles, and guarded objects, *object-specific role assignments* play the key role in enabling ORAC (cf. Requirement R01 and R04). To establish a sound understanding, Definition 4.4 formally introduces object-specific role assignments.

Definition 4.4. (Object-specific Role Assignment) An object-specific role assignment is defined as a tuple $osra = (A, OE, osr, GO, scgo)$ where

- $A \subseteq A_{all}$ is a set of agents,
- $OE \subseteq OE_{all}$ is a set of organizational entities,
- $osr = (P, SC, keySC, AddSC, scp_{sc}, scp_{hi}, scpParams, OSR_{req}) \in OSR_{all}$ is a object-specific role,
- $GO \subseteq GO_{all}$ is a set of guarded objects,
- $scgo : SC \rightarrow \mathcal{P}(GO_{all})$ is a function that assigns a set of guarded objects $GO_{sc} \subset GO$ to every role scope $sc \in SC$; while $keySC$ may only be assigned to exactly one $go \in GO_{all}$, every additional scope $sc \in SC, sc \neq keySC$ may be assigned to an arbitrary number of guarded objects $go_1, ..., go_n \in GO_{all}$,

$OSRA_{all}$ denotes the set of all object-specific role assignments.

By incorporating organizational entities, a set of agents can be assigned to an object-specific role assignment instead of being referenced directly. For example, an organization role (e.g., a *director*) may be used to assign agents being directors in an organization. Furthermore, new agents may be added to the organizational role of a *director* without need to adjust the current object-specific role assignments.

Before creating an object-specific role assignment, it needs to be determined which object-specific roles may be assigned to an agent or to organizational entities. As discussed, only an object-specific role with a corresponding key scope or a valid additional scope may be assigned to a given guarded object. By using different scopes (cf. Definitions 4.3 and 4.4), an agent may obtain a range of object-specific roles in relation to the guarded objects in an information system (cf. Fig. 7).

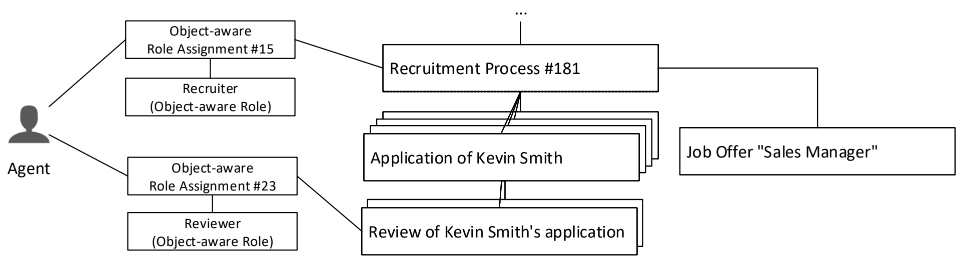
...
Object-aware Role Assignment #15
Recruiter (Object-aware Role)
Recruitment Process #181
Agent
Application of Kevin Smith
Job Offer "Sales Manager"
Object-aware Role Assignment #23
Reviewer (Object-aware Role)
Review of Kevin Smith's application

Fig. 7: Example of an Agent obtaining overlapping Object-Specific Roles

For any guarded object, an agent may want to dynamically switch his role to perform a desired action based on another object-specific role. Note that this aspect is interwoven with the semantics of deliberately performing actions in different roles. As a consequence, the user interface of the information system should allow switching roles during run time (cf. Section 5).

In addition, the scope-related parameters ($scpParams$) of the referenced object-specific role are crucial for creating object-specific role assignments. Accordingly, they need to be carefully set to ensure the desired ORAC functionality at run time. If parameter $OnCreationDefault$ is set for the key scope of an object-specific role, an object-specific role assignment will be automatically created for the agent who intends to create a guarded object (cf. Requirement R07). In this context, the given object-specific role and the guarded object are taken into account. Consequently, if parameter $OnGrantDefault$ is set for any scope, the role will be recommended to the agent who wants to assign it to another agent (and organizational entity respectively).

Finally, parameter $ScopeManager$ indicates that an object-specific role has the maximum number of privileges regarding the defined key scope. Obviously, for every defined scope, there is exactly one object-specific role with parameter $ScopeManager$. To support controlled ORAC, object-specific roles with parameter $ScopeManager$ may be utilized to support a review process regarding the creation or the update of an object-specific role assignment. For example, an agent may want to assign a different object-specific role to another agent. Then, one or several agents, who obtain an object-specific role with parameter $ScopeManager$ being assigned to a parental guarded object, take the decision whether or not the object-specific role assignment can be created.

Besides the parameters of an object-specific role, the manual assignment of an object-specific role to other agents can be controlled as well. First, the creation of an object-specific role assignment in relation to a specific guarded object may be limited by a dedicated privilege. In consequence, only those agents with an object-specific role covering such a privilege may create an object-specific role assignment linking an agent to an object-specific role in relation to a given guarded object. Furthermore, an agent cannot assign any object-specific role with privileges other than the ones the agent obtains with its current object-specific role assignment.

Once an object-specific role assignment is created, it is not supposed to be adjusted at run time. However, if an organizational entity shall be changed, the assignment may be adjusted without causing any problem. Hence, a review process is needed. If an agent shall be added to an object-specific role assignment, the latter may either be adjusted or an entirely new assignment be created. Note that such a decision should follow a consistent policy. As the object-specific role assignment would change its semantics , its referenced guarded object or linked object-specific role must not be updated or replaced. Hence, if an agent wants to perform such a change, a new object-specific role assignment has to be created, whereas the old one needs to be deleted.

5. Realizing Object-specific Role-based Access Control

To underline the benefits provided by ORAC, this section deals with the fundamental aspects of realizing object-specific role-based access control. In particular, we address the necessary steps for setting up ORAC at design time and consider its actual enforcement during run time.

As a prerequisite for ORAC, the object model including the guarded object types and their relationship types need to be defined. Moreover, privileges need to be specified in accordance with Definition 4.2. Based on these preparatory steps, an object-specific role *administrator* to whom all privileges in relation to the information system are assigned needs to be defined. In this context, ORAC benefits from a root guarded object type acting as the top of the emerging object model instance. Considering the HR scenario, for example, an *HR System* guarded object type may be introduced (cf. Fig. 8). Consequently, the key scope of object-specific role *administrator* refers to the guarded object type *HR System* (as *targetGOT*).

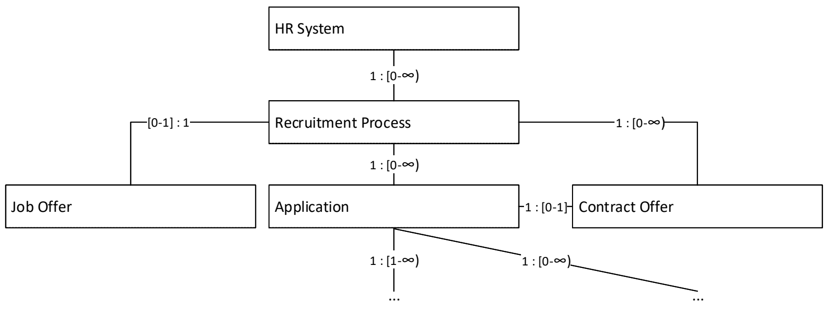

Fig. 8: Use of a Root Guarded Object (HR System)

After booting up an information system with integrated ORAC, an agent with object-specific role *administrator* may first define other object-specific roles. For example, an agent obtaining the object-specific role *administrator* in relation to the guarded object *HR System*, may define the object-specific roles of a *manager*, *recruiter*, and *applicant*. The ORAC-based system may thereby assist the administrator to optimally cover all required privileges, e.g., through the proper visualization of sets of privileges.

Once the object-specific roles have been set up and the information system is running, the access to guarded objects needs to be properly checked. Note that this constitutes a key feature of ORAC, integrating and interpreting all specified components as well as the given object model instance in order to decide whether an agent may access a specific action. For this purpose, the current object-specific role assignment of the agent is determined. Two options exist to accomplish this:

(1) The most specific role assignment is taken automatically. ORAC considers the object-specific role assignments of the guarded object an action may be performed on. If the current agent is referenced by one of these object-specific role assignments, the latter will be further considered. If no object-specific role assignment can be found, the procedure will be repeated for the parental guarded objects of the given one until an object-specific role assignment is found. If no object-specific role assignment can be found at all, an error will be returned indicating that the action cannot be performed.
(2) The agent itself determines the current object-specific role assignment it uses when interacting with the information system (e.g., via user interface). This approach can be further improved as the most specific role assignment is automatically determined and shown to the agent before invoking the action. If there exist several possible object-specific role assignments, the agent may still change the object-specific role assignment depending on preferences.

