Enabling Ad-Hoc Changes to Object-Aware Processes

Kevin Andrews, Sebastian Steinau, and Manfred Reichert
Institute of Databases and Information Systems
Ulm University, Germany
Email: {firstname.lastname}@uni-ulm.de

***Abstract*—Contemporary process management systems support users during the execution of repetitive, predefined business processes. However, when unforeseen situations occur, which are not part of the process model serving as the template for process execution, contemporary process management technology is often unable to offer adequate user support. One solution to this problem is to allow for ad-hoc changes to process models, i.e., changes that may be applied on the fly to a running process instance. As opposed to the widespread activity-centric process modeling paradigm, for which the support of instance-specific ad-hoc changes is well researched, albeit not supported by most commercial solutions, there is no corresponding support for ad-hoc changes in other process support paradigms, such as artifact-centric or object-aware process management. This paper presents concepts for supporting such ad-hoc changes in object-aware process management, and gives insights into the challenges we tackled when implementing this kind of process flexibility in the PHILharmonicFlows process execution engine. The development of such advanced features is highly relevant for data-centric BPM, as the research field is generally perceived as having low maturity when compared to activity-centric BPM.**

***Index Terms*—process flexibility, ad-hoc change, object-aware process**

I. Introduction

As one of the main advantages of using a process management system in enterprises, changes to real-world business processes can often be incorporated into the flow of data between users and IT systems by simply changing the process models in the process management system [1]. This allows processes to be updated and improved over time, supporting more cases that were not thought of during their initial modeling. However, process models are often not detailed enough to adequately support each and every possible variant of process execution. Furthermore, there are process variants that occur so rarely, that incorporating them into the process model would increase its complexity at a far too low benefit. In these cases, *ad-hoc changes* to running process instances become necessary, a topic that has been addressed many times for traditional, activity-centric process management systems. This paper offers a fundamental approach for introducing the concept of ad-hoc process model changes to object-aware process management, i.e, a data-driven and data-centric process support paradigm. In particular, we detail how the paradigm helps to ensure run-time correctness of changed process instances as well as the replay-based method we use to reconstruct the process state after ad-hoc changes.

The remainder of the paper is structured as follows. First, Section II offers a discussion of fundamentals. Section III presents the requirements for supporting ad-hoc changes in an object-aware process execution engine. The main contribution is provided by Section IV, which presents an original concept for ad-hoc changes to object-aware processes. A discussion of threats to validity, based on a sophisticated prototypical implementation, can be found in Section V. Section VI discusses related work. Finally, Section VII gives a short summary of the contribution.

II. Fundamentals

As the conceptual foundations of object-aware process management are crucial for understanding the contribution, this section offers an overview thereof.

A. Object-aware Process Management

PHILharmonicFlows, the object-aware process management framework we are using as a test-bed for the concepts presented in this paper, has been under development for many years at Ulm University [2]–[4]. This section gives an overview on the concepts necessary to understand the remainder of the paper. PHILharmonicFlows takes the idea of a data-driven and data-centric process management system and improves it by introducing the concept of *objects*. One such object exists for each business object present in a real-world business process. As can be seen in Fig. 1, an object consists of data, in the form of *attributes*, and a state-based process model describing the *object lifecycle*.

The attributes of the *Transfer* object (cf. Fig. 1) include *Amount*, *Date*, and *Approved*. The *lifecycle process*, in turn,

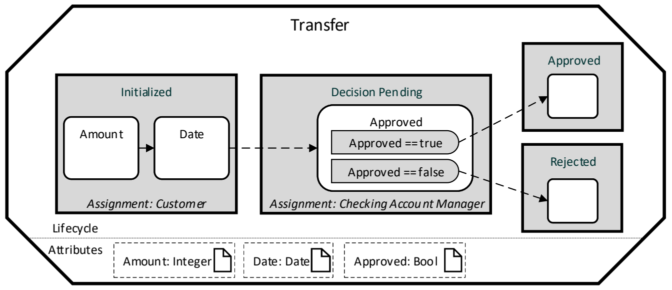


Fig. 1. Example Object Including Lifecycle Process

describes the different *states (Initialized*, *Decision Pending*, *Approved*, and *Rejected)*, an instance of a *Transfer* object may enter during process execution. Each state contains one or more *steps*, each referencing exactly one of the object attributes, and enforcing the respective attribute is written at run-time. The steps are connected by *transitions*, which arrange them in a sequence. The state of the object changes when all steps in a state are completed. Finally, alternative paths are supported in the form of *decision steps*, an example of which is the *Approved* decision step.

As PHILharmonicFlows is *data-driven*, the lifecycle process for the *Transfer* object can be understood as follows: The initial state of a *Transfer* object is *Initialized*. Once a *Customer* has entered data for the *Amount* and *Date* attributes, the state changes to *Decision Pending*, which allows an *Account Manager* to input data for *Approved*. Based on the value for *Approved*, the state of the *Transfer* object changes to *Approved* or *Rejected*. Obviously, this fine-grained approach to modeling a business process increases complexity when compared to the activity-centric paradigm, where the minimum granularity of a user action is one atomic activity or task, instead of an individual data attribute.

However, the object-aware approach allows for *automated form generation* at run-time. This is facilitated by the lifecycle process of an object, which dictates the attributes to be filled out before the object may switch to the next state, resulting in a personalized and dynamically created form. An example of such a form, derived from the lifecycle process in Fig. 1, is shown in Fig. 2.

Note that a single object and its resulting form only constitute one part of a complete PHILharmonicFlows process. To allow for complex executable business processes, many different objects and users may have to be involved [4]. It is noteworthy that *users* are simply special objects in the object-aware process management concept. The entire set of objects and relations present in a PHILharmonicFlows process is denoted as the *data model*, an example of which can be seen in Fig. 3. In addition to the objects, the data model contains information about the *relations* existing between them. A relation constitutes a logical association between two objects, e.g., a relation between a *Transfer* and a *Checking Account*.

At run-time, each of the objects can be instantiated many times as so-called *object instances*. The lifecycle processes present in the various object instances may be executed concurrently at run-time, thereby improving performance. Furthermore, the relations can be instantiated at run-time, e.g., between an instance of a *Transfer* and a *Checking Account*, thereby associating the two object instances with each other. The resulting meta information, i.e., that the *Transfer* in question belongs to the *Checking Account*, can be used to coordinate the processing of the two object instances with each other. Fig. 4 shows an example of a *data model instance* executed at run-time.

