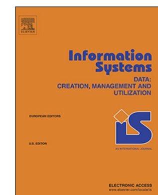
ARTICLE IN PRESS

Information Systems xxx (xxxx) xxx

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is



Enabling runtime flexibility in data-centric and data-driven process execution engines

Kevin Andrews *, Sebastian Steinau, Manfred Reichert

Institute of Databases and Information Systems, Ulm University, Germany

ARTICLE INFO

Article history:
Received 14 March 2019
Received in revised form 20 August 2019
Accepted 30 September 2019
Available online xxxx
Recommended by Gottfried Vossen

Keywords:
Business process flexibility
Ad-hoc change
Object-aware processes
Data-centric processes
Data-driven processes

ABSTRACT

Contemporary process management systems support users during the execution of predefined business processes. However, when unforeseen situations occur, which are not part of the process model serving as the template for process execution, contemporary 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 properly supported by most commercial process engines, there is no corresponding support for ad-hoc changes in other process support paradigms, such as artifact-centric or object-aware process management. This article presents concepts for supporting ad-hoc changes in data-centric and data-driven processes, and gives insights into the challenges to be tackled when implementing this kind of process flexibility in the PHILharmonicFlows process execution engine. We evaluated the concepts by implementing a proof-of-concept prototype and applying it to various scenarios. The development of advanced flexibility features is highly relevant for data-centric processes, as the research field is generally perceived as having low maturity compared to activity-centric processes.

© 2019 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

As one of the major advantages of using process management technology in enterprises, the interactions between users and IT systems can be adapted quickly when changes to real-world business processes occur [1]. These adaptations are enabled by changing the corresponding process models in a process management system [2]. In particular, this allows processes to be updated and improved, supporting more process execution variants not thought of during initial modeling. However, process models are often not detailed enough to adequately support each and every possible process execution variant. Furthermore, process variants exist that occur so rarely that incorporating all their details into the process model would increase complexity at low benefit. In these cases, *ad-hoc changes* to running process instances become necessary, a topic that has been addressed many times for activity-centric process management systems [3–6].

1.1. Problem statement

Commonly, data-centric and data-driven process support paradigms are considered to be more flexible in regards to process execution than the well-established activity-centric paradigm [7]. This can be explained with the fact that activity-centric processes are usually well structured and only offer possibilities to deviate from a fixed path at gateways or decision points. Besides that, in activity-centric processes, process activities are executed in exactly the order predefined by the process modeler. In contrast, data-driven and data-centric process support paradigms, such as artifact-centric processes [8] or case handling [9], are more flexible, using various mechanisms to define constraints on which activities are available for execution. Furthermore, as these paradigms are data-driven, the availability of data drives process execution, instead of the completion of activities as is the case in activity-centric processes. Note that this leads to an increased flexibility as the order of activity execution is largely up to the user, as long as the defined constraints are adhered to. However, even for processes whose execution is based on these paradigms, the need might arise to perform changes to these constraints at runtime, constituting *ad-hoc changes*.

In particular, we chose to extend an existing data-centric and data-driven process support paradigm, *object-aware process management*, with innovative concepts enabling ad-hoc changes. The necessity of this work was established in the context of several related projects and industrial case studies. In these we created multiple object-aware process models in various

* Corresponding author.
E-mail address: kevin.andrews@uni-ulm.de (K. Andrews).

https://doi.org/10.1016/j.is.2019.101447
0306-4379/© 2019 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

domains, such as intralogistics, e-learning, human resource management, and healthcare. We identified various scenarios in these domains, in which ad-hoc changes to processes could offer benefits even when supported by the inherently flexible object-aware paradigm. In particular, we wanted to create a concept for incorporating ad-hoc changes into object-aware processes that is not only conceptually sound, but actually usable at the operational level, both from a user and a performance perspective. Accordingly, our research questions were the following:

RQ1 How can we enable ad-hoc changes to running object-aware process models at runtime without disruptions?

RQ2 How can we make this feature flexible enough such that it allows changing every aspect of a process model?

RQ3 How can we make the concept efficient enough that it can be used for evolving real processes with hundreds or thousands of instances?

1.2. Contribution

This article offers a fundamental approach for introducing the concept of ad-hoc process changes at runtime to object-aware process management [10]. The major contributions are threefold:

1. We provide a detailed set of concepts, algorithms, and implementation details that enable ad-hoc changes to object-aware processes at run-time.
2. We detail how we leverage the object-aware paradigm to ensure runtime correctness of ad-hoc changed process instances, as well as a method to reconstruct a consistent process state after introducing ad-hoc changes.
3. We suggest additional extensions to the initial *ad-hoc change concept* that ensure its performance viability for scenarios involving changes that affect large numbers of process instances.

This article provides a significant extension of the work we introduced in [11]. While in [11] we presented an initial concept for ad-hoc changes in object-aware processes, the evaluation of the concept had not been completed. In this article, we not only present a revised and improved version of the initial concept, which is no longer adversely impacted by the quality of the process model, but also a thorough evaluation of the original concept and the presented revisions and extensions. The evaluation, which extends our previous work, was completed by utilizing a proof-of-concept implementation of the concepts in various scenarios in the course of multiple projects. Furthermore, we conducted performance measurements to quantify the improvements we introduced while revising and improving the initial concept. In summary, the new evaluation shows that ad-hoc changes are not only feasible, but also very useful in various scenarios.

1.3. Methodology & outline

We approached the solutions to our research questions and the development of the contribution with the same design-science based research methodology we have been employing for the development of the object-aware and data-driven process engine PHILharmonicFlows, of which Section 2 gives a short overview required for understanding this work. Section 3 presents the requirements we identified for ad-hoc changes in the object-aware paradigm. The main contributions of this article, Sections 4 and 5, are structured along the design-science iterations we went through when developing the concept for ad-hoc changes. The initial iteration on the idea was developed for the smallest scope in an object-aware process, i.e., one single object, and is described in Section 4.1. The concept was validated in a series of single case mechanism experiments and then extended to include support for entire process models (cf. Section 4.2). As the initial idea for supporting entire processes did not function perfectly in all scenarios, it was extended with additional algorithms, which we describe in Section 4.3. Further iterations on the concept concerned performance of the contribution, the results of which we present in Section 5. A description of multiple scenarios, in which the contribution was evaluated in terms of a sophisticated prototypical implementation, can be found in Section 6. Section 7 discusses related work and Section 8 gives a short summary and an outlook.

2. Fundamentals

This section provides and overview of the conceptual foundations of object-aware process management, which are crucial for understanding this work.

2.1. Object-aware process management

PHILharmonicFlows, the object-aware process management framework we are using as a test-bed for the concepts presented in this article, has been under development at Ulm University for many years [12–15]. PHILharmonicFlows takes the idea of a data-driven and data-centric process management system, enhancing it with the concepts of *objects* and *object relations*. For each business object present in a real-world business process one such object exists. 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 data-driven *object lifecycle*.

The attributes of the *Transfer* object (cf. Fig. 1) include *Amount*, *Date*, and *Approved*. The *lifecycle process*, in turn, describes the different *states (Initialized*, *Decision Pending*, *Approved*, and *Rejected)*, an instance of a *Transfer* object may enter during process execution. Each state comprises one or more *steps*, each referencing exactly one of the object attributes and enforcing that the respective attribute is written at runtime. The steps are connected by *transitions*, which arrange them in a sequence. The state of the object changes after all steps in a state are completed, i.e., after all corresponding attributes are written. Finally, alternative paths are supported in terms of *decision steps*, an example of which is the *Approved* decision step (cf. Fig. 1).

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

As a major benefit, the object-aware approach allows for *automated form generation* at runtime. 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. This information is combined with a set of read and write permissions, resulting in a personalized and dynamically created user 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 forms only constitute one part of a complete PHILharmonicFlows process. To

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

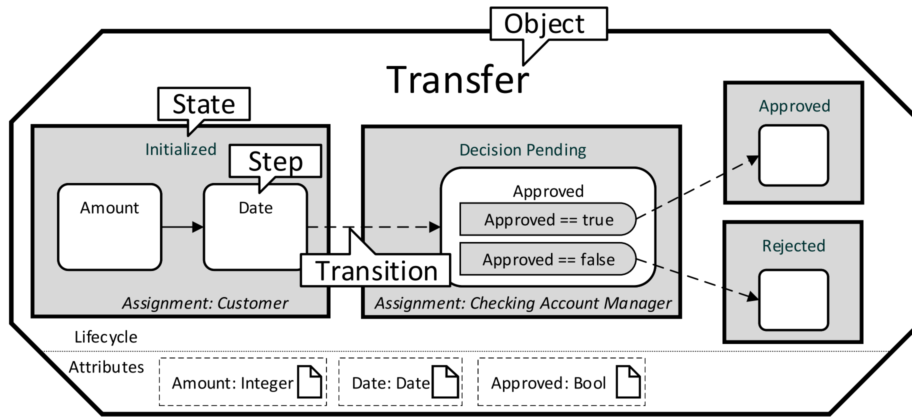


Fig. 1. Example object including lifecycle process.

Bank Transfer – Decision
Amount 27.000 €
Date 03.06.2017
Approved* true
Submit

Fig. 2. Example form.

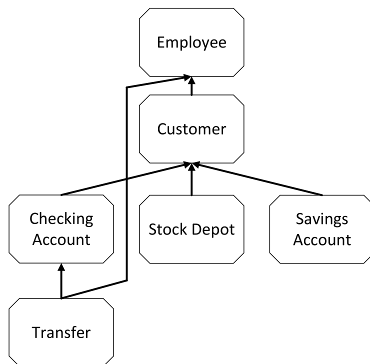


Fig. 3. Design time data model.

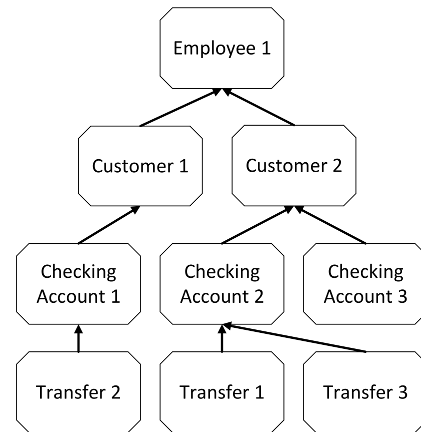


Fig. 4. Data model instance.

allow for more complex executable business processes, many different objects and users may have to be involved [16]. It is noteworthy that *users* are simply special objects in the object-aware process management concept. The entire set of objects present in a PHILharmonicFlows process is denoted as the *data model*, an example of which is depicted in Fig. 3. Note that this is a simplified representation of a data model, omitting advanced concepts such as cardinalities and roles (see [12,14] for details).

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 *Transfer* and a *Checking Account*. At runtime, each of the objects may be instantiated many times as so-called *object instances*. The lifecycle processes present in the various object instances may then be executed concurrently at runtime, thereby improving overall system performance. Furthermore, the relations can be instantiated at runtime, 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, expressing 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 [12]. Fig. 4 shows an example of a *data model instance* executed at runtime.

Finally, complex object coordination is supported as well. The latter becomes necessary as business processes often consist of hundreds or thousands of interacting business objects [16,17], whose concurrent processing needs to be synchronized at certain states. As objects publicly advertise their state information, the current state of an object can be utilized as an abstraction for coordinating its execution with other 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*.

In our current proof-of-concept prototype, the various conceptual elements of object-aware processes, i.e., objects, relations, and coordination processes, are implemented as microservices. For each object instance, relation instance, or coordination process instance, one microservice instance is created at runtime, turning the implementation, PHILharmonicFlows, into a distributed process management system for object-aware processes.

2.2. Process model evolution and ad-hoc changes

As motivated in Section 1, 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* [2].

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

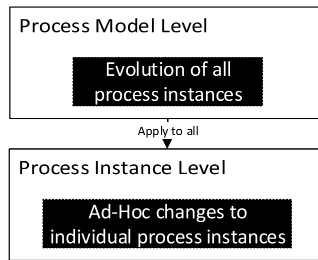


Fig. 5. Change granularity levels for activity-centric processes.

Deferred process model evolutions are changes that are introduced by deploying updated process model versions without applying the changes to already existing process instances. In essence, a deferred process model evolution simply corresponds to the introduction of a new process model version, which then exist in parallel to older versions. Therefore, as existing process instances remain untouched, as they are executing the (still existing) older version of the process model, this is a rather trivial case.

Immediate process model evolutions not only allow for the process model to be updated, but also try to migrate already running process model instances to the new model version. In general, an immediate migration poses significant challenges to a process management system. Take the migration of process instances that have already executed parts of the process model to which changes are made. This poses a significant challenge as it is hard to ensure process consistency after the change [18]. Immediate process model evolutions are required in 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.

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 one central entity to which all these changes are applied, i.e., the *process model* [2]. While evolutionary changes might be applied directly to the process model all corresponding process instances are derived from, ad-hoc changes are solely applied to a specific process instance. 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.

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, whereas ad-hoc changes may be applied to data model instances and object instances, analogously to the activity-centric paradigm. However, considering that additional object instances may be created at any point during process execution, with only two levels of granularity, it is not 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 is defined: the object instance level. The resulting three change granularity levels for object-aware processes are depicted in Fig. 6.

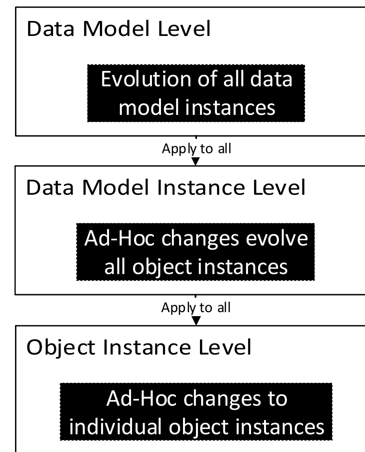


Fig. 6. Change granularity levels for object-aware processes.

It is noteworthy that ad-hoc changes to objects on the data model instance level are propagated to all existing and future 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 offers support for a more complete set of change operations. In addition to the changes possible for object instances, i.e., adding attributes and permissions as well as editing the lifecycle process, one may also introduce changes to the data model instance itself, such as adding 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. On the other hand, the object instance level is limited to changes of 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 constitute the focus of this article.

3. Requirements

This section presents major requirements we elicited regarding the support of ad-hoc changes in object-aware process management. On one hand, they were derived from the general requirements for activity-centric processes [2] and adapted as necessary. On the other, we considered data model change operations in a number of object-aware processes we analyzed in various domains, such as intralogistics, e-learning, human resource management, and healthcare [19]. Finally, we developed an extensive framework for systematically comparing and evaluating the capabilities of various data-centric process support paradigms in the course of which we managed to identify further requirements [7].