After determining an object-specific role assignment, the actual access control enabled through ORAC can be enforced based on the identified entities. The latter encompass the given agent, object-specific role assignment, and required privilege (to perform the desired action) as well as the guarded object the action shall be performed on. To facilitate the understanding of realizing ORAC, the algorithm for enforcing ORAC is shown in Listing 1 in pseudo code.

Listing 1: Enforce Object-specific Role-based Access Control

```
boolean enforceORAC(Agent a, OSRA osra, Privilege p, GO go){
 // go through all the scopes of the role assignment
 for (scgo : osra.getSCGO()){
  // current scope references the given guarded object?
  if (scgo.getGO().equals(go)){
   // Is given privilege in set of scope-specific privileges?
   if (scgo.getScope().getScopePrivileges().contain(p))
    return true;
  }
  else {
   // is the go of scgo a (transitive) parent of go?
   boolean isParent = isParentOf(scgo.getGO(), go);
   if (isParent)
    // Is given privilege in set of hierarchical privileges?
    if (scgo.getScope().getHierarchPrivileges().contain(p))
     return true;
  }
 }
 return false;
}
```

The algorithm underlines that it has to be first examined whether the object-specific role assignment directly references the given guarded object. Note that this check has to be performed separately for every defined scope. If such a check is positive, the set of scope-related privileges is evaluated to check whether it contains the required privilege. If the object-specific role assignment does not reference the given guarded object, it has to be evaluated whether the object-specific role assignment references any parental guarded object (*isParentOf()*). In this case, the set of hierarchical privileges will be evaluated with respect to the required privilege.

To cope with changes at run time, ORAC have to feature a high degree of flexibility. For example, new object-specific roles may be added, existing ones be adjusted, or object-specific role assignments be created, changed or removed (cf. Section 4.4). As a major benefit of ORAC, the object-specific role assignments are indirectly updated automatically as soon as privileges are added to or removed from a corresponding object-specific role. This way, privileges can be quickly granted or removed to any number of agents obtaining a particular object-specific role.

Due to the continuous updates of an information system, additional or changed features typically result in new or updated privileges. Therefore, after booting up the information system, the set of privileges and their relationship to object-specific roles have to be re-evaluated to allow for an ORAC evolution (cf. Section 6).

6. Evaluation

To enable the use of the ORAC approach in various application domains as well as to demonstrate its feasibility, we developed a proof-of-concept prototype. The latter is based on state-of-the-art technologies that utilize the human resource application scenario presented in Section 2. To underline the general applicability of ORAC, we further conducted two case studies (cf. Section 6.2) in which we evaluate ORAC with scenarios different from the one presented in Section 2.

6.1. *Proof-of-Concept Prototype*

To demonstrate the feasibility of the ORAC approach and to emphasize the integrability with state-of-the-art concepts and technologies, a proof-of-concept prototype was built based on a service-oriented architecture and the Java EE 7 platform. Its architecture is shown in Fig. 9.

The prototype includes all entities of the ORAC approach as well as the algorithm presented in Section 5. To demonstrate the ORAC functionality, we captured the human resource application scenario presented in Section 2. Accordingly, the ORAC prototype represents a lean human resource recruitment system, which is protected by ORAC, featuring services to, for example, start recruitment processes, add job offers, or update applications. To ease the demonstration of the prototype's capabilities, a specific use case service was implemented allowing for the quick creation of an exemplary recruitment process with many associated objects (e.g., a job offer, applications). The executable source code of the proof-of-concept prototype as well as detailed instructions regarding its usage can be found in a publicly available

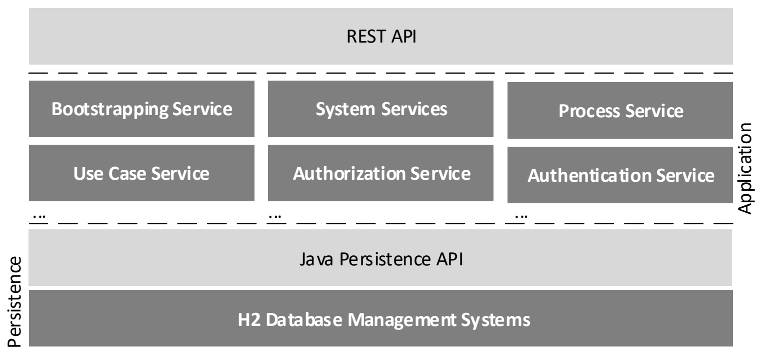

Fig. 9: Architecture of the ORAC Proof-of-Concept Prototype

repository[c] that is free to use.

