Enabling Fine-grained Access Control in Flexible Distributed Object-aware Process Management Systems

Kevin Andrews, Sebastian Steinau, and Manfred Reichert
Institute of Databases and Information Systems
Ulm University, Germany
Email: {firstname.lastname}@uni-ulm.de

***Abstract*—To increase flexibility, object-aware process management systems enable data-driven process execution and dynamic generation of form-based tasks at run-time. Therefore, a powerful access control concept becomes necessary to define which data elements users may read or write at a given point in time during process execution. The access control concept we present in this paper has been realized in the context of the PHILharmonicFlows framework, which provides a distributed data-driven process execution engine. We present solutions that allow for complex as well as fine-grained permissions and roles, which are granted depending on the states of processes and data elements. We show how one can resolve authorization queries in real-time over multiple business objects and process instances. This constitutes a significant advantage over centralized access control systems.**

***Index Terms*—access control, authorization, permissions, roles, process management, scalability, PHILharmonicFlows**

I. Introduction

When dealing with the management of human-centric business processes one of the greatest challenges concerns flexibility. Traditional process management systems, which are based on the activity-centric process management paradigm, allow for the definition of activities and the order in which these activities must be executed such that the overall process may complete. In particular, the flexibility of these systems depends on the structure of the design-time process model. For example, a process model that contains many loops and alternate execution paths is inherently and trivially more flexible than a linear process. This allows process participants to handle exceptions, errors, and special use-cases as part of the normal process flow, instead of forcing them to work around the limitations of the process model by executing work "outside" the process management system. As a drawback, however, process designers must foresee all possible alternate execution paths when creating a process model. Obviously, this is no simple task, which becomes evident when studying the large amount of research devoted to increasing the flexibility of process execution in cases where the process model is insufficiently prepared for exceptional control flow. Most research focuses on allowing the control flow to be manipulated directly by process participants, for example by inserting control flow elements, such as activities and transitions, at run-time [1].

However, these approaches do not cover all possible scenarios, as, for example, generating entirely new forms or other activities at run-time is a very cumbersome task. Part of the problem is that, in most current process management systems, the permissions to read or write data elements are set rigidly per activity, making it impossible to automatically generate additional forms at run-time based on permissions, as these simply do not exist outside the context of a specific activity. Additionally, users that belong to a certain role have all permissions granted by that role at all times, i.e., finely-grained access to individual data elements, depending on factors such as process state, is impossible. Therefore, as automatic form generation is not possible, process participants must be able to manually introduce new forms into running process instances. To facilitate this, they must define which data elements should be read- or writable in the new form and by whom, making such a feature very complex. This is especially problematic when considering that one of the goals of process management systems is to hide details, such as data elements, from process participants. In summary, this means that most research into flexibility is only useful for cases where the control flow needs to be adapted and not enhanced with new activities being added to the control flow.

Another, entirely different, approach to the problem of flexibility, is to not structure processes along preexisting activities (e.g. forms), as it is done in the activity-centric paradigm, but around the business data. There exist many approaches that, in some sense, follow this idea [2]–[5]. They enable flexibility based on global permissions that allow process participants to read and write the individual data elements , i.e., users may interact with such data-centric process management systems at run-time based on automatically generated forms. The latter, in turn, can be generated by examining the role a user has and the permissions this role gives him in relation to a certain data element. Most approaches further add a state, either to the data elements or to the entire process, which may change during the execution of a process instance. Utilizing the state concept, permissions can be defined more precisely, granting them not only based on a role, but also on the state of individual data elements or the entire process. As data-centric approaches to process management require permissions that allow users to interact with generated forms, access control in some form is, trivially, a requirement for these approaches to actually function in real-world scenarios.

Expanding on these basic ideas, PHILharmonicFlows, an object-aware process management framework currently under development at Ulm University, enables fine-grained access control in order to be able to automatically generate forms for the respective users at run-time. As PHILharmonicFlows is object-aware, several data elements are aggregated into business objects, each representing an entity that the process relies on, such as a *checking account*, *customer*, or *transfer*. Moreover, the access control system of PHILharmonicFlows is very flexible, even allowing permissions to be granted based on the relations that individual objects have to each other. A simple example of this could be that the role *checking account manager* may only read the *amount* data element of *transfer* objects related to *checking account* objects, which are, in turn, related to *customer* objects assigned to the *checking account manager* in question.

Note that this constitutes a significant improvement compared to the rather rudimentary access control systems used by other data-centric approaches. This conceptual access control approach constitutes the first contribution of this paper. Furthermore, as PHILharmonicFlows is being implemented as a distributed and scalable process management system, the access control concept presented in this paper utilizes the possibilities offered by the distributed PHILharmonicFlows architecture to enable fast real-time resolution of roles and permissions across a cluster of computers. These additional considerations are the second contribution of this paper.

As object-aware process management itself, without the access control system, is already far from being trivial, the approach, as well as its current implementation as a microservice based process engine, is discussed in Section II. The requirements for an access control system, which must not only function in a completely new kind of process management system, but also in a distributed environment, is presented in Section III. Section IV then presents our solutions to the challenges these requirements create, addressing both design-time and run-time issues. Finally, Sections V and VI offer a discussion of related work as well as a summary and an outlook on future work.

II. Fundamentals

A. Object-aware Process Management

PHILharmonicFlows, the object-aware process management framework we are using as a test-bed for the concepts presented in this paper, has been under development for many years at Ulm University [6]–[11]. This section gives an overview of the PHILharmonicFlows concepts necessary to understand the remainder of the paper. PHILharmonicFlows takes the basic idea of a data-driven and data-centric process management system and improves it by introducing the concept of *objects*. One such digital object exists for each business object present in a real-world business process. As can be seen in Fig. 1, a PHILharmonicFlows object consists of data, in the form of *attributes*, and a state-based process model describing the object *lifecycle*.

