PHILharmonicFlows: Towards a Framework for Object-aware Process Management

Vera Künzle[1,2] and Manfred Reichert[1]
[1]Institute of Databases and Information Systems, Ulm University, Germany
{vera.kuenzle,manfred.reichert}@uni-ulm.de
[2]Persis GmbH, Heidenheim, Germany

Abstract. Companies increasingly adopt process management systems (PrMS) which offer promising perspectives for more flexible and efficient process execution. However, there still exist many processes in practice which are not adequately supported by contemporary PrMS. We believe that a major reason for this deficiency stems from the unsatisfactory integration of processes and data in existing PrMS. Despite emerging approaches which address this integration, a unified and comprehensive understanding of object-awareness in connection with process management is still missing. To remedy this deficiency, we extensively analyzed various processes from different domains which are not adequately supported by existing PrMS. As a major insight we learned that in many cases comprehensive process support requires *object-awareness*. In particular, process support has to consider object behavior as well as object interactions, and should therefore be based on two levels of granularity. Besides this, object-awareness requires data-driven process execution and integrated access to processes and data. This paper presents basic properties of object-aware processes as well as fundamental requirements for their operational support. It further introduces our PHILharmonicFlows framework which addresses these requirements and enables object-aware process management in a comprehensive manner. Finally, we evaluate this framework along several process scenarios. We believe that a holistic approach integrating data, processes and users offers promising perspectives in order to overcome the numerous limitations of contemporary PrMS.

Keywords: Process-aware Information Systems, Object-aware Process Management, Data-driven Process Execution

1. Introduction

Aligning information systems in a process-oriented way constitutes a challenging task for any enterprise [8,32]. In this context process management systems (PrMS) offer promising perspectives for the comprehensive support of the process lifecycle [33]. Typically, the architecture of PrMS-based information systems comprises four basic *building blocks* that relate to data, functions, processes and users (cf. Fig. 1a). While database management systems enable the separation of data management from function logic, PrMS foster the separation of process logic from function logic (i.e., application code). For this purpose, PrMS provide generic functions for modeling, executing and monitoring processes. This separation of concerns is a well established principle in computer science in order to increase maintenance and to reduce implementation costs [9].
When using existing PrMS a business process is typically defined in terms of a set of activities representing the business functions and a set of ordering constraints defining their execution sequence [2]. During runtime a process-oriented view (e.g., worklists) is provided to end-users. What is done during activity execution, however, is out of the control of the PrMS. Consequently, most PrMS consider an activity as a black-box in which application data is managed by invoked application components (except routing data and process variables). In general, whether or not an activity becomes activated during runtime depends on the state of preceding activities.

1.1 Problem

Traditional PrMS have been primarily designed for supporting highly structured, repetitive business processes [18]. Various other processes, however, are not adequately supported by these PrMS [1,4,19,26,29]. For example, [26] characterizes the latter as *unstructured* or *semi-structured processes,* which are *knowledge-intensive* and *driven by user decisions.* Other authors, in turn, argue that the business functions to be integrated with these processes cannot be *straight-jacketed into activities* [4,29].

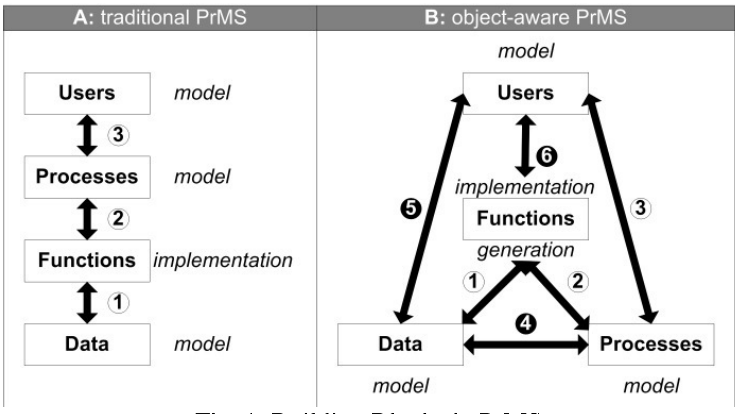


Fig. 1: Building Blocks in PrMS

Generally, we believe that a more systematic and comprehensive analysis of these processes and their properties contributes to a better understanding of the deficiencies of contemporary PrMS. In the PHILharmonicFlows project we conducted such an analysis and studied processes from different areas including human resource management [13,14] and paper reviewing. Basically, we believe that the identified limitations of existing PrMS can be traced back to the unsatisfactory integration of the different building blocks (cf. Fig. 1b).

In particular, our process analyses revealed that advanced process support necessitates *object-awareness* in many cases; i.e., business processes and business objects cannot be treated independently from each other. This observation has been confirmed by other work as well [4,6,10,16,19,21], though a holistic support as well as a comprehensive understanding of the implications of object-awareness is still missing. In this context it is not sufficient to look at different perspectives separately; instead, we must also understand the inherent relationships that exist between the different building blocks and the aspects they cover. This poses the major characteristics for process and data integration:

1. The behavior of the involved objects must be taken into account during process execution.
2. Interactions between objects must be considered.
3. Since the progress of a process mainly depends on available object instances and on their attribute values, process execution has to be accomplished in a data-driven manner.
4. Authorized users must be able to access and manage process-related objects at any point in time.

Adequate support of object-aware processes must consider all elements of the underlying data structure, which comprises objects, object attributes and object relations. Besides this, as we will show, comprehensive support of object-aware processes will entail other pivotal features like the ability to define and execute activities at different levels of granularity or advanced capabilities for user integration and authorization.

Though there exist several approaches that target at a tighter integration of business processes and business data [4,6,10,16,19,20,21,22,23,31], none of them supports all four characteristics mentioned above in an integrated and comprehensive way. In particular, there are only few approaches which consider process execution issues as well.

1.2 Contribution

This paper first discusses fundamental properties of processes that require a tighter integration of business data, business processes, business functions, and users. In the following we denote these processes as *object-aware*. The described properties and requirements significantly extend the work we presented in [13,14]. While in [13,14] we focussed on the basic challenges for integrating processes, data and users, this paper explicitly describes the properties of object-aware processes in detail and elicits major requirements for their effective support. Overall, we believe that more profound research on object-aware processes, including a systematic analysis of their properties, will contribute to overcome some of the fundamental limitations known from existing PrMS. The main part of the paper introduces our PHILharmonicFlows[1] framework for object-aware process management. In particular, this framework includes components for both the modeling and the execution of object-aware processes. Opposed to existing approaches, which only focus on individual characteristics, PHILharmonicFlows provides a comprehensive approach. In particular, we combine *object behavior based on states* with *data-driven process execution*. Further, we enable *process coordination* taking the relations between the involved object instances into account. In this context, coordination is not only possible along direct object relations (e.g., a review directly refers to a paper). Additionally, the processing of object instances can be coordinated based on their inter-relationships within the overall data structure (i.e., taking transitive as well as transverse relationships between object instances into account). Finally, at runtime, *integrated access to business processes, business functions and business data* is enabled. For this purpose, PHILharmonicFlows provides generic functions for automatically creating end-user components like worklists, form-based activities andoverview tables containing relevant object instances.

The remainder of this paper is organized as follows: Section 2 presents the research method we applied. The properties of object-aware processes are introduced in Section 3; they are underpinned by a detailed literature study in Section 4. Following this, fundamental requirements are elicited in Section 5. Section 6 then introduces the PHILharmonicFlows framework for object-aware process management, which addresses these requirements. In Section 7, we describe a proof-of-concept realization and give insights into some of the lessons learned when applying it to real-world processes. We conclude with a summary and outlook in Section 7.

2. Research Methodology

The overall goal of this paper is to provide a *framework for object-aware process management* in order to overcome some of the fundamental limitations of contemporary PrMS. Such a framework needs to provide both components for process modeling and process execution. This section summarizes the research methodology we applied for designing this framework (cf. Fig. 2).

Starting with the basic observation that there exist processes which are not adequately supported by existing PrMS, we defined the following research questions:

[1] Process, Humans and Information Linkage for harmonic Business Flows

Research Question 1:
What are the common properties of these processes?
Research Question 2:
Which requirements must be fulfilled by a PrMS in order to adequately enable these properties?
Research Question 3:
How to support these requirements in an integrated process support framework?

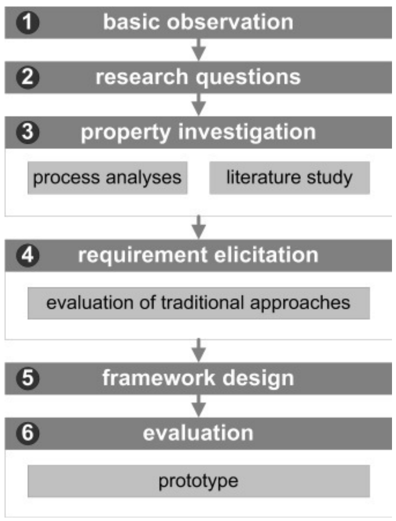


Fig. 2: Applied research methodology

Property investigation
We based property investigation (cf. Research Question 1) on two pillars: First, we identified and analyzed business processes that are not adequately supported by current PrMS. Second, we backed up our findings with an extensive literature study.

Process analyses
- *Data Source:* Due to the limitations of existing PrMS many business applications (e.g., ERP systems) do not rely on PrMS, but contain hard-coded process logic instead. In order to guarantee that the processes we analyzed in the context of our property investigation are not "self-made" examples, but constitute real-world processes with high practical relevance, we evaluated the processes as implemented in existing application software. Amongst others, we analyzed the processes as implemented in the human resource management system *Persis* and the reviewing system *Easychair* [13,14]. However, our evaluation was not restricted to the inspection of user interfaces solely. In addition, one of the authors gathered extensive practical experiences as software developer of the Persis system; i.e., we have deep insights into the application code of this system as well as the implemented processes. Finally, we reinforced our results by additionally talking to system users as well as system consultants.
- *Selection Criteria:* We evaluated the processes (and additional features) based on the main building blocks of existing PrMS which comprise processes, data, functions, and users (cf. Fig. 1a). In particular, we focussed on the interdependencies that exist between these building blocks. Finally, we restricted ourselves to properties relating to process modeling, execution and monitoring.

Literature study
- *Ensuring importance:* We complemented our process analysis by an extensive literature study. This way we were able to show that other authors also consider the identified properties as relevant.
- *Ensuring completeness:* In order to not exclude important properties, we compared our analysis results with existing literature. Though we were able to identify additional properties (e.g., relating to process change and process evolution), they did not directly relate to process modeling, execution and monitoring. Therefore, we omit them in the context of this paper.
- *Ensuring generalisation:* During our literature study, we identified several approaches that target at a tight integration of processes and data. Interestingly, some authors referred to similar application examples as we do, while addressing different properties. Based on these insights we contrasted the different application examples with the total set of identified properties. This way, we were able to demonstrate two things: first, the properties are related to each other. Second, broad support for them is required by a variety of processes from different application domains.

Requirements Elicitation
In order to elicit the basic requirements for enabling the support of object-aware processes, we compared the identified properties with the main characteristics of traditional PrMS [15]. More precisely, we evaluated which properties cannot be directly supported when applying traditional imperative, declarative and data-driven approaches [15]. Though the identified requirements are not complete in the sense that they cover all aspects one can think of, their fulfilment is indispensable for enabling the fundamental properties in respect to the modeling and execution of object-aware processes.

Evaluation
In order to evaluate our framework we have developed a proof-of-concept prototype for the modeling as well as the runtime environment of PHILharmonicFlows. We applied this prototype to a real-world case. We additionally evaluated the developed concepts along other process scenarios (i.e., order handling, house building and vacation requests) which were different from the ones we considered in the context of our process analyses. Finally, we elaborated the benefits of our approach when applying it to these processes as well as lessons learned.

3. Properties of Object-aware Processes

We first introduce a characteristic example of an *object-aware process*. As illustrated in Fig. 3, we consider a (simplified) scenario for recruiting people as known from human resource management. Along this scenario, we describe fundamental properties of object-aware processes, which we gathered during our process analyses (cf. Section 2). We categorize them along the main building blocks of existing PrMS; i.e., users, processes, functions, and data. In order to discuss why these building blocks cannot be treated independently from each other, we focus on the major relationships between them.

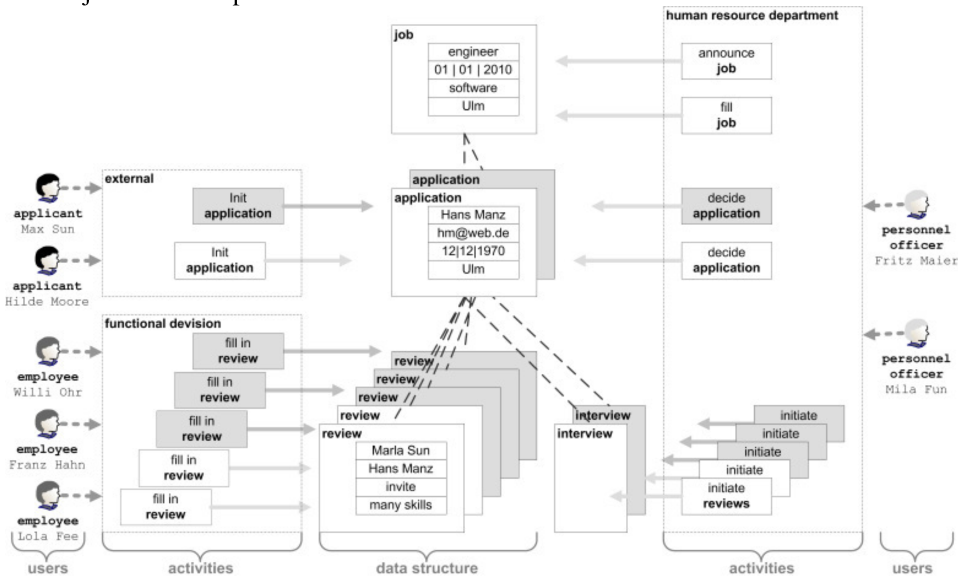


Fig. 3: Example of a recruitment process from the human resource domain

Recruitment process: In the context of recruitment, `applicants` may apply for `job vacancies` via an Internet online form. Before an `applicant` may send her `application` to the respective company, specific information (e.g., `name`, `e-mail address`, `birthday`, `residence`) must be provided. Once the `application` has been submitted, the responsible `personnel officer` in the human resource department is notified. The overall process goal is to decide which `applicant` shall get the job. Since many `applicants` may apply for a `vacancy`, usually, different `personnel officers` are involved in handling the `applications`.
If an `application` is ineligible the `applicant` is immediately rejected. Otherwise, `personnel officers` may request internal `reviews` for each `applicant`. Depending on the concerned `functional divisions`, the concrete number of `reviews` may differ from `application` to `application`. Corresponding `review` forms have to be filled by `employees` from `functional divisions` until a certain `deadline`. `Employees` may either *refuse* or *accept* the requested `review`. In the former case, they must provide a `reason`. Otherwise, they make a `proposal` on how to proceed; i.e., they indicate whether the applicant shall be `invited` for an interview or be `rejected`. In the former case an additional `appraisal` is needed.
After the `employee` has filled the `review` form she submits it back to the `personnel officer`. In the meanwhile, additional `applications` might have arrived; i.e., `reviews` relating to the same or to different applications may be requested or submitted at different points in time. In this context, the `personnel officer` may flag the `reviews` he already evaluated. The processing of the `application` proceeds while corresponding `reviews` are created; e.g., the `personnel officer` may check the `CV` and study the `cover letter` of the `application`. Based on the incoming `reviews` he makes his `decision` on the `application` or initiates further steps (e.g., `interviews` or additional `reviews`). Further, he does not have to wait for the arrival of all `reviews`; e.g., if a particular `employee` suggests `hiring` the `applicant` he can immediately follow this recommendation.

In the following we discuss basic properties of object-aware business processes along this realistic example.

3.1 Properties relating to data

As illustrated in Fig. 4a, data is managed based on *object types* which are *related* to each other. Each object type comprises a set of *attributes*. Object types, their attributes, and their inter-relations form a *data structure*. At runtime the different object types comprise a varying number of inter-related *object instances*, whereby the concrete instance number can be restricted by lower and upper bounds (i.e., *cardinalities*). Furthermore, object instances of the same object type may differ in both their *attribute values* and their *inter-relations* (cf. Fig. 4b); e.g., for one `application` two `reviews` and for another one three `reviews` might be requested. In the following, we denote an object instance which is directly or transitively referenced by another one as *higher-level* object instance; e.g., an `application` is a higher-level object instance of a set of `reviews` (cf. Fig. 4). By contrast, an object instance which directly or transitively references another object instance is denoted as *lower-level* object instance; e.g., `reviews` are lower-level object instances of an `application` object (cf. Fig. 4).

