Enabling Conformance Checking for Object Lifecycle Processes

Marius Breitmayer[1 [0000−0003−1572−4573]], Lisa Arnold[1 [0000−0002−2358−2571]], and Manfred Reichert[1 [0000−0003−2536−4153]]

1 Institute of Databases and Information Systems, Ulm University, Germany
{marius.breitmayer,lisa.arnold,manfred.reichert}@uni-ulm.de

Abstract. In object-aware process management, processes are represented as multiple interacting objects rather than a sequence of activities, enabling data-driven and highly flexible processes. In such flexible scenarios, however, it is crucial to be able to check to what degree the process is executed according to the model (i.e., guided behavior). Conformance checking algorithms (e.g., Token Replay or Alignments) deal with this issue for activity-centric processes based on a process model (e.g., specified as a petri net) and a given event log that reflects how the process instances were actually executed. This paper applies conformance checking algorithms to the behavior of objects. In object-aware process management, object lifecycle processes specify the various states into which corresponding objects may transition as well as the object attribute values required to complete these states. The approach accounts for flexible lifecycle executions using multiple workflow nets and conformance categories, therefore facilitating process analysis for engineers.

Keywords: data-centric process management · conformance checking · process analysis · object lifecycle processes

1 Introduction

Activity-centric approaches to business process management focus on the control-flow perspective of business processes, i.e., the order in which individual activities shall be executed. Consequently, activity-centric processes consist of activities that must be executed in a pre-specified order. While activities may require data during their execution, their actual specification (i.e., the data provided during activity execution) is considered as a black-box. Alternative paradigms such as data-centric and -driven process management [21] represent a process in terms of multiple interacting objects, allowing for greater flexibility through the use of declarative rules and generated forms. The individual behavior of an object is usually data-driven and described in terms of lifecycle processes. A lifecycle process specifies the states an object may transition during its lifecycle and the data required to complete each state. It, therefore, enables a white-box approach regarding process data. Examples of object-centric and data-driven process management approaches include artifact-centric processes [11], case handling [5], and

object-aware process management [16]. Despite the inherent flexibility of object-centric and data-driven approaches, the problem of not always executing a lifecycle process according to its pre-specified behavior applies to this paradigm as well. Deviations may be caused by users behaving differently than expected, ad-hoc behavioral changes [7], or errors introduced during the modeling or deployment of the lifecycle processes. Dynamic behavioral changes, for example, enable a variety of runtime adaptations of the lifecycle process model of a particular object. Examples include the insertion, reordering and deletion of lifecycle states, the insertion or deletion of object attributes, or objects in general. Furthermore, dynamic changes may be applied to individual objects and lifecycle instances (i.e., ad-hoc changes), respectively, as well as to lifecycle models in general (i.e., lifecycle evolution).

Another layer of complexity for checking conformance of object-centric and data-driven processes is their inherent flexibility. In a nutshell, the lifecycle process modeled for each business object describes its *guided* behavior, while accounting for *tolerated* state transitions. Nevertheless, there exist additional executions, which deviate from the modeled behavior, but correspond to correct executions of a lifecycle process as well. Moreover, the latter may occur within individual lifecycle states (i.e., when setting the attributes by filling corresponding form fields) as well as at transitions between them.

Assume a *Student* submits a solution to an exercise using a form. As long as all required form fields are set, the submission may be handed in. The order in which the form is filled, however, is arbitrary allowing for deviations from the underlying lifecycle model that was used to generate the form and its logic (i.e., its *guided* behavior). In a nutshell, the execution of object lifecycle processes operates within implicitly defined boundaries, and, therefore, tolerates certain deviations during lifecycle execution.

The approach presented in this paper is capable of identifying which object lifecycle process executions conform with the guided behavior of lifecycle processes, which executions are tolerated due to the built-in flexibility of lifecycle processes, and which executions constitute deviations from the lifecycle process.

The paper is structured as follows: Section 2 introduces PHILharmonicFlows, our approach to object-centric and data-driven process management. Section 3 describes the problem addressed by the paper. Section 4 describes the granularity and flexibility of object lifecycle processes and discusses how we can formally represent the latter through various workflow nets. In Section 5, we introduce conformance categories derived from conformance checking results in the context of object lifecycle processes. Section 6 evaluates our approach using multiple event logs. In Section 7, we relate our work to existing approaches for conformance checking. Section 8 summarizes the paper and provides an outlook.

2 Fundamentals

PHILharmonicFlows enhances the concept of object-centric and data-driven process management with the concept of *objects*. Each real-world business object is

represented as one such object. The latter comprises data, represented in terms of *attributes,* and a state-based process model describing object behavior in terms of an *object lifecycle model* (cf. Fig. 1).

The data- and process-aware e-learning system PHoodle, a sophisticated application implemented with PHILharmonicFlows, for example, includes objects such as *Lecture*, *Exercise,* and *Submission*. For the *Submission* object (cf. Fig. 1), attributes include *Exercise*, *E-Mail*, *Files*, and *Points*. The corresponding object lifecycle process is shown in Fig. 1. It describes the object behavior in terms of *states* (*e.g., Edit*, *Submit*, *Rate*, and *Rated*) as well as *state transitions*. Furthermore, each state may comprise several *steps* (e.g., steps *Exercise*, *E-Mail,* and *Files* in state *Edit*), with each step referring to exactly one object attribute. In other words, the steps of a lifecycle process define which attributes need to be written before completing the state and transitioning to the next one.

At runtime, a lifecycle allows for the dynamic and automated generation of forms (cf. Fig. 1). Accordingly, data acquisition in PHILharmonicFlows is based on the information modeled in both states and steps.