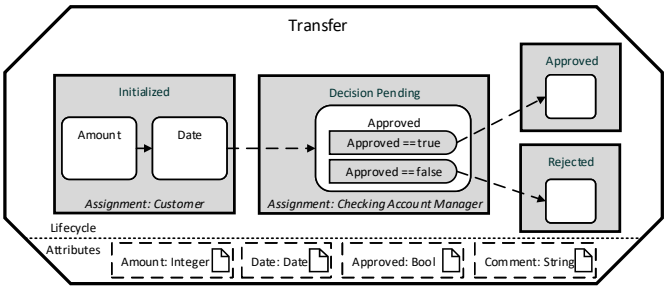

Fig. 1. Example PHILharmonicFlows Object

The attributes encapsulated in the *Transfer* object (cf. Fig. 1) are *Amount*, *Date*, *Approval*, and *Comment*. The *lifecycle process*, in turn, describes the different *states (Initialized*, *Decision Pending*, *Approved*, and *Rejected)*, an instance of a *Transfer* object may have during process execution. Each state, in turn, contains one or more *steps*, each referencing one of the object attributes. The steps are connected by *transitions*, allowing them to be arranged into a sequence. When a transition between two steps from different states is activated at run-time, the state of the object changes. Finally, PHILharmonicFlows supports alternative paths in the form of *decision steps* containing *predicate steps.* An example of these can be seen in the *Approved* decision step in the *Decision Pending* state.

In summary, as PHILharmonicFlows is data-driven, the lifecycle process for the *Transfer* object can be understood as follows: The initial state of a *Transfer* object is *Initialized*. Once a *Customer* has entered data for the *Amount* and *Date* attributes, the state changes to *Decision Pending*, which allows a *Checking Account Manager* to input data for *Approved*. Based on the value for *Approved*, the state of the *Transfer* changes to *Approved* or *Rejected*.

A single object, however, is only part of a complete PHILharmonicFlows process. To allow for complex, executable processes, many different objects and users may have to be involved [10]. It is noteworthy that users are simply special objects in the object-aware process management concept. The lifecycle processes present in the various objects are executable concurrently at run-time, thereby improving performance. The entire set of objects (including users) present in a PHILharmonicFlows process is denoted as the *data model*, an example of which can be seen in Fig. 2.

The data model contains all objects participating in a process as well as the *relations* existing between them. A relation constitutes a logical association between two objects, e.g., a relation between a *Transfer* and a *Checking Account*. Such a relation can be instantiated at run-time between two concrete object instances of types *Transfer* and *Checking Account*, thereby associating the two object instances with each other. The resulting meta information, i.e., the information that the *Transfer* in question belongs to a certain *Checking Account*, can be used to coordinate the processing of the two objects with each other.

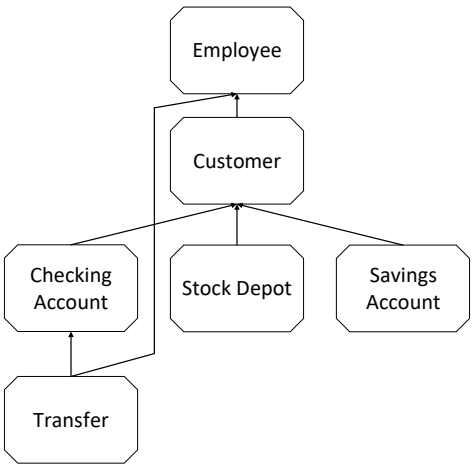

Fig. 2. Example PHILharmonicFlows Data Model

Finally, complex object coordination, which becomes necessary as most processes consist of numerous interacting business objects, is possible in PHILharmonicFlows as well [10]. As objects publicly advertise their state information, the current state of an object can be utilized to coordinate with other objects, corresponding to the same business process, through a set of constraints, defined in a separate *coordination process*. As a simple example, consider a constraint stating that a *Transfer* may only change its state to *Approved* if there are less than 4 other *Transfers* already in the *Approved* state for one specific *Checking Account*.

The various components of PHILharmonicFlows, i.e., objects, relations, and coordination processes, are implemented as separate microservices, turning PHILharmonicFlows into a fully distributed process management system. For each object instance, relation instance, or coordination process instance present at run-time, one microservice is spawned. The individual microservices communicate with each other, exactly mirroring the conceptual ideas of PHILharmonicFlows presented in this section. Each microservice only holds data representing the attributes of its object. Furthermore, the microservice only executes the lifecycle process of the object it is assigned to. The only information visible outside the individual microservices is the current "state" of the object, which is, in turn, used by the microservice representing the coordination process to properly coordinate the objects' interactions with each other.

As the actual implementation architecture of PHILharmonicFlows is close to its core conceptual ideas, the implementation of additional concepts, such as access control, can be realized closely to their conceptual ideas as well. As a flip-side, access control, especially when permissions concern multiple objects and therefore multiple running microservices, is far from being trivial and must take additional factors and requirements into consideration compared to a more traditional engine implementation.

B. Role-based Access Control

As the access control concept presented in this paper relies on the basic concepts of Role-Based Access Control (RBAC), this section offers a quick overview of RBAC [12]. The goal of RBAC is to only allow users to access and edit information which need for completing their tasks. Furthermore, RBAC offers an improvement over earlier access control concepts as it removes unnecessary administrative overhead. Fig. 3 gives an abstract overview of the elements the basic RBAC concept offers.

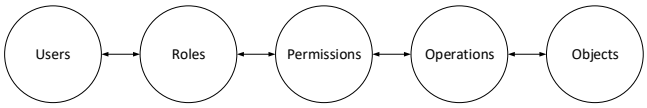

Fig. 3. RBAC elements

- *Objects* are business objects that can be interacted with, e.g., data or functions.
- *Users* are the individual process participants that wish to interact with the *objects*.
- *Operations* are the various ways in which users may interact with an *object*, e.g., writing a data element or executing a function.
- *Roles* allow *users* to be grouped logically, easing the administrative overhead of managing access control.
- *Permissions* allow mapping *operations* on *objects* to *roles*, i.e., they control the access *users* have to *object operations*.