Finally, complex object coordination, which becomes necessary as most processes consist of numerous interacting business objects, is possible as well [4]. As objects publicly advertise their state information, the current state of an object can be utilized as an abstraction to coordinate with other

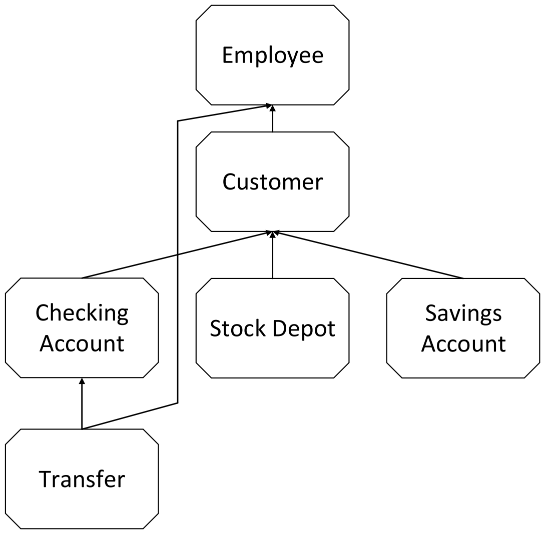
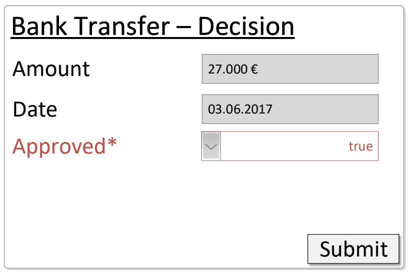


Fig. 3. Design-Time Data Model



Amount	27.000 €
Date	03.06.2017
Approved*	true

Fig. 2. Example Form

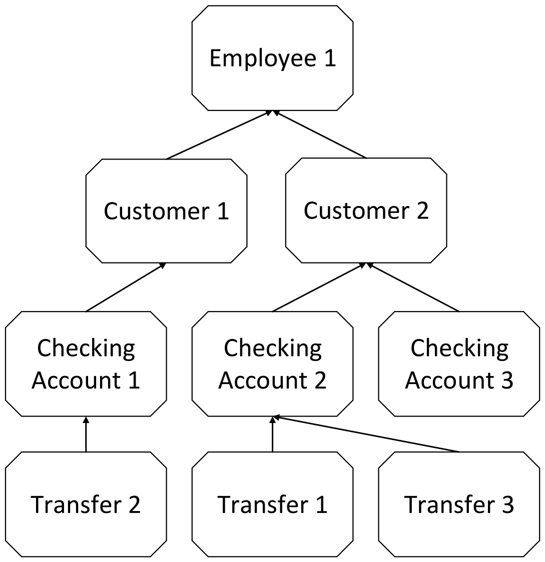


Fig. 4. Data Model Instance

objects corresponding to the same business process through a set of constraints, defined in a separate *coordination process*. As an example, consider a constraint stating that a *Transfer* may only change its state to *Approved* if there are less than 4 other *Transfers* already in the *Approved* state for one specific *Checking Account*.

The various components of PHILharmonicFlows, i.e., objects, relations, and coordination processes, are implemented as microservices, turning PHILharmonicFlows into a fully distributed process management system for object-aware processes. For each object instance, relation instance, or coordination process instance one microservice is present at run-time. Each microservice only holds data representing the attributes of its object. Furthermore, the microservice only executes the lifecycle process of the object it is assigned to. The only information visible outside the individual microservices is the current "state" of the object, which, in turn, is used by the microservice representing the coordination process to properly coordinate the objects' interactions with each other.

B. Process Model Evolution and Ad-hoc Changes

As motivated in Section I, business processes models are subject to different types of changes. These can be categorized into: *deferred process model evolutions*, *immediate process model evolutions*, and *ad-hoc changes* [1].

Deferred process model evolutions are relatively simple and supported by most process management systems as they merely require the process model to be changed and redeployed. As existing process model deployments and their process instances remain untouched, this is a rather trivial task. More information on typical change patterns to process models can be found in [5].

Immediate process model evolutions on the other hand are more challenging as they not only allow for the process model to be updated, but also try to migrate already running process model instances to the newer version. Such an immediate migration poses significant challenges to a process management system, such migrating process instances that have already executed past the point in the process model to which changes were made [6]. Immediate process evolutions are required for use cases where the running process instances must not continue execution based on the old process model. As an example, consider a faulty web-service call in the process model that has to be fixed for all running instances as soon as possible.

Finally, *ad-hoc changes* constitute a special case of immediate process model evolution in which only one specific running process model instance has to be changed. This allows users to deviate from the predefined process in various ways, e.g. to execute two activities in a different order as originally intended. Enabling ad-hoc changes reduces the complexity of the process model as not every single possible variant of process execution has to be predefined.

In activity-centric process management, there is always one central entity to which all these changes are applied, the *process model*. While evolutionary changes might be applied directly to the process model all process instances are derived from, ad-hoc changes are always applied to the process instance itself. Each process instance has, at least conceptually, its own copy of the process model, which can be changed individually. These two *change granularity levels* possible in activity-centric processes are depicted in Fig. 5.

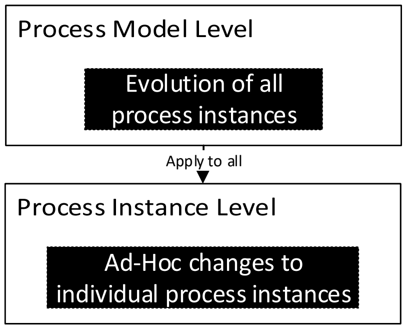

Process Model Level
Evolution of all process instances
Apply to all
Process Instance Level
Ad-Hoc changes to individual process instances

Fig. 5. Activity-centric Change Granularity Levels

Regarding object-aware process management, these two change granularity levels exist as well. Specifically, evolutionary changes may be made to the data model and its objects, while ad-hoc changes may be applied to data model instances and object instances, analogously to the activity-centric case. However, considering that more object instances may be created at any point during process execution, with only two levels of granularity it is not quite clear what an ad-hoc change to an object actually constitutes. To ensure that users can express whether they wish to only change one individual object instance or all existing and future instances of an object in the data model instance, a third level of granularity has to be defined: the object instance level. The resulting three change granularity levels possible in object-aware processes are depicted in Fig. 6.

It is noteworthy that ad-hoc changes to objects on the data model instance level are propagated to all existing and future

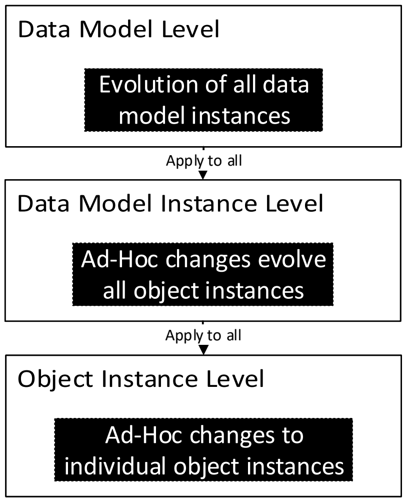

Data Model Level
Evolution of all data model instances
Apply to all
Data Model Instance Level
Ad-Hoc changes evolve all object instances
Apply to all
Object Instance Level
Ad-Hoc changes to individual object instances

Fig. 6. Object-aware Change Granularity Levels

object instances. For example, if an attribute is added to an object on the data model instance level, all object instances in the data model instance will have the new attribute. However, if the same change is introduced at the object instance level, only the specific object instance the change is applied to will have the additional attribute.

Finally, the data model instance level has support for a complete set of change operations. In addition to the changes that are possible for object instances, i.e., adding attributes, permissions, and editing the lifecycle process, one may also introduce changes to the data model instance itself, such as adding new objects or relations. More precisely, ad-hoc changes to the data model instance level allow changing everything that is possible in the regular modeling environment, i.e., completeness is ensured. The object instance level, on the other hand, is limited to changes to the conceptual elements local to any one object instance, e.g., adding a step. Both ad-hoc changes to the data model instance level and the object instance level are the focus of this paper.

III. Requirements

This section presents major requirements we identified for supporting ad-hoc changes in object-aware process management. On the one hand, they were derived from the requirements for activity-centric processes and adapted as necessary. On the other, we considered data model change operations in a number of analyzed object-aware processes [7].

Requirement 1: *(Change Atomicity)* Existing object instances should not reflect ad-hoc changes immediately, as individual changes to an object instance may render it semantically or syntactically incorrect until other changes are applied. An example if this could be the insertion of a step into a lifecycle process that has no incoming or outgoing transitions. Even if the missing transitions were added shortly afterwards, there would be a time span in which the individual change of adding a step would constitute a syntactical error in an object-aware lifecycle process. Therefore, if this change were introduced to a running process instance would cause run-time failures. Therefore, a capability must be developed that allows for multiple changes to introduced to running process instances in an atomic fashion. In the simple example of adding a step, the entire atomic change would therefore consist of adding the step and all transitions, ensuring that the running process instances are never in an incorrect state.

Requirement 2: *(Correctness)* The changes that can be applied to object instances should result in a correct process model, i.e., the verification criteria that are applied prior to process deployment must apply to ad-hoc changes as well. Reiterating the previous example of adding a new step to a lifecycle process, the entire atomic change (i.e. set of individual changes) that should be introduced to the running object instance must always result in a correct underlying process model.

Requirement 3: *(Run-time Consistency)* An object instance must never enter a lifecycle process state it could not be in if it were re-executed in an identical fashion after an ad-hoc change. For example, if a required step is added in a state that the lifecycle process of the object instance in question has already progressed past, it would be inconsistent for the object instance to remain in the later state without having completed the newly required step in the earlier state. This is due to the fact that -newly- created object instances could never progress past the new step without providing data for the associated attribute, but the existing object instance would have already progressed past this point.

Requirement 4: *(Model Consistency)* When combining ad-hoc changes to the entire data model instance with prior ad-hoc changes to individual object instances, conflicting changes must be resolved. Consider an object instance that has an additional transition between two steps, added as an ad-hoc change on the object instance level at run-time. If a process modeler were to introduce an additional ad-hoc change at the data model instance level that, e.g., the deletion of one of the steps that the additional transition is connected to, this changes to a specific object instance would be in conflict with the change that affects all existing object instances.

Requirement 5: *(Concurrency)* Change operations must be executable while the process instance is running, without hindering the execution of other object instances not concerned by the changes. This is in contrast to activity-centric process management, where a single process instance often corresponds to a single business case. To be precise, we aim at offering a solution that allows for ad-hoc changes to individual object instances without affecting the performance of other object instances. Explicitly excluded from this work, however, is a broader discussion on concurrent ad-hoc changes the same object instance, as this can be solved with trivial locking mechanisms, i.e., simply disallowing multiple users to conduct changes to the same object instance at the same time.

Requirement 6: *(Coordination)* As object instances can be coordinated with each other based on their current state [4], state changes due to ad-hoc changes must be handled correctly. Such state changed may arise when required steps are inserted at earlier points in an object instance lifecycle process, as the example in Requirement 3 portrays. Furthermore, through the removal of individual steps from a lifecycle process, the lifecycle process may also advance to a later state as the result of an ad-hoc change as well. Both cases must be handled correctly by the coordination process to ensure that other object instances react to the changes correctly.

Requirement 7: *(Completeness)* The set of possible operations for ad-hoc changes must be complete in the sense that all aspects of the process model editable at design-time must also be editable at run-time. Note that this work does not contain a discussion on which ad-hoc change operations make “sense” from a user perspective as we are of the opinion that the concept we develop for ad-hoc changes should support any operation required to create or alter a process model. Keeping the user side of the concept from being too overwhelming or powerful is a user interface concern as long as the concept and implementation of the ad-hoc change operations is complete enough to support any change operation deemed important.

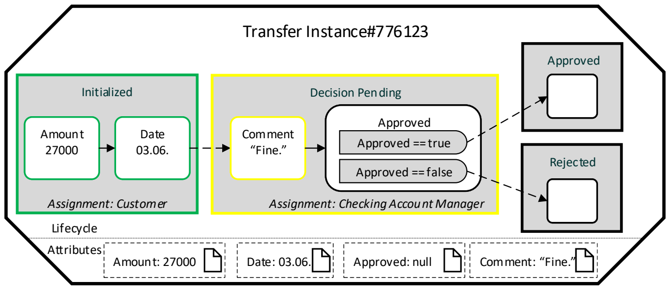


Fig. 7. Transfer Object Instance with *Comment* Attribute and Step

IV. Ad-Hoc Changes in Object-Aware Processes

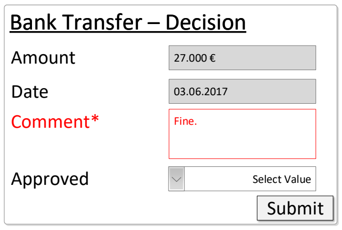
This section presents the fundamental concepts we developed for enabling ad-hoc changes to instances of object-aware processes, all of which are fully implemented in the PHILharmonicFlows execution engine.

A. Object Instance Level Changes

An ad-hoc change to an object instance can be required by users for many reasons. As objects consist of various attributes, permissions, and a lifecycle process, a simple ad-hoc change could be adding a new attribute and a corresponding lifecycle step to the object. An example is given in Fig. 7 by the additional attribute *Comment*, and the corresponding step in the *Decision Pending* state. The *Transfer* object this instance was created from can be viewed in Fig. 1.

Note that this change affects the single object instance depicted in Fig. 7, not any other existing or future instances of *Transfer*. This is due to the fact that changing the template for creating new *Transfer* object instances, the *Transfer* object (cf. Fig. 1), remains untouched as the ad-hoc change is introduced on the object instance level and not on the data model instance level. Such changes to all existing and future instances of an object are discussed in Section IV-B.

From a user perspective, the introduction of this change would alter the form generated from this object at run-time. While an unmodified *Transfer* object instance would display the Form depicted in Fig. 2 to a *Checking Account Manager* when the object enters the *Decision Pending* state, after the ad-hoc, the instance displays a slightly different form to the *Checking Account Manager*. This altered form, which can be viewed in Fig. 8, displays an input field for the *Comment* attribute and sets it as mandatory, as required by the corresponding step inserted into the *Decision Pending* state of the *Transfer* lifecycle process.


Bank Transfer – Decision

Amount	27.000 €
Date	03.06.2017
Comment*	Fine.
Approved	Select Value

Submit

Fig. 8. Dynamically Generated Form with Change

Supporting ad-hoc changes on the object instance level is accompanied by a number of challenges, which must be solved to lay the foundation for data model instance level changes. Our concept solves these challenges in line with the requirements from Section III.

To help understand our concept, we introduce the notion of a *change log entry*. A change log entry is a historical change operation that was applied to some element being part of a data model *M*.

Definition 1. (Change Log Entry)
A tuple L=(S,A,P,T) is called change log entry if the following holds:

- S, the source of the log entry, corresponds to any object-aware construct (e.g. object, relation, state, etc.) with $S \in M$
- A is a modeling action that can be applied to S
- P is a set of parameters with which A was applied to S
- T is the logical timestamp of the modeling action

One such log entry is created for each modeling action a user completes when creating or changing data model *M*, thereby constituting the *change log* of *M*. Example 1 shows a concrete change log entry for the creation of a new string attribute, *Comment*, in the *Transfer* object.

Example 1. *(Change Log Entry)*

$$l_{14} = \begin{cases} S & object : Transfer \\ A & AddAttribute \\ P & [name : "Comment", type : String] \\ T & 14 \end{cases}$$

The logical timestamp T of l_{14} holds the value 14, signifying that it is the 14th change to the data model. Specifically, tracking the logical timestamp of modeling actions across the entire data model becomes necessary to allow sorting them in the original order, across the various objects they are attached to, which becomes necessary when reconstructing data models from their change logs. Reconstruction can be used for fairly trivial tasks, such as creating an identical copy of a data model by replaying its change log, i.e, repeating each modeling action step by step, but also more complex use cases, such as defining a delta of change log entries that constitutes a data model variant [8].

The availability of such change logs allows viewing an object-aware data model and the therein contained objects from a new perspective: as the result of the application of all modeling actions logged in the change log. Furthermore, this perspective can be applied to data model instances and object instances, as the individual instances are all based on their models which then can be recreated by repeating the modeling actions contained in the change log entries.

However, in object-aware process management an object instance is not merely defined by the attributes and lifecycle process model of the object it was instantiated from, but also by the data values present for each attribute at a given point in time during the processing of the object instance, i.e., the execution of its lifecycle process. This is due to the fact that object-aware processes are inherently data-driven (cf. Section II), meaning that the execution progress (i.e., the state) of each object is defined by its attribute values, and, furthermore, that the current states of the individual objects are used by the coordination process to determine the execution progress of the entire data model instance.

Taking these facts into account, we can offer an alternate definition of an object instance, which deviates from the one found in literature on object-aware process management [3]. The previous definition focused on the actual object-aware constructs that comprise the object, such as attributes, attribute values, permissions and all elements of the object lifecycle process.

Definition 2. (Log-defined Object Instance)
A tuple O=(log, data) is called log-defined object instance if the following holds:

- log is a set of change log entries L (cf. Definition 1)
- data is a mapping of values to object attributes

As $O.log$ contains a set of change log entries with logical timestamps, recreating the sequence of actions (with accompanying parameters) necessary to create O in its current state is trivial. Furthermore, once the object has been created from the logs, it becomes possible to assign to each attribute a the value $O.data[a]$. In essence this entire procedure allows us to serialize an object instance in a running data model instance to its equivalent log-defined object instance, and then to recreate an identical copy of the original instance. However, this makes little sense, as the point of ad-hoc changes to object instances is not to create identical object instance copies, but change existing object instances. Still, several reasons exist why serializing and deserializing objects to and from logs constitutes a fundamental building block for our concept.

When viewing an object instance as a log-defined object instance, i.e., under the premise that the object instance is merely the result of the sequence of modeling actions necessary to create the object it was instantiated from, as well as the data values for its attributes, it becomes clear that any additional log entry not present in the log entries of the original *Transfer* object would indicate that the object instance was changed in an ad-hoc fashion.

Therefore, combining the fact that we can create copies of object instances using their log-defined form with the ad-hoc addition of new log entries, we can create ad-hoc changed copies of objects instead of identical ones. An abstract view of the procedure, related to our running example, i.e., adding a *Comment* attribute to a *Transfer* instance, *Transfer#77#TEMP*, is shown in Fig. 9.

Note that there are some extra steps involving the temporary *Transfer#77#TEMP* object. These steps become necessary to support some of the requirements stated in Section III. Requirement 1, for example, states that atomicity of multiple changes has to be ensured, as individual changes might render an already running object instance in an incorrect state according to the syntactic and semantic correctness criteria of object-aware process management. This means that changes belonging together must be completed in an atomic fashion assuming that these changes result in a semantically correct object (cf. Requirement 2). Both these requirements necessitate the creation of a temporary copy of the object instance, (cf. Fig. 9, Marking **(1)**).