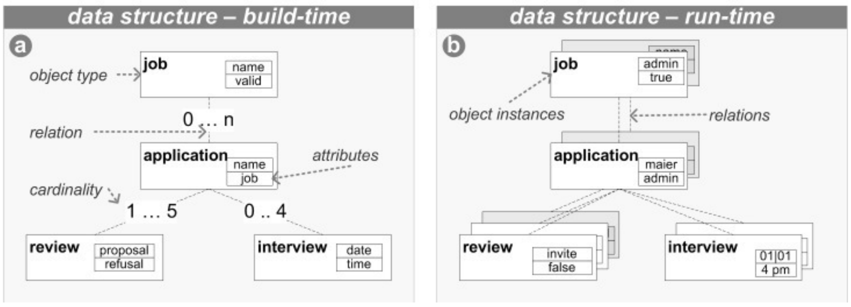


Fig. 4: Data structure at build- and runtime

Relationship between data and users. In order to access data at any point in time, user roles are associated with permissions to create and delete object instances as well as to read/write their attribute values.

Relationship between data and activities. During the execution of activities, object instances may be created, changed or deleted. In particular, object instances are changed by updating corresponding object attribute values.

Relationship between data and processes. In order to reach a certain process goal and to proceed with the execution of a process, usually, certain object instances with specific attribute values must be available. Consequently, certain data is mandatorily required during process execution and the progress of a process instance should be based on attribute changes. Undesired updates of these attribute values, however, have to be prevented after reaching certain states. For example, after an `employee` from a `functional division` has submitted his `review`, she is no longer allowed to change the value of attribute `recommendation`. For this reason, data authorization cannot be handled independently from process execution; i.e., data authorization for a particular object instance needs to consider the progress of its corresponding process instance.

3.2 Properties relating to processes

The modeling of processes and data constitute two sides of the same coin and therefore should correspond to each other. In accordance to data modeling (cf. Fig. 4), therefore, the modeling and execution of processes is based on two levels of granularity: *object behavior* and *object interactions*.

Relationship between process and data. As illustrated in Fig. 5c, the first level of process granularity concerns the *behavior* of object instances. It is expressed in terms of states and the transitions between them; i.e., for each object type a separate process definition representing an *object-life cycle* exists. In addition, object behavior determines in which order object attributes have to be (mandatorily) written, and what valid attribute settings are. Consequently, each state postulates specific attribute values to be set. Generally, a state can be expressed in terms of a particular data condition referring to a number of attributes of the respective object type. As example consider object type `review` and its states as depicted in Fig. 5. In state `accepted` any value for attribute `appraisal` is required and the value of attribute `proposal` must either be 'reject' or 'invite'.

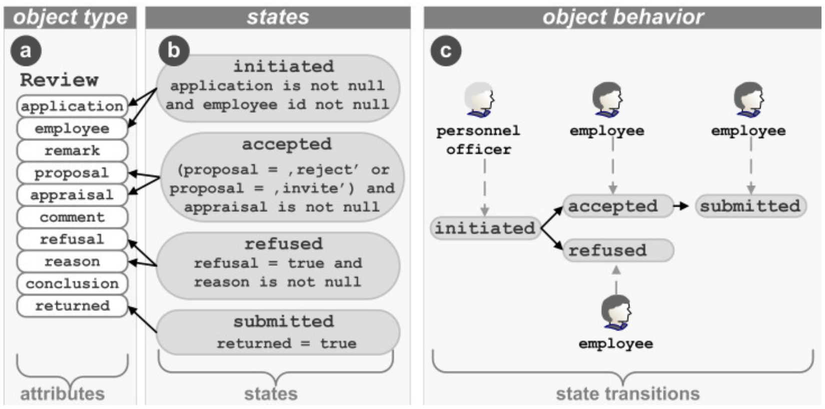


Fig. 5: Object behavior defined based on states and transitions

At runtime, the creation of a process instance is directly coupled with the creation of its corresponding object instance. Furthermore, a particular state of an object instance is reached as soon as the corresponding data condition evaluates to true. Generally, for each object type multiple object instances may exist (cf. Fig. 6a). These object instances may be created or deleted at arbitrary points in time; i.e., the corresponding *data structure* dynamically evolves depending on the type and number of created object instances as well as on their relations. Consequently, individual object instances may be in different states at a certain point in time. For example, several `reviews` might have been requested for a particular `applicant`. While one of them might be in state `initiated`, others might have already reached state submitted. Taking the behavior of individual object instances into account, we obtain a complex *process structure* in correspondence to the given data structure (cf. Fig. 6b).

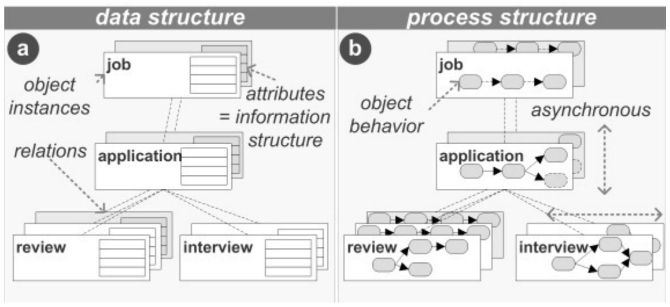


Fig. 6: Data structure and corresponding process structure

The second level of process granularity comprises the interactions that take place between the instances of different object types. More precisely, whether or not a particular process instance may proceed also depends on the progress of other process instances:

Object interactions within the recruitment process (cf. Fig. 7): A personnel officer announces a job. Following this, applicants may init applications for this job. After submitting an application the personnel officer requests internal reviews for it. If an employee acting as referee proposes to invite the applicant the personnel officer will conduct an interview. Based on the results of reviews and interviews the personnel officer decides about in the application. Finally, in case of acceptance the applicant is hired.

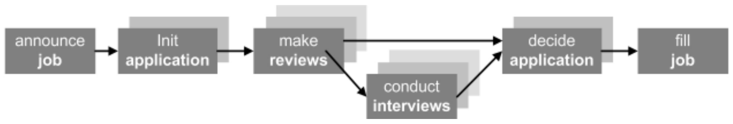


Fig. 7: Process definition based on object interactions

As can be seen from this scenario, the behavior of individual object instances of the same and of different type needs to be coordinated considering their inter-relations as well as their asynchronous processing. Regarding the latter, a personnel officer may continue processing an application while relating reviews are initiated. Regarding the former, a personnel officer is not allowed to read the result of a review before the employee has submitted it. Here, the dynamic number of object instances must be taken into account as well (cf. Fig. 8). The personnel officer, for example, may only reject an application immediately if all reviewers propose its rejection. In this context, interactions between object instances are considered in a broader sense; i.e., even if no direct reference between them exists (e.g., transitive references). More precisely, interdependencies do not only depend on direct relations between object instances, but also on arbitrary relationships taking the individual position of each object instance within the complex process structure into account. For example, consider reviews and interviews corresponding to the same application; i.e., an interview can only be conducted if an employee proposes to invite the applicant.

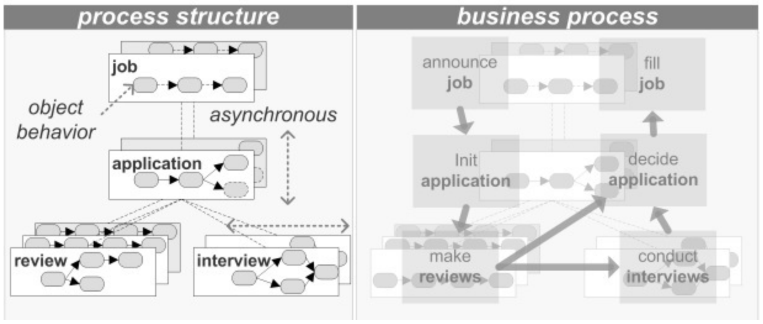


Fig. 8. Process structure at build- and runtime

Relationship between process and activities. In order to proceed with the execution of a process, activities for creating object instances as well as for editing object attributes need to be performed; i.e., mandatorily required data must be provided. In this context, process execution will be not blocked if required data is not available. Instead, if required data is missing a corresponding activity will be automatically created and assigned to the respective users. Furthermore, users may terminate an activity (even if required data is missing) or re-execute it later on.

Relationship between process and user. The execution of activities, which are mandatorily needed for progressing with the process, are assigned to authorized users (i.e., user roles). However, we have to ensure that respective users are authorized to change the attributes processed during activity execution; i.e., process authorization must comply with data authorization. Another relationship addresses the progress of process execution. Basically, the latter depends on available object instances and their attribute values. However, this process enactment may be interrupted in certain situations. In particular, at some points during process execution explicit agreements are needed for proceeding with the flow of control even if data required to reach the next process steps is already available.

3.3 Properties relating to users

Typically, access control is based on role-based mechanisms. In the context of object-aware processes the following properties need to be additionally taken into account.

Relationship between users and data. The assignment of roles to users depends on the relationships of the user to the object instance he may access. For example, an `applicant`, is allowed to read his own `job application`, but not the ones of other `applicants`. Furthermore, for users a data-oriented view is provided in order to access and manage data for which they are authorized at any point in time; e.g., based on overview tables that contain certain object instances. Besides changing object instances users may also decide whether or not they want to create additional object instances for a certain object type. Note that the creation of an additional object instance automatically leads to the creation of a corresponding process instance.

Relationship between users and processes. In addition to the aforementioned *data-oriented view*, for each user a *process-oriented view* (e.g., worklists) is provided. Based on it, upcoming activities can be assigned to the right user at the right point in time. If there exist alternative execution paths within the process instance relating to a particular object instance, users may explicitly decide which path to select.

Relationship between users and activities. Human activities are executed by authorized users. Depending on the data processed by this activity and depending on defined process authorizations, the respective activity is either mandatory or optional for an authorized user.

3.4 Properties relating to activities

Activities can be divided into *form-based* and *black-box* activities. While form-based activities provide *input fields* (e.g., text-fields or checkboxes) for writing and *data fields* for reading selected attributes of object instances, black-box activities enable complex computations as well as the integration of advanced functionalities (e.g., sending e-mails or invoking web services).

Relationship between activities and data. During the execution of both black-box and form-based activities, object instances are created, changed or deleted. Form-based activities can be further divided into instance-specific activities, batch activities and context-specific activities depending on the object instances being processed. *Instance-specific activities* correspond to exactly one object instance (cf. Fig. 9a). When executing such activity, selected attributes of that object instance are read, written or updated using a form (e.g., the form an applicant can use for entering his `application` data). A *context-sensitive activity* also refers to a particular object instance, but additionally includes fields corresponding to higher- or lower-level object instances (cf. Fig. 9b). For example, when an `employee` fills in a `review,` additional information about the corresponding `application` should be provided (i.e., attributes belonging to the `application` for which the `review` is requested). When integrating lower-level object instances, usually, an instance collection is considered. For example, when a `personnel officer` edits an `application`, all corresponding `reviews` should be visible. Generally, many object instances may exist for a particular object type. In this context, *batch activities* allow users to change a collection of selected object instances in one go, i.e., attribute values are set using one form and are then assigned to all object instances (cf. Fig. 9c); e.g., a `personnel officer` might want to flag a collection of `reviews` as "evaluated" in one go. Or, as soon as an `applicant` is hired for a `job`, for all other `applications` value *reject* should be assignable to attribute `decision` by filling one form. Consequently, form-based activities comprise input fields for reading/writing attribute values. Whether or not an input field is displayed for a particular user, however, depends on his data authorizations.

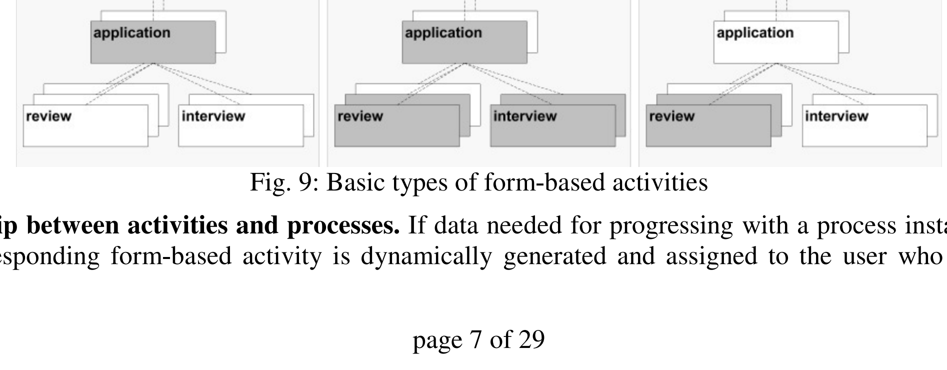


Fig. 9: Basic types of form-based activities

Relationship between activities and processes. If data needed for progressing with a process instance is missing, a corresponding form-based activity is dynamically generated and assigned to the user who owns corresponding permissions. In turn, if required data is already available, process execution proceeds without executing an activity; i.e., using optional activities it is possible to provide data before it is required for proceeding with the process. Whether or not certain attribute values are mandatorily required may also depend on other object attribute values; i.e., there may be interdependencies between the input fields of a form.

Relationship between activities and users. First, depending on respective data and process authorizations, a particular activity might be mandatory for one user, while being optional for another one. Second, users are enabled to re-execute certain activities as long as they have not explicitly agreed to proceed with the flow of control.

3.5 Summary

In summary, process modeling and process execution should be closely correlated with data. As illustrated in Fig. 10, processes should be based on two levels of granularity. On the one hand, behavior of individual object instances needs to be considered; on the other hand, interactions between them must be taken into account. Furthermore, data-driven process execution is indispensable in the given context. Finally, integrated access to processes and data should be enabled. Overall this integration of processes and data will also foster the smooth integration of functions and users.

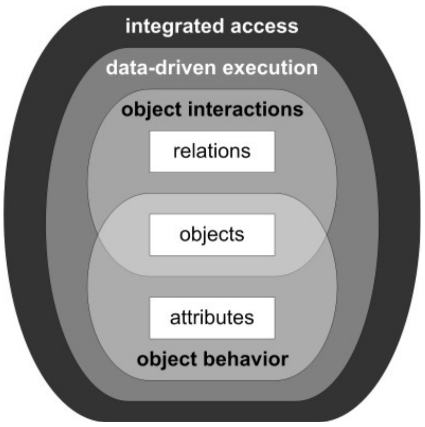


Fig. 10: Main characteristics of object-aware processes

4. Literature Study

In a technical report [15] we have already shown that only limited support for object-aware processes is provided by traditional imperative and declarative approaches. However, there exist extensions of these paradigms which are related to the properties we identified. This section summarizes the results of a respective literature study in regard to a tighter integration of data, users and activities during process modeling and execution. Regarding existing literature, the need for modeling processes in tight integration with data has been recognized by several authors [4,6,10,16,17,19,20,21,22,23,31]. All these approaches confirm the high relevance for process and data integration. In the following we focus on their main characteristics in respect to data integration: *object behavior*, *object interactions*, *data-driven process execution*, and *integrated access to data*.

Object behavior. To enable consistency between process and object states, extensions of imperative approaches based on *object life cycles* (OLC) have been proposed. These extensions include object life cycle compliance [16], object-centric process models [21], business artifacts [6], data-driven process coordination [19], and object-process methodology [10]. To be more precise, an OLC defines the states of an object and the transitions between them. Activities, in turn, are associated with pre-/post-conditions in relation to objects states; i.e., the execution of an activity depends on the current state of an object and triggers the activation of a subsequent state. However, none of these approaches explicitly maps states to attribute values. Consequently, if certain pre-conditions cannot be met during runtime, it is not possible to dynamically react to this; i.e., a data-driven OLC execution is supported by no approach.

Object interactions. Object-centric process models [22] as well as models for the data-driven coordination of processes [19] enable the definition of *object interactions* based on OLCs. Thus, it is possible to asynchronously execute object-specific process instances and to coordinate them at certain points during their execution. Similar support is provided by the object-process methodology [10] as well as by business artifacts [6,17]. Regarding the latter, coordination of the processing of individual artifacts must be defined in a declarative style using constraints. In addition, the object-process methodology only covers process modeling. Consequently, neither the object-process methodology nor business artifacts provide direct support for executing process models. The missing operational semantics, in turn, often results in hard-coded process logic (e.g., [6,17] argue that rule engines are inefficient in the context of distributed systems).
The asynchronous coordination of process instances has been also addressed by Proclets [1]; i.e., lightweight processes communicating with each other via messages. Although process coordination is not explicitly based on the underlying data structure (i.e., processes can be defined at an arbitrary level of granularity), individual processes can be defined in respect to object types. However, the definition of object states is not considered. Since processes are modeled in terms of black-box activities the described data-driven execution is not supported.
Finally, [23,31] describes *product-based processes* [23,31] in which relations between atomic data elements are established; i.e., activities always relate to at least one atomic data element and the process structure corresponds

to the relations between these data elements. However, neither complex objects nor the varying numbers of data elements at runtime are taken into account.