Example 1 illustrates the interactions between the different RBAC elements and concepts in a typical real-world process scenario.

Example 1. User *Employee1* needs to perform operation *edit balance* on *CheckingAccount1*. Permission *p1* associates the *edit balance* operation of *checking account* objects to the role *checking account manager*. Therefore, *Employee1* needs to be authorized to activate the *checking account manager* role, allowing him to perform the *edit balance* operation on *CheckingAccount1*.

Example 1 refers to the concept of role activation, an important part of RBAC. In RBAC, users are not simply assigned to a role and have all permissions belonging to that role at all times. Instead, users are statically *assigned* roles, which they may *activate*, if they are *authorized* to do so. This addition of only activating roles when users are authorized allows for greater permission assignment flexibility in systems using RBAC. The authorization to activate roles depends on authorization constraints which can be defined when assigning roles. The exact nature of these constraints depends on the information system implementing RBAC, as well as the concrete use-cases present. While most systems hide the actual activation of roles from users in the interest of usability, in some it is necessary to prevent users from activating roles with conflicting permissions.

III. Requirements

Before delving into the details of our solution to access control in a distributed, object-aware process management system, we present fundamental requirements with respect to complex and fine-grained access control. In the initial research into object-aware process management systems, numerous requirements were identified for an access control system which utilizes the advantages of the object-aware paradigm [6], [8]. These requirements have since been extended and partially revamped as the concept of object-aware process management was developed into PHILharmonicFlows: a fully distributed object-aware process management system. As fine-grained access control is essential for offering dynamically generated forms to users at run-time, the capabilities of the system were extended considerably over time. Finally, as the implementation of PHILharmonicFlows is based on microservices, the requirements were extended even further to take the challenges presented by the distributed microservice-based architecture into account.

The main elements of any RBAC system are present in PHILharmonicFlows as well, i.e., *users*, *roles*, *permissions*, *operations*, and *objects* (cf. Section II-B). As roles provide a mapping between users and their permissions, they are required to reduce the administrative efforts for managing permission assignments. Note that, without roles, each new user would have to be assigned each permission separately. Using roles, one can statically associate users and permissions without any complicated n:m mapping. However, this is also not an ideal system, as new users or users whose roles may have to be changed must be managed by some form of administrative entity. In larger corporations this can be a cumbersome task, as there is a constant influx of new employees and existing ones switch to different jobs, requiring different permissions. Note that this constitutes a challenge in all sorts of information systems, not just process management systems. A more sophisticated approach, which we aim to achieve with PHILharmonicFlows, is to dynamically activate various roles for authorized users at run-time, in line with RBAC notions of *role authorization* and *role activation*, explained in Section II-B. As this cannot be done "magically", we need to leverage parts of the PHILharmonicFlows concept to this end, leading us to Requirement 1.

Requirement 1 (Dynamic Authorization of Permissions and Roles)**.** The access control system should use the conceptual elements of object-aware process management for dynamically authorizing roles and permissions at run-time.

As every user is represented by an object, the factor determining whether or not a user is authorized to activate a role must somehow be decided by differences in the instantiated objects at run-time. Basically, there are only two factors that distinguish two objects of the same type at run-time. On the one hand, there are the values of the attributes, which, together with the lifecycle process, determine the object state. On the other, there are the relations an object has to other objects. As users are represented in terms of objects, these distinguishing factors apply to them as well. The first of these two factors leads to Requirement 2.

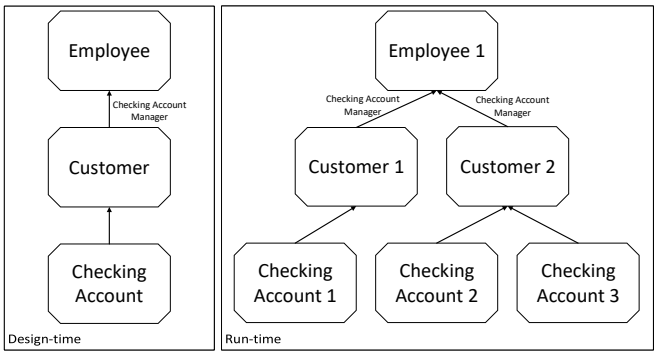

Fig. 4. Roles Derived from Relations at Design-time and Run-time

Requirement 2 (Role Authorization Depending on Data)**.** Role authorization conditions should be definable on the attribute values of the object representing the user.

Example 2. An *Employee* user is authorized to activate the *Checking Account Manager* role if his *Department* attribute has the value "Account Management".

When examining the second distinguishing factor between objects, concerning the relation of an object or user to other objects, an opportunity to increase the expressiveness of the role system in PHILharmonicFlows is presented. If roles can be authorized dynamically based on relations, this allows roles to be activated in regards to specific other objects, leading to Requirement 3.

Requirement 3 (Role Authorization Depending on Relations)**.** Role authorization conditions should be definable on the relations of the object representing the respective user to other objects. These roles should be specific to the related objects granting them.

Example 3. An *Employee* user is authorized to activate the *Checking Account Manager* role in regards to a specific *Checking Account* object as he is related to a *Customer* user who, in turn, is related to the *Checking Account* object.

To clarify Example 3, Fig. 4 shows the structure of the data model at design-time on the left, as well as the individual object instances created at run-time, including the relation instances that exist between them, on the right. The *Checking Account Manager* role is defined on the relation between *Employee* and *Customer* at design-time. At run-time, an *Employee* related to a *Customer* is authorized as the *Checking Account Manager* for that *Customer*. Moreover, he may activate the role in regards to all *Checking Accounts* related to the *Customer*.

