A Modeling Paradigm for Integrating Processes and Data at the Micro Level

Vera Künzle[1] and Manfred Reichert[1]

Institute of Databases and Information Systems, Ulm University, Germany
{vera.kuenzle,manfred.reichert}@uni-ulm.de

Abstract. Despite the widespread adoption of BPM, there exist many business processes not adequately supported by existing BPM technology. In previous work we reported on the properties of these processes. As a major insight we learned that, in accordance to the data model comprising object types and object relations, the modeling and execution of processes can be based on two levels of granularity: object behavior and object interactions. This paper focuses on micro processes capturing object behavior and constituting a fundamental pillar of our framework for object-aware process management. Our approach applies the well established concept of modeling object behavior in terms of states and state transitions. Opposed to existing work, we establish a mapping between attribute values and objects states to ensure compliance between them. Finally, we provide a well-defined operational semantics enabling the automatic and dynamic generation of most end-user components at run-time (e.g., overview tables and user forms).

Key words: Object-aware Processes, Data-driven Process Execution

1 Introduction

Process Management Systems (PrMS) enable the modeling, execution and monitoring of business processes [1]. Despite their widespread adoption, there exist many knowledge-intensive processes which cannot be "straight-jacketed into activities" [2, 3]. Prescribing an activity-centred process model for them would lead to a "contradiction between the way processes can be modeled and the preferred work practice" [4]. Moreover, PrMS do not provide integrated access to application data. In particular, end-users cannot access application data at any point in time (assuming proper authorization). In this context, overview tables (e.g., data reports) and user forms constitute important components. The latter provide (data) input fields (e.g., textfields, checkboxes) for reading and writing selected attribute values of object instances. Further, many activities of a process model are implemented as forms. As known from practice, however, implementing the logic of these forms and other user components causes high implementation efforts.

In the PHILharmonicFlows[1] project, we are developing concepts, methods and tools for realizing object- and process-aware information systems [5]. In particular, we are targeting at a flexible integration of business data, business processes, business functions, and users to overcome limitations known from activity-centered PrMS. In addition, we aim at the automatic generation of end-user components; e.g., tables giving an overview on a collection of object instances and form-based activities (including their internal logic). This way, not only generic process support, but also generated application functionality shall be provided.

This paper introduces a fundamental pillar of our PHILharmonicFlows framework by introducing an advanced paradigm for the modeling and run-time support of object behavior, i.e., the processing of individual object instances. The latter provides the foundation of object-aware processes involving multiple object instances of the same and of different object type. Like existing work considering object behavior during process execution [6, 7, 8, 9, 10, 11, 12, 13] our approach applies the well established concept of modeling object behavior in terms of states and state transitions. Opposed to existing approaches (e.g., case handling), however, PHILharmonicFlows enables a mapping between attribute values and objects states and therefore ensures compliance between them. In addition, integrated access to application data is provided. Here, not only generic process support, but also generated application functionality is provided. Finally, the presented execution paradigm combines data-driven process execution with activity-oriented aspects.

Section 2 illustrates the research methodology we applied and discusses major requirements for the modeling and run-time support of object behavior. An overview of our framework is given in Section 3. We introduce its underlying data model in Section 4 and the modeling of object behavior in Section 5. Section 6 deals with authorization issues targeting at the automatic generation of (form-based) activities at run-time. The corresponding execution paradigm is discussed in Section 7. Section 8 investigates related work and Section 9 closes with a summary and outlook.

2 Research Methodology

To better understand the characteristics of processes that are well supported by existing technology and those handled insufficiently, we analyzed many processes from domains like healthcare, human resource management, and automotive engineering [14, 15, 5]. As fundamental insight we gained from these studies that many processes require *object-awareness*; i.e., full integration of application data consisting of object types, object attributes, and object relations. In previous work we identified the properties of these processes [5, 16] and discussed challenges to be tackled for integrating processes, data and users [14, 15]. A major finding was that there are strong relationships between process support and data

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

management. In accordance to the data model comprising *object types* and *object relations*, therefore, the modeling and execution of processes is based on two levels of granularity as well: *object behavior* and *object interactions*. Regarding object behavior the following properties are significant: For each *object instance* a corresponding *process instance* should exist controlling its processing. *Object attribute values* reflect the progress of this process instance. While certain attribute values can be optionally assigned, others are mandatorily required in order to reach a particular process goal. For this purpose, *mandatory activities* need to be enabled and assigned to responsible users if required information is missing. In addition, *optional activities* for reading and writing attribute values at any point in time should be supported. Generally, one has to ensure that *object state* and *process state* are compliant with each other. Further, it should be possible to enter data at the moment it becomes available (i.e., using optional activities). In particular, users should be allowed to enter data up-front; i.e., before the corresponding mandatory activity becomes enabled. For this purpose, it should be possible to drive process execution based on data and to dynamically react upon attribute value changes. Mandatory activities no longer needed (due to an up-front data entry) should then be automatically skipped. Moreover, users should be enabled to re-execute activities until they explicitly commit their completion. Generally, different ways for reaching a process goal exist. In our context, this selection might be also based on explicit user decisions. When filling in forms, certain attribute values might become mandatory on-the-fly; i.e., whether or not an object attribute is mandatory may depend on other object attribute values. It should therefore be possible to manage the internal flow of control within particular activities (e.g. user forms) as well. Finally, such integration of process and data necessitates advanced concepts for user integration; i.e., process authorization must be compliant with data authorization and vice versa. While certain users must execute an activity mandatorily in the context of a particular object instance, others may be authorized to optionally execute this activity.