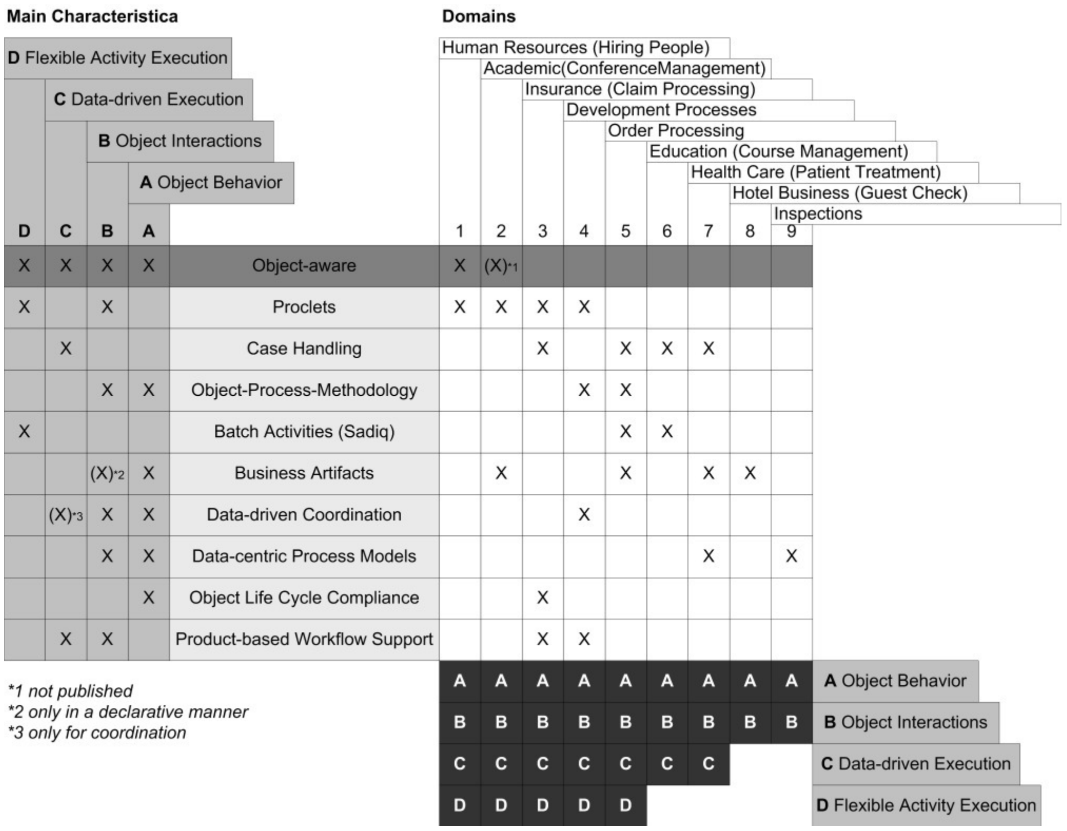
Data-driven execution. In *case handling systems* [4] activities are described as forms in relation to atomic data elements; i.e., processes are defined in an activity-centred way. These data elements can either be *free*, *mandatory* or *optional*. An activity is considered as being completed if all associated mandatory data elements have an assigned value. This way, a data-driven process execution is enabled. Based on *free data elements*, business data not directly relevant for process control or for activity inputs can be added to the process model. Free data elements are assigned to the *case description* (i.e., process model) and can be changed at any point in time by all users. However, only atomic data elements are provided. *Data integration* in terms of object types and their inter-relations is not considered.

Integrated access to data. The need that user assignment of activities should be not only based on user roles, but also take the processed application data into account, is addressed in [24,25]. Here, the authors suggest the concept of "activity-based object-individual resolution of roles". In this context, it should be additionally possible to consider relations between users and object instances (see also [5,11,12]). Finally, [30] and [7] motivate that user assignments should also consider the permissions in respect to the data that is processed by the respective activity. For this purpose, permissions for accessing data and functions in the context of a specific task are defined.
The need for accessing process-related application data at any time during process enactment is motivated in [28]. The authors suggest "instance-based user groups" which grant access to all data of the process instances a particular user is involved in. A similar concept is discussed in [4].
A more flexible execution of activities is provided by declarative approaches [3], which do not want to enforce users to work on activities in a strict execution order (as imposed in traditional imperative approaches). Instead, processes can be defined in terms of constraints prohibiting undesired execution orders of activities. Thus, optional activities can be easily realized. However, current declarative approaches provide only limited support for object-aware processes [15]. [29] motivates the need for batch-activities; i.e., to group activities from different process instances together. However, the grouping only considers activities; data is not taken into account.

As illustrated in Fig. 11, none of the existing approaches considers the identified main characteristics of object-aware processes (i.e., object behavior, object interactions, data-driven execution, and integrated access to data) in a comprehensive and integrated way. In order to underline the high practical value of the different characteristics, Fig. 11 further indicates their relevance in respect to the application domains considered in literature. Since existing approaches partially consider the same scenarios while addressing different properties we can conclude that the characteristics are related to each other and are also required by a variety of processes. For each domain the results are illustrated in the additional rows below the listed approaches. As example, consider order processing as illustrated in Column 5. Order processing was considered as illustrating scenario by Case Handling [4], Object-Process Methodology [10], Batch Activities [29], and Business Artifacts [6,17]. While Case Handling addresses the need for a data-driven execution, the Object-Process-Methodology and Business Artifacts motivate the consideration of object behavior and object interactions. In addition, [29] describes the need for executing several activities in one go. Altogether this indicates that support for all these characteristics is urgently needed in order to adequately support order processes.

As we can also conclude from Fig. 11, a comprehensive approach for object-aware process management is still missing. Also note that Fig. 11 does not make a difference between process modeling and process execution. Though some approaches (e.g., object-process methodology [10]), provide rich capabilities for process modeling, they do not cover runtime issues. In addition to the discussed work, there exist approaches which provide support for single characteristics. As examples consider project management systems or enterprise resource planning systems with integrated workflow component (e.g., SAP Business Objects) [27]. However, in these areas a comprehensive approach for supporting object-aware processes with the described properties is missing as well.

Overall, the conducted literature study has confirmed the high relevance of the indicated properties in respect to the support of object-aware processes (cf. left part of Fig. 11). It further has confirmed that their support is needed in many application domains (cf. right part of Fig. 11).



Main Characteristica

D Flexible Activity Execution
C Data-driven Execution
B Object Interactions
A Object Behavior

Domains

1 Human Resources (Hiring People)
2 Academic(ConferenceManagement)
3 Insurance (Claim Processing)
4 Development Processes
5 Order Processing
6 Education (Course Management)
7 Health Care (Patient Treatment)
8 Hotel Business (Guest Check)
9 Inspections

D	C	B	A		1	2	3	4	5	6	7	8	9
X	X	X	X	Object-aware	X	(X)*1							
X		X		Proclets	X	X	X	X					
	X			Case Handling			X		X	X	X		
		X	X	Object-Process-Methodology				X	X				
X				Batch Activities (Sadiq)					X	X			
		(X)*2	X	Business Artifacts		X			X		X	X	
	(X)*3	X	X	Data-driven Coordination				X					
		X	X	Data-centric Process Models							X		X
			X	Object Life Cycle Compliance			X						
	X	X		Product-based Workflow Support			X	X					
					A	A	A	A	A	A	A	A	A
					B	B	B	B	B	B	B	B	B
					C	C	C	C	C	C	C		
					D	D	D	D	D				

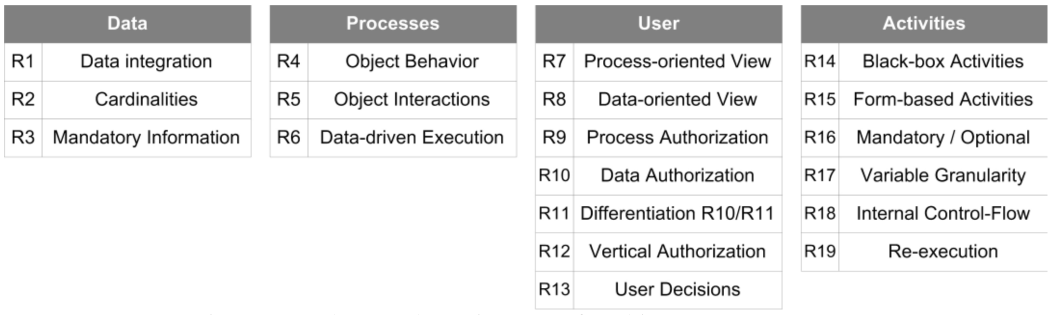
A Object Behavior
B Object Interactions
C Data-driven Execution
D Flexible Activity Execution

*1 not published
*2 only in a declarative manner
*3 only for coordination

Fig. 11: Supported characteristics by existing approaches and considered applications

5. Requirements

In a technical report [15] we have already shown that traditional approaches for PrMS are unable to support object-aware processes. In particular, existing imperative, declarative and data-driven approaches have shown limitations when dealing with the identified properties. When comparing the main characteristics of existing activity-centred approaches with our identified properties, we derived major requirements for the support of object-aware processes. This section summarizes these requirements (cf. Fig. 12). We categorize them again along the main building blocks of PrMS. Further, we illustrate them along the recruitment example as introduced in Section 3.



Data		Processes		User		Activities	
R1	Data integration	R4	Object Behavior	R7	Process-oriented View	R14	Black-box Activities
R2	Cardinalities	R5	Object Interactions	R8	Data-oriented View	R15	Form-based Activities
R3	Mandatory Information	R6	Data-driven Execution	R9	Process Authorization	R16	Mandatory / Optional
				R10	Data Authorization	R17	Variable Granularity
				R11	Differentiation R10/R11	R18	Internal Control-Flow
				R12	Vertical Authorization	R19	Re-execution
				R13	User Decisions		

Fig. 12: Fundamental requirements for object-aware processes

5.1 Data

R1 (Data integration). Data should be manageable in terms of object types comprising object attributes and relations to other object types.

R2 (Cardinalities). A varying number of object instances corresponding to each object type must be considered. Further, it should be possible to restrict the relations between object instances through *cardinality constraints*.

R3 (Mandatory information). To reach the next object instance state from the current one, certain object attribute values are required. For this, a form-based activity with mandatory input fields needs to be assigned to the worklists of authorized users. When executing this activity, specific input fields referring to mandatorily required attributes have to be filled. Other input fields may be optionally set.

5.2 Processes

R4 (Object behavior). It should be possible to determine in which order and by whom the attributes of a particular object instance have to be (mandatorily) written, and what valid attribute value settings are. In addition, when executing black-box activities the involved object instances need to be in certain states. Consequently, for each object type its behavior should be definable in terms of states and transitions. In particular, it should be possible to drive process execution based on data and to dynamically react on changes of object attributes. Consequently, it is crucial to map object states to object attribute values.

R5 (Object interactions). Generally, a process deals with a varying number of object instances of the same and of different object types. In addition, for each processed object instance its behavior must be considered. In this context, it should be possible to process object instances in a loosely coupled manner (i.e., *concurrently* to each other) and to *synchronize* their execution where needed. Consequently, a process structure should be formed based on object interactions. Additionally, the integration of black-box activities should be possible.

R6 (Data-driven process execution). Mandatory activities are obligatory for process execution; i.e., they enforce the setting of object attribute values as required in order to progress with the process. In principle, it should be possible to set respective attributes also up front (i.e., in an earlier phase of the process) by executing optional activities; i.e., before the mandatory activity normally writing this attribute becomes activated. In the latter case, the mandatory activity no longer needed should be automatically skipped when being activated.

5.3 Users

R7 (Process-oriented view). During process execution some activities have to be mandatorily executed while others are optional. To ensure that mandatory activities are executed at the right point in time, they must be assigned to the worklists of authorized users when they become activated.

R8 (Data-oriented view). Access to data should be granted at any point in time given the required authorizations; i.e., not only during the execution of a particular activity.

R9 (Process authorization). For each mandatory activity at least one user or user role should be assigned to it at runtime. Regarding a form-based activity, each user who may execute it must have the permissions for reading/writing corresponding attribute values [7].

R10 (Data authorization). To provide access to data at any point in time, we need to define permissions for creating and deleting object instances as well as for reading/writing their attributes. However, attribute changes contradicting to object behavior should be prevented. For this, the progress of the process has to be taken into account when granting permissions to change objects attributes [7,28]. Otherwise, if committed attribute values were changed afterwards, object instance state would have to be adjusted to cope with dirty reads. Generally, data permissions should be made dependable on the states as captured by object behavior. This is particularly challenging for context-sensitive and batch activities, since attribute changes have to be valid for all selected instances.

R11 (Differentiating authorization and user assignment). When executing mandatory activities particular object attributes have to be set. To determine which user shall execute a pending mandatory activity, her permissions for writing object attributes need to be evaluated. While certain users must execute an activity mandatorily in the context of a particular object instance, others might be authorized to optionally execute this activity; i.e., mandatory and optional permissions should be distinguishable. In particular, a mandatory activity should be only added to the worklists of users having "mandatory permissions". Users with "optional permissions", in turn, may change the corresponding attributes when executing optional activities.

R12 (Vertical authorization). Usually, human activities are associated with actor expressions (e.g., user roles). We denote this as *horizontal authorization*. Users who may work on respective activities are determined at runtime based on these expressions. For object-aware processes, however, the selection of potential actors should not only depend on the activity itself, but also on the object instances processed by it [24,25]. We denote this as *vertical authorization*.

R13 (User decisions). Generally, different ways for reaching a process goal may exist. Usually, the selection between such alternative execution paths is based on history data; i.e., on completed activities and available process-relevant data. In our context, this selection might be also based on explicit user decisions.

5.4 Activities

R14 (Black-box activities). To ensure proper execution of black-box activities, we need to be able to define preconditions on attribute values of processed object instances. If their input parameters belong to different object instances, their inter-relationships should be controllable. Opposed to form-based activities, which should be automatically generated by the runtime system, for each black-box activity an implementation is required.

R15 (Form-based activities). A form-based activity comprises a set of atomic *actions.* Each of them corresponds to either an *input field* for writing or a *data field* for reading the value of an object attribute. Which attributes may be written or read in a particular form-based activity depends on the user invoking this activity and the state of the object instance. Consequently, a high number of form variants exists. Since it is costly to implement them all, it should be possible to automatically generate form-based activities at runtime.

R16 (Mandatory and optional activities). Depending on the state of object instances certain activities are mandatory for progressing with the control-flow. At the same time, users should be allowed to optionally execute additional activities (e.g., to write certain attributes even if they are not required at the moment).

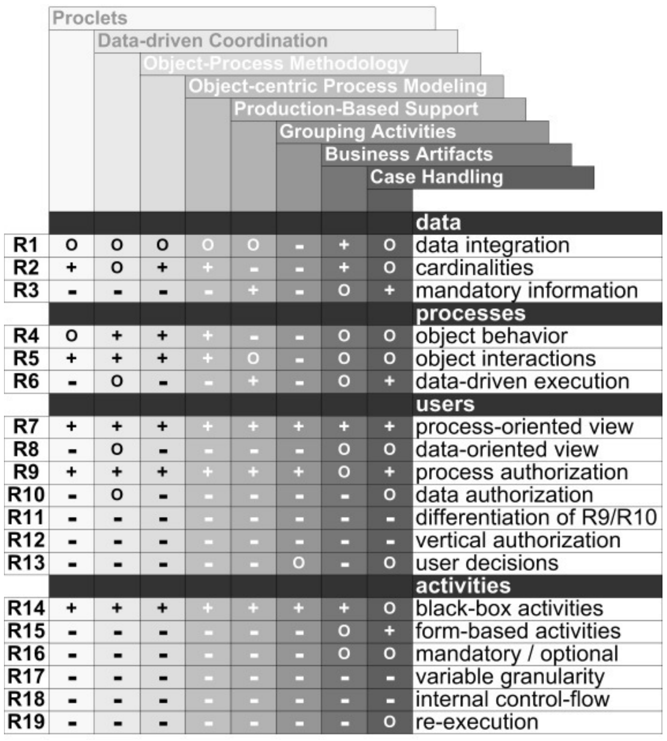
R17 (Variable granularity). As discussed, support for instance-specific, context-sensitive, and batch activities is required. Regarding *instance-specific* activities, all actions refer to attributes of one particular object instance, whereas *context-sensitive* activities comprise actions referring to different, but related object instances (of potentially different type). Since *batch activities* involve several object instances of the same type, for them each action corresponds to exactly one attribute. Consequently, the attribute value must be assigned to all referred object instances. Depending on their preference, users should be allowed to freely choose the most suitable activity type for achieving a particular goal. Finally, executing several black-box activities in one go should be supported.

R18 (Control-flow within user forms). Whether certain object attributes are mandatory when processing a particular activity might depend on other object attribute values; i.e., when filling a form certain attributes might become mandatory on-the-fly. Such control flows specific to a particular form should be considered.

R19 (Re-execution). Users should be allowed to re-execute a particular activity (i.e., to update its attributes), even if all mandatory object attributes have been already set.

5.5 Summary

Fig. 13 gives an overview on how well the different approaches discussed in Section support the requirements elicited in this section. Due to lack of space we abstain from a detailed discussion of this evaluation.