Additionally to users and roles, an access control system needs permissions. The latter are indispensable, as without them there is no reason to maintain roles, or even the concept of users, in an information system. Usually, however permis-

sions are merely the means to ensure that inexperienced or malicious users are only able to use functions or edit data they rely on in order to complete their tasks. Note that in object-aware process management permissions are central to process execution, as they determine the exact structure of the dynamically generated forms the users interact with at run-time. For example, a user who may write a certain attribute value, is presented with an input field for that attribute when he views the form for its object at run-time.

Obviously, this increases complexity compared to more traditional (i.e. activity-centric) process management systems, where permissions are defined per activity and not for each data attribute separately. However, it does offer the advantage of allowing fully dynamic form generation, completely eliminating the need for creating forms and associated data mappings at design-time. In PHILharmonicFlows this is facilitated by an object's lifecycle process, that dictates which attributes have to be filled out before the object may switch to the next state, as well as the read/write permissions, that allow users with certain roles to fill out additional, mostly optional attributes. Together, this results in a personalized and dynamically created form, which is then displayed to the respective user at run-time. An example of such a form, derived from the lifecycle process of the object displayed in Fig. 1 and a write permission for the *Comment* attribute, is shown in Fig. 5.

Obviously, read/write permissions may depend on the current state of an object, otherwise a person with a specific activated role and, therefore, the permissions belonging to that role, would always interact with exactly the same form when viewing an object at run-time. In contrast to activity-centric process management systems, where granting permissions based on the state of an entire process instance is commonplace, as role assignments are per-task, the permissions offered in PHILharmonicFlows have to be more fine-grained and granted depending on the state of individual objects. To this end, we extend the RBAC concepts of *role assignment*, *role authorization*, and *role activation* with the notions of *permission assignment*, *permission authorization,* and *permission activation*. In a nutshell, this means that permissions are not simply "granted" to all users that have an active role containing the permission. Instead, the permissions

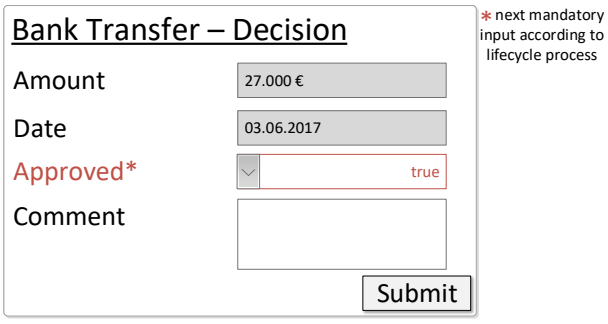

Fig. 5. Example PHILharmonicFlows Form

themselves have authorization constraints allowing the access control system to allow or deny their activation. One possible constraint, i.e., the state of the object a permission grants an operation on, is formulated in Requirement 4.

Requirement 4 (Permission Authorization Depending on Object State)**.** Permissions authorization conditions should be definable on the current state of an object.

Example 4. An *Employee* with an active *Checking Account Manager* role in respect to a *Transfer* object may activate the permission to write a *Comment* attribute when the *Transfer* is in the *Decision* state.

An object-aware process management system, which fulfills access control Requirements 2, 3, and 4, already turns out to be very flexible and can be used to model numerous access control scenarios. However, a common scenario is not covered yet: the ability to authorize the activation of permissions based on attribute values of affected objects. Note that this is extremely useful when assigning tasks to different process participants using a distribution key. This is common in many information systems, for example issue/bug tracking software with tickets. The task assignment distribution key in an issue tracking software could be, for example, the system or software affected by the issue or the issue severity. Both factors can be used by the issue tracking software to determine the correct process participant an issue ticket shall be assigned to. In essence, this means that the authorization to activate the permission to access and solve the ticket is granted dynamically, based on attributes of the ticket in question. In a more generic concept, such as object-aware process management, this can be seen as granting permissions based on data values.

Requirement 5 (Permission Authorization Depending on Data)**.** Permissions authorization conditions should be definable on the current attribute values an object has.

Example 5. An *Employee* with an active *Checking Account Manager* role in respect to a *Transfer* object may only activate the permission to edit the form for the *Decision Pending* state if the value of the *Amount* attribute is less than 50.000 €, otherwise an *Employee* with an active *Supervisor* role is authorized to activate the permission to complete the *Decision Pending* form instead.

As we aim to fulfill all these requirements with PHILharmonicFlows, the question remains as to how they are realizable in a fully distributed computing environment. Considering that each object, whether it represents a user or a business object, can potentially be located on a different physical node of a cluster, thanks to the microservice architecture, the authorization of most permission involves at least two objects. On the one hand, the object the permission concerns must be involved to ensure that the permission is authorized, on the other, the user object must be involved to ensure that the user has an active role containing the permission in question.

As both the role and the permission authorization can be dynamically granted or revoked at any point in time during

process execution, based on data, object states, or relations between objects, the queries on both ends have to be run in real-time for every permission or role authorization query. Furthermore, this real-time solution needs to scale well with increasing numbers of users, as the generation of forms at run-time relies on the the access control system. This leads us to Requirement 6.

Requirement 6 (Scalable Real-Time Permission and Role Authorization)**.** Permission and role authorization should be determinable in real-time, without sacrificing scalability.

Example 6. A *Customer* may edit a pending *Transfer* and raise the *Amount* to over 50.000 € (cf. Example 5), meaning that an *Employee* with an active *Checking Account Manager* Role is no longer authorized to activate the permission to approve the transfer. This dynamic change of permissions should become immediately visible to process participants, i.e., there is no time window in which the access control system grants outdated permissions.