The temporary object instance copy is editable. For example, in our current implementation of object-aware process management, PHILharmonicFlows, we allow editing the underlying lifecycle process model in the modeling environment. After the temporary object instance is edited and verified for correctness, the changes applied to it can be propagated to the original, "live" object instance in an atomic fashion. To be precise, the change log entries created while editing (cf. Fig. 9, Marking **(2)**) constitute the delta of the ad-hoc change, i.e, the differences between the original object instance and an ad-hoc changed object instance. To express this formally, we introduce the log delta Δ between two instances of the same object.

Definition 3. (Log Delta Δ)
A set $\langle l_n, \ldots, l_m \rangle$ is called the log delta Δ between $O\#i$ and $O\#j$ if the following holds:

- l_i is a change log entry $\forall i \in \langle n, \ldots, m \rangle$
- $O\#i$ and $O\#j$ are log-defined object instances of the same object O
- $O\#i.log$ and $O\#j.log$ are the change log entries of $O\#i$ and $O\#j$, respectively
- $\langle l_n, \ldots, l_m \rangle = O\#i \Delta O\#j = O\#i.log \setminus O\#j.log$

In the example from Fig. 9, $Transfer\#21 \Delta Transfer\#77 = \langle l_{14}, l_{15}, l_{16} \rangle$ holds, i.e., the structural difference between the unchanged instance and the ad-hoc changed instance is determined by the actions logged in l_{14}, l_{15}, and l_{16}. As previously stated, editing the temporary copy allows support for Requirement 1, as the original object instance stays untouched until the ad-hoc change is completed. Furthermore, before completing the second copy operation (cf. Fig. 9, Marking (3)), the entire set of applied changes can be verified using static model verification before the ad-hoc changes go "live", thereby supporting Requirement 2. Finally, after completing this second copy operation, two *Transfer#77* object instances exist, the original, unchanged instance, and the instance copied from the temporary instance *Transfer#77#TEMP*. In fact, this temporary instance also still exists. As shown in Fig. 9, Marking **(4)**, these extra copies must be deleted, which causes the ad-hoc changed instance to become part of the running process, replacing the unchanged instance in one atomic operation. The algorithm underlying the concept is shown in Algorithm 1.

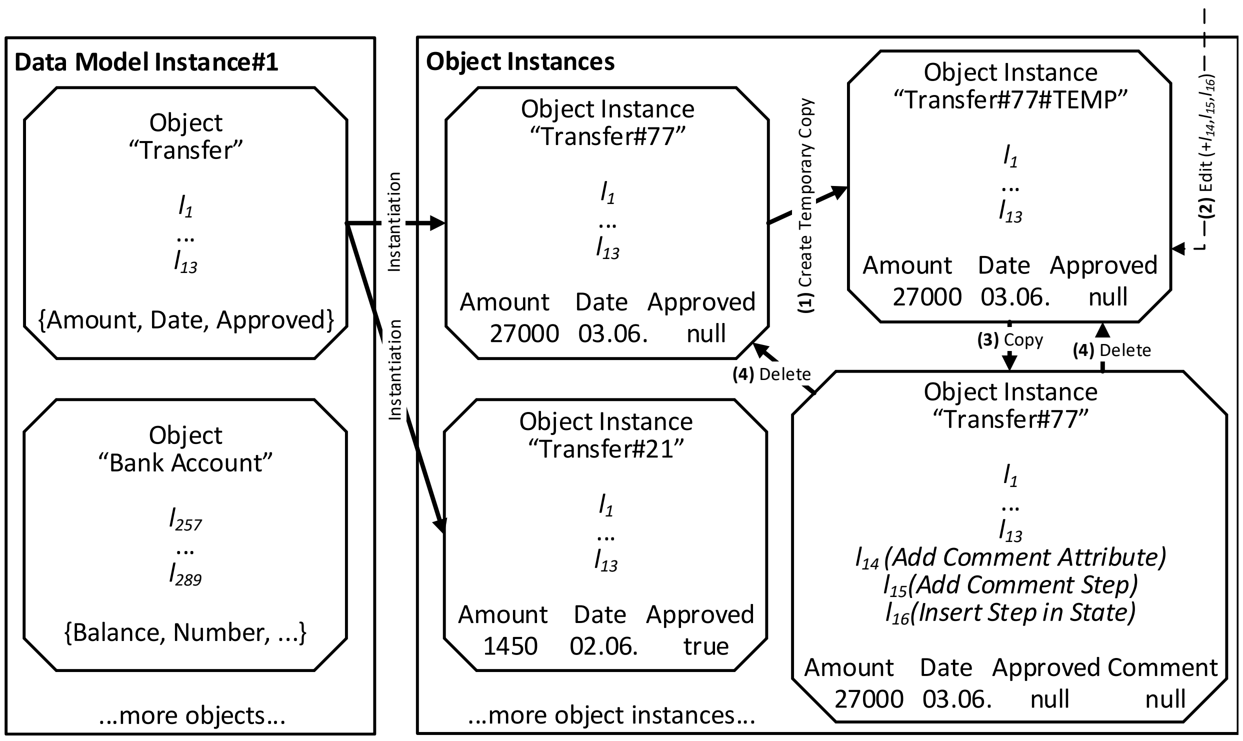


Fig. 9. Creating Ad-hoc Changed Object Instance

Algorithm 1 Creating Ad-hoc Changed Object Instance

```
Require: $O.log, O.data$    ▷ log entries and data of log-defined object instance O
  $O_{temp} \leftarrow$ new
  for all $l$ in $O.log$ do    ▷ copy O by change log replay
    $O_{temp}.replayChangeLog(l)$
  end for
  $allowediting(O_{temp})$    ▷ log($O_{temp}$) altered via changes in modeling tool
  if $modelVerificationErrors(O_{temp}) = 0$ then    ▷ ensure change is valid
    $O_{adhoc} \leftarrow$ new
    for all $l$ in $log(O_{temp})$ do    ▷ copy $O_{temp}$ by change log replay
      $O_{adhoc}.replayChangeLog(l)$
    end for
    for all $d$ in $O.data$ do    ▷ insert attribute values from O
      $O_{adhoc}.changeAttributeValue(d)$ ▷ each value advances the lifecycle
    end for
    $delete(O_{temp})$
    $delete(O)$
    $O \leftarrow O_{adhoc}$
  end if
```

It is important to mention that the lifecycle process of all object instances is data-driven. Thus, it gets re-executed instantly after copying, based on the lifecycle process itself and the current data values. This becomes necessary when aiming to support Requirement 3. As the latter states, all processes must be run-time consistent at all times. For object instances that have progressed to a particular state, this would usually mean that inserting required data input steps in earlier states would not be possible as the object instance could not have reached its current state after the change. However, by forcing re-execution, our concept ensures that object instances always have a consistent run-time state. To be more precise, if changes were introduced which require data input in states prior to the one the original object is currently in, the ad-hoc changed object simply executes to the step which requires data and stops execution until a user has entered the newly required data value. Once this is done, the rest of the data imported from the original object instance is used to complete the lifecycle process to the point it was at before the ad-hoc changes were incorporated.

