Enabling Personalized Visualization of Large Business Processes through Parameterizable Views

Manfred Reichert,
Jens Kolb
Ulm University, Germany
{manfred.reichert,
jens.kolb}@uni-ulm.de

Ralph Bobrik
Detecon AG
Switzerland
ralph.bobrik@
detecon.com

Thomas Bauer
Hochschule Neu-Ulm
Neu-Ulm, Germany
thomas.bauer
@hs-neu-ulm.de

ABSTRACT

Process-aware information systems (PAISs) need to support personalized views on business processes since different user groups have distinguished perspectives on these processes and related data. Existing PAISs, however, do not provide mechanisms for creating and visualizing such process views. Typically, processes are displayed to users in exactly the same way as originally modeled. This paper presents a flexible approach for creating personalized process views based on parameterizable operations. Respective view-building operations can be flexibly composed in order to hide process information or abstract from it in the desired way. Depending on the chosen parameterization of the operations applied, we obtain process views with more or less relaxed properties (e.g., regarding the degree of information loss or soundness). Altogether, the realized view concept enables a more flexible visualization of large business processes satisfying the needs of different user groups.

1. INTRODUCTION

The field of Enterprise Engineering combines system engineering and strategic management to streamline companies in terms of organisational structure, products and business processes. Therefore process-aware information systems (PAISs) are an integral part to support business processes at an operational level. Further, PAISs strictly separate 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]. In this context, companies have to deal with a large number of business processes involving different domains, organizational units, and tasks. Generally, these processes are stored in central process repositories [2]. In large companies, such repositories can easily contain several thousands of process models [3]. Each of these process models comprises a large number of activities, and involve a multitude of user groups. Usually, each of these user groups needs a different view on the process with customized visualization and information granularity [4]. For example, managers rather prefer an abstract overview, whereas process participants need a detailed view of those process parts they are involved in. Thus, personalized process visualization is a much needed PAIS feature. Despite its practical importance, current PAISs do not offer adequate visualization support. Often, process models are displayed to the user as drawn by the process designer. Generally, these process models are too complex and, hence not comprehensible to most users, e.g. due to the numerous technical activities to execute the process. Some tools allow altering the graphical appearance of a process and hiding selected aspects (e.g., data flow). However, sophisticated and flexible concepts for creating and managing user-specific views are missing.

To elaborate basic visualization requirements several case studies [5] were conducted resulting in three dimensions of process visualization [6]. *First*, it must be possible to reduce complexity by hiding or aggregating process information not being relevant in the given context. *Second*, the notation and appearance of process nodes (e.g., activities and data objects) should be customizable [6]. *Third*, different presentation formats (e.g., process graph, table) need to be supported.
Our Proviado project provides a generic framework for visualizing large processes, which considers these three dimensions. Fig. 1 sketches the corresponding architecture. For visualizing application-spanning processes, process data from heterogeneous sources is integrated and translated into the Proviado process format (a) [7]. The way a process shall be displayed to a particular user role is defined by a *Visualization Model* comprising relevant configuration data (b), like the process part to be visualized or the notation to be used. Taking this information the *Visualization Engine* generates an abstracted process schema (c) and adapts the graphical representation of its nodes (d). Finally, the user-specific view, which may include run-time data (e.g., status infor-

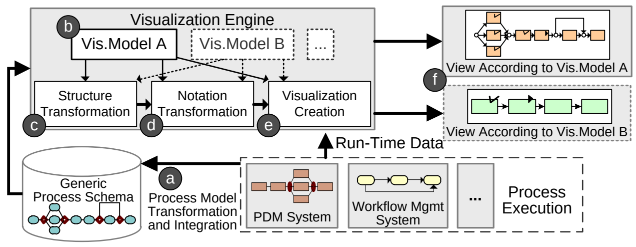

Figure 1: Proviado Framework

mation, application data) is created (e). Using multiple Visualizations Models allows us to realize customized and personalized process visualizations (f).

This paper focuses on the parameterizable *Structure Transformation* component (Fig. 1c), i.e., on the provision of a flexible component for building process views. Such a component must cover a variety of use cases. For example, it should be possible to build views only containing activities the current user is involved in or only showing non-completed process regions. As another example, consider executable process models which often contain technical activities (e.g., data transformation steps) to be excluded from visualization. Finally, selected process nodes may have to be hidden or aggregated to meet confidentiality demands. Proviado allows creating respective process views based on well defined, parameterizable view operations. These rely on both graph reduction and graph aggregation techniques. While the former can be used to remove nodes from a process model, the latter are applied to abstract from certain process information (e.g., aggregating several activities to one abstract node). Proviado additionally supports the flexible composition of such basic view operations to realize more sophisticated process abstractions. The basic idea of building process views has been already sketched in [8, 9]. In this paper we introduce more advanced view operations and their formal properties. Further, we use the view-building operations for enabling parameterization of high-level view operations.

The implementation of view operations is by far not trivial and may require complex process schema transformations. Besides activity aggregations may become complex, e.g. when aggregating non-connected activities. Furthermore, it should be possible to flexibly configure the properties of a process view depending on application needs. If the focus is on process visualization, for example, view properties may be relaxed. Opposed to this, more rigid constraints become necessary if views shall be updatable or executable.

Section 2 gives background information needed for understanding this paper. In Section 3 we introduce the formal foundations of parameterizable process views. Section 4 gives insights into practical issues and presents more complex examples for defining and creating process views. Section 5 discusses related work and Section 6 concludes with a summary.

2. BACKGROUNDS

Each process is represented by a process schema consisting of process nodes and the control flow between them (cf. Fig. 2). For control flow modeling, control gateways (e.g., ANDsplit, XORsplit) and control edges are used.

Definition 1. A *process schema* is defined by a tuple $P = (N, E, EC, NT)$ where:

- N is a set of process nodes,
- $E \subset N \times N$ is a precedence relation (notation: $e = (n_{src}, n_{dest}) \in E$),
- $EC : E \to Conds \cup \{\text{TRUE}\}$ assigns optionally transition conditions to control edges,
- $NT : N \to \{Activity, ANDsplit, ANDjoin, ORsplit, ORjoin, XORsplit, XORjoin\}$ assigns to each $n \in N$ a node type $NT(n)$; N is divided into disjoint sets of activity nodes A ($NT = Activity$) and gateways S ($NