Flexible Data Acquisition in Object-aware Process Management

Sebastian Steinau, Kevin Andrews, and Manfred Reichert

Institute of Databases and Information Systems, Ulm University, Germany
{sebastian.steinau,kevin.andrews,manfred.reichert}@uni-ulm.de

Abstract. Data-centric approaches to business process management, in general, no longer require specific activities to be executed in a certain order, but instead data values must be present in business objects for a successful completion. While this holds the promise of more flexible processes, the addition of the data perspective results in increased complexity. Therefore, data-centric approaches must be able to cope with the increased complexity, while still fulfilling the promise of more flexible processes. Object-aware process management specifies business processes in terms of objects as well as their lifecycle processes. Lifecycle processes determine how an object acquires all necessary data values. As data values are not always available in the order the lifecycle process of an object requires, the lifecycle process must be able to flexibly handle these deviations. Object-aware process management provides operational semantics with flexible data acquisition built into it, instead of tasking the process modeler with pre-specifying a flexible process. At the technical level, the flexible data acquisition is accomplished with process rules.

1 Introduction

Data-centric modeling paradigms part with the activity-centric paradigm and instead base process modeling and enactment on the acquisition and manipulation of business data. In general, a data-centric process no longer requires certain activities to be executed in a specific order for successful completion; instead certain data values must be present, regardless of the order in which they are acquired. Activities and decisions consequently rely on data conditions for enactment, e.g., an activity becomes executable once required data values are present. While this holds the promise of vastly more flexible processes in theory, it is no sure-fire success. The increased complexity from considering the data perspective in addition to the control-flow perspective requires a thoughtful design of any approach for modeling and enacting data-centric processes. This design must enable the flexibility of data-centric processes, while still being able to manage the increased complexity.

Object-aware process management [11] is a data-centric approach that aims to address this challenge. In the object-aware approach, business data is held in *attributes*. Attributes are grouped into *objects*, which represent logical entities in real-world business processes, e.g., a loan application or a job offer. Each object

has an associated *lifecycle process* that describes which attribute values need to be present for successfully processing the object. Lifecycle processes adopt a modeling concept that resembles an imperative style, i.e., the model specifies the default order in which attribute values are required. Studies have indicated that imperative models posses advantages in understandability when compared to declarative models, which are known for flexibility [14]. While the imperative style allows for an easy modeling of lifecycle processes, it seemingly subverts the flexibility promises of the data-centric paradigms, as imperative models are rather rigid. However, in object-aware process management, the operational semantics of lifecycle processes allow for data to be entered at any point in time, while ensuring correct process execution. The imperative model provides only the basic structure. This has the advantage that modelers need not concern themselves with modeling flexible processes, instead the flexibility is built into the operational semantics of lifecycle processes.

The functional specifications of the operational semantics of lifecycle processes have partially been presented in previous work [10]. This paper expands upon this work and contributes extended functionality and the technical implementation of the operational semantics, provided in the PHILharmonicFlows prototype. This technical implementation is based on the *process rule framework*, a lightweight, custom process rule engine. The framework is based on event-condition-action (ECA) rules, which enable reacting to every contingency the functional specification of the operational semantics allows for, i.e., correct lifecycle process execution is ensured. The process rule framework will further provide the foundation for implementing the execution semantics of *semantic relationships* and *coordination processes*, the object-aware concept for coordinating objects and their lifecycle processes [16]. With the transition of PHILharmonicFlows to a hyperscale architecture [2], the process rule framework is fully compatible with the use of microservices, enabling a highly concurrent execution of lifecycle processes with large numbers of user interactions.

The remainder of the paper is organized as follows. Section 2 provides the fundamentals of object-aware process management. In Section 3, the extended operational semantics are presented. The process rule framework at the core of the operational semantics implementation is discussed in Section 4. Finally, Section 5 elaborates on related work, whereas Section 6 concludes the paper with a summary and an outlook.

2 Fundamentals

Object-aware process management organizes business data in form of objects, which comprise attributes and a lifecycle process describing object behavior. PHILharmonicFlows is the implementation of the object-aware concept to business process management. Object-aware process management distinguishes design-time entities, denoted as *types* (formally ^{T}), and run-time entities, denoted as *instances* (formally ^{I}). Collectively, they are referred to as entities. At run-time, types can be instantiated to create one or more corresponding instances. For

the purpose of this paper, object instance (cf. Definition 1) and lifecycle process instance (cf. Definition 2) definitions are required. The corresponding type definitions can be found in [11]. The "dot" notation is used to describe paths, e.g., for accessing the name of a particular object instance. $\perp$ describes the undefined value.

Definition 1. *An **object instance** ω^I has the form $(\omega^T, n, \Phi^I, \theta^I)$ where the following definitions hold:*

- ω^T *refers to the object type from which this object instance has been generated.*
- *n is the name of the object instance.*
- *Φ^I is a set of attribute instances ϕ^I, where $\phi^I = (n, \kappa, v_\kappa)$, with n as the attribute instance name, κ as the data type (e.g., string, bool, integer), and v_κ as the value of the attribute instance.*
- *θ^I is the lifecycle process (cf. Definition 2) describing object behavior.*

An object's lifecycle process (cf. Definition 2) is responsible for acquiring data values for the attributes of the object.