	Proclets	Data-driven Coordination	Object-Process Methodology	Object-centric Process Modeling	Production-Based Support	Grouping Activities	Business Artifacts	Case Handling	
									data
R1	o	o	o	o	o	-	+	o	data integration
R2	+	o	+	+	-	-	+	o	cardinalities
R3	-	-	-	-	+	-	o	+	mandatory information
									processes
R4	o	+	+	+	-	-	o	o	object behavior
R5	+	+	+	+	o	-	o	o	object interactions
R6	-	o	-	-	+	-	o	+	data-driven execution
									users
R7	+	+	+	+	+	+	+	+	process-oriented view
R8	-	o	-	-	-	-	o	o	data-oriented view
R9	+	+	+	+	+	+	o	+	process authorization
R10	-	o	-	-	-	-	-	o	data authorization
R11	-	-	-	-	-	-	-	-	differentiation of R9/R10
R12	-	-	-	-	-	-	-	-	vertical authorization
R13	-	-	-	-	-	o	-	o	user decisions
									activities
R14	+	+	+	+	+	+	+	o	black-box activities
R15	-	-	-	-	-	-	o	+	form-based activities
R16	-	-	-	-	-	-	o	o	mandatory / optional
R17	-	-	-	-	-	-	-	-	variable granularity
R18	-	-	-	-	-	-	-	-	internal control-flow
R19	-	-	-	-	-	-	-	o	re-execution

Fig. 13: Evaluation of existing approaches

6. The PHILharmonicFlows Framework for Object-aware Process Management

We now introduce our PHILharmonicFlows framework for object-aware process management. It comprises several components for the integrated support of data and processes, and addresses the aforementioned requirements. Object types and relations are defined in a data model, while object behavior is controlled by a process whose execution is driven by changes of the object attributes. The framework further provides support for coordinating the execution of different processes and the interactions of their relating objects, respectively, while taking object relations from the data model into account. Finally, an authorization component is provided to control access to data and processes. In the following we provide an overview of the different components of this framework. We focus on their core functionalities and interrelations. Due to lack of space we omit a description of the formal semantics of the models driving the execution of processes as well as the coordination of their interactions. Instead we provide informal descriptions and illustrate basic concepts along our running example. Section 7 further gives insights into our proof-of-concept prototype and its application to practical scenarios.

6.1 Overview

Fig. 14 gives an overview of the different components of the PHILharmonicFlows framework. Basically, it provides a modeling as well as a runtime environment to enable full lifecycle support of object-aware processes.

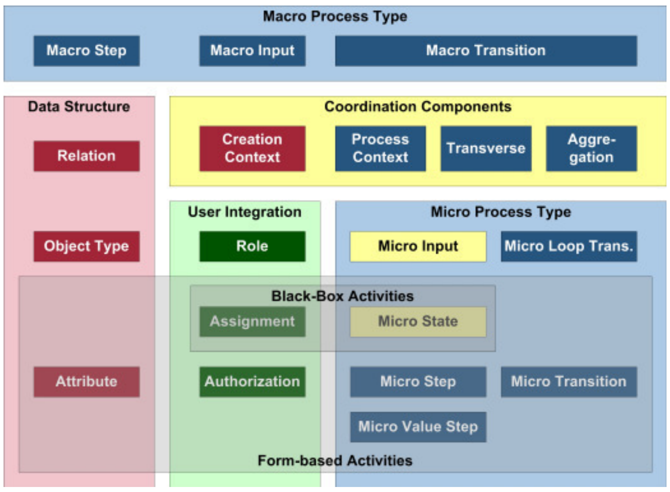


Fig. 14: Components of the PHILharmonicFlows Framework

The *modeling environment* of the PHILharmonicFlows framework enforces a well-defined modeling methodology that governs the definition of processes at different levels of granularity. More precisely, the framework differentiates between *micro* and *macro processes* in order to capture both *object behavior* and *object interactions*. As opposed to existing approaches, in which activities and their execution constraints (e.g., precedence relations) are explicitly specified, PHILharmonicFlows allows to define processes in tight integration with data. As a prerequisite, object types and their relations need to be captured in a data model. For each *object type* then a *micro process type* has to be specified. The latter defines the behavior of corresponding object instances and consists of a set of *states* and the *transitions* between them. Each state, in turn, is associated with a set of *object type attributes*. At runtime, a micro process instance being in a particular state may then only proceed if specific values are assigned to the attributes of this state; i.e., a *data-driven process execution* is enabled. Usually, this is accomplished with *form-based activities* to be executed by authorized users.
Process authorization is based on *user roles* which can be assigned to the different *states* of a micro process. When such state becomes enabled at runtime, respective users have to mandatorily set the values for all object attributes relating to this state in order to proceed with the flow of the micro process; i.e., they have to apply a number of so called *micro steps* to accomplish these attribute changes. *Optional access to data*, in turn, is enabled asynchronously to process execution and is based on permissions for creating and deleting object instances as well as for reading/writing their attributes. The latter must take the current progress of the corresponding micro process instance into account. For this purpose, PHILharmonicFlows maintains a comprehensive *authorization table* which assigns data permissions to user roles dependent on the different states of the micro process type.

If an object instance is created for a given object type a corresponding micro process instance is automatically created as well. Taking the relations between the object instances of the overall *data structure* into account, the corresponding micro process instances additionally form a complex *process structure*; i.e., their execution needs to be coordinated according to the given data structure. In PHILharmonicFlows this can be realized by means of macro processes. Such a *macro process* refers to parts of the data structure and consists of *macro steps* as well as *macro transitions* between them. Opposed to micro steps which relate to single attributes of a particular object type, a macro step refers to a whole object type. As we show in the following, based on macro processes PHILharmonicFlows is able to hide the complexity of large process structures from modelers. Opposed to existing approaches, in addition, various synchronization constraints may be defined for coordinating the interactions between the object instances of the same as well as of different object types. Although processes are tightly integrated with data, PHILharmonicFlows enables the integration of black-box activities (e.g., to automatically invoke a web service or to send an e-mail) as well.

Further, the runtime environment provides *data-* as well as *process-oriented views* to end-users; i.e., authorized users may invoke activities for accessing data at any point in time as well as activities needed in order to proceed within the micro process flow. PHILharmonicFlows is based on a well-defined formal semantics, which enables us to automatically generate most end-user components of the runtime environment (e.g., tables giving an overview on object instances, user worklists, form-based activities). Thus, an implementation is only required for black-box activities which enable, for example, the execution of complex computations or the integration of existing applications.

6.2 Data

As discussed, process and data modeling constitute two sides of the same coin. Consequently, the proper integration of the data model constitutes a fundamental requirement for any framework supporting object-aware processes **(cf. R1: data integration)**. PHILharmonicFlows uses a *relational data model* which is based on *object types* as well as their *attribute* and *relation types* (cf. Fig. 15). Thereby, attribute types represent *atomic data elements* which describe the properties of the respective object type. Concerning relation types, in addition, minimal and maximal *cardinalities* can be specified. Since normalization constitutes an integral part of the relational model, all relations form 1-to-many relationships; i.e., many-to-many-relationships have to be dissolved in our approach by using additional 1-to-many-relations. Finally, for each object type exactly one *key attribute type* exists. Based on its key attribute value, an object instance can be uniquely identified at runtime.

Example (Data model): Regarding our recruitment example relevant object types include `job offer`, `application`, `review`, `interview`, and `participant` (cf. Fig. 15). Each `application`, for instance, corresponds to exactly one `job offer`. For each `interview`, in turn, at the minimum two and at the maximum five `participants` must be available. Besides this, each object type is characterized by a set of attributes. As example consider attribute `proposal` of object type `review`. Using this attribute the respective `reviewer` makes a proposal on how to proceed with the corresponding `application`.

6.2.1 Relationships

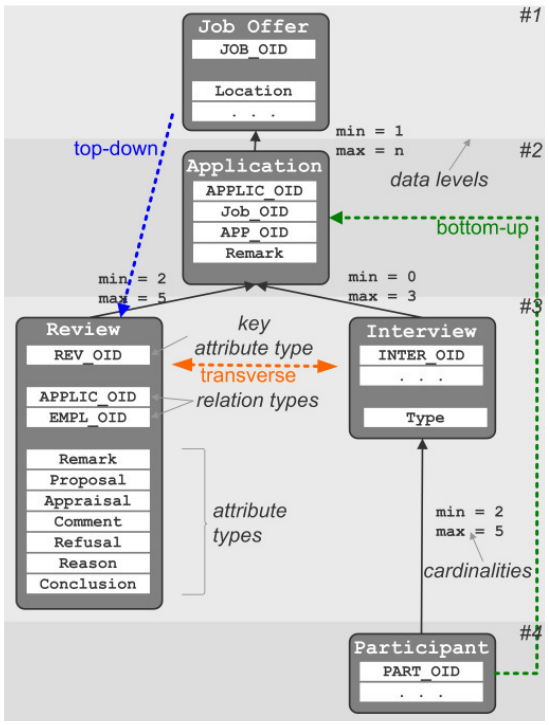


Fig. 15: Data model in PHILharmonicFlows

As we will see in the following, it is not always sufficient to only consider direct relations between object instances (e.g., indicating that a given `review` object corresponds to a particular `application` object). In the context of process coordination and user authorization, in addition, indirect (i.e., transitive) relationships may have to be taken into account (e.g., in order to access all `reviews` related to a particular `job offer`). To cover this, PHILharmonicFlows structures a data model into different *data levels*. All object types which do not refer to any other object type are placed on Level #1. Further, as illustrated in Fig. 15, any other object type is always assigned to a lower data level as the object types it references. For the sake of simplicity, we do not discuss self-references and cyclic relations here, though we consider them in our overall framework.

Example (Data levels): A `job offer` does not reference any other object type. Consequently, this object type is placed on the highest data level. Since the object type `review` includes a relation to the object type `application`, the `review` object type is placed on a lower data level as the object type `application`.

We denote an object type `A` which directly or indirectly references an object type `B` as *lower-level object type* of `B`. Corresponding to this, an object type which is directly or indirectly referenced by other object types is denoted as *higher-level object type*. As further illustrated in Fig. 15, a *relationship* from an object type `A` to an object type `B` is categorized as *top-down* (*bottom-up*) if `B` is a lower-level (higher-level) object type of `A`. Furthermore, we categorize a relationship between object types `A` and `B` as *transverse* if there exists another object type `C` of which both `A` and `B` are lower-level object types (i.e., `A` and `B` have a higher-level object type in common). Otherwise (i.e., neither of the categorizations *top-down*, *bottom-up* and *transverse* apply) the relationship between two object types is categorized as *general*; i.e., there exists no explicit relationship between them.

Example (Top-down relationship): A `review` object type includes a relation to the `application` object type.

The latter, in turn, has a relation pointing to the `job offer` object type. Thus, the `job offer` object type is transitively referenced by the `review` object type. Consequently, there exists a top-down relationship from the `job offer` object type to the `review` object type.

Example (Bottom-up relationship): A `participant` object type includes a relation to the `interview` object type. The latter, in turn, has a relation pointing to the `application` object type. Consequently, the `participant` object type transitively references the `application` object type; i.e., there is a bottom-up relationship from the `participant` object type to the `application` object type.

Example (Transverse relationship): Both the `review` object type and the `interview` object type include a relation to the `application` object type; i.e., there exists a transverse relationship between these two object types.

6.2.2 Integrating Users

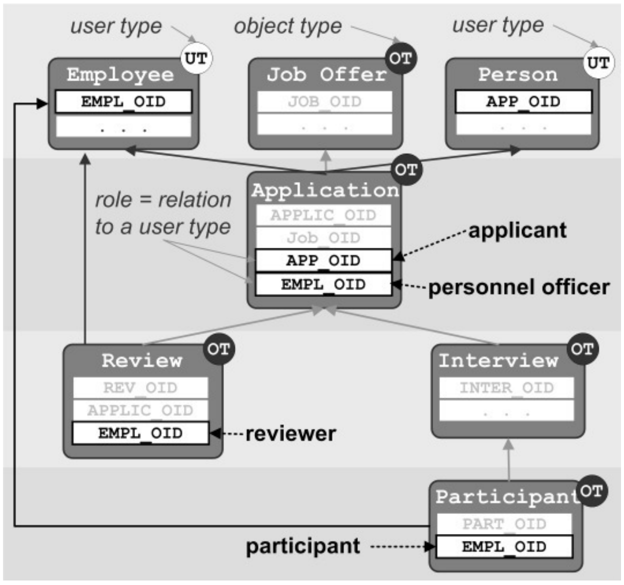


Fig. 16: User integration in PHILharmonicFlows

Regarding the discussed dependencies between data and user (cf. Section 3), user permissions do not only depend on roles, but also on relationships to object instances. Consequently, it is not sufficient to manage the user / role definitions as well as the organizational model independent from application data. Instead, PHILharmonicFlows allows to integrate users with objects. For this purpose, an object type can be flagged as *user type* (cf. Fig. 16). At runtime, each object instance corresponding to a user type represents a particular user. In order to authenticate him (i.e., if he logs on), each user type also includes attributes which uniquely identify a user at runtime (e.g., username and password). Furthermore, each user type represents a *user role*.

As advantage of this close integration between data and users, we can automatically determine additional user roles from a given user type and its relations to object types. We denote such derivable user role as *relation role*.

Example (User types): Regarding our recruitment example, relevant user types are `applicant` and `employee`.

Example (Relation roles): In our recruitment example relevant roles include `applicant`, `personnel officer`, `reviewer`, and `participant` (in an `interview`).

Usually, for a particular user, her access to data should be restrictable to a subset of the object instances of an object type; e.g., a `reviewer` should only be enabled to access `applications` for which she has to prepare a `review` and not for other ones. We denote such fine-grained access control as *instance-specific role assignment*. More precisely, while relation roles are determined based on the relation types (i.e., at build time), instance-specific role assignment is based on the relations at runtime. For this purpose, PHILharmonicFlows allows to restrict access on the object instances of a certain object type **(cf. R12: vertical authorization)**. In this context, it is further possible to consider the relationship between a certain user and a respective object instance (i.e., to consider whether or not there exists a top-down, bottom-up or transverse relationship).

Example (Instance-specific role based on a top-down relationship): `Applicants` may only access their own `applications`.

Example (Instance-specific role based on a bottom-up relationship): `Participants` may only access those `interviews` in which they participated.

Example (Instance-specific role based on a transverse relationship): `Reviewers` may only access those `interviews` which belong to an `application` they have to evaluate.

Based on a given data model, at runtime a corresponding data structure, which comprises a collection of object instances and their relations, dynamically evolves. An important aspect in this context concerns the cardinalities of object type relations. For this purpose, PHILharmonicFlows maintains a *creation context* for each higher-level object instance, which observes whether the minimum number of required lower-level object instances is available. More precisely, PHILharmonicFlows automatically ensures that no required lower-level object instance relating to a higher-level one is missing. If a lower-level object instance is missing, a corresponding activity is automatically assigned to the worklist of authorized users; i.e., an activity for creating additional object instances relating to a certain higher-level one must be mandatorily executed. Similarly, this creation context is used to disable the creation of lower-level object instances after the maximum cardinality has been reached **(cf. R2: cardinalities)**. Generally, the concrete number of lower-level object instances may depend on user decisions as long as the defined cardinality constraints are fulfilled **(cf. R13: user decisions)**.

Example (Creation Context): If for a particular `interview` object instance less than two `participants` are available, a corresponding mandatory activity is automatically generated and assigned to the worklists of the responsible `personnel officer`. In turn, if five `participants` have been already specified, it is no longer possible to define additional ones. However, if the number of available `participants` is between two and four the `personnel officer` may optionally specify additional ones.

Data authorization requires permissions for creating and deleting object instances as well as for reading/writing corresponding attribute values. In this context the explicitly defined data model as well as the permissions defined for each user role provide the basis for dynamically creating overview tables (e.g., containing all object instances a particular user may access) and form-based activities (e.g., comprising those object attributes that can be read/written by a user in the given context) at runtime. Since data authorization also depends on the progress of a process, however, it also necessitates the presence of a respective process model. Note that data authorization issues cause huge implementation efforts in existing business applications. PHILharmonic Flows therefore supports the specification of explicit process models describing object behavior and enabling us to link data authorization not only to user roles but also to process states.

6.3 Micro Processes

To enable object-aware processes their modeling must consider two well-defined levels of granularity. More precisely, *object behavior* and *object interactions* have to be captured. The former necessitates a specific process definition for each object type, while the latter requires the definition of processes involving several object types (cf. Section 6.4). This section focuses on the specification of object behavior, while object interactions are considered in Section 6.5.
In PHILharmonicFlows for each object type a corresponding *micro process* can be defined **(cf. R4: object behavior)**. At runtime, the creation of a micro process instance is then directly coupled with the creation of a corresponding object instance. One particular challenge emerging in this context is to coordinate the processing of an individual object instance among different users taking into account data authorizations as well. Another challenge is to define the internal flow regarding the execution of a form-based activity; i.e., to define the flow between its input fields. Besides such form-based activities, black-box activities have to be integrated as well.
At runtime, PHILharmonicFlows enables the automatic generation of form-based activities and their internal logic. Note that we define object behavior based on data in order to make mandatorily required data (i.e., attribute values) transparent. Generally, the progress of a micro process may depend on the availability of data as well as on user decisions. In the former situation a data-driven process execution should be enabled. In the latter case, even though required data are available the micro process may only proceed with its execution after a user has explicitly committed this.

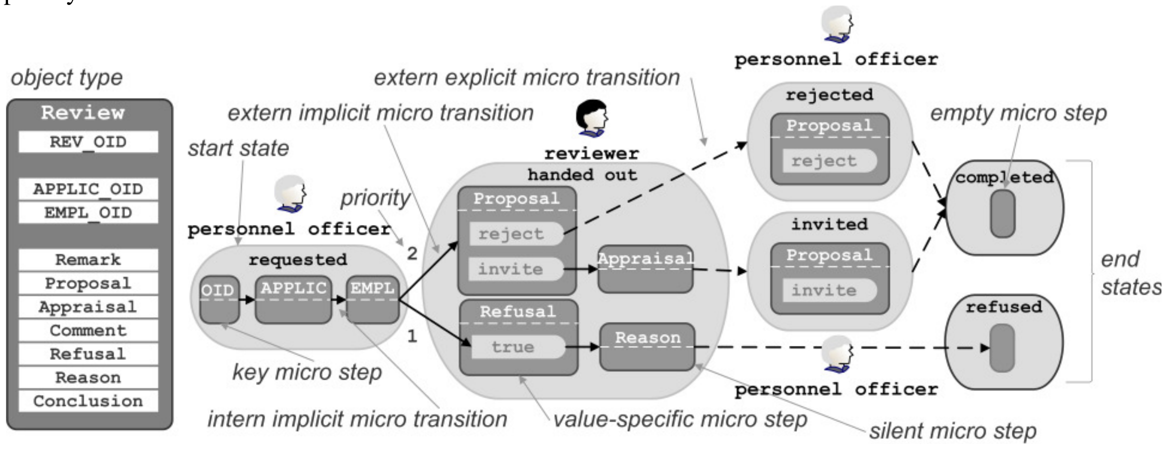


Fig. 17: Micro process of a review object as modeled in PHILharmonicFlows

Example (States): Each instance of object type `review` must be `requested` by a `personnel officer` and is then `handed out` to a `reviewer`. The latter can either `refuse` the `review` request or fill out the corresponding `review` form. In particular, the `reviewer` may either suggest to `reject` or `invite` the `applicant`. If the `reviewer` submits the `review`, the `personnel officer` has to evaluate the provided feedback. In order to assist him, the micro process includes two separate states, one for `reviews` proposing the rejection (i.e., state `rejected`) and one for `reviews` proposing the invitation (i.e., state `invited`). Overall, this example comprises three activities which must be mandatorily executed in order to proceed within process execution.

Similar to existing approaches that consider object behavior (cf. Section 4), PHILharmonic Flows provides a state-based approach. As illustrated in Fig. 17, *states* serve as the basis for coordinating the mandatory activities of a micro process among different user roles. For this purpose, the user roles need to be assigned to the different states **(cf. R9: process authorization)**. Opposed to other approaches, however, PHILharmonic Flows clearly defines which attribute values must be available in order to leave a certain process state.

6.3.1 Micro Steps

Opposed to other state-based approaches, process states are explicitly defined in respect to attributes; i.e., it depends on attribute values whether or not a certain process state can be reached or left. For this purpose, each state consists of several *micro steps*. Each of them represents a mandatory write access on a particular attribute or relation of the processed object instance **(cf. R3: mandatory information)**. Note that single micro steps do not represent activities, but solely refer to one atomic action (e.g., editing an input field within a form). For reaching such a micro step at runtime, a value for its corresponding attribute or relation is mandatorily required. We denote those micro steps which are completed when an arbitrary value is set as *atomic micro step*.

Example (Atomic micro steps): When initiating a `review` within micro state `requested`, a `personnel officer` must specify a corresponding (job) `application` object as well as an `employee` who shall fill in the `review`.

Further, different values (or value ranges) of an object attribute or relation are explicitly specified in the corresponding micro step in case they influence object behavior. In the following we denote micro steps with finite number of attribute values (or relevant value ranges) as *value-specific*. Within user forms value-specific micro steps can be displayed using combo boxes or radio buttons. This way, invalid values can be avoided and process control can be based on meaningful object attributes.

Example (Value-specific micro steps): Concerning micro state `handed out`, the `reviewer` has to assign a value either to attribute `proposal` or to attribute `refusal`; i.e., he has two options. As opposed to the micro steps within the previous state, for which "arbitrary" values can be set, attributes `proposal` and `refusal` require specific values. For example, the `proposal` may either be to *invite* or *reject* the `applicant`. For object attribute `refusal`, in turn, only value *true* is relevant for process control.

6.3.2 States

Each micro process has a unique *start state*, *intermediate micro states* and one or several *end states*. At any point during process execution exactly one state is enabled. More precisely, the start state becomes enabled when a respective object instance is created. It is then processed as any other state. As soon as an end state is reached, the micro process instance is completed. More precisely, end states do not require the setting of any object attribute and therefore neither include atomic nor value-specific micro steps. However, we add a *silent micro step* (i.e., a micro step that does not refer to any object attribute; cf. Fig. 17) to each end state in order to be able to connect him with other states (cf. Section 6.5). Furthermore, silent micro steps can be added to intermediate micro states. This way, it is possible to model activities for mandatorily reading a certain object instance (cf. *explicit external micro transitions* in Section 6.5).

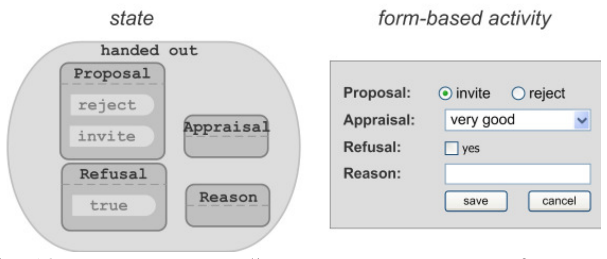


Fig. 18: Form corresponding to state `handed out` of a `review`

Generally, for each micro state (except end micro states) the required attribute values are defined by its associated atomic and value-specific micro steps. When a micro state becomes enabled during the processing of an object instance, usually, a corresponding form-based activity for entering the required attribute values is automatically generated (cf. Fig. 18).

Example (Start micro state): Consider Fig. 17. If the `personnel officer` requests a new `review`, a corresponding micro process instance is created and its start state `requested` is enabled.

Example (Intermediate micro states): As example for intermediate micro states consider the states `handed out`, `rejected` and `invited`. Regarding the former, the respective `reviewer` has to fill in the `review` form. If the latter is enabled the `personnel officer` has to evaluate the `proposals`.

Example (End micro states): If the `reviewer` refuses the `review` request the `review` will immediately reach the end state `refused`. Opposed to this, if the `review` is provided the end state `completed` will be reached after the `personnel officer` has evaluated the provided feedback.

6.3.3 Internal Micro Transitions

Ideally, users can be guided in setting the required attribute values when processing a form-based activity; e.g., by highlighting respective input fields. In order to capture such internal logic in respect to the setting of different object attributes, their corresponding micro steps can be linked using *micro transitions*. In this context, we denote a micro transition linking two micro steps within the same micro state as *internal micro transition*. Based on them we define the internal logic of a mandatory, form-based activity; i.e., the default order in which the input fields of the corresponding form shall be edited **(cf. R18: control-flow within user forms)**.

Example (Internal micro transitions): Consider the micro state `handed out` in Fig. 17. For object attribute `appraisal` a value is only required if attribute `proposal` has value *invite*.

Consider an outgoing micro transition of a value-specific micro step. Generally, such a transition can be either associated with a specific value of the respective object attribute or with the whole micro step. In the former case, the subsequent micro step (to which the micro transition targets) will be only enabled if the specific value is actually assigned to this object attribute at runtime. Regarding an incoming micro transition of a value-specific micro step, in turn, it is possible to connect it to a specific attribute value in order to express that in the given case a specific attribute value is required.

The internal logic we define for micro steps shall help to guide users in filling corresponding forms. However, when executing a form-based activity, users should be flexible in which order they actually process its micro steps. Despite any predefined sequence of micro steps, users should be able to freely choose their preferred execution order; i.e., the order in which required values are assigned to object attributes does not necessarily have to coincide to the one defined for the corresponding micro steps. In particular, at runtime a micro step can be completed as soon as a value is assigned to its object attribute **(cf. R6: data-driven process execution)**.

Example (Data-driven execution): Consider the start state `requested` of the micro process depicted in Fig. 17. When filling the respective form, a `personnel officer` may want set a value for relation `employee` although he is guided to first fill in the input field relating to relation `application`. Consequently, if a value for relation `application` is set afterwards, the subsequent micro step relating to relation `employee` is automatically completed.

By using optional activities, and these constitute another important feature of our approach, authorized users may read/write object attributes asynchronously to the normal execution of the respective micro process instance. This way attribute values may be provided before they are mandatorily required for process execution. If a (specific) value for an attribute is written up front the respective micro step will be automatically completed immediately after its activation. In this context, we have to consider the case that micro steps may have several subsequent micro steps (i.e., each micro step may contain more than one outgoing micro transitions). Here, PHILharmonicFlows ensures that only one subsequent micro step is completed at any point during process execution. i.e., always exactly one micro step (and thereby one micro state) can be reached. For this purpose, different *priorities* as illustrated in Fig. 17 can be assigned to micro transitions. Only the subsequent micro step which is connected with the micro transition having the highest priority is then completed.

Example (Priorities): A `reviewer` may set the values for attributes `proposal` and `refusal` up front, i.e., while micro state `requested` is enabled (cf. Fig. 17). In such case, the micro steps corresponding to object attributes `proposal` and `refusal` will be auto-completed as soon as the micro state `handed out` is enabled. In this situation, the `review` will be `refused` since the incoming micro transition of micro step `refusal` has higher priority.

Note that the way we define micro processes (i.e., object behavior) does not forbid parallel execution. More precisely, during the execution of a particular activity (i.e., the micro process is in a particular micro state), parallel processing of disjoint sets of mandatory as well as optional object attributes is possible. In addition, different users may concurrently process forms corresponding to the same object instance and the same state of its corresponding micro process instance respectively. In this context, known mechanisms for concurrent data access are applied.

6.3.4 External Micro Transitions

As opposed to internal micro transitions, an *external micro transition* connects two micro steps belonging to different micro states; i.e., its firing triggers a subsequent micro state. Generally, a micro state is enabled as soon as one of its micro steps is triggered by an incoming external transition; i.e., the data-driven execution paradigm is applied for activating subsequent states as well. However, in order to also support scenarios in which an activity-based paradigm (i.e., the user explicitly commits the completion of the activity he has worked on) is more favorable, we differentiate external micro transitions either as implicit or explicit. An *implicit micro transition* is fired as soon as its source micro step completes. This way the data-driven execution paradigm is also applied for activating subsequent states.

Example (Implicit external micro transitions): A `reviewer` may fill in the `review` as soon as the `employee` has set a value for object attributes `application` and `employee` (cf. Fig. 17).

An *explicit external micro transition*, in turn, additionally requires that a user explicitly confirms the completion of its source micro state; i.e., even if the target micro step (and target micro state respectively) can be reached, the progress of the micro process is blocked until an authorized user explicitly confirms that the current micro state can be left **(cf. R13: user decisions)**. As example consider Fig. 17 where the dashed lines represent explicit external micro transitions. In order to authorize users to commit the completion of activities explicitly, an explicit micro transition is associated with a user role **(cf. R9: process authorization)**.

Example (Explicit external micro transitions): Consider micro state `handed out` in Fig. 17. Based on the described semantics a `reviewer` may re-execute the activity for filling in her `review` even if all mandatory object attributes have been already set. This gives her the flexibility to change her `proposal` if desired. However, after she commits to leave the micro state `handed out`, this is no longer possible.

When re-executing a mandatory activity, previously reached micro steps of the corresponding micro state have to be re-initialized. Generally, the micro steps of the current micro state will be reset if an already written attribute value is changed afterwards **(cf. R19: re-execution)**.

Example (Re-initialization): If the `reviewer` refuses the requested `review`, the next action is to commit the migration of the micro process instance to the subsequent micro state `refused`. However, the `reviewer` may want to re-think her decision and fill in the `review` after all; i.e., she may re-execute the activity, which requires that already reached micro steps (corresponding to attributes `refusal` and `reason` in our case) must be re-initialized.

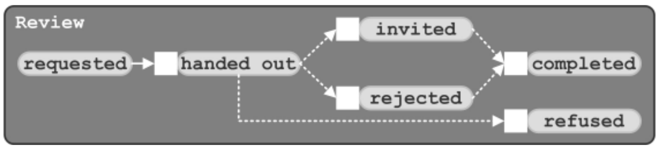


Fig. 19: State-based view of a micro process

For each defined micro process, PHILharmonic Flows automatically generates state-based views which are further used for process synchronization, authorization and monitoring (cf. Fig. 19).

Similar to existing approaches, PHILharmonicFlows allows to integrate *black-box activities* (e.g., sending e-mails or invoking web services) by assigning them to particular states. In this context, we differentiate between black-box activities with parameters corresponding to the same object type and black-box activities with parameters referring to several object types. While the latter are included during *macro process modeling* (cf. Section 6.5), the former can be directly assigned to single micro states within the micro process definition of the corresponding object type **(cf. R14: black-box activities)**. As opposed to existing approaches, however, PHILharmonicFlows supports the differentiation between a data- and an activity-driven execution paradigm at runtime. For the latter, the black-box activity must be categorized as *required*. This means that the micro state to which the activity belongs can only be left if the activity is actually executed. For this purpose, a required black-box activity is assigned to the worklist of authorized users (or is automatically executed in case no user role is specified) when the respective state is enabled. By contrast, for a *non-required* black-box activity it is possible to leave its respective micro state even if the activity was not executed.

Concerning a particular state of a micro process instance, different mandatory activities may be assigned to the worklist of a responsible user. These comprise form-based activities for setting object attributes, activities for committing changes before migrating to a subsequent state, and required black-box activities **(cf. R7: process-oriented view)**.

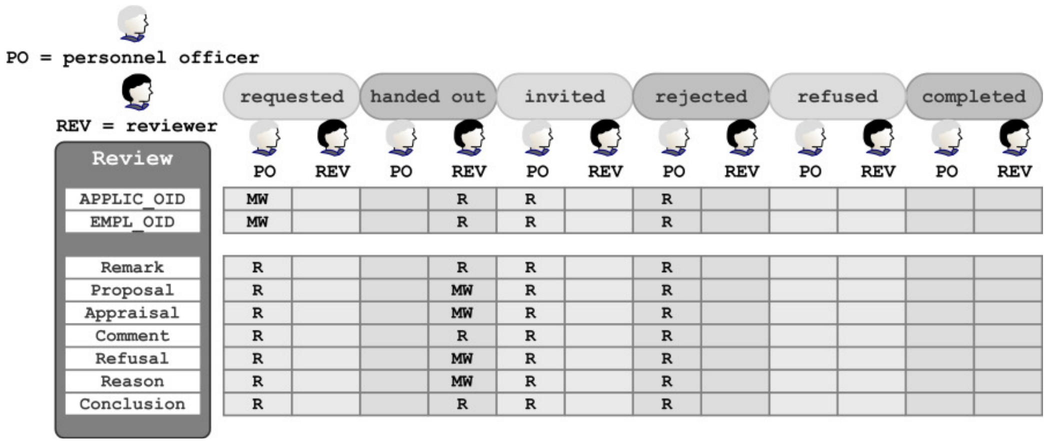
In order to support system designers, a sceleton of a micro process is automatically created for each object type. It comprises one start and one end state. While the latter includes a silent micro step, the former comprises one atomic micro step referring to the key attribute of the corresponding object type. Since attribute values for key attributes are automatically created, the first micro step is automatically reached during the creation of the corresponding micro process instance. Furthermore, micro processes are based on a well-defined operational semantics which is needed for their proper execution as well as for the automatic generation of the user interfaces (e.g., form-based activities, overview tables including object instances, worklists) relating to the runtime environment. Other concepts which are out of the scope of this paper include, for instance, the modeling and realization of backward jumps, the consideration of time events and the handling of runtime exceptions.
As already discussed, data authorization (cf. Section 6.2) and process authorization cannot be treated independently from each other. A suitable concept for integrating the permissions for reading/writing object data with user assignments relating to process execution is discussed in the following section.

6.4 Process and Data Authorization

In literature there exist several approaches that provide integrated access to business data and business processes [4,5,7,11,12,24,25,30]. However, these approaches do not consider the fact that (optional) access to process-related data often depends on the current progress of the respective process instances; i.e., whether or not a user may access certain object attributes also depends on the state of the corresponding micro process. Further, they do also not consider that the optional activities which enable read/write access to object instances may look different from user to user; i.e., different users may be allowed to read/write different attribute values in a certain state. Finally, another challenge to be tackled is to ensure that process authorization complies with corresponding data authorization and vice versa. As we will show, PHILharmonicFlows considers all these aspects.

6.4.1 Authorization Table

When considering the above mentioned interdependencies and constraints, one and the same activity might be mandatory for a particular user, while being optional for another one. Consequently, data and process authorization must be clearly differentiated as well. Generally, it should be possible that different users (i.e., user roles) may have different access rights on object attributes in a given micro process state. For this purpose, PHILharmonicFlows *automatically generates* a specific *authorization table* in accordance to the defined micro process type. As example consider Fig. 20, which depicts the authorization table that can be automatically generated from the micro process type depicted in Fig. 17. As illustrated in this figure, using an authorization table one can define which user role may *read write* which object attributes in the different states of a micro process instance **(cf. R10: data authorization)**. To ensure proper authorization, in PHILharmonicFlows each user who may be assigned to a micro state automatically obtains the permissions required for writing the corresponding object attributes.


PO = personnel officer
REV = reviewer

Review	requested		handed out		invited		rejected		refused		completed	
	PO	REV	PO	REV	PO	REV	PO	REV	PO	REV	PO	REV
APPLIC_OID	MW			R	R		R					
EMPL_OID	MW			R	R		R					
Remark	R			R	R		R					
Proposal	R			MW	R		R					
Appraisal	R			MW	R		R					
Comment	R			R	R		R					
Refusal	R			MW	R		R					
Reason	R			MW	R		R					
Conclusion	R			R	R		R					

Fig. 20: Automatically generated authorization table for the review object type

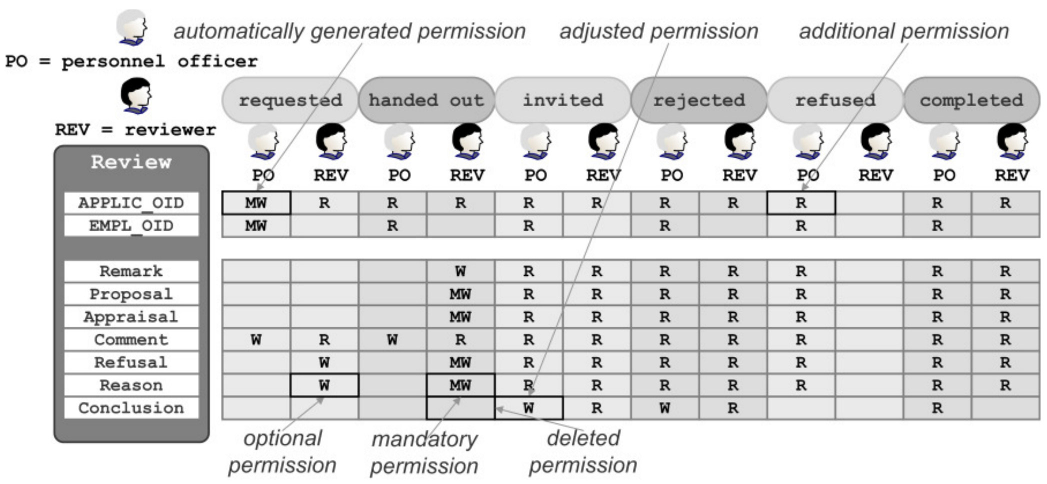
Example (Automatically assigned permissions): Since a user with role `reviewer` is assigned as responsible person to state `handed out` (cf. Fig. 20), she automatically has mandatory write permissions for all attributes corresponding to the micro steps of this state. Thus, she owns mandatory write permissions for attributes `proposal`, `appraisal`, `refusal`, and `reason`. Further, for all other attributes she owns read permissions in state `handed out`.

To enable privacy at a fine-grained level, we allow designers to adjust the originally generated authorization table as desired (cf. Fig. 21). During the execution of a particular mandatory activity, users should be able to optionally read / write additional object attributes event though these attributes are not related to micro states of the current micro state. To realize this feature, PHILharmonicFlows allows to extend the afore-created authorization table with additional read /write permissions for selected object attributes. Even if no mandatory activity is assigned to a user, he should be authorized to execute optional activities (if desired) by starting from the *data-oriented view*. Generally, this also enables users who are not involved in the execution of mandatory activities to read/write object attributes if desired. Since permissions can be also restricted to specific micro states, it becomes additionally possible to make access to optional activities dependent on the progress of the process instance **(cf. R16: mandatory and optional activities)**.
Generally, not every user who is allowed to write required attribute values in a particular micro state should be forced to also execute the corresponding mandatory activity. To be able to differentiate between user assignment (i.e., activities a user has to do) and authorization (i.e., activities a user may do) we further distinguish between *mandatory* and *optional permissions* in respect to writing object attributes. Only for users with mandatory permissions, a mandatory activity is assigned to their worklist **(cf. R11: differentiating authorization and user assignment)**.

Example (Adjusted permission): For the `reviewer`, a read permission was automatically generated for attribute `remark`. This permission can be changed into an optional write permission if required. Thus, in addition to her mandatory write permissions the `reviewer` is enabled to optionally write attribute `remark`.

Example (Additional permission): The `personnel officer` may optionally read `reviews` which have reached an end state.

Example (Delete permissions): Since the `reviewer` is assigned to state `handed out`, she automatically owns mandatory write/read permissions in this state. However, she is not allowed to read attribute `conclusion`. For this purpose, the respective permissions can be deleted after the generation of the authorization table.

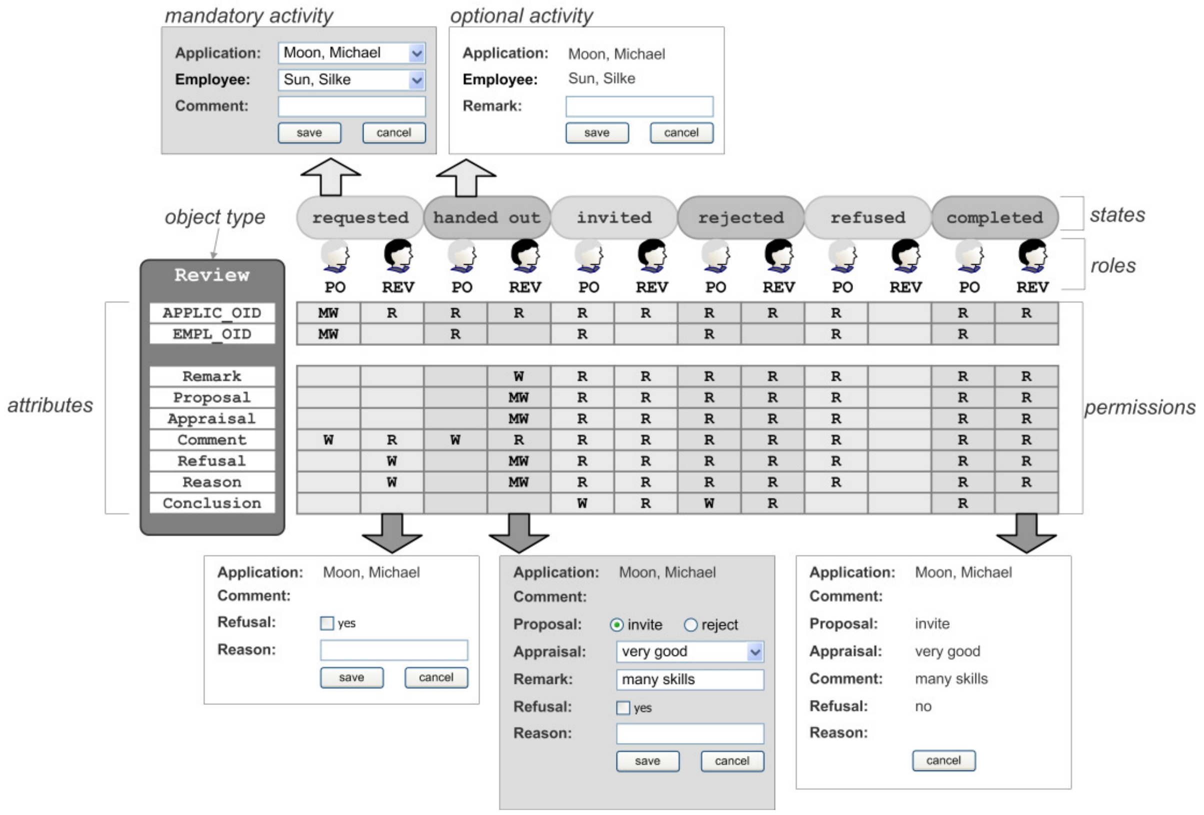


	requested		handed out		invited		rejected		refused		completed	
	PO	REV	PO	REV	PO	REV	PO	REV	PO	REV	PO	REV
APPLIC_OID	MW	R	R	R	R	R	R	R	R		R	R
EMPL_OID	MW		R		R		R		R		R	
Remark				W	R	R	R	R	R		R	R
Proposal				MW	R	R	R	R	R		R	R
Appraisal				MW	R	R	R	R	R		R	R
Comment	W	R	W	R	R	R	R	R	R		R	R
Refusal		W		MW	R	R	R	R	R		R	R
Reason		W		MW	R	R	R	R	R		R	R
Conclusion					W	R	W	R			R	

Fig. 21: Adjusted authorization table for the review object type

6.4.2 Generation of Form-based Activities

On the one hand a generated authorization table constitutes the basis for access control in respect to object-aware processes and relating data; on the other hand it provides the foundation for automatically and dynamically generating *user-specific activity forms* at runtime (cf. Fig. 22). A dynamically generated form contains (editable) input fields for those object attributes for which the respective user has the permission to write them in the current state of the micro process instance **(cf. R15: form-based activities)**. Similar to this, PHILharmonic Flows generates a *data-oriented view*, e.g., by providing overview tables for each object type listing its corresponding object instances. Based on respective tables, optional (form-based as well as black-box) activities may be invoked on an object instance **(cf. R8: data-oriented view)**.



mandatory activity:

Application:	Moon, Michael
Employee:	Sun, Silke
Comment:	

optional activity:

Application:	Moon, Michael
Employee:	Sun, Silke
Remark:	

	requested		handed out		invited		rejected		refused		completed	
	PO	REV	PO	REV	PO	REV	PO	REV	PO	REV	PO	REV
APPLIC_OID	MW	R	R	R	R	R	R	R	R		R	R
EMPL_OID	MW		R		R		R		R		R	
Remark				W	R	R	R	R	R		R	R
Proposal				MW	R	R	R	R	R		R	R
Appraisal				MW	R	R	R	R	R		R	R
Comment	W	R	W	R	R	R	R	R	R		R	R
Refusal		W		MW	R	R	R	R	R		R	R
Reason		W		MW	R	R	R	R	R		R	R
Conclusion					W	R	W	R			R	

Application:	Moon, Michael
Comment:	
Refusal:	☐ yes
Reason:	

Application:	Moon, Michael
Comment:	
Proposal:	[✓ invite] reject
Appraisal:	very good
Remark:	many skills
Refusal:	☐ yes
Reason:	

Application:	Moon, Michael
Comment:	
Proposal:	invite
Appraisal:	very good
Comment:	many skills
Refusal:	no
Reason:	

Fig. 22: Automatic generation of form-based activities in PHILharmonicFlows

Since for object type `review` six different states are defined (cf. Fig. 22), in principle, for each involved user role six different form-based activities can be generated.

As another fundamental contribution of PHILharmonicFlows, process instances and activities in which authorized users are involved are not strictly linked to each other. In particular, it is also possible to execute a particular activity (i.e., a certain form) in relation to a collection of object instances of the same object type (i.e., *batch activity*). Then, entered attribute values are assigned to all selected object instances in one go. Further, a user may invoke additional object instances of different (i.e., related) types; i.e., he may want to perform a *context-*

sensitive activity (cf. Section 3). For generating the corresponding activity, the currently enabled micro states of the considered object instances as well as the permissions assigned to the users in these particular states are considered **(cf. R17: variable granularity)**.

Example (Batch activity): An `employee` may reject (i.e., assign attribute `decision`) a set of `applications` in one go.

Example (Context-sensitive activity): Consider an activity involving one `application` object instance and one `job` object instance. Here, an `employee` may hire an `applicant` and fill the `job` position, while executing one activity. In particular, it is possible to assign a value to attribute `decision` of an `application` object instance and a value for attribute `occupied` of a job object type in one go.

This way, a very flexible execution of activities is enabled. Since object type `review` has 6 different states and 2 involved user roles, in total, 12 different instance-specific activities may be generated. Further on, since an `application` object instance comprises 5 states, there exist 120 different context-sensitive activities for a `review` including its corresponding `application`.
However, whether or not a particular state of a micro process instance can be reached (i.e., which activities can be actually executed) also depends on the progress of other (relating) micro process instances. More precisely, the execution of a micro process instance must be blocked in certain situations. The handling of such interdependencies between different micro process instances constitutes a fundamental feature of the PHILharmonicFlows approach and is introduced in the following section.

6.5 Macro Processes

Generally, business processes may involve multiple object instances of the same and of different type. In PHILharmonicFlows such multi-object processes can be modeled in terms of *macro processes*. As opposed to a micro process, which defines *object behavior* (i.e., the behavior of object instances relating to a particular object type), a macro process enables the specification of *object interactions*. Thus, a clear separation of two different abstraction layers for process modeling is provided.
As discussed individual object instances may be related to each other and are coupled with a corresponding micro process instance. Consequently, at runtime a complex process structure emerges which covers multiple object instances and their interactions. Existing approaches enable process coordination either without considering the underlying data structure [1] or solely based on direct object relations [19,20,21,22]. The latter means that coordination support is only provided for object instances that are directly related to each other; e.g., the processing of an `application` object may be coordinated with the one of a corresponding `job` object.
Opposed to this PHILharmonicFlows explicitly considers the diverse relationships that may exist between different object instances. This means that micro process instances can be coordinated even if no direct relation between the corresponding object instances exists; e.g., micro process instances related to `interview` objects may be coordinated with the ones corresponding to `review` objects. In this context, a particular challenge is to enable such a flexible coordination of micro process instances, while their execution is based on a data-driven paradigm. Thereby, the dynamically evolving number of object instances as well as their asynchronous execution have to be taken into account. In particular, micro processes as well as macro processes are defined and executed based on data; i.e., object types, object attributes, and object states. Note that this turns the integration of black-box-activities also into a challenging task.

An actual process structure may comprise dozens up to hundreds of micro process instances together with their numerous interdependencies [20]. To hide its complexity from process modelers and end-users, PHILharmonicFlows allows to define *macro process types* in a "flat" and very compact way.
As illustrated in Fig. 23, at the type level a *macro process* consists of several *macro steps* and *macro transitions*. As opposed to traditional process modeling approaches, where process steps are defined in terms of activities, a macro step *ot(s)* always refers to an object type *ot* together with a corresponding state *s*. In contrast to micro processes, which are defined based on attributes, a macro process, in turn, is defined based on object types. In this context, the *states* as defined within micro process definitions play an important role; i.e., within a macro process definition they provide the basis for coordination. At runtime, a particular *macro step ot(s)* then represents those object instances of type *ot* whose corresponding micro process instance is in state *s*. To express the dependencies between micro states belonging to micro process instances of different type, individual macro steps can be connected with each other using *macro transitions* **(cf. R5: object interactions)**. As example consider the macro transition describing a dependency between `job offers` in state `published` and `applications` in state `sent`.
Generally, both parallel and alternative execution paths can be defined within a macro process. More precisely, each macro step may comprise a number of *macro inputs*. Each macro transition then connects a macro step with one macro input of a subsequent macro step. Generally, more than one macro transition may point to a particular macro input. Whether or not the particular macro step can be reached during runtime then depends on the activation of its macro inputs. More precisely, a macro step is only reachable if at least one of its macro inputs becomes activated: i.e., all incoming macro transitions of a macro input are fired. Based on this, alternative execu-

tion paths can be defined using several macro inputs; i.e., OR-semantics is enabled. To enable parallel execution, in turn, several macro transitions can be connected to one macro input; i.e., AND-semantics is enabled.