To properly incorporate guarded object types (cf. Section 2.1) in the domain-specific object model, we made use of the abstract base class *GuardedObject*, which features an abstract method *getParentGuardedObjects*. Based on this approach, it can be ensured that the parental guarded objects of a considered guarded object can be always retrieved and checked for ORAC in a sound way (cf. Section 5). In turn, privileges are specified in accordance with Definition 4.2. To exploit the tight integration of privilege definitions and information system development, we relied on Java *annotations*. In particular, all services in the *application layer* were enriched with the annotation *RequiresAccessControl* as shown in Listing 2.

As advantage, Java reflections may be employed to gather all defined privileges while starting the information system (cf. Requirement R02). Afterwards, the persistence layer is queried to evaluate whether the privileges have been persisted before. If not, all gathered privileges are persisted the first time. Alternatively, existing privileges, which were persisted before, are compared with the gathered ones to derive necessary adaptations (i.e, adding new ones or removing existing ones).

Listing 2: Privilege Definition using Java Annotations

```
@RequiresAccessControl(targetGuardedObjectType=Process.class,
contextGuardedObjectType=Workspace.class, actionType='ADD')
public Process addProcess(Long agentId, Long osraId,
 Long workspaceId, Process newProcess) {
  ...
}
```

The second benefit of annotations comes into play when enforcing access control at run time. Assume that object-specific roles have been created and privileges have already been allocated to them. Then, an agent obtaining an object-specific role through a corresponding role assignment may perform an action. For example,

[c]https://bitbucket.org/dbis/orac-prototype/overview

the agent may want to add a new (recruitment) process (cf. Listing 2). As soon as the call of the client is approaching the application service annotated with the *RequiresAccessControl* annotation, the call is interrupted with a Java interceptor enforcing access control. Figure 10 shows the procedure of enforcing ORAC with the help of an interceptor.

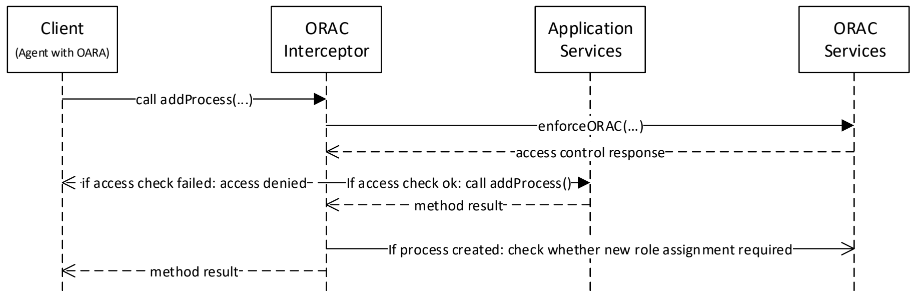

Fig. 10: Enforcing ORAC with an Interceptor

For enforcing ORAC, the interceptor may use the parameters passed to the application service. Regarding the *addProcess* service (cf. Listing 2), the interceptor takes the identifier of the current agent (*agentId*), the identifier of the given object-specific role assignment (*osraId*), and the identifier of the workspace (*workspaceId*) into account to retrieve the *agent*, the *object-specific role assignment* and the *guarded object* of interest (workspace with id *workspaceId*). Using these entities, the algorithm from Listing 1 is executed to decide whether the desired action may be performed. If access is granted and the *addProcess* service is successfully executed afterwards, the interceptor may asynchronously trigger an ORAC service checking whether a new object-specific role assignment needs to be created for the considered agent and the recently created recruitment process (cf. Section 4.4). In consequence, Requirement R07 is properly addressed by the ORAC implementation at run time.

Overall, the developed proof-of-concept prototype demonstrates that the ORAC approach can be well integrated into the design and run time of an information system relying on state-of-the-art technologies. Notably, the presented implementation relies on *convention over configuration* principle: the *RequiresAccessControl* annotations as well as the signature of the specific services need to be coherently defined by developers. The presented interceptor may then easily enforce object-specific role-based access control, which is independent from the specific implementations of the application services. Thereby, an *aspect-oriented* integration of ORAC can be achieved to prevent developers from mixing functional source code with authorization-related one.

6.2. *Case Studies*

We conducted two case studies to demonstrate the applicability of ORAC. For both case studies, we analyzed the application scenarios at hand as well as an existing information system in order to take the current support into account. Thereupon, we investigate the applicability of ORAC regarding the encountered scenarios. In the scope of the first case study, we analyzed auctioning processes[14] and a web-based auctioning system[d]. In the second case study, we analyzed the general particularities of project management[23] and a common open source project management system[e].

Case Study 6.1. (Auctioning) If someone wants to auction off an article in an auctioning system, she typically signs into an auctioning system as a *user* and then creates an *offer* in certain category of the auction system (cf. Fig. 11). When creating the offer, the user is linked to the offer as a *vendor*. Usually, an offer (object) contains various attributes, e.g., starting price or payment terms. In addition, the offer comprises a child object *product description* with various attributes such as the product description or pictures of the product. If a user of the auctioning system is interested in purchasing the offered product, he will make a tender and, consequently, a *bid* object is created in relation to the *offer* object. Furthermore, the user who has given a tender now acts as a *bidder* in relation to the *offer* (object). As soon as auctioning has come to an end, the user who has issued the highest bid will act as a *buyer* in relation to the *offer*.

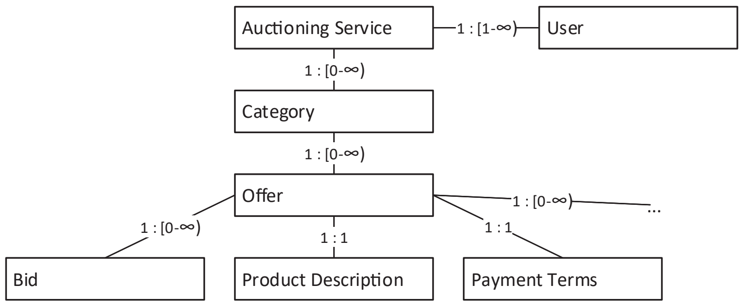

Fig. 11: Case Study Auctioning Service

Studying the auctioning case and its properties in detail shows that ORAC is able to support the application scenario well. Regarding the auctioning system (key scope), at least two object-specific roles are required: *user* and *administrator*. Further, there are three object-specific roles for guarded objects of type *offer* (key scope): *vendor*, *bidder*, and *buyer*. In the role of a *vendor*, users may update their

[d]www.phpprobid.com
[e]www.openproject.com

offer and see the details of every issued bid (hierarchical privileges). In turn, a user assigned as a *bidder* may see some more details of the offer and the details of his bids. Thus, the object-specific role *bidder* requires an additional scope regarding the guarded object type *bid*. Finally, a user assigned as a vendor to an object of type *offer*, may see further details of the offer, e.g, payment terms as well as the bids he issued for the offer (additional scope).

Case Study 6.2. (Project Management) In a project management system (cf. Fig. 12), a project is created and used to increase the awareness and transparency for project stakeholders, to support project managers in coordinating the involved persons more effectively, and to keep track of the project progress in order to ensure the quality of work results. Therefore, a *user* of a project management system may create a project in order to act as a *project admin* afterwards. Using the created project, various subordinated project-related entities (e.g., milestones, phases, or tasks) are added or updated to document and manage the progress of the project. To enable collaboration among project participants, the latter are added by the *project admin* to the project. Then, they may act as *project members* in relation to the (virtual) project. As a result, the *project admin* may assign tasks to project members on demand in order to coordinate work, to document the assignments, and to eventually achieve the objectives together.

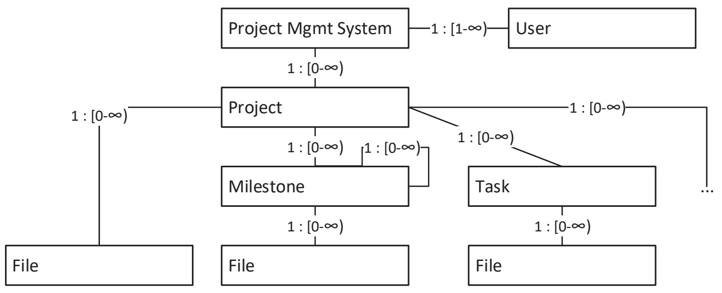

Fig. 12: Case Study Project Management

Analyzing the project management case show that the project management system also employs different roles that can be well supported by ORAC. In particular, the system provides—similar to the auctioning service—the two roles *user* and *administrator* on “system level”. Hence, ORAC can easily match these roles with object-specific ones targeting the project management system. Furthermore, project management employs the project-specific roles *project admin*, *project member*, and *project reader*. Interestingly, the project management system also enables administrators to create roles specific to work packages. Note that these roles feature a scope as well (e.g., objects of type *project*). However, the roles cannot support the hierarchical application of privileges or an *overlapping of roles*. In ORAC, an agent may concurrently obtain both the roles of a system user and a project member. Finally, the definition and use of roles are customized and wired in the project management system, i.e., it does not rely on a generic approach like ORAC.

