Enabling Process Variants and Versions in Distributed Object-Aware Process Management Systems

Kevin Andrews, Sebastian Steinau, and Manfred Reichert

Institute of Databases and Information Systems, Ulm University, Germany
{firstname.lastname}@uni-ulm.de

Abstract. Business process variants are common in many enterprises and properly managing them is indispensable. Some process management suites already offer features to tackle the challenges of creating and updating multiple variants of a process. As opposed to the widespread activity-centric process modeling paradigm, however, there is little to no support for process variants in other process support paradigms, such as the recently proposed artifact-centric or object-aware process support paradigm. This paper presents concepts for supporting process variants in the object-aware process management paradigm. We offer insights into the distributed object-aware process management framework PHILharmonicFlows as well as the concepts it provides for implementing variants and versioning support based on *log propagation* and *log replay*. Finally, we examine the challenges that arise from the support of process variants and show how we solved these, thereby enabling future research into related fundamental aspects to further raise the maturity level of data-centric process support paradigms.

Keywords: business processes, process variants, object-aware processes

1 Introduction

Business process models are a popular method for companies to document their processes and the collaboration of the involved humans and IT resources. However, through globalization and the shift towards offering a growing number of products in a large number of countries, many companies are face a sharp increase of complexity in their business processes [4,5,11]. For example, automotive manufacturers that, years ago, only had to ensure that they had stable processes for building a few car models, now have to adhere to many regulations for different countries, the increasing customization wishes of customers, and far faster development and time-to-market cycles. With the addition of Industry 4.0 demands, such as process automation and data-driven manufacturing, it is becoming more important for companies to establish maintainable business processes that can be updated and rolled out across the entire enterprise as fast as possible.

However, the increase of possible process variants poses a challenge, as each additional constraint derived from regulations or product specifics either leads to larger process models or more process models showing different variants of otherwise identical processes. Both scenarios are not ideal, which is why there has been research over the past years into creating more maintainable process variants [5,7,11,13]. As previous research on process variant support has focused on activity-centric processes, our contribution provides a novel approach supporting *process variants in object-aware processes*. Similar to case handling or artifact-centric processes, object-aware processes are inherently more flexible than activity-centric ones, as they are less strictly structured, allowing for more freedom during process execution [1,3,6,10,12]. This allows object-aware processes to support processes that are very dynamic by nature and challenging to formulate in a sequence of activities in a traditional process model.

In addition to the conceptual challenges process variants pose in a centralized process server scenario, we examine how our approach contributes to managing the challenges of modeling and executing process variants on an architecture that can support scenarios with high scalability requirements. Finally, we explain how our approach can be used to enable updatable versioned process models, which will be essential for supporting schema evolution and ad-hoc changes in object-aware processes.

To help understand the notions presented in the contribution we provide the fundamentals of object-aware process management and process variants in Section 2. Section 3 examines the requirements identified for process variants. In Section 4 we present the concept for variants in object-aware processes as the main contribution of this paper. In Section 5 we evaluate whether our approach meets the identified requirements and discuss threats to validity as well as persisting challenges. Section 6 discusses related work, whereas Section 7 provides a summary and outlook on our plans to provide support for migrating running process instances to newer process model versions in object-aware processes.

2 Fundamentals

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 paper, has been under development for many years at Ulm University [2,8,9,16,17]. This section gives an overview on the PHILharmonicFlows concepts necessary to understand the remainder of the paper. PHILharmonicFlows takes the basic idea of a data-driven and data-centric process management system and improves it by introducing the concept of *objects*. One such object exists for each business object present in a real-world business process. As can be seen in Fig. 1, a PHILharmonicFlows object consists of data, in the form of *attributes*, and a state-based process model describing the *object lifecycle*.

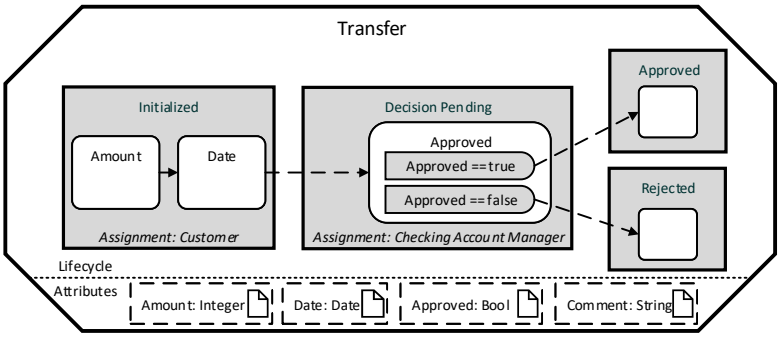


Fig. 1: Example Object Including Lifecycle Process

The attributes of the *Transfer* object (cf. Fig. 1) include *Amount*, *Date*, *Approval*, and *Comment*. The *lifecycle process*, in turn, describes the different *states (Initialized*, *Decision Pending*, *Approved*, and *Rejected)*, an instance of a *Transfer* object may have during process execution. Each state contains one or more *steps*, each referencing exactly one of the object attributes, thereby forcing that attribute to be written at run-time. The steps are connected by *transitions*, allowing them to be arranged in a sequence. The state of the object changes when all steps in a state are completed. Finally, alternative paths are supported in the form of *decision steps*, an example of which is the *Approved* decision step.

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

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

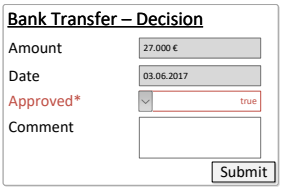
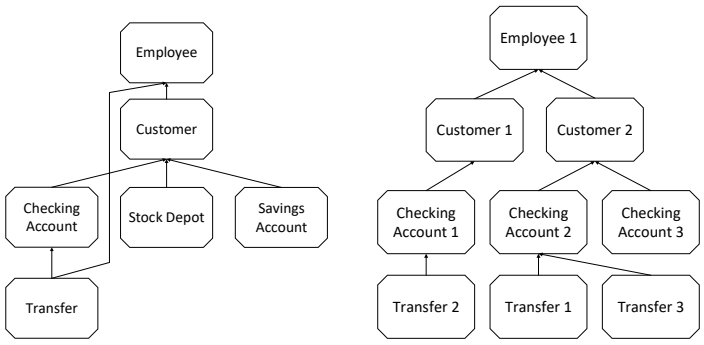


Fig. 2: Example Form

Note that a single object and its resulting form only constitutes one part of a complete PHILharmonicFlows process. To allow for complex executable business processes, many different objects and users may have to be involved [17]. It is noteworthy that *users* are simply special objects in the object-aware process management concept. The entire set of objects (including those representing

users) present in a PHILharmonicFlows process is denoted as the *data model*, an example of which can be seen in Fig. 3a. At run-time, each of the objects can be instantiated to so-called *object instances*, of which each represents a concrete instance of an object. The lifecycle processes present in the various object instances are executable concurrently at run-time, thereby improving performance. Fig. 3b shows a simplified example of an *instantiated data model* at run-time.



(a) Design-time (b) Run-time

Fig. 3: Data Model

In addition to the objects, the data model contains information about the *relations* existing between them. A relation constitutes a logical association between two objects, e.g., a relation between a *Transfer* and a *Checking Account*. Such a relation can be instantiated at run-time between two concrete object instances of a *Transfer* and a *Checking Account*, thereby associating the two object instances with each other. The resulting meta information, i.e., the information that the *Transfer* in question belongs to a certain *Checking Account*, can be used to coordinate the processing of the two objects with each other.

Finally, complex object coordination, which becomes necessary as most processes consist of numerous interacting business objects, is possible in PHILharmonicFlows as well [17]. As objects publicly advertise their state information, the current state of an object can be utilized as an abstraction to coordinate 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*.

The various components of PHILharmonicFlows, i.e., objects, relations, and coordination processes, are implemented as microservices, turning PHILharmonicFlows into a fully distributed process management system for object-aware processes. For each object instance, relation instance, or coordination process

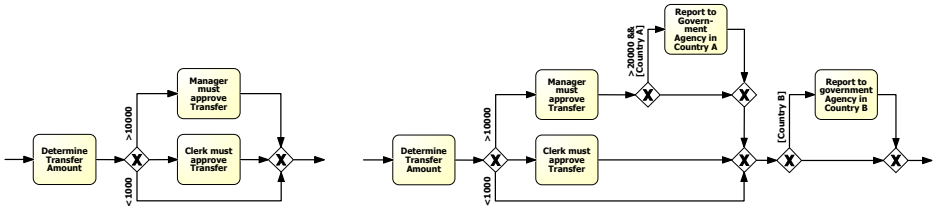
instance one microservice is present at run-time. Each microservice only holds data representing the attributes of its object. Furthermore, the microservice only executes the lifecycle process of the object it is assigned to. The only information visible outside the individual microservices is the current "state" of the object, which, in turn, is used by the microservice representing the coordination process to properly coordinate the objects' interactions with each other.