Requirement 1 (*Change Atomicity*)**.** Existing object instances should not reflect ad-hoc changes immediately, as individual changes to an object instance might render it semantically or syntactically incorrect until other changes are applied as well. As an example consider the insertion of a single 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 constitutes a syntactical error in an object lifecycle

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

ARTICLE IN PRESS

process. Consequently, if this change was introduced to a running process instance runtime failures would be caused. In general, a capability must be developed that allows introducing multiple changes to running process instances in an atomic fashion. In the example of adding a step, the atomic change would therefore consist of adding the step and all transitions, ensuring that the running process instances never enter an incorrect state.

Requirement 2 (*Correctness*)**.** The changes that may be applied to object instances should result in a correct process model again, i.e., the verification criteria applied at design time must apply in the context of ad-hoc changes as well. Reiterating the previous example of adding a step to a lifecycle process, the entire atomic change (i.e., sequence of individual changes) to the running object instance must result in a correct lifecycle process model again.

Requirement 3 (*Runtime Consistency*)**.** An object instance must never enter a lifecycle process state it would not be able to reach if it were re-executed in an identical fashion after an ad-hoc change. For example, consider an object-instance with a lifecycle process that has two states, *A* and *B*, with a few steps each. Assume further that the object instance has completed state *A* and is currently in state *B*. If an ad-hoc change was to add a step in state *A,* which the lifecycle process has already progressed past, it would be inconsistent for the object instance to remain in state *B* without having completed the newly required step in state *A*. 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. However, the existing object instance would have already progressed past this point before the ad-hoc change.

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 need to be resolved. Consider an object instance with an additional transition between two steps, added by an ad-hoc change on the object instance level at runtime. If a process modeler introduced an additional ad-hoc change at the data model instance level, e.g., the deletion of one of the steps the additional transition is connected to, this change to a specific object instance would be in conflict with the change affecting all existing object instances.

Requirement 5 (*Concurrency*)**.** Change operations should be applicable while the corresponding process instance is running, without hindering the execution of other object instances not concerned by the changes. Note that this is in contrast to activity-centric process management, where a single process instance often corresponds to a single business case. Specifically, we aim to offer 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 of the same object instance, as this can be trivially solved with locking mechanisms, i.e., by 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 [16], the state changes caused by ad-hoc changes need to be handled correctly. Such state changes may become necessary when required steps are inserted at earlier points in an object instance lifecycle process, as portrayed by the example introduced in the context of Requirement 3. Furthermore, through the removal of individual steps from a lifecycle process, the latter may advance to a different state as the result of an ad-hoc change. Both cases need to be handled correctly by the coordination process to ensure that other object instances react correctly to the changes.

Requirement 7 (*Completeness*)**.** The set of operations for expressing ad-hoc changes need to be complete in the sense that all aspects of the process model editable at design time should be editable at runtime as well. This allows for maximum flexibility when conducting changes, even if the entire set of operations is not necessary for most business cases. Additional concerns, such as determining the ideal subset of modeling operations that should be made available to users at runtime for conducting ad-hoc changes, are out of the scope of this article, but will be examined in future work.

Requirement 8 (*Algorithmic Performance*)**.**Though an ad-hoc change is an expensive operation, the concept should be reusable in future work concerning schema evolution, with potentially large numbers of data model instances receiving ad-hoc changes from an updated schema version of the data model at the same time. In general, one may expect that larger data models and their instances will require more computational efforts to migrate to ad-hoc changed versions than smaller ones. However, the scaling should be at most linear in respect to the number of entities contained in the data model.

4. 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 concepts have been fully implemented in the PHILharmonicFlows process execution engine.

4.1. Object instance level changes

An ad-hoc change to an object instance can be required by users for various reasons. As objects consist of multiple attributes, permissions, and a lifecycle process, a simple ad-hoc change would be the addition of an attribute as well as a corresponding lifecycle process step to the object instance at runtime. An example is depicted in Fig. 7 by the additional attribute *Comment*, and the corresponding step in the *Decision Pending* state. 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, i.e., the *Transfer* object (cf. Fig. 1), remains untouched as the ad-hoc change is only introduced on the object instance level, but not on the data model instance level. Such changes to all existing and future instances of an object are discussed in Section 4.2.

Note that from a user perspective the introduction of this change would automatically alter the form generated from this object at runtime. 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. However, after introducing the ad-hoc change, the instance displays a slightly different form to the *Checking Account Manager* (cf. Fig. 8). This altered form 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.

Supporting ad-hoc changes on the object instance level is accompanied by a number of challenges that need to be tackled in order to lay the foundation for data model instance level changes. The following concept solves these challenges in line with the requirements discussed in Section 3.

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

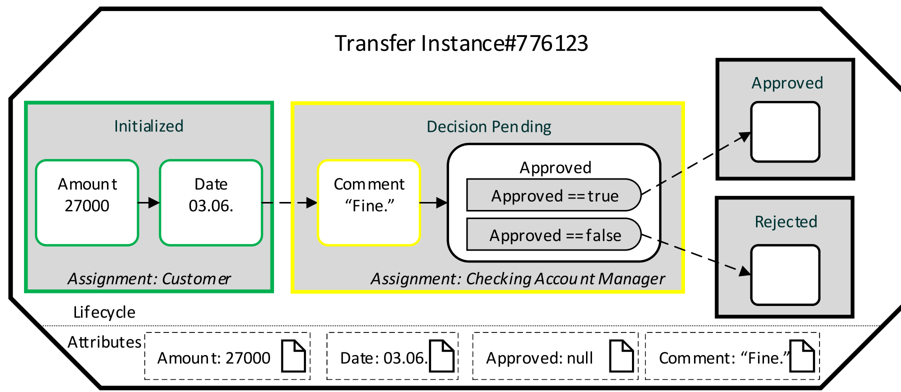


Fig. 7. Transfer object instance with added *Comment* attribute and step.

Bank Transfer – Decision
Amount 27.000 €
Date 03.06.2017
Comment* Fine.
Approved Select Value
Submit

Fig. 8. Dynamically generated form after ad-hoc change.

4.1.1. Change log entries

Fundamental to our concept is the notion of *change log entry*. A change log entry represents a change operation that was applied to some entity that is part of an object-aware data model. We omit the definition of a data model in this article (see [12] instead) as, for the intent of the presented concept, the data model merely serves as a container for the other conceptual elements present in an object-aware process model, e.g., objects and relations.

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 entity (e.g. object, relation, and coordination process)
- A is a modeling action that was applied to S (e.g. *AddAttribute*)
- P is a set of parameters with which A was applied to S
- T is the logical timestamp of the modeling action

For each modeling action completed by a user when creating or changing a data model, one such log entry is created. The sum of these log entries constitutes the *change log* of a data model. 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*)**.**

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

The logical timestamp T of l_{14} holds 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. Note that this 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 for more complex use cases related to changes, as the ones discussed in this article.

4.1.2. Log-defined object instances

The availability of change logs allows viewing an object-aware data model and the objects it contains from a new perspective, i.e., as the result of the application of all modeling actions recorded in the change log. Note that this perspective may be applied to both the data model instances and the object instances, as all individual instances are based on their models, which can be recreated by repeating the modeling actions contained in the change log entries.

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 can be explained by the fact that object-aware processes are inherently data-driven (cf. Section 2), meaning that the execution progress (i.e., the state) of each object is defined by its attribute values and 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 this 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 [10]. In particular, the previous definition focused on the actual object-aware entities that comprise the object, such as attributes, attribute values, permissions, and all entities 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 sequence of change log entries L (cf. Definition 1)
- data is a mapping of object attributes to values

As *O.log* contains log entries with logical timestamps, recreating the sequence of actions (with their corresponding 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 its value $O.data[a]$. In essence, this entire procedure allows serializing 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 to change existing object instances.

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

ARTICLE IN PRESS

Still, there are several reasons for considering the serialization and deserialization of objects to and from logs as a fundamental building block for our concept.

4.1.3. Introducing ad-hoc changes

In the following, we view all object instances as a log-defined ones, i.e., under the premise that an 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 that were supplied for its attributes. This way, it becomes clear that any additional log entry not present in the log entries of the original object would indicate that the object instance was changed in an ad-hoc fashion. 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, which is related to Example 1 (i.e., adding a *Comment* attribute to the *Transfer* instance *Transfer#77#TEMP*) is shown in Fig. 9.

Note that some extra steps involving the temporary *Transfer#77#TEMP* object instance (cf. Fig. 9) become necessary. These steps shall support some of the requirements introduced in Section 3. According to Requirement 1 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 [13]. This means that semantically related changes must be completed in an atomic fashion assuming that they result in a semantically correct object (cf. Requirement 2). Note that both requirements necessitate the creation of a temporary copy of the object instance (cf. Fig. 9, Marking **(1)**). To this end, we utilize *log replaying* to create the copies.

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

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

- l_i is a change log entry $\forall i = n \ldots m$
- $O\#i$ 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 $O\#j$
- $\langle l_n, \ldots, l_m \rangle \equiv O\#i \Delta O\#j \equiv (O\#i.log \setminus O\#j.log) \cup (O\#j.log \setminus O\#i.log)$

In the example from Fig. 9 (after the ad-hoc change has been completed), $Transfer\#21 \Delta Transfer\#77 = \langle l_{14}, l_{15}, l_{16} \rangle$ 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 for the support of 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", i.e., Requirement 2 is met. Finally, after completing this second copy operation, two *Transfer#77* object instances exist: the original (i.e., unchanged) instance and the one copied from temporary instance *Transfer#77#TEMP*. In fact, the latter still exists. As shown in Fig. 9, Marking **(4)**, these extra copies need to be deleted, which causes the ad-hoc changed instance to become part of the running process, replacing the unchanged instance in one atomic operation.

Algorithm 1: Creating ad-hoc changed object instance

```
Require: O.log, O.data ▷ log entries and data of log-defined object instance O
 1: O_temp ← new
 2: for all l in O.log do                      ▷ copy O by change log replay
 3:     O_temp.replayChangeLog(l)
 4: end for
 5: allowediting(O_temp)        ▷ log(O_temp) altered via changes in modeling tool
 6: if modelVerificationErrors(O_temp) = 0 then      ▷ ensure change is valid
 7:     O_adhoc ← new
 8:     for all l in log(O_temp) do       ▷ copy O_temp by change log replay
 9:         O_adhoc.replayChangeLog(l)
10:     end for
11:     for all d in O.data do           ▷ insert attribute values from O
12:         O_adhoc.changeAttributeValue(d)  ▷ each value advances the lifecycle
13:     end for
14:     delete(O_temp)
15:     delete(O)
16:     O ← O_adhoc
17: end if
```

Algorithm 1 describes the concept in pseudo code. Note that the *allowediting()* function mentioned in Line 5 simply allows editing the temporary object instance copy in the regular PHILharmonicFlows modeling environment.

The algorithm pauses execution until the user has modeled the ad-hoc change and continues execution at Line 6 once the user signals that the ad-hoc changes should be propagated to the "live" object instance. Note that the lifecycle process of all object instances is data-driven, i.e., the lifecycle process advances automatically when providing values to the attributes referenced in lifecycle steps. Thus, the lifecycle process gets re-executed instantly after copying and changing the attribute values of the copy, based on the lifecycle process itself and the current data values assigned to the attributes (cf. Line 12). After the re-execution is complete, object instance O is hot-swapped with the new object instance O_{adhoc}, minimizing disruptions, in line with research question *RQ1*. Note that re-execution is necessary to support Requirement 3.

4.1.4. Ensuring runtime consistency

As Requirement 3 states, all processes need to be *runtime 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 enforcing re-execution, our approach ensures that object instances always have a consistent runtime state. In detail, if changes are introduced that require data input in states prior to the one the original object is currently in, the ad-hoc changed object simply executes all steps up until the newly inserted step (which requires data currently not present). Here,

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

ARTICLE IN PRESS

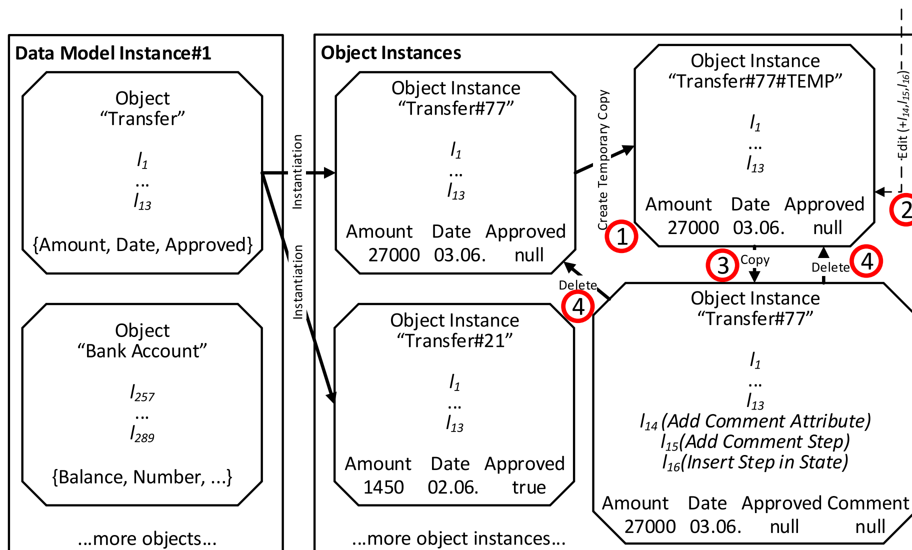


Fig. 9. Creating ad-hoc changed object instance.

the lifecycle process stops execution until a user has entered the newly required data value. Once this is done, the remaining data imported from the original object instance is used to execute the lifecycle process to the state it was in before the ad-hoc changes were introduced. The fine-grained rules for lifecycle execution established in [13] and the data-driven execution approach in general support this flexible style of instant and consistent re-execution. Further note that simple re-executability is one major advantage object-aware processes have over activity-centric processes in regards to ad-hoc changes as this ensures that the process is always in a consistent state.

Example 2 (*Implicit Ad-Hoc Changes to Generated Forms*)**.** The form in Fig. 2 was generated from an unchanged *Transfer* object instance. In turn, Fig. 8 shows the updated form immediately after applying ad-hoc changes that introduce a *Comment* attribute and the corresponding lifecycle step. As the *Comment* attribute is now required before the *Approved* attribute, in line with the ad-hoc changes to the lifecycle process, the form generated for the *Decision Pending* state updates accordingly.

Note that the capability of adding or changing form logic is innovative not just for a process management system, but even for more specialized information systems, such as ERP or CRM systems. Even the simple example of adding a single field at run-time and marking it as required is an incredible headache in most contemporary information systems. Considering more advanced examples, such as inserting entirely new states or changing permissions and, therefore, the flow of data between the information system and its users at run-time, the concept constitutes a considerable step forward towards a dynamically evolving information system based on the object-aware process support paradigm.