The lifecycle of a *Submission* object (cf. Fig. 1) can be interpreted as follows:

Edit is the initial state of the *Submission* object as it has no incoming transitions. After a student has provided data for steps *Exercise, E-Mail* and *Files*, the *Submission* may transition to state *Submit*.
State *Submit*, in turn, shall enable students to alter their submission prior to the exercise deadline by following the backwards transition. This allows returning back to state *Edit*, hence enabling changes to attributes *Exercise, E-Mail* and *Files*. A *Submission* automatically transitions from state *Submit* to *Rate* once the exercise deadline is reached, and tutors may then rate the final submission. In state *Rate*, a *Tutor* may read the provided attribute values from previous steps, provide data for step *Points,* and transition the submission to state *Rated*. Rated is the end state in which students may check their points.

This simple example emphasizes the importance of data executing an object lifecycle process. While an object instance may only be in one active state at a time, we also support choices [7, 20].

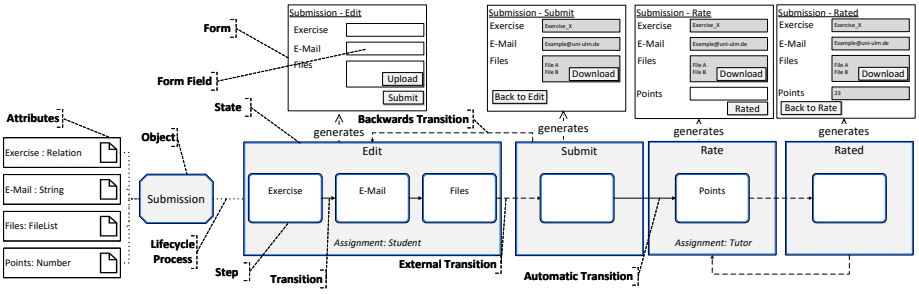

Fig. 1. Example Object Lifecycle Process of Object *Submission*

Generally, a business process not only comprises one single object, but involves multiple objects (e.g., *Submissions*, *Lectures* or *Exercises*) and their corresponding lifecycles. In PHILharmonicFlows, these objects are captured in a conceptual *data model* together with their semantic relations (including cardinality constraints) [16]. A semantic relation denotes a logical association between two objects (e.g., a relation between a *Lecture* and an *Exercise* implies that multiple exercises may be related to a single lecture). At runtime, each object may be instantiated multiple times, each representing an individual *object instance* [7]. The lifecycle processes of different object instances are then executed concurrently. Additionally, relations (e.g., between a *Lecture* instance and an *Exercise* instance) are instantiated enabling associations between two object instances. This results in novel information (e.g., one *Exercise* instance belongs to another *Lecture* instance), and intertwines the executed instances [16].

3 Problem Statement

Conformance checking leverages information from process event logs to correlate a process model with reality in order to assess its quality with respect to the behavior documented in the event log [2]. Thus, conformance checking measures how the recorded behavior of a process fits to its modeled representation by, for example, calculating a corresponding fitness value. Fitness measures to which extend a model can represent the behavior documented in an event log. This requires different representations of a process with respect to the recorded (e.g., an event log) and modeled behavior (e.g., a Petri net of a lifecycle process).

For activity-centric processes, where activities are mostly considered as black-boxes, existing conformance checking algorithms relate a process model to an event log. Deviations from the process model are identified reducing the calculated fitness value between event log and process model. In the context of object-aware process management, where the behavior of objects is modeled using a white-box approach (i.e., object behavior is explicitly modeled) and the execution of object lifecycle processes is data-driven (i.e., based on the availability of data) this problem is of great importance as well. However, due to the object-centric and data-driven processing of object lifecycles and the flexibility offered in this context [7], conformance issues are more challenging to address.

When processing object lifecycles, executions may deviate from the modeled lifecycle process, but still remain correct executions, due to the built-in flexibility of lifecycle processes. For example, the attributes of the form generated for state *Edit* in Fig. 1 may be filled in any order to complete the state (though there is a guidance in which order the form fields shall be filled according to the pre-specified sequence of steps within a state) or attribute values may be changed after having been set before. Additionally, *Submission* objects may return to previous states using backwards transitions (cf. Fig. 1). Both scenarios reflect tolerated execution behavior, but also deviations from the guided lifecycle behavior (cf. Tables 1 and 2).

Consequently, conformance checking of object lifecycle processes must account for both the guided and the tolerated lifecycle executions for individual states as well as the transitions between states. Tolerated executions are specified with respect to the order of states (i.e., the order in which single forms shall be processed) as well as the order of the steps within a state (i.e., the order in which fields within a form are organized). Tables 1 and 2 illustrate the different behavioral categories on two granularity levels for object lifecycle processes. Process engines capable of executing object-centric processes such as PHILharmonicFlows [16] or FLOWer [5] may generate all three behavior categories at runtime through the use of advanced concepts such as dynamic changes [6].

Conformance checking for object-aware lifecycle processes is multi-dimensional. On one hand, several levels of granularity exist at which fitness needs to be measured (state- and step-level, cf. Section 4). On the other, for each granularity level, it needs to be distinguished between guided and tolerated behavior to identify actual deviations. Consequently, a single fitness metric might not be sufficient to cover both dimensions.