As the authorization conditions for roles and permissions shall based on attribute values (cf. Requirements 2 and 5), additional challenges present themselves at run-time. As Requirement 6 states, using real-time resolution of permission and role authorization queries is necessary to ensure that changing attribute values are immediately reflected when resolving authorization conditions. Additionally, the commonly used strategy of caching the results of such queries to improve performance is not applicable in this scenario. In particular, the use of cached authorization query results could result in role or permission activations which should have been prohibited instead. Additionally, caching could lead to the inverse problem of denying role or permission authorization when it should have been granted. In information systems where roles and permissions can not change based on as many factors as in an object-aware process management system, caching might be acceptable. However, in order to utilize the strengths of the object-aware paradigm to its fullest, we must find other strategies to cope with the run-time complexity of the access control system.

Requirement 7 (Adequate Performance without Caching Results)**.** Permission and role authorization should not utilize the strategy of caching results to improve performance, instead ensuring adequate real-time performance through other means, in support of Requirement 6.

This section gave an overview of the most important requirements for an access control system we identified in the context of the object-aware process management approach. Obviously, these are not all requirements relevant to an access control system in general. However, we believe that the presented ones are the most challenging and fitting for a process management system, especially one which is data-driven and object-aware, such as PHILharmonicFlows. This paper focuses on the issues that are academically challenging and interesting from a conceptual standpoint and not on actual security concerns of the access control system, such as communication security, password management and identity verification. However, our concepts and prototypical implementations are extendable to include such security-oriented provisions in various ways, as described in [13].

IV. Enabling Fine-grained Access Control

This section presents the concepts and architecture we developed to fulfill the requirements that we identified for a flexible, distributed access control system integrated into an object-aware process management system. The microservice based architecture, we selected for implementing PHILharmonicFlows, allows us to handle incoming access control requests independently of one another, however, it also adds complexity to some of the concepts. To fulfill the requirements from Section III, various aspects must be considered, for both for the design- and the actual run-time of PHILharmonicFlows processes.

A. Design-time aspects

As explained in Section III, two major factors determine whether or not a particular user has the permission to perform a certain action at run-time: the roles the user has currently *activated* and whether the permissions belonging to these roles are *authorized* for the object and operation in question. The currently implemented permissions focus on allowing process modelers to shape the dynamically generated forms for the different roles present in a given real-world process. An overview of selected permissions is given in Table I. An affiliation between two conceptual elements is denoted by subscript.

Permission Assignment: Obviously, every permission, no matter for which operation it is granted, needs to be statically assigned to a role *r*. Only users that have *activated* role *r* are considered when resolving *authorization* for a permission

TABLE I
Basic PHILharmonicFlows Permissions

Operation	Parameters	Description
Read Attribute	Role r Object o Attribute a_o State s_o Condition c_p	Allows users with active role r to read the value of attribute a_o while o is in state s_o and the permission authorization condition c_p is true.
Write Attribute	Role r Object o Attribute a_o State s_o Condition c_p	Allows users with active role r to write the value of attribute a_o while o is in state s_o and the permission authorization condition c_p is true.
Execute State	Role r Object o State s_o Condition c_p	Allows users with active role r to open the form for s_o while o is in state s_o and the permission authorization condition c_p is true.
Change State	Role r Object o Transition t_o State s_o Condition c_p	Allows users with active role r to change the object state using transition t_o while o is in state s_o and the permission authorization condition c_p is true.
Instantiate Object	Role r Object o	Allows users with active role r to create an instance of object o.

p_r. The permission authorization is resolved at run-time and determines whether a user may activate permission p_r. This depends on the permission authorization condition c_p, which, in turn, can be set to any expression that is based on attributes of object o, e.g., [*Amount* $>$ 50.000].

These permission authorization conditions, which exist for all operations applied to objects that are already instantiated (i.e., all operations except for *instantiate object*) become necessary to satisfy Requirement 5. As the permission authorization conditions are defined depending on attribute values, they enable such scenarios as presented in Example 5. Furthermore, all permissions that support permission authorization conditions have parameters referring to an object o as well as a state s_o belonging to object o. This allows process modelers to limit permission assignment to objects that are in a specific state. Note that such constraints become necessary to meet Requirement 4, i.e., permission authorization at run-time depending on the current state of an object. This allows for far more flexible permission assignment when compared to systems that are based on the state of the entire process instance.

Role Assignment: As each permission *p* is assigned to a role *r* at design-time, only users who are authorized to activate *r* at may activate p_r. Determining the users assigned to *r*, and, hence, the users to be considered for role authorization at run-time, is therefore essential. The role assignment is done statically at design-time, while role authorization is resolved at run-time, similar to the above presented concept for permissions.

In general, roles are assignable to all user objects present in a PHILharmonicFlows data model at design-time. This means that for a user object (e.g. *Employee*) a set of roles may be statically assigned. In line with the general RBAC concept, the actual role authorization and activation are, however, run-time concerns. A role assigned to a user object this ways is denoted as a *global role,* as permissions attached to these roles potentially apply to all object instances of a certain type at run-time. Additionally, to fulfill Requirement 2, a global role *r* may have a role authorization condition c_r, which limits role authorization at run-time to those users whose attribute values fulfill c_r.

However, in order to fulfill Requirement 3, not all roles can be simply assigned to the various user objects present in a PHILharmonicFlows data model. Requirement 3 mandates that role authorization at run-time must be resolvable based on the relations an object representing the respective user has to other objects. We tackle the related challenge by allowing roles to be attached not only to user objects, but also relations, at design-time. To be more precise, roles may be attached to a relation between a user and another object, thereby only assigning the role to the user in respect to objects attached along that relation. At run-time an instance of that relation must exist between the user and the target object in order for role *authorization* to occur for the user. Therefore, we denote a role attached to a relation as a *relation role*. A relation role may also have a role authorization condition c_r, limiting role authorization at run-time depending on user attribute values. Note that this allows for even greater flexibility when modeling a process. Furthermore, it ensures that a large number of authorization scenarios can be covered by this concept. An example of one such relation role is shown in Fig. 4.

