Dynamically Switching Execution Context in Data-Centric BPM Approaches

Kevin Andrews(✉), Sebastian Steinau, and Manfred Reichert

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

Abstract. In contemporary business process management software, the context in which a process is executed is largely static. While the execution of the process itself may be flexible, on-the-fly changes to the context, i.e., physical or logical surroundings, are either limited or impossible. This paper presents concepts for enabling context switching at runtime for the object-aware process management paradigm. Such context switches are enabled at various granularity levels, such as shifting entire process instances to different systems, or migrating sub-processes between different parent processes. We further contribute the algorithms employed in our proof-of-concept implementation and discuss use cases in which context switching capabilities can be utilized. Implementing these advanced concepts helps showcase the maturity of data-centric BPM.

Keywords: Object-aware processes · BPM · Process context switching

1 Introduction

The context in which a process is executed determines essential factors at runtime. These range from trivial ones, e.g. whether or not a process may be executed, to complex factors, such as the selection of sub-process variants at runtime. Making business process management systems (BPMS) *context-aware* increases the flexibility of processes they execute by supporting business rules that are enforced based on the context [1]. Informally, process context is defined as "the minimum set of variables containing all relevant information that impacts the design and execution of a business process" [2], which emphasizes the importance of context for process execution. However, contemporary BPMS do not allow changing the execution context of running process instances, even though this would increase flexibility. Consider a recruitment process from the HR domain, in which applicants apply for a job offer, as an example in which flexibility could be increased by allowing for process context switches at runtime. More specifically, the context of a job application process corresponds to the job offer an applicant applies to, as well as meta information such as the department the respective job is allocated to. Furthermore, consider an unsolicited application.

S. Nurcan et al. (Eds.): BPMDS 2020/EMMSAD 2020, LNBIP 387, pp. 3–19, 2020.
https://doi.org/10.1007/978-3-030-49418-6_1

In the first case, while the context of the application process seems to be clear at the beginning, during the course of a job interview, it might be decided that the applicant would better fit a different job at another department. In the second case, parts of the context, such as a concrete job offer, are missing entirely and can only be determined after the process starts. Although both cases can be partially handled in an activity-centric BPMS by adding gateways and loops into the process model, this would make the process model unnecessarily complex. Therefore, most companies handle cases like these by forcing applicants to resubmit their application to a different job offer, which, in future, might cause confusion due to multiple applications from the same person.

This paper presents solutions to these issues for object-aware BPM, a data-centric process management paradigm, by enabling dynamic process context switches without requiring process model changes. The paper builds upon previous work that led to the development of the PHILharmonicFlows process engine and contributes fundamental research into the notion of process context in data-centric BPM paradigms. The fundamentals of object-aware BPM are explained in Sect. 2. The notion of process context in object-aware processes is examined in Sect. 3. The concepts and algorithms for enabling context switching are presented in Sect. 4. An overview of our prototype implementation is given in Sect. 5. Section 6 discusses related work and Sect. 7 summarizes the paper.

2 Backgrounds

The PHILharmonicFlows implementation of object-aware BPM, a data-centric BPM paradigm, has been under development for many years and serves as a testbed for the concepts presented in this paper [3,4]. PHILharmonicFlows takes the idea of a data-driven BPMS and enhances it with the concept of *objects*. An *object* describes the structure of its contained data and process logic at design-time whereas an *object instance* holds concrete data values and executes the process logic at runtime. This may be compared to the concept of a table and its rows in a relational database. For each business object present in a real-world business process one such object exists. We further examine the concept of objects utilizing an *Application* object from the HR domain. As can be seen in Fig. 1, the object consists of data, in the form of *attributes*, and a state-based process model describing the data-driven *object lifecycle*.

As object-aware BPM is *data-driven*, the lifecycle execution of an instance of the *Application* object is as follows: The initial state is *Created*. Once an *Applicant* has entered data for attributes *Job Offer*, *Applicant*, and *CV*, he or she may trigger the transition to the *Sent* state. This causes the *Application* to change its state to *Sent*, in which it waits until the reviewing period is over,

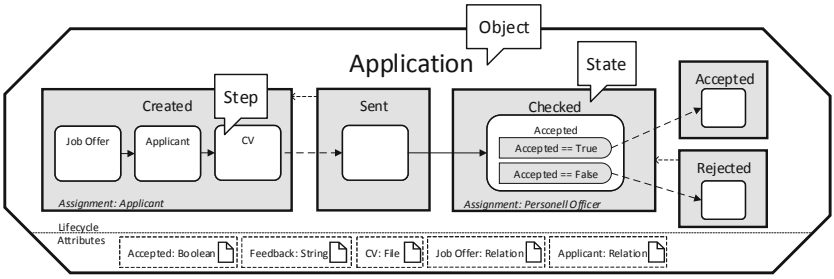


Fig. 1. Example object including lifecycle process (Application)

after which it automatically enters state *Checked*. As *Checked* is assigned to a *Personnel Officer*, a user with that role must input data for the *Accepted* attribute. Based on the value of *Accepted*, the state either changes to *Accepted* or *Rejected*.