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 lifecycle process as well as attributes and permissions. However, expanding upon the presented concept by additionally 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 runtime, performing ad-hoc changes on individual object instances might be too time consuming for users to be a feasible approach.

4.2. 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 this 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 2, 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 constitute an *evolutionary change*, as they propagate to all existing and future object instances present in the given data model instance (cf. Fig. 6). Two core aspects are crucial to enable ad-hoc changes to data model instances:

1. The data model instance has to be ad-hoc editable and changeable without affecting the deployed data model it was instantiated from.
2. Changes made to objects must propagate to all corresponding object instances. This poses additional challenges if some of the object instances have prior individual ad-hoc changes applied (cf. Requirement 4).

As explained in the context of Definition 1, all modeling actions performed on a data model are recorded in the change log. However, change log entries may not only be used to create a log-defined view on an individual object instance (cf. Definition 2), but also on an entire data model instance, including all contained objects, relations, and coordination processes. Note that there is a fundamental difference between the log-defined view of an object instance and the one of a data model instance. As the data model instance itself does not hold any data, its execution state is defined by the data of its object instances as well as the execution state of the coordination process. This, in turn, solely depends on the relations that exist between the object instances and their current states [16]. On a side note, the log-defined views on relations and coordination processes that belong to a data model instance merely consist of change log entries. As they have no associated data of their own, their trivial definitions are omitted.

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

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

- 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 log-defined relation instances between objects
- coords is a set of log-defined coordination process instances

The log-defined view of the data model instance allows creating a temporary copy. Analogously to ad-hoc changes at the object instance level, this copy 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 a full scale static model verification, a prerequisite to meet Requirement 2.

The following reuses parts of the running example, the addition of a *Comment* attribute as well as a 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 runtime as well (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 re-execution does not apply to the data model instance itself, but only to the affected object instances. Instead, we determine the log delta (cf. Definition 3) between the original and the temporary data model instance. Obviously, $DataModelInstance\#1 \Delta DataModelInstance\#1\#TEMP = \langle l_{14}, l_{15}, l_{16}, l_{17} \rangle$, i.e., we can use the log delta to identify the log entries created by the user when editing the temporary data model instance and prepare them for distribution to all affected object instances (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 or she 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. Regarding the modified running example this means that both *Transfer* instances must have the *Comment* attribute added. This process is depicted in Fig. 10, Markings **(4)**–**(6)**. Note that this procedure is almost analogous to the one incorporating ad-hoc changes to individual object instances (cf. Section 4.1). In fact, the ad-hoc changes applied to the object instances are the evolutionary changes propagated from the objects to their respective instances. In summary, the presented concept allows for ad-hoc changes to running process instances supported by the object-aware paradigm. While the examples focus on ad-hoc changes to objects and individual object instances, in the current prototype (cf. Section 6.1) the concept is implemented with support for ad-hoc changes to relations and coordination processes as well. Therefore, the concept can be utilized to change every aspect of a process model (cf. research question *RQ2*).

4.3. Interdependent ad-hoc changes

As emphasized in Section 4.2, ad-hoc changes may not only be applied to objects. In particular, the relations and coordination processes present in an object-aware process instance are designed to produce the same kind of change logs as objects when modeling them. Therefore, moving away from the illustrative examples presented so far, a log-defined data model instance (cf. Definition 4) not only consists of its own logs, but also logs for contained objects, relations, and coordination processes.

The reason for relations having their own set of logs can be found in the permission system employed by object-aware process management (cf. [14]). In a nutshell, object-aware process management employs an RBAC (Role Based Access Control) approach. In RBAC, permissions are not assigned to users directly, but grouped into roles instead, which reduces administrative overhead. In object-aware process management, simple scenarios for granting such roles at runtime, e.g. granting role *Checking Account Manager* with respect to an object instance representing an employee and having an attribute *Department* with value *Account Management,* are supported. More advanced scenarios, such as granting roles based on object relations at runtime, are covered as well. Note that this flexible approach has a drawback, which we identified when developing the ad-hoc change concept for data model instances.

Example 3 (*Side Effects caused by Relation-based Roles*)**.** Consider granting role *Checking Account Manager* not based on an attribute, but instead to any *Employee* object having a relation to a *Customer* (cf. Fig. 4). The role is then granted on a per-relation basis. Regarding the example depicted in Fig. 4, the *Checking Account Manager* role is granted to *Employee1* for *Customer1* and *Customer2* only because he has a direct relation to them. In consequence, ad-hoc changes to any of the involved objects may have side-effects on the *Checking Account Manager* role configured in the relation between the objects *Employee* and *Customer* at design time. Consider the deletion of an attribute as an ad-hoc change to the *Customer* object. As a data model instance level change, this would delete the attribute from all existing *Customer* instances, as well as future ones. However, if the *Checking Account Manager* role grants a permission to write the deleted attribute, runtime errors might occur if the role or the permission is not updated.

The same considerations apply to coordination processes. Though not the main focus of this article, the coordination process is a fundamental element of an object-aware process, as it allows defining and controlling constraints between object instances based on their current states.

Example 4 (*Side Effects caused by Coordination Constraints*)**.** A simple coordination constraint could be that only 4 *Transfer* objects related to the same *Checking Account* may be in state *Approved* at the same time. As is the case with relations, ad-hoc changes to objects may have side-effects on the coordination process instances that continuously monitor all running object instances and enforce their defined constraints. An obvious example would be an ad-hoc change deleting the *Approved* state from the *Transfer* object, as the coordination process relies on this state at runtime.

Initially, the ad-hoc change concept relied on the PHILharmonicFlows modeling tool to ensure that interdependent changes, as illustrated in Examples 3 and 4, are properly reflected in the relations and coordination processes belonging to a data model instance. In the same way, correctness is enforced when designing the initial process model with the modeling tool. However, this approach uses a static analysis of the data model with a set of

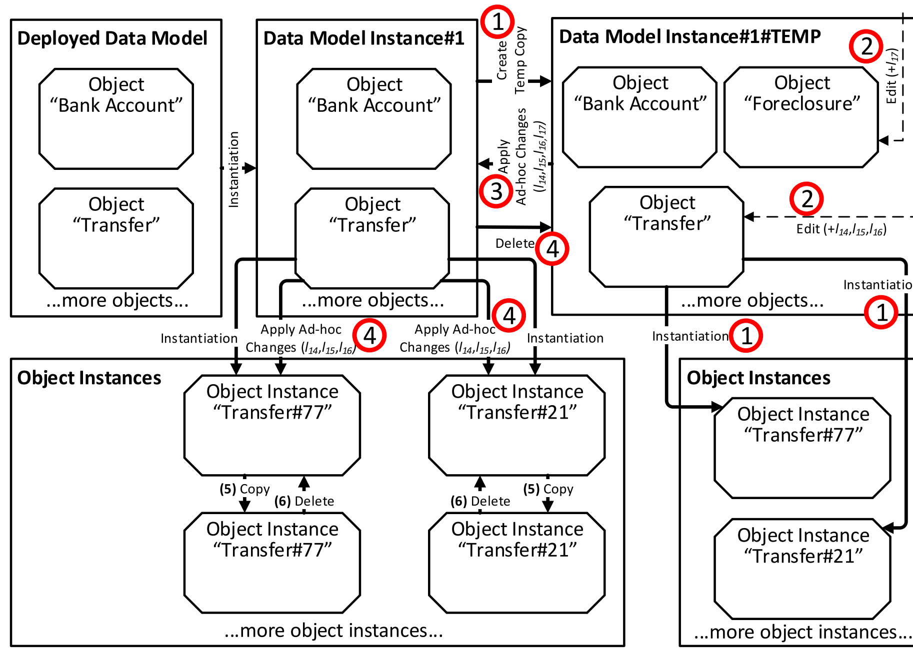


Fig. 10. Creating an ad-hoc changed data model instance.

modeling rules that are hard-coded into the modeling tool. Consequently, it is not adequate for covering possible future scenarios in which ad-hoc changes need to be propagated from one data model to another without any user interaction, e.g. in the context of a schema evolution. To cope with this issue, we developed Algorithm 2 which detects ad-hoc changes that adversely affect other entities of the object-aware process instance in question and fixes them. For the algorithm to work, the definition of a change log entry (cf. Definition 1) has to be extended to include the necessary meta information for log interdependence analysis. The extended definition (cf. Definition 5) includes the category of the modeling action applied by the log entry (*Create*, *Update*, or *Delete*) as well as the affected entity.

Definition 5 (*Extended Change Log Entry*)**.** A tuple L = (S,A,C,E,P,T) is called extended change log entry when:

- S is the source of the log entry, corresponding to any object-aware entity (e.g. object, relation, or coordination process)
- A is a modeling action that may be applied to S
- C is the category of A with $C \in \{Create, Update, Delete\}$
- E is an entity within S (e.g. state, step, attribute, permission) affected by A with $E \in S$
- P is a set of parameters with which A was applied to S
- T is the logical timestamp of the modeling action

Additional meta information has to be generated by all modeling actions to be able to detect interdependencies between log entries. In essence, in any log-defined data model instance, there are multiple sets of extended change log entries (cf. Definition 4).

- One log for the data model itself, containing log entries for high-level actions (e.g. creating new objects or relations).
- One log per object present in the data model, containing log entries that describe the modeling of the object and its lifecycle process (e.g., adding steps or attributes).
- One log per relation present in the data model, containing log entries describing the modeling of the relation (e.g., assigning relation-based roles).
- One log per coordination process present in the data model, containing log entries that describe the modeling of the coordination process (e.g., configuring coordination constraints for interdependent objects).

As an example consider the data model for the PHoodle E-Learning Platform[1] (cf. Section 6.2), one of the real-world data models we use for testing our concepts. This data model consists of 7 different objects, 10 relations, and 1 coordination process. Consequently, in this example, 19 logs with a total of 479 change log entries need to be analyzed for interdependencies in order to prevent inconsistent changes to objects, relations, or coordination processes when conducting ad-hoc changes to the data model instance at runtime.

Considering that an ad-hoc change can be understood as the log delta between the logs before and after an ad-hoc change (cf. Definition 3), the necessary algorithmic work boils down to analyzing whether any of the new log entries, which are part of the ad-hoc change, have interdependencies with any of the existing log entries. Note that, with the metadata available from the extended change logs, this constitutes an inexpensive task from a computational point of view. The basic idea of finding and removing interdependencies between logs is captured in Algorithm 2. In essence, the algorithm tries to find *Delete* log entries and to remove other log entries relying on the deleted entity.

In detail, one must loop over the log entries $\Delta.log$ introduced by the ad-hoc change, and for each log entry $L^{\Delta} \in \Delta.log$ with category $L^{\Delta}.C = Delete$, loop over the logs present in data model instance M, i.e., $M.log$, $M.objs$, $M.rels$, and $M.coords$ (cf. Algorithm 2, Lines 4–5). While looping, one must remove all logs entries $L^M \in M$ with category $L^M.C = Update$ and the entity affected by the deletion, $L^{\Delta}.E$, in their parameter set $L^M.P$ (cf. Lines 6–10). Effectively, this deletes all change log entries that ever altered an existing entity to rely on the now deleted entity. The results of this algorithm are *pruned* log-based objects, relations, and coordination processes, which are guaranteed to not reveal any runtime problems after introducing the ad-hoc changes and rebuilding the data model instance from the logs. Effectively, all dependencies

[1] The download links for the log-based representation of this data model, along with others, can be found in the footnotes of Section 6.4.

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

ARTICLE IN PRESS

Algorithm 2: Fixing log interdependency issues

Require: $\Delta.log, M$ ▷ *ad-hoc change log and log-defined data model instance M*
1: $allLogs[] \leftarrow M.logs \cup M.objs \cup M.rels \cup M.coords$ ▷ *simple looping later on*
2: $affectedEntities[] \leftarrow$ **new** ▷ *set of entities that will have to be rebuilt later*
3: **for all** l^{Δ} in $\Delta.log$ **do**
4: **if** $l^{\Delta}.C = Delete$ **then**
5: **for all** l^{M} in $allLogs$ **do** ▷ *find entries affected by ad-hoc delete*
6: **if** $l^{M}.C = Update$ **then**
7: **for all** p in $l^{M}.P$ **do** ▷ *loop over parameters*
8: **if** $p = l^{\Delta}.E$ **then** ▷ *parameter is entity deleted by* l^{Δ}
9: $affectedEntities[]$.**add**$(l^{M}.S)$ ▷ *log source needs rebuild*
10: $delete(l^{M})$
11: **end if**
12: **end for**
13: **end if**
14: **end for**
15: **end if**
16: **end for**
17: **for all** $entity$ in $affectedEntities[]$ **do** ▷ $entity \in M.objs \cup M.rels \cup M.coords$
18: $entityInstances[] \leftarrow getInstances(entity)$ ▷ *get all instances of entity*
19: **for all** $instance$ in $entityInstances[]$ **do**
20: $delete(instance)$
21: $instance \leftarrow$ **new**
22: **for all** l in $entity.log$ **do** ▷ *affected entries are no longer present*
23: $instance.replayChangeLog(l)$ ▷ *rebuild the instance from logs*
24: **end for**
25: **end for**
26: **end for**

on objects deleted by the log representing the ad-hoc change are deleted in the other logs representing the pre-existing entities in the data model. Algorithm 2 summarizes the entire procedure.

Obviously, the extension to our definition of log entries as well as the development of Algorithm 2 and the other concepts described in this section created significant efforts. However, these concepts do not merely serve the purpose of having a clean and automated way of fixing inconsistent ad-hoc changes on the fly. Instead, the concepts presented in this section are reused to a large extent to improve the performance of the entire ad-hoc change concept. In particular, the additional meta information provided by the extended log definitions can be utilized in a modified version of Algorithm 2 to prune logs in order to reduce the number of steps necessary for creating temporary copies of objects and data models, a concept which is explained in detail in Section 5.1.

5. Performance considerations

In Section 4.3, we examined one class of problems that might arise when performing ad-hoc changes to entities of a data model instance at runtime and not properly adapting other entities to those changes. To cope with this challenge, we developed Algorithm 2, which prunes logs in order to remove log entries that cause dependencies on entities removed by ad-hoc changes. The pruned log is then used to recreate instances of those entities having no dependency on the entity deleted by the ad-hoc change. Note that this works for any log-based entity present in an object-aware data model, including objects and relations.