An example can be seen when comparing the forms displayed in Figs. 2 and 8. While the form in Fig. 2 is generated from an unchanged *Transfer* object instance, Fig. 8 shows the updated form immediately after applying the ad-hoc changes introducing the new attribute *Comment* and the corresponding lifecycle step. As the *Comment* attribute is required before setting the *Approved* attribute, in line with the ad-hoc changes to the lifecycle process, the form generated for the *Decision Pending* state updates accordingly.

The scope of changes possible with this initial concept is limited to modeling elements that are directly attached to individual object instances, i.e., steps, states and transitions in the object's lifecycle process, as well as attributes and permissions. However, expanding upon the presented concept by enabling ad-hoc changes at the data model instance level removes this restriction. Finally, due to the large number of possible object instances in one single data model instance at run-time, performing ad-hoc changes on individual object instances might be too time consuming for users to be a feasible approach.

B. Data Model Instance Level Changes

After presenting the concept for introducing ad-hoc changes to individual object instances, we move on to the more challenging task of applying ad-hoc changes at the data model instance level. Note, that allows performing ad-hoc change operations on any part of a data model instance, i.e., the relations, the coordination processes, and the objects themselves. As explained in Section II, changes applied at the data model instance level do not propagate to the deployed data model. In consequence, the changes applied to one data model instance do not affect other data model instances created from the same deployed data model. However, ad-hoc changes on the data model instance level do constitute an evolutionary change, as they propagate to all existing and future object instances present in the data model instance (cf. Fig. 6).

To facilitate ad-hoc changes to data model instances, two core aspects are necessary. First, the data model instance has to be ad-hoc editable and must be changeable without affecting the deployed data model it was instantiated from. Second, modeling changes made to objects must propagate to all corresponding object instances, which poses additional challenges if some of the object instances have prior individual ad-hoc changes applied (cf. Requirement 4).

As explained in Definition 1, all modeling actions performed on a data model are logged in the change log. Change log entries can, however, not only be used to create a log-defined view on an object instance (cf. Definition 2), but also of an entire data model instance. There is, however, a fundamental difference between the log-defined view of an object instance and the log-defined view of a data model instance. As the data model instance itself does not hold any data, its execution state is defined by the data in its object instances, as well as the execution state of the coordination process. This, in turn, depends solely on the relations that exist between the object instances, as well as their current states [4]. The log-defined view of a data model instance is defined as follows.

Definition 4. Log-defined Data Model Instance
A tuple M=(log, objs, rels) is called log-defined data model instance if the following holds:

- log is a sequence of change log entries L (cf. Definition 1)
- objs is a set of log-defined object instances O (cf. Definition 2)
- $rels \subseteq objs \times objs$ is a set of relations between objects

The log-defined view of the data model instance allows for creating a temporary copy. Analogously to ad-hoc changes at the object instance level, this is used to meet Requirement 1, as incomplete ad-hoc changes are not applied to the "live" data model instance the users are working on. Additionally, it allows for full scale static model verification, a prerequisite of Requirement 2.

The following re-uses parts of the running example, the addition of a *Comment* attribute and corresponding step to the *Transfer* object. However, the change is now applied to the entire *Transfer* object and, in consequence, all associated *Transfer* object instances. Furthermore, we extend the example with the ad-hoc addition of a new object, *Foreclosure*, to the data model instance. Adding a new object is possible on the data model instance level as all changes that are possible at design-time may be incorporated into a data model instance at run-time (cf. Requirement 7). The entire process of applying these ad-hoc changes to a data model instance is shown in Fig. 10.

The basic idea for incorporating ad-hoc changes to the data model instance level is the same as for the object instance level. However, there is a fundamental difference, as the data model itself is not "executed" like the lifecycle process of an object instance. This means that the re-execution does not apply to the data model instance, i.e., there is no need to recreate the entire data model instance (however, one must still re-execute all contained and changed object instances). Instead, we determine the log delta between the original and the temporary data model instance. Clearly, $DataModelInstance\#1 \Delta DataModelInstance\#1\#TEMP = \langle l_{14}, l_{15}, l_{16}, l_{17} \rangle$, i.e., exactly the log entries created by the user when editing the temporary data model instance (cf. Fig. 10, Marking **(2)**).

Due to the editing of a copy of the data model instance, which includes all object instances, the concept further meets Requirement 4. To be more precise, during editing a user can be warned by the modeling user interface, that the change he wants to apply to an object is in conflict with a previously applied ad-hoc change on one of the existing object instances. Once a user has finished editing the temporary data model instance, the changes described in the log entries are applied to the original data model instance (cf. Fig. 10, Marking **(3)**).

Finally, the existing object instances have to be migrated to their updated objects, i.e., in the modified example, both *Transfer* instances must have the *Comment* attribute added. This process, depicted in Fig. 10, Markings **(4)**, **(5)**, and **(6)**, is almost analogous to the process of incorporating ad-hoc changes to individual object instances (cf. Section IV-A). In fact, the ad-hoc changes that are applied to the object instances in this case are the evolutionary changes propagated from the objects present in the data model instance.

In summary, the presented concept allows for ad-hoc changes to running object-aware process instances. While our examples are focused on ad-hoc changes to objects and individual object instances, the concepts can actually be adapted to relations and coordination processes as well.