ORAC is capable of widely supporting both auctioning processes and project management scenarios with their access control requirements. Through object-specific roles and role assignments, users may be accurately given access to the objects they *own* as well as to the corresponding actions they may execute. Moreover, the typical roles, which are required in both case studies, can be well supported by object-specific roles. However, there exist specific requirements which may require system-specific adjustments to the generic ORAC approach. In the auctioning case, for example, an object-specific role assignment (to object-specific role *buyer*) must be created automatically for the user who has issued the highest bid.

7. Related Work

RBAC[25,7] is the most commonly known access control approach. In RBAC, roles are allocated to users and every role is connected with a set of privileges (often called permissions). However, the privileges of traditional RBAC are not made explicit. Instead, for every action there is a list of roles that may execute it. Consequently, in the source code of an information system, it is frequently checked whether the subject, who intends to perform an action, has roles granting the action. As common benefits, RBAC is appreciated for its simplicity and ease of manageability. Furthermore, one can audit which resources and functions are accessible by which roles. As major drawbacks, RBAC lacks a fine-grained access control (cf. Section 1) and, as a result, scalability becomes an issue due to the tremendous number of roles that may have to be managed. In particular, RBAC neither allows for object-/instance-specific access control nor can it deal with dynamically changing environment attributes (e.g., time of day and current location of a subject).

Numerous adaptations of RBAC have been proposed to address these shortcomings. First, an approach providing parameterized privileges and role templates was proposed[9]; a role is limited to only access a subset of objects based on the instantiated parameters. Expanding this basic concept of configuration through parameters, parameterized role-based access control approaches[1,8] aim to overcome the explosion of dedicated roles and to enable a more fine-grained access control. However, the approach suffers from the complex assignment of subjects to the parameterized roles at run time. Furthermore, with increasing number of parameters, the management of parameterized role instances becomes challenging. An approach to use attributes for the automated assignment of users to roles was proposed in literature[2]. Note that in combination with parameterized roles, the integrated management of roles, parameters, attributes, and rules constitutes an even bigger challenge for managing and realizing proper access control policies.

Integrating RBAC with the object-oriented paradigm, the approach of object-oriented role-based access control[4] was proposed to control information flows within object-oriented systems. The key idea is to control information flows among objects, which may be dynamically created or removed. Objects obtain roles that equip them with privileges enabling and limiting the access to other objects. Finally, another approach[17,3] addresses the integration of RBAC in a distributed object-aware process management system to enable fine-grained access control. The approach is limited to five key privileges and focuses on the integration of the objects' attribute values and states with access control decisions.

Overall, the above-mentioned approaches address parts of an object-specific role-based access control, but lack an integrated view on subjects, privileges, object-specific roles, assignments, and objects. ORAC exclusively provides the integration of hierarchical privileges (cf. Requirement R05) and the support of different role scopes (cf. Section 4.3) to optimally support dynamic object model instances.

To address the limitations of RBAC-based approaches, several approaches propose attribute-based access control (ABAC) [20,6,13,10]. With ABAC, no roles have to be defined up-front—at least as long as role names are not used as attributes. Instead, policies are defined based on the attributes of the subjects and the resources to be accessed[13]. Consequently, ABAC enables an expressive, fine-grained access control based on rules evaluating attributes. Listing 3 shows an example of a policy expressed in the terms of the Abbreviated Language for Authorization (ALFA)[f] to specify the access of recruiters to the recruitment process.