B. Run-time aspects

Having explained how the static role and permission *assignment* is handled at design-time in PHILharmonicFlows, we now present the more complex aspects of how role and permission *authorization* as well as *activation* are managed. Considering that our access control concept extends the classic RBAC model of role assignment, authorization, and activation with permission assignment, authorization and activation (cf. Section III), we have run-time resolution workloads not only on the role side, but on the permission side as well. These are necessary, as we not only have to check whether a user is authorized to activate a certain role, but also if he is authorized to activate the permission for the operation he wishes to perform.

At first glance, this might seem to be disadvantageous, as not only role authorization, but also permission authorization need to be checked at run-time. However, the increase in access control flexibility and realizable scenarios is necessary to enable fine-grained access to object attributes and, therefore, the generation of personalized user forms at run-time. Furthermore, we show that the performance hit is reduced significantly by the microservice architecture we utilize for the implementation. As it should be now clear that every access control scenario in PHILharmonicFlows concerns a role authorization as well as a permission authorization at run-time, Example 1 can be modified to more precisely illustrate these two procedures.

Example 7. *Employee1* needs to perform the operation *Edit Balance* on *CheckingAccount1*. Permission *p1* associates the *write attribute "Balance"* operation of *Checking Account* objects to the relation role *rr1*, which is defined on the relation between the *Employee* objects and *Customer* objects. Relation role *rr1* corresponds to the *checking account manager* role from Example 1. Furthermore, *p1* has a permission authorization condition c_{p1} of [*SecurityLevel* == 0], limiting the activation of the permission to *Checking Account* objects whose *SecurityLevel* attribute has the value 0. Finally, the relation role *rr1* has a role authorization condition c_{rr1} of [*Department* == "*AccountManagement*"], limiting the activation to *Employee* users whose *Department* attribute has the value "AccountManagement". In summary, this means that *Employee1* needs to be authorized to activate the role *rr1*, and permission *p1* needs to be authorized to be activated for *CheckingAccount1*.

Distributed Approach: The separation of the entire process logic and data into the individual object instances at run-time [11] allows spreading the large amounts of requests to the access control system that occur during normal process execution among the various object instances, making the access control

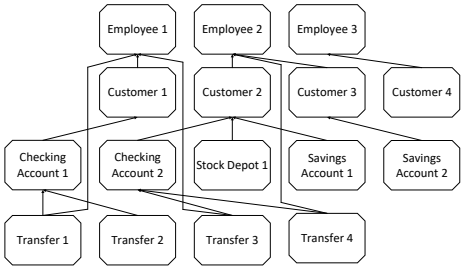

Fig. 6. PHILharmonicFlows Object Instances at Run-time

system fully distributed. As the microservice implementation leverages these conceptual opportunities, the resolution of role and permission authorization can be distributed among the microservices hosting the process instance at run-time. Note that this is highly scalable, compared to a classic centralized database approach with one or more tables holding role and permission information.

As each of the object instances displayed in Fig. 6 has its own lifecycle process and attributes, they are independent of each other, except for the relations existing between them. The implementation of the object and relation instances as microservices, that only communicate with each other over well-defined message interfaces, allows us to replicate the conceptual elements exactly in the implementation architecture. Assuming that the object and relation instances displayed in Fig. 6 are part of a currently running PHILharmonicFlows process instance, we can use them to show how we resolve both role and permission authorization.

Role and Permission Descriptors: In Example 7, *Employee1* needs to perform the *edit balance* operation on *CheckingAccount1*. Therefore, the access control system has to resolve whether *Employee1* has a permission he may activate to complete the *edit balance* operation. The generated form for *CheckingAccount1* can then either display or hide the operation to *Employee1*. Assuming that *edit balance* is a *write attribute* operation on the *balance* attribute, permission *p1*, which is assigned to relation role *rr1*, is defined using the following permission descriptor

$$p1 \begin{cases} type & WriteAttribute \\ r & rr1 \\ o & CheckingAccount \\ a_o & Balance \\ s_o & Opened \\ c_{p1} & [SecurityLevel = 0] \end{cases}$$

The relation role *rr1*, in turn, is defined by the following role descriptor:

$$rr1 \begin{cases} relation & CustomerToEmployee \\ name & CheckingAccountManager \\ perms & [p1, p2, p5, ...] \\ c_{rr1} & Department = \text{“}AccountManagement\text{”} \end{cases}$$

Obviously, *p1* is not the only permission defined in the data model, just as *rr1* is not the only role defined in the data model. In any real-world process there are many roles and permissions defined for various scenarios, which necessitates a strategy for finding role-permission combinations that are assigned to the user requesting access. As Section IV-A explained, the role and permission *assignments* are static, as opposed to role and permission *authorizations*. We use this fact to optimize the way we spread role and permission information across the microservices at run-time in order to reduce unnecessary communication overhead. We replicate the static permission assignment information, such as the permission descriptor for *p1* shown above, to all user instances present in the process instance. This way, the information which permissions can be authorized, is available to all microservices representing user instances. As every byte of information that is locally available to a microservice, i.e., is present in-memory at run-time, must be stored redundantly for every microservice, reducing memory consumption is necessary in microservice based architectures. To facilitate this, we analyze which roles are *assignable* to a given user during object instantiation. The goal hereby is to only replicate information concerning permissions attached to roles that are assigned to the user in question.

Authorization Queries: The interfaces we define in the user microservices are very simple, as shown by the following example for the write permission:

```
bool hasWritePerm(long objInstId, int attrId, int stateId)
```

As each user microservice has one such interface for every permission, queries can be directed to the respective user microservice when generating a form. An example of the entire resolution procedure is shown in Fig. 7.