V. Evaluation

For Requirements 1-4, we laid out our solutions in Section IV. This Section evaluates the remaining requirements. Requirement 5 states that the migration of multiple object instances in response to an ad-hoc change to a data model instance should occur in parallel and independent of the execution of other objects. To facilitate this we chose a fully distributed microservice-based implementation architecture for the PHILharmonicFlows process execution engine. In this architecture, which mirrors the conceptual elements of object-aware process management, each object and object instance is a separate microservice. This allows us to complete copy and change operations as displayed in Fig. 10 in parallel, as they

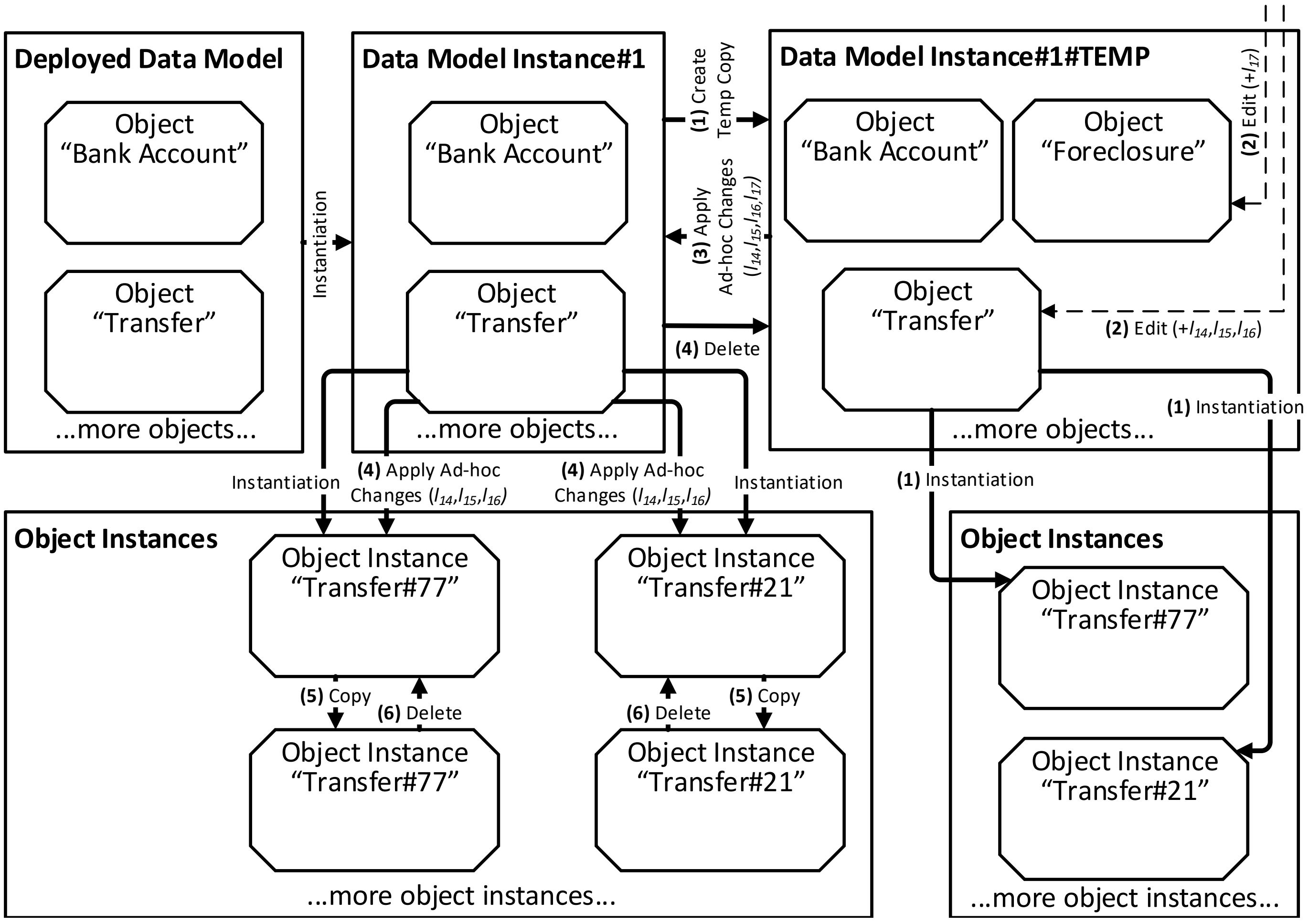

Deployed Data Model
Object "Bank Account"
Object "Transfer"
...more objects...
Instantiation
Data Model Instance#1
Object "Bank Account"
Object "Transfer"
...more objects...
(1) Create Temp Copy
(3) Apply Ad-hoc Changes (I_{14},I_{15},I_{16},I_{17})
(4) Delete
Data Model Instance#1#TEMP
Object "Bank Account"
Object "Foreclosure"
Object "Transfer"
...more objects...
(2) Edit (+I_{17})
(2) Edit (+I_{14},I_{15},I_{16})
(1) Instantiation
Instantiation
(4) Apply Ad-hoc Changes (I_{14},I_{15},I_{16})
(4) Apply Ad-hoc Changes (I_{14},I_{15},I_{16})
Instantiation
(1) Instantiation
Object Instances
Object Instance "Transfer#77"
Object Instance "Transfer#21"
(5) Copy
(6) Delete
(6) Delete
(5) Copy
Object Instance "Transfer#77"
Object Instance "Transfer#21"
...more object instances...
Object Instances
Object Instance "Transfer#77"
Object Instance "Transfer#21"
...more object instances...

Fig. 10. Creating Ad-hoc Changed Data Model Instances

are actually being executed on separate microservices. Furthermore, we solve the minor issue mentioned in Requirement 5, involving multiple users concurrently conducting ad-hoc changes to the same object instance, by utilizing the the *actor* concurrency model for our microservices. [9]. In essence, the actor model forces each microservice to behave as if it had only one conceptual thread, i.e., each microservice can only complete one action at a time, solving many concurrency problems, including concurrent ad-hoc changes. Scalability still ensured through the use of a large number of microservices for data model instance, one for each object instance.

As an ad-hoc change to a data model instance might trigger changes to a large number of object instances, this also causes a large number of copies of the changed object instances to be re-executed based on their updated lifecycle processes or attributes. While Requirement 5 still holds, as the microservice-based architecture is inherently scalable, there is also a bottleneck. When the lifecycles of the affected object instances are re-executed, they continuously change their states. In turn, because of this, the coordination process instance they are assigned to determines the overall execution state of the data model instance. As set out by Requirement 6, this is essential to ensure that changed object instances are coordinated properly and do not leave the data model instance in an inconsistent state. While we have not yet measured the time ad-hoc changes take in very large data model instances, we have ensured that we can improve the speed through horizontal scalability utilizing the aforementioned microservice architecture. While this is still a threat to validity, we find that it is more important to favor correct coordination over speed. Furthermore, we assume that under real-world conditions, ad-hoc changes to data model instances will not occur frequently. This assumption is based on the fact that frequent ad-hoc changes indicate a shortcoming in the process model, which could instead be alleviated by deploying an updated version of the model instead of conducting ad-hoc changes on each instance.