Listing 3: Simplified Declaration of an ABAC Policy

```
policy accessRecruiter{
 target clause user.department == HR and user.type == employee
 ...
 rule allowReadUpdateAccessToRecruitmentProcess{
  target clause objectType == "RecruitmentProcess"
  condition recruitmentProcess.assignedRecruiter == user.id
  permit
 }
 rule allowReadUpdateAccessToApplication{
  target clause (action == "read" or action == "update")
   and objectType == "Application"
  condition application.recruitmentProcess.assignedRecruiter
   == user.id
  permit
 }
 ...
}
```

As illustrated in Listing 3, for every object type corresponding policy rules need to be defined in ABAC. In a more complex scenario, each of the potentially large set of policies might comprise a high number of rules to accommodate the attributes in

[f]`https://en.wikipedia.org/wiki/ALFA_(XACML)`

the access control decisions. Furthermore, if a user fills several roles in a information system (e.g., manager and recruiter), the possible roles and their privileges are encoded across rules and with attributes of object types (e.g., *assignedRecruiter*). To define the required policies and rules, a detailed understanding of the objects, the attributes and the business processes are needed. As a result, users themselves cannot easily allocate privileges to roles (cf. Requirements R01 and R02).

To properly determine the privileges a specific group of users (i.e., a role) shall obtain, the relevant rules need to be first identified. Moreover, considerable domain knowledge is necessary to correctly analyze the identified rules. However, to determine a subject's set of privileges in respect to a running information system, the full set of access rules needs to be instantiated with subject and object attributes[5]. In comparison, in ORAC, the set of object-specific role assignments may be used to derive the roles an agent obtains in relation to given objects.

To enhance the comparison, the hierarchy of objects in an object model instance cannot be easily incorporated in ABAC. Instead, it must be statically described with the help of object attributes (cf. *condition* of rule *allowReadUpdateAccessToApplication*). For an unlimited hierarchy of objects (e.g., a growing tree structure), it is not possible to adequately specify a rule comparable to the one presented in Listing 3. In ORAC, corresponding privileges are linked to the set of hierarchical privileges of an object-specific role (cf. Requirement R05), and the given object hierarchy is dynamically resolved at run time. ORAC significantly differs from ABAC through the availability and provision of roles (which are important in many information systems), the rather simple assignment of pre-specified privileges to object-specific roles (instead of powerful rules using any kind of attributes), and the integration of the hierarchy of objects into the access control approach.

Finally, there exist various approaches that combine role- with attribute-based access control, e.g., by constraining the set of privileges allocated to a role by the usage of attributes at run time[15]. In particular, constraint rules incorporating current attribute values determine the privileges to be excluded. A formal model constraining role privileges via rules that evaluate attributes was proposed[12]. This *role-centric attribute-based access control* (RABAC) extends RBAC with privilege filtering policies.

8. Conclusion

We presented an approach for realizing an object-specific role-based access control (ORAC), which enables the support of sophisticated scenarios with complex object models. The approach was established by gathering the key use cases and requirements in relation to object-specific access control in contemporary information systems. Based on a solid formal foundation, ORAC allows for the rich modeling of object-specific roles by allocating object-related as well as hierarchical privileges at both design and run time. Especially, this privilege allocation enables users to dynamically access created objects without any changes to be applied to ORAC itself. To enforce access control at run time, the creation of object-specific role assignments can be controlled by configuration parameters and safety checks (is an agent allowed to assign a role). The proof-of-concept implementation demonstrated the feasibility of ORAC and emphasized the benefits of integrating ORAC based on state-of-the-art technologies (annotations, interceptors). Finally, to evaluate the practical applicability of ORAC, it was applied in two application scenarios in the shape of case studies.

In future research, we will perform experiments using the prototype to evaluate the scalability of ORAC in larger scenarios. In this context, we will examine the impact of caching and indexing guarded object instances in order to optimize the process of retrieving parental object instances at run time.