Detecting log interdependencies is not the only goal of the developed algorithm, which can be extended to enable far more general log pruning. In general, performance is a critical factor in process management systems, an issue that was investigated in related works on activity-centric process engines [20–22]. Note that performance considerations are even more crucial for object-aware processes, as the granularity at which interactions with the process engine occur is much more fine-grained compared to activity-centric engines. Obviously, the performance of the ad-hoc change concept presented in Section 4 relies on the change log entries that are created when modeling an object-aware data model. To be more precise, the speed at which the change operations can be performed scales linearly with the number of log entries created for the data model at design time. In essence, to adhere to Requirement 8 (i.e., ensuring that the performance of the concept is sufficient for applying it in the context of large-scale schema evolutions), the number of log entries should be minimal. Note that there is no way to ensure that process modelers do not complete modeling actions they redo differently later on. This is simply impossible as the creation of a process model constitutes an iterative procedure to some extent, assuming that neither the process modeler nor the requirements for the model are “perfect”.

When analyzing the logs of object-aware process models created by students with the PHILharmonicFlows modeling tool [23], we could show that, on average, every third modeling operation was undone by some means later on.[2] While this might be a symptom of the paradigm shift away from activity-centric to object-aware process modeling, it points out the problem at hand: The performance of the concepts presented in this paper does not depend on the size of the final data model, but on the size of the modeling log. Obviously, this contradicts Requirement 8 as it leads to longer turnaround times for many procedures involved in the core concepts of this article, such as *creating temporary copies of objects* and *rebuilding entities from their log-based representations*. To remedy this, we extended our concept with several performance optimizing techniques revolving around the logs and their usage during *copy* and *replay* operations at runtime. This ensures that our concept is efficient enough for evolving real-world processes with many instances (cf. research question *RQ3*). The following sections present concepts for *log pruning*, *log grouping*, and *log parallelization*.

5.1. Log pruning

Log pruning is indispensable to improve the performance of the ad-hoc change concept. It ensures that log entries which have no effect on the resulting data model are pruned from the log. The general idea for log pruning is the same as for identifying and fixing log interdependencies when applying ad-hoc delete operations (cf. Section 4.3). Furthermore, the idea relies on the extended change log notion from Definition 5. Regarding Algorithm 2, it becomes evident that pruning a log entry from a log prevents the corresponding modeling action the log entry would complete during log replay from being introduced to the model. Using this knowledge, one can optimize a log through pruning.

We identified two cases in which pruning can be used to this end. First, the simple case of pruning *Update* log entries made obsolete by later *Update* log entries, and, second, pruning log entries that are made obsolete by later *Delete* logs. An example of the former is changing the display name of an attribute, and then changing it later for whatever purpose. This would cause two extended change log entries, as shown in Example 5, to be created during modeling. Obviously, the first log entry, i.e., l_{23}, is deprecated by l_{37}.

[2] Links to a selection of these logs can be found in the footnotes in Section 6.4.

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

Example 5 (*Renaming Twice*).

$$l_{23} = \begin{cases} S & \textit{object : Transfer} \\ A & \textit{SetAttributeDisplayName} \\ C & \textit{Update} \\ E & \textit{AmountAttribute} \\ P & [\textit{attribute : AmountAttribute}, \\ & \textit{name : ``Transferred Amount''}] \\ T & 23 \end{cases} \qquad l_{37} = \begin{cases} S & \textit{object : Transfer} \\ A & \textit{SetAttributeDisplayName} \\ C & \textit{Update} \\ E & \textit{AmountAttribute} \\ P & [\textit{attribute : AmountAttribute}, \\ & \textit{name : ``Amount Transferred''}] \\ T & 37 \end{cases}$$

Generally, we can define two pruning rules as follows:

Rule 1 (*Pruning Update Logs*). If multiple log entries exist with category $C = Update$ as well as same modeling action A and affected entity E, only keep the one with the largest timestamp T.

Regarding Example 5, this means that l_{23} is pruned from the log. As all *Update* modeling actions necessary in object-aware processes can be structured in this idempotent way, Rule 1 may apply to anything from label updates to permissions or coordination conditions.

Rule 2 (*Pruning Delete Logs*). If a Delete log entry is found that deletes an entity, prune the Create log entry responsible for the creation of said entity.

The benefits of Rule 2 are manifold: not only the *Create* log entry itself can be pruned from the log, but all subsequent *Update* log entries that modified the created entity as well. Furthermore, the *Delete* log entry itself may be pruned from the log. Finally, with the *Create* log entry not present in the log anymore, any other log entries with interdependencies on the now deleted entity can be pruned as well, in a cascading fashion. Note that such *cascading pruning* might delete a large number of log entries recursively in certain scenarios, leading to massive reductions in log sizes and, therefore, reductions in the number of modeling actions needed to rebuild the entities contained in the log. This has been especially true for some of our real-world applications of the pruning algorithm, as *Delete* pruning can recursively prune entire objects that were created in many steps but then deleted later on (cf. Section 6.4). Example 6 shows a simple log with a sequence of entries that can be pruned by the log pruning algorithm.

Example 6 (*Object Deletion*).

$$l_{57} = \begin{cases} S & \textit{object : Transfer} \\ A & \textit{AddAttribute} \\ C & \textit{Create} \\ E & \textit{CommentAttribute} \\ P & [\textit{name : ``Comment''}] \\ T & 57 \end{cases} \qquad l_{58} = \begin{cases} S & \textit{object : Transfer} \\ A & \textit{SetAttributeDisplayName} \\ C & \textit{Update} \\ E & \textit{CommentAttribute} \\ P & [\textit{attribute : CommentAttribute}, \\ & \textit{name : ``Manager Comment''}] \\ T & 58 \end{cases}$$

$$l_{59} = \begin{cases} S & \textit{relation : CustomerToEmployee} \\ A & \textit{AddAttributeWritePermission} \\ C & \textit{Create} \\ E & \textit{CommentWrite} \\ P & [\textit{attribute : CommentAttribute}, \\ & \textit{state : Decision Pending}] \\ T & 59 \end{cases} \qquad l_{60} = \begin{cases} S & \textit{object : Transfer} \\ A & \textit{AddStep} \\ C & \textit{Create} \\ E & \textit{CommentStep} \\ P & [\textit{attribute : CommentAttribute}, \\ & \textit{state : Decision Pending}] \\ T & 60 \end{cases}$$

$$l_{61} = \begin{cases} S & \textit{relation : CustomerToEmployee} \\ A & \textit{AddPermissionToRole} \\ C & \textit{Create} \\ E & \textit{AccountManagerRole} \\ P & [\textit{permission : CommentWrite}, \\ & \textit{role : AccountManager}] \\ T & 61 \end{cases} \qquad l_{82} = \begin{cases} S & \textit{object : Transfer} \\ A & \textit{DeleteAttribute} \\ C & \textit{Delete} \\ E & \textit{CommentAttribute} \\ P & [\textit{attribute : CommentAttribute}] \\ T & 82 \end{cases}$$

Algorithm 3: Complete log pruning algorithm

```
Require: logEntries[]   ▷ logs to prune, e.g. from data model and ad-hoc changes
 1: for all logEntry in logEntries[] do
 2:   logEntriesToPrune[] ← new
 3:   if logEntry.C = Delete then   ▷ prune logs dependent on this delete
 4:     for all logEntry2 in logEntries[] do   ▷ double for loop
 5:       if logEntry.E = logEntry2.E or logEntry.E ∈ logEntry2.P then
 6:         logEntriesToPrune[].add(logEntry2)
 7:       end if
 8:     end for
 9:   else if logEntry.C = Update then   ▷ prune logs overridden by this update
10:     for all logEntry2 in logEntries[] do   ▷ double for loop
11:       if logEntry.E = logEntry2.E and logEntry.A = logEntry2.A then
12:         if logEntry.T > logEntry2.T then   ▷ same E and A, lower T
13:           logEntriesToPrune[].add(logEntry2)
14:         end if
15:       end if
16:     end for
17:   end if
18:   for all logEntryToPrune in logEntriesToPrune[] do
19:     if logEntryToPrune.C = Create then   ▷ check for pruned creates
20:       deleteLogEntry ← new
21:       deleteLogEntry.C ← Delete
22:       deleteLogEntry.E ← logEntryToPrune.E
23:       logEntries[].add(deleteLogEntry)   ▷ add "fake" delete → cascade
24:     end if
25:   end for
26:   logEntries[] ← logEntries[] − logEntriesToPrune[]
27: end for
28: return logEntries[]
```

Basically, with the meta information provided by the extended log entries shown in Example 6, the pruning algorithm identifies l_{82} as a *Delete* log entry. Furthermore, in $l_{82}.E$, the *Comment* attribute is identified as the entity affected by the deletion. By searching through all other log entries present in the data model instance *M*, i.e., *M.log*, *M.objs*, *M.rels*, and *M.coords*, and pruning all log entries with $l.E = CommentAttribute$ or $CommentAttribute \in l.P$, several other log entries (i.e., l_{57}, l_{58}, l_{59}, and l_{60}) may be pruned from the logs as well.

To ensure that there are no more log entries left that depend on any of the pruned log entries, we have to repeat the pruning for any pruned *Create* logs in a cascading fashion. In Example 6, pruning l_{82} causes l_{57} to be pruned, which, in turn, causes l_{58}, l_{59}, and l_{60} to be also pruned, as these log entries have the *Comment* attribute in their parameter list. Note that l_{61} is not pruned, as it has no "direct" interdependence with l_{82}, neither via $l_{61}.E$ nor $l_{61}.P$. To ensure pruning of l_{61}, it becomes necessary to re-apply the pruning algorithm as if a *Delete* log entry for the *Comment Write* permission (l_{59}) exists as well. If we insert a *Delete* log entry for each pruned *Create* log entry, re-applying the pruning algorithm will prune all log entries directly dependent on previously pruned log entries. Applying this logic recursively ensures that all directly or indirectly dependent log entries are pruned. Algorithm 3 summarizes the pruning of *Update* and *Delete* logs entries.

5.2. Log grouping

Another large impact on performance, which we identified when testing the ad-hoc change concept, is the way the log

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

replay itself is conducted. As stated, we use logical timestamps to ensure that log entries are replayed in the exact order the process modeler created them at design time. While in certain situations this is required due to log interdependencies, it is also possible to modify Algorithm 2 to find groups of log entries that are *not* dependent on each other.

Example 7 (*Groupable Logs*)**.** Consider two groups of log entries, G1 and G2, that are created at design time by modeling, for instance, two separate states and their steps. If the log entries in G1 and G2 have no interdependencies, it does not matter which group is replayed first. Theoretically, G1 and G2 could even be replayed simultaneously. Obviously, the log entries *within* a group still have to be executed in sequence.

A concrete case in which logs are groupable as presented in Example 7 is given by the log entries from Example 6. The six log entries show interdependencies we can analyze with Algorithm 2. From the result we can derive the following groups of log entries: $G1 = \langle l_{57}, l_{58}, l_{60} \rangle$, $G2 = \langle l_{59}, l_{61} \rangle$, and $G3 = \langle l_{82} \rangle$. In essence, G1 creates an attribute, sets its name, and creates a step for the attribute in the object lifecycle process. G2 then creates a permission for the attribute and assigns it to a role. Finally, G3 deletes the attribute that, when applying the pruning algorithm, would delete all the other logs. Clearly, the order of the groups cannot be changed, as G2 and G3 rely on the attribute created by the log entries in G1. However, through this grouping and re-ordering of the individual log entries, the groups may be replayed en-bloc to the entities they affect. In terms of the proof-of-concept implementation of this concept, we utilize the fact that all log entries in a group have the same source and, therefore, may be replayed to a copy of that source in one single call.

Regarding our current implementation of object-aware process management, PHILharmonicFlows, all modeling and log replay actions are completed by making calls to the various microservices that represent the entities present in a data model (e.g. objects or relations). Grouping the log entries before replaying them to the microservices reduces the communication overhead of replaying logs considerably (cf. Section 6.4). As log grouping only offers such a large benefit due to the nature of our implementation, which is completely distributed, we chose to omit the grouping algorithm from this article.

5.3. Log parallelization

To maximize the effects of the log pruning and grouping concepts (cf. Sections 5.1 and 5.2) log parallelization was implemented. This critical performance optimization allows for the log groups to be replayed in parallel. Clearly, at design time, the modeling actions were completed step-by-step by a process modeler. However the modeling actions can be re-organized in an optimal way through pruning and grouping, allowing for the parallelized replay of independent log groups. To be more precise, a log-based data model with pruned log sets grouped into *M.log*, *M.objs*, *M.rels*, and *M.coords* must be created using the pruning and grouping concepts. The maximum possible parallelization can then be achieved by first replaying the *M.log* sequence, which invokes the create operations for all the objects and relations. Once these exist, all log sequence groups for objects, relations, and coordination processes, as described in *M.objs*, *M.rels*, and *M.coords,* may be replayed in parallel.

Again, while this is a vast improvement over replaying the logs sequentially, one by one, we recognize that the benefits of parallelized log replay are mostly due to the distributed nature of our engine (cf. Fig. 12). Nonetheless, this allows our engine and, by extension, our concept, to support ad-hoc changes to large numbers of object instances in parallel (cf. research question *RQ3*).

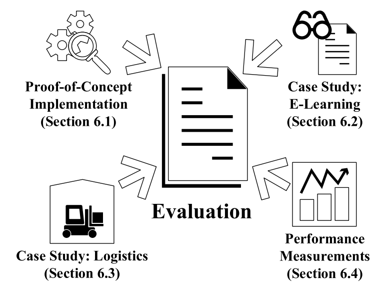


Fig. 11. Evaluation structure.

6. Evaluation

This section thoroughly evaluates the concepts for ad-hoc changes to object aware processes (i.e., objects and data models) presented in Section 4. Furthermore, we provide an extensive evaluation of the additional techniques we developed (cf. Section 5) to improve the overall performance of conducting ad-hoc changes at runtime. The goal of these evaluations, which provide a major addition to our previous work [11] is to show that ad-hoc changes to object-aware processes are not only feasible according to the research questions presented in this article (cf. Section 1.1), but also mature enough for being implemented and utilized in various scenarios.