VI. Related Work

As the maturity of the data-centric BPM field is generally considered low compared to activity-centric approaches, execution concepts and engines are rare. Directly related work, i.e., data-centric approaches offering flexibility in the form of ad-hoc changes to running process instances, is virtually non-existent to the best of our knowledge.

However, note that related data-centric BPM approaches already a offer high level of flexibility in their process execution, as is the case in artifact-centric business processes [10] or case handling [11], which is be expected due to the largely declarative modeling nature [12]. The tooling support for case handling (i.e. FLOWer [11]), for instance, offers ad-hoc flexibility in the form of skip or redo capabilities. However, this flexibility is restricted to control-flow aspects.

The DEZ-Flow engine [13], built upon the artifact-centric approach, allows for the definition of declarative rules which allow for ad-hoc changes to running instances at predefined points. While these rules are editable at run-time, the approach does not cover every possible deviation from the standard process as the process model remains unchanged.

A declarative process support approach, which also enables ad-hoc changes, is presented in [14]. Thereby, a change is defined by adding, deleting or updating the constraint set of a particular process instance or process type.

COREPRO [15] supports the assembly of products from product components, as it is commonplace in the automotive industry. This requires the adaptation and coordination of large process structures, represented by individual data objects. Adaptation is done directly to running instances, and changes that lead to an inconsistent process state are prevented.

In-depth research into model changes in Adaptive Case Management (ACM) is conducted in [16]. Specifically, the conduction of change operations is examined to determine the impact they have on a given GSM model. However, the paper is limited in respect to run-time aspects for the adaptation of existing process instances.

Finally, there are many approaches to process flexibility in activity-centric BPM, but their ad-hoc change support is limited to moving, skipping, etc. entire activities in contrast to the fine-grained support PHILharmonicFlows offers. Additionally, activity-centric ad-hoc change concepts do not allow for the migration of all existing process instances, as they can not be re-executed in the integrated fashion presented in this work. This frequently leads to scenarios where running process instances are simply not migratable for certain changes [1] [17] [18] [19].

VII. Summary and Outlook

The concepts presented in this paper allow for a multitude of ad-hoc changes to object-aware process instances, both to individual object instances and entire data model instances. The concepts are designed in a way that allows for their use in a microservice-based process engine, PHILharmonicFlows, utilized by us as a proof-of-concept demonstration of the presented concepts. Furthermore, as object-aware process management has an inherently tight integration between process logic and data, our proof-of-concept has capabilities that extend beyond those of existing, activity-centric ad-hoc change solutions.

We intend to address the threats to validity revolving around the performance of the developed solution (cf. Section V) in a future paper, once we have determined an adequate test setup for change scalability. This will be done in conjunction with the further research on the topic of full data model evolution, including all attached data model instances and their object instances, as this is where performance might become an issue.

While the presented solution might not be evaluated for usability in real-world, it is important to note that the actual implementation of such advanced concepts is crucial as a proof of concept for the entire field of data-centric BPM, as the availability of tooling is central to increasing maturity.

Acknowledgments: This work is part of the ZAFH Intralogistik, funded by the European Regional Development Fund and the Ministry of Science, Research and the Arts of Baden-Württemberg, Germany (F.No. 32-7545.24-17/3/1)

References

[1] M. Reichert and B. Weber, *Enabling flexibility in process-aware information systems: challenges, methods, technologies*. Springer Science & Business Media, 2012.

[2] K. Andrews, S. Steinau, and M. Reichert, "Towards hyperscale process management," in *Proc EMISA*, 2017, pp. 148–152.

[3] V. Künzle and M. Reichert, "PHILharmonicFlows: towards a framework for object-aware process management," *JSME*, vol. 23, no. 4, pp. 205–244, 2011.

[4] S. Steinau, V. Künzle, K. Andrews, and M. Reichert, "Coordinating business processes using semantic relationships," in *Proc CBI*, 2017, pp. 143–152.

[5] B. Weber, M. Reichert, and S. Rinderle-Ma, "Change patterns and change support features," *DKE*, vol. 66, no. 3, pp. 438–466, 2008.

[6] W. Song, X. Ma, and H.-A. Jacobsen, "Instance migration validity for dynamic evolution of data-aware processes," *IEEE TSE*, 2018.

[7] C. M. Chiao, V. Künzle, and M. Reichert, "Object-aware process support in healthcare information systems," *IJAL*, vol. 5, no. 1 & 2, pp. 11–26, 2013.

[8] K. Andrews, S. Steinau, and M. Reichert, "Enabling process variants and versions in object-aware process management systems," in *Proc CAiSE Forum*, 2018.

[9] G. Agha and C. Hewitt, "Concurrent programming using actors," in *Distributed Artificial Intelligence*. Elsevier, 1988, pp. 398–407.

[10] D. Cohn and R. Hull, "Business artifacts: A data-centric approach to modeling business operations and processes," *IEEE TCDE*, vol. 32, no. 3, pp. 3–9, 2009.

[11] W. M. P. Van der Aalst, M. Weske, and D. Grünbauer, "Case handling: a new paradigm for business process support," *DKE*, vol. 53, no. 2, pp. 129–162, 2005.

[12] W. M. van Der Aalst, M. Pesic, and H. Schonenberg, "Declarative workflows: Balancing between flexibility and support," *Computer Science-Research and Development*, vol. 23, no. 2, pp. 99–113, 2009.

[13] W. Xu, J. Su, Z. Yan, J. Yang, and L. Zhang, "An artifact-centric approach to dynamic modification of workflow execution," in *OTM*. Springer, 2011, pp. 256–273.

[14] M. Pesic, M. H. Schonenberg, N. Sidorova, and W. M. P. van der Aalst, "Constraint-based workflow models: Change made easy," in *Proc CoopIS'07*, 2007, pp. 77–94.

[15] D. Müller, M. Reichert, and J. Herbst, "Flexibility of data-driven process structures," in *Proc BPM*. Springer, 2006, pp. 181–192.

[16] R. Eshuis, R. Hull, and M. Yi, "Property preservation in adaptive case management," in *International Conference on Service-Oriented Computing*. Springer, 2015, pp. 285–302.

[17] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, and W. van der Aalst, "Process flexibility: A survey of contemporary approaches," in *Advances in enterprise engineering I*. Springer, 2008, pp. 16–30.

[18] S. Rinderle, "Schema evolution in process management systems," Ph.D. dissertation, Uni Ulm, 2004.

[19] B. Weber, M. Reichert, J. Mendling, and H. A. Reijers, "Refactoring large process model repositories," *Computers in Industry*, vol. 62, no. 5, pp. 467–486, 2011.