Data-Driven Evolution of Activity Forms in Object- and Process-Aware Information Systems

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. Object-aware processes enable the data-driven generation of forms based on the object behavior, which is pre-specified by the respective object lifecycle process. Each state of a lifecycle process comprises a number of object attributes that need to be set (e.g., via forms) before transitioning to the next state. When initially modeling a lifecycle process, the optimal ordering of the form fields is often unknown and only a guess of the lifecycle process modeler. As a consequence, certain form fields might be obsolete, missing, or ordered in a non-intuitive manner. Though this does not affect process executability, it decreases the usability of the automatically generated forms. Discovering respective problems, therefore, provides valuable insights into how object- and process-aware information systems can be evolved to improve their usability. This paper presents an approach for deriving improvements of object lifecycle processes by comparing the respective positions of the fields of the generated forms with the ones according to which the fields were actually filled by users during runtime. Our approach enables us to discover missing or obsolete form fields, and additionally considers the order of the fields within the generated forms. Finally, we can derive the modeling operations required to automatically restructure the internal logic of the lifecycle process states and, thus, to automatically evolve lifecycle processes and corresponding forms.

Keywords: data-centric process management, event log, process improvement, process enhancement, generated forms

1 Introduction

Activity-centric approaches to business process management (BPM) focus on the order in which the activities of a business process shall be executed (i.e., the control-flow perspective), whereas other perspectives, such as the data required during process execution, are considered as second-class citizens [17]. Moreover, the activities of a process are usually treated as a black box by the process execution environment. As a consequence, additional efforts, such as the manual specification of the user forms implementing a human task become necessary when implementing activity-centric processes. By contrast, data-centric and -driven approaches to BPM (see [20] for an overview) treat data as first-class citizens by representing a business process in terms of multiple interacting

objects with a particular focus on (data-driven) *object behavior* and *object interactions*. Usually, the data-driven behavior of a single object (e.g., order, invoice, or exercise) is described in terms of a *lifecycle process*, which specifies the allowed object states, the respective object transitions as well as the data required (i.e., object attributes to be set) to complete each step. In turn, this enables a white-box approach with respect to process data that allows for an increased flexibility due to declarative rules and automatically generated forms based on the respective lifecycle process logic decreasing implementation efforts. Examples of data-centric process management approaches include case handling [10], artifact-centric processes [2], and object-aware processes [15].

The automated generation of forms at runtime not only decreases implementation efforts, but also introduces challenges for lifecycle modeling. While forms are well established [7], the internal form logic is often unclear to the form modeler and implementer, respectively. In general, the order in which the fields of a form may be accessed (i.e., the logic for writing certain object attributes specified by a lifecycle state) is not always evident at lifecycle process modeling time. Moreover, end users might prefer a different sequence of filling the form fields than the one considered as being intuitive by the modeler. If the order of a generated form (i.e., the modeled sequence of writing object attributes within a state) is not intuitive for users, higher mental efforts as well as more user interactions are required and, thus, form completion times increase, while at the same time user satisfaction and effectiveness decrease [14].

In the context of data-centric and -driven process management, a lifecycle process specifies the sequence in which the various user forms as well as their form fields are displayed to users, including more complex logic (e.g., conditional form fields) as well. The order in which forms are displayed is specified by the logic between states, whereas the logic of the steps within a state determines the content of the corresponding generated form. When executing data-centric processes, event logs record about the order in which the form fields are actually filled. Thus, process mining techniques provide promising perspectives for evolving the user forms. Note that the ability to evolve user forms offers promising perspectives for evolving information systems.

The approach presented in this paper is capable of analyzing an event log, comparing it with the lifecycle process used to generate the forms, and discovering potential improvements that can be realized by adding, deleting or reordering the auto-generated forms and their corresponding fields. Moreover, the approach is able to derive the operations required to dynamically evolve the information system [4] and its lifecycle process, allowing for the auto-optimization of the generated forms at runtime.

This paper is structured as follows: Section 2 introduces fundamentals. Section 3 describes our proposed approach, whereas Section 4 elaborates on deriving corresponding positions. Section 5 describes how we identify improvements. In Section 6 we describe how we derive suitable improvement actions. Section 7 evaluates our approach. In Section 8, we relate our approach to existing approaches. Section 9 summarizes the paper and provides an outlook.

2 Backgrounds

This section introduces PHILharmonicFlows, our approach to object-centric and data-driven process management. Further, it introduces concepts for process model evolution and ad-changes used for form evolution.

2.1 PHILharmonicFlows

PHILharmonicFlows enhances the concept of data-centric and -driven process management with the concept of *objects*. In PHILharmonicFlows, each business object of the real world is represented as one object. An object, in turn, is described by its data and represented in terms of *attributes*. Its behavior is expressed by a state-based *object lifecycle process* model.

Based on PHILharmonicFlows we implemented *PHoodle*, a sophisticated data- and process-aware e-learning application, and ran it over one semester with a total of 137 users and 39890 transactions. This application includes objects such as *Lecture*, *Exercise, Attendance,* and *Submission*. Fig. 1 depicts the lifecycle process of object *Exercise* and the auto-generated form of the corresponding state *Edit*.

Each *state* of the lifecycle process (e.g., *Edit*, *Published*, *Past Due* and *End*), in turn, may comprise several *steps* (e.g., steps *Lecture*, *Name*, *Points*, *Due Date* and *Exercise Files* in state *Edit*). Each of these steps refers to a write access on a specific object attribute. In other words, the steps of a lifecycle process define the attributes required to complete the state. Once all required attributes have been written, the respective state is completed, and the object may transition to its next state.

At runtime, object lifecycle processes allow for the automated and dynamic generation of forms (cf. Fig. 1 for the form of state *Edit*) based on the order set out by the lifecycle process for the steps of the respective state. Accordingly, data acquisition is based on the information modeled in lifecycle processes. The auto-generated form of state *Edit*, which is shown in Fig. 1, orders the form fields according to the internal logic of state *Edit* in the depicted lifecycle process.