We have already shown that only limited support for these properties is provided by existing imperative, declarative, and data-driven process support paradigms [16]. To ensure the relevance, completeness and generalizability of the identified properties we performed a literature study concerning extensions of the basic paradigms (i.e., imperative, declaratives and data-driven ones) [5]. Finally, we are currently developing a proof-of-concept prototype of our framework.

3 PHILharmonicFlows Framework

This section gives a short overview of our PHILharmonicFlows framework which enforces a well-defined modeling methodology governing the definition of processes at different levels of granularity and being based on a well-defined formal operational semantics. More precisely, the framework differentiates between *micro* and *macro processes* in order to capture both *object behavior* and *object interactions*. As a prerequisite, object types and their relations need to be captured in a data model. Following this, for each *object type* 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 is associated with a set of *object type attributes*. At runtime, a micro process instance being in a particular state may only proceed if specific values are assigned to the attributes associated with this state; i.e., a *data-driven process execution* is applied. *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, PHILharmonicFlows maintains a comprehensive *authorization table* assigning data permissions to user roles depending on the different states of the micro process type. 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* consists of *macro steps* as well as *macro transitions* between them. Opposed to micro steps that relate to single attributes of a particular object type, a macro step refers to an entire object type and a particular state. For each macro transition, a corresponding synchronization component must then be specified. This way, PHILharmonicFlows is able to hide the complexity of large process structures from modelers as well as from end-users. The synchronization components enable the coordination of interactions between the object instances of the same as well as of different object types. Opposed to existing approaches, it is possible to additionally consider the cardinalities between object instances. In particular, whether or not a particular object instance should be (mandatorily or optionally) created depends on the relation cardinalities and on synchronization components specified within the macro process.

4 Modeling Data

As opposed to existing approaches, in which activities and their execution constraints (e.g., precedence relations) are explicitly specified, PHILharmonicFlows allows defining processes by taking *object types* as well as *object interactions* into account. For this purpose, the proper integration of data constitutes a fundamental requirement. Regarding existing process support approaches, however, the data and process perspectives are mostly integrated in one and the same model leading to complex and overloaded models being difficult to maintain. PHILharmonicFlows, in turn, supports the definition of data and processes in separate, but well integrated models. Thus, it retains the well established principle of separating concerns [17].

Due to the widespread use of the relational data model, PHILharmonicFlows is based on relational concepts as well. In this paper, we restrict the data perspective to object types and object attributes (see [5] for our basic idea on how

to treat object relations). As example consider review processes for job applications as known from the human resource area. In this real world example, which we simplified for the sake of clarity, `reviews` are used to evaluate applications and are provided by employees from functional divisions. Based on the results of the `reviews` the personnel officer from the human resource department decides which applicant may get the offered job. For this purpose, a `review` object type may comprise attribute types like `issue date`, `proposal` (e.g., to invite or reject the applicant), `appraisal`, `remark`, `comment`, `reason`, `consideration` (indicating whether the result of the review was used to initiate further actions), and `appointment` (suggested date for interview) (cf. Fig. 1a). Attribute types are represented by atomic data elements with a specific data type (i.e., integer, decimal, string, boolean, date). Arrays and sets, in turn, are captured as relating object types (but are out of the scope of this paper). Finally, an object type comprises a set of attribute types defining its properties.

Let *Identifiers* be the set of all valid identifiers over a given alphabet.

Definition 1. *An **attribute type** is a tuple attrType = (name, type) where*

- *name ∈ Identifiers is an identifier.*
- *type ∈ {INTEGER,DECIMAL,STRING,BOOLEAN,DATE} is a basic data type.*

Further, AttrTypes denotes the set of all definable attribute types and AttrValues the set of all possible attribute values given the above set of data types.

Definition 2. *An **object type** is a tuple oType = (name, AttrTypeSet) where*

- *name ∈ Identifiers is an identifier.*
- *AttrTypeSet ⊂ AttrTypes is a finite set of attribute types.*

Further, $\forall\ attrType_1, attrType_2 \in AttrTypeSet$:
$attrType_1.name = attrType_2.name \Rightarrow attrType_1 \equiv attrType_2$.
Finally, OTypes corresponds to the set of all definable object types.