2.2 Process Variants

Simply speaking, a process variant is one specific path through the activities of a process model, i.e., if there are three distinct paths to completing a business goal, three process variants exist. As an example, take the process of transferring money from one bank account to another, for which there might be three alternate execution paths. For instance, if the amount to be transferred is greater than \$10,000, a manager must approve the transfer, if the amount is less than \$10,000, a mere clerk may approve said transfer. Finally, if the amount is less than \$1,000, no one needs to approve the transfer. This simple decision on who has to approve the transfer implicitly creates three variants of the process.

As previously stated, modeling such variants is mostly done by incorporating them into one process model as alternate paths via choices (cf. Fig. 4a). As demonstrated in the bank transfer example, this is often the only viable option, because the amount to be transferred is not known when the process starts. Clearly, for more complex processes, each additional choice increases the complexity of the process model, making it harder to maintain and update.

To demonstrate this, we extend our previous example of a bank transfer with the addition of country-specific legal requirements for money transfers between accounts. Assuming the bank operates in three countries, A, B, and C, country A imposes the additional legal requirement of having to report transfers over \$20,000 to a government agency. On the other hand, Country B could require the reporting of all transfers to a government agency, while country C has no such requirements. The resulting process model would now have to reflect all these additional constraints, making it substantially larger (cf. Fig. 4b).



(a) Base (b) Including extra Requirements

Fig. 4: Bank Transfer Process

Obviously, this new process model contains more information than necessary for its execution in one specific country. Luckily, if the information necessary to choose the correct process variant is available before starting the execution of the process, a different approach can be chosen: defining the various process variants as separate process models and choosing the right variant before starting the process execution. In our example this can be done as the country is known before the transfer process is started. Therefore, it is possible to create three country-specific process model variants, for countries A, B, and C, respectively. Consequently, each *process model variant* would only contain the additional constraints for that country not present in the *base process model*.

This reduces the complexity of the process model from the perspective of each country, but introduces the problem of having three different models to maintain and update. Specifically, changes that must be made to those parts of the model common to all variants, in our example the decision on who must approve the transfer, cause redundant work as there are now multiple process models that need updating. Minimizing these additional time-consuming workloads, while enabling clean variant-specific process models, is a challenge that many researchers and process management suites aim to solve [5,7,11,14,15].

3 Requirements

The requirements for supporting process variants in object-aware processes are derived from the requirements for supporting process variants in activity-centric processes, identified in our previous case studies and a literature review [5,7,11].

Requirement 1 *(Maintainability)* Enabling maintainability of process variants is paramount to variant management. Without advanced techniques, such as propagating changes made to a base process to its variants, optimizing a process would require changes in all individual variants, which is error-prone and time-consuming. To enable the features that improve maintainability, the base process and its variants must be structured as such (cf. Req. 2). Furthermore, process modelers must be informed if changes they apply to a base process introduce errors in the variants derived from them (cf. Req. 3).

Requirement 2 *(Hierarchical structuring)* As stated in Req. 1, a hierarchical structure becomes necessary between variants. Ideally, to further reduce workloads when optimizing and updating processes, the process variants of both lifecycle and coordination processes can be decomposed into further sub-variants. This allows those parts of the process that are shared among variants, but which are not part of the base process, to be maintained in an intermediate model.

Requirement 3 *(Error resolution)* As there could be countless variants, the system should report errors to process modelers automatically, as manual checking of all variants could be time-consuming. Additionally, to ease error resolution, the concept should allow for the generation of resolution suggestions. To be able to detect which variants would be adversely affected by a change to a base model, automatically verifiable correctness criteria are needed, leading to Req. 4.

Requirement 4 *(Correctness)* The correctness of a process model must be verifiable at both design- and run-time. This includes checking correctness before a pending change is applied in order to judge its effects. Additionally, the effects of a change on process model variants must be determinable to support Req. 5.

Requirement 5 *(Scalability)* Finally, most companies that need process variant management solutions maintain many process variants and often act globally. Therefore, the solutions for the above requirements should should be scalable, both in terms of computational complexity as well as in terms of the manpower necessary to apply them to a large number of variants. Additionally, as the PHILharmonicFlows architecture is fully distributed, we have to ensure that the developed algorithms work correctly in a distributed computing environment.

4 Variants and Versioning of Process Models

This section introduces our concepts for creating and managing different deployed versions and variants of data models as well as contained objects in an object-aware process management system. We start with the *deployment concept*, as the variant concept relies on many of the core notions presented here.

4.1 Versioning and Deployment using Logs

Versioning of process models is a trivial requirement for any process management system. Specifically, one must be able to separate the model currently being edited by process modelers from the one used to instantiate new process instances. This ensures that new process instances can always be spawned from a stable version of the model that no one is currently working on. This process is referred to as *deployment*. In the current PHILharmonicFlows implementation, deployment is achieved by copying an *editable data model*, thereby creating a *deployed data model*. The deployed data model, in turn, can then be instantiated and executed while process modelers keep updating the editable data model.

As it is necessary to ensure that already running process instances always have a corresponding deployed model, the deployed models have to be *versioned* upon deployment. This means that the deployment operation for an editable data model labeled "M" automatically creates a deployed data model M_{T38} (*Data Model M, Timestamp 38*). Timestamp $T38$ denotes the logical timestamp of the version to be deployed, derived from the amount of modeling actions that have been applied in total. At a later point, when the process modelers have updated the editable data model M and they deploy the new version, the deployment operation gets the logical timestamp for the deployment, i.e., $T42$, and creates the deployed data model M_{T42} (*Data Model M, Timestamp* 42). As M_{T38} and M_{T42} are copies of the editable model M at the moment (i.e., timestamp) of deployment, they can be instantiated and executed concurrently at run-time. In particular, process instances already created from M_{T38} should not be in conflict with newer instances created from M_{T42}.

The editable data model M, the two deployed models M_{T38} and M_{T42} as well as some instantiated models can be viewed in Fig. 5. The representation of each model in Fig. 5 contains the set of objects present in the model. For example, $\{X, Y\}$ denotes a model containing the two objects X and Y. Furthermore, the editable data model has a list of all modeling actions applied to it. For example, *L13:[+X]* represents the 13th modeling action, which added an object labeled "X". The modeling actions we use as examples throughout this section allow adding and removing entire objects. However, the concepts can be applied to any of the many different operations supported in PHILharmonicFlows, e.g., adding attributes or changing the coordination process.

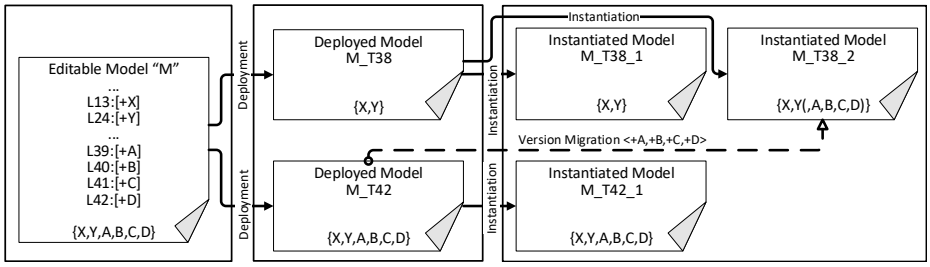


Fig. 5: Deployment Example

To reiterate, versioned deployment is a basic requirement for any process management system and constitutes a feature that most systems offer. However, we wanted to develop a concept that would, as a topic for future research, allow for the migration of already running processes to newer versions. Additionally, as we identified the need for process variants (cf. Sect. 1), we decided to tackle all three issues, i.e., *versioned deployment*, *variants*, and *version migration of running processes*, in one approach.

Deploying a data model by simply cloning it and incrementing its version number is not sufficient for enabling version migration. Version migration requires knowledge about the changes that need to be applied to instances running on a lower version to migrate it to the newest version, denoted as $M_{T38} \Delta M_{T42}$ in our example. In order to obtain this information elegantly, we log all actions a process modeler completes when creating the editable model until the first deployment. We denote these log entries belonging to M as $logs\,(M)$. To create the deployed model, we replay the individual log entries $l \in logs\,(M)$ to a new, empty, data model. As all modeling actions are deterministic, this recreates the data model M step by step, thereby creating the deployed copy, which we denote as M_{T38}. Additionally, as replaying the logs in $logs\,(M)$ causes each modeling action to be repeated, the deployment process causes the deployed data model M_{T38} to create its own set of logs,, $logs\,(M_{T38})$. Finally, as data model M remains editable after a deployment, additional log entries may be created and

added to $logs\,(M)$. Each consecutive deployment causes the creation of another deployed data model and set of logs, e.g. M_{T42} and $logs\,(M_{T42})$.

As the already deployed version, M_{T38} has its own set of logs, i.e., $logs\,(M_{T38})$, it is trivial to determine $M_{T38}\Delta M_{T42}$, as it is simply the *set difference*, i.e., $logs\,(M_{T42}) \setminus logs\,(M_{T38})$. As previously stated, $M_{T38}\Delta M_{T42}$ can be used later on to enable version migration, as it describes the necessary changes to instances of M_{T38} when migrating them to M_{T42}. An example of how we envision this concept functioning is given in in Fig. 5 for the migration of the instantiated model M_{T38_2} to the deployed model M_{T42}.

To enable this logging-based copying and deployment of a data model in a distributed computing environment, the log entries have to be fitted with additional meta information. As an example, consider the simple log entry l_{42} which was created after a user had added a new object type to the editable data model:

$$l_{42} = \begin{cases} dataModelId & 6123823241189 \\ action & AddObjectType \\ params & [name : "Object\,D"] \\ timestamp & 42 \end{cases}$$

Clearly, the log entry contains all information necessary for its replay: the id of the data model or object the logged action was applied to, the type of action that was logged, and the parameters of this action. However, due to the distributed microservice architecture PHILharmonicFlows is built upon, a logical timestamp for each log entry is required as well. This timestamp must be unique and sortable across all microservices that represent parts of one editable data model, i.e., all objects, relations, and coordination processes. This allows PHILharmonicFlows to gather the log entries from the individual microservices, order them in exactly the original sequence, and replay them to newly created microservices, thereby creating a deployed copy of the editable data model.

Coincidentally, it must be noted that the example log entry l_{42} is the one created before deployment of M_{T42}. By labeling the deployment based on the timestamp of the last log entry, determining the modeling actions that need to be applied to an instance of M_{T38} to update it to M_{T42} can be immediately identified as the sequence $\langle l_{39}, l_{40}, l_{41}, l_{42}\rangle \subset logs\,(M_{T42})$, as evidenced by the example in Fig. 5.

4.2 Variants

As previously stated, we propose reusing the logging concept presented in Sect. 4.1 for creating and updating variants of data models and contained objects. In Sect. 4.1, we introduced two example data models, M_{T38} and M_{T42}, which were deployed at different points in time from the same editable data model. Additionally, we showed that the differences between these two deployed models are the actions applied by four log entries, namely $\langle l_{39}, l_{40}, l_{41}, l_{42}\rangle$. Expanding upon this idea, we developed a concept for creating variants of data models using log entries for each modeling action, which we present in this section.

An example of our concept, in which two variants, $V1$ and $V2$, are created from the editable data model M, is shown in Fig. 6. The *editable base model*, M, has a sequence of modeling actions that were applied to it and logged in $\langle l_1, .., l_{42} \rangle$. Furthermore, the two variants of M where created at different points in time, i.e., at different logical timestamps. Variant $V1$ was created at timestamp $T39$, i.e., the last action applied before creating the variant had been logged in l_{39}.

As we reuse the deployment concept for variants, the actual creation of a *data model variant* is, at first, merely the creation of an identical copy of the editable data model in question. For variant $V1$, this means creating an empty editable data model and replaying the actions logged in the log entries $\langle l_1, .., l_{39} \rangle \subseteq logs\,(M)$, ending with the creation of object A. As replaying the logs to the new editable data model M_{V1} creates another set of logs, $logs\,(M_{V1})$, any further modeling actions that process modelers only apply to M_{V1} can be logged in $logs\,(M_{V1})$ instead of $logs\,(M)$. This allows us to add or remove elements not altered in the base model or other variants. An example is given by the removal of object A in $l_{40} \in logs\,(M_{V1})$, an action not present in $logs\,(M)$ or $logs\,(M_{V2})$.

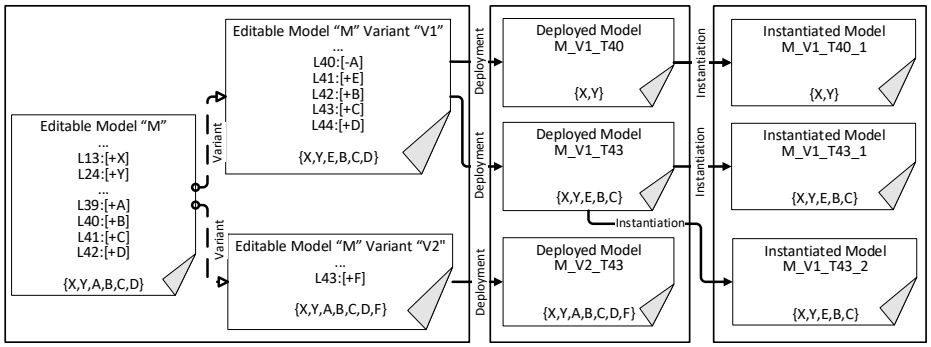


Fig. 6: Variant Example

Up until this point, a variant is nothing more than a copy that can be edited independently of the base model. However, in order to provide a solution for maintaining and updating process variants (cf. Req. 1), the concept must also support the automated propagation of changes made to the base model to each variant. To this end, we introduce a hierarchical relationship between editable models, as required by Req. 2, denoted by $\sqsubset$. In the example (cf. Fig. 6), both variants are beneath data model M in the variant hierarchy, i.e., $M_{V1} \sqsubset M$ and $M_{V2} \sqsubset M$. For possible sub-variants, such as $M_{V2_{V1}}$, the hierarchical relationship is *transitive*, i.e., $M_{V2_{V1}} \sqsubset M \iff M_{V2} \sqsubset M$.

To fulfill Req. 1 when modeling a variant, e.g. $M_{V1} \sqsubset M$, we utilize the hierarchical relationship to ensure that all modeling actions applied to M are propagated to M_{V1}, always ensuring that $logs\,(M_{V1}) \subseteq logs\,(M)$ holds. This is done by replaying new log entries added to $logs\,(M)$ to M_{V1}, which, in turn, creates new

log entries in $logs\,(M_{V1})$. As an example, Fig. 7 shows the replaying of one such log, $l_{40} \in logs(M)$ to M_{V1}, which creates log entry $l_{42} \in logs\,(M_{V1})$.

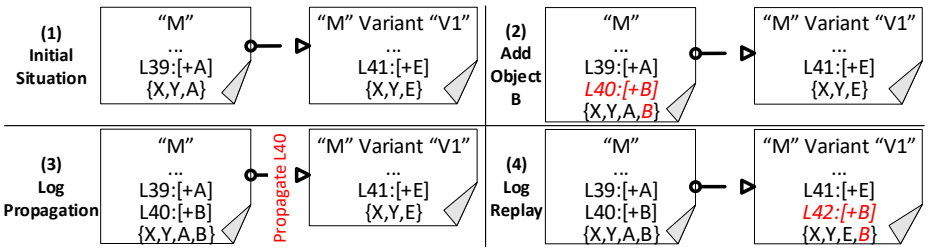


Fig. 7: Log Propagation Example

In the implementation, we realized this by making the propagation of the log entry for a specific modeling action part of the modeling action itself, thereby ensuring that updating the base model, including all variants, is atomic. However, it must be noted that, while the action being logged in both editable data models is the same, the logs have different timestamps. This is due to the fact that M_{V1} has the variant-specific log entries $\langle l_{40}, l_{41} \rangle \subset logs\,(M_{V1})$ and $l_{40} \in logs(M)$ is appended to the end of $logs\,(M_{V1})$ as $l_{42} \in logs\,(M_{V1})$. As evidenced by Fig. 6, variants created this way are fully compatible with the existing deployment and instantiation concept. In particular, from the viewpoint of the deployment concept, a variant is simply a normal editable model with its own set of logs that can be copied and replayed to a deployed model.

5 Evaluation

The presented concept covers all requirements (cf. Sect. 3) as we will show in the following. The main requirement, and goal of this research, was to develop a concept for maintainable process variants of object-aware data models and contained objects (cf. Req. 1). We managed to solve this challenge by introducing a strict hierarchical structure between the base data model, variants, and even sub-variants (cf. Req. 2). Furthermore, our solution ensures that the variants are always updated by changes made to their base models. As presented in Sect. 4.2, this is done by managing logs with logical timestamps and replaying them to variants that are lower in the hierarchy. This ensures that any modeling action applied to a variant will always take into consideration the current base model. However, this strict propagation of all modeling actions to all variants poses additional challenges. For instance, expanding on the situation presented in Fig. 6, a modeling action that changes part of the lifecycle process of object A could be logged as $l_{43} \in logs(M)$, causing the log to be replayed to variant $V1$. However, $V1$ does not have an object A anymore, as is evidenced by the set

of objects present, i.e., $\{X, Y, E, B, C, D\}$. Clearly, this is due to the fact that $l_{40} \in logs\,(M_{V1})$ removed object A from that variant.

As it is intentional for variant $V1$ to not comprise object A, this particular case poses no further challenge, as changes to an object not existing in a variant can be ignored by that variant. However, there are other scenarios to be considered, one of which is the application of modeling actions in a base model that have already been applied to a variant, such as introducing a transition between two steps in the lifecycle process of an object. If this transition already exists in a variant, the log replay to that variant will create an identical transition. As two transitions between the same steps are prohibited, this action would break the lifecycle process model of the variant and, in consequence, the entire object and data model it belongs to. A simplified example of the bank transfer object can be seen next to a variant with an additional transition between *Amount* and *Date* to showcase this issue in Fig. 8. The problem arises when trying to add a transition between *Amount* and *Date* to the base lifecycle process model, as the corresponding log entry gets propagated to the variant, causing a clash.

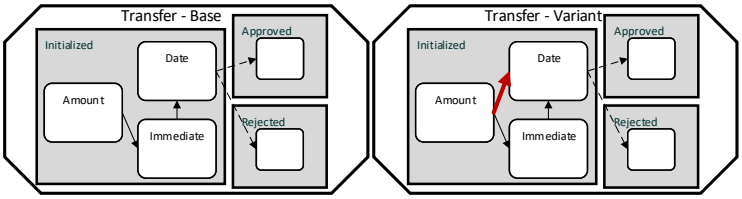


Fig. 8: Conflicting actions example

To address this and similar issues, which pose a *threat to validity* for our concept, we utilize the existing data model verification algorithm we implemented in the PHILharmonicFlows engine [16]. In particular, we leverage our distributed, micro-service based architecture to create clones of the parts of a variant that will be affected by a log entry awaiting application. In the example from Fig. 8, we can create a clone of the microservice serving the object, apply the log describing the transition between *Amount* and *Date,* and run our verification algorithm on the clone. This would detect any problem caused in a variant by a modeling action and generate an error message with resolution options, such as deleting the preexisting transition in the variant (cf. Reqs 3 and 4). In case there is no problem with the action, we apply it to the microservice of the original object. How the user interface handles the error message (e.g., offering users a decision on how to fix the problem) is out of the scope of this paper, but has been implemented and tested as a proof-of-concept for some of the possible errors.

All other concepts presented in this paper have been implemented and tested in the PHILharmonicFlows prototype. We have headless test cases simulating a multitude of users completing randomized modeling actions in parallel, as well as

around 50.000 lines of unit testing code, covering various aspects of the engine, including the model verification, which, as we just demonstrated, is central to ensuring that all model variants are correct. Furthermore, the basic mechanism used to support variants, i.e., the creation of data model copies using log entries, has been an integral part of the engine for over a year. As we rely heavily on it for deploying and instantiating versioned data models (cf. Sect. 4.1), it is utilized in every test case and, therefore, thoroughly tested.

Finally, through the use of the microservice-based architecture, we can ensure that time-consuming operations, such as verifying models for compatibility with actions caused by log propagation, are highly scalable and cannot cause bottlenecks [2]. This would hardly be an issue at design-time either way, but we are ensuring that this basis for our future research into run-time version migration, or even migration between variants, is highly scalable (cf. Req. 5). Furthermore, the preliminary benchmark results for the distributed PHILharmonicFlows engine, running on a cluster of 8 servers with 64 CPUs total, are promising. As copying data models using logs is central to the concepts presented in this paper, we benchmarked the procedure for various data model sizes (5, 7, and 14 objects) and quadrupling increments of concurrently created copies of each data model. The results in Table 1 show very good scalability for the creation of copies, as creating 64 copies only takes twice as long as creating one copy. The varying performance between models of only slightly different size can be attributes to the fact that some of the more complex modeling operations are not yet optimized.

Table 1: Results

Example Process	Objects	1 Copy	4 Copies	16 Copies	64 Copies
Recruitment	5	880ms	900ms	1120ms	2290ms
Intralogistics	7	2680ms	2830ms	4010ms	9750ms
Insurance	14	4180ms	4470ms	7260ms	12170ms

6 Related Work

Related work deals with modeling, updating, and managing of process variants in the activity-centric process modeling paradigm [5,7,11,13,15], as well as the management of large amounts of process versions [4].

The Provop approach [5] allows for flexible process configuration of large process variant collections. The activity-centric variants are derived from base processes by applying change operations. Only the set of change operations constituting the delta to the base process is saved for each variant, reducing the amount of redundant information. Provop further includes variant selection techniques that allow the correct variant of a process to be instantiated at run-time, based on the context the process is running in.

An approach allowing for the configuration of process models using questionnaires is presented in [13]. It builds upon concepts presented in [15], namely the

introduction of variation points in process models and modeling languages (e.g. C-EPC). A process model can be altered at these variation points before being instantiated, based on values gathered by the questionnaire. This capability has been integrated into the APROMORE toolset [14].

An approach enabling flexible business processes based on the combination of process models and business rules is presented in [7]. It allows generating ad-hoc process variants at run-time by ensuring that the variants adhere to the business rules, while taking the actual case data into consideration as well.

Focusing on the actual procedure of modeling process variants, [11] offers a decomposition-based modeling method for entire families of process variants. The procedure manages the trade-off between modeling multiple variants of a business process in one model and modeling them separately.

A versioning model for business processes that supports advanced capabilities is presented in [4]. The process model is decomposed into block fragments and persisted in a tree data structure, which allows versioned updates and branching on parts of the tree, utilizing the tree structure to determine affected parts of the process model. Unaffected parts of the tree can be shared across branches.

Our literature review has shown that there is interest in process variants and developing concepts for managing their complexity. However, existing research focuses on the activity-centric process management paradigm, making the current lack of process variant support in other paradigms, such as artifact- or data-centric, even more evident. With the presented research we close this gap.

7 Summary and Outlook

This paper focuses on the design-time aspects of managing data model variants in a distributed object-aware process management system. Firstly, we presented a mechanism for copying editable design-time data models to deployed run-time data models. This feature, by itself, could have been conceptualized and implemented in a number of different ways, but we strove to find a solution that meets the requirements for managing process variants as well. Secondly, we expanded upon the concepts created for versioned deployment to allow creating, updating, and maintaining data model variants. Finally, we showed how the concepts can be combined with our existing model verification tools to support additional requirements, such as error messages for affected variants.

There are still open issues, some of which have been solved for activity-centric process models, but likely require entirely new solutions for non-activity-centric processes. Specifically, one capability we intend to realize for object-aware processes is the ability to take the context in which a process will run into account when selecting a variant.

When developing the presented concepts, we kept future research into truly flexible process execution in mind. Specifically, we are currently in the process of implementing a prototypical extension to the current PHILharmonicFlows

engine that will allow us to upgrade instantiated data models to newer versions. This kind of version migration will allow us to fully support schema evolution. Additionally, we are expanding the error prevention techniques presented in our evaluation to allow for the verification of data model correctness for already instantiated data models at run-time. We plan to utilize this feature to enable ad-hoc changes of instantiated objects and data models, such as adding an attribute to one individual object instance without changing the deployed data model.

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. Van der Aalst, W.M.P., Weske, M., Grünbauer, D.: Case handling: a new paradigm for business process support. Data & Knowledge Engineering 53(2), 129–162 (2005)
2. Andrews, K., Steinau, S., Reichert, M.: Towards hyperscale process management. In: Proc EMISA. pp. 148–152 (2017)
3. Cohn, D., Hull, R.: Business artifacts: A data-centric approach to modeling business operations and processes. IEEE TCDE 32(3), 3–9 (2009)
4. Ekanayake, C.C., Rosa, M.L., ter Hofstede, A.H.M., Fauvet, M.C.: Fragment-based version management for repositories of business process models. In: On the Move to Meaningful Internet Sys: OTM, pp. 20–37 (2011)
5. Hallerbach, A., Bauer, T., Reichert, M.: Capturing variability in business process models: the provop approach. JSEP 22(6-7), 519–546 (2010)
6. Hull, R.: Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In: Proc WS-FM. pp. 1–24 (2010)
7. Kumar, A., Yao, W.: Design and management of flexible process variants using templates and rules. Computers in Industry 63(2), 112–130 (2012)
8. Künzle, V.: Object-aware process management. Ph.D. thesis, Ulm Univ (2013)
9. Künzle, V., Reichert, M.: PHILharmonicFlows: towards a framework for object-aware process management. JSME 23(4), 205–244 (2011)
10. Marin, M., Hull, R., Vaculín, R.: Data centric BPM and the emerging case management standard: A short survey. In: Proc BPM. pp. 24–30 (2012)
11. Milani, F., Dumas, M., Ahmed, N., Matulevičius, R.: Modelling families of business process variants: A decomposition driven method. Inf Sys 56, 55–72 (2016)
12. Reichert, M., Weber, B.: Enabling flexibility in process-aware information systems: challenges, methods, technologies. Springer Science & Business Media (2012)
13. Rosa, M.L., Dumas, M., ter Hofstede, A.H.M., Mendling, J.: Configurable multi-perspective business process models. Inf Sys 36(2), 313–340 (2011)
14. Rosa, M.L., Reijers, H.A., van der Aalst, W.M.P., Dijkman, R.M., Mendling, J., Dumas, M., García-Bañuelos, L.: APROMORE: An advanced process model repository. Expert Sys with Applications 38(6), 7029–7040 (2011)
15. Rosemann, M., van der Aalst, W.M.P.: A configurable reference modelling language. Inf Sys 32(1), 1–23 (2007)
16. Steinau, S., Andrews, K., Reichert, M.: A modeling tool for PHILharmonicFlows objects and lifecycle processes. In: Proc BPMD (2017)
17. Steinau, S., Künzle, V., Andrews, K., Reichert, M.: Coordinating business processes using semantic relationships. In: Proc CBI. pp. 143–152 (2017)