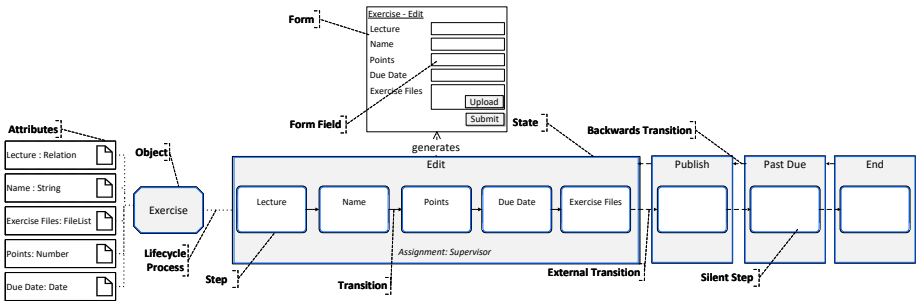

Fig. 1. Simplified Exercise Lifecycle Process with Generated Form for State Edit

Note that, in general, a business process not only comprises one single object, but involves multiple interacting objects such as *Submissions*, *Lectures* and *Exercises* as well as their corresponding lifecycle processes. In PHILharmonicFlows, a *data model* captures all relevant objects (including their attributes) and the semantic relations between them (including cardinality constraints) [15]. 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 [4]. The lifecycle processes of different object instances are then executed concurrently. Additionally, relations between object instances instantiated enabling associations between them resulting in a relational process structure at runtime [19]. This results in novel information and intertwines the executed instances [15].

2.2 Process Model Evolution and Ad-hoc Changes

Process Model Evolution [18] and *Ad-hoc changes* [4] allow performing runtime changes to object-aware processes, including lifecycle processes and, therefore, the auto-generated forms [4]. Amongst others, corresponding changes may include the insertion, deletion and reordering of both lifecycle states and steps.

Process Model Evolution Process model evolution is concerned with changes introduced to the process model by deploying updated process models to existing process instances [18]. In this context, *deferred process model evolution* is accompanied by the introduction of new process model versions, which may then co-exist with older model versions. Therefore, existing process instances may be executed according to the old (i.e., outdated) process model versions.

In contrast, *immediate process model evolution* tries to migrate running process model instances to the new model version, allowing for a greater flexibility at runtime. In PHILharmonicFlows, we implemented *immediate process model evolutions* [4], which additionally enable improvements of already running lifecycle process models (e.g., the insertion, deletion or reordering of the states and steps of a lifecycle process) [3]. In turn, this allows for the dynamic optimization of lifecycle processes, including the auto-generated forms, at runtime.

Ad-hoc Changes Ad-hoc changes constitute a particular type of *immediate process model evolution,* in which a specific running process model instance becomes changed.

Ad-hoc changes allow, for example, inserting, deleting, or reordering the steps within a state of a lifecycle process instance. This, in turn, allows users to deviate from the pre-specified process model in various ways, while also reducing model complexity as not every possible execution variant needs to be modeled in advance [3]. For an in-depth introduction to ad-hoc changes, we refer interested readers to [4].

3 Proposed Approach

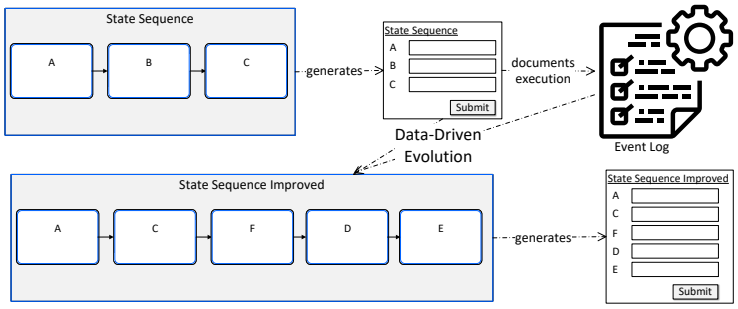

Fig. 2. Proposed Approach

The goals of our approach for evolving lifecycle processes as well as their auto-generated forms are two-fold: First, we want to identify in which way lifecycle processes can be improved to minimize ad-hoc changes such as the insertion and deletion of the states and steps. Ad-hoc changes usually require some intervention from process supervisors (e.g., the approval of insertions and deletions of lifecycle states and steps). Second, we want to improve lifecycle states and, thus, the auto-generated forms, concerning the control flow logic in which the steps of a state are organized and the order in which states may become active. This allows generating more intuitive forms, that are based on actual form executions rather than on the subjective perception of a modeler during lifecycle process specification.

To identify corresponding improvements, we analyze the actual interactions users have had with the implemented system documented in an event log. Note that during these interactions, users may utilize the process flexibility enabled by PHILharmonicFlows [4], such as filling auto-generated forms in an arbitrary order or initiating ad-hoc changes (e.g., by dynamically adding or deleting form fields). We (anonymously) document these user interactions with various object instances, for example, writing attribute *Points* in state *Edit* of object instance *Exercise2*, in an event log (cf. Fig. 7). The latter is then compared with the lifecycle process model, which, in turn, enables us to automatically evolve the lifecycle processes and, thus, the forms dynamically generated during their execution.

In such an event log, one may assign a *position* to each interaction documented. We enable the comparison between modeled and actual lifecycle process behavior by assigning positions to the states as well as the steps of a lifecycle processes. This way, we may compare the position of a state or step of the model with the one recorded in the event log to discover potential improvements with respect to both the order and assignment of steps and states. We are able to detect whether steps (i.e., form fields) are filled in the pre-specified order, in the pre-specified state, and whether states or steps are added or deleted at runtime due to ad-hoc changes. Consequently, we can identify actions for evolving and improving lifecycle process models and execute them using the concepts introduced in Section 2.2. Our approach is illustrated in Fig. 2.