Table 1. State Level Behavior

Behavior	Description	Example (cf. Fig. 1)
Guided	The lifecycle reaches its end state without following any backwards transitions during lifecycle execution.	< Edit, Submit, Rate, Rated >
Tolerated	The lifecycle reaches its end state, but backwards transitions were chosen during lifecycle execution.	< Edit, Submit, Edit, Submit, Rate, Rated, Rate, Rated >
Deviating	The lifecycle transitions to a non-reachable or unspecified state during its execution.	< Edit, Submit, Edit, Rated > < Edit, Submit, NewState, Rate, Rated >

Table 2. Step Level Behavior

Behavior	Description	Example (State Edit of Fig. 1)
Guided	All fields of the form of a lifecycle state are filled according to the pre-specified order of steps.	< Exercise, E-Mail, Files >
Tolerated	All mandatory fields of the form have been filled prior to state completion.	< E-Mail, Exercise, Files > < Files, E-Mail, Exercise >
Deviating	Required steps have been skipped or additional steps (i.e., attributes) have been added.	< Files> < Files, Exercise, Points >

4 Granularity and Flexibility of Lifecycle Processes

When analyzing object lifecycle executions, we need to account for the granularity (i.e., state and step level), while distinguishing between guided, tolerated and deviating behavior (cf. Tables 1 and 2). We, therefore, transform an object lifecycle process into a set of workflow nets, while accounting for granularity as well as the various degrees of flexible execution.

4.1 Granularity of Object Lifecycle Processes

To account for the granularity of object lifecycle processes, we analyze the behavior of each lifecylce process on two granularity levels, i.e., state and step level (cf. Fig. 2). Concerning the state level, we focus on the transitions between states (e.g., state *Edit* must be completed before state *Submit* may be activated). On step level, we analyze the logic of the steps within a state. Note that this logic is used to guide users through a form. Fig. 2 depicts the two granularity levels. The transformation of an object lifecycle process into a set of workflow nets of different granularity allows checking the conformance with respect to different levels of an object lifecycle (i.e., state and step level) separately. Furthermore, we are able to categorize deviations with respect to their origin, i.e., we can analyze whether a deviation results from unplanned state changes or from the flexible processing of a single state (i.e., the processing of its form). In turn, this allows for a more fine-grained conformance checking enabling data-driven process improvement. By solely considering the granularity, we are able to identify the origin of a deviation. However, we are unable to distinguish whether or not the latter are *tolerated* due to built-in flexibility. We therefore need to consider flexibility on both levels as well.

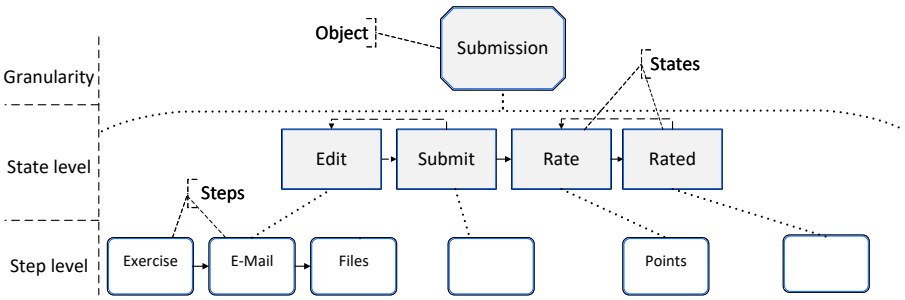

Fig. 2. Granularity of Object Lifecycle Processes (derived from Fig. 1)

4.2 Flexibility in Object Lifecycle Processes

In general, object lifecycle processes define the *guided behavior* of an object and provide corresponding user guidance during lifecycle execution. However, due

to their flexible execution nature [20], object lifecycle processes allow for *deviating behavior* at both granularity levels (cf. Tables 1 and 2). When checking the conformance of an object lifecycle process with an event log, differentiating between these categories offers promising perspectives for lifecycle improvement on one hand, but introduces additional challenges on the other. Note that conformance checking of object lifecycle processes must therefore account for both the granularity levels and the flexible execution behavior.

We address this challenge by distinguishing between *guided* and *tolerated behavior.* Accordingly, we use multiple workflow nets that can be derived from lifecycle processes on both granularity levels. This allows distinguishing between guided, tolerated, and deviating behavior on both granularity levels.

Definition 1. *Workflow Net [1]*
A Workflow Net is a Petri net N = (P,T,F, i, o) where:
P constitutes a finite set of places and T a finite set of transitions, $P \cap T \neq \emptyset$*,*
$F \subseteq (P \times T) \cup (T \times P)$ *represents a set of directed arcs, called flow relation.*
i is the source place ($\bullet i = \emptyset$) *and o constitutes the sink place* ($o\bullet = \emptyset$)
All other nodes are on a path from i *to o.*

Note that the workflow net depicted in Fig. 3 contains 6 places, 5 transitions, and 10 arcs. We use multiple workflow nets as representations of the lifecycle process behaviors described in Tables 1 and 2. This, in turn, allows us to distinguish between guided, tolerated, and deviating behavior.

4.3 Flexibility on State Level

On the state level, guided behavior is affected when backwards transitions are followed during lifecycle execution (cf. Table 1).

Guided state level behavior corresponds to activate the states of the lifecycle process exactly according to its specified order (cf. Fig. 1 and Table 1). By following a backwards transition, one may return to a preceding state, which has already been passed. In turn, this indicates that this state has not been properly processed such that object attribute changes become necessary (e.g., by uploading an updated file and assigning it to attribute *Files* in state *Edit*). In one such scenario of the submission lifecycle process (cf. Fig. 1), students may alter their submission by jumping back to state *Edit* following the corresponding backwards transition. When checking conformance, such backwards transitions still correspond to tolerated, but not to guided behavior (cf. Table 1). Regarding the lifecycle process from Fig. 1, the guided behavior on the state level translates to the workflow net displayed in Fig 3. Each state is translated to a place in the workflow net, external and automatic state transitions are translated to transitions between these places, whereas backwards transitions are neglected (i.e., they correspond to tolerated behavior). Note that choices between states may be represented as well.