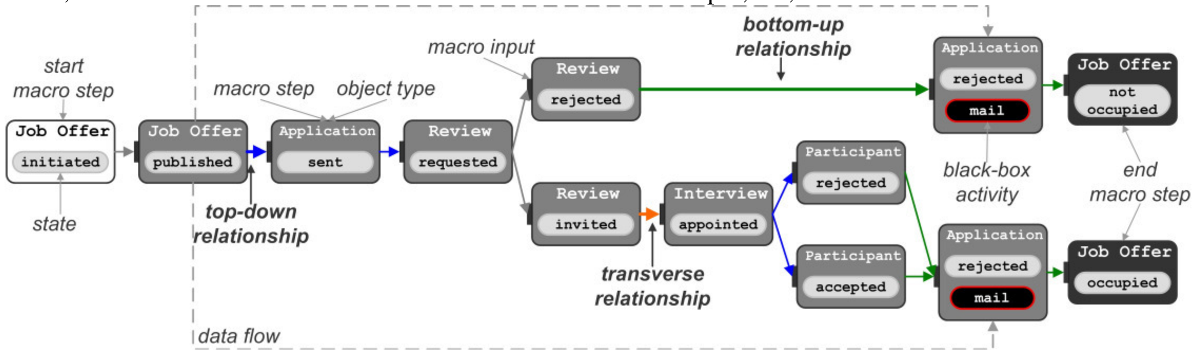


Fig. 23: Example of a macro process as modeled in PHILharmonicFlows

Example (Macro process): Fig. 23 depicts an example of a macro process. At first, the `job offer` must be `initiated` and `published`. After that `applicants` can `send` their corresponding `applications`. Following this, a `personnel officer` needs to `request reviews` for each `application`. If all `reviewers` recommend to `reject` the `applicant` the `application` will be `rejected`. In turn, if at least one `reviewer` proposes to `invite` the `applicant` an `interview` will be arranged. The subsequent activities depend on the proposed actions of the `participants`. The concrete semantics of the depicted kinds of relationships is explained below.

Whether or not the particular state relating to the subsequent macro step can be reached at runtime depends on the respective state of one (or more) object instances relating to the previous macro step. In this context, the relationship between the object type of the source macro step and the one of the target macro step have to be taken into account. Respective relationships can be automatically determined based on the given data structure (cf. Section 6.2). Depending on the concrete type of relationship, a specific coordination component must be defined for each macro transition. More precisely, while for each top-down relationship (cf. Section 6.2) a *process context* has to be specified, bottom-up relationships (cf. Section 6.2) require an *aggregation*. For relationships of type *transverse*, in turn, a corresponding *transverse* coordination component should be available. We explain this in more detail in the following.

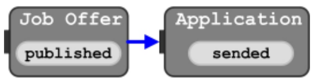


Fig. 24: Macro transition requiring a process context

As example for a process context consider the macro transition between `published job offers` and `sent applications` (cf. Fig. 24).

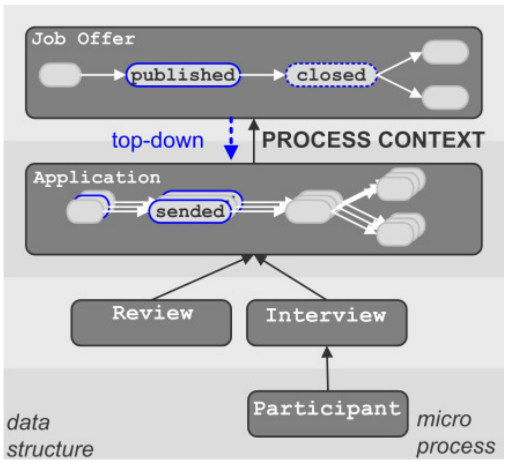


Fig. 25: Process Context

Using a *process context*, the execution of several lower-level micro process instances can be coordinated with the one of a higher-level micro process instance (cf. Fig. 25). In particular, whether or not the state of a lower-level micro process instance that corresponds to the target macro step can be reached depends on one (or more) states of the higher-level micro process instance. Each process context contains at least the state relating to the source macro step. However, additional states can be added to a process context if required. This way, a more asynchronous execution of different micro process instances is enabled.

Example (Process context): An `applicant` may only `send` an `application` if the relating `job offer` is either `published` or `running` (cf. Fig. 25).

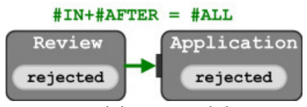


Fig. 26: Macro transition requiring an aggregation

As example of an aggregation, consider the macro transition between `rejected reviews` and `rejected applications` (cf. Fig. 26).

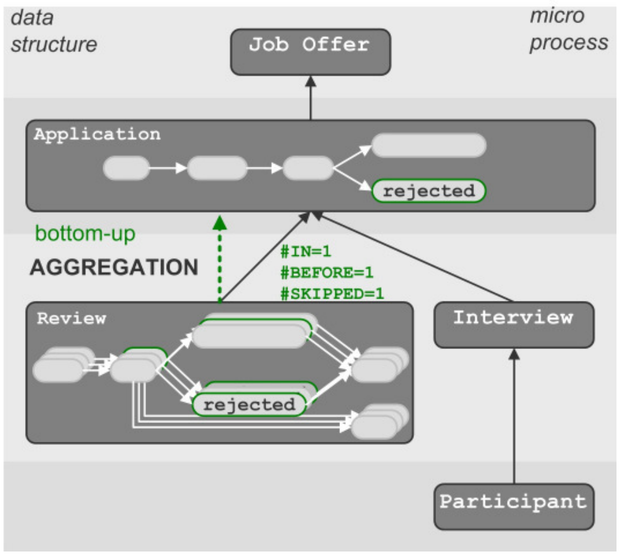


Fig. 27: Aggregation

Aggregations constitute another fundamental concept for coordinating micro process instances. When compared to the aforementioned process context, aggregations work the other way around; i.e., one higher-level micro process instance is coordinated with a number of relating lower-level micro process instances.
As illustrated in Fig. 27, whether or not the state of each micro process instance corresponding to the target macro step may be enabled depends on its lower level micro processes instances. More precisely, the execution of the higher-level micro process instance depends on the states of the lower-level micro process instances which correspond to the source macro step (see below example). For this purpose, PHILharmonicFlows provides an aggregated view on the lower-level micro process instances. In order to get an overview on how far the lower-lever process instances have proceeded in respect to the state belonging to the source macro step, each macro step is associated with a number of counters (cf. Tab. 1).

#ALL	Total number of corresponding lower-level micro process instances.
#BEFORE	Number of corresponding lower-level micro process instances for which the state specified in the source macro step is not enabled. However, it is still possible to enable this state later on.
#IN	Number of corresponding lower-level micro process instances for which the state specified in the source macro step is enabled.
#AFTER	Number of corresponding lower-level micro process instances for which the state specified in the source macro step is not enabled. However, the state was enabled before.
#SKIPPED	Number of corresponding lower-level micro process instances for which the state specified in the source macro step is not enabled. Here, it is not possible to enable the state later on; i.e., an alternative execution path was taken.

Tab. 1: Counters of an aggregation

Example (Aggregation): An `application` may only be `rejected` if all `reviewers` have proposed to `reject` the `applicant`. In this context `completed reviews` corresponding to an `application` are considered as well.

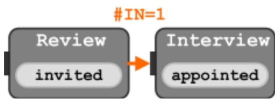


Fig. 28: Macro transition requiring a transverse

Finally, as example of a transverse coordination component, consider the macro transition between `invited reviews` and `appointed interviews` (cf. Fig. 28).

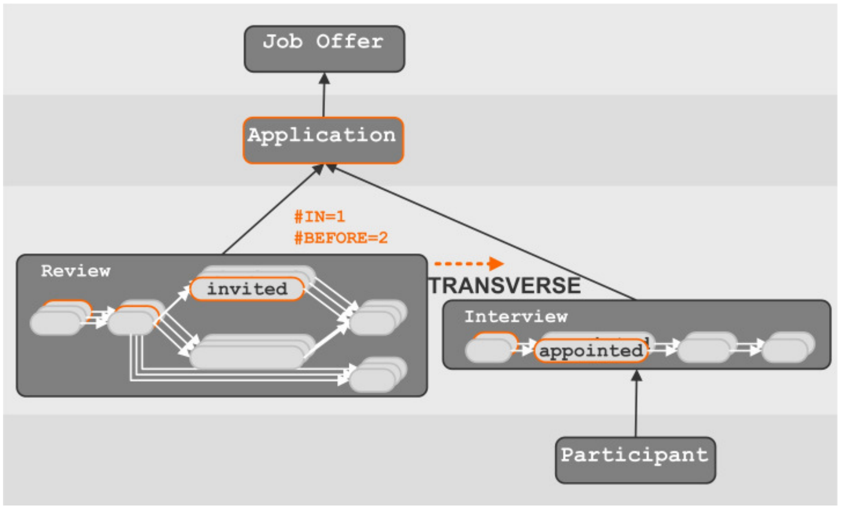


Fig. 29: Transverse

As last kind of coordination support components we consider *transverse* coordination components. They describe a dependency between a micro process instance (which corresponds to the target macro step) and a number of micro process instances (which correspond to the source macro step) (cf. Fig. 29). For this purpose, an aggregation is evaluated in respect to a common higher-level micro process instance.

Example (Transverse): It is only possible to `appoint` a date for an `interview` if at least one `review` proposes `invitation` of the `applicant`.

As illustrated in Fig. 23, a black-box activity with multiple object instances as input can be also included in a macro process definition by assigning it to a macro step **(cf. R14 black-box activities)**. A black-box activity enables complex computations as well as the integration of advanced functionalities (e.g., sending e-mails or invoking web services). Here, our approach provides additional advantages in respect to the implementation of business functions. In particular, it becomes possible to additionally consider the data relationships between the different input parameters (i.e., object instances) of an activity and business function respectively. This enables the implementation of more flexible business functions. As example consider the data flow dependency as illustrated in Fig. 23.

Example (Black-box activity): If an `application` is `rejected` or `accepted` the corresponding `applicant` receives a notification `mail`. This `mail` should include information about the corresponding `job offer` to which the `applicant` applied (cf. Fig. 23).

Generally, during the execution of a large process structure deadlock situations might occur. For example, consider a process context which includes states that belong to an alternative path. Since individual micro process instances are executed asynchronously, it might be further possible that aggregation predicates never evaluate to true. Consequently, deadlock situations cannot be prevented. However, we do not exclude such modeling because some real world scenarios require respective coordination constraints. In particular, deadlocks occur in real life situations as well. For this reason, PHILharmonicFlows provides a detection algorithm for identifying newly emerged deadlocks; this algorithm is based on known deadpath elimination techniques. Further, PHILharmonicFlows provides monitoring facilities to visualize deadlocks to users and to assist them in dissolving them.

Example (Deadlock): If all `reviewers refuse` to fill in the `review`, the execution of the `application` micro process instance is blocked.

If a higher-level micro process instance terminates (i.e., reaches an end state) all lower-level micro process instances, which still have not reached an end state, are determined. We denote these as *bypassed micro process instances*. For these micro process instances several exception handling possibilities are provided; e.g., the micro process instances can be skipped or be assigned to another higher-level micro process instance. This way, correct termination of all micro process instances within the process structure can be ensured.

Example (Bypassed micro process instances): If an `application` is `accepted` the corresponding `job offer` is `occupied`. In this situation, all other `applications` belonging to the `job offer` must be determined and be handled in a specific way.

Altogether, a macro process enables the modeling of large business processes, but abstracts from the complex process structure underlying them. Using the introduced counters and abstraction mechanisms, at runtime a macro process serves as a sophisticated monitoring facility which provides an abstracted view of the complex process structure under execution. Finally, whether or not mandatory activities from a particular micro process instance will be actually assigned to user worklists does not only depend on the progress of this micro process instance, but also on the coordination components in which this micro process instance is involved.

6.6 Summary

With PHILharmonicFlows we provide a comprehensive framework that covers the major characteristics of object-aware processes: object behavior, object interactions, data-driven process execution, and integrated access to processes and data. As illustrated in Table 2, each of the requirements identified in Section 5 is met. While Requirement R4 (i.e., object behavior), for example, is supported through micro process modeling, Requirement R5 (i.e., object interactions) is covered by the concept of macro processes. A data-driven process execution (cf. Requirement R6) is provided through micro steps and related object attribute changes. Since states are based on micro steps the execution of macro processes is data-driven as well.

Data		
R1	Data integration	integrated data model including object types, relations and attributes
R2	Cardinalities	definition of minimal and a maximal cardinalities consideration of the dynamic number of object instances at runtime
R3	Mandatory information	macro processes and cardinalities define required object instances micro processes define required attribute values
Processes		
R4	Object Behavior	micro processes
R5	Object interactions	macro processes
R6	Data-driven execution	micro steps within micro processes; macro steps using states
Users		
R7	Process-oriented view	automatic generation of worklists at runtime
R8	Data-oriented view	automatic generation of overview tables at runtime
R9	Process authorization	user assignment based on states and explicit micro transitions mandatory permissions maintained in authorization table

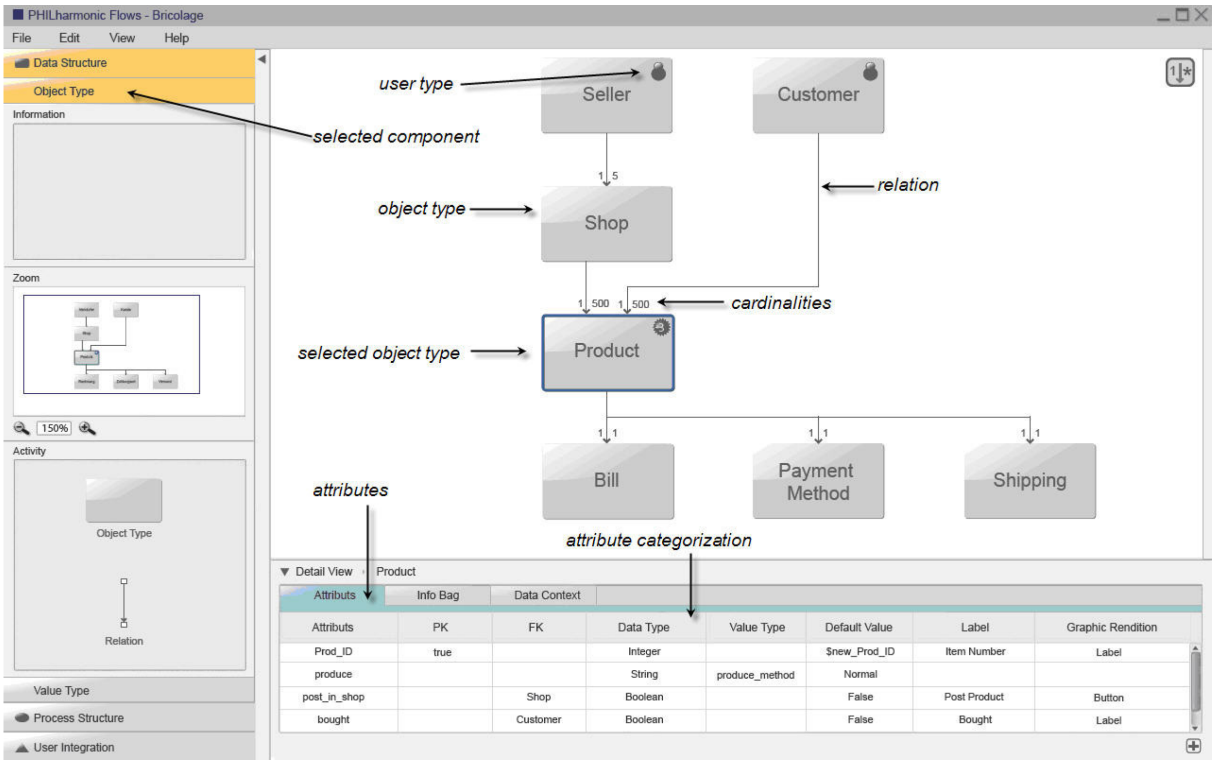
R10	Data authorization	state-based permissions for creating and deleting object instances as well as for reading/ writing attribute values
R11	differentiation of R9/R10	differentiation between mandatory and optional permissions
R12	Vertical authorization	integration of application data and user administration instance-specific user roles
R13	User decisions	explicit micro transitions user may freely choose the desired number of object instances (as long as cardinality constraints are fufilled)
Activities		
R14	Black-box activities	assignment to states within micro and macro processes
R15	Form-based activities	automatic generation based on attributes, permissions and states
R16	mandatory and optional activities	differentiation between mandatory and optional permissions
R17	variable granularity	automatic generation of form-based activities including different sets of object instances
R18	internal control-flow	micro steps and internal micro transitions
R19	re-execution	data-driven execution and explicit external micro transitions

Tab. 2: Requirements and Supporting Features in PHILharmonicFlows

7. Proof-of Concept

This section gives some impressions on our proof-of-concept prototype and illustrates its use along a characteristic order handing process. Furthermore, we introduce additional use cases relating to house building and vacation requests. Finally, we discuss the benefits of our framework as well as lessons learned along these process scenarios.

In order to evaluate the presented concepts we prototypically implemented both the modeling and the runtime environment of PHILharmonicFlows. As first use case consider "Bricolage", an online shop which enables business owners to sell their products. The considered process includes several steps ranging from the selling of products to their delivery to customers. Fig. 30 depicts the data model for this use case as it can be created with the modeling environment of PHILharmonicFlows. Altogether this model consists of seven object types. Furthermore, the micro process for object type `product` (cf. Fig. 31) comprises several states for adding a product to the shop, for ordering this product, and for shipping it to the customer. Besides this, the cancellation of an order is considered in the micro process as well. Altogether the developed concepts have proven to be well suited for covering the Bricolage use case.



Attributs	PK	FK	Data Type	Value Type	Default Value	Label	Graphic Rendition
Prod_ID	true		Integer		$new_Prod_ID	Item Number	Label
produce			String	produce_method	Normal		
post_in_shop		Shop	Boolean		False	Post Product	Button
bought		Customer	Boolean		False	Bought	Label

Fig. 30: Data model for "Bricolage" in PHILharmonicFlows

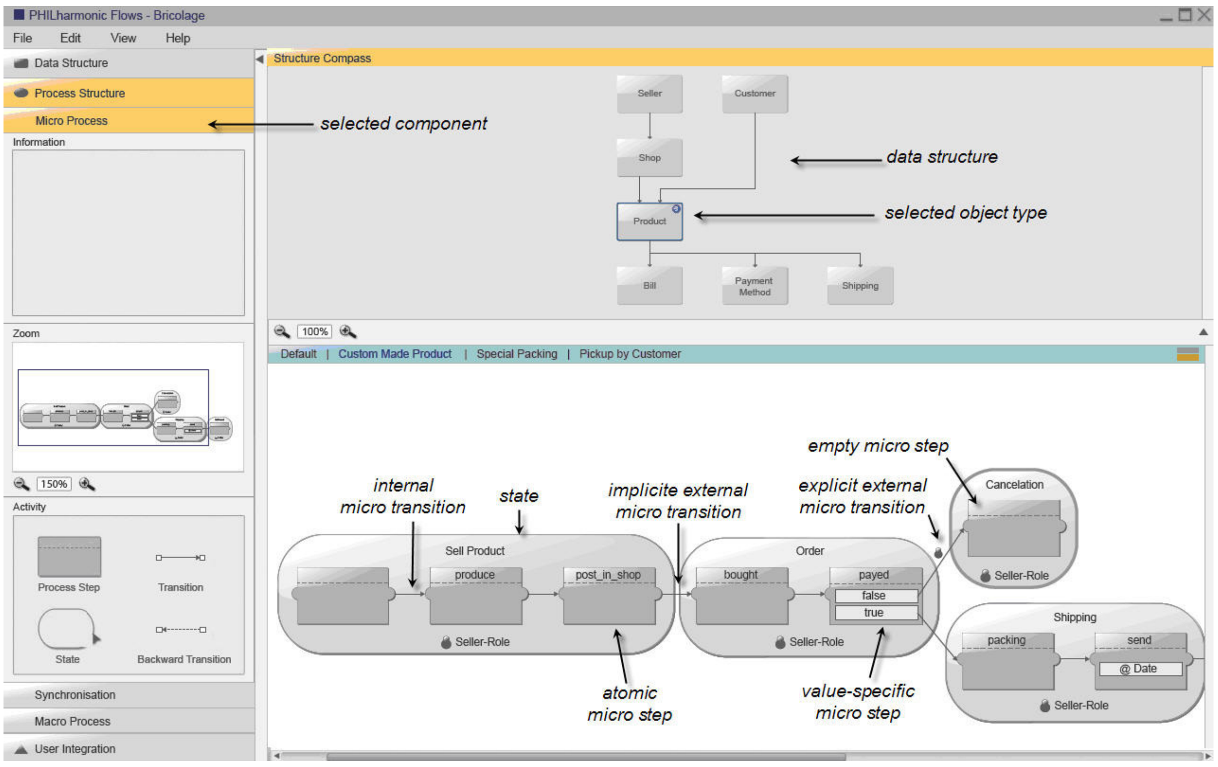


Fig. 31: Product micro process for "Bricolage" in PHILharmonicFlows

In order to further evaluate the developed concepts, we applied them to two additional cases – a house building process and a process for handling vacation requests. Since the former case deals with a complex, long running process that involves numerous object types, we use it for evaluating the modeling of macro processes. The latter case relates to vacation requests, i.e., it deals with short running processes. Therefore, we use it to provide a more sophisticated example concerning the modeling of micro processes.

We first consider the house building process. Fig. 32 and Fig. 33 depict extracts of the data model as well as the macro process we created for this case.

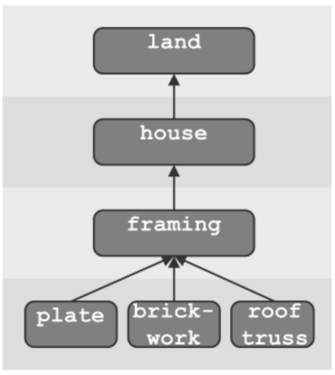


Fig. 32: Part of the data model for house building

As illustrated in Fig. 32, each house is built on a parcel of land. It further consists of a framing which comprises a base plate, brickwork and a roof truss. Regarding the corresponding macro process in Fig. 33, the order in which corresponding object instances are created depends on their relations to each other. The first macro transitions refer to top-down relationships. The subparts belonging to the framing must be constructed in sequence within a narrow time frame; i.e., the construction of a particular subpart depends on the construction of other subparts (of different type). In particular, a bricklayer may only start his work when the plate is dry. Consequently, the macro transitions which connect the macro steps relating to these individual sub parts refer to transverse relationships. Finally, whether or not the framing is finished depends on the completion of its subparts. For this purpose, the last macro transition refers to a bottom-up relationship.

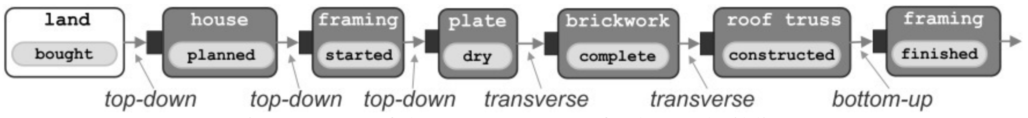


Fig. 33: Part of the macro process for house building

As another proof-of-concept consider the micro process for a vacation request as illustrated in Fig. 34. An employee applying for a leave has to specify a start as well as an end date. In this context, two micro steps are introduced to ensure that the start data lies in the future and the end data follows the start date; both micro steps are assigned to the same state. Following this, the employee submits her request to her substitute who has to decide whether he is willing to take over. If he agrees the request is submitted to the manager of the employee. She has to decide whether to agree with the request or to refuse it. If the substitute is not willing to take over the employee may decide whether to re-submit the request or to cancel it. The latter is also allowed for already agreed upon requests.

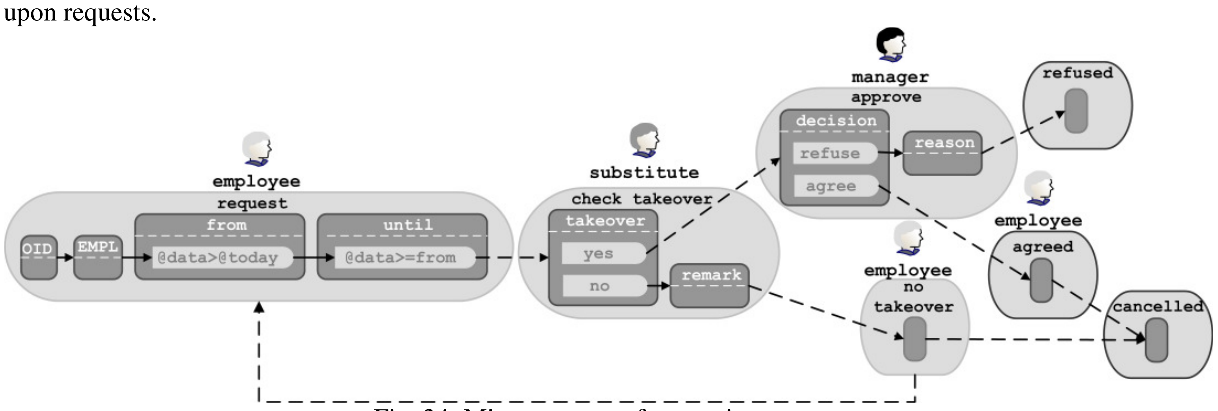


Fig. 34: Micro process of a vacation request

Applying our proof-of-concept prototype and the developed concepts to the above process scenarios has proven that our basic modeling approach works well. Amongst others it confirmed that the unambiguous granularity of processes enables a clear modeling methodology. The modeling of micro processes, which comprise micro steps defined in terms of object attributes, enabled the required data-based modeling. Here, advanced features like time events were supported as well. The ordering of attributes as well as their grouping to states also turned out to be very intuitive for modelers.
Process coordination based on object types and their corresponding states has proven to be extremely useful as well. In particular, it provided an adequate and comprehensible level of abstraction to users. The "flat" way of modeling processes additionally facilitated modeling. Finally, the different kinds of relationships offered in respect to process coordination contributed to hide the complexity of the actual process structure from both modelers and end users.
At runtime, the integrated view on processes and data offers numerous benefits. For example, state-based process monitoring enables a more natural and intuitive view on business processes for end users. Since activities are not pre-fixed and are not rigidly associated with one process, a more flexible process execution becomes possible. First, through the execution of optional activities authorized users may accomplish certain actions up front, i.e., before they are mandatorily required during process execution. Similarly, completed activities may be re-executed if desired. Second, batch activities improve the processing of a large number of object instances.
However, our evaluation has also revealed some limitations of our approach to be tackled in future work. For example, the historization of attribute values and state changes was considered as important feature for enabling proper process traceability. Further, different specializations of an object type may require different models for related micro processes; i.e., we must cope with process variants. For example, consider hand-made products and mechanically produced ones which require different process definitions. Other scenarios, in turn, required an overlap and synchronization of different macro process instances. As example consider the interdependencies between selling procedures for products and warehousing processes. Further, at this stage of our project we have focused on the modeling and the execution phase in process lifecycle (including exception handling). Additional challenges, however, emerge from the support of ad-hoc changes during runtime as well as from schema evolution. Regarding the latter, the evolution of data models needs to be compliant with the one of the respective process models.

8. Summary and Outlook

In this paper we made several contributions. First, we provided a systematic analysis of processes, which are currently not adequately supported by existing PrMS. Second, we elaborated the properties of these processes and elicited major requirements for making them object-aware. Third, we presented the PHILharmonicFlows framework for object-aware process management and evaluated it along several process scenarios.
PHILharmonicFlows supports the definition of data and processes in separate, but well integrated models. Thus, it retains the well established principle for separation of concerns. As opposed to existing approaches it explicitly considers the relationships between processes, functions, data and users. Furthermore, PHILharmonicFlows addresses process modeling, execution and monitoring, and it provides generic functions for the model-driven generation of end user components (e.g., form-based activities). Opposed to related work on process and data integration, we consider all components of the underlying data structure; i.e., objects, relations and attributes. For this purpose, we enable the modeling of processes at different levels of granularity. In particular, we combine object behavior based on states with data-driven process execution. Further, we provide advanced support for process coordination as well as for the integrated access to business processes, business functions and business data. We believe that the described framework offers promising perspectives for overcoming many of the limitations of contemporary PrMS.

In our future work we will elaborate on formal properties and correctness notions in respect to object-aware processes. Furthermore, we will extend our framework by addressing additional issues like historization, traceability, process variability, and process flexibility (e.g., to enable ad-hoc changes and schema evolution).

References

[1] W.M.P. van der Aalst, P. Barthelmess, C. Ellis, J.Wainer: *Workflow Modeling using Proclets*, In Proc. CoopIS'00, LNCS 1901, pp.198-209, 2000

[2] W.M.P. van der Aalst, K. van Hee: *Workflow-Management - Models, Methods and Systems*, MIT Press, 2004

[3] W.M.P. van der Aalst, M. Pesic, H. Schonenberg: *Declarative workflows: Balancing between flexibility and support*, Computer Science - Research and Development, 23(2):99-113, 2009

[4] W.M.P. van der Aalst, M. Weske, D. Grünbauer: *Case Handling: A new Paradigm for Business Process Support*, Data and Knowledge Engineering, 53(2):129-162, 2005

[5] J. Barkley, K. Beznosov, J. Uppal: *Supporting Relationships in Access Control Using Role Based Access Control*, In Proc. RBAC'99, pp.55-65, 1999

[6] K. Bhattacharya, R. Hull, J. Su: *A Data-Centric Design Methodology for Business Processes,* Handbook of Research on Business Process Management, 2009

[7] R.A. Botha: *CoSAWoE - A Model for Context-sensitive Access Control in Workflow Environments*, PhD thesis, Rand Afrikaans University, 2002

[8] B. Mutschler, M. Reichert, J. Bumiller: *Unleashing the Effectiveness of Process-oriented Information Systems: Problem Analysis, Critical Success Factors and Implications.* IEEE ToSMC, 38(3):280-291, 2008

[9] E. Dijkstra: *A Discipline of Programming*, Prentice-Hall, 1976.

[10] D. Dori: *Object-Process Methodology*, Springer, 2002

[11] J. Hu, A. Weaver: *A Dynamic, Context-Aware Security Infrastructure for Distributed Healthcare Applications*, In Proc. PSPT'04, 2004

[12] A. Kumar, N. Karnik, G. Chafle: *Context Sensitivity in Role-based Access Control*, SIGOPS, 36(3), pp.53-66, 2002

[13] V. Künzle, M. Reichert: *Towards Object-aware Process Management Systems: Issues, Challenges, Benefits*, In Proc. BPMDS'09, LNBIP 29, pp.197-210, 2009

[14] V. Künzle, M. Reichert: *Integrating Users in Object-aware Process Management Systems: Issues and Challenges*, In Proc. BPD'09, LNBIP 43, pp. 29-41, 2009

[15] V. Künzle, M. Reichert: *Object-aware Business Processes: Properties, Requirements, Existing Approaches*, Technical Report UIB-2010-06, University of Ulm, 2010

[16] J. Küster, K. Ryndina, H. Gall: *Generation of Business Process Models for Object Life Cycle Compliance*, In Proc. BPM'07, LNCS 4714, pp.165-181, 2007

[17] R. Liu, K. Bhattacharya, F. Y. Wu: *Modeling Business Contexture and Behavior Using Business Artifacts*, Advanced Information Systems Engineering, Vol.4495, p.324-339, 2007

[18] F. Leymann, D. Roller: *Production workflow: concepts & techniques*, Prentice Hall, 2000

[19] D. Müller, M. Reichert, J. Herbst: *Data-driven Modeling and Coordination of Large Process Structures*, In Proc. CoopIS'07, LNCS 4803, pp.131-149, 2007

[20] D. Müller, M. Reichert, J. Herbst: *A New Paradigm for the Enactment and Dynamic Adaptation of Data-driven Process Structures*, In Proc. CAiSE '08, LNCS 5074, pp.48-63, 2008

[21] G.M. Redding, M. Dumas, A. H. M. ter Hofstede, A. Iordachescu: *Transforming Object-oriented Models to Process-oriented Models*, In Proc. BPM'07, LNCS 4928, pp.132-143, 2008

[22] G.M. Redding, M. Dumas, A. ter Hofstede, A. Iordachescu: *A flexible, object-centric approach for business process modelling*, Service Oriented Computing and Applications, pp. 1-11, Springer, 2009

[23] H. A. Reijers, S. Liman, W. M. P. van der Aalst: *Product-based Workflow Design, Management Information Systems*, 20(1): 229-262, 2003

[24] M. Rosemann, M. zur Mühlen: *Modellierung der Aufbauorganisation in Workflow-Management-Systemen: Kritische Bestandsaufnahme und Gestaltungsvorschläge*, EMISA-Forum, 3(1):78-86, 1998

[25] M. Rosemann, M. zur Mühlen: *Organizational Management in Workflow Applications: Issues and Perspectives*, Inf. Technol. and Management, 5(3-4):271-291, 2004

[26] B. Silver: *Case Management: Addressing Unique BPM Requirements*. BPMS Watch, Industry Trend Reports, 2009

[27] N. Schroeder, U. Spinola, J. Becker: *SAP Records Management*, SAP PRESS, 2009

[28] S. Wu, A. Sheth, J. Miller, Z. Luo: *Authorization and Access Control Of Application Data in Workflow-Systems*, JIIS, 18(1): 71-94, 2002

[29] S.W. Sadiq, M.E. Orlowska, W. Sadiq, K. Schulz: *When workflows will not deliver: The case of contradicting work practice*, In Proc. BIS'05, pp. 69-84, 2005

[30] R.S. Sandhu, R. K. Thomas: *Task-based Authorization Controls (TBAC): A Family of Models for Active and Enterprise-oriented Authorization Management*, In Proc. IFIP'97, pp.166-181, 1997

[31] I. Vanderfeesten, H. A. Reijers, W. M. P. van der Aalst: *Product-based Workflow Support: Dynamic Workflow Execution*, In Proc. CAiSE '08, LNCS 5074, pp.571-574, 2008

[32] M. Weske: *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007

[33] B. Weber, M. Reichert, W. Wild, S. Rinderle-Ma: *Providing Integrated Life Cycle Support in Process-Aware Information Systems.* IJCIS , 18(1):115-165, 2009