4 Position-based Lifecycle and Event Log Representation

When comparing the lifecycle process executions captured in an event log with a given lifecycle process model, a suitable representation is required for both the event log and the lifecycle process model. This representation should enable an efficient comparison as well as the easy detection of deviations between actual behavior and the behavior captured in the lifecycle process model. In the following, we propose a *position-based approach* representing both event logs and the logic of lifecycle processes in a homogeneous way.

4.1 Lifecycle Process Step Positions

Each step is associated with two positions. The first one corresponds to its relative position within the state it belongs to (see the positions with red labels in Fig. 3), whereas the second position expresses the relative position of the corresponding state in the entire lifecycle process (see the positions with green labels in Fig. 3). Note that this distinction allows positioning lifecycle steps in relation to both the other steps of the corresponding state as well as the steps of other states.

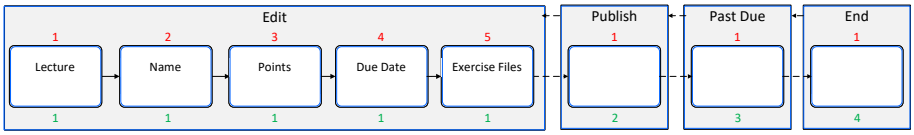

Fig. 3. Positions for Lifecycle Process Submission (Red: Step, Green: State)

As discussed in Section 2, lifecycle processes capture *object behavior* allowing for basic control flow patterns such as sequence and choice within and across states. While choices between states (e.g., to express that an object may transition to either state A or B) are possible, an object must not be in two states at the same time[1]. To be more precise, lifecycle processes must not contain parallel splits between states. Remember that the sequence of steps within an individual lifecycle state is utilized to auto-generate a form as well as its internal logic guiding users in filling the form fields (e.g., indicating the field to be edited next after writing a specific field). Due to the high runtime flexibility of both object-aware processes and auto-generated forms, however, users need not adhere to this guidance when filling the respective forms. As soon as all mandatory attributes are set, a state may be completed independent from the order in which the form fields (i.e., steps) were actually edited. Choices within a state are represented by displaying or hiding form fields at runtime.

In the following, we present the patterns that may be used to model the behavior of a lifecycle state, the forms that can be auto-generated from these patterns, and the positions assigned to the steps of the respective state.

[1] Note that PHILharmonicFlows allows for the concurrent processing of multiple lifecycle process instances (of same or different type) in the context of a multi-object business process. The concurrent processing is controlled by a coordination process.

Sequence If the steps of a state are organized sequentially (cf. Fig. 4), their position can be derived in a straightforward manner. To each step its position is assigned according to the order of the steps within the state (see the red numbers in Fig. 4). The form and its cursor control during form processing are organized accordingly.

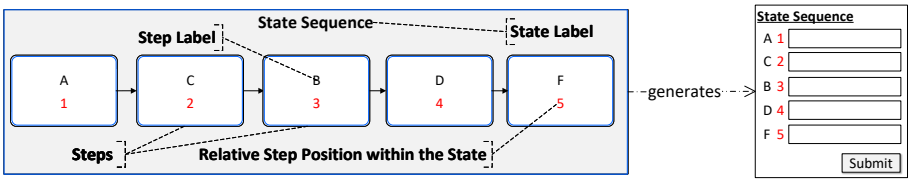

Fig. 4. A Sequential State with the Generated Form

Choices of equal length If the steps of a state are organized using a choice construct of equal length (i.e., the alternative paths all have the same number of steps, cf. Fig. 5), we derive their position by allocating the same position to multiple steps in different paths. For example, in Fig. 5, alternative steps D and B both have the same position.

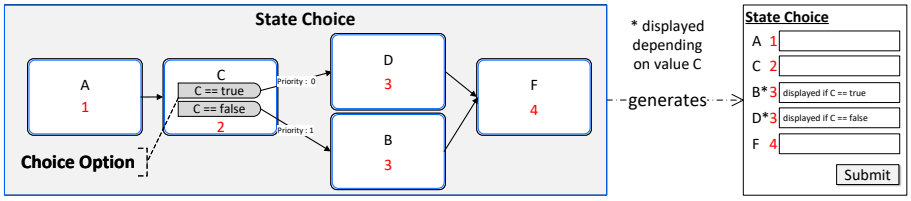

Fig. 5. A State with Choice and the Generated Form

Choices of different lengths If the steps of a state are organized using a choice construct with paths of different lengths (i.e., the alternative paths do not all comprise the same number of steps, cf. Fig. 6), the above approach must not be applied. In the lifecycle process from Fig. 6, for example, the position of the step following both alternative paths (i.e., step E in Fig. 6) depends on the path previously chosen based on the attribute value provided in the context of step B in Fig. 6. In this example, the position of step E will be 3 if the bottom path is chosen, and 6 if the top path is taken. In general, we treat each possible path of steps through the lifecycle process as an individual sequence. This enables us to properly represent positions for choice constructs of different lengths. Note that if no step joins the choice construct, each possible path through the lifecycle state is also represented as an individual sequence.

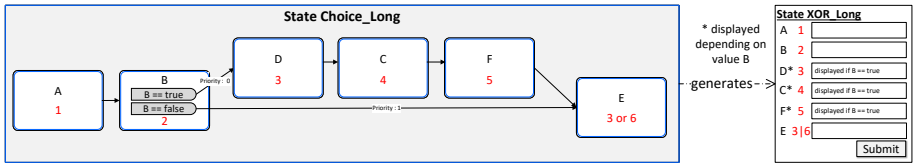

Fig. 6. A State with different Length Choice and the Generated Form

4.2 Lifecycle Process State Positions

To each state of a lifecycle process we assign a relative position as well. We can accomplish this based on the same patterns as presented in Section 4.1. Additionally, we assign to each step the relative position of its state as well. Consequently, to all steps of a specific lifecycle state the same number is assigned in this context. An example derived from the lifecycle process illustrated in Fig. 1 is presented in Tab. 1.

