Updatable Process Views for User-centered Adaption of Large Process Models

Jens Kolb, Klaus Kammerer and Manfred Reichert

Institute of Databases and Information Systems
Ulm University, Germany
{jens.kolb,klaus.kammerer,manfred.reichert}@uni-ulm.de
http://www.uni-ulm.de/dbis

Abstract. The increasing adoption of process-aware information systems (PAISs) has resulted in large process model collections. To support users having different perspectives on these processes and related data, a PAIS should provide personalized views on process models. Existing PAISs, however, do not provide mechanisms for creating or even changing such process views. Especially, changing process models is a frequent use case in PAISs due to changing needs or unplanned situations. While process views have been used as abstractions for visualizing large process models, no work exists on how to change process models based on respective views. This paper presents an approach for changing large process models through updates of corresponding process views, while ensuring up-to-dateness and consistency of all other process views on the process model changed. Respective update operations can be applied to a process view and corresponding changes be correctly propagated to the underlying process model. Furthermore, all other views related to this process model are then migrated to the new version of the process model as well. Overall, our view framework enables domain experts to evolve large process models over time based on appropriate model abstractions.

1 Introduction

Process-aware information systems (PAISs) provide support for business processes at the operational level. A PAIS strictly separates process logic from application code, relying on explicit *process models*. This enables a separation of concerns, which is a well established principle in computer science to increase maintainability and to reduce costs of change [1]. The increasing adoption of PAISs has resulted in large process model collections. In turn, each process model may refer to different domains, organizational units and user groups, and comprise dozens or even hundreds of activities [2]. Usually, the different user groups need customized views on the process models relevant for them, enabling a personalized process abstraction and visualization [3]. For example, managers rather prefer an abstract process overview, whereas process participants need a detailed view of the process parts they are involved in. Hence, providing personalized process views is a much needed PAIS feature. Several approaches for

creating process model abstractions based on process views have been proposed [4,5,6]. However, these proposals focus on creating and visualizing views, but do not consider another fundamental aspect of modern PAISs: change and evolution [1,7]. More precisely, it is not possible to change a large process model through editing or updating one of its view-based abstractions. Hence, process changes must be directly applied to the core process model, which constitutes a complex as well as error-prone task for domain experts, particularly in connection with large process models. To overcome this limitation, in addition to view-based process abstractions, users should be allowed to change large process models through updating related process views.
In the *proView*[1] project, we address these challenges by not only supporting the creation and visualization of process views, but additionally providing change operations enabling users to modify a process model through updating a related process view. In this context, all other views associated with the changed process model are migrated to its new version as well. Besides view-based abstractions and changes, *proView* allows for alternative process model representations (e.g., tree-based, form-based, and diagram-based) as well as interaction techniques (e.g., gesture- vs. menu-based) [8,9,10]. Overall goal is to enable domain experts to "interact" with the (executable) process models they are involved in.

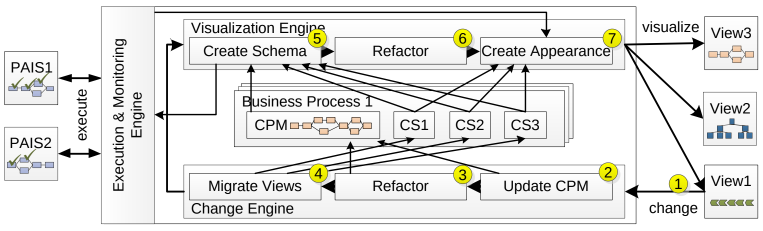


Fig. 1. The proView Framework

Fig. 1 gives an overview of the *proView* framework: A *business process* is captured and represented through a *Central Process Model (CPM)*. In addition, for a particular CPM, so-called *creation sets (CS)* are defined. Each creation set specifies the schema and appearance of a particular process view. For defining, visualizing, and updating process views, the *proView* framework provides engines for *visualization*, *change*, and *execution & monitoring*.
The *visualization engine* generates a process view based on a given CPM and the information maintained in a creation set CS, i.e., the CPM schema is transformed to the view schema by applying the corresponding *view creation operations* specified in CS (Step ⑤). Afterwards, the resulting view schema is *simplified* by applying well-defined *refactoring operations* (Step ⑥). Finally, Step ⑦ customizes the visual appearance of the view (e.g., creating a tree-, form-, or activity-based visualization [5,8]). Section 2 provides insights into these steps.

[1] http://www.dbis.info/proView

When a user updates a view schema, the *change engine* is triggered (Step ①). First, the view-based model change is propagated to the related CPM using well-defined change propagation algorithms (Step ②). Next, the schema of the CPM is simplified (Step ③), i.e., behaviour-preserving refactorings are applied to foster model comprehensibility (e.g., by removing surrounding gateways not needed anymore). Afterwards, the creation sets of all other views associated with the CPM are migrated to the new CPM schema version (Step ④). This becomes necessary since a creation set may be contradicting with the changed CPM schema. Finally, all views are recreated (Steps ⑤-⑦) and presented to users. Section 3 presents the view update operations and migration rules required to change business processes through editing and updating process views. Section 4 then discusses related work and Section 5 summarizes the paper.

2 Fundamentals on Process View Creation

Section 2.1 defines the notion of *process model* and useful functions. Section 2.2 then discusses how *process views* can be created and formally represented in *proView* (i.e., Step ⑤, Fig. 1). Section 2.3 introduces behaviour-preserving process model refactorings enabling *lean* and *comprehensible* process views.

2.1 Process Model

A process model is represented by a *process schema* consisting of *process nodes* and the *control flow* between them (cf. Fig. 2). For control flow modeling, *gateways* and *control flow edges* are used (cf. Definition 1).