This fine-grained approach to modeling the processing of a single business object 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. However, as one of the major benefits, 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 states. This information is combined with permissions, resulting in a personalized form with interaction logic. An example of such a form, derived from the *Application* object from Fig. 1, is shown in Fig. 2.

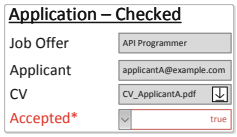


Fig. 2. Form

Note that a single object and its resulting forms are only part of a complete business process. To allow for more complex business processes, many different objects and users may have to be involved [3]. It is noteworthy that *users* are simply special objects in the object-aware paradigm. The entire set of objects present in a PHILharmonicFlows process is the *data model*, an example of which can be seen in Fig. 3, with objects representing users, e.g. *Employee*, marked in green.

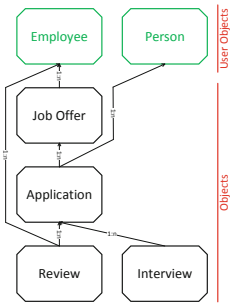


Fig. 3. Data model (Color figure online)

In addition to the objects and users, the data model contains information about the *relations* existing between them. A relation constitutes a logical association between two objects, e.g., a *Job Offer* and an *Application*. At runtime, each of the objects may be instantiated many times as *object instances*. Note that the lifecycle processes present in the various object instances may be executed concurrently at runtime, thereby improving overall system performance. Relations may also be instantiated at runtime, e.g., between an instance of a *Review* and an *Application*, thereby associating the two object instances with each other. The resulting meta information, expressing that the *Review* in question belongs to the *Application*, can be used to coordinate the processing of the two object instances with each other at runtime [3]. Figure 4 shows an example of a *data model instance* at runtime.

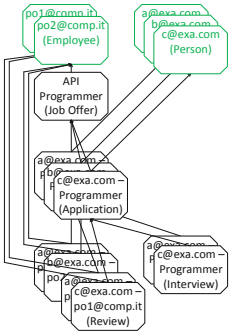


Fig. 4. Data model instance

The coordination of object instances is necessary as business processes often consist of hundreds or thousands of interacting business objects [5], whose concurrent processing needs to be synchronized at certain states. As object instances publicly advertise their state information, the current state of an object instance (e.g. *Sent* or *Check*ed) can be used for coordinating its processing (i.e., execution) with other object instances corresponding to the same business process through a set of constraints and rules, defined in a separate *coordination process* [3]. As an example, consider a simple subset of constraints stating the following:

1. An *Application* must be in state *Sent* for *Reviews* to be *Prepared* for it.
2. An *Application* may only be *Checked* once its corresponding *Reviews* are either in state *Reject Proposed* or *Invite Proposed*.

A coordination process with these constraints is shown in Fig. 5. The transitions describe the kind of relation that exists between the objects referenced by the steps on either side. For example, between steps *Review - Reject Proposed* and *Application - Checked*, a *bottom-up* coordination exists, as there is a bottom-up many-to-one relation between *Review* and *Application* in the data model (cf. Fig. 3). This enables advanced coordinations, based on the information delivered by relations at runtime, e.g. that at least 5 *Reviews* must be either in states *Reject Proposed* or *Invite Proposed* for an *Application* to enter state *Checked* [3].

3 Determining Object-Aware Process Context

This section examines the concepts we developed for enabling *process context switching*. To reiterate, an object-aware process instance consists of a data model instance that comprises many object instances. Further, there are relation instances between associated object instances. Finally, the coordination process

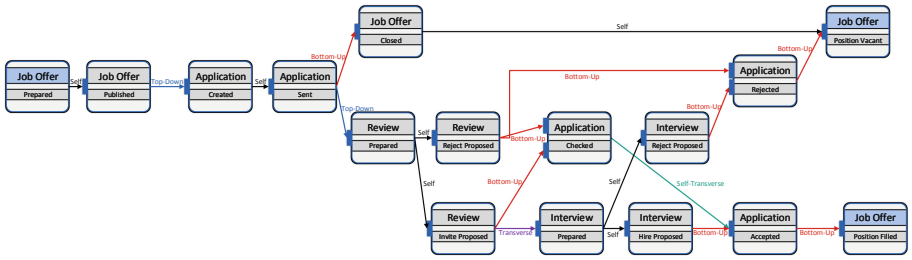


Fig. 5. Coordination process (Recruitment - Job Offer)

instance monitors the object instances and coordinates their execution. Consequently, in an object-aware process instance, many different processes, such as lifecycle processes and the coordination process, are executed concurrently. As one can not simply determine a *single* process context for this collection of largely independent processes, this section presents four points of view, or *scopes*, one may use to examine the combined process context.