5 Modeling Object Behavior

Our PHILharmonicFlows framework enforces a well-defined modeling methodology governing 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*. This section introduces our modeling approach for micro process types capturing object behavior.

For each *object type* one specific *micro process type* comprising a number of *micro step types* has to be defined. Each micro step type, in turn, is associated with an attribute type and describes an elementary action (e.g. writing the object attribute). By connecting micro step types using *micro transition types*, we obtain their default execution order. Further, *state types* can be used to realize mandatory activities comprising a subset of the micro step types; i.e., they are

used to coordinate actions between different users. Thereby, the micro step types belonging to the same state type and their relations reflect the internal logic of an activity, whereas state types are used to coordinate the execution of several activities among different user roles.

Fig. 1b shows the micro process type describing the behavior of the `review` object type (cf. Fig. 1a). Each `review` must be created by a personnel officer and then be filled out by an employee. The latter can either refuse the `review` request or fill out the corresponding `review` form. In the latter case, the personnel officer has to evaluate the feedback provided by the employee. For this purpose, our example comprises four state types. These represent mandatory activities involving two roles (cf. Fig. 1b). Further, each micro process type includes at least one *end state*; i.e., an object state in which no further actions are mandatorily required. Our `review` micro process type has two end states, namely `evaluated` and `closed` (cf. Fig. 1b).

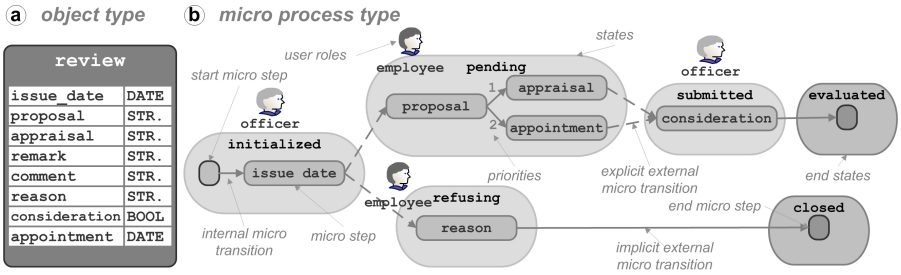

Fig. 1. Review object type and micro process type

We first discuss how to specify the internal logic of a *mandatory activity* as captured by a state and its corresponding micro steps. For each state type, we define a set of corresponding actions. More precisely, each *state type* comprises several *micro step types*. Each of them may represent a mandatory write access to a particular object attribute. Note that single micro step types do not represent activities, but solely refer to one atomic action (e.g., editing an input field within a form). As we will show later, we do not need to explicitly model activities (and corresponding forms), but can *automatically generate* them. Here, also optional input fields as well as read-only data fields are integrated based on a sophisticated authorization table (cf. Fig. 3).

Each micro step type may refer to a specific attribute type. For reaching a micro step during run-time, a value for its corresponding attribute is mandatorily required. As illustrated in Fig. 1b, when initializing a `review` (i.e., state `initialized` is activated), a personnel officer must specify an `issue date`. Besides, state types do not require further actions. As example consider end states and states which only require an explicit commit by a responsible user (e.g., enabling mandatory reading). Respective state types only comprise micro step types not referring to any attribute type.

When executing mandatory activities, users should be guided in setting required attribute values (e.g., by highlighting respective input fields in a form).

Regarding state `pending` in Fig. 1b, for example, after setting the value of attribute `proposal`, either the value of attribute `appraisal` or attribute `appointment` is required next. To capture such logic for the setting of object attributes, their micro step types can be linked using *micro transition types*. Based on them, we can define the *internal logic* of a mandatory activity; e.g., the default order in which the input fields of the corresponding form shall be edited. If a micro step type (e.g., `proposal`) contains more than one outgoing micro transition type (i.e., an alternative processing), we ensure that only one of them is fired at runtime; i.e., always one micro step (and thereby also one state) can be reached.[2] Regarding our example, the values of attributes `proposal` and `appointment` are usually set when enabling respective micro steps. However, an employee may set these values early on. In such case, the micro steps corresponding to these object attributes will be immediately completed when they are reached. If values for both `appointment` and `appraisal` are available, we have to ensure that only one micro step is activated. For this purpose, different *priorities* as illustrated in Fig. 1b can be assigned to micro transition types. Regarding our example, the micro step referring to attribute `appraisal` would be reached because the corresponding incoming transition has a higher priority as the one of the micro step referring to attribute `appointment`. If only a value for attribute `appointment` was available, in turn, its corresponding micro step would be activated.

Definition 3. *An **micro process type** is a tuple micProcType = (oType, MicStepTypeSet, MicTransTypeSet) where*

- *oType = (name, AttrTypeSet) ∈ OTypes is the object type whose behavior is described by micProcType.*
- *MicStepTypeSet is a finite set of micro step types with micStepType = (name, attrType) ∈ MicStepTypeSet having the following meaning:*
  - *name ∈ Identifiers is an identifier.*
  - *attrType ∈ AttrTypeSet ∪ {NULL} is an attribute type or undefined.*
- *MicTransTypeSet ⊂ MicStepTypeSet × MicStepTypeSet × ℕ is a finite set of micro transition types with micTransType = (source, target, priority) ∈ MicTransTypeSet having the following meaning:*
  - *source ∈ MicStepTypeSet is the source micro step type of micTransType.*
  - *target ∈ MicStepTypeSet is the target micro step type of micTransType.*
  - *priority ∈ ℕ is the priority of micTransType.*

MicProcTypes denotes the set of all definable micro process types.

PHILharmonicFlows provides support for backward jumps within micro processes as well (resetting attribute values where required). However, due to lack

[2] Though an object instance is always in exactly one processing state, this does not prohibit parallel execution. During the execution of an activity, parallel processing of disjoint sets of mandatory as well as optional object attributes is always possible. In addition, different users may concurrently process forms corresponding to the same object instance. In this context, known mechanisms for synchronizing concurrent data access can be applied.

of space we only consider acyclic micro process types here. Each micro process type contains exactly one start micro step type which does not refer to any attribute type and has no incoming transitions (cf. Def. 4a). Further, a micro process type must comprise at least one end micro step type which does not refer to an attribute type and has no outgoing micro transition type (cf. Def. 4b). All other micro step types, in turn, must have at least one incoming (cf. Def. 4c) and at least one outgoing micro transition type (cf. Def. 4d). To ensure this we introduce functions for structural analysis of *micProcType = (oType, MicStepTypeSet, MicTransTypeSet) ∈ MicProcTypeSet*. In particular, *intrans: MicStepTypeSet* $\mapsto$ N_0 (*outrans: MicStepTypeSet* $\mapsto$ N_0) determines for each micro step type the number of its incoming (outgoing) micro transition types. To ensure that only one micro step is activated at any point during run-time, micro transition types having the same source micro step type must be associated with different priorities (cf. Def. 4e).

Definition 4. *Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet) ∈ MicProcTypes be an acyclic micro process type referring to an object type oType = (name, AttrTypeSet) ∈ OTypes. Then:*

a) $\nexists$ $startMicStepType_{micProcType}$ *= (name, NULL) ∈ MicStepTypeSet with intrans(*$startMicStepType_{micProcType}$*) = 0; i.e., there exists exactly one start micro step type.*

b) $|\ EndMicStepTypes_{micProcType}\ | \geq 1$ *with* $EndMicStepTypes_{micProcType}$ *:= {micStepType = (name, NULL) ∈ MicStepTypeSet | outtrans(micStepType) = 0}; i.e., there exists at least one end micro step type.*

c) $\forall$ *micStepType ∈ MicStepTypeSet -* $\{startMicStepType_{micProcType}\}$*: intrans(micStepType) ≠ 0.*

d) $\forall$ *micStepType ∈ MicStepTypeSet -* $EndMicStepTypes_{micProcType}$*: outtrans(micStepType) ≠ 0.*

e) $\forall$ $transType_i \in MicTransTypeSet$*, i=1,2:* $transType_1 \neq transType_2 \wedge transType_1.source = transType_2.source \Rightarrow transType_1.priority \neq transType_2.priority$.

Several micro step types can be aggregated to a particular state type:

Definition 5. *Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet) ∈ MicProcTypes be an acyclic micro process type. A* ***state type*** *of a micro process type micProcType is a tuple stateType = (name, sMicStepTypeSet) where*

- *name ∈ Identifiers is an identifier.*
- *sMicStepTypeSet ⊆ MicStepTypeSet is a finite set of micro step types.*

$StateTypes_{micProcType}$ *is a finite set of state types defined on micProcType.*

In Fig. 1b, `pending` is an example of a state type comprising the micro step types `proposal`, `appraisal` and `appointment`. Generally, different state types have disjoint sets of micro step types (cf. Def. 6a) and each micro step type must belong to exactly one state type (cf. Def. 6b). In addition, each *end* micro step

type must belong to a state type comprising no other micro step types (cf. Def. 6c). Further, the micro step types belonging to the same state type must be connected with each other (cf. Def. 6d).