Definition 2. *A **lifecycle process instance** θ^I has the form $(\omega^I, \Sigma^I, \Gamma^I, T^I, \Psi^I, E_\theta, \mu_\theta)$ where the following definitions hold:*

- *ω^I refers to the object instance to which this lifecycle process belongs.*
- *Σ^I is a set of **state instances** σ^I, with $\sigma^I = (n, \Gamma_\sigma^I, T_\sigma^I, \Psi_\sigma^I, \mu_\sigma)$ where*
  - *n is the state name.*
  - *$\Gamma_\sigma^I \subset \Gamma^I$ is subset of steps γ^I.*
  - *$T_\sigma^I \subset T^I$ is a subset of transitions τ^I.*
  - *$\Psi_\sigma^I \subset \Psi^I$ is a subset of backwards transitions ψ^I.*
  - *μ_σ is the state marking.*
- *Γ^I is a set of **step instances** γ^I, with $\gamma^I = (\phi^I, \sigma^I, T_{in}^I, T_{out}^I, P^I, \lambda, \mu_\gamma, d_\gamma)$ where*
  - *$\phi^I \in \omega^I.\Phi^I$ is an optional reference to an attribute instance ϕ^I from Φ^I of object instance ω^I. Default is $\perp$.*
    *If $\phi^I = \perp$, the step is denoted as an **empty step instance**.*
  - *$\sigma^I \in \Sigma^I$ is the state instance to which this step instance γ^I belongs.*
  - *$T_{in}^I \subset T_\sigma^I$ is the set of incoming transition instances τ_{in}^I.*
  - *$T_{out}^I \subset T_\sigma^I$ is the set of outgoing transition instances τ_{out}^I.*
  - *P^I is a set of **predicate step instances** ρ^I, P^I may be empty, with $\rho^I = (\gamma^I, \lambda)$ where*
    - *γ^I is a step instance.*
    - *λ is an expression representing a decision option.*
    *If $P^I \neq \emptyset$, the step instance γ^I is called a **decision step instance.***
  - *λ is an optional expression representing a computation.*
    *If $\lambda \neq \perp$, the step instance γ^I is called a **computation step instance.***
  - *μ_γ is the step marking, indicating the execution status of γ^I.*
  - *d_γ is the step data marking, indicating the status of ϕ^I.*

- T^I is a set of **transition instances** τ^I, with $\tau^I = (\gamma^I_{source}, \gamma^I_{target}, ext, p, \mu_\tau)$ where
  - $\gamma^I_{source} \in \Gamma$ is the source step instance.
  - $\gamma^I_{target} \in \Gamma$ is the target step instance.
  - $ext := \gamma^I_{source}.\sigma^I = \gamma^I_{target}.\sigma^I$ is a computed property, denoting the transition as external, i.e., it connects steps in different states.
  - p is an integer signifying the priority of the transition.
  - μ_τ is the transition marking.
- Ψ^I is a set of **backwards transition instances** ψ^I, Ψ^I may be empty, with $\psi^I = (\sigma^I_{source}, \sigma^I_{target}, \mu_\psi)$ where
  - $\sigma^I_{source} \in \Sigma^I$ is the source state instance.
  - $\sigma^I_{target} \in \Sigma^I$ is the target state instance, $\sigma^I_{target} \in Predecessors(\sigma^I_{source})$.
  - μ_ψ is the backwards transition marking.
- E_θ is the event storage for θ^I, storing execution events ϵ^E.
- μ_θ is the lifecycle process marking.

All sets are finite and must not be empty unless specified otherwise. The function Predecessors: $\sigma^I \to \Sigma^I$ determines a set of states from which σ^I is reachable. The function Successors is defined analogously.

Note that for the sake of brevity the value of a step γ^I refers to the value of the corresponding attribute $\gamma^I.\phi^I$. Furthermore, correctness criteria have been omitted from Definitions 1 and 2. For the sake of clarity, a lifecycle process is described by a directed acyclic graph with one start state and at least one end state. Figure 1 shows object instance *Bank Transfer* with attributes and the lifecycle process. The object instance represents a simplified transfer of money from one account to another. For this purpose, the states and the steps of a lifecycle process can be used to automatically generate forms. When executing a process, the auto-generated forms are filled in by authorized users. The PHILharmonicFlows authorization system and its connection to form auto-generation has been discussed in [1].

As depicted in Figure 1, the state $\sigma^I_{Initialized}$ contains steps γ^I_{Amount} and γ^I_{Date}, signifying that values for attributes ϕ^I_{Amount} and ϕ^I_{Date} are required during process execution. For the sake of brevity, the properties of an entity (e.g., the name of a step γ) may be written as a subscript, e.g., γ_{Amount} for the first step in Figure 1. The form corresponding to $\sigma^I_{Initialized}$ contains input fields for steps γ^I_{Amount} and γ^I_{Date}. This means a state represents a form, whereas the steps represent form fields. The $\phi^I_{Comment}$ field is an optional field visible to a user due to the authorization system of PHILharmonicFlows. In state $\sigma^I_{Decision}$, a decision step $\gamma^I_{Approval}$ represents the approval of the bank for the money transfer. The automatically generated form displays $\gamma^I_{Approval}$ as a drop-down field. End states $\sigma^I_{Approved}$ and $\sigma^I_{Rejected}$ display an empty form, as the contained steps are empty (cf. Definition 2). Transitions determine at run-time which attribute value is required next, an external transition also determines the next state. Backwards transitions allow returning to a previous state, e.g., to correct a data value.
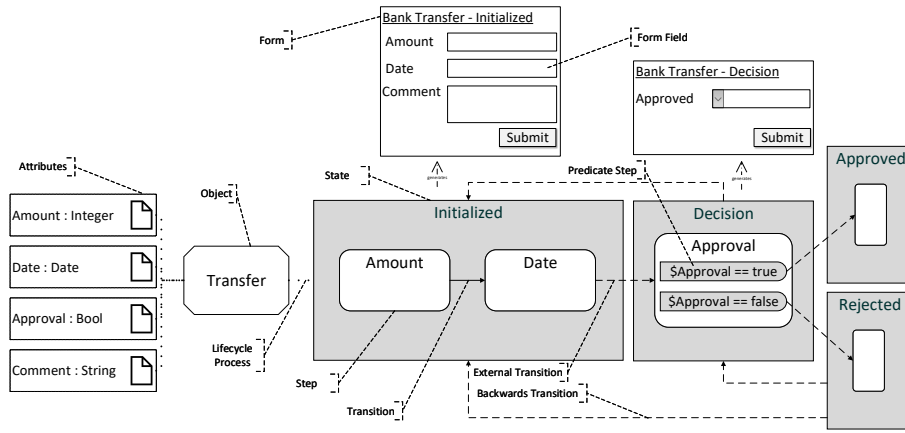
Fig. 1. Example object and lifecycle process of a transfer

3 Lifecycle Process Operational Semantics

Data acquisition in PHILharmonicFlows is achieved through forms, which can be auto-generated from lifecycle process models θ^I. A form itself is mapped to a state σ^I of the lifecycle process θ^I, form fields are mapped to steps γ^I. As a consequence, the operational semantics emulate the behavior of electronic and paper-based forms, following a "best of both worlds"-approach. Paper-based forms provide a great overview over the form fields, i.e., every form field may be viewed at any point in time. Furthermore, they provide a reasonable default structure, but allow filling form fields at any point in time and in any order, e.g., starting to fill in form fields in the middle of the form is possible. In turn, electronic forms usually provide less overview, i.e., viewing subsequent forms is not possible before having filled out all mandatory fields in the current form. In contrast to paper-based forms, however, electronic forms are able to only display relevant fields, especially in context of decision branching. For example, an electronic anamnesis form at a physician's office may skip the questions related to pregnancy entirely if the patient is male. Additionally, electronic forms allow for data values to be easily changed as well as input verification, e.g., ensuring that a date has the correct format or all mandatory form fields possess a value. PHILharmonicFlows combines the advantages of both paper-based and electronic forms, providing flexibility in entering data while ensuring a correct lifecycle process execution.

3.1 Lifecycle Process Execution

For realizing the combined benefits, the progress of a lifecycle process θ^I is determined by its active state σ_A^I , i.e., marking $\sigma^I.\mu_\sigma = Activated$. Only one state σ^I may be active at any point in time. Per default, the form of the active

state is displayed to a user when executing a lifecycle process θ^I. However, the user may choose to display forms of other states. When processing θ^I, the active state changes, depending on data availability and decision results. For example, in regard to Figure 1, starting the execution of the lifecycle process activates $\sigma^I_{Initialized}$. If values for steps γ^I_{Amount} and γ^I_{Date} are available (cf. Section 3.2), $\sigma^I_{Initialized}$ may be marked as $\mu_\sigma = Confirmed$, and the next state $\sigma^I_{Decision}$ may become active, i.e., $\sigma^I_{Decision}.\mu_\sigma = Activated$. Depending on the value of $\gamma^I_{Approval}$, either $\sigma^I_{Approved}$ or $\sigma^I_{Rejected}$ becomes active. As both states are end states, the execution of θ^I terminates. The active state possesses a crucial role in the execution of θ^I, as consequences from data acquisition or decisions are only evaluated for the active state. For example, providing value $true$ to $\gamma^I_{Approval}$ does not trigger the decision, if $\sigma^I_{Initialized}$ is the currently active state. This is to avoid inconsistent processing states, e.g., because a previous decision may make filling out a state σ^I obsolete due to dead-path elimination [11].

For states $Successors(\sigma^I_A)$, data values may be entered, but processing only occurs once a state becomes active. All successor states possess marking $\mu_\sigma = Waiting$. If a user enters values for steps γ^I, these values will be stored and taken into account if the corresponding state $\gamma^I.\sigma^I$ becomes active. To indicate the status of the corresponding attribute value, steps possess a *data marking* d_γ. When setting data value for a step $\gamma^I_{hasValue}$, where the state instance σ^I has $\mu_\sigma = Waiting$, the data marking of $\gamma^I_{hasValue}$ is set to $d_\gamma = Preallocated$. Should σ^I become active during process execution, $d_\gamma = Preallocated$ will indicate that a value is present and will not be required anymore (cf. Section 3.2).

States that have already been processed, i.e., $Predecessors(\sigma^I_A)$, will either have $\mu_\sigma = Confirmed$ or $\mu_\sigma = Skipped$. States with marking $\mu_\sigma = Confirmed$ have previously been active, whereas skipped states have undergone a dead-path elimination. For reasons of data integrity, the values of steps in skipped or confirmed states must not be altered at any point in time. If allowed, inconsistencies and unpredictable execution behavior may occur. For example, changing values of decisions steps in an uncontrolled way might activate currently eliminated states, whereas currently active states become eliminated. However, it must be possible to correct mistakes for previously entered and accidentally confirmed data. Therefore, *backwards transitions* (cf. Definition 2) allow for the reactivation of confirmed states in a controlled way, where the data may be altered in a consistent and safe way and subsequent changes in decisions can be handled properly.

3.2 State Execution

While PHILharmonicFlows is capable of generating forms from states and steps, these forms are static. However, there are dynamic aspects to a form, e.g., the indication which value is required next or which external transition or backwards transition may be committed. For this purpose, a lifecycle process θ^I provides *execution events* ϵ^E and an *event storage* E_θ. Execution events are dynamically created when processing a lifecycle process θ^I. When auto-generating a form,

the static form is enriched with dynamic information from E_θ and displayed to the user. Execution events have different subtypes, namely *request events, completion events,* and *invalidation events.* When request events are created, they are stored in E_θ and are then used to enrich the form. Completion and invalidation events remove request events from E_θ, when a request event are either fulfilled or no longer valid, respectively. The usage of the event storage E_θ in conjunction with the generated static forms allows multiple users access to the same form, due to the centralized storage of the dynamic form data. Using E_θ further allows preserving dynamic data over multiple sessions, i.e., a user may partially fill out a form, close it and do something else, and later return and continue where the user previously stopped. It is also possible that another user finishes filling out the form. In general, storing execution events ϵ^E ensures consistency regardless of any user interaction with the forms.

The creation and removal of execution events is primarily determined by the respective marking μ of states, steps, transitions, and backwards transitions. For steps with an attribute (i.e., $\gamma^I.\phi^I \neq \bot$), data marking d_γ is also taken into account. For example, if step γ^I_{Amount} of Figure 1 has marking $\mu_\gamma = Enabled$, but $\gamma^I_{Amount}.d_\gamma = Unassigned$ holds, an *attribute value request event* is created and stored in E_θ after some intermediate processing steps. If a user access the form for $\sigma^I_{Initialized}$, form field for γ^I_{Amount} is tagged with an asterisk, indicating that a value is

Fig. 2. Form enriched with execution events

mandatory (cf. Figure 2). As soon as the user provides a value for the γ^I_{Amount} form field , the data marking for γ^I_{Amount} is updated to $d_\gamma = Assigned$. This indicates that a value has been successfully provided for γ^I_{Amount}. As a consequence, the attribute value request event in E_θ is no longer necessary. Therefore, setting $d_\gamma = Assigned$ triggers a completion event removing the attribute value request event from E_θ. After the completion event has occurred, more markings change in a cascading fashion, until the next step γ^I_{Data} receives $\mu_\gamma = Enabled$. The data marking $\gamma^I_{Date}.d_\gamma = Unassigned$ triggers the same chain of events and marking changes as with γ^I_{Amount}.

To illustrate the automatic handling of preallocated data values, it is assumed that another user has already provided value `false` for $\gamma^I_{Approval}$ in state $\sigma^I_{Decision}$, i.e., $\gamma^I_{Approval}.d_\gamma = Preallocated$ holds. As $\sigma^I_{Decision}$ is not currently the active state (i.e., $\mu_\sigma = Waiting$), decision step $\gamma^I_{Approval}$ is not evaluated. When reaching $\gamma^I_{Approval}$ from γ^I_{Date} after a state change and is marked with $\mu_\gamma = Enabled$, no attribute value request event is created due to $d_\gamma = Preallocated$. Instead, the data marking is immediately changed to $d_\gamma = Assigned$. Consequently, the completion event for providing a value is omitted. As $\gamma^I_{Approval}$ is a decision step, value `false` subsequently leads to the activation of state $\sigma^I_{Rejected}$ (cf. Figure 1), in which θ^I terminates. Note that the end state remains active

despite the termination of the lifecycle process instance. In general, the operational semantics of lifecycle processes ensure that a previously provided value requires no further user interaction by default. However, a user may still change the value afterwards should he wish to do so. Overall, the user may flexibly enter and alter data and the operational semantics ensures data integrity.

Continuing the example, where currently the lifecycle process has terminated and $\sigma^I_{Rejected}$ is the active state, a user decides he wants to revise his decision for approval and thus change the value of $\gamma^I_{Approval}$ from `false` to `true`. After $\sigma^I_{Rejected}$ had become active, two backwards transition instances ψ^I_{ToInit} and ψ^I_{ToDec} became confirmable, i.e., their marking changed to $\mu_\psi = Confirmable$. As a consequence, two *backwards transition confirm request events* are first create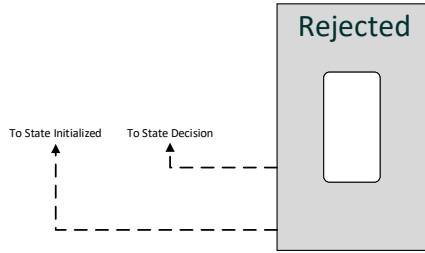d, one for each backwards transition, and then stored in E_θ. This allows going back to state $\sigma^I_{Initialized}$, using ψ^I_{ToInit}, or go back to $\sigma^I_{Decision}$, using ψ^I_{ToDec}. However, only one state may be active at once. Therefore, only one backwards transition may be taken. To revise the value of $\gamma^I_{Approval}$, ψ^I_{ToDec} must be confirmed. Confirming ψ^I_{ToDec} causes its marking to change to $\mu_\psi = Ready$. Analogously to a step, a completion event is created, which removes the corresponding backwards transition confirm request event from E_θ. Subsequently, $\sigma^I_{Rejected}$ is marked as $\mu_\sigma = Waiting$ and $\sigma^I_{Decision}$ is marked as $\mu_\sigma = Activated$, which allows altering the value of $\gamma^I_{Approval}$ to `true`. As $\sigma^I_{Rejected}$ is no longer active, ψ^I_{ToInit} and ψ^I_{ToDec} become marked as $\mu_\psi = Waiting$. Resetting the markings of both ψ^I_{ToInit}, ψ^I_{ToDec}, and $\sigma^I_{Rejected}$ to *Waiting* enables their reuse, e.g., if the value of $\gamma^I_{Approval}$ remains unchanged and the same path is taken again.

Fig. 3. Backwards transitions

With state $\sigma^I_{Decision}$ becoming active again, it is possible to change the value of $\sigma^I_{Approval}$. However, the backwards transition confirm request event belonging to ψ^I_{ToInit} is still stored in E_θ, despite ψ^I_{ToInit} having been marked with $\mu_\psi = Waiting$, i.e., confirming ψ^I_{ToInit} is no longer possible. Obviously, this constitutes an inconsistency between the forms and the lifecycle process. The form displays a button with the option that ψ^I_{ToInit} can be confirmed, but on pressing the button the PHILharmonicFlows system produces an error and other, possibly worse, side effects. A a consequence, the operational semantics include invalidation events, with the purpose to remove invalid or obsolete execution events from event storage E_θ. An invalidation event occurs when entities with a request event, e.g., backwards transitions, are not successfully completed, but become

changed due to other circumstances, e.g., the confirmation of another backwards transition.

Request events, completion events, and invalidation events are used in many other situations than discussed above. The principles, however, are the same, and, embedded in the overall operational semantics, provide a robust, but flexible way to acquire data values for lifecycle processes. The imperative-style modeling of lifecycle processes, from which forms can be auto-generated directly, significantly reduces modeling time and efforts. The operational semantics provide the necessary flexibility to users interacting with the forms. Furthermore, the use of forms and the emulation of standard form behavior simplifies the usage of the PHILharmonicFlows system for non-expert users.

Overall, this section described the functional aspects of the operational semantics of lifecycle processes. The technical implementation of these operational semantics with the *Process Rule Framework* is presented in Section 4.

4   The Process Rule Framework

In the description of the operational semantics of lifecycle processes, at the lowest level, progress is driven by the change of markings. Marking changes elicit the creation of execution events, which, in turn, results in user actions, e.g., the provision of a data value for an attribute. This user interaction is reflected in the lifecycle process by setting new markings. This may be viewed as a chain of events, and, consequently, event-condition-action rules are used as the basis for the technical implementation of the operational semantics. In PHILharmonicFlows, a specialized variant of ECA rules, denoted as *process rules,* is employed. Process rules and the means to specify them constitute one part of the *process rule framework.* To create an execution sequence such as the one described in Section 3.2, process rules need to form *process rule cascades,* i.e., a rule triggers an event, which may trigger another rule, which again triggers an event. Furthermore, process rules are uniquely suited to deal with the different eventualities emerging during the execution of lifecycle processes. For example, a state σ^I may become active in context of normal process execution progress or due to the use of a backwards transition ψ^I. Subsequently, different follow-up measures may be required, e.g., the resetting of markings for steps $\gamma^I \in \sigma^I.\Gamma^I$ in case the backwards transition became activated.

The basic definition of a process rule is shown in Definition 3. In order to distinguish the symbols from symbols used in the definition object instances, superscript ^{R} is used.

Definition 3. *A process rule p^R has the form $(\epsilon, e^T, C^R, A^R)$ where the following definitions hold:*

- *ϵ is an event triggering the evaluation of the rule.*
- *e is an entity type, e.g., a step type γ^T.*
- *C^R is a set of preconditions in regard to e^T.*
- *A^R is a set of effects.*

Process rules p^R may be evaluated, i.e., their preconditions C^R are checked and, if all are fulfilled, the effects are applied. An evaluation is triggered when the event ϵ occurs. Events ϵ are always raised by a particular entity instance e^I, e.g., a step γ^I or an transition τ^I. e^T is an entity type that provides the context for defining conditions and effects. Furthermore, it provides an implicit precondition, meaning a rule is not evaluated if the entity instance e^I raising ϵ was not created from e^T. Preconditions C^R check different properties of an entity, e.g., whether the entity has a specific marking. Effects A^R apply different effects to an entity, e.g., setting the marking of an entity. Note that preconditions and effects are not limited to properties belonging to instances of e^T. They may also access or set properties of neighbor entities. For example, a rule defined for a step γ^T may have effects that set markings for its outgoing transitions $\tau^I_{out} \in \gamma^I.T^I_{out}$.

```
public class MarkingRuleMr14A : AbstractEffectRule<TransitionInstance>
{
    public MarkingRuleMr14A()
    {
        Name = "Marking Rule Mr14A";
        ShortDescription = "Marking steps and predicate steps as Enabled if the transition is internal and marked as Ready";

        PreconditionFor< TransitionInstance,TransitionMarkings>(trans => trans.Marking).IsMarked(TransitionMarkings.Ready);
        PreconditionFor< TransitionInstance,bool>(trans => trans.IsExternal).SatisfiesPredicate(x => !x);

        //If the step is a decision step, mark its predicate steps also as Enabled
        EffectForEach<PredicateStepInstance, StepMarkings>(
            trans => trans.Target.Cast<DecisionStepInstance>().PredicateSteps.Select(x => x.Marking))
            .AssignMarking(StepMarkings.Enabled)
            .When(trans => trans.Target is DecisionStepInstance);

        //Mark the target micro step as Enabled
        EffectFor<AbstractStepInstance, StepMarkings>(trans => trans.Target.Marking).AssignMarking(StepMarkings.Enabled);
    }
}
```

Fig. 4. Fluent interface definition of a marking rule in code

In the PHILharmonicFlows implementation, process rules are created in code. Figure 4 shows an example of how a process rule is represented in code. Process rules are often subject to change, as new features for PHILharmonicFlows are added or errors in lifecycle process execution are resolved. In order to be able to quickly adapt, the process rule framework uses a fluent interface for process rule specification. This allows for a high readability and maintainability.

The operational semantics introduced in Section 3 allow identifying different use cases for process rules. For example, one type of process rule raises execution events based on specific markings, while another type reacts to user input and sets appropriate markings. Consequently, process rules are subdivided based on their purpose. The type determines the general type of preconditions and effects, e.g., preconditions of marking rules check predominantly for specific markings. The different types of process rules are summarized in Table 1. Request rule, completion rule, and invalidation rule are subsumed with the term execution rule (ER).

Rule	Abbreviation	Event	Preconditions	Effects
Marking Rule	MR	Marking Event	Markings	Markings
Request Rule	QR	Marking Event	Markings	Request Event
Completion Rule	CR	Marking Event	Markings	Completion Event
Invalidation Rule	IR	Marking Event	Markings	Invalidation Event
Reaction Rule	RR	User Input Event	User Input	Markings

Table 1. Overview over the types of process rules

The most common event that is raised during the execution of a lifecycle process instance is a *marking event.* An entity instance e^I raises a marking events whenever its marking $e^I.\mu$ is changed. In order to determine which process rule needs to be applied, the event is gathered by the *process rule manager* (PRM) of the lifecycle process. The process rule manager is a small and lightweight execution engine for process rules and constitutes the other part of the process rule framework. Figure 5 shows a schematic view of the process rule manager and its interactions with the lifecycle process and the (auto-generated) forms.
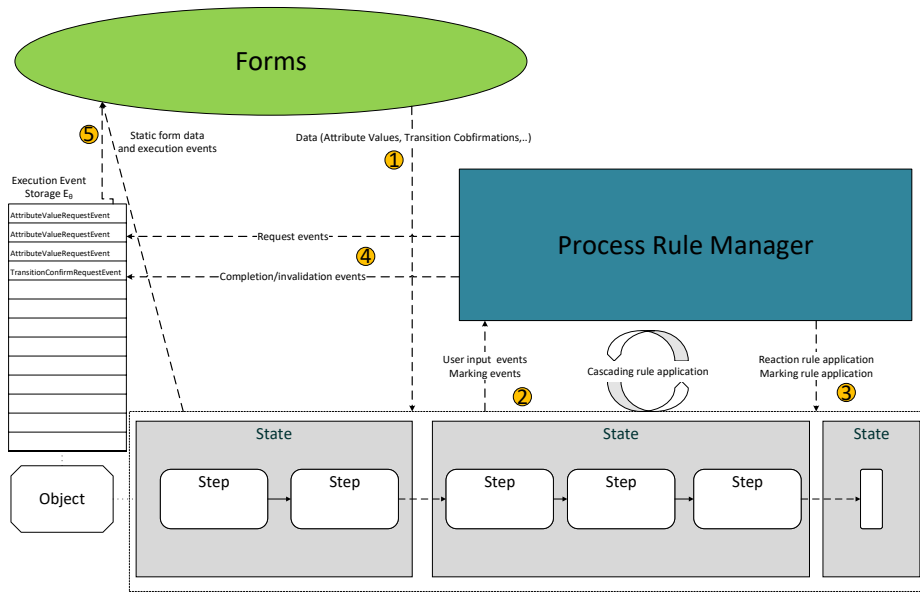

Fig. 5. Process rule manager and schematic process rule application

Starting at ① in Figure 5, data has been entered into a form field. The data is then passed onto the lifecycle process θ^I and the corresponding step γ^I. As γ^I has received a value, the step raises a user input event ②. The event is passed on to the process rule manager, which receives all events from its corresponding

lifecycle process θ^I and evaluates appropriate rules, i.e., process rules p^R with $p^R.e^T = \sigma^T$ are not evaluated if the entity creating the event has type γ^T. The implicit precondition significantly reduces the search space for process rule application and must not be specified by a programmer. Once the PRM has identified all currently applicable rules, the effects of each rule are applied. In the example, the PRM identifies a reaction rule and applies its effects to the appropriate entities in the lifecycle process ③.

Applying the effects from the reaction rule application raises marking events, which trigger a completion rule and a marking rule in the PRM. The completion rule raises a completion event ④, removing the request event for the mandatory form field from event storage E_θ of θ^I. In parallel, the marking rule sets markings for the outgoing transitions T^I of step γ^I. This again creates marking events, resulting in a cascade of marking rules, i.e., the PRM alternates between ② and ③ in Figure 5. The process rule cascade stops when the next step becomes marked with $\mu_\gamma = Enabled$. This raises a request event, which is deposited in event storage E_θ ④. When a user views a form, the updated event storage E_θ and the static form data is combined into a new form ⑤. When the user enters data for the next form field, the cycle starts again at ①.