Tolerated state level behavior covers backwards transitions, which allow returning to a previous state to account for foreseeable exceptions during lifecycle

Fig. 3. Guided State Level Workflow Net for Object Submission

process execution as well. While these (still planned) deviations are not covered by the *guided* state behavior (cf. Fig. 3), their execution at runtime is still *tolerated*. Considering the Submission lifecycle process (cf. Fig 1), for example, tolerated behavior allows returning from state *Submit* to state *Edit* as well as from state *Rated* to *Rate*. This enables two additional scenarios. First, students may return their submission to state *Edit*, which allows them to change their previous submissions (e.g., update uploaded files). Second, tutors may return submissions to state *Rate,* which allows changing the value of the lifecycle process step *Points*, e.g., if the tutor overlooked mistakes in a previously rated submission. While the lifecycle model from Fig. 1 accounts for such scenarios, the latter indicate that previous state completions were incorrect (e.g., upload of a wrong file). We generate the "tolerated net" (cf. Fig. 4) similar to the guided behavior, but do not neglect backwards transitions. Algorithm 1 describes the generation of both guided and tolerated nets on state granularity in pseudo code.

Fig. 4. Tolerated State Level Workflow Net for Object Submission

Algorithm 1 Workflow Net Generation Algorithm on State Level Granularity

Require: OLP, NetType ▷ Object lifecycle process, guided or tolerated behavior
  $PetriNet \leftarrow$ **new**
  $PetriNet.addPlace(source)$
  $PetriNet.addPlace(sink)$
  for all *states* in OLP **do**
    $PetriNet.addPlace(state)$
  end for
  for all t in OLP **do** ▷ Transitions of OLP
    if $t.source.state \neq t.target.state$ **then** ▷ Transition is external or backwards
      if $NetType = guided$ AND $t.type = backwards$ **then**
        //Do Nothing
      else
        $PetriNet.addTransition(t)$
        $PetriNet.addArcs(t.source, t.target)$ ▷ Arcs to connect t with in- and output places
      end if
    end if
  end for
  $PetriNet.makeWFN()$ ▷ Connect sink and source to places of first and last states

4.4 Flexibility on Step Level

As opposed to the state level, step level behavior covers intra-state behavior, i.e., the steps of a specific state and the order and constraints for their execution.

In PHILharmonic Flows, the behavior on step level is reflected by the control flow logic for processing the corresponding form. The behavior on step level is therefore directly connected to the actual data acquisition, i.e., steps of a state and their order are used to automatically generate role-specific forms at runtime.

Each step of a state corresponds to a single attribute that may be set while the state is active. A state may only be completed once all mandatory attributes have been set, i.e., its corresponding form has been properly filled. Note that the transitions between the steps of a given state are used to organize fields in the generated forms (and cursor control). For the step level, guided behavior means obeying the pre-specified execution logic of the steps when setting the attributes (cf. Table 2). Each attribute may be changed any number of times when processing the respective form field, and conditional attributes are possible as well. Considering the generated forms, guided behavior corresponds to the form being filled according to the pre-specified order of steps (cf. Table 2). Fig. 5 depicts the guided net derived for state *Edit* of the *Submission* object.

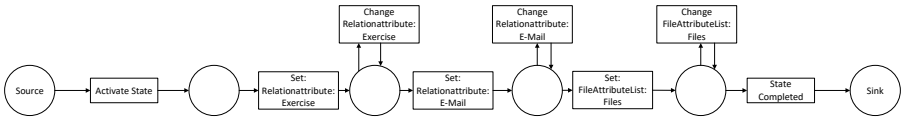

Fig. 5. Guided Step Level Workflow Net for State Edit

For the tolerated state level behavior there only exist two constraints. First, an attribute (i.e., a step) needs to be set before it may be changed. Second, a state may only be completed once all mandatory attributes have been set. The workflow net depicted in Fig. 6 represents this behavior for state *Edit* of the *Submission* object (cf. Fig. 1).

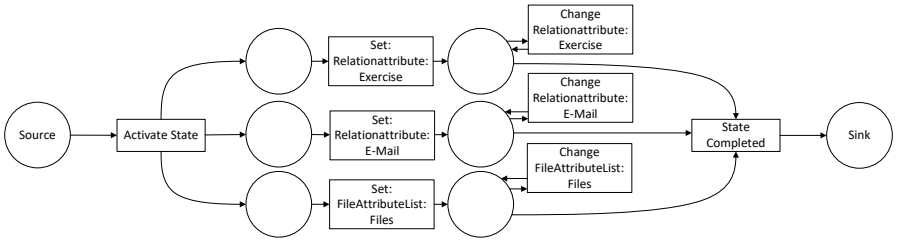

Fig. 6. Tolerated Step Level Workflow Net for State Edit

On step level, conditional steps (e.g., suppose a statement would be required if step *Points* in state *Rate* is below a certain threshold) may be represented through choice constructs and corresponding, additional state completion transitions. Algorithm 2 generates both the guided and tolerated workflow net (cf. Fig 6) on step level in pseudo code.

Algorithm 2 Workflow Net Generation Algorithm on Step Level Granularity

```
Require: OLP, NetType, OLPstate        ▷ object lifecycle process, guided/tolerated, selected state
  PetriNet ← new
  PetriNet.addPlace(source)
  PetriNet.addPlace(sink)
  for all step in OLP.getSteps(OLPstate) do
      PetriNet.addBehavior(step, NetType)      ▷ one or two places & transitions (Set & Change)
  end for
  if NetType = tolerated then
      PetriNet.addAndConnect()   ▷ Use tolerated syntax to connect places with Sink and Source
  else
      for all t in OLP do                                                    ▷ Transitions of OLP
          if t.source.state = t.target.state AND t.source = OLPState then
              PetriNet.connectSteps(t)                ▷ Connect steps according to transition of OLP
          end if
      end for
  end if
  PetriNet.makeWFN()              ▷ Add & connect one place (guided), connect sink and source
```

5 Conformance Checking of Lifecycle Processes

After having shown how to generate the workflow nets for each granularity level while at the same time accounting for lifecycle flexibility, we utilize these nets to check conformance for each level individually. On both granularity levels, however, using workflow nets that reflect guided behavior (e.g., Figs. 3 or 5) only enables us to check whether an object lifecycle was executed according to the guided behavior (i.e., both states and steps are executed based on the order described by the guided behavior). Deviations from this behavior may, in turn, be tolerated by the object lifecycle process due to its built-in flexibility.

A similar scenario arises when only using workflow nets representing tolerated behavior (e.g., Figs. 4 or 6), which represents all lifecycle executions that may be realized without any interventions from process supervisors (i.e., all correct executions). Based on the fitness value between the workflow nets representing tolerated behavior on one hand and an event log on the other, we are only able to check whether an object lifecycle process instance changed states, or a state was processed according to the tolerated behavior. We are unable to figure out which object lifecycle process instances were executed according to guided behavior (i.e., whether states were transitioned, or steps were processed according to the guided behavior).

When considering both nets of a granularity level in combination (e.g., Figs. 3 and 4 or 5 and 6), we can calculate two fitness values for both the state and

the step level (i.e., for each state). One fitness value corresponds to the fitness regarding guided behavior, the other to tolerated behavior for a selected level of granularity (i.e, state or step level). In combination, the two fitness values allow categorizing each lifecycle process instance according to the behavior and granularity levels described in Tables 1 and 2. Furthermore, we can distinguish whether deviations from the guided behavior are due to exceptions or due to the flexibility inherent to object-aware processes.

Note that we use Alignments [3] to calculate the resulting fitness values - alignments are the de-facto standard approach to evaluate fitness [12] and are able to cope with log entries unrelated to the lifecycle process model (e.g., executing an unmodeled step or state). Alignments connect execution traces with a valid execution sequence from a process model through log, model or synchronous moves [3, 10]. Fitness is then calculated using costs of identified log and model moves. Other algorithms (e.g., Token Replay [8]) may be used as well.

5.1 Conformance Categories

When categorizing lifecycle process executions, we either focus on the state changes (i.e., state level) or individual states (i.e., step level) recorded in the event log. Usually, event data are recorded during the execution of a process and consist of cases and activities [2]. In the context of object lifecycle processes, we use cases to identify individual object instances. A case comprises information on object lifecycle process states and steps (i.e., on whether the form field corresponding to a step has been set, changed or an object instance has transitioned to another state) in terms of activities. Note that this allows for the use of existing conformance checking algorithms (e.g., alignments) to lifecycle processes. During conformance checking, we consider event log subsets based on the granularity levels of each lifecycle process. Fig. 7 depicts an example event log from the *Phoodle* scenario.

	User	Case	Filter		Activity		Attribute Value	Timestamp
1	**User**	**Object Instance**	**Object State**	**Object Type**	**Method**	**Parameter 1**	**Parameter 2**	**Timestamp**
2	person78@uni-ulm.de	person78@uni-ulm.de_Di 12:00 gerade	Edit	Tutorial	Activate State			29.04.2019 23:56:17
3	person78@uni-ulm.de	person78@uni-ulm.de_Di 12:00 gerade	Edit	Tutorial	Set: RelationAttribute	Lecture	Datenbanken	29.04.2019 23:56:20
4	person78@uni-ulm.de	person78@uni-ulm.de_Di 12:00 gerade	Edit	Tutorial	Set: StringAttribute	Slot	Diensta	29.04.2019 23:56:24
5	person78@uni-ulm.de	person78@uni-ulm.de_Di 12:00 gerade	Edit	Tutorial	Change StringAttribute	Slot	Dienstag 1	29.04.2019 23:56:32
6	person78@uni-ulm.de	person78@uni-ulm.de_Di 12:00 gerade	Edit	Tutorial	Change StringAttribute	Slot	Dienstag 12	29.04.2019 23:56:34
7	person78@uni-ulm.de	person78@uni-ulm.de_Di 12:00 gerade	Edit	Tutorial	Change StringAttribute	Slot	Dienstag 12.00	29.04.2019 23:56:43
8	person78@uni-ulm.de	person78@uni-ulm.de_Di 12:00 gerade	Edit	Tutorial	Change StringAttribute	Slot	Dienstag 12.00 - 12.30	29.04.2019 23:56:49
9	person78@uni-ulm.de	person78@uni-ulm.de_Di 12:00 gerade	Edit	Tutorial	Change StringAttribute	Slot	Dienstag 12.00 - 12.30 herade	29.04.2019 23:57:01
10	person78@uni-ulm.de	person78@uni-ulm.de_Di 12:00 gerade	Edit	Tutorial	Change StringAttribute	Slot	Dienstag 12.00 - 12.30 gerade K	29.04.2019 23:57:11
11	person78@uni-ulm.de	person78@uni-ulm.de_Di 12:00 gerade	Edit	Tutorial	Change StringAttribute	Slot	Dienstag 12.00 - 12.30 gerade KW	29.04.2019 23:57:15
12	person78@uni-ulm.de	person78@uni-ulm.de_Di 12:00 gerade	Edit	Tutorial	State completed			29.04.2019 23:57:16
13	person78@uni-ulm.de	person78@uni-ulm.de_Di 12:00 ungerade	Edit	Tutorial	Activate State			29.04.2019 23:57:21
14	person78@uni-ulm.de	person78@uni-ulm.de_Di 12:00 ungerade	Edit	Tutorial	Set: RelationAttribute	Lecture	Datenbanken	29.04.2019 23:57:23
15	person78@uni-ulm.de	person78@uni-ulm.de_Di 12:00 ungerade	Edit	Tutorial	Set: StringAttribute	Slot	Dienstag	29.04.2019 23:57:26