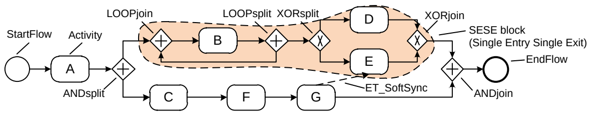


Fig. 2. Example of a Process Model

Definition 1 (Process Model). *A process model is defined as a tuple* $P = (N, E, EC, NT, ET)$ *where*

- N *is a set of nodes (i.e., activities and gateways),*
- $E \subset N \times N$ *is a precedence relation* ($e = (n_{src}, n_{dest}) \in E$ *with* $n_{src} \neq n_{dest}$*),*
- $EC : E \rightarrow Conds \cup \{\text{TRUE}\}$ *assigns transition conditions to control edges,*
- $NT : N \rightarrow \{StartFlow, EndFlow, Activity, ANDsplit, ANDjoin, XORsplit, XORjoin, LOOPsplit, LOOPjoin\}$ *assigns a node type* $NT(n)$ *to each node* $n \in N$*;* N *is divided into disjoint sets of activity nodes* A ($NT = Activity$) *and gateways* S ($NT \neq Activity$)*, i.e.,* $N = A \cup S$*, and*
- $ET : E \rightarrow \{ET\_Control, ET\_SoftSync, ET\_Loop\}$ *assigns a type* $ET(e)$ *to each edge* $e \in E$.

Definition 1 can be used for representing the schemas of both the *Central Process Model (CPM)* and its associated *process views*. Note that this definition focuses on control flow. In particular, it can be applied to existing activity-oriented modeling languages, but is not restricted to a specific one. This paper uses BPMN as notation due to its widespread use. We further assume that a process schema is *well-structured*, i.e., sequences, branchings (of different semantics), and loops are specified as blocks with well-defined start and end nodes having the same gateway type. These blocks—also known as SESE blocks (cf. Definition 2)—may be arbitrarily nested, but must not overlap (like, e.g., blocks in BPEL). To increase expressiveness, *sync edges* are supported, which allow for a *cross-block* synchronization of parallel activities (as BPEL links do). For example, in Fig. 2, activity E must not be enabled before G is completed.

Definition 2 (SESE). *Let $P = (N, E, EC, NT, ET)$ be a process model and $X \subseteq N$ be a subset of activity nodes (i.e., $NT(n) = Activity,\ \forall n \in X$). Then: Subgraph P' induced by X is called* SESE *(Single Entry Single Exit) block iff P' is connected and has exactly one incoming and one outgoing edge connecting it with P. Further, let $(n_s, n_e) \equiv MinimalSESE(P, X)$ denote the start and end node of the minimum SESE comprising all activities from $X \subseteq N$.*

How to determine SESE blocks is described in [11]. Since we presume a well-structured process schema, a minimum SESE can be always determined.
To determine the predecessor and successor of a single node or SESE block within a process model $P = (N, E, EC, NT, ET)$, operations $n_p = pred(P, N^{'})$ and $n_s = succ(P, N^{'})$ with $N^{'} \subseteq N$ are provided. Thereby n_p is the only node having exactly one outgoing edge $e_p = (n_p, n) \in E,\ n \in N^{'}$. In turn, since $N^{'}$ represents a SESE, e_p is the only incoming edge of any node in $N^{'}$ connecting it with P. Similarly, $succ(P, N^{'})$ returns the node directly succeeding set $N^{'}$.

2.2 Process View Creation

To create a process view on a process model, the latter has to be *abstracted*. For this, *proView* provides *elementary view creation operations*. In turn, these may be combined to realize *high-level view creation operations* (e.g., *show all my activities and their precedence relation*) in order to support users in creating process views easily [12]. At the elementary level, two categories of operations are required: *reduction* and *aggregation*. An elementary *reduction* operation hides an activity of the original process model in the process view created. For example, operation $RedActivity(V, n)$ removes node n together with its incoming and outgoing edges, and inserts a new edge linking the predecessor of n with its successor in view V (cf. Fig. 3a). A formal definition can be found in [12,13].
An *aggregation* operation, in turn, takes a set of activities as input and combines them into an abstracted node in the process view. For example, operation $AggrSESE(V, N^{'})$ removes all nodes of the SESE block, containing activities from set $N^{'}$ (including their edges), and inserts an abstract activity in the resulting process view instead (cf. Fig. 3b). Furthermore, elementary operation

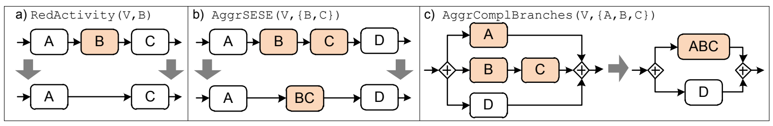


Fig. 3. Examples of Process View Creation Operations

$AggrComplBranches(V, N^{'})$ aggregates complete branches of an XOR/AND block to a branch with one abstracted node. $N^{'}$ must contain the activities of these branches (i.e., activities between split and corresponding join gateway) that shall be replaced by a single branch with one aggregated node (cf. Fig. 3c). Generally, a process view can be created through the consecutive application of elementary operations on a process model. Remember that this process model represents a particular business process and is denoted as *Central Process Model (CPM)*. A particular CPM may have several associated views. Note that the presented view operations consider other process perspectives (e.g., data elements and data flow) as well; due to lack of space we omit further details.