When a user fills out a form, the form is expected to tell the user immediately which form field is required next after providing data for a form field. Long processing times are prohibitive for the usability of the PHILharmonicFlows system. In order to have full control over processing times and the tight connection of process rules with lifecycle process entities, it was decided to implement the PRM as a custom, lightweight rule engine. A custom PRM implementation offers a fine-grained control over process rule application. By default, the PRM handles events in the order in which they arrive (FIFO principle). However, in several cases, the handling of specific events needed to be delayed or accelerated in order to ensure a form processing in compliance with the operational semantics. For example, an event e_τ triggering the transition τ^I from a source state σ^I_{source} to a target state σ^I_{target} is, under certain circumstances, raised before all steps $\gamma^I \in \sigma^I_{source}.\Gamma^I$ have been processed. This results in errors in the application of the process rules, as the target state σ^I_{target} already received $\mu_\sigma = Activated$ when events from $\gamma^I \in \sigma^I_{source}.\Gamma^I$ arrive at the PRM. To prevent such errors, the handling of the state transition event e_τ must be delayed until all steps γ^I in the source state σ^I_{source} have finished processing. As a consequence, the PRM was extended with a priority queue that retains the FIFO principle, but allows assigning different priorities to events, accelerating or delaying them as needed.

Besides the advantages for the application of process rules, the lightweight nature of the PRM also proves beneficial for the transition of PHILharmonicFlows to a microservice-based architecture. The PRM was initially conceived as a monolithic rule engine, i.e., all lifecycle processes use the same instance of the PRM. Currently, PHILharmonicFlows is moving towards a hyperscale architecture [2], based on a microservice framework. A microservice is a lightweight and independent service that performs single functions and interacts with other microservices in order to realize a software application. In this new hyperscale

architecture, an object and its lifecycle process are implemented as a single microservice. A continued use of a single PRM instance generates a significant performance overhead due to the necessary message exchanges between the PRM and the microservices. The single PRM instance is a bottleneck and puts a limit on the scalability of the microservice-based architecture, i.e., it would no longer be warranted to designate the PHILharmonicFlows system as hyperscale. Furthermore, the communication overhead and the delays of process rule application in the PRM, due to the high number of events simultaneously created by the object instance microservices, would negatively affect the performance of the auto-generated forms.

Fortunately, the lightweight nature of the PRM offers a satisfactory solution. By integrating an instance of the PRM into the microservice of each object instance, no message exchanges between PRM and lifecycle process are required. Furthermore, a PRM instance is only responsible for exactly one lifecycle process instance. This eliminates the delays in rule application due to the processing of other lifecycle processes. This solution offers sufficient performance for displaying dynamic forms while retaining the hyperscale property of the PHILharmonicFlows microservice-based architecture. The approach to integrate a PRM instance into a microservice will also be used with the implementation of coordination processes, where it will provide the same benefits.

5 Related Work

Opus [5,7] is a data-centric process management system that bases its processes on Petri nets. Petri nets are a popular and well-established formalism for modeling business processes. Additionally, Petri nets provide several verification techniques, e.g., soundness checks or deadlock detection, which may also be applied to verify process model correctness. In Opus, the Petri net formalism is extended with structured data tuples, which substitute the places of a standard Petri net. The transitions of this extended Petri net provide operations on the data, e.g., operations derived from operations of relational algebra. The Opus approach does not support automatically generating forms from process models. Furthermore, Petri nets are inherently more rigid in their execution and do provide the same built-in flexibility as PHILharmonicFlows and the operational semantics of lifecycle processes. However, Opus is capable to explicitly model the different execution paths to provide flexible process execution. Opus provides an implemented prototype [6].

Case Handling [15,17] defines a case in terms of activities and data objects. Activities are ordered in an acyclic graph in which edges represent precedence relations. To execute an activity, all precedence relations before the activity must be fulfilled. Furthermore, the execution of an activity is restricted by data bindings. A data binding represents a condition so that a data object must have a specific value at run-time. The values of the data objects are acquired by forms, which are associated with activities. While case handling possesses forms, it is unclear whether these can be auto-generated from the activities or must

be created manually. While both case handling and PHILharmonicFlows use an acyclic graph to represent processes, the operational semantics for lifecycle processes in PHILharmonicFlows allows for data to be acquired at any point in time. A case acquires data by activities and that activities have a precedence relation, the same flexibility in regard to data acquisition is not possible.

The Guard-Stage-Milestone (GSM) meta-model [9] is a declarative notation for specifying artifact-centric processes [3,8,12]. An artifact consists of an information model, i.e., attributes and a lifecycle model. The lifecycle model is specified using GSM. Its operational semantics are based on Precedent-Antecedent-Consequent rules and possess different, equivalent formulations [4]. In GSM, tasks provide the means to write attributes and acquire data. As a declarative language, guards, stages and milestones may be used in such a way that flexible data acquisition, within certain constraints, becomes possible. Tasks may be defined so that attributes may be written at any point in time and may be restricted, if necessary. Lifecycle processes defined in GSM are able to react to the newly acquired data and may be more flexible than lifecycle processes in PHILharmonicFlows. However, as a drawback much of this flexibility in data acquisition must be implemented by the process modeler. Furthermore, the is no auto-generation of forms from GSM-specified lifecycle models within the artifact-centric approach.

6 Summary and Outlook

The technical implementation of the operational semantics of object-aware process management is achieved by *process rules*, which govern the changing of markings and the creation of execution events. Therefore, Section 4 presents the process rule framework, which is important for two aspects. First, the process rule framework makes sure lifecycle processes execute correctly and also provides the technical basis for the operational semantics of coordination processes in PHILharmonicFlows. Coordination processes, as the name suggests, coordinate lifecycle processes of object, so that complex business processes can be realized. Its operational semantics will also be based on the process rule framework. Second, a performant, efficient and lightweight technical basis for enacting lifecycle processes and coordination processes is crucial for the transition of PHILharmonicFlows to a hyperscale architecture. The operational semantics of lifecycle processes provide a flexible acquisition of data, while modeling efforts are minimal due to an modeling style that is akin to an imperative style. The flexibility is not provided by the lifecycle process model, but by the operational semantics. The model of the lifecycle process and the operational semantics together provide the means to auto-generate dynamic forms.

References

1. Andrews, K., Steinau, S., Reichert, M.: Enabling Fine-grained Access Control in Flexible Distributed Object-aware Process Management Systems. In: 21st IEEE Int'l Conf. on Enterprise Distributed Object Computing (EDOC) (2017)

2. Andrews, K., Steinau, S., Reichert, M.: Towards Hyperscale Process Management. In: 8th Int'l Workshop on Enterprise Modeling and Information Systems Architectures (EMISA). pp. 148–152. CEUR Workshop Proceedings, CEUR-WS.org (2017)
3. Cohn, D., Hull, R.: Business Artifacts: A Data-centric Approach to Modeling Business Operations and Processes. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 32(3), 3–9 (2009)
4. Damaggio, E., Hull, R., Vaculín, R.: On the Equivalence of Incremental and Fixpoint Semantics for Business Artifacts with Guard–Stage–Milestone Lifecycles. Information Systems 38(4), 561–584 (2013)
5. Haddar, N., Tmar, M., Gargouri, F.: A Framework for Data-Driven Workflow Management: Modeling, Verification and Execution. In: 24th Int'l Conf. on Database and Expert Systems Applications (DEXA). vol. 8055, pp. 239–253. Springer (2013)
6. Haddar, N., Tmar, M., Gargouri, F.: Opus framework: A Proof-of-Concept Implementation. In: IEEE/ACIS 14th Int'l Conf. on Computer and Information Science (ICIS). pp. 639–641 (2015)
7. Haddar, N., Tmar, M., Gargouri, F.: A Data-centric Approach to Manage Business Processes. Computing 98(4), 375–406 (2016)
8. Hull, R., Damaggio, E., de Masellis, R., Fournier, F., Gupta, M., Heath,III, Fenno Terry, Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P.N., Vaculín, R.: Business Artifacts with Guard-Stage-Milestone Lifecycles: Managing Artifact Interactions with Conditions and Events. In: 5th ACM Int'l Conf. on Distributed Event-based System (DEBS), 2011. pp. 51–62. ACM (2011)
9. Hull, R., Damaggio, E., Fournier, F., Gupta, M., Heath,III, Fenno Terry, Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P.N., Vaculín, R.: Introducing the Guard-Stage-Milestone Approach for Specifying Business Entity Lifecycles. In: 7th Int'l Workshop on Web Services and Formal Methods (WS-FM) 2010. Lecture Notes in Computer Science, vol. 6551, pp. 1–24. Springer (2011)
10. Künzle, V., Reichert, M.: A Modeling Paradigm for Integrating Processes and Data at the Micro Level. In: 12th Int'l Working Conf. on Business Process Modeling, Development and Support (BPMDS). pp. 201–215. Lecture Notes in Business Information Processing, Springer (2011)
11. Künzle, V., Reichert, M.: PHILharmonicFlows: Towards a Framework for Object-aware Process Management. Journal of Software Maintenance and Evolution: Research and Practice 23(4), 205–244 (2011)
12. Nigam, A., Caswell, N.S.: Business Artifacts: An Approach to Operational Specification. IBM Systems Journal 42(3), 428–445 (2003)
13. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: Full Support for Loosely-Structured Processes. In: 11th IEEE Int'l Conf. on Enterprise Distributed Object Computing (EDOC). p. 287 (2007)
14. Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., Reijers, H.A.: Imperative versus Declarative Process Modeling Languages: An Empirical Investigation. In: Business Process Management Workshops: BPM 2011 Int'l Workshops, Revised Selected Papers, Part I. pp. 383–394. Springer (2012)
15. Reijers, H.A., Rigter, J.H.M., van der Aalst, W.M.P.: The Case Handling Case. International Journal of Cooperative Information Systems 12(03), 365–391 (2003)
16. Steinau, S., Künzle, V., Andrews, K., Reichert, M.: Coordinating Business Processes Using Semantic Relationships. In: 19th IEEE Conf. on Business Informatics (CBI). pp. 33–43. IEEE Computer Society Press (2017)
17. van der Aalst, W.M.P., Weske, M., Grünbauer, D.: Case Handling: A New Paradigm for Business Process Support. Data & Knowledge Engineering 53(2), 129–162 (2005)