4.3 Leveraging Lifecycle Process Model Positions

Based on the presented patterns, to each step we can assign its relative position within its corresponding state. Moreover, to each state we can assign its relative position within the lifecycle process (cf. Tab. 1 for the representation of object *Exercise*). Note that the lifecycle process of object *Exercise* does not contain a choice construct, and, consequently, only pattern *sequence* is used.

Table 1. Representation of Exercise Lifecycle Process Model (derived from Fig. 1)

ObjectType	State	Step	Step Position	State Position
Exercise	Edit	Lecture	1	1
Exercise	Edit	Name	2	1
Exercise	Edit	Points	3	1
Exercise	Edit	Due Date	4	1
Exercise	Edit	Exercise Files	5	1
Exercise	Publish	Silent	1	2
Exercise	Past Due	Silent	1	3
Exercise	End	Silent	1	4

4.4 Event Log Positions

We now describe how we represent the position of a step regarding the actual execution of the instances of the corresponding lifecycle process. An event log generated by an object-centric and data-driven approach like PHILharmonicFlows comprises information about the execution of an object-aware process. Fig. 7 depicts an extract of the event log corresponding to the execution of state *Edit* of the *Exercise* lifecycle process instance *Exercise Sheet 1*. In general, the event log records which user (column *User ID*) executes which operation (column *Method*) on which object instance (column *Instance*) at what point in time (column *Timestamp*). Additionally, each entry of the event log contains information on which parameter values have been passed (columns *Parameter1* and *2*), the current state of the object instance at the time the event was recorded (column *State*), and the object type (column *Type*). Columns *Position* and *First Position* (c.f., Fig. 7) are explained in the following.

When interacting with a form at runtime, users may write a form field multiple times, e.g., in case a value provided in a previous form field becomes changed. This behavior is then documented in the event log in terms of multiple write access events corresponding to the same form field (i.e., step). To represent the order in which the auto-generated form was actually filled at runtime, we sort the recorded events according to their timestamps and add two columns for each event log entry of an object instance. Column *Position* assigns multiple write accesses of a form field their respective positions each time. Each write access is assigned its position in the event log. Column *First Position* only reflects the order concerning first write access to a form field as only the first event log entry corresponding to a write access is documented. The difference is illustrated for step *Description* in the event log from Fig. 7. Column *Position* assigns to this step the positions 4, 6, and 7, whereas *First Position* assigns position 3 to the first access, neglecting subsequent entries. This differentiation allows customizing our approach by utilizing domain knowledge, e.g., when users change form fields regularly, *First Position* might be more suitable, whereas *Position* is able to account for, e.g., multiple interactions with a form field. Positions of states are derived in a similar way.

User ID	Instance	State	Type	Method	Parameter1	Parameter2	Timestamp	Position	First Position
employee2@uni-ulm.de	Blatt 1	Edit	Exercise	InstantiateObjectTypeAndLink	Lecture	Datenbanken	23.04.2019 09:00	1	1
employee2@uni-ulm.de	Blatt 1	Edit	Exercise	ChangeAttributeValue	Name	Blatt	23.04.2019 09:00	2	2
employee2@uni-ulm.de	Blatt 1	Edit	Exercise	ChangeAttributeValue	Name	Blatt 1	23.04.2019 09:00	3	
employee2@uni-ulm.de	Blatt 1	Edit	Exercise	ChangeAttributeValue	Description		23.04.2019 09:00	4	3
employee2@uni-ulm.de	Blatt 1	Edit	Exercise	ChangeAttributeListValue	Exercise Files	Blatt1.pdf	23.04.2019 09:00	5	4
employee2@uni-ulm.de	Blatt 1	Edit	Exercise	ChangeAttributeValue	Description	a	23.04.2019 09:00	6	
employee2@uni-ulm.de	Blatt 1	Edit	Exercise	ChangeAttributeValue	Description		23.04.2019 09:00	7	
employee2@uni-ulm.de	Blatt 1	Edit	Exercise	ChangeAttributeValue	Begin Date	25.04.2019 00:00:00	23.04.2019 09:01	8	5
employee2@uni-ulm.de	Blatt 1	Edit	Exercise	ChangeAttributeValue	Due Date	09.05.2019 00:00:00	23.04.2019 09:01	9	6
employee2@uni-ulm.de	Blatt 1	Edit	Exercise	ChangeAttributeValue	Maximum Points	25	23.04.2019 09:01	10	7
employee2@uni-ulm.de	Blatt 1	Edit	Exercise	ChangeAttributeListValue	Solution Files	Blatt1-Lsg.pdf	23.04.2019 09:03	11	8
employee1@uni-ulm.de	Blatt 1	Edit	Exercise	ChangeAttributeValue	Due Date	10.05.2019 00:00:00	08.05.2019 13:47	12	

Fig. 7. Event Log Positioning Phoodle Excercise State Edit

After grouping all event log entries by *(object) type*, *(object) state* and *(object) instance,* we assign positions (cf. Section 4) to each write access in the event log (i.e., methods *ChangeAttributeValue*, *ChangeAttributeListValue*, and *InstaticateObjectTypeAndLink* in Fig. 7). Note that we additionally filter the event log for the different paths of a choice construct if necessary.

We then calculate the "average position" for each step across all object instances (cf. Fig. 8). The latter correspond to the average position in which the form field is filled in by users, according to the event log. In addition, this allows ordering the fields (i.e., the lifecycle process steps) of a form (i.e., the lifecycle process states) using a ranking. Thereby, the rank of each step documents the position in which the form field was filled, whereas the rank of each state documents the position in which the form was displayed in relation to the other forms of the lifecycle process. Fig. 8 depicts the positions (columns *Position Step Log* and *Position State Log*) as well as resulting ranks (columns *Rank Step Log* and *Rank State Log*) from a real-world deployment of PHoodle (cf. Section 7 for more details on the event log). The average position according to which, for example, step *Description* was edited is 4.6. After ranking all steps and states based on their average position in the event log, we obtain the order in which

the auto-generated form fields of state *Edit* of object *Exercise* are usually filled as well as the position in which the form was displayed.