Definition 3 (Process View). *Let CPM be a process model. A process view V(CPM) is described through a creation set* $CS_V = (CPM, Op, PS)$ *with:*

– $CPM = (N, E, EC, NT, ET)$ *is the process model underlying the view and denoted as Central Process Model,*
– $Op = \langle Op_1, \ldots, Op_n \rangle$ *is the sequence of elementary view creation operations applied to CPM:* $Op_i \in \{RedActivity, AggrSESE, AggrComplBranches\}$,
– $PS = (PS_1, \ldots, PS_m)$ *is a tuple of parameters and corresponding parameter values defined for a specific view.*

Definition 3 expresses that a process view can be created through the consecutive application of the operations contained in the corresponding creation set. In this context, configuration parameters (shortly: *parameter*) are required to describe how high-level operations shall be mapped to elementary view creation operations depending on the selected nodes in the CPM (see [12] for details). Section 3 will show that these parameters are required to enable automatic change propagation from a view to its underlying CPM.
A *view node n* either directly corresponds to node n of the CPM or it abstracts a set of CPM nodes. $CPMNode(V, n)$ reflects this by returning either node n or a node set N_n of $CPM = (N, E, EC, NT, ET)$, depending on the creation set $CS_V = (CPM, Op, PS)$ *with* $Op = \langle Op_1, \ldots, Op_k \rangle$.

$$CPMNode(V, n) = \begin{cases} n & n \in N \\ N_n & \exists Op_i \in Op : N_n \xrightarrow{Op_i} n \end{cases}$$

2.3 Refactoring Operations

When creating process views, unnecessary control flow structures might result due to the generic nature of the view creation operations applied, e.g., single branches of a parallel branching might be empty or a parallel branching only have one remaining branch. In such cases, gateways can be removed in order to obtain a more comprehensible schema of the process view. For example, the view in Fig. 4a is created by reducing activity B. Afterwards, an AND block only surrounding activity C remains. In this case, the surrounding AND gateways can be removed without losing the predecessor/successor relations of the view activities (i.e., behaviour is preserved). Fig. 4b shows another example reducing activity B within a sequence. Afterwards, the synchronizing relationships become obsolete and hence can be removed. Fig. 4c shows an example of nested AND gateways which may be combined to simplify the model.

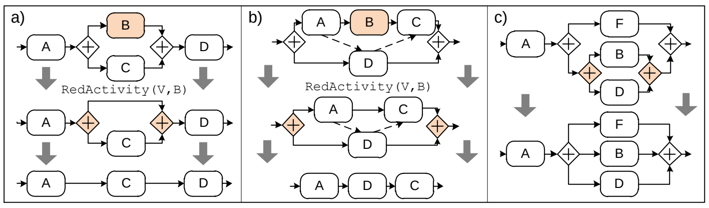


Fig. 4. Examples of View Refactoring Operations

The *proView* framework offers a set of operations for refactoring the schema of process views, without affecting the dependencies of activities within the view and hence without changing behavioural semantics [13].

3 Changing Processes through Updatable Process Views

Process views are not only required for enabling personalized process visualization through abstracting the underlying CPM. They also shall provide the basis for changing large process models based on appropriate abstractions. Section 3.1 describes how updates of a process view can be accomplished and then propagated to the underlying CPM. Section 3.2 presents migration rules for updating all other process views associated with the changed CPM as well.

3.1 Updating Process Views

When allowing users to change a business process model based on a personalized process view, it has to be ensured that this change can be automatically propagated to the underlying CPM without causing syntactical or semantical errors. Hence, well-defined view update operations are required guaranteeing for a proper propagation of view updates to the corresponding CPM. Table 1 gives

Operation	Parameter & Value	Description
$InsertSerial(V, n_1, n_2, n_{new})$	InsertSerialMode = { **EARLY**, LATE, PARALLEL}	Inserts activity n_{new} between n_1 and n_2 in view V. The parameter describes the propagation behaviour of this insertion.
$InsertParallel(V, n_1, n_2, n_{new})$ $InsertCond(V, n_1, n_2, n_{new}, c)$ $InsertLoop(V, n_1, n_2, n_{new}, c)$	InsertBlockMode = { **EARLY_EARLY**, EARLY_LATE, LATE_EARLY, LATE_EARLY}	Inserts activity n_{new} as well as an AND/ XOR/Loop block surrounding the SESE block defined by n_1 and n_2 in view V. The first (last) part of the parameter value before (after) the underline specifies the propagation behaviour of the split (join) gateway.
$InsertBranch(V, g_1, g_2, c)$	InsertBranchMode = { **EARLY**, LATE}	Inserts an empty branch between split gateway g_1 and join gateway g_2 in view V. In case of conditional branchings or loops, a condition c is required.
$InsertSyncEdge(V, n_1, n_2)$	-	Inserts a sync edge from n_1 to n_2 in V, where n_1 and n_2 belonging to different branches of a parallel branching.
$DeleteActivity(V, n_1)$	DeleteActivityMode = { **LOCAL**, GLOBAL}	Deletes activity n_1 in view V. The parameter decides whether the activity is deleted *locally* (i.e., reduced in the view) or removed from the CPM (i.e., *global*).
$DeleteBranch(V, g_1, g_2)$	-	Deletes an empty branch between gateways g_1 and g_2 in view V.
$DeleteSyncEdge(V, n_1, n_2)$	-	Deletes a sync edge between activities n_1 and n_2 in view V.
$DeleteBlock(V, g_1, g_2)$	DeleteBlockMode={ **INLINE**, DELETE}	Deletes an AND/XOR/Loop block enclosed by gateways g_1 and g_2 in view V. The parameter describes whether elements remaining in the block shall be *inlined* or *deleted*.

Table 1. Update Operations for Process Views

an overview of the view update operations supported by *proView*.

Propagating view changes to the underlying CPM is not straightforward. In certain cases, there might be ambiguities regarding the propagation of the view change to the underlying CPM. For example, it might not be possible to determine a unique position for inserting an activity in the CPM due to the abstractions applied when creating the view (cf. Fig. 5).
Consider the example from Fig. 5. Inserting activity Y in view $V1$ and propagating this change to the underlying CPM results in a unique insert position, i.e., this view update can be automatically propagated to the CPM without need for resolving any ambiguity. By contrast, inserting activity X in view $V1$ allows for several insert positions in the related CPM. More precisely, there are ambiguities in how to transform the view change into a corresponding CPM change, i.e., X may be inserted directly after activity A or directly before activity C. Note that this ambiguity is a consequence of the reduction (i.e., deletion of B) applied when creating the view. However, when propagating view updates to a CPM, users should not be burdened with resolving such ambiguities. Hence, to enable automated propagation of view updates to a CPM, *proView* supports parameterizable propagation policies. Hereafter, we introduce parameterizable view update operations that can be configured differently to automatically propagate view updates to a CPM resolving ambiguities if required (cf. Table 1).

```
Algorithm 1: InsertSerial($\mathbf{V, n_1, n_2, n_{new}}$)
Pre  $n_1^{'} = last(CPMNode(V, n_1)),\ n_2^{'} = first(CPMNode(V, n_2))$
Post $if(succ(CPM, n_1^{'}) == n_2^{'})$
        $InsertNode(CPM, n_1^{'}, n_2^{'}, n_{new}, Activity)$
     $else\ switch(InsertSerialMode):$
        $EARLY: InsertNode(CPM, n_1^{'}, succ(CPM, \{n_1^{'}\}), n_{new}, Activity)$
        $LATE: InsertNode(CPM, pred(CPM, \{n_2^{'}\}), n_2^{'}, n_{new}, Activity)$
        $PARALLEL: (n_s, n_j) = MinimalSESE(CPM, \{n_1^{'}, n_2^{'}\})$
                $InsertNode(CPM, pred(CPM, \{n_s\}), n_s, g_s, ANDsplit)$
                $InsertNode(CPM, n_j, succ(CPM, \{n_j\}), g_j, ANDjoin)$
                $InsertEdge(CPM, g_s, g_j, ET\_Control)$
                $InsertNode(CPM, g_s, g_j, n_{new}, Activity)$
                $InsertEdge(CPM, n_1^{'}, n_{new}, ET\_SoftSync)$
                $InsertEdge(CPM, n_{new}, n_2^{'}, ET\_SoftSync)$
```
Table 2. View Update Operation: InsertSerial

For example, consider view update operation *InsertSerial* in Fig. 5. Here, parameter *InsertSerialMode* defines whether X is inserted directly after A (i.e., *InsertSerialMode=EARLY*) or directly before C (i.e., *InsertSerialMode=LATE*). Each configuration parameter has a default value (printed in bold in Table 1), but can be set specifically for any view and stored in *parameter set PS* of creation set CS (cf. Section 2.2). We exemplarily provide algorithms for operations *InsertSerial* and *InsertParallel* to indicate how a view change can be transformed into a corresponding CPM change taking such parameterizations into account.

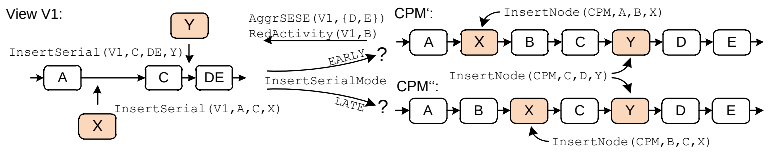


Fig. 5. Ambiguity when Propagating View Changes to the CPM

InsertSerial. As shown in Fig. 5, $InsertSerial(V, n_1, n_2, n_{new})$ adds an activity to the schema of process view V. Activity n_1 describes the node directly *preceding* and n_2 the node directly *succeeding* the activity n_{new} to be added to process view V. Algorithm 1 (cf. Table 2) shows how a view change described by operation *InsertSerial* can be transformed into a schema change of the related CPM. First of all, the nodes of the CPM corresponding to n_1 and n_2 are determined. If one of these nodes is an aggregated one, *CPMNode* returns a set of nodes. In this case, *first/last* returns the first/last node within this set (regarding control flow). When applying this change, it is checked whether nodes $n_1^{'}$ and $n_2^{'}$ (i.e., corresponding CPM nodes of n_1 and n_2) are direct neighbours. In this case, n_{new} can be directly inserted between these two nodes by applying the basic change operations *InsertNode* and *InsertEdge* to the CPM (cf. Table 3). In turn, if $n_1^{'}$ and $n_2^{'}$ are no direct neighbours in the CPM[2], it must be decided where to insert the activity, taking the value of parameter *InsertSerialMode* into account. As shown in Table 2, when setting this parameter to *EARLY*, the activity is directly inserted after $n_1^{'}$ (cf. Table 3). In turn, when choosing value *LATE*, it is inserted directly before $n_2^{'}$. Finally, parameter value *PARALLEL* determines the minimum SESE block containing activities $n_1^{'}$ and $n_2^{'}$. This is followed by adding an AND block surrounding the SESE block. The latter is accomplished by adding an *ANDsplit* and *ANDjoin* gateway as well as an empty branch between them. Finally, n_{new} is added to this empty branch. To ensure that the same precedence relations as for the process view are obeyed, sync edges from $n_1^{'}$ to n_{new} and from n_{new} to $n_2^{'}$ are inserted as well.

Algorithm 2: InsertNode($\mathbf{P}, \mathbf{n_1}, \mathbf{n_2}, \mathbf{n_{new}}$, node_type)	
Pre	$succ(P, n_1) = n_2, \{n_1, n_2\} \subseteq N,\ P = (N, E, EC, NT, ET)$
Post	$NT(n_{new}) = node\_type$
	$N^{'} = N \cup \{n_{new}\}$
	$e_1 = (n_1, n_{new}), e_2 = (n_{new}, n_2)\ with\ ET(e_1) = ET(e_2) = ET\_Control$
	$E^{'} = E \setminus \{(n_1, n_2)\} \cup \{e_1, e_2\}$
Algorithm 3: InsertEdge($\mathbf{P}, \mathbf{n_1}, \mathbf{n_2}$, edge_type)	
Pre	$\{n_1, n_2\} \subseteq N,\ P = (N, E, EC, NT, ET)$
Post	$e_{new} = (n_1, n_2), ET(e_{new}) = edge\_type$
	$E^{'} = E \cup \{e_{new}\}$

Table 3. Basic Process Model Change Operations

We now show that the transformation of a view update (as defined by *InsertSerial*) to a corresponding change of the underlying CPM, followed by the recreation of this view, results in the same view schema as one obtains when directly inserting this activity in the view. We consider this as a fundamental quality property of our view update propagation approach. For this purpose, we introduce the notion of *dependency set* (cf. Definition 4).

Definition 4 (Dependency Set). *Let $P = (N, E, EC, NT, ET)$ be a process model. Then: $\mathbb{D}_P = \{(n_1, n_2) \in N \times N | n_1 \preceq n_2, NT(n_1) = NT(n_2) = Activity\}$ is denoted as dependency set. It reflects all direct control flow dependencies between two activities.*

For example, the dependency set of the *CPM* depicted in Fig. 5 is $\mathbb{D}_{CPM'} = \{(A, X), (X, B), (B, C), (C, Y), (Y, D), (D, E)\}$.

Theorem *(InsertSerial Equivalence) Let CPM be a central process model and $\mathbb{D}_{CPM}$ be the corresponding dependency set. Further, let V be a view on CPM with creation set $CS_V = (CPM, Op, PS)$ and corresponding dependency set $\mathbb{D}_V$. Then: Inserting n_{new} in V can be realized by applying $InsertSerial(V, n_1, n_3, n_{new})$. Concerning the dependency set, propagating this change operation to the CPM results in the same view schema than one obtains when inserting n_{new} directly in V.*

[2] e.g., when creating the view, the CPM might have been reduced by deleting activities or gateways due to refactorings of the view schema

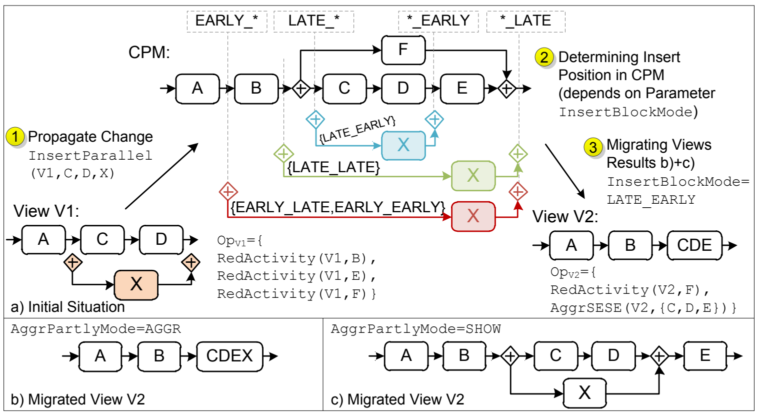
As shown, *RedActivity* and related refactorings may cause ambiguities. Hence, their influence on the dependency set has to be discussed. Applying *RedActivity* (V, n_2) with $(n^{'}, n_2), (n_2, n^{''}) \in E$ to a process schema with dependency set $\mathbb{D}$ results in $\mathbb{D}^{'} = \mathbb{D} \setminus \{(n^{'}, n_2), (n_2, n^{''})\} \cup \{(n^{'}, n^{''})\},\ n^{'}, n^{''} \in N$.

Proof: Inserting n_{new} directly in view V results in dependency set $\mathbb{D}^{'}_V = \mathbb{D}_V \cup \{(n_1, n_{new}), (n_{new}, n_3)\} \setminus \{(n_1, n_3)\}$. When inserting n_{new} in the CPM, we have to distinguish four cases:
Case 1: No activity is reduced between n_1 and n_3, i.e., no parameter is required and $\mathbb{D}^{'}_{CPM} = \mathbb{D}_{CPM} \cup \{(n_1, n_{new}), (n_{new}, n_3)\} \setminus \{(n_1, n_3)\} = \mathbb{D}^{'}_V$.
Case 2-4: An activity (activity set) is reduced between n_1 and n_3, i.e., ambiguities occur and parameter *InsertSerialMode* becomes relevant.
Case 2: InsertSerialMode=EARLY results in $\mathbb{D}^{'}_{CPM} = \mathbb{D}_{CPM} \cup \{(n_1, n_{new}), (n_{new}, n_2)\} \setminus \{(n_1, n_2)\}$ and $RedActivity(n_2) \in Op$ with $\{(n_1, n_2), (n_2, n_3)\} \subset \mathbb{D}_{CPM}$. Without loss of generality, we may assume that just one activity is reduced between n_1 and n_3. Next, view V is recreated with $RedActivity(n_2)$; this results in $\mathbb{D}^{''}_V = \mathbb{D}^{'}_{CPM} \setminus \{(n_{new}, n_2), (n_2, n_3)\} \cup \{(n_{new}, n_3)\} = \mathbb{D}_{CPM} \cup \{(n_1, n_{new}), (n_{new}, n_2)\} \setminus \{(n_1, n_2)\} \setminus \{(n_{new}, n_2), (n_2, n_3)\} \cup \{(n_{new}, n_3)\} = \mathbb{D}^{'}_V$.
Case 3: InsertSerialMode=LATE: similar to EARLY, whereby n_{new} is inserted directly before n_3.
Case 4: InsertSerialMode=PARALLEL: results in $\mathbb{D}^{'}_{CPM} = \mathbb{D}_{CPM} \cup \{(n_1, n_{new}), (n_{new}, n_3)\}$ and $RedActivity(n_2) \in Op$ with $\{(n_1, n_2), (n_2, n_3)\} \subset \mathbb{D}_{CPM}$. Next, V is recreated with $RedActivity(n_2)$; this results in $\mathbb{D}^{''}_V = \mathbb{D}^{'}_{CPM} \setminus \{(n_1, n_2), (n_2, n_3)\} \cup \{(n_1, n_3)\}$. At this point, the parallel branching is still remaining in the graph, i.e., one branch containing n_{new} and an empty branch due to reductions. Finally, refactorings remove unnecessary branchings: $\mathbb{D}^{'''}_V = \mathbb{D}^{''}_V \setminus \{(n_1, n_3)\} = \mathbb{D}^{'}_V$.

InsertParallel. When inserting an activity in parallel to existing activities by applying *InsertParallel* to a view, again the transformation of this change to a corresponding CPM change might raise ambiguities regarding the positions the ANDsplit and ANDjoin gateways shall be inserted. Fig. 6a illustrates this. To deal with this ambiguity, parameter *InsertBlockMode* must be set. It allows configuring the positions at which the ANDsplit (i.e., $EARLY\_*$, $LATE\_*$) and ANDjoin respectively (i.e., $*\_EARLY$, $*\_LATE$) shall be inserted.

```
Algorithm 4: InsertParallel(V, n_1, n_2, n_new)
  Pre   $(n_1, n_2)$ is SESE in view $V$
        $n_1^{'} = last(CPMNode(V, n_1)),\ n_2^{'} = first(CPMNode(V, n_2))$
  Post  $if(pred(V, last(CPMNode(V, n_1)))! = last(CPMNode(V, pred(V, n_1))))$
          $switch(InsertBlockMode)$
            $EARLY\_*:\ n_1^{'} = succ(CPM, CPMNode(V, pred(n_1)))$
            $LATE\_*:\ n_1^{'} = last(CPMNode(V, n_1))$
        $if(pred(V, last(CPMNode(V, n_2)))! = last(CPMNode(V, pred(V, n_2))))$
          $switch(InsertBlockMode)$
            $*\_EARLY:\ n_2^{'} = first(CPMNode(V, n_2))$
            $*\_LATE:\ n_2^{'} = succ(CPM, CPMNode(V, pred(V, n_2)))$
        $(n_s, n_j) = MinimalSESE(CPM, \{n_1^{'}, n_2^{'}\})$
        $InsertNode(CPM, pred(CPM, \{n_s\}), n_s, g_s, ANDsplit)$
        $InsertNode(CPM, n_j, succ(CPM, \{n_j\}), g_j, ANDjoin)$
        $InsertEdge(CPM, g_s, g_j, ET\_Control)$
        $InsertNode(CPM, g_s, g_j, n_{new}, Activity)$
```
Table 4. View Update Operation: InsertParallel

Table 4 provides a detailed view of the *InsertParallel* operation: n_1/n_2 denotes the start/end of the SESE block to which activity n_{new} shall be added in parallel. When transforming this view update to a corresponding CPM change, it must be decided where to add the ANDsplit and the ANDjoin gateways in case of ambiguities. Regarding the ANDsplit, for example, it is checked whether the direct predecessor of n_1 in the CPM is the same as in view schema V. If this is not the case, parameter $InsertBlockMode$ is used to decide whether to position the ANDsplit at the earliest or latest possible location in the CPM. The same procedure is applied in respect to the ANDjoin. After determining the corresponding insert positions in the CPM, a minimum SESE block is determined to properly insert the surrounding AND block with a branch containing n_{new}. Fig. 6a shows an example illustrating how different insert positions depending on the parameter value are chosen. Note that, independent of the concrete parameter value and insert position respectively, the user of view V always gets the same model when re-applying the view creation and refactoring operations on the CPM. Similar to *InsertParallel*, the propagation of a change expressed in terms of operations *InsertConditional* or *InsertLoop* can be accomplished. In addition to insert join/split gateways, branching condition c has to be set to guarantee proper process execution (cf. Table 1).



```
OpV1={
RedActivity(V1,B),
RedActivity(V1,E),
RedActivity(V1,F)}
```

```
OpV2={
RedActivity(V2,F),
AggrSESE(V2,{C,D,E})}
```

Fig. 6. Updating the CPM after a View Change

3.2 Migrating Process Views to a new CPM Version

When changing a CPM through updating one of its associated views, all other views defined on this CPM must be updated as well. More precisely, it must be guaranteed that all process views are up-to-date and hence users always interact with the current version of a process model and related views respectively. To ensure this, after propagating a view change to a CPM, the creation sets of all other process views must be migrated to the new CPM version (cf. Definition 3). Note that in certain cases this creation set will contradict to the CPM, e.g., an activity might be inserted in a branch, which is aggregated through an AggrComplBranches operation. In this case, the operation has to be adapted including the new activity. Table 5 provides *migration rules* required to migrate creation sets of associated views after updating the CPM.

Migration Rule M1: $\exists AggrSESE/AggrComplBranches(V, N_a) = Op_1 : N_a \supset \{pred(V, N_c), succ(V, N_c)\},\ Op_1 \in Op$ $\Rightarrow$ AggrComplMode=SHOW: $Op' = Op \setminus Op_1$ AggrComplMode=AGGR: $Op' = Op \setminus Op_1 \cup \{AggrSESE/AggrComplBranches(V, N_a \cup N_c)\}$
Migration Rule M2: $\exists AggrSESE/AggrComplBranches(V, N_a) = Op_1 : pred(V, N_c) \in N_a \oplus succ(V, N_c) \in N_a$ $\Rightarrow$ AggrPartlyMode=SHOW: $Op' = Op \setminus Op_1$ AggrPartlyMode=AGGR: $Op' = Op \setminus Op_1 \cup \{AggrSESE/AggrComplBranches(V, N_a \cup N_c)\}$
Migration Rule M3: $\exists RedActivity(V, pred(V, N_c))) = Op_1 \wedge RedActivity(V, succ(V, N_c)) = Op_2,\ Op \supset \{Op_1, Op_2\}$ $\Rightarrow$RedComplMode=SHOW: no action required RedComplMode=RED: $Op' = Op \cup Op_N,\ Op_N = \{n \in N_c \mid RedActivity(V, n)\}$
Migration Rule M4: $\exists RedActivity(V, pred(V, N_c))) = Op_1 \oplus RedActivity(V, succ(V, N_c)) = Op_2,\ Op \supset \{Op_1, Op_2\}$ $\Rightarrow$RedPartlyMode=SHOW: no action required RedPartlyMode=RED: $Op' = Op \cup Op_N,\ Op_N = \{n \in N_c \mid RedActivity(V, n)\}$