If there is no permission-role combination allowing an *Employee* user to write the *Balance* attribute on *Checking Account* objects, which are in the *Opened* state, this can be detected in-memory, by an an *Employee* microservice. This is possible without communicating with other microservices, as all necessary information for a negative answer is present in all *Employee* user instances. Alternatively, a microservice may find a write permission in its memory that matches the *attributeId* and *stateId* passed to the *hasWritePermission* interface (1). Searching for a matching permission requires an *O(n)* search over the list of permissions contained in the object (2). If a permission is found, the role authorization for corresponding role must be resolved (3).

Role Authorization: The role authorization is always checked before the permission authorization, as role authorization always depends on data locally available to the microservice, whereas the permission authorization condition always depends on attribute values of the object instance referenced by the *objectInstanceId* parameter. This can reduce communication overhead as, in Example 7, if the role authorization fails, the *Employee1* microservice does not have to contact the *CheckingAccount1* microservice to request the current value of the *SecurityLevel* attribute.

The role authorization itself can be checked entirely without communicating with other microservices, as each microservice

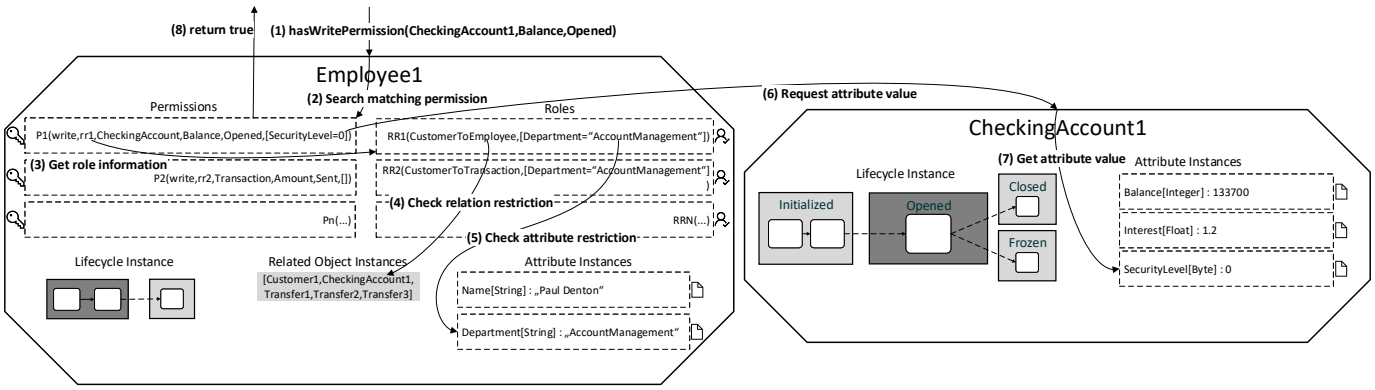

Fig. 7. Access Control Query Processing

representing a user instance at run-time has the following information locally available:

- role descriptors for all assigned roles,
- current attribute values of all attributes,
- list of object instance ids connected via relations.

The static role descriptors are transferred to the user microservice along with the permission descriptors during instantiation, whereas the attribute values are always available as the microservice for a user, just as for any other object, manages the data storage for all locally available attribute values. Finally, the list of related objects is updated with every instantiated or deleted relation at run-time, to avoid unnecessary traversals of the entire object graph to determine if a relationship to another object exits. Utilizing this locally available information, a microservice can determine whether the user it represents is authorized to activate the role for a given permission without any communication overhead, simply by searching common map data structures. For global roles, this merely involves checking the current attribute values for the attribute referred in the role authorization condition. Additionally, relation roles require a contains-search for the object id passed to the permission interface in the list of related object instances. In particular, for role *rr1* from Example 7, this means searching for the object id *CheckingAccount1* in the related object instances list (4). This ensures that the relation required by the relation role descriptor exists. If it does, a search for the the *Department* attribute is conducted and its current value is compared to "AccountManagement" (5). As the attributes and related objects are stored in map data structures, the search complexity for both these operations is *O(1)*.

Permission Authorization: Once role authorization has been determined, the final step a user microservice has to complete is to determine permission authorization. As the latter depends on a permission authorization condition such as c_{p1}, permission authorization requires communication with *CheckingAccount1* to determine the attribute value of *SecurityLevel*. In turn, this requires communication between *Employee1* and *CheckingAccount1* (6), as well as an *O(1)* search for the *SecurityLevel* attribute, conducted in the *CheckingAccount1* microservice (7). Finally the, *Employee1* microservice returns the result of the permission query to the caller (8).

In summary, through the optimizations we apply to the conceptual access control system of PHILharmonicFlows, we have been able to realize a scalable real-time access control solution, thereby fulfilling Requirement 6. Currently, we are working on further optimizations, such as eliminating the *O(n)* search necessary for finding a matching permission assignment at the start of each access control query (cf. Fig. 7). We aim at finding a data structure that allows us to improve search times to *O(1),* while keeping memory consumption acceptable, alternatively we propose using techniques developed for databases, such as indexes or binary search sorting.

Our claim of having created a scalable access control system which is well integrated with the PHILharmonicFlows concept is supported by the fact that the query processing, as shown in Fig. 7, can run on as many microservices spread across a cluster as there are users represented by those microservices. This is enabled by our microservice based architecture, implemented using the Microsoft Service Fabric Reliable Actors Framework. Additionally, we can use the reliable actors framework to deal with the single bottleneck scenario we have identified: multiple user objects resolving access control queries that involve permissions with permission authorization conditions requiring concurrent access to the same object instance. In this scenario, multiple queries would have to wait for concurrent access to an attribute value of the same object instance and, therefore, microservice. However, we can alleviate this bottleneck using read-only replicas of the microservice in question, a feature of the reliable actors framework.

V. Related Work

As an access control system is mandatory for most information systems, there exist numerous works on the topic, most of which concern some form of RBAC-based system [12]. As this paper focuses on access control in process management systems, and specifically in object-aware process management systems, we choose to exclude related work on access control in other information systems.