3.1 Process Context in the Scope of a Lifecycle Process

When examining process context, the notion of *scope* becomes important. In the scope of a lifecycle process, the context is the object instance the lifecycle process is executed in, or, more specifically, the set containing all current attribute values present in the object instance. This complies with the definition in [2]. As an object-aware lifecycle process is entirely data-driven, the process context, i.e., the object instance, contains all necessary execution information, i.e., the attribute values. As an example, take the lifecycle process instance shown in Fig. 6.

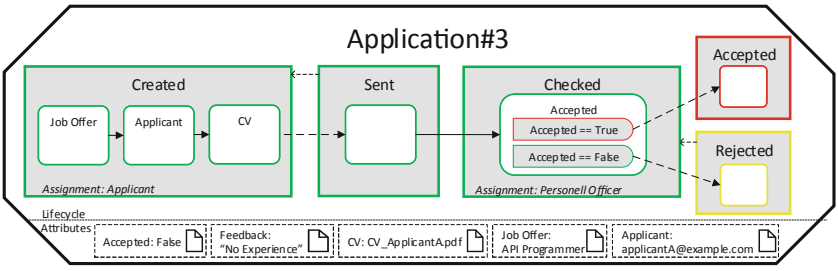


Fig. 6. Lifecycle process of an Application object instance

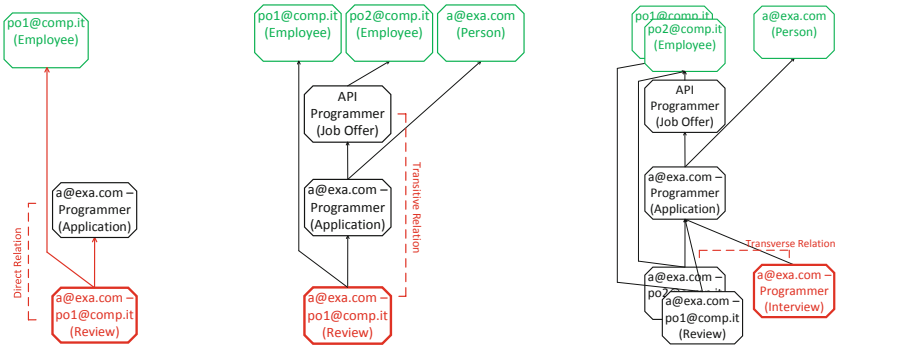
The lifecycle process is currently in state *Rejected*, as all attribute values from the previous states have already been written. This includes attribute *Accepted* with its value *False*, which forced the corresponding decision step in *Checked* to trigger the transition to *Rejected*. In turn, this led to the current state of the

object instance being *Rejected*. Note that these attribute values always lead to the same state if the lifecycle process is re-executed. In summary, if the scope of the process context is limited to a lifecycle process, the context will solely consist of attribute values. However, this scope is too limited for most purposes.

3.2 Process Context in the Scope of a Single Object Instance

The value of the *Accepted* attribute of *Application#3*, shown in Fig. 6, is set to *False*. While a personnel officer may immediately accept or reject an applicant, it is more realistic that the application is first reviewed and applicants are invited for an interview before making a decision. In the data model of the recruitment example (cf. Fig. 3), these reviews and interviews are represented by objects, *Review* and *Interview*, with their own lifecycle processes. Note that *Review* and *Interview* constitute so-called *lower-level objects* of *Application*, as level-wise they are below *Application* in the data model due to the layouting of the graph that is based on incoming and outgoing relations. This layout makes the parent/child relations between objects evident, e.g., it becomes obvious from Fig. 3 that *Reviews* belong to *Applications*. These relations, together with the notion of scope, are crucial when determining process context.

As stated before, the smallest scope is given by a single lifecycle process, for which the process context is the object it is associated with. The next larger scope is the scope of an object instance, e.g., one instance of a *Review* object. Clearly, a *Review* is always conducted by an *Employee* for a specific *Application*, which can be deduced by examining the relations to the higher-level objects of *Review* (cf. Fig. 3). Therefore, the process context of a *Review* object instance is given by the *Employee* and *Application* object instances it is related to, i.e., all higher-level object instances of the *Review*. Taking the example from Fig. 4, the process context of one of the *Reviews* is shown in Fig. 7a.



(a) Direct Context (*Review*) (b) Transitive Context (*Review*) (c) Coordination Context (*Interview*)

Fig. 7. Process contexts

Note that Fig. 7a is captioned "Direct Context", as opposed to the "Transitive Context" shown in Fig. 7b. The difference between these notions is that the transitive context includes *all* object instances that are higher-level object instances of the review object instance, not just those that are directly related. Note that even the *transitive context* of an object instance does not cover the complete process context in the scope of a single object instance, as not all object instances are contained that can have an impact on the execution of the object instance in question.

Specifically, the coordination process, which determines the interactions between object instances at runtime, provides additional context for object instances. (cf. Fig. 5). In particular, the coordination process allows process modelers to define constraints, such as that an *Interview* may only be *Prepared* if there are *Reviews* in state *Invite Proposed*, which are *transversely* related to them, i.e., via the same *Application* (cf. Fig. 5). The existence of such a constraint means that the execution of an *Interview* object instance may be impacted by a *Review* object instance, which is not be part of the transitive context of the *Interview*, as *Review* is not a higher-level object of *Interview*. Therefore, the coordination context is the most complete context in the scope of an object instance, as it contains all other object instances that may impact this instance. Figure 7c shows an example of the coordination context of an *Interview* object instance.

3.3 Process Context in the Scope of Multiple Object Instances

This section examines process context in the scope of multiple related objects. Note, for instance, that the process context of, e.g., an instance of the *Application* object, would be comprised of (1) the higher-level object instances it is related to (e.g., a *Job Offer*) as well as (2) other objects that may impact the execution of the *Application*, as defined in the coordination process (e.g., *Reviews*). However, when switching this process context to a completely different one, we need to consider that an *Application* is not an independent entity in a data model instance. Simply deleting all relation instances from an *Application* object instance, e.g. *Application#3*, and attaching it to a different *Job Offer* object instance, e.g. from *Job Offer API Programmer* to *Job Offer UI Programmer*, would leave all the *Review* and *Interview* object instances as orphans in the current data model instance. Therefore, concepts for switching the process context of object instances must always consider the context of all dependent object instances as well.

The set of dependent object instances can be defined as all transitively related lower-level object instances. Note that this is an inversion of the logic for finding transitive process context (cf. Fig. 7b). As an example, the aggregation of the dependent object instances of an *Application* object instance may be algorithmically determined by recursively evaluating incoming relations from lower-level object instances, the result of which is shown in Fig. 8.

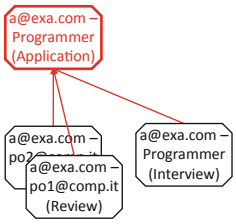


Fig. 8. Dependent object instances

In summary, the process context of an individual object must always include its dependent object instances, as switching the context of an individual object instance with lower-level instances attached to it would lead to orphaned object instances. In essence, this means recursively calling the functions that determine the dependent object instances and the coordination process context of all identified instances, which, in most cases, leads to the entire data model instance (cf. Fig. 4) being identified as the process context of an object instance. Consequently, we must examine the notion of process context in the scope of a data model instance.

3.4 Process Context in the Scope of the Data Model Instance

In contrast to the concept of a "process instance" from the realm of activity-centric BPMSs, an object-aware data model instance does not have any input parameters or meta information it holds at runtime, apart from the object instances, relation instances, and coordination process instances it comprises. Consequently, as opposed to an activity-centric process instance, whose context would include the input parameters provided to the process instance upon its creation, a data model instance possesses no process context information. Returning to the recruitment management process example, there might be multiple instances of the recruitment data model running at the same time across different departments in a company. The context in which they are executed, such as the department, is not captured in the data model. However, this also means that there is no conceptual challenge in changing the context of a data model instance. Specifically, moving the data model instance from one server to another is merely an administrative challenge.

4 Enabling Dynamic Process Context Switching

In Sect. 3, we presented the scopes one has to consider when determining what constitutes process context in an object-aware process. There is no simple way of taking a single object instance or other conceptual element and determining its process context in a general fashion, as, when including all constraints and relations, the process context of a single object instance consists of all object instances present in a data model instance. Without additional concepts, there is no way to remove an object instance from its process context and re-insert

Algorithm 1. Re-Execute Lifecycle with altered Attribute Values

Require: $oi, newAttributes[]$ ▷ object instance, new attribute values
1: $o \leftarrow$ **getObject**(oi) ▷ get underlying object of oi
2: $oi_{temp} \leftarrow$ **instantiate**(o) ▷ create an empty instance of o
3: **for all** a in $oi.attributes[]$ **do** ▷ copy O by change log replay
4: **if** a **not in** $newAttributes[]$ **then** ▷ if a is not being replaced
5: $newAttributes[] \leftarrow a$ ▷ append attribute values from oi
6: **end if**
7: **end for**
8: $c \leftarrow$ **getCoordinationProcess**(oi)
9: **for all** a in $newAttributes[]$ **do** ▷ insert attribute values from O
10: $oi_{temp}.changeAttributeValue(a)$ ▷ each value advances the lifecycle, re-executing it step by step
11: $c.update(oi)$ ▷ notify coord. process if state changes
12: **end for**
13: $delete(oi)$
14: $oi \leftarrow oi_{temp}$ ▷ replace all pointers to original instance

it into another, as this would also change the context of other object instances, causing inconsistencies. This section presents concepts to enable changing or switching only parts of the process context of one or more object instances. We facilitate this with (a) the help of algorithms that perform the actual context changes, and (b) the inherent execution flexibility of object-aware BPM, which allows fixing inconsistent processes at runtime with dynamically generated forms.

4.1 Enabling Changes to the Context of a Lifecycle Process

The basic building block for enabling process context changes in object-aware process management is to enable context changes at the smallest scope possible, i.e., the process context of a lifecycle process (cf. Sect. 3.1). To reiterate, the process context of the lifecycle process being executed in an object instance corresponds to the supplied attribute values. As the lifecycle process is data-driven, its execution is advanced when certain data becomes available. The context of a lifecycle process, therefore, changes continuously, which drives process execution. This data-driven approach allows for the re-execution of a lifecycle process instance based on a *replay algorithm*. To be more precise, the data-driven nature of lifecycle processes ensures that the lifecycle process is re-executed in an identical fashion if the attribute values, i.e., the process context, remains unchanged. However, as we *want* to be able to change attribute values and then re-execute the lifecycle process, we extended the algorithm for re-executing a lifecycle process instance with the ability to alter attribute values (cf. Algorithm 1).

Note that it is not necessary to allow users to trigger this kind of process context change, as it is merely considered a building block for the higher-level user-facing context changes. Algorithm 1 is essential as it allows the lifecycle process to be re-executed when the object instance it belongs to switches its process context.

4.2 Enabling Changes to the Context of an Object Instance

The context of an object instance can be considered to be the data model instance itself. However, it is possible to alter only specific parts of the process context identified in Sect. 3.2, i.e., *direct context*, *transitive context*, and *coordination context*. Starting with the *direct context*, we developed a concept for enabling the exchange of directly related higher-level object instances, thereby altering or even entirely switching process context. As an example consider an *Application* related to a *Job Offer*. During a job interview, it turns out that *Applicant a@exa.com* is better qualified for a different *Job Offer*, e.g. *UI Programmer*. The personnel officer *po2@comp.it* may want to switch the context of the *Application* object instance from *API Programmer* to *UI Programmer* as shown in Fig. 9.

The changes necessary for switching the direct process context of the *Application* object instance shown in Fig. 9 are (a) removing the relation between the *Application* object instance and the *API Programmer Job Offer* object instance and (b) adding a new relation between the orphaned *Application* object instance and the *UI Programmer* object instance.

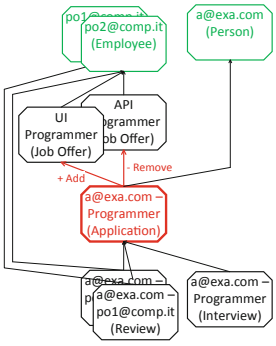


Fig. 9. Direct process context switch

These changes are inherently supported by object-aware processes. However, an impact analysis becomes necessary to determine which steps are required to restore consistency. As, up until now, we only take the direct context of the single object instance into account, the analysis must merely check that the new relation to *UI Programmer* is instantiatable, adhering to any cardinality or coordination constraints the data model may impose. An example of a cardinality constraint could be that each *Job Offer* object instance may have at most five *Application* object instances attached to it. Furthermore, a coordination constraint preventing the creation of the new relation could be that the *UI Programmer* object instance is not in state *Published* yet, which is necessary for *Applications* to be attached to it, according to the coordination process shown in Fig. 5. If none of these constraints is violated by the new relation, it may be created, causing the direct context of the *Application* to be switched to the *UI Programmer* object instance.

Algorithm 2 ensures that the context switch adheres to the constraints imposed on relations between, e.g., *Job Offers* and *Applications*, i.e., the object instances affecting the direct context of the *Application* object instance. Although Algorithm 2 is not overly complex, it is an important foundation that ensures that the direct context may be switched, thereby ensuring consistency of the data model instance after the process context change. If the algorithm finds a violation, the process context of the *Application* must not be switched.

Algorithm 2. Check Direct Context Constraints

Require: oi_s, oi_t ▷ source and target of new relation
1: $o_s \leftarrow$ **getObject**(oi_s) ▷ get underlying object of oi_s
2: $o_t \leftarrow$ **getObject**(oi_t) ▷ get underlying object of oi_t
3: $r \leftarrow$ **getRelation**(o_s, o_t) ▷ get relation between o_s and o_t
4: **if count**$(oi_t.incomingRelations) >= r.maxCard$ **then**
5: **return** false ▷ violated relation cardinality
6: **end if**
7: $c \leftarrow$ **getCoordinationProcess**(oi_t)
8: **for all** s in $c.steps$ **do**
9: **if** $s.object = o_s$ **and** $s.state = oi_s.currState$ **then**
10: **if constraintPreventsRelation**(s, oi_s, oi_t) **then**
11: **return** false ▷ violated coordination constraint
12: **end if**
13: **end if**
14: **end for**
15: **return** true ▷ allow relation instance between oi_s and oi_t

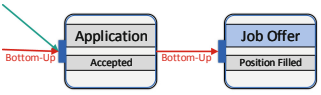


Fig. 10. Coordination process (cf. Fig. 5)

The direct process context change shown in Fig. 9 may be also viewed from a different angle. The change clearly impacts the context of the *Application* object instance, but it impacts the coordination context of both *Job Offer* instances as well. Assuming that the lifecycle process of the *Application* object instance is in state *Accepted* when the context switch occurs, the excerpt of the coordination process shown in Fig. 10 would force changes to the *Job Offers UI Programmer* and *API* Programmer. The new context, i.e., *Job Offer UI Programmer*, then has a lower-level *Application* object instance in state *Accepted*, which allows it to transition to state *Position Filled* as shown in the coordination process excerpt in Fig. 10. This causes the coordination process to inform the *UI Programmer* object instance that it may advance to state *Position Filled* as soon as the context change creating the new relation instance occurs.

Conversely, the old context, i.e., *Job Offer API Programmer*, no longer has an *Application* in state *Accepted* related to it. Therefore, if it is already in state *Position Filled*, this switch would introduce an inconsistency that needs to be resolved. This can be facilitated by re-executing the lifecycle process instance of the *API Programmer* object instance. As shown in Algorithm 1, a lifecycle process instance must always notify the coordination process when a state change occurs. This is, however, just part of the regular execution of a lifecycle process. Furthermore, this is explicitly done when re-executing a lifecycle process as part of a coordination context change, to ensure that state transitions, which were allowed in the old coordination context, are still valid in the new one. To be more precise, for the example of the *Accepted Application* being removed from *Job Offer API Programmer*, the re-execution of the *Job Offer* would be blocked in a state before *Position Filled*, as the coordination process no longer has knowledge of an *Accepted Application* attached to *Job Offer API Programmer*. Once the coordination process has been notified and all affected object instances were

advanced or reverted into the appropriate states according to the process context changes, they were impacted by, process consistency is restored. Note that a change to the context of one object instance might have a cascading impact on others, requiring a re-execution or lifecycle advancement to restore consistency.

4.3 Enabling Context Changes to Multiple Object Instances

The final, and most complete, case of process context change is switching the process context of multiple object instances at the same time. We re-use the example of moving an *Application* object instance from one *Job Offer* to another. However, this time we assume that the new process context for the *Application* is a *Job Offer* in a different data model instance, albeit instantiated from the same data model. Furthermore, we employ the concept of dependent object instances presented in Sect. 3.3 to move the applicant, i.e. *Person a@exa.com*, his *Application*, and all other dependent object instances (*Reviews*, *Interviews*) in one atomic operation. Figure 11 shows the concrete example of moving *a@exa.com* to a different data model instance containing *Job Offer C++ Programmer*.

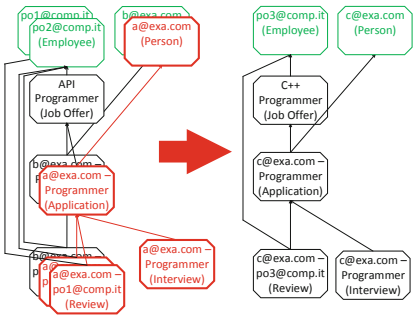


Fig. 11. Switching context of multiple object instances

While the process context changes in the previous examples were rather small in scope, e.g., consisting of the replacement of a single relation to a parent object instance by another, the above change is conducted in the scope of multiple object instances at the same time. Ensuring consistency before the change would require the user to determine replacement relations for all relation instances to objects not existing in the new data model instance. As an example, consider the relation between one of the *Review* object instances and the *Employee* assigned as a reviewer, e.g. *po2@comp.it* (cf. Fig. 11). In the new data model instance (i.e. the other department), *po2@comp.it* does not exist, causing the relation instance

between the *Review* and the *Employee* to be deleted. One way to solve this is to require the determination of replacement relations for each deleted relation, as previously suggested. Instead, once again, we leverage the flexible (re-)execution supported by object-aware lifecycle processes to elegantly solve this problem.

To be precise, we delete the relations to all objects not present in the new data model instance. Furthermore, we delete the attribute values referencing the relations, e.g. the *Job Offer* attribute in the *Application* object (cf. Fig. 6). Moreover, due to the presence of the *Job Offer* relation attribute as a step in the lifecycle process, an instance of the *Application* object must not progress past state *Created* without a value for *Job Offer* being provided. Coincidentally, a value for an attribute with the data type "relation" is provided by creating a relation to another object and vice versa. However, deleting the value of an attribute, once execution has progressed past the state it is required in, causes a lifecycle inconsistency. For example this happens when the *Applications* and *Reviews* are moved between the two data model instances, causing the relations to the no longer existing *Employees* and *Job Offers* to be deleted. If we trigger a re-execution of the lifecycle process instance of all object instances with now deleted relation instances (cf. Algorithm 1), the data-driven lifecycle process reacts by executing, for example, the *Application* object instance until the end of the *Created* state, and then waiting for user input.

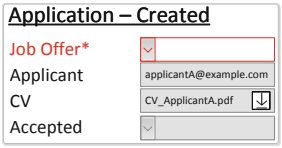


Fig. 12. *Application* after context switch

Here, the dynamic form generation capabilities (cf. Sect. 2) are utilized. After changing the process context, the *Application* is missing a *Job Offer*. The form shown in Fig. 12 is generated and added to the worklist of a personnel officer, allowing him to select the *C++ Programmer*. Once the *Job Offer* is selected, the data-driven lifecycle execution advances the *Application* object to its previous state. Similar forms are generated for both *Reviews*, and once all three forms are completed, process consistency is restored.

5 Prototypical Implementation and Real-World Use-Cases

In the PHILharmonicFlows implementation of object-aware process management, the higher level conceptual elements are implemented as microservices. For each object instance, relation instance, or coordination process instance, one microservice instance is created at runtime, turning the implementation into a fully distributed object-aware process management system.

Note that all the information from the various microservices can be utilized at runtime to generate an entire user interface, complete with navigation and form elements for a specific data model – a goal of the object-aware process management paradigm from the very beginning. A screenshot of the current user interface for end-users (with demo data) is shown in Fig. 13. The engine and user interface are currently evaluated in a large scale real-world deployment in the context of a course with hundreds of students, utilizing an object-aware process representing an e-learning platform called PHoodle (**PH**ILharmonicFlows M**oodle**)[1]. This study enables us to evaluate scalability, usability, and advanced concepts such as dynamic context switching. While, for the sake of brevity, we solely offered examples in the context of the recruitment data model, there are numerous scenarios from different domains in which the concepts from object-aware process management can be useful. We found multiple use-cases in our various data models, such as the re-assignment of a transport job from a robot to a human worker in our logistics research[2], or a student assigned to the wrong lecture in PHoodle, both powered by the PHILharmonicFlows engine.

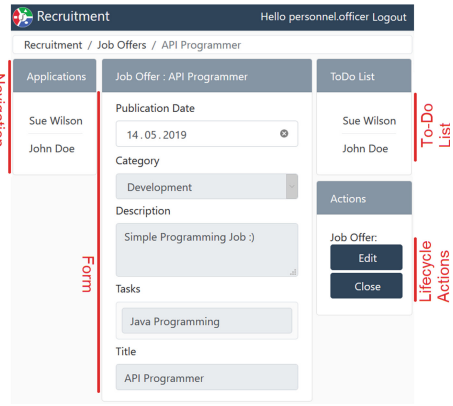


Fig. 13. PHILharmonicFlows UI

In the context of the real-world PHoodle deployment, we had cases of students wishing to switch their tutorials, while retaining access to completed worksheet submissions. As we had not thought of this possibility at design-time, it is a perfect use-case for context switching, as it may be employed to move an *Attendance* (representing a student), with all related *Submissions*, from one *Tutorial* to another. Our experience has shown that supporting this additional dimension of flexibility empowers users by offering them additional actions without increasing model complexity. For example, in PHoodle, we hide the complexity of reassigning a *Student* from one *Tutorial* to another by offering an *Employee* viewing object instance representing the *Student* a simple drop-down menu generated by the PHILharmonicFlows form logic for selecting a different *Tutorial*. When a *Tutorial* is selected, PHILharmonicFlows completes the process context change on all dependent object instances, deleting and creating relations, re-executing lifecycle processes, and updating the coordination process, in a single click.

[1] Feel free to log in to the live instance at https://phoodle.dbis.info Username: *edoc.demo@uni-ulm.*de Password: *edoc.demo*.

[2] https://www.youtube.com/watch?v=oGKjK7K76Ck.

6 Related Work

LateVa [6], enables automated late selection of process fragments based on process context. In essence, a process model does not define all possible variations but contains variation points that are replaced with process fragments at runtime depending on the process context. The actual replacement is done by the "fragment recommender", based on data mined from historical process instances.

The inclusion of "pockets of flexibility" into workflow models is proposed in [7]. Each pocket contains multiple individual process fragments that can be rearranged at runtime according to the needs of the process context, allowing for greater flexibility at certain points. CaPI (context-aware process injection) [8] analyses process context and allows injecting process fragments into extension areas. These fragments and the context in which they may be injected are determined at design-time using a sophisticated modeling tool.