Table 5. Process View Migration Rules

Regarding migration rule *M1*, N_c denotes the set of nodes added to the CPM. If the direct predecessor and successor of this node set are both aggregated to the same abstract node (i.e., both are element of set N_a, which is aggregated through AggrSESE or AggrComplBranches), the migration rule will be applied. In this case, there exist two options: either node set N_c is included in the aggregation or this aggregation is removed and the change is shown to the user. This can be expressed by parameter *AggrComplMode* for each view: parameter value *SHOW* suggests removing the aggregation operations in the creation set, while value *AGGR* (default) extends the aggregated node set by the new nodes in N_c. If only one of the nodes (i.e., the predecessor or successor of N_c) is included in an aggregation, migration rule *M2* is applied. In this case, parameter *AggrPartlyMode* expresses whether the aggregation shall be expanded (i.e., *AGGR*) or resolved (i.e., *SHOW*). Fig. 6bc present examples of this operation.
Migration rules *M3* and *M4* handle changes within reduced node sets. Analogous to the handling of aggregation operations, migration rule *M3* is applied if both the predecessor and successor of node set N_c are removed due to a reduction. In turn, *M4* is applied if exactly only one of these two nodes is reduced. In this case, parameter *RedComplMode* (or *RedPartlyMode*) and its values (*SHOW* or *RED* (default)) determine whether node set N_c is visible or reduced in the view. After migrating all creation sets belonging to a CPM, the corresponding views are recreated (cf. Fig. 1). Applying a change to the CPM and recreating the process views afterwards allows us to guarantee that all views are up-to-date.
Since the recreation of a process view is expensive, several optimization techniques are applied. First, instead of recreating all process views, this is only accomplished for those views affected by the change. Second, when changing the creation set, the visualization engine exactly knows which parts of the process view have changed and respective parts are then recreated.

4 Related Work

In the context of cross-organizational processes, views have been applied for creating abstractions of partner processes hiding private process parts [6,14,15,16]. However, process views are manually specified by the process designer, but do not serve as abstractions for changing large process models as in *proView*.
An approach providing predefined process view types (i.e., human tasks, collaboration views) is presented in [4]. As opposed to *proView*, this approach is limited to these pre-specified process view types. In particular, these views are not used as abstractions enabling process change. In turn, [17] applies graph reduction techniques to verify structural properties of process schemas. The *proView* project accomplishes this by enabling aggregations that use high-level operations. In [18] SPQR-tree decomposition is applied when abstracting process models. This approach neither takes other process perspectives (e.g., data flow) nor process changes into account.
The approach presented in [19] determines semantic similarity between activities by analyzing the schema of a process model. The similarity discovered is used to abstract the process model. However, this approach neither distinguishes between different user perspectives on a process model nor provides concepts for manually creating process views.
An approach for creating aggregated process views is described in [20]. It proposes a two-phase procedure for aggregating parts of a process model not to be exposed to the public. However, process view updates to evolve or adapt processes are not considered.
View models serving monitoring purpose are presented in [21,22]. Focus is on the run-time mapping between process instances and views. Further, the views have to be pre-specified manually by the designer.
In turn, [23] aligns technical workflows with business processes. It allows detecting changes through behavioural profiles and propagating them to change regions of the corresponding technical model. These regions indicate the schema region to which the change belongs. Automatic propagation is not supported. Similarly, [24] describes a mapping model between a technical workflow and a business process. An automatic propagation of changes is not supported.
For defining and changing process models, various approaches exist. [25] presents an overview of frequently used patterns for changing process models. Further, [7] summarizes approaches enabling flexibility in PAISs. In particular, [26] presents an approach for adapting well-structured process models without affecting their correctness properties. Based on this, [27] presents concepts for optimizing process models over time and migrating running processes to new model versions properly. None of these approaches takes usability issues into account, i.e., no support for user-centered changes of business processes is provided.
The *proView* framework provides a holistic framework for personalized view creation. Further, it enables users to change business processes based on their views and guarantees that other views of the process model are adapted accordingly. None of the existing approaches covers all these aspects and is based on rigid constraints not taking practical requirements into account.

5 Conclusion

We introduced the *proView* framework and its formal foundation; *proView* supports the creation of personalized process views and the view-based change of business processes, i.e., process abstractions not only serve visualization purpose, but also lift process changes up to a higher semantical level. A set of update operations enables users to update their view and to propagate the respective schema change to the underlying process model representing the holistic view on the business process. Parameterization of these operations allows for automatically resolving ambiguities when propagating view changes; i.e., the change propagation behaviour can be customized for each view. Finally, we provide migration rules to update all other process views associated with a changed process model. Similar to the propagation, per view it can be decided how much information about the change should be displayed to the user.
The *proView* framework described in this paper is implemented as a client-server application to simultaneously edit process models based on views [28]. The implementation proves the applicability of our framework. Furthermore, user experiments based on the implementation are planned to test the hypothesis that view-based process changes improve the handling and evolution of large process models. Overall, we believe such view-based process updates offer promising perspectives to better involve process participants and domain experts in evolving their business processes.

References

1. Weber, B., Sadiq, S., Reichert, M.: Beyond Rigidity - Dynamic Process Lifecycle Support: A Survey on Dynamic Changes in Process-Aware Information Systems. Computer Science - Research and Development **23**(2) (2009) 47–65
2. Weber, B., Reichert, M., Mendling, J., Reijers, H.A.: Refactoring Large Process Model Repositories. Computers in Industry **62**(5) (2011) 467–486
3. Streit, A., Pham, B., Brown, R.: Visualization Support for Managing Large Business Process Specifications. In: Proc. 3rd Int'l Conf. Business Process Management (BPM'05). (2005) 205–219
4. Tran, H.: View-Based and Model-Driven Approach for Process-Driven, Service-Oriented Architectures. TU Wien, Dissertation (2009)
5. Bobrik, R., Bauer, T., Reichert, M.: Proviado - Personalized and Configurable Visualizations of Business Processes. In: Proc. 7th Int'l Conf. Electronic Commerce & Web Technology (EC-WEB'06), Krakow, Poland (2006) 61–71
6. Chiu, D.K., Cheung, S., Till, S., Karlapalem, K., Li, Q., Kafeza, E.: Workflow View Driven Cross-Organizational Interoperability in a Web Service Environment. Information Technology and Management **5**(3/4) (July 2004) 221–250
7. Reichert, M., Weber, B.: Enabling Flexibility in Process-aware Information Systems - Challenges, Methods, Technologies. Springer (2012)
8. Kolb, J., Reichert, M.: Using Concurrent Task Trees for Stakeholder-centered Modeling and Visualization of Business Processes. In: Proc. S-BPM ONE 2012, CCIS 284. (2012) 237–251

9. Kolb, J., Rudner, B., Reichert, M.: Towards Gesture-based Process Modeling on Multi-Touch Devices. In: Proc. 1st Int'l Workshop on Human-Centric Process-Aware Information Systems (HC-PAIS'12), Gdansk, Poland (2012) 280–293
10. Kolb, J., Hübner, P., Reichert, M.: Automatically Generating and Updating User Interface Components in Process-Aware Information Systems. In: Proc. 10th Int'l Conf. on Cooperative Information Systems (CoopIS'12). (2012) to appear
11. Johnson, R., Pearson, D., Pingali, K.: Finding Regions Fast: Single Entry Single Exit and Control Regions in Linear Time. In: Proc. Conf. on Programming Language Design and Implementation (ACM SIGPLAN'94). (1993)
12. Reichert, M., Kolb, J., Bobrik, R., Bauer, T.: Enabling Personalized Visualization of Large Business Processes through Parameterizable Views. In: Proc. 26th Symposium On Applied Computing (SAC'12), Riva del Garda (Trento), Italy (2012)
13. Bobrik, R., Reichert, M., Bauer, T.: View-Based Process Visualization. In: Proc. 5th Int'l Conf. on Business Process Management, Brisbane, Australia (2007) 88–95
14. Chebbi, I., Dustdar, S., Tata, S.: The View-based Approach to Dynamic Inter-Organizational Workflow Cooperation. Data & Know. Eng. **56**(2) (2006) 139–173
15. Kafeza, E., Chiu, D.K.W., Kafeza, I.: View-Based Contracts in an E-Service Cross-Organizational Workflow Environment. In: Techn. E-Services. (2001) 74–88
16. Schulz, K.A., Orlowska, M.E.: Facilitating Cross-Organisational Workflows with a Workflow View Approach. Data & Knowledge Engineering **51**(1) (2004) 109–147
17. Sadiq, W., Orlowska, M.E.: Analyzing Process Models Using Graph Reduction Techniques. Information systems **25**(2) (2000) 117–134
18. Polyvyanyy, A., Smirnov, S., Weske, M.: The Triconnected Abstraction of Process Models. In: Proc. 7th Int'l Conf. on Business Process Management. (2009)
19. Smirnov, S., Reijers, H.A., Weske, M.: A Semantic Approach for Business Process Model Abstraction. In: Advanced Information Systems Engineering, Springer Berlin / Heidelberg (2011) 497–511
20. Eshuis, R., Grefen, P.: Constructing Customized Process Views. Data & Knowledge Engineering **64**(2) (2008)
21. Shan, Z., Yang, Y., Li, Q., Luo, Y., Peng, Z.: A Light-Weighted Approach to Workflow View. APWeb 2006 (2003) (2006) 1059–1070
22. Schumm, D., Latuske, G., Leymann, F., Mietzner, R., Scheibler, T.: State Propagation for Business Process Monitoring on Different Levels of Abstraction. In: Proc. 19th ECIS. Number Ecis, Helsinki, Finland (2011)
23. Weidlich, M., Weske, M., Mendling, J.: Change Propagation in Process Models using Behavioural Profiles. Proc. 6th IEEE Int'l Conf. Services Comp. (2009) 33–40
24. Buchwald, S., Bauer, T., Reichert, M.: Bridging the Gap Between Business Process Models and Service Composition Specifications. In: Service Life Cycle Tools and Technologies: Methods, Trends and Advances. IGI Global (2011) 124–153
25. Weber, B., Reichert, M., Rinderle, S.: Change Patterns and Change Support Features - Enhancing Flexibility in Process-Aware Information Systems. Data & Knowledge Engineering **66**(3) (2008) 438–466
26. Reichert, M., Dadam, P.: ADEPTflex - Supporting Dynamic Changes of Workflows Without Losing Control. Journal of Intelligent Inf. Sys. **10**(2) (1998) 93–129
27. Rinderle, S., Reichert, M., Dadam, P.: Flexible Support of Team Processes by Adaptive Workflow Systems. Distributed and Par. Databases **16**(1) (2004) 91–116
28. Kolb, J., Kammerer, K., Reichert, M.: Updatable Process Views for Adapting Large Process Models: The proView Demonstrator. In: Proc. of the Business Process Management 2012 Demonstration Track, Tallinn, Estonia (2012) to appear