[14] presents an access control system for activity-centric workflow management systems, which relies on predicate-based access to data. Additionally, [14] offers a concept for process instance-based roles, which we address for object-aware process management with relation roles (cf. Section IV-B). Furthermore, [14] identified the need for permission authorization based on the data context of a business object, akin to Requirement 5. However, the solutions presented in [14] fall short in the granularity aspect, as the actual assignment of permissions is done per activity.

The work presented in [15] introduces the "conflicting entities" paradigm for supporting separation of duties in ways standard RBAC cannot. This extension to RBAC allows for the specification of constraints to ensure that users can not have conflicting permissions or roles activated simultaneously in process environments. The run-time implementation can then check for conflicting role or permission assignments, and force users to choose one of the conflicting options. This would be an interesting extension to our access control system in the PHILharmonicFlows concept and will be taken into consideration in the future.

[16] presents a language to express both static and dynamic authorization constraints. These notions are very similar to the general RBAC notions of role assignment and role authorization, and are presented in a formalized manner in the paper. The authors propose precomputing the static (i.e., assignment) constraints and merely evaluating the dynamic (i.e., authorization) constraints at run-time. The paper offers good theoretical and formal groundwork for an access control system, however, performance and flexibility of the solution are not analyzed in detail.

Finally, [17] describes a formal framework for an aspect that we have not yet covered in our current research, adapting the access control system to changes in organizational structure. The authors introduce a formal framework for the controlled evolution of organizational models and related access control constraints. In PHILharmonicFlows, we plan on covering this aspect when tackling ad-hoc changes and schema evolution challenges in the context of object-aware data models, as the organizational structure in object-aware process management is an integral part of the data models themselves.

VI. Summary and Outlook

The access control system presented in this paper is optimized to be as scalable and flexible as possible, supporting a multitude of access control scenarios, while still ensuring that there are no bottlenecks present. To achieve this, we leveraged not only the conceptual possibilities offered by the object-aware process management approach, but also a fully distributed implementation, built using microservices for execution in cloud-based compute clusters.

Our intent for the future is to show that both are viable, i.e., that the object-aware approach is applicable to many real-world scenarios and that the implementation of the core components, such as the access control system, are better and more scalable than existing solutions.

As the basic PHILharmonicFlows framework is conceptually complete, we are currently working on details, such as access control or ad-hoc changes to running processes instances. Additionally, we are developing test scenarios for large scale performance evaluations using the cloud. Up until now, we have examined most performance and scalability factors from an architectural and mathematical standpoint. However, we intend to fully evaluate the scalability of the implemented engine empirically in future research.

References

[1] M. Reichert and B. Weber, *Enabling flexibility in process-aware information systems: challenges, methods, technologies*. Springer Science & Business Media, 2012.

[2] N. Haddar, M. Tmar, and F. Gargouri, "A data-centric approach to manage business processes," *Computing*, vol. 98, no. 4, pp. 375–406, 2016.

[3] R. Hull, E. Damaggio, F. Fournier, M. Gupta, F. T. Heath III, S. Hobson, M. Linehan, S. Maradugu, A. Nigam, P. Sukaviriya *et al.*, "Introducing the guard-stage-milestone approach for specifying business entity lifecycles," in *International Workshop on Web Services and Formal Methods*. Springer, 2010, pp. 1–24.

[4] W. M. Van der Aalst, M. Weske, and D. Grünbauer, "Case handling: a new paradigm for business process support," *Data & Knowledge Engineering*, vol. 53, no. 2, pp. 129–162, 2005.

[5] D. Cohn and R. Hull, "Business artifacts: A data-centric approach to modeling business operations and processes," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 32, no. 3, pp. 3–9, 2009.

[6] V. Künzle and M. Reichert, "Integrating users in object-aware process management systems: Issues and challenges," in *International Conference on Business Process Management*. Springer, 2009, pp. 29–41.

[7] V. Künzle and M. Reichert, "Philharmonicflows: towards a framework for object-aware process management," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 4, pp. 205–244, 2011.

[8] 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)*, vol. 2, no. 2, pp. 19–46, April 2011.

[9] V. Künzle, "Object-aware process management," Ph.D. dissertation, University of Ulm, 2013.

[10] S. Steinau, V. Künzle, K. Andrews, and M. Reichert, "Coordinating business processes using semantic relationships," in *IEEE 19th Conference on Business Informatics (CBI)*, 2017.

[11] K. Andrews, S. Steinau, and M. Reichert, "Towards hyperscale process management," in *Proceedings of the 8th International Workshop on Enterprise Modeling and Information Systems Architectures (EMISA)*, 2017.

[12] D. Ferraiolo, J. Cugini, and D. R. Kuhn, "Role-based access control (rbac): Features and motivations," in *Proceedings of 11th annual computer security application conference*, 1995, pp. 241–48.

[13] M. Swanson and B. Guttman, *Generally accepted principles and practices for securing information technology systems*. National Institute of Standards and Technology, Technology Administration, US Department of Commerce, 1996.

[14] S. Wu, A. Sheth, J. Miller, and Z. Luo, "Authorization and access control of application data in workflow systems," *Journal of Intelligent Information Systems*, vol. 18, no. 1, pp. 71–94, 2002.

[15] R. A. Botha and J. H. P. Eloff, "Separation of duties for access control enforcement in workflow environments," *IBM Systems Journal*, vol. 40, no. 3, pp. 666–682, 2001.

[16] E. Bertino, E. Ferrari, and V. Atluri, "The specification and enforcement of authorization constraints in workflow management systems," *ACM Transactions on Information and System Security (TISSEC)*, vol. 2, no. 1, pp. 65–104, 1999.

[17] S. Rinderle and M. Reichert, "A formal framework for adaptive access control models," in *Journal on data semantics IX*. Springer, 2007, pp. 82–112.