Fig. 7. Event Log Example for State Edit of Object Tutorial (anonymized due to GDPR)

Concerning the state level, we consider those event log entries that are related to state changes (i.e., the transitions in Figs. 3 + 4). In contrast, the step

level conformance checking considers events related to the writing of object attributes (i.e., the transitions in Figs. 5 + 6). Consequently, an event log subset either documents how object instances transitioned between object states or how individual states were processed (i.e., in which order the fields of its corresponding form, i.e., object attributes, were set). Aligning a log subset with the two corresponding workflow nets (cf. Figs 3 + 4 for the state level of object lifecycle *Submission* or Figs. 5 + 6 for state *Edi*t), we obtain two fitness values. Based on the latter, we can categorize each lifecycle process instance into one of the following categories, depending on the behavior captured in the event log:

- **Guided behavior:** The fitness value obtained for the guided net equals to 1. The tolerated fitness also equals to 1 as it generalizes the guided behavior.
  - **State level:** the state changes of the object lifecycle process instance fully comply with the guided behavior.
  - **Step level:** the steps of a state were executed according to the guided behavior (i.e., the generated form was filled in from top to bottom).
- **Tolerated behavior:** The fitness from the guided net is below 1, but the fitness from the tolerated net still equals 1. Deviations from the guided behavior model captured in this category are *correct* executions due to the built-in flexibility. However, process improvements might be possible.
  - **State level:** the object lifecycle process instance changed states correctly, though its execution utilizes the built-in flexibility of lifecycle processes (e.g., by follwoing backwards transitions)
  - **Step level:** the steps were executed correctly with respect to the built-in flexibility of the generated forms (e.g., the form was filled in any order).
- **Deviating behavior:** Both fitness values are below 1. Deviations occurred that are not tolerated by the built-in flexibility of object lifecycle processes.
  - **State level:** deviations include, but are not limited to states not being reachable from the currently active state, to ad-hoc (i.e., not prespecified) backwards transitions, or to state changes not allowed by the object lifecycle process.
  - **Step level:** deviations may result from states that are completed while not all required attributes (i.e., steps) are set, steps not being part of the lifecycle process state are set, steps are changed after the state is completed or steps are changed before being set.

5.2 Leveraging Conformance Categories for Process Analysis

When analyzing object-centric and data-driven processes, the introduced conformance categories provide useful insights for process engineers into potential model improvements. In turn, this facilitates problem detection through the discovery of actual deviations and tolerated behavior. To identify whether object lifecycle instances deviate from the lifecycle model (e.g., through ad-hoc changes executed by process supervisors) or are executed according to the tolerated behavior yields useful information for improving and evolving lifecycle processes.

Deviating behavior on the state level, for example, indicates that ad-hoc changes were required during lifecycle process execution. In this scenario, the

lifecycle model does not allow for all the behavior required in practice, i.e., is has turned out to be too restrictive during the processing of certain state transitions.

Furthermore, tolerated behavior in the processing of a certain state may indicate that the ordering of the steps within this state might not be optimal.

Conformance categories are capable of prioritizing improvement efforts. On both granularity levels, tolerated behavior captures behavior inherently supported by the built-in flexibility and, thus, not requiring any interventions by process supervisors. This indicates improvement potential with respect to usability (i.e., forms may be optimized by reordering steps of a state with highly tolerated behavior). Deviating behavior, in turn, covers behavior due to either implementation mistakes or explicit interventions by process supervisors (e.g., through ad-hoc changes at runtime). As a result, when analyzing object lifecycle processes, deviating behavior should be investigated with higher priority. Our approach provides guidance for process engineers in analyzing and improving lifecycle processes.

6 Experimental Evaluation[1]

To demonstrate the applicability of our approach for checking the conformance of lifecycle processes, we implemented a proof-of-concept prototype. The latter includes a translator that enables the generation of the different workflow nets based of an object-aware process [16]. The implementation of this translator uses *python* and the *pm4py* framework [9]. The implemented algorithms are illustrated in terms of pseudo-code in Algs. 1 and 2. To evaluate the conformance checking approach described in Section 5, we used multiple event logs to check their conformance with each of the derived workflow nets. First, we generated event logs using the extended Petri net playout feature of *pm4py* for the derived workflow nets to simulate all allowed process executions. For this purpose, we generated all traces that are allowed according to each workflow net, up to a trace length of 10. However, any other trace length would be possible as well. The resulting event logs contain 6 (tolerated state level), 1 (guided state level), 8334 (tolerated step level) and 56 (guided step level) traces respectively.

Table 3. Object Submission - Playout

% of traces	State Level		Step Level (State *Edit*)	
	Guided Log	Tolerated Log	Guided Log	Tolerated Log
Guided Behavior	100 %	16.66 %	100 %	0.6719 %
Tolerated Behavior	0 %	83.34 %	0 %	99.3281 %

[1] All event logs are provided at https://www.researchgate.net/project/Lifecycle-Conformance-Checking-RCIS

The results from Table 3 show that we are able to distinguish between tolerated and guided behavior using the described workflow nets for conformance checking. The generated event logs, however, do not contain actual deviations (cf. Section 5) as they represent all allowed playouts of the derived workflow nets. All simulated traces, therefore, belong to either guided or tolerated behavior. On state level (cf. Table 3), 16.66% of the tolerated traces fit the guided behavior (i.e., Category Guided Behavior State Level for Tolerated Log). Concerning the step level granularity of state *Edit* (cf. Table 3), only 0.6719% of the tolerated traces fit to the actual guided behavior (i.e., Category Guided Behavior Step Level for Tolerated Log). This indicates the high degree of flexibility an object lifecycle process allows for a state with only 3 steps and a trace length up to 10.

To evaluate whether the approach is able to identify deviating behavior in an event log, we generated an additional event log that contains deviating behavior as well. For this purpose, we randomly simulate behavior within an event log that may not only represent tolerated and guided, but also deviating behavior, by randomly picking from the set of transitions. In practice, such behavior can be observed in the context of ad-hoc changes or implementation mistakes (e.g., while collecting event logs). Alg. 3 indicates how we generated the event logs used for checking conformance. We generated event logs with 1000 traces of random length between 5 and 8 in order to group traces according to the categories presented in Section 5.

Algorithm 3 Algorithm to Generate Event Logs with Deviations

```
Require: PetriNet, TraceNumber, TraceLength        ▷ Petri net, number of log traces and length
  OriginalTransitions = PetriNet.transitions.copy()
  for trace = 0 to TraceNumber do
      Transitions = OriginalTransitions
      Eventlog.add(Initial State)                  ▷ Activate State or Source to first state
      for i = 0 to TraceLength do
          Transition = random.choice(Transitions)  ▷ Pick random transition from net
          if Transition = "Set:" then   ▷ Probibit, that one step is set multiple times in a trace
              Eventlog.add(Transition)
              Transition.remove(Transition)
          else
              Eventlog.add(Transition)
          end if
      end for
      Eventlog.add(FinalState)                     ▷ State completed or last state to sink
  end for
```

Table 4 shows the categories into which the randomly generated traces are assigned according to their behavior documented in the event log. We are not only able to differentiate between guided and tolerated behavior but can also identify deviating behavior with the approach. However, note that the event logs used in the sketched evaluation constitute two edge cases of object lifecycle process executions, as they either contain no deviations (cf. Table 3) or a high ratio of deviations (cf. Table 4).

We further evaluated the approach using an event log we collected from a real-world deployment of *Phoodle* (cf. Section 2) in which 133 students used the

Table 4. Categories for Object Submission - Random

% of traces (#)	state level	step level (state Edit)
Guided Behavior	0.1 % (1)	0.4 % (4)
Tolerated Behavior	0.2 % (2)	7.0 % (70)
Deviating Behavior	99.7 % (997)	92.6 % (926)

system during a university course over a period of 4 months (cf. Fig. 7). When applying the approach, all 51 lifecycle process instances of object *Tutorial* showed deviating step level behavior for state *Edit* (cf. Table 5). Upon closer inspection, according to the event log, attribute *Lecture* was set in state *Edit* (cf. lines 3 and 14 in Fig. 7), while the lifecycle process required attribute *Tutor*. In the next step, we repaired the event log to set the correct attribute, and thus 28 of 51 tutorial lifecycle process instances corresponded to guided and 4 to tolerated behavior. Note that the remaining 19 instances had additional deviations not related to the repaired deviation (e.g., attributes were changed after the state had been completed using ad-hoc changes [7]).

Table 5. Phoodle Log Tutorial - State Edit

% of traces (#)	Initial	Repaired
Guided Step Level Behavior	0 % (0)	54.90 % (28)
Tolerated Step Level Behavior	0 % (0)	7.85 % (4)
Deviating Step Level Behavior	100 % (51)	37.25% (19)

Overall, the evaluation has shown that we are able to pinpoint which granularity level of an object lifecycle process is non-conforming (i.e., deviations regarding state transitions or individual states). Additionally, we can account for the flexible (i.e., tolerated) execution of object lifecycle processes through the use of multiple workflow nets, therefore enabling sophisticated and holistic deviation detection.

7 Related Work

This work is related to two research areas: conformance checking and object-/data-centric process management. In [19], conformance checking is presented as multidimensional quality metrics for processes and their corresponding event logs. Furthermore, best-effort metrics to assess the different quality dimensions are introduced. One algorithm to check conformance is Token Replay [19], which can identify those parts of the process model and event log that fit together. Furthermore, it enables diagnostics related to deviations by replaying the event log

on a Petri net covering the execution behavior of the process model. Additional algorithms have emerged that enable conformance assessment by aligning process model and event log [3]. Alignments have already been adapted to various scenarios, e.g., large processes [18] or declarative processes [17]. Recently, another token replay approach emerged [8]. Finally, first approaches for discovering object-centric Petri nets have been proposed [4].

Some conformance checking approaches exist for artifact-centric conformance checking [13–15] as well. To some degree these approaches are similar to ours. However, differences arise due to the fact that we focus on business objects instead of proclets, artifacts, or UML diagrams. Compared to [13], our approach does not use UML state and activity diagrams to generate the Petri net. While [14] uses conformance checking, the presented approach is able to identify behavioral and interaction conformance with respect to proclets (Petri nets, including communication ports). As a result, no translation from proclets to Petri nets is required. The work presented in [15] focuses on the interaction between multiple artifacts rather than the behavior of object lifecycle processes in isolation. Furthermore, to the best of our knowledge, none of the existing approaches accounts for flexibility during conformance checking of data-centric and -driven processes.