References

1. A. E. Abdallah and E. J. Khayat. A Formal Model for Parameterized Role-Based Access Control. In *Formal Aspects in Security and Trust*, pages 233–246. Springer, 2005.
2. M. A. Al-Kahtani and R. Sandhu. A Model for Attribute-Based User-Role Assignment. In *18th Annual Computer Security Applications Conference*, pages 353–362, 2002.
3. K. Andrews, S. Steinau, and M. Reichert. Enabling Fine-grained Access Control in Flexible Distributed Object-aware Process Management Systems. In *21st IEEE International Enterprise Distributed Object Computing Conference (EDOC 2017)*. Quebec City, Canada, 2017.
4. S.-C. Chou. Embedding role-based access control model in object-oriented systems to protect privacy. *Journal of Systems and Software*, 71(1-2):143–161, 2004.
5. E. Coyne and T. R. Weil. ABAC and RBAC: Scalable, Flexible, and Auditable Access Management. *IT Professional*, 15(3):14–16, 2013.
6. N. Dan, S. Hua-Ji, C. Yuan, and G. Jia-Hu. Attribute Based Access Control (ABAC)-Based Cross-Domain Access Control in Service-Oriented Architecture (SOA). In *2012 International Conference on Computer Science and Service System (CSSS)*, pages 1405–1408, 2012.
7. D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST Standard for Role-based Access Control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.
8. J. Fischer, D. Marino, R. Majumdar, and T. Millstein. Fine-Grained Access Control with Object-Sensitive Roles. In *European Conference on Object-Oriented Programming*, pages 173–194, 2009.
9. L. Giuri and P. Iglio. Role templates for content-based access control. In *Proceedings of the second ACM workshop on Role-based access control*, pages 153–159, 1997.
10. V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo. Attribute-Based Access Control. *Computer*, 48(2):85–88, 2015.
11. J. Huang, D. M. Nicol, R. Bobba, and J. H. Huh. A Framework Integrating Attribute-based Policies into Role-based Access Control. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 187–196, New York, NY, USA, 2012. ACM.
12. X. Jin, R. Krishnan, and R. Sandhu. A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 41–55, 2012.
13. X. Jin, R. Sandhu, and R. Krishnan. RABAC: Role-Centric Attribute-based

Access Control. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 84–96, 2012.

14. V. Krishna. *Auction Theory.* Academic Press/Elsevier, Burlington, MA, 2nd edition, 2010.
15. D. R. Kuhn, E. J. Coyne, and T. R. Weil. Adding Attributes to Role-Based Access Control. *IEEE Computer*, 43(6):79–81, 2010.
16. V. Künzle. *Object-Aware Process Management.* Dissertation, Ulm University, Ulm, 2013.
17. V. Künzle and M. Reichert. Integrating Users in Object-aware Process Management Systems: Issues and Challenges. In *Proceedings BPM'09 Workshops, 5th Int. Workshop on Business Process Design (BPD'09)*, pages 29–41, 2009.
18. V. Künzle and M. Reichert. Striving for Object-aware Process Support: How Existing Approaches Fit Together. In *1st Int'l Symposium on Data-driven Process Discovery and Analysis (SIMPDA'11)*, 2011.
19. V. Künzle, B. Weber, and M. Reichert. Object-aware Business Processes: Fundamental Requirements and their Support in Existing Approaches. *International Journal of Information System Modeling and Design (IJISMD)*, 2(2):19–46, 2011.
20. B. Lang, I. Foster, F. Siebenlist, R. Ananthakrishnan, and T. Freeman. A Flexible Attribute Based Access Control Method for Grid Computing. *Journal of Grid Computing*, 7(2):169–180, 2009.
21. R. Lenz and M. Reichert. IT support for healthcare processes – premises, challenges, perspectives. *Data & Knowledge Engineering*, 61(1):39–58, 2007.
22. D. Müller, M. Reichert, and J. Herbst. Data-Driven Modeling and Coordination of Large Process Structures. In *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, volume 4803, pages 131–149. 2007.
23. PMI. *A Guide to the Project Management Body of Knowledge.* PMI Project Management Institute, Newtown Square, Penn, 2000.
24. R. Pryss, N. Mundbrod, D. Langer, and M. Reichert. Supporting Medical Ward Rounds through Mobile Task and Process Management. *Information Systems and e-Business Management*, 13(1):107–146, 2015.
25. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based Access Control Models. *Computer*, 29(2):38–47, 1996.
26. R. S. Sandhu and P. Samarati. Access Control: Principle and Practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.
27. H. Shen and P. Dewan. Access Control for Collaborative Environments. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 51–58, 1992.
28. S. Steinau, K. Andrews, and M. Reichert. The Relational Process Structure. In *Advanced Information Systems Engineering*, pages 53–67, 2018.
29. J. Tiedeken, M. Reichert, and J. Herbst. On the Integration of Electrical/Electronic Product Data in the Automotive Domain. *Datenbank Spektrum*, 13(3):189–199, 2013.
30. J. Wainer, P. Barthelmess, and A. Kumar. W-RBAC—A Workflow Security Model Incorporating Controlled Overriding of Constraints. *International Journal of Cooperative Information Systems*, 12(04):455–485, 2003.