Type	State	Step	Position Step Log	Rank Step Log	Position State Log	Rank State Log
Exercise	Edit	Lecture	1,0	1	1	1
Exercise	Edit	Name	2,0	2	1	1
Exercise	Edit	Exercise Files	3,6	3	1	1
Exercise	Edit	Description	4,6	4	1	1
Exercise	Edit	Begin Date	5,6	5	1	1
Exercise	Edit	Due Date	6,2	6	1	1
Exercise	Edit	Maximum Points	6,4	7	1	1
Exercise	Edit	Solution Files	6,6	8	1	1

Fig. 8. Position and Rank Event Log of State Edit

5 Data-driven Evolution of Forms

The following approach utilizes the positions of the states and steps in a lifecycle process as well as the positions of the corresponding entries in the event log to identify possible improvements of the lifecycle process.

In a first step, we join the two representations using an SQL-like full outer join syntax on *(object) type*, *(object) state,* and *step,* respectively. This enables us to compare the actual position of states and steps, as documented in the event log, with the corresponding positions according to the modeled lifecycle process.

We compare the position of a state according to the event log (cf. column *Position State Log* in Fig. 9) with the position of this state in the lifecycle process model (cf. column *Position State Model* in Fig. 9) and calculate the difference between the two. This enables us to check whether or not the order in which the forms are displayed to the users complies with the modeled order. If the two positions deviate from each other, we can determine the new position of the state. Note that for object *Exercise* (cf. Fig. 9) the order in which the forms have been displayed complied with the lifecycle process model, whereas the analysis for object *Submission* revealed that the ordering of states *Rated* and *Waiting* may be changed for the model to better comply with the actual execution recorded by the event log.

Type	State	Position State Log	Position State Model	State Difference	State Position New
Exercise	Edit	1	1	0	1
Exercise	Publish	2	2	0	2
Exercise	Past Due	3	3	0	3
Exercise	End	4	4	0	4
Submission	Edit	1	1	0	1
Submission	Submit	2	2	0	2
Submission	Waiting	4	3	1	4
Submission	Rated	3	4	-1	3
Submission	Inspected	5	5	0	5

Fig. 9. State Position Analysis Objects Exercise and Submission

Furthermore, we can check whether certain steps (i.e., form fields) were never written, were written in another state (i.e., form) than pre-specified, or steps were added to a state, indicated by *NaN* values in the corresponding columns of the outer join. Step *Solution Files* in Fig. 10, for example, has not been specified in the lifecycle model as column *Step Position Model* is *NaN*, but set at (average) position 6.6 (or rank 8 respectively) in the event log. Additionally, the silent steps in states *Publish*, *Past Due,* and *End* have not been documented in the event log, indicated by the *NaN-values* in columns *Position Log* and *Rank Log* in Fig. 10 respectively. Note that silent steps are not recorded in the event log as no attribute is written when executing them.

Subsequently, we calculate the difference between the position recorded in the event log and the one reflected by the lifecycle model by subtracting the step position in the model from the corresponding rank in the event log (cf. column *Difference* in Fig. 10). This enables us to check which steps are placed at the correct position (i.e., column *Difference* equals 0) and which ones need to be relocated (i.e., column *Difference* does not equal 0). Steps with a difference of *NaN* are either not contained in the event log or the lifecycle model and are possible candidates for addition or deletion.

Type	State	Step	Position Log	Rank Log	Step Position Model	Step Difference	Step Position New
Exercise	Edit	Lecture	1	1	1	0	1
Exercise	Edit	Name	2	2	2	0	2
Exercise	Edit	Description	4,6	4	3	1	4
Exercise	Edit	Exercise Files	3,6	3	4	-1	3
Exercise	Edit	Begin Date	5,6	5	5	0	5
Exercise	Edit	Due Date	6,2	6	6	0	6
Exercise	Edit	Maximum Points	6,4	7	7	0	7
Exercise	Publish	Silent	NaN	NaN	1	NaN	NaN
Exercise	Past Due	Silent	NaN	NaN	1	NaN	NaN
Exercise	End	Silent	NaN	NaN	1	NaN	NaN
Exercise	Edit	Solution Files	6,6	8	NaN	NaN	8

Fig. 10. Step Position Analysis Object Exercise

6 Lifecycle Process Improvement Actions

This section introduces process improvement patterns for lifecycle processes and the improvement actions that can be derived from them. Following the patterns, we can automatically derive the modeling operations needed to evolve the corresponding process model accordingly. That means, we are able to dynamically evolve the forms during runtime using the concepts introduced in Section 2.2.

6.1 Correct Positions

Ideally, the states and steps are correctly positioned and the position in the lifecycle process model complies with the rank of the average position in the event log. Consequently, no actions would be required in this case as the generated form (i.e., the lifecycle state) is displayed and executed exactly according to the logic

used for its generation; e.g., this applies to steps *Lecture*, *Name*, *Begin Date*, *Due Date,* and *Maximum Points* in Fig. 10. The steps are correctly positioned if column *Difference* equals to 0. Consequently, no improvement action is required.

6.2 Missing States and Steps

Missing states and steps can be identified based on the *NaN* values contained in the comparison depicted in Fig. 10. Certain states and steps may be missing either in the event log, if a state is never reached or a form field is never filled, or the lifecycle process model, in case a state or step is added by executing corresponding ad-hoc changes at runtime [4].

Missing states in the event log indicate that either the state has never been reached during lifecycle process execution, or it does only contain one silent step, and, therefore, no events related to that state are recorded in the event log. Note that such states are candidates for being deleted. However, as silent states are often used in the context of coordinating interacting objects, checking coordination constraints prior to the deletion becomes necessary.

If the missing state in the event log is not part of any process coordination constraint [19], it may be deleted.

Missing steps in the event log indicate that the corresponding form field has never been filled. This may be the case, for example, if steps are deleted in an ad-hoc manner or alternative paths of a choice construct have never been used. Furthermore, *silent steps* (e.g., the steps in states *Publish*, *Past Due,* and *End* of Fig. 1) correspond to steps in which no attribute needs to be set. To be more precise, there may be no event log entries for silent steps, as no object attributes are required. We can identify missing steps in the event log, if the step is not a silent step (i.e., column *Step* != "Silent") and columns *Position Log* or *Rank Log* contain empty values. If a step is missing in the event log, we may execute the corresponding modeling operations, i.e., the identified step and its transitions are deleted from the lifecycle process. Additionally, we reconnect the remaining steps according to the previously defined order.

Assume, for example, that step *Due Date* in state *Edit* (cf. Fig. 1) is missing in the event log, i.e., the event log does not contain any events related to this step. Step *Due Date* as well its two transitions are then deleted. Moreover, step *Points* is connected with step *Exercise Files* through a directed transition.

Missing states in the lifecycle process model indicate that an object reached a state that has not been foreseen in the lifecycle process model. This may happen if ad-hoc changes are applied to a lifecycle process at runtime, due to which a new state (and at least one corresponding step) was added to the object instance. If such dynamically defined states are recorded in the event log, they can be added to the lifecycle process at the identified position. This

requires the insertion of the state, the corresponding steps, and the transitions to correctly integrate the new state into the lifecycle process model.

Suppose a new state *Pending* with attribute *Date* is added to the lifecycle of object *Exercise* between states *Edit* and *Publish* (cf. Fig. 1). This would then require the insertion of state *Pending* and attribute *Date,* the insertion of a new transition between state *Date* and the silent step in state *Publish,* and the re-linking of the existing transition from step *Exercise Files* to step *Date.*

Missing steps in the lifecycle process model indicate that steps (i.e., form fields) have been written during the execution of lifecycle process instances that were previously not specified in the lifecycle process (state). Such steps are represented in the lifecycle process part of the outer join (e.g., columns *Step Position* and *State Position* in Fig. 10). We can discover missing steps in the lifecycle process model through *NaN* values in columns *State Position* and *Step Position* (e.g., an additional form field might be required, or an attribute be written in a state other than the one foreseen in the lifecycle process model). In Fig. 10, step *Solution Files* was executed according to the event log, but is not contained in the lifecycle process and, therefore, should be added at the position suggested by the event log (cf. column *Step Position New* in Fig. 10).

As example assume, that the additional step *Solution* is required after executing step *Exercise Files* in state *Edit* (cf. Fig. 1). The needed operations are to add step *Solution*, link it to step *Exercise Files* (through an additional lifecycle transition), and re-link the existing transition from step *Exercise Files* to step *Solution.*

6.3 Auto-Adjusting the Form Logic

While the previously discussed improvement actions have dealt with the addition or deletion of steps from a lifecycle process, another important aspect is to identify of the correct logic of the steps within a state (i.e., the execution order of the steps). Recall that this logic is utilized by PHILharmoniFlows to auto-generate a form with corresponding user guidance. A step executed in the context of a state might not be ideally positioned for a user filling out the form, but users may flexibly choose the order in which they actually fill in the form. The event log that records the order of the latter, therefore, contains the information "how" users interact with the form. Consequently, the actual order of the steps discovered from the event log allows re-organizing the ordering of the steps within a state. By subtracting the step position of the lifecycle process model from the rank in the event log (cf. column *Difference* in Fig. 10), we obtain the difference between the position in the event log and the position in the lifecycle process model. If this difference does not equal 0, the steps within the state are not ideally ordered, i.e., users prefer filling the form in another order. Furthermore, we can identify the new position for each step of a lifecycle process by adding columns *Difference* and *Step Position Model.*

In the comparison presented in Fig. 10, this is the case for steps *Description* and *Exercise Files.* According to the event log, the rank of the average

position over all lifecycle process instances of step *Exercise Files* is 3, and 4 for step *Description* (i.e., step *Exercise Files* is executed before step *Description*), essentially switching their positions.

The modeling operations needed to implement this change are to delete the associated transitions between the states and to add new ones according to the new ordering. In the scenario described in Fig. 10, this includes the deletion of transitions between steps *Name*, *Description*, *Exercise Files*, and *Begin Date* and their re-linking by adding new transitions between steps *Name* and *Exercise Files*, *Exercise Files* and *Description*, and between *Description* and *Begin Date.*

7 Evaluation

To evaluate the presented approach, we applied it to an event log[2] we obtained in the context of a real-world deployment of our data- and process-aware e-learning system *PHoodle*, which we had implemented with PHILharmonicFlows. During its use, *PHoodle* replaced the established Moodle e-learning platform for a course with more than hundred students from Management Science over one semester. During this experiment we gathered the system logs from the PHILharmonicFlows process engine, including data of users (anonymized due to General Data Protection Regulation), object instances, object states, object types, and provided attribute values, together with the corresponding timestamps (cf. Fig. 7 for an example event log). In total, the e-learning system event log consists of 39890 entries including information on 848 object instances of 9 different object types (cf. Table 2). Note that column *Number of log entries* corresponds to the number of interactions such as the setting of an attribute, including users displaying an object at a given state (e.g., a student checks the due date of an exercise). Column *Number of interactions* represents those log entries that refer to the setting of an attribute value (e.g., a supervisor provides files in step *Exercise Files* of state *Edit* - cf. Fig. 1) or to transitions between states of a lifecycle process (e.g., state *Edit* is completed and an exercise transitions to state *Publish* cf. Fig. 1).

When applying our approach to this PHoodle scenario (with First Position for lifecycle steps, cf. Section 4.4), we identified several potential improvements: For object *Attendance* (cf. Fig. 11), we could derive the following improvements:

States — Regarding the lifecycle process states of object *Attendance*, the comparison of event log and lifecycle process suggests moving state *Unassign Tutorial* to position 1, and state *Start* to position 2, essentially switching positions of the two states. States *Assign Tutorial* and *End* are positioned correctly.

Steps — Regarding steps, the approach suggests adding steps *Lecture* (position 1) and *Person* (position 2) to state *Unassign Tutorial,* while moving the existing step *Tutorial* to position 3. Furthermore, step *Person* in state *Start* should be removed as it has never been set in the event log.

[2] Event log provided: https://www.researchgate.net/project/CoopIS-Phoodle-Data

Table 2. PHoodle Log Statistics

Object	Number of objects	Number of log entries	Number of interactions
Person	133	290	274
Attendance	137	3233	584
Download	14	4574	152
Employee	2	14	8
Exercise	5	7323	110
Lecture	1	11741	14
Submission	498	10689	3920
Tutor	6	116	18
Tutorial	52	1910	443
Total:	848	39890	5523

For object *Exercise* (cf. Fig. 11), we propose the following improvement actions:

States The ordering of the states of object *Exercise* complies with the one recorded in the event log. Therefore, no improvement is needed.

Steps The steps corresponding to state *Edit* of object *Exercise* may be improved by switching the positions of steps *Exercise Files* and *Description* and adding the step *Solution Files* at position 8.

	State Analysis						Step Analysis					
Object	State	Log State	Model State	Difference State	Position State New	State Improvement	Step	Log Step	Model Step	Difference Step	Position Step New	Step Improvement
Attendance						Move State	Lecture	1	*NaN*	*NaN*	1	Add Step Lecture: State Unassign Tutorial, Position 1
Attendance	Unassign Tutorial	1	2	-1	1	Unassign Tutorial to position 1	Person	2	*NaN*	*NaN*	2	Add Step Person: State Unassign Tutorial, Position 2
Attendance							Tutorial	3	1	2	3	Move Step Tutorial to position 3
Attendance	Start	2	1	1	2	Move State Start to position 2	Lecture	1	1	0	1	Correct
Attendance							Person	*NaN*	2	*NaN*	*NaN*	Remove Step Person
Attendance	Assign Tutorial	3	3	0	3	Correct	Achieved Points	1	1	0	1	Correct
Attendance	End	*NaN*	4	*NaN*	4	Correct (Silent)	Silent	*NaN*	1	*NaN*	*NaN*	Correct (Silent)
Exercise							Lecture	1	1	0	1	Correct
Exercise							Name	2	2	0	2	Correct
Exercise							Exercise Files	3	4	-1	3	Move Step Exercise Files to Position 3
Exercise							Description	4	3	1	4	Move Step Description to Position 4
Exercise	Edit	1	1	0	1	Correct	Begin Date	5	5	0	5	Correct
Exercise							Due Date	6	6	0	6	Correct
Exercise							Maximum Points	7	7	0	7	Correct
Exercise							Solution Files	8	*NaN*	*NaN*	8	Add Step Solution Files: State Edit, Position 8
Exercise	End	*NaN*	4	*NaN*	4	Correct (Silent)	Silent	*NaN*	1	*NaN*	*NaN*	Correct (Silent)
Exercise	Past Due	*NaN*	3	*NaN*	3	Correct (Silent)	Silent	*NaN*	1	*NaN*	*NaN*	Correct (Silent)
Exercise	Publish	*NaN*	2	*NaN*	2	Correct (Silent)	Silent	*NaN*	1	*NaN*	*NaN*	Correct (Silent)

Fig. 11. Excerpt of the PHoodle Comparison - Lifecycle Process vs. Event Log

We also applied the identified improvement actions to the corresponding lifecycle processes. Depending on the respective action, this either resulted in an alternative ordering of the displayed forms or forms that better comply with the actual execution through the addition, reordering or deletion of lifecycle steps. Fig. 12 depicts the improvement of State *Edit* for the lifecycle process of object *Exercise* including the generated and improved forms.

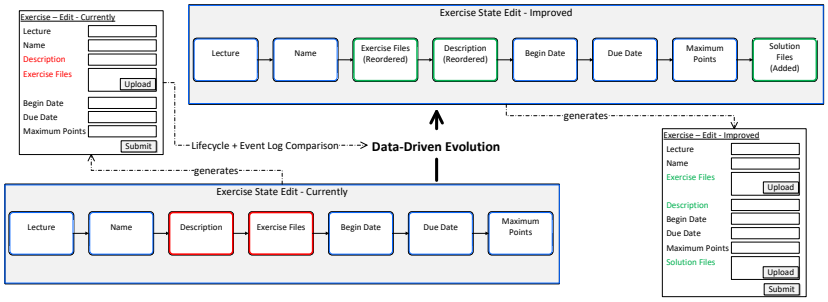

Fig. 12. Improvement of State Edit for Object Exercise

8 Related Work

The work presented in this paper is part of two research areas: user forms in information systems and business process improvement.

User forms have already been subject to research for a long time, e.g., alignment of user form labels [12] and guidelines for usable web forms [7]. The guidelines tackle user form elements such as, for example, content, layout, input types, error handling, and the submission of user forms. However, the ordering of fields in a form is only mentioned as "keep questions in an intuitive sequence". The presented approach enables us to automatically derive such an intuitive sequence from the event log and to auto-adapt the generated forms accordingly at runtime using techniques known from process model evolution [18].

Process improvement is concerned with model repair and extension. Model repair changes a process model for it to better fit real executions, whereas extension is concerned with adding additional perspectives to a process model. Regarding model repair, [13] proposes a technique that preserves the original model structure by introducing subprocesses to the model in order to permit replaying a given event log on the repaired model. Conformance checking results are used to identify in which part of the process a subprocess needs to be added, whereas discovery algorithms mine the to-be-added subprocesses. As our approach does not follow the activity-centric paradigm (like [13] does), similarity is not a concern. Our approach changes the logic of user forms generated at runtime rather than the actual "control-flow" of the business processes. In other words, our approach improves the order and logic of forms presented to users rather than the activities to be executed. Furthermore, due to the flexible nature of forms in the context of data-driven processes, deviations (e.g., filling a form in a different order) from the modeled logic are implicitly tolerated as well.

The repair approach presented in [5] transforms BPMN process models and event logs into a Prime Event Structure (PES) to identify patterns regarding task, sequence flow, and gateway modifications. Identified discrepancies are then displayed to users on top of the model to decide on individual fixes. In contrast, our approach focuses more on the usability aspect during process execution rather than the ordering of activities.

The work presented in [11] focuses on repairing inconsistencies in declarative process models, which are more flexible compared to imperative models. The approach identifies and then deletes the smallest possible set of constraints to regain consistent models at design time. An approach for repairing declarative process models at runtime is presented in [16]. In our approach, we focus on the logic of steps (and therefore the logic of forms displayed at runtime) encapsulated in object lifecycle process states, used to guide users through the corresponding form. However, as long as forms are fully filled, no inconsistencies occur.

According to [1], model extension is "a type of process enhancement where a new perspective is added to the process model by cross-correlating it with the log.". Typically, model extension focuses on the organizational or temporal perspective. The temporal perspective [6] focuses on identifying the process fragments with extended times as interesting for process improvement actions. In contrast, the organizational perspective [9] focuses on adding associations between roles and the execution of processes to a model.

9 Conclusions and Outlook

This paper presented an approach for automatically improving lifecycle processes based on the behavior that can be observed in an event log. We introduced various control flow patterns of lifecycle processes as well as their auto-generated form at runtime. We then characterize the steps and states of object lifecycle processes by allocating their positions. Additionally, we assign corresponding positions to the relevant log entries. The latter are then analyzed and aggregated for the event log, allowing for a representation of the average position of a step as recorded in the event log. In other words, we analyze in which order users filled in forms at runtime.

We further compare this position for each step with the modeled position. This, in turn, enables us to identify obsolete and missing steps in the lifecycle process model. Additionally, we are able to check whether the logic within lifecycle states (i.e., the user guidance when filling in forms) is ideal, or whether the user guidance can be improved by adapting the logic used to generate the form.

Additionally, we are able to derive the required modeling operations that enable PHILharmonicFlows to perform the corresponding process model evolution that implements identified improvements.

In future work we will extend the presented approach in a two-fold manner: First, we plan to combine it with conformance categories [8] and heuristics to further account for the frequency of changes to lifecycle process models. Second, we plan to use the infrequent (user-specific) behavior to individually adapt lifecycle processes based on previous executions from individual users.

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. 01IS20027B)

References

1. van der Aalst, W.M.P.: Process Mining: Data Science in Action. Springer (2016)
2. 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)
3. Andrews, K., Steinau, S., Reichert, M.: A tool for supporting ad-hoc changes to object-aware processes. In: Demo Track of the 22nd International EDOC Conference (EDOC 2018). pp. 220–223. IEEE Computer Society Press (October 2018)
4. Andrews, K., Steinau, S., Reichert, M.: Enabling runtime flexibility in data-centric and data-driven process execution engines. Information Systems **101** (2021)
5. Armas Cervantes, A., van Beest, N.R.T.P., La Rosa, M., Dumas, M., García-Bañuelos, L.: Interactive and incremental business process model repair. In: OTM 2017 Conferences. pp. 53–74. Springer (2017)
6. Ballambettu, N.P., Suresh, M.A., Bose, R.P.J.C.: Analyzing process variants to understand differences in key performance indices. In: Advanced Information Systems Engineering. pp. 298–313. Springer (2017)
7. Bargas-Avila, J., Brenzikofer, O., Roth, S., Tuch, A., Orsini, S., Opwis, K.: Simple but crucial user interfaces in the world wide web: Introducing 20 guidelines for usable web form design. In: User Interfaces. IntechOpen, Rijeka (2010)
8. Breitmayer, M., Arnold, L., Reichert, M.: Enabling conformance checking for object lifecycle processes. In: 16th International Conference on Research Challenges in Information Science (RCIS 2022). LNBIP, Springer (2022)
9. Burattin, A., Sperduti, A., Veluscek, M.: Business models enhancement through discovery of roles. In: 2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM). pp. 103–110 (2013)
10. Cohn, D., Hull, R.: Business artifacts: A data-centric approach to modeling business operations and processes. IEEE Data Eng. Bull. **32**(3), 3–9 (2009)
11. Corea, C., Nagel, S., Mendling, J., Delfmann, P.: Interactive and minimal repair of declarative process models. In: BPM Forum. pp. 3–19. Springer (2021)
12. Das, S., McEwan, T., Douglas, D.: Using eye-tracking to evaluate label alignment in online forms. In: Proceedings of the 5th Nordic Conference on HCI. pp. 451–454. NordiCHI '08, Assoc. for Computing Machinery (2008)
13. Fahland, D., van der Aalst, W.M.P.: Model repair - aligning process models to reality. Inf. Syst. **47**, 220–243 (2015)
14. Hassenzahl, M.: User experience (ux): Towards an experiential perspective on product quality. In: Proceedings of the 20th Conference on l'Interaction Homme Machine. pp. 11–15. IHM '08, Association for Computing Machinery (2008)
15. Künzle, V., Reichert, M.: PHILharmonicFlows: towards a framework for object-aware process management. J of Soft Maint & Evo **23**(4), 205–244 (2011)
16. Maggi, F.M., Westergaard, M., Montali, M., van der Aalst, W.M.P.: Runtime verification of ltl-based declarative process models. In: Runtime Verification. pp. 131–146. Springer (2012)
17. Reichert, M.: Process and data: Two sides of the same coin? In: 20th Int'l Conf on Cooperative Information Systems (CoopIS'12). pp. 2–19. Springer (2012)
18. Reichert, M., Weber, B.: Enabling flexibility in process-aware information systems: challenges, methods, technologies. Springer Science & Business Media (2012)
19. Steinau, S., Andrews, K., Reichert, M.: The relational process structure. In: CAiSE 2018. pp. 53–67. No. 10816 in LNCS, Springer (2018)
20. 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. Softw & Sys Modeling **18**(4), 2679–2716 (2019)