The approaches presented in [6–8] follow a similar approach, limiting process context flexibility to predefined regions of a process model. Our approach aims to remove this limitation through relaxation. Instead of defining regions in which flexibility is possible, we allow for context changes except for in some situations.

Controlled evolution of process choreographies are examined in [9]. A process choreography describes the interactions of business processes with partner processes in a cross-organizational setting. [9] examines ways to gauge the impact of changes to processes with respect to their partner processes. This is a similar problem to the one examined in this paper, determining how to understand the impact that process context changes have on other object instances that are part of the same data model instance.

The concept of batch regions is introduced in [10]. A batch region is a part of a process model that may be executed in a single batch if there are other process instances available corresponding to the same context. Similar capabilities may be introduced to object-aware process management by extending the context switching concepts detailed in this paper to aggregate different object instances with similar contexts and executing them in batches.

Finally, [11] presents the context-oriented programming (COP) paradigm, which introduces a number of interesting aspects that could be incorporated into our future research. Combining our contribution with COP, which allows for objects in a programming language to behave differently depending on the context they are executed in, would be an interesting research direction. COP introduces *layering* for grouping behavioral variants of code with selectors that choose the correct variant after a context switch occurs at runtime. Similar notions could be used to extend the research presented in this paper.

7 Summary and Outlook

This paper used the running example of a recruitment data model to examine how process context can be freely switched and changed in object-aware processes. We presented a detailed examination of the notion of process context in object-aware processes, as well as the concepts and algorithms we developed to enable process context changes in our proof-of-concept implementation of an object-aware process management system – PHILharmonicFlows. In essence, we leverage the highly flexible execution provided by the object-aware process management paradigm to adapt running process instances to incurred process context changes. Some issues are still open, such as finding a generic solution for cases in which external services (e.g. payment services or e-mails) are used. Nonetheless, the presented concept constitutes an advancement for data-centric BPM. Together with our work on ad-hoc changes [4], the presented research brings us a step closer to a fully fledged data-centric process engine with which we can demonstrate the many flexibility advantages that data-centric processes have, thereby increasing the perceived maturity of data-centric BPM.

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-Wuerttemberg, Germany (F.No. 32-7545.24-17/3/1).

References

1. Saidani, O., Nurcan, S.: Context-awareness for business process modelling. In: 3rd International Conference on Research Challenges in Information Science, pp. 177–186. IEEE (2009)
2. Rosemann, M., Recker, J.C.: Context-aware process design. In: 18th International Conference on Advanced Information Systems Engineering (CAiSE) Workshops, pp. 149–158 (2006)
3. Steinau, S., Andrews, K., Reichert, M.: The relational process structure. In: Krogstie, J., Reijers, H.A. (eds.) CAiSE 2018. LNCS, vol. 10816, pp. 53–67. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91563-0_4
4. Andrews, K., Steinau, S., Reichert, M.: Enabling runtime flexibility in data-centric and data-driven process execution engines. Inf. Syst. (2019)
5. Müller, D., Reichert, M., Herbst, J.: Flexibility of data-driven process structures. In: Eder, J., Dustdar, S. (eds.) BPM 2006. LNCS, vol. 4103, pp. 181–192. Springer, Heidelberg (2006). https://doi.org/10.1007/11837862_19
6. Murguzur, A., Sagardui, G., Intxausti, K., Trujillo, S.: Process variability through automated late selection of fragments. In: Franch, X., Soffer, P. (eds.) CAiSE 2013. LNBIP, vol. 148, pp. 371–385. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38490-5_35
7. Sadiq, S., Sadiq, W., Orlowska, M.: Pockets of flexibility in workflow specification. In: S.Kunii, H., Jajodia, S., Sølvberg, A. (eds.) ER 2001. LNCS, vol. 2224, pp. 513–526. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45581-7_38
8. Mundbrod, N., Grambow, G., Kolb, J., Reichert, M.: Context-aware process injection. In: Debruyne, C., et al. (eds.) OTM 2015. LNCS, vol. 9415, pp. 127–145. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26148-5_8

9. Rinderle, S., Wombacher, A., Reichert, M.: On the controlled evolution of process choreographies. In: 22nd International Conference on Data Engineering, ICDE 2006, p. 124. IEEE (2006)
10. Pufahl, L., Meyer, A., Weske, M.: Batch regions: process instance synchronization based on data. In: 18th International Enterprise Distributed Object Computing Conference, pp. 150–159. IEEE (2014)
11. Hirschfeld, R., Costanza, P., Nierstrasz, O.M.: Context-oriented programming. J. Object Technol. **7**(3), 125–151 (2008)