8 Summary and Outlook

This paper presented an approach for checking the conformance of single object lifecycle processes. We introduced two granularity levels for enacting lifecycle processes granularity levels as well as built-in flexibility concepts. We then incorporate them during conformance checking to differentiate between guided, tolerated, and deviating behavior. Checking conformance with multiple nets allows categorizing each lifecycle execution based on the behavior captured in the event log. Furthermore, we are able to account for the flexible nature of object lifecycles through conformance categories that allow us to distinguish between deviations tolerated due to the flexibility of object lifecycles and actual deviations. Additionally, we can account for flexibility regarding transitions between states, and the behavior of individual states. When analyzing data-centric and -driven processes, conformance categories provide guidance for process engineers with respect to which parts of object lifecycle processes are of particular interest.

In future work, we plan to extend the presented approach in a two-fold manner: First, we plan to incorporate constraints between object lifecycle processes. This will allow us to further improve conformance checking of object-centric processes regarding the inter-object granularity level. The latter considers constraints between different object lifecycle processes, rather than lifecycle processes in isolation. Second, we want to provide detailed information on the origin of the deviation to further facilitate process improvement.

Acknowledgments This work is part of the SoftProc project, funded by the KMU Innovativ Program of the Federal Ministry of Education and Research, Germany (F.No. 01IS20027A)

References

1. van der Aalst, W.M.P.: Verification of workflow nets. In: Application and Theory of Petri Nets 1997. pp. 407–426. Springer (1997)
2. van der Aalst, W.M.P., Adriansyah, A., De Medeiros, A.K.A., Arcieri, F., Baier, T., Blickle, T., Bose, J.C., Van Den Brand, P., Brandtjen, R., Buijs, J., et al.: Process mining manifesto. In: Int' Conf' on BPM. pp. 169–194. Springer (2011)
3. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.: Replaying history on process models for conformance checking and performance analysis. WIREs Data Mining and Knowledge Discovery **2**(2), 182–192 (2012)
4. van der Aalst, W.M.P., Berti, A.: Discovering Object-centric Petri Nets. Fundamenta informaticae **175**(1/4), 1–40 (2020)
5. van der Aalst, W.M.P., Weske, M., Grünbauer, D.: Case handling: a new paradigm for business process support. DKE **53**(2), 129–162 (2005)
6. Andrews, K., Steinau, S., Reichert, M.: Enabling ad-hoc changes to object-aware processes. In: 22nd International Enterprise Distributed Object Computing Conference (EDOC 2018). pp. 85–94. IEEE Computer Society Press (October 2018)
7. Andrews, K., Steinau, S., Reichert, M.: Enabling runtime flexibility in data-centric and data-driven process execution engines. Information Systems **101** (2021)
8. Berti, A., van der Aalst, W.M.P.: A Novel Token-Based Replay Technique to Speed Up Conformance Checking and Process Enhancement, pp. 1–26. Springer (2021)
9. Berti, A., van Zelst, S.J., van der Aalst, W.M.P.: Process mining for python (pm4py): Bridging the gap between process- and data science. CoRR **abs/1905.06169** (2019)
10. Carmona, J., Dongen, B., Solti, A., Weidlich, M.: Conformance Checking: Relating Processes and Models (01 2018). https://doi.org/10.1007/978-3-319-99414-7
11. Cohn, D., Hull, R.: Business artifacts: A data-centric approach to modeling business operations and processes. IEEE TCDE **32**(3), 3–9 (2009)
12. Dunzer, S., Stierle, M., Matzner, M., Baier, S.: Conformance checking: A state-of-the-art literature review. CoRR **abs/2007.10903** (2020)
13. Estañol, M., Munoz-Gama, J., Carmona, J., Teniente, E.: Conformance checking in UML artifact-centric business process models. Softw. Syst. Model. **18**(4) (2019)
14. Fahland, D., de Leoni, M., van Dongen, B.F., van der Aalst, W.M.P.: Behavioral conformance of artifact-centric process models. In: Bus Inf Sys. Springer (2011)
15. Fahland, D., de Leoni, M., van Dongen, B.F., van der Aalst, W. M. P.: Conformance checking of interacting processes with overlapping instances. In: Business Process Management. pp. 345–361. Springer Berlin Heidelberg (2011)
16. Künzle, V., Reichert, M.: PHILharmonicFlows: towards a framework for object-aware process management. JSME **23**(4), 205–244 (2011)
17. de Leoni, M., Maggi, F.M., van der Aalst, W.M.P.: Aligning event logs and declarative process models for conformance checking. In: BPM. pp. 82–97. Springer (2012)
18. Munoz-Gama, J., Carmona, J., W. M. P. van der Aalst: Single-entry single-exit decomposed conformance checking. Information Systems **46**, 102–122 (2014)
19. Rozinat, A., W. M. P. van der Aalst: Conformance checking of processes based on monitoring real behavior. Information Systems **33**(1), 64–95 (2008)
20. Steinau, S., Andrews, K., Reichert, M.: Executing lifecycle processes in object-aware process management. In: Data-Driven Process Discovery and Analysis. pp. 25–44. Lecture Notes in Business Information Processing, Springer (2017)
21. Steinau, S., Marrella, A., Andrews, K., Leotta, F., Mecella, M., Reichert, M.: Dalec: A framework for the systematic evaluation of data-centric approaches to process management software. Software & Systems Modeling **18**(4), 2679–2716 (2019)