Definition 6. *Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet) ∈ MicProcTypes be an acyclic micro process type and let* $stateType_i \in StateTypes_{micProcType}$, *i = 1,2 be two state types. Then:*

a) $stateType_1 \equiv stateType_2 \Rightarrow$
$stateType_1.sMicStepTypeSet \cap stateType_2.sMicStepTypeSet = \emptyset$
b) $\forall$ *micStepType* $\in$ *MicStepTypeSet:* $\nexists$ *stateType* $\in StateTypes_{micProcType}$*:*
micStepType $\in$ *stateType.sMicStepTypeSet*
c) $\forall$ *micStepType* $\in EndMicStepTypes_{micProcType}$*:* $\nexists$ *stateType* $\in$
$StateTypes_{micProcType}$*: micStepType* $\in$ *stateType.sMicStepTypeSet* $\wedge$
$|$ *stateType.sMicStepTypeSet* $| = 1$.
d) $\forall$ *stateType* $\in StateTypes_{micProcType}$*:* $micStepType_i \in$
stateType.sMicStepTypeSet, i = 1,2 $\wedge$ *micStepType*$_2$ *is a successor of* $micStepType_1 \Rightarrow$ *all micro step types on the path from* $micStepType_1$ *to* $micStepType_2$ *belong to stateType.sMicStepTypeSet as well.*

We further denote *micro transition types* that connect *micro step types* belonging to different *state types* as *external micro transition types.*
Formally: *isexternal: MicTransTypes* $\mapsto$ *BOOLEAN* with:

$$\text{isexternal(mtt)}:=\begin{cases} TRUE, & \exists stateType_i \in StateTypes_{micProcType}, i = 1,2, \\ & \text{with } stateType_1 \neq stateType_2 \\ & \wedge mtt.source \in stateType_1.sMicStepTypeSet \\ & \wedge mtt.target \in stateType_2.sMicStepTypeSet \\ FALSE, & \text{else} \end{cases}$$

As example consider the micro transition type connecting micro step type `consideration` and the one belonging to state `evaluated` in Fig. 1b. At run-time, the firing of an external micro transition triggers a new micro state; i.e., the data-driven execution paradigm is also applied for activating subsequent states. For example, a `review` reaches state `evaluated` as soon as the responsible personnel officer has assigned the value of attribute `consideration` (cf. Fig. 1b). Opposed to a purely data-driven activation, however, some scenarios may require that a responsible user explicitly commits the completion of an activity he has worked on. As example consider state `pending`. An employee may re-execute the activity of filling in the `review` form until he explicitly commits to submit the review back to the personnel officer. To capture this in a micro process type we flag external micro transition types either as implicit or explicit:
explicit: MicTransType $\mapsto$ *BOOLEAN* defines whether a particular micro transition type *micTransType* (with *isexternal(micTransType = TRUE)*) is marked as explicit. As illustrated in Fig. 2a, to ensure that only one state of a micro process instance is activated during run-time, external micro transition types having the same micro step type as source must be defined as explicit ones (cf. Def. 7a). Certain scenarios require explicit user decisions. For example, after a personnel

officer has initiated a `review`, the responsible employee may decide whether to fill in the `review` or to refuse it. In particular, a user decision is required if a micro step has more than one outgoing external, explicit micro transition. In this case, the responsible user has to decide which subsequent state shall be activated. To ensure this, we have to ensure that the target micro step types of explicit external micro transition types having the same source belong to different state types (cf. Fig. 2b and Def. 7b).

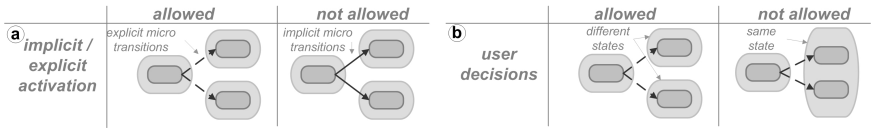

Fig. 2. Structural correctness of external transition types

Definition 7. *Let micProcType = (oType, MicStepTypeSet, MicTransTypeSet) ∈ MicProcTypes be an acyclic micro process type. Then:*
∀ $micTransType_i$ ∈ MicTransTypeSet, i=1,2 with
$micTransType_1 \neq micTransType_2 \wedge micTransType_1.source = micTransType_2.source \wedge isexternal(micTransType_i) = TRUE$, i=1,2
Then:

a) $explicit(micTransType_i) = TRUE$, i=1,2
b) ∃ $stateType_i \in StateTypes_{micProcType}$, i=1,2 with $stateType_1 \neq stateType_2$ ∧ $micTransType_i.target \in sMicStepTypeSet_i$, i=1,2

6 Data and Process Authorization

Generally, we associate state types and explicit micro transition types with user roles in order to be able to determine actors being responsible for mandatory activities, branching decisions, and commitments during run-time. In addition, it must be possible that different users (i.e., roles) may have different access rights on object attributes in a particular micro state. To achieve this, PHILharmonicFlows automatically generates a specific *authorization table* in accordance to the defined micro process type. Based on authorization tables one can define which user role may *read / write* which object attributes in the different states of a micro process (cf. Fig. 3). To ensure proper authorization, each user role assigned to a state type automatically obtains the permissions required for writing the object attributes to which the micro step types of this state type refer (see the shaded boxes in Fig. 3). The generated authorization table may be adjusted by assigning additional optional permissions allowing for the execution of optional activities. Generally, this allows users not being involved in the execution of mandatory activities to own permissions for reading or writing object attributes; e.g., a manager may read or write selected object attributes within state `submitted`. Generally, not every user being allowed to write required

attribute values in a particular 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.

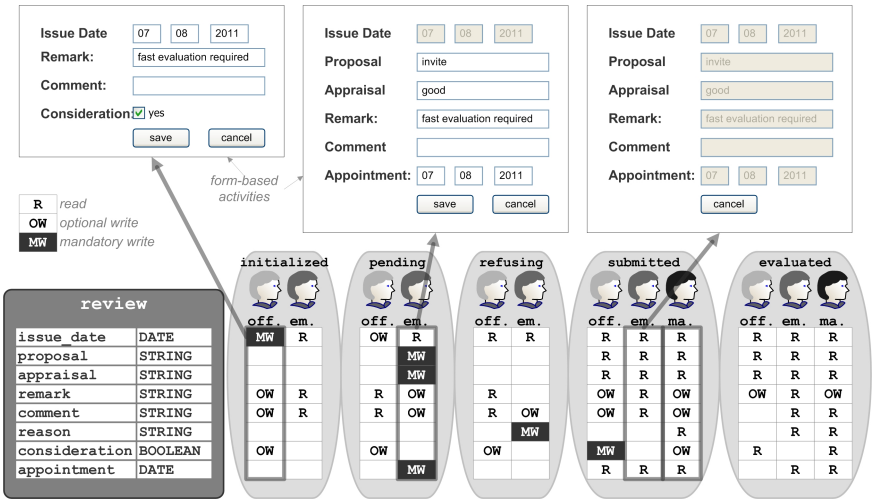

Fig. 3. Authorization table and generation of form-based activities

7 Execution of Micro Processes

Our approach is based on a well-defined formal semantics. In particular, this enables us to automatically generate most end-user components of the run-time environment; e.g., tables giving an overview on object instances and form-based activities. Regarding the latter, the presented authorization table provides the basis for automatically generating user-specific activity forms (cf. Fig. 3); i.e., each user owing respective read and write permissions in a certain (micro process) state may execute a corresponding form-based activity. The *processing state* of an individual micro process instance is defined by the current marking of its states, micro steps, and micro transitions (cf. Fig. 4).

Instantiation. In the following, we refer to our example to demonstrate how a micro process is executed: First of all, when creating a new `reviews` object instance, a corresponding micro process instance is automatically generated and initialized. Thereby, the start micro step is marked as CONFIRMED and the state to which it belongs is marked as ACTIVATED. All other states, in turn, are initially set to WAITING. Further, the outgoing micro transition of the start step is marked as READY, whereas all other micro transitions are initially marked as WAITING. In our example, the incoming internal micro transition

of micro step `issue date` is marked as READY. This, in turn, leads to marking ENABLED of the target micro step of this micro transition. Then, for this micro step a value of its corresponding attribute has to be assigned. All other micro steps belonging to the start state (state `initialized` in our example), in turn, are marked as READY, whereas micro steps not belonging to the start state are marked as WAITING. This differentiation enables us to highlight input fields being relevant for process execution in the currently activated state.

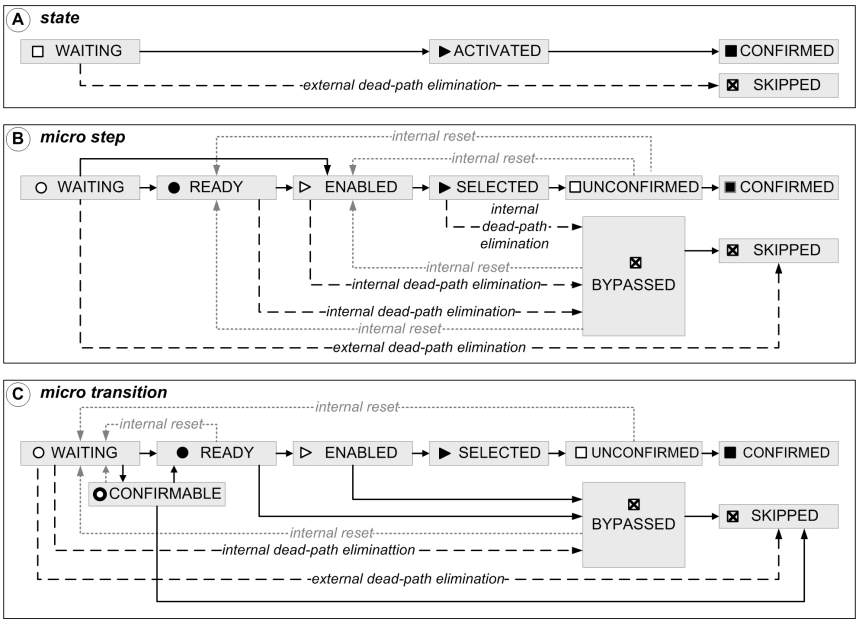

Fig. 4. Operational semantics for states, micro steps and micro transitions

Execution. Starting in state `pending`, micro step `invite` is automatically reached if a value is assigned to the corresponding attribute. Then micro step `invite` is marked as UNCONFIRMED (cf. Fig. 5a). Following this, an employee must provide a value for at least one of the attributes `appraisal` or `appointment`. If for one of these attributes (e.g., `appointment`) a value is set the corresponding micro step is marked as SELECTED (cf. Fig. 5b). The respective micro transition, in turn, is marked as ENABLED. Since no value for attribute `appraisal` is provided (i.e., only one outgoing micro transition is reachable), the priorities of the micro transitions are not relevant. Thus, the ENABLED micro transition can be marked as SELECTED (cf. Fig. 5c). In this case, we omit the other path by performing an *internal dead-path elimination* (cf. Fig. 5d). For this purpose, all micro transitions and steps belonging to the non-selected execution path are marked as BYPASSED (i.e., a micro step is marked as BYPASSED if all incoming micro transitions are marked as BYPASSED).

However, as long as this state change has not been confirmed, an employee may still change attribute settings; i.e., he may want to set the value of attribute

`appraisal`. To accomplish this, an *internal reset* of the currently activated state is performed (cf. Fig. 5e). Generally, micro steps and transitions will be reset if an attribute value corresponding to a micro step marked as UNCONFIRMED or BYPASSED is changed. If a value for both attribute `appraisal` and attribute `appointment` is assigned (cf. Fig. 5f) (i.e., more than one micro transition becomes ENABLED), we ensure that only one of the micro transitions is actually fired; i.e., always one micro step (and one micro state) can be reached. For this purpose, only the micro transition with the highest priority is SELECTED (cf. Fig. 5g). The other one is marked as BYPASSED using the internal dead-path elimination. If a state is marked as CONFIRMED afterwards, micro steps with marking BYPASSED and transitions are finally marked as SKIPPED.

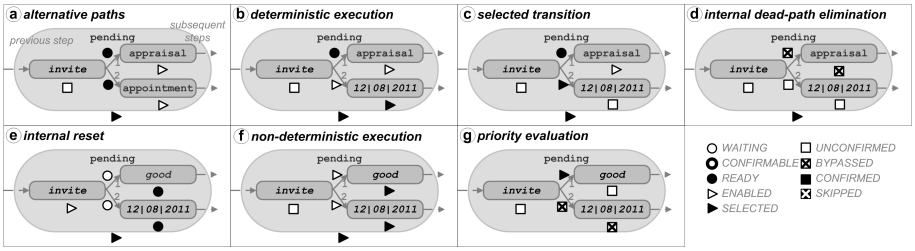

Fig. 5. Execution

Changing the state. After a micro step is marked as UNCONFIRMED, outgoing micro transitions are either marked as READY or CONFIRMABLE. More precisely, external micro transitions, for which an explicit user commitment is required, are marked as CONFIRMABLE. Consequently, a mandatory activity enabling this commitment is automatically assigned to the worklist of the responsible user. In our example, after initializing a review, two external, explicit micro transitions are marked as CONFIRMABLE requiring a respective user decision. If one of them is selected, its marking changes from CONFIRMABLE to READY. Opposed to this, implicit micro transitions (internal and external ones) are immediately marked as READY. If an external micro transition is marked as READY, the currently activated state is marked as CONFIRMED. In addition, all corresponding micro steps as well as internal micro transitions (currently marked as UNCONFIRMED) are re-marked as CONFIRMED as well. Following this, the subsequent state (i.e., state pending in our example) is marked as ACTIVATED and its micro steps as READY. The target micro step of the SELECTED external micro transition (i.e., micro step `proposal`) is marked as ENABLED. For this micro step a value of its attribute has to be set Moreover, we perform an *external dead-path elimination* in order to mark micro steps, micro transitions, and states as SKIPPED that can no longer be activated.

Despite any predefined form logic (e.g., sequence) of micro steps, users are allowed 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 of the corresponding micro steps. In particular, at run-time a micro step can be completed as soon as a value is assigned to its object at-

tribute; e.g., an employee may set the value of attribute `appraisal` although he is guided to first fill in the input field relating to attribute `proposal`. If the value of object attribute `proposal` is set afterwards, the subsequent micro step relating to object attribute `appraisal` is automatically completed. In principle, an entire mandatory activity can be skipped if all required attribute values are assigned in a previous state.

Termination. Finally, execution of a micro process instance terminates if one state containing an end micro step is marked as SELECTED. Opposed to other micro steps, which are marked as UNCONFIRMED, while the state they belong to is marked as ACTIVATED, end micro steps are immediately marked as CONFIRMED. Using the introduced internal and external dead-path elimination, we can ensure that all other states, micro steps and micro transitions are then either marked as CONFIRMED or SKIPPED.

8 Related Work

In [16] we have shown why existing imperative, declarative, and data-driven (i.e., Case Handling [3]) process support paradigms are unable to adequately support object-aware processes. However, to enable consistency between process and object states, extensions of these approaches based on object life cycles (OLC) have been proposed. These extensions include object life cycle compliance [9], object-centric process models [10, 11], business artifacts [8], data-driven process coordination [6], and object-process methodology [18]. 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 run-time, it is not possible to dynamically react to this. In addition, generic form logic is not provided in a flexible way; i.e., there is no automatic generation of forms taking the individual attribute permissions of a user as well as the progress of the corresponding process into account. Finally, opposed to these approaches, PHILharmonicFlows captures the internal logic of an activity as well.

9 Summary and Outlook

In this paper, we introduced the modeling and execution of micro processes which are a fundamental pillar of our PHILharmonicFlows framework for object-aware process management. To enable high flexibility, form-based activities are automatically generated taking the respective user and the current process state into account. Our approach is based on precise rules enabling syntactical correctness as well as on a well-defined operational semantics. Moreover, PHILharmonicFlows goes far beyond the concepts presented in this paper. In future papers

we will discuss how to support backward jumps, time events, black-box activities, and specific attribute values. Regarding the latter, for instance, whether or not a particular attribute value becomes mandatory on-the-fly may depend on the concrete value of an attribute belonging to a previous micro step (e.g., an appointment needs only be defined if the employee proposes to invite the applicant). Moreover, future papers will report on the other components of our framework. As example consider *macro processes* which refer to multiple object instances of various object types. Here, issues related to the object-centred coordination and synchronization of related process instances need to be addressed.

References

1. Aalst, W., Hofstede, A., Weske, M.: Business Process Management: A Survey. In: Proc. BPM'03. LNCS 2678 (2003) 1–12
2. Silver, B.: Case Management: Addressing unique BPM Requirements. BPMS Watch (2009) 1–12
3. Aalst, W., Weske, M., Grünbauer, D.: Case Handling: A new Paradigm for Business Process Support. DKE **53**(2) (2005) 129–162
4. Sadiq, S., Orlowska, M., Sadiq, W., Schulz, K.: When workflows will not deliver: The case of contradicting work practice. In: Proc. BIS'05. (2005)
5. Künzle, V., Reichert, M.: PHILharmonicFlows: Towards a Framework for Object-aware Process Management. Journal of Software Maintenance and Evolution: Research and Practice (2011)
6. Müller, D., Reichert, M., Herbst, J.: Data-Driven Modeling and Coordination of Large Process Structures. In: Proc. CoopIS'07. LNCS 4803 (2007) 131–149
7. Gerede, C., Su, J.: Specification and Verification of Artifact Behaviors in Business Process Models. In: Proc. ICSOC'07. (2007) 181–192
8. Bhattacharya, K., Hull, R., Su, J. In: A Data-Centric Design Methodology for Business Processes. IGI Global (2009) 503–531
9. Küster, J., Ryndina, K., Gall, H.: Generation of Business Process Models for Object Life Cycle Compliance. In: Proc. BPM'07. LNCS 4714 (2007) 165–181
10. Redding, G., Dumas, M., Hofstede, A., Iordachescu, A.: Transforming Object-oriented Models to Process-oriented Models. In: Proc. BPM'07 Workshops. LNCS 4928 (2007) 132–143
11. Redding, G.M., Dumas, M., Hofstede, A., Iordachescu, A.: A flexible, object-centric approach for business process modelling. SOCA (2009) 1–11
12. Reijers, H., Liman, S., Aalst, W.: Product-Based Workflow Design. Management Information Systems **20**(1) (2003) 229–262
13. Vanderfeesten, I., Reijers, H., Aalst, W.: Product-Based Workflow Support: Dynamic Workflow Execution. In: Proc. CAiSE'08. LNCS 5074 (2008) 571–574
14. Künzle, V., Reichert, M.: Towards Object-aware Process Management Systems: Issues, Challenges, Benefits. In: Proc. BPMDS'09. (2009) 197–210
15. Künzle, V., Reichert, M.: Integrating Users in Object-aware Process Management Systems: Issues and Challenges. In: Proc. BPD'09. (2009) 29–41
16. Künzle, V., Weber, B., Reichert, M.: Object-aware Business Processes: Fundamental Requirements and their Support in Existing Approaches. International Journal of Information System Modeling and Design **2**(2) (2010)
17. Dijkstra, E.: A Discipline of Programming. Prentice-Hall (1976)
18. Dori, D.: Object-Process Methodology. Springer (2002)