The evaluation of the concepts is split into four sections (cf. Fig. 11). Section 6.1 presents our proof-of-concept implementation, PHILharmonicFlows, including a technical overview of the process engine, a concrete example of an ad-hoc change in our tooling, and a discussion on how the implementation meets the requirements set forth in Section 3. In Section 6.2 we present a case study in which we employed ad-hoc changes to the data model of an e-learning platform. Section 6.3, in turn, deals with a case study involving both human resources and service robots. In this study we investigated the need for ad-hoc changes and used them to demonstrate two solutions for flexible intralogistics. Finally, Section 6.4 presents a single-case mechanism experiment evaluating the performance of the concept to try and quantify the fulfillment of research question *RQ3* and Requirement 8 in particular. The goals of this evaluation are to

- show examples of scenarios in which ad-hoc changes can be introduced to increase process flexibility in ways not otherwise possible without causing disruptions to process flow (cf. research question *RQ1*).
- show that the proof-of-concept implementation, including underlying concepts, are mature and flexible enough to allow for their usage in a real-world scenario (cf. research question *RQ2*).
- show that the concepts, with the additions presented in Section 5, are not only usable, but also scalable for utilization in larger scenarios. (cf. research question *RQ3*)

6.1. Proof-of-concept implementation

All concepts presented in this article were implemented in the PHILharmonicFlows proof-of-concept prototype, which we are currently utilizing and evaluating in a number of real-world scenarios, two of which are discussed in more detail in Sections 6.2 and 6.3. Section 6.1.1 presents a technical overview on how the concept for ad-hoc changes is realized in our implementation. Section 6.1.2 presents an example of the steps necessary for a user to introduce a typical ad-hoc change to an object instance in the provided tools. Finally, Section 6.1.3 discusses the requirements

ARTICLE IN PRESS

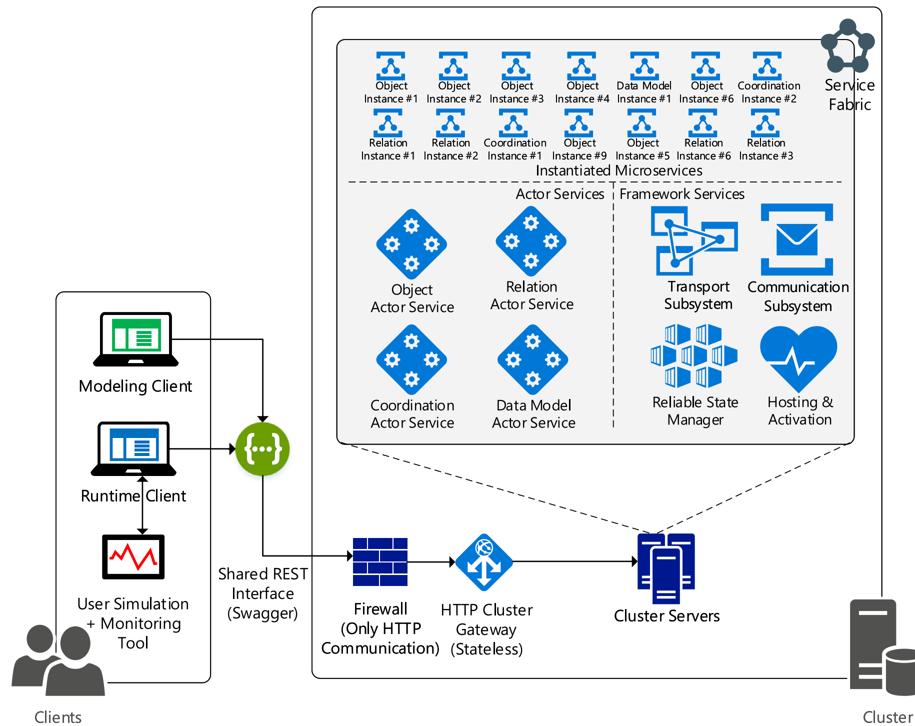


Fig. 12. PHILharmonicFlows implementation architecture.

for ad-hoc changes set forth in this article and shows how they are supported by the implementation. This evaluation demonstrates the integration of the concept into a real process engine implementation and shows that it is usable through a simple graphical user interface.

6.1.1. Implementation overview

The various conceptual elements of object-aware processes, i.e., objects, relations, and coordination processes, are implemented as microservices according to the *actor concurrency model* [24], turning PHILharmonicFlows into a distributed process management system. For each object instance, relation instance, and coordination process instance one microservice instance is present at runtime. Each microservice only holds the data representing the attributes of its object instance. Furthermore, the microservice only executes the lifecycle of the object instance it is assigned to. The only information visible outside the individual microservices is the current "state" of the object, which is used by the microservice representing the coordination process to properly coordinate the object instances' interactions with each other. As previously stated, the implementation employs the *actor concurrency model*, which enforces that only one thread can be active in any microservice at a time. This keeps the issues, which would usually arise from the coordination of this large amount of concurrently executing lifecycle processes, minimal. Furthermore, representing each conceptual element as a microservice allows for an implementation close to the conceptual ideas of object-aware process management.

To support our microservice architecture, we chose the open source Service Fabric[3] framework. Note that many of the concepts presented in Section 4 are implemented almost exactly as presented due to the utilization of the microservice architecture. This tight coupling of concepts and implementation helps us in verifying that the presented algorithms and techniques function well when integrating them into a process engine. In particular, the distributed architecture plays a major role in ensuring that ad-hoc changes scale well (cf. *RQ3*). This is due to the fact that the operations of an ad-hoc change, such as creating temporary copies and re-executing lifecycles, can be performed in parallel without a bottleneck. An overview on the implementation architecture of the PHILharmonicFlows process engine and tooling is given in Fig. 12.

The example from Section 4.2 (cf. Fig. 10), an ad-hoc change to the data model instance *DataModelInstance*#1, is a good illustration of the high overlap of concept and implementation. Clearly, the concept contains many steps in which data, especially log entries, are transferred between object instances. This communication between object instances during ad-hoc changes takes place exactly as described between their respective microservice instances in the engine and can even be monitored in real-time through our monitoring tool.

To be more precise, in PHILharmonicFlows (cf. Fig. 12), an ad-hoc change, such as the one shown in Fig. 10, is triggered by the PHILharmonicFlows modeling tool. The modeling tool can use the provided REST interface to communicate with the cluster hosting the microservices and request the creation of an ad-hoc editable copy of data model *DataModelInstance*#1. Service Fabric routes the request to the microservice instance representing *DataModelInstance*#1, which, in turn, contacts the *Data Model Actor Service* (cf. Fig. 12), causing the instantiation of a new microservice, *DataModelInstance*#1#*TEMP*. After completing this step, all log entries present in *DataModelInstance*#1 are pruned (cf. Section 5.1), grouped (cf. Section 5.2), and transferred to the microservice representing the data model instance copy. Here, they are replayed in parallel (cf. Section 5.3), causing *DataModelInstance*#1#*TEMP* to become an exact copy of *DataModelInstance*#1.

The modeling tool can now be used to edit the temporary copy of the data model. Each modeling operation causes the modeling tool to send a change log entry to the microservice of *DataModelInstance*#1#*TEMP*. When the user chooses to trigger the actual ad-hoc change, all newly created log entries present

[3] https://github.com/Microsoft/service-fabric-services-and-actors-dotnet.

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

ARTICLE IN PRESS

in *DataModelInstance#1#TEMP* are pruned, grouped, transferred to *DataModelInstance#1*, and replayed there. If any of the log entries contained in the ad-hoc change do not affect the data model itself, but instead one of the contained objects, they are forwarded to all microservices representing instances of that object. Effectively, this ensures that ad-hoc changes to objects conducted on the data model level are propagated to all instances of those objects. Finally, if any object instances were affected by the ad-hoc change, their corresponding microservices trigger re-execution of the contained lifecycle processes, thereby ensuring process consistency across the entire data model instance. Obviously, this entire procedure, as it is currently implemented in PHILharmonicFlows, mirrors the concept presented in Section 4.2.

6.1.2. Ad-hoc changes from a user perspective

This section gives a short overview of the user perspective on an ad-hoc change to a data model from the human resources domain that supports the process of reviewing job applications. Example 8 shows a typical ad-hoc change that might be introduced to an object. The execution monitoring view of one of the object instances that would be affected by this change is shown in Fig. 13.

Example 8 (*Ad-Hoc Insertion*)**.** After applicants have submitted job applications to a company they must go through a reviewing procedure. Assume that it is decided to additionally give managers the opportunity to comment on each job application review before a detailed applicant assessment is conducted. As it is unclear whether this change to the data model will improve the review quality significantly or just increase turnaround times, the change is not incorporated into the base data model, but only into already existing *Review* objects as an ad-hoc change.

To realize the ad-hoc change described in Example 8, one must insert a new state in the *Review* object, e.g. *Manager Review,* as well as an attribute and a corresponding step *Comment* in this new state, and connect transitions to the predecessor and successor states. To facilitate this, the proof-of-concept prototype offers an *Edit Model* button in the runtime view (cf. Fig. 13), which creates an ad-hoc editable copy of the object instance according to Algorithm 1. Furthermore, it opens the PHILharmonicFlows modeling tool and displays the temporary copy. Hence, the object may be manipulated and changed in exactly the same way as if it were the initial modeling for a new object at design time. As all modeling operations are permitted and every modeling operation creates a change log entry that is applied to the live object instance when changes are propagated, all modeling operations are valid ad-hoc changes. In essence, **any** aspect of the process model may be ad-hoc changed, ensuring that Requirement 7 holds and answering *RQ2*. Note that our goal is to demonstrate technical feasibility of the approach through the implementation. Improving usability and comprehension for end user interactions is outside the scope of this article.

Fig. 14 shows the modeling tool after the new state, step, attribute, and transitions have been added to the lifecycle model of the *Review*. While the user is editing this temporary copy of the object, ad-hoc change log entries are created for each modeling action the user completes. Once all changes are modeled, the user may propagate the changes introduced to the *Review* object to one or more of the currently running *Review* object instances, depending on the respective change scenario. Before the actual propagation is done, the log entries are pruned and grouped (cf. Section 5). This ensures that the ad-hoc changes actually propagated to the running object instances become as minimal as possible. Note that even in this small example it might occur that the user unintentionally added a wrong transition and deleted it afterwards, or renamed the new state twice. With log pruning, this "extra" work is ignored when replaying the ad-hoc change log to the object instances. The result of the ad-hoc change to the running instance from Fig. 13 can be seen in Fig. 15.

6.1.3. Discussion

This section offers a discussion on how well the proof-of-concept prototype covers the requirements set forth in Section 3 and whether the research questions posed in Section 1.1 are answered. As Section 6.1.2 has just shown how it is possible for users to employ ad-hoc changes to an object instance, we begin our discussion with a quick overview over which requirements had to be fulfilled by the engine to enable support of the example from Section 6.1.2. This overview is intended to help understanding which parts of the implementation assist in fulfilling each of the requirements and is followed by a more general examination of the requirements. The ad-hoc change introduced in Example 8

- *was atomic and applied in one transaction (Requirement 1).*
  This is enabled on the conceptual side through the creation of temporary object instance copies which are then hot-swapped into the live data model instance once the user has completed the ad-hoc changes (cf. Algorithm 1). On the implementation side, the use of individual microservices for each object instance, which can exist independently of others, helps ensure that an ad-hoc change to one object instance does not affect other object instances, which is an integral part of answering research question *RQ1*.
- *used a selection of all available modeling operations (Requirement 7).*
  As modeling operations are translated to log entries by the implementation, any modeling operation supported by the modeling tool can be applied as an ad-hoc change to an object instance or data model instance as well. In essence, using the logs generated by modeling operations in this way instead of defining a set of specific valid "change operations", like many related approaches do answers research question *RQ2*.
- *was ensured to be correct (Requirements 2 and 4).*
  As a static model verification of a changed process model may be performed by the modeling tool before changes are propagated to a live object instance, the modeling tool can prevent syntactical or semantic errors in the same way as it is possible for a newly created model.
- *was ensured to be minimal (Requirement 8).*
  The log pruning algorithm (cf. Algorithm 3) ensures that creating the temporary copies of data model instances, which can be edited in the modeling tool, is faster. Furthermore, it can also prune mistakes made when modeling the actual ad-hoc changes before they are propagated to a potentially large number of running object instances. Without the pruning algorithm in place, a change which is revised several times before it is actually propagated would cause the process engine to perform unnecessary steps as there would be a large number of unnecessary log entries, which is why pruning the logs is an essential part of answering *RQ3*.
- *was runtime-consistent and coordinatable by the coordination process (Requirements 3 and 6).*
  This is ensured by the re-execution, which we enforce as part of the actual ad-hoc change propagation to live object instances (cf. Algorithm 1). As shown in Example 8, affected object instances require the new mandatory attribute *Comment*. The re-execution of each affected object instance halts at the exact point at which the new attribute is required. This even applies to instances, such as the one shown in Fig. 15, which have already executed beyond the point where the corresponding step was inserted. While this can

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

ARTICLE IN PRESS

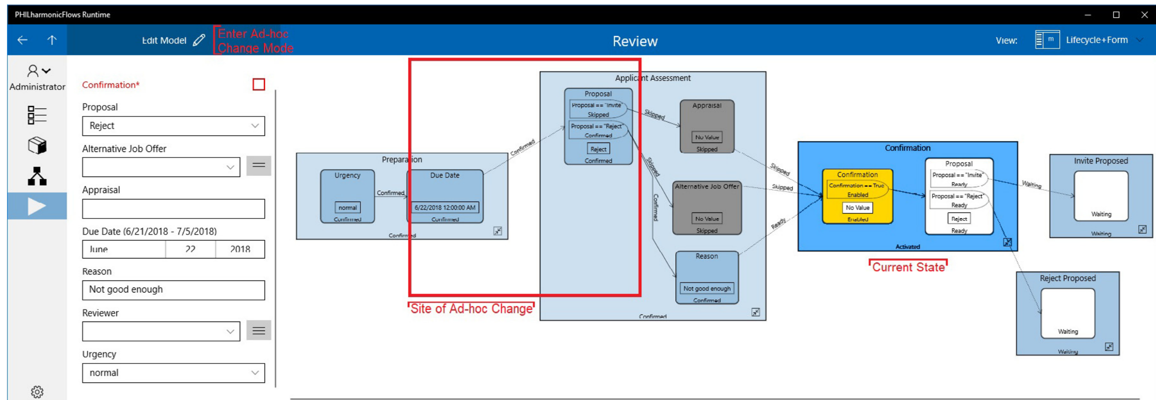


Fig. 13. Lifecycle process of a review object without ad-hoc changes (the box indicates where the ad-hoc changes from Fig. 14 will occur).

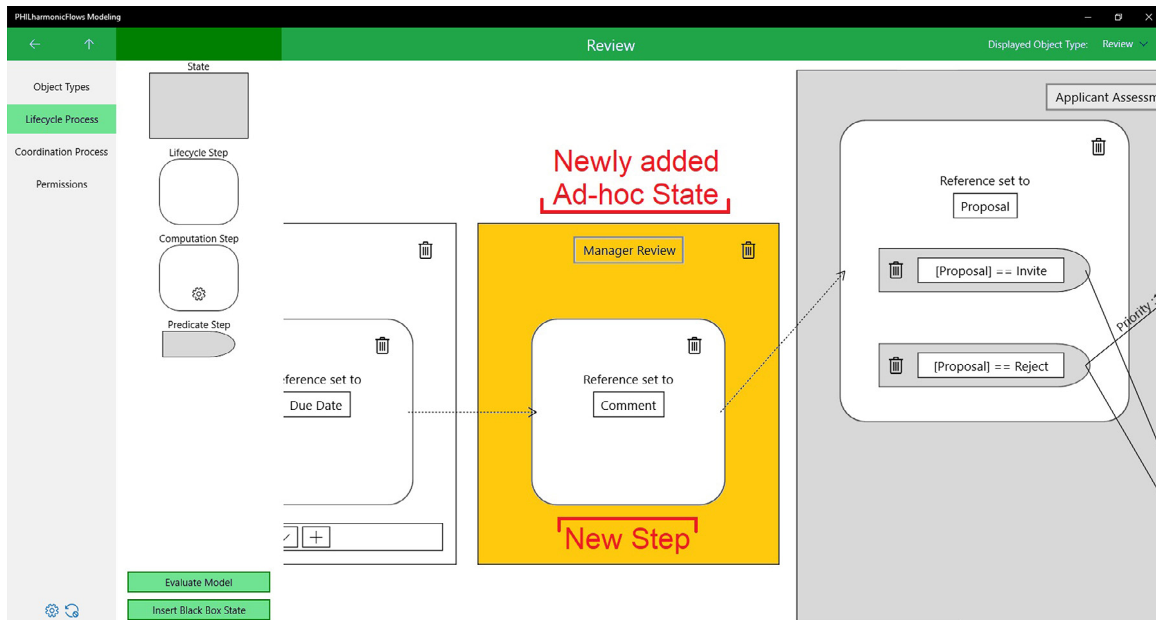


Fig. 14. Modeling tool with ad-hoc changed lifecycle (cf. Fig. 13).

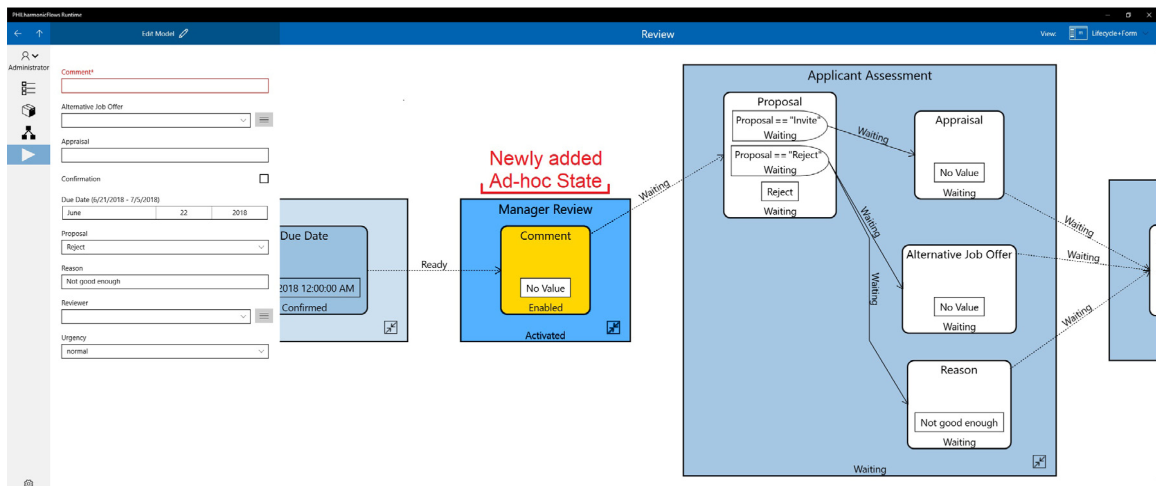


Fig. 15. Review object with ad-hoc added state *Manager Review*.

potentially cause an object instance to be in a different state than before the ad-hoc change, it ensures that the object remains consistent with the changed process model, and, by extension, with any coordination processes.

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

ARTICLE IN PRESS

- *offered the possibility of continued execution of all other object instances (Requirement 5).*
  As the user was editing the model of a copy of the *Review* object, existing instances of the *Review* object could continue execution normally until the moment in which the changes were propagated. Clearly, this is another large contribution to answering *RQ1* by minimizing disruptions, as the alternatives, such as blocking all review object instances while the user is editing the underlying process model, would be disruptive to the process flow. Furthermore, through the usage of the *actor concurrency model* (cf. Section 6.1.1), the actual time span in which the changes are propagated to the live object instances is ensured to lock out all users from the object instances, as only one thread may be active in a microservice at any given time. However, the implementation handles this gracefully, as interactions with individual microservices are queued and continue as normal afterwards. Therefore, from a user perspective, opening the generated form for an object instance, which is currently being hot-swapped for an ad-hoc changed version, just looks like a minor user interface lag as their request is queued to execute after the changes are propagated.

Of course, a single example is not sufficient to confirm that all requirements are met by the concept and its implementation. Thus, we must examine some of them in greater detail.

In Section 4, we have already shown how the developed solution meets Requirements 1–4, i.e., change atomicity, correctness, runtime consistency, and model consistency.

As discussed in Section 6.1.1, we chose a fully distributed microservice-based implementation architecture for the PHILharmonicFlows process engine. This allows us to cover Requirement 5, i.e., we can ensure that the migration of multiple object instances in response to an ad-hoc change of a data model instance can be accomplished in parallel and independently of the execution of other objects. In this context, log grouping (cf. Section 5.2) and log parallelization (cf. Section 5.3) allows processing copy and log replay operations concurrently. As these operations form the basis of the concept for ad-hoc changes, the ability to execute them on separate microservices provides support for Requirement 5, i.e., concurrency.

As an ad-hoc change to a data model instance may cause changes to a large number of object instances, this also causes their re-execution based on the updated lifecycle processes. As set out by Requirement 6, object instances need to be coordinated properly when introducing an ad-hoc change. Thus, it is essential to ensure that changed object instances are coordinated by their respective coordination processes. If this was not the case, they might leave the data model instance itself in an inconsistent state. When the lifecycle processes of the affected object instances are re-executed in the course of an ad-hoc change, they continuously change their states. As a response, the coordination process instance they are assigned to determines the overall execution state of the data model instance, thereby ensuring that Requirement 6 is met. While this necessary communication impacts performance of the developed concept, it is largely due to the distributed architecture as the microservices representing the object instances affected by the ad-hoc change have to communicate with the microservices representing the coordination processes.

Requirement 7, i.e., having the complete set of modeling operations available for use in ad-hoc changes, is supported in the implementation through the use of the same PHILharmonicFlows modeling tool, which is also employed when creating new data models at design time. It creates the change log entries, on which the ad-hoc change concept relies for any modeling action the user takes, as long as it is syntactically correct.

Finally, concerning Requirement 8, i.e., ensuring sufficient algorithmic performance independently of the actual implementation and hardware, the log pruning algorithm ensures that no unnecessary logs are replayed when conducting an ad-hoc change. However, in our concrete implementation, we also utilize the log grouping and parallelization concepts (cf. Section 5) in combination with the microservice architecture to ensure scalability for large deployments.

6.2. Case study: E-learning-platform

A sophisticated scenario we evaluated in practice was the inclusion of ad-hoc changes into the data model of an e-learning platform called PHoodle,[4] which we implemented using a PHILharmonicFlows data model. This evaluation shows that ad-hoc changes have practical applications and can be utilized to rectify modeling errors at runtime, even in production scenarios.

PHoodle is currently being used by us in one of our lectures to research how well the complexity of an object-aware process management system can be hidden from end-users and what problems arise with completely generic software in real-world environments. Furthermore, as the platform is used instead of the regular Moodle e-learning platform the university provides, it will help us evaluate whether an entirely generic and model-based approach to creating software, such as PHILharmonicFlows, is viable from an end-user perspective. The platform is entirely powered by an object-aware data model, a download link to which can be found in Section 6.4. The online front-end is fully generic in respect to the data model instances it allows interaction with and is entirely end-user oriented. To be more precise, it contains exactly one line of code specific to the e-learning platform functionality: the entity ID of the object-aware data model to connect to on the PHILharmonicFlows server. Fig. 16 shows a typical PHoodle menu as it is displayed to a lecture supervisor after logging in and selecting a lecture. Note that the entire user interface is generated from the information available in a single object instance, and is completely tailored to the user viewing the object instance at runtime.

While the web user interface does not support the model-based ad-hoc changes discussed in Section 6.1 directly, ad-hoc changes are supported by connecting our modeling tool to the same data model instance on the PHILharmonicFlows server that powers PHoodle at runtime. Indeed, we were forced to make use of this possibility during a pilot study. Due to an incorrectly modeled permission, students with the tutor role in one lecture could edit the properties of exercise sheets for other lectures in which they were only attendees. We successfully utilized the ad-hoc change mechanism to correct the error while the system was running, keeping all user data intact. The correctly functioning form from the perspective of a student is shown in Fig. 17.

The screenshots from Figs. 16 and 17 were made on the live system, but on a demonstration data set to remain GDPR compliant. Note that such an ad-hoc change is by far more difficult to realize, if supported at all, in contemporary enterprise resource planning (ERP) systems due to the multiple application layers affected. The generic approach utilized by PHILharmonicFlows ensures that changes to a data model are reflected in the persistence layer, logic layer, and presentation layer of the modeled application immediately.

Finally, the deployment of the PHoodle data model in the course of a lecture has allowed us to gather performance data from ad-hoc changes to a real-world data model instance. The instance in question ran for a full semester (exactly 100 days) and consists of 136 users (students, supervisors and tutors). It

4 https://phoodle.dbis.info.

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

ARTICLE IN PRESS

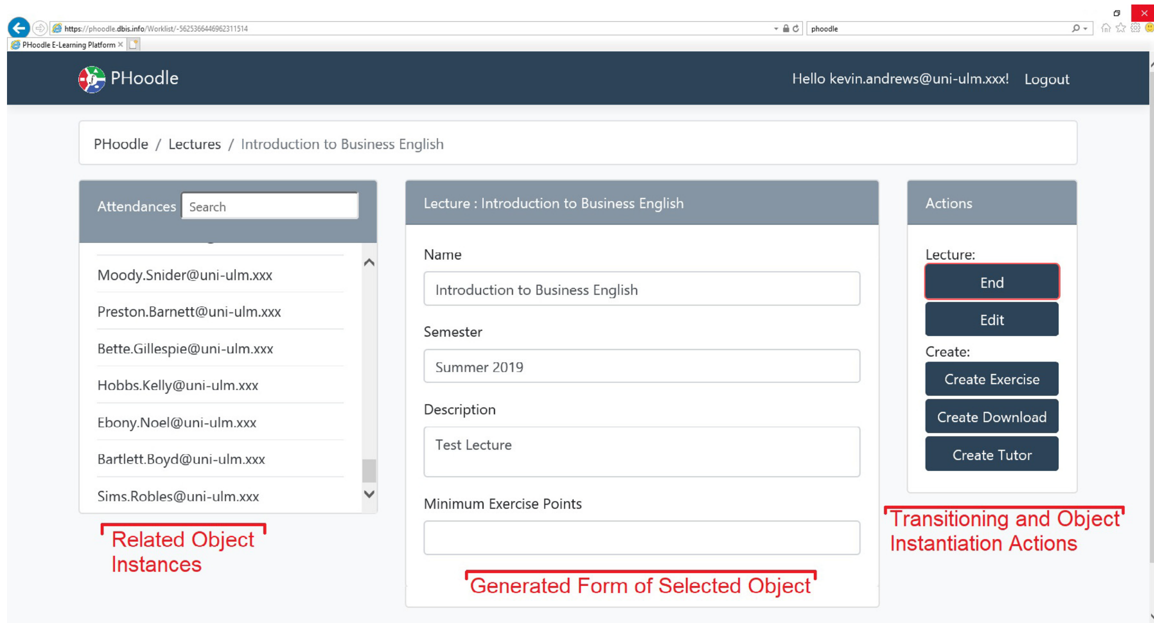


Fig. 16. PHoodle — *Supervisor* overview.

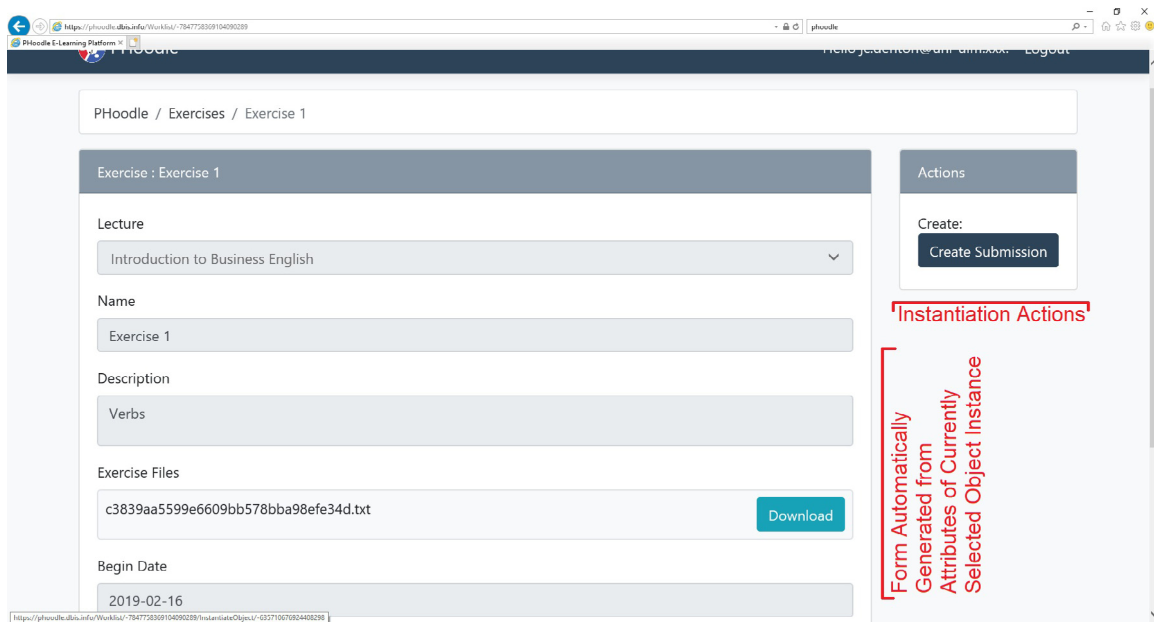


Fig. 17. PHoodle — *Exercise* object.

logged over 40,000 user interactions and currently consists of 2898 microservices representing 848 object instances, 1542 relation instances, 5 coordination processes, and 503 uploaded files. A worst-case scenario ad-hoc change (i.e. the change affects all object instances) to this large data model instance takes, on average, 4.916 ms, of which only 157 ms are utilized for re-executing all affected 848 object instances.

6.3. Case study: service robot logistics

The ad-hoc change concept presented in this article has been further evaluated in multiple intralogistics scenarios in the course of a large project.[5] This evaluation shows that ad-hoc changes can be applied to highly automated processes involving complex external systems, such as service robots. In the scenarios, service robots and humans are directed by PHILharmonicFlows while conducting various types of commissioning processes. For the scenarios to function with real robots, we created a worklist-like interface for robots to interact with PHILharmonicFlows. Specifically, the forms generated for object instances were made available for machine consumption on a specially designed socket interface. Introducing this abstraction layer enables us to treat compatible robots as users in PHILharmonicFlows. This allows us to present them with worklists, which they use to commission or transport products (cf. Fig. 18). To support the scenarios, we created a data model consisting of *Employee*, *Transport Robot*, *Commission Robot*, *Transport*, *Commission*, and *Product* (cf. Fig. 19). Basically, an instance of the *Product* object contains all the meta

5 http://zafh-intralogistik.de.

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

ARTICLE IN PRESS

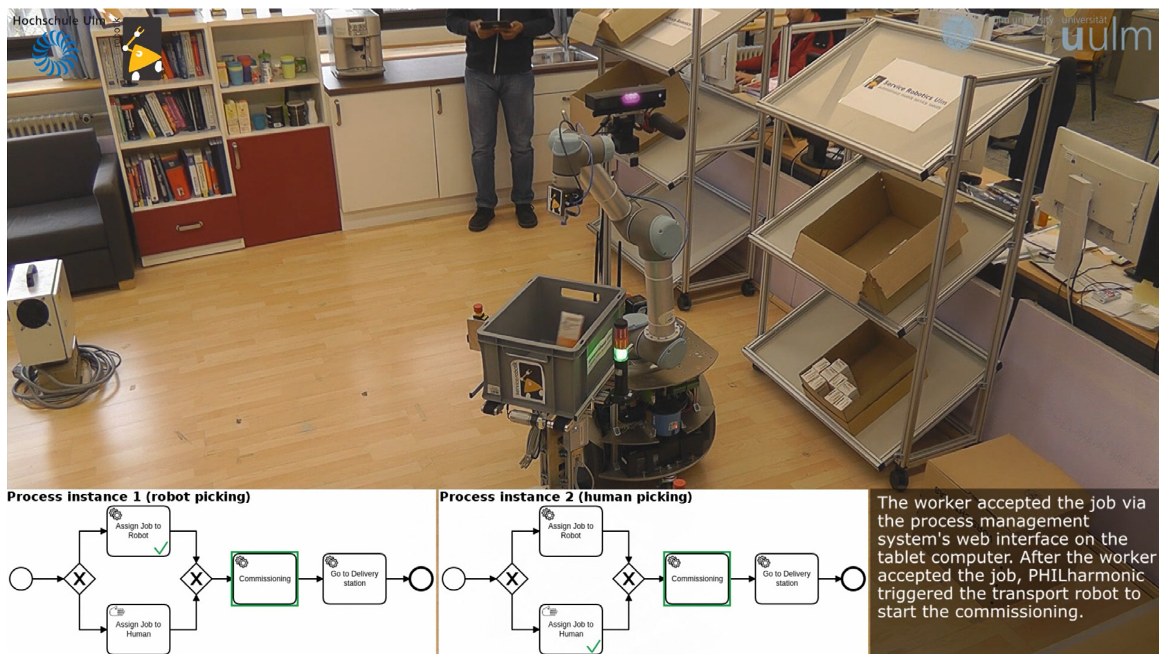


Fig. 18. Screenshot of a video demonstrating collaboration between robots and humans (BPMN based views on the involved PHILharmonicFlows objects overlaid for simplicity).

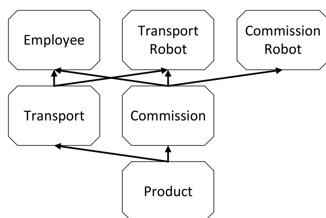


Fig. 19. Intralogistics data model.

information, such as description and shelve location, which a robot or human needs to commission a product from a shelve. The three kinds of workers, i.e., humans, transport robots, and commission robots, are each represented by their own object. This becomes necessary as the commission robots have picking arms, and entirely different functions than transport robots. Finally, the *Transport* and *Commission* objects represent a transporting or commissioning job. For example, at runtime, a *Transport* object related to a *Product* object and an *Employee* object creates a new worklist entry for a human worker. Conversely, if the *Transport* object is related to a *Product* and a *Transport Robot* instead, a worklist entry is created for one of the transport robots. In either case, the created worklist entry contains the information needed to carry the real-world product described in the *Product* object instance to wherever the *Transport* object instance dictates. This in itself allows for flexible execution of simple intralogistics based processes. For example, the decision whether the transporting job is completed by a human or a robot can depend on the attributes of the *Product* object instance attached to the *Transport* object instance in question.

The following video showcases some of the scenarios evaluated:

https://www.youtube.com/watch?v=UtO1Dc1B3Cs

In particular, Scenario 2 (at 3:25 in the above video) showcases an ad-hoc change in which the *Commission* object has a new state with some steps added. The change constitutes an additional process step for the human employee in which he is instructed to add an extra item, a promotional flyer, to the already commissioned box after one of the robots delivers it to him. The benefits of being able to incorporate ad-hoc changes such as this to existing processes, without having to reconfigure robots or update task lists for human workers, are immense. As the changes are model-based, and use the same tooling for humans and robots, turnaround times for implementing changes is low compared to existing solutions.

6.4. Performance measurements

As known from literature, sufficient performance and scalability are crucial success factors for any process-aware information system. Therefore, we have considered performance aspects throughout the entire development of the concepts presented in this article. The ad-hoc change concept (cf. Section 4) relies on two core aspects: *log replay* and *re-execution of lifecycle processes*. On one hand, log replay is used to create temporary copies of entities, such as objects and data models, and to propagate ad-hoc changes back from the temporary copies to their live counterparts. On the other, re-execution ensures that lifecycle processes are always consistent and coordinated properly in the context of a data model instance. Section 6.4.1 examines our previous work on the performance of lifecycle execution, whereas Section 6.4.2 provides measurements on the performance of the copy operation conducted using log replay.

6.4.1. Previous work on scalability and performance measurement

As both performance and scalability are crucial aspects, we developed the PHILharmonicFlows process engine from the ground up to be highly scalable and fully distributed through the use of microservices. We published implementation details as well as the results of our single-case mechanism experiments on scalability and performance in [25].

A short summary of the results presented in [25] is given to demonstrate why the presented measurements (cf. Section 6.4.2) are focused on the copy operation. In essence, we conducted experiments on different data model instances with different

ARTICLE IN PRESS

Table 1
Data model statistics.

Data model name		# entities	# Entries unpruned	#Entries pruned	Achieved reduction
Recruitment	Totals	32	3880	1550	60%
	Data Model	1	109	31	72%
	Objects	11	3413	1372	60%
	Relations	15	37	19	49%
	Coordinations	5	321	128	60%
Employee Self-Service	Totals	45	6573	3066	53%
	Data Model	1	50	44	12%
	Objects	21	5992	2642	56%
	Relations	21	122	70	43%
	Coordinations	2	409	310	24%
E-Learning Platform	Totals	21	487	479	2%
	Data Model	1	20	20	0%
	Objects	9	317	317	0%
	Relations	10	123	115	7%
	Coordinations	1	27	27	0%

combinations of objects with up to 11,110 instances and measured the time it took to execute their lifecycles processes. The confidence interval for the execution of the most complex experiment was [6.6–8.4] s. The experiment concurrently executed the lifecycle processes of all 11,110 object instances from start to finish, involving a total of over 20,000 microservices at the same time. The corresponding result with the same mix of objects, but only involving 1,110 object instances was [0.4–0.5] s. Furthermore, we noted perfectly linear scaling of the execution times with the number of steps present in the lifecycle processes. Therefore, we consider the re-execution times of object instances a very minor factor in the context of the ad-hoc change concept. This is supported by our measurement results from conducting ad-hoc changes to the real-world data model instance presented in Section 6.2, in which it was shown that even in the absolute worst case, i.e., that every single object instance in the entire data model instance is affected by an ad-hoc change, the actual re-execution only takes on average 157 ms for a large data model instance of almost 3000 microservices. In consequence, we chose to focus our efforts on optimizing the copy operation.

6.4.2. Measuring the effectiveness and performance of log replay and copy operations

This section presents the results of performance measurements we conducted while evaluating our concept with respect to research question *RQ3*, which shall investigate how the concept can be made efficient enough to be able to handle large numbers of ad-hoc change operations concurrently. In particular, the goal is to assess whether the optimizations and extensions presented in Section 5 improve the speed of ad-hoc copy operations significantly.

The time it takes to rebuild an entity from its log-based representation depends on the number of steps a process modeler applied when creating the entity at design time (cf. Section 5). This performance issue, which we alleviated through log pruning, has a large impact on the time it takes to create temporary copies of data model instances as well as replay changes from the temporary instances back to the original instance—two core ideas of our concept for ad-hoc changes (cf. Section 4.2). This section further examines this aspect and provides measurement results for copy operations on three different data models. The three real-world data sets used for the measurements are complete object-aware data models, that we modeled in three different case studies.

The *Recruitment* data model is a typical human resources application, supporting the process of hiring new employees and reviewing their applications; this data model was already presented in Section 6.1.2. The *Employee Self-Service* data model, in turn, is a portal for employees to perform common tasks such as putting in vacation requests or managing their contact information. Both models were created by students as projects and are not in real-world use. However, they offer insights into the benefits that log pruning offers for models that were created with many errors, redos and redundancies. Finally, the *E-Learning Platform* model (cf. Section 6.2) was not created using our regular modeling tools, but instead manually, as an almost perfectly pruned log file in JSON format. This became necessary to test the performance impact of the pruning algorithm itself when applying it to optimized logs (cf. Table 2).

The three data models are available for download in their log-based form from Mendeley Data.[6] We supply them once in their original form after exporting them from the modeling tool, and once in their pruned form after running Algorithm 3. Table 1 gives insights into the models as well as their complexity in terms of the number of entities present in the model (cf. Col. *Entities*), as well as the number of log entries before (cf. Col. *Unpruned*) and after (cf. Col. *Pruned*) executing the log pruning algorithm. Furthermore, we calculated the reduction in log size that log pruning algorithm achieves (cf. Col. *Reduction*).

We chose the copy operation of a running data model instance as a benchmark to demonstrate the performance improvements that can be achieved based on the concepts presented in Section 5. We repeated the measurements with different combinations of the three log replay optimization algorithms (i.e. log pruning, log grouping, and log parallelization) to show the effects they have, individually and in combination, on the time it takes to complete copy operations of entire data model instances at runtime. Note that log parallelization may only be applied to grouped and pruned logs—otherwise, the correct order of concurrently executed modeling operations cannot be ensured while replaying the logs.

In general, measuring the performance of a distributed software system is a non-trivial task, as there are a large number of factors, such as random network delays or network optimization algorithms, which are out of the control of the application programmer, but can influence measurement results. Therefore, we adopted the algorithm presented in [26], Annex A, to ensure mathematically sound confidence intervals for our results. More specifically, this algorithm describes a method for determining whether a measurement was run often enough to be at least 95% confident in a given confidence interval, which we require to have a width of at most 10% of the maximum value measured. Table 2 depicts the results of our measurements for each optimization algorithm introduced in this paper (cf. Col. *Optimization*) along with the amount of runs (cf. Col. *Runs*) that were necessary to achieve over 95% confidence (cf. Col. *Confidence*) in the measured confidence interval (cf. Col. *Confidence Interval*).

The results from Table 2 show that the performance optimizations introduced in Section 5 have a major impact on the time it takes to create temporary copies of data models using the log replay method. As these copy operations are a core pillar of the ad-hoc change concept and absolutely necessary for ensuring that it meets Requirements 1, 2 and 4, the performance increase is considered to be very valuable. This is especially true for our future work on schema evolution.

However, the copy operations we measured in this experiment are only half of the computational work that accompanies an ad-hoc change, as we laid out in Section 4 (cf. Algorithm 1, Lines 7–10). The other half is the data-driven re-execution of the object instances after the ad-hoc changes are applied (cf. Algorithm 1,

[6] https://data.mendeley.com/datasets/9nym9xykvs/1.

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

ARTICLE IN PRESS

Table 2
Measurement results.

Data Model name	Log optimization	Confidence interval milliseconds	Confidence $1-\alpha$	Runs n
Recruitment	Unoptimized	[23517–24415] ms	96,88%	6
	Group	[1963–2060] ms	96,14%	12
	Prune	[6890–7029] ms	96,88%	6
	Prune/Group	[1536–1570] ms	96,88%	6
	Prune/Group/Parallelize	[1242–1292] ms	96,09%	8
Employee Self-Service	Unoptimized	[44210–45206] ms	96,88%	6
	Group	[3746–3916] ms	96,88%	6
	Prune	[13542–14216] ms	96,88%	6
	Prune/Group	[3280–3579] ms	96,88%	6
	Prune/Group/Parallelize	[2606–2678] ms	96,88%	6
E-Learning Platform	Unoptimized	[1448–1546] ms	96,88%	6
	Group	[632–693] ms	96,88%	6
	Prune	[1459–1527] ms	96,88%	6
	Prune/Group	[687–732] ms	96,14%	11
	Prune/Group/Parallelize	[406–438] ms	96,88%	6

Lines 11–13). In terms of a data-driven process engine, however, this is no different than a regular execution of the corresponding lifecycle processes, in which all data values are supplied at the same time. We explicitly chose not to conduct measurements for these execution times in the context of this article, as we already completed such measurements in previous work [25].

While the performance gains from grouping and parallelization of logs are substantial, they are very specific to our current implementation architecture using distributed microservices. This is due to the fact that microservices can handle parallelized workflows very well, as they do not have central bottlenecks, such as a database. However, they perform sub-optimally when there is a large communication overhead, such as the one introduced if logs are not grouped and, therefore, re-played one by one. This causes a large number of remote procedure calls between cluster nodes. Therefore, the important thing to note is the effectiveness of the log pruning algorithm, as it offers benefits irrespective of the implementation architecture. Comparing the results for the *Recruitment* data model (cf. Table 1), where the pruning algorithm reduced the log size by an average of 60%, we can see a superlinear reduction in measurement time of 71% between unoptimized logs and pruned logs in Table 2. This can be explained with the fact that in the *Recruitment* data model an unusually large number, 72%, of computationally expensive modeling operations, such as creating new relations, could be pruned from the data model logs. This just shows how important log pruning is in the context of ad-hoc changes, as these gains are independent of the implementation architecture or hardware capabilities.

Finally, consider the *E-Learning Platform* results for unoptimized vs pruned logs. As stated previously, the corresponding data model was created without using the PHILharmonicFlows modeling tools, but instead by directly creating a log file in JSON format. This allowed us to craft the data model with almost no unnecessary log entries for modeling actions that would be pruned (cf. Table 1). We utilize this model as a benchmark to determine how much impact the pruning algorithm itself has on the total execution time. The pruning algorithm has almost the same algorithmic complexity, not dependent on whether or not it actually prunes any logs. In consequence, this shows that the overhead introduced by the algorithm itself is entirely negligible, as it is within the confidence interval for both measurements.

7. Related work

As the maturity of the data-centric process support paradigm is generally considered as low compared to activity-centric approaches, both execution concepts and execution engines are rare. Consequently, to the best of our knowledge, directly related work, i.e., data-centric approaches offering flexibility in terms of ad-hoc changes to running process instances, is virtually non-existent.

Note that other data-centric approaches like artifact-centric business processes [27] or case handling [9] already offer a high level of flexibility during process execution, which is to be expected due to the largely declarative modeling nature [28]. The tooling support for case handling (i.e. FLOWer [9]), for instance, offers ad-hoc flexibility in terms of skip or redo capabilities. As a drawback, this flexibility is restricted to control-flow aspects.

The DEZ-Flow engine [29], built upon the artifact-centric approach, allows defining declarative rules, which allow for ad-hoc changes to running instances at predefined points. While these rules are editable at runtime, the approach itself does not cover every possible deviation from the standard process as the process model remains unchanged.

Declarative process modeling approaches offer a similar built-in flexibility compared to data-centric approaches. Instead of requiring designers to specify how the process shall be executed, they only have to state what shall be done during process execution. With declarative approaches ad-hoc changes become less frequent. However, ad-hoc changes still can be an issue (e.g., a constraint might have to be violated for a particular instance due to an unforeseen situation). Further, constraints themselves may evolve over time, which raises the challenge of propagating changes to ongoing instances. A declarative process support approach, DECLARE, which enables ad-hoc changes is presented in [30]. Thereby, a change is defined by adding constraints, deleting constraints or updating the constraint set of a particular process instance or process type. However, DECLARE neglects the data perspective as well as performance and scalability issues. Another notable declarative approach, which allows for a more complete approach, is provided by DCRGraphs [31]. This activity-centric approach enables ad-hoc changes through DCRGraph adaptations, but does not allow for changes at that fine-grained level as supported in PHILharmonicFlows.

COREPRO [17] supports the assembly of products from product components, as it is commonplace in (automotive) engineering. In particular, this requires the adaptation and coordination of large process structures, represented by individual data objects. Adaptation is accomplished directly to running instances, and changes leading to an inconsistent process state are prevented.

Research on model changes in Adaptive Case Management (ACM) is presented in [32]. Specifically, the conduction of change operations is examined to determine their impact on a given Guard-State-Milestone (GSM) model. However, the paper is limited in respect to aspects for adapting running process instances.

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.

Agile BPM [33] is supported by the Fujitsu Interstage BPM Process Manager. In Agile BPM, users may assign simple tasks to other users in a completely ad-hoc fashion with a concept entitled *dynamic tasking*. A dynamic task constitutes a short task description text, with some additional meta-information, e.g. who it shall be assigned to and by when the task shall be completed. Users may create such tasks on the fly, assign them to other users, and monitor their completion. The capabilities of the dynamic tasking system are limited to small tasks that can be described textually, and cannot involve complex data flow or external systems as is the case for PHILharmonicFlows.

There are many approaches to process flexibility in activity-centric BPM, but their ad-hoc change support is limited to changing entire activities (e.g. moving or skipping an activity[7]) as opposed 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 [2,34–36].

Finally, activity-centric approaches exist that apply log replay in the context of ad-hoc changes [37]. [38] presents an activity-centric approach that enables case-based ad-hoc adaptation of running process instances. The approach employs process adaptation cases that record adaptation episodes from the past. The recorded changes (i.e., logs) can then be automatically transferred to a new process instance being in a similar situation of change. The case-based adaptation method uses the so-called anchor mapping algorithm to identify the parts of the target process where to apply the changes, and log replay is used to introduce the change to this process. A comparable approach enabling case- and log-based ad-hoc adaptations is CBRFlow [39].

8. Summary and outlook

The concepts presented in this article 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 were designed in a way that allows for their use in a microservice-based process engine, PHILharmonicFlows, utilized by us as a proof-of-concept for the presented concepts in multiple scenarios. As object-aware process management has an inherently tight integration between process logic and data, this proof-of-concept has capabilities that go far beyond those of ad-hoc changes in activity-centric approaches.

In regards to the research questions, we are confident that we have answered all of them with the concept and the extensions presented in this article. In particular, the feasibility of a solution in which ad-hoc changes are enabled *(RQ1)* is mainly answered by the creation of copies of the "live" object instances that allow for further interactions with the object instances while ad-hoc changes are being prepared. The flexibility and usability of the proposed solution *(RQ2)* on the other hand is answered through our use of modeling operation logs. These are applied to the models underlying the object instances or data model instances the change operation are applied to, instead of the more common approach of defining a set of predefined valid change operations. Combining this with our re-execution mechanism ensures that any possible modeling operation is a valid ad-hoc change as well. Finally, the scalability of the proposed solution *(RQ3)* is answered through our improvements of the initial concept by pruning the logs the concept heavily relies upon, reducing the necessary steps to copy and re-execute an ad-hoc changed object or data model instance.

Having presented the optimizations we developed on top of the key parts of the ad-hoc change concept we intend to further address the remaining issues concerning the performance of the developed solution. However, we first have to elicit the exact requirements companies are facing in respect to performance and scalability of ad-hoc changes as well as an adequate test setup for change scalability on a much larger scale, i.e., with potentially hundreds of users interacting with a data model while it is being changed. This will be accomplished in conjunction with further research on the topic of data model schema evolution, including all attached data model instances as well as their object instances. Note that this is where performance might become an issue. Furthermore, while we have not yet measured the time ad-hoc changes take in very large real-world data model instances, we can improve the speed through horizontal scalability utilizing the microservice architecture of PHILharmonicFlows.

Finally, even though this article presents a concept for allowing ad-hoc changes to all conceptual elements present in an object-aware process, we will conduct further research to determine which ad-hoc change operations are actually needed from the user perspective. Furthermore, the results of the PHoodle study (cf. Section 6.2) will provide valuable insights into the usability of generic process management approaches from an end-user perspective, especially in direct comparison with purpose-built software solutions.

While the presented solution might not be evaluated for usability in larger scale real-world business applications, we have employed it successfully to a number of scenarios with our proof-of-concept implementation. Additionally, 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 and awareness [7].

Declaration of competing interest

The authors confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

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. Weske, Business process management architectures, in: Business Process Management, Springer, 2012, pp. 333–371.
[2] M. Reichert, B. Weber, Enabling Flexibility in Process-aware Information Systems: Challenges, Methods, Technologies, Springer, 2012.
[3] M. Reichert, P. Dadam, ADEPT flex-supporting dynamic changes of workflows without losing control, J. Intell. Inf. Syst. 10 (2) (1998) 93–129.
[4] R. Müller, U. Greiner, E. Rahm, Agentwork: a workflow system supporting rule-based workflow adaptation, Data Knowl. Eng. 51 (2) (2004) 223–256.
[5] G. Vossen, M. Weske, The WASA2 object-oriented workflow management system, in: SIGMOD Conf., 1999, pp. 587–589.
[6] M. De Leoni, M. Mecella, G. De Giacomo, Highly dynamic adaptation in process management systems through execution monitoring, in: Int Conf on Business Process Management, Springer, 2007, pp. 182–197.
[7] S. Steinau, A. Marrella, K. Andrews, F. Leotta, M. Mecella, M. Reichert, DALEC: a framework for the systematic evaluation of data-centric approaches to process management software, Softw. Syst. Model. 18 (4) (2019) 2679–2716.

[7] For an overview on characteristic activity-centric change patterns, we refer interested readers to Chapter 2 of [2].

ARTICLE IN PRESS

[8] R. Hull, E. Damaggio, F. Fournier, M. Gupta, F.T. Heath, S. Hobson, M. Linehan, S. Maradugu, A. Nigam, P. Sukaviriya, Introducing the guard-stage-milestone approach for specifying business entity lifecycles, in: Int Workshop on Web Services and Formal Methods, 2010, pp. 1–24.

[9] W.M.P. van der Aalst, M. Weske, D. Grünbauer, CaSe handling: a new paradigm for business process support, Data Knowl. Eng. 53 (2) (2005) 129–162.

[10] V. Künzle, M. Reichert, Philharmonicflows: towards a framework for object-aware process management, J. Softw. Maintenance Evol. Res. Practice 23 (4) (2011) 205–244.

[11] K. Andrews, S. Steinau, M. Reichert, Enabling ad-hoc changes to object-aware processes, in: 22nd Int Enterprise Distributed Object Computing Conf, EDOC, IEEE, 2018, pp. 85–94.

[12] S. Steinau, K. Andrews, M. Reichert, The relational process structure, in: 30th Int Conf on Advanced Information Systems Engineering, CAiSE, Springer, 2018, pp. 53–67.

[13] V. Künzle, Object-aware process management (Ph.D. thesis), Ulm University, 2013.

[14] K. Andrews, S. Steinau, M. Reichert, Enabling fine-grained access control in flexible distributed object-aware process management systems, in: 21st Int Enterprise Distributed Object Computing Conf, EDOC, IEEE, 2017, pp. 143–152.

[15] K. Andrews, S. Steinau, M. Reichert, Enabling process variants and versions in distributed object-aware process management systems, in: Forum of the 30th Int Conf on Advanced Information Systems Engineering, CAiSE Forum, 2018, pp. 1–15.

[16] S. Steinau, V. Künzle, K. Andrews, M. Reichert, Coordinating business processes using semantic relationships, in: 19th IEEE Conf on Business Informatics, CBI, 2017, pp. 143–152.

[17] D. Müller, M. Reichert, J. Herbst, Flexibility of data-driven process structures, in: BPM'06 Int Workshops, Workshop on Dynamic Process Management, DPM, Springer, 2006, pp. 181–192.

[18] W. Song, X. Ma, H.-A. Jacobsen, Instance migration validity for dynamic evolution of data-aware processes, IEEE Trans. Softw. Eng. (2018) (early access).

[19] C.M. Chiao, V. Künzle, M. Reichert, Object-aware process support in healthcare information systems: requirements, conceptual framework and examples, Int. J. Adv. Life Sci. 5 (1 & 2) (2013) 11–26.

[20] V. Ferme, A. Ivanchikj, C. Pautasso, A framework for benchmarking BPMN 2.0 workflow management systems, in: Int Conf on Business Process Management, BPM, Springer, 2016, pp. 251–259.

[21] V. Ferme, M. Skouradaki, A. Ivanchikj, C. Pautasso, F. Leymann, Performance comparison between BPMN 2.0 workflow management systems versions, in: Enterprise, Business-Process and Information Systems Modeling, Springer, 2017, pp. 103–118.

[22] M. Skouradaki, V. Ferme, C. Pautasso, F. Leymann, A. van Hoorn, Micro-benchmarking BPMN 2.0 workflow management systems with workflow patterns, in: Int Conf on Advanced Information Systems Engineering, CAiSE, Springer, 2016, pp. 67–82.

[23] S. Steinau, K. Andrews, M. Reichert, A modeling tool for PHILharmonicFlows objects and lifecycle processes, in: Proceedings of the BPM Demo Session, BPMD, 2017.

[24] G. Agha, C. Hewitt, Concurrent programming using actors: Exploiting large-scale parallelism, in: Int Conf on Foundations of Software Technology and Theoretical Computer Science, Springer, 1985, pp. 19–41.

[25] K. Andrews, S. Steinau, M. Reichert, Engineering a highly scalable object-aware process management engine using distributed microservices, in: 26th Int Conf on Cooperative Inf Sys, CoopIS, Springer, 2018, pp. 80–97.

[26] J.-Y. Le Boudec, Performance evaluation of computer systems, EPFL Press, 2010.

[27] D. Cohn, R. Hull, Business artifacts: A data-centric approach to modeling business operations and processes, IEEE Data Eng. Bull. 32 (3) (2009) 3–9.

[28] W.M.P. van der Aalst, M. Pesic, H. Schonenberg, Declarative workflows: Balancing between flexibility and support, Comput. Sci. Res. Dev. 23 (2) (2009) 99–113.

[29] W. Xu, J. Su, Z. Yan, J. Yang, L. Zhang, An artifact-centric approach to dynamic modification of workflow execution, in: Proc. 19th Int Conf on Cooperative Information Systems, CoopIS, Springer, 2011, pp. 256–273.

[30] M. Pesic, H. Schonenberg, N. Sidorova, W.M.P. van der Aalst, Constraint-based workflow models: change made easy, in: Proc. 15th Int Conf on Cooperative Information Systems, CoopIS, 2007, pp. 77–94.

[31] T.T. Hildebrandt, Flexible, adaptable, and compliant business systems with dynamic condition response graphs, in: Int Workshop on Formal Methods for Analysis of Business Systems, 2016, pp. 1.

[32] R. Eshuis, R. Hull, M. Yi, Property preservation in adaptive case management, in: Int Conf on Service-Oriented Computing, ICSOC, Springer, 2015, pp. 285–302.

[33] K. Swenson, Taming the Unpredictable: Real World Adaptive Case Management: Case Studies and Practical Guidance, Future Strategies Inc., 2011.

[34] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, W.M.P. van der Aalst, Process flexibility: A survey of contemporary approaches, in: Advances in Enterprise Engineering I, Springer, 2008, pp. 16–30.

[35] S. Rinderle, Schema evolution in process management systems (Ph.D. thesis), Ulm University, 2004.

[36] B. Weber, M. Reichert, J. Mendling, H.A. Reijers, Refactoring large process model repositories, Comput. Ind. 62 (5) (2011) 467–486.

[37] S. Rinderle, M. Reichert, M. Jurisch, U. Kreher, On representing, purging, and utilizing change logs in process management systems, in: International Conference on Business Process Management, Springer, 2006, pp. 241–256.

[38] M. Minor, R. Bergmann, S. Görg, Case-based adaptation of workflows, Inf. Syst. 40 (2014) 142–152.

[39] B. Weber, W. Wild, R. Breu, CBRFlow: enabling adaptive workflow management through conversational case-based reasoning, in: Advances in Case-Based Reasoning (ECCBR) 2004, 2004, pp. 434–448.

Please cite this article as: K. Andrews, S. Steinau and M. Reichert, Enabling runtime flexibility in data-centric and data-driven process execution engines, Information Systems (2019) 101447, https://doi.org/10.1016/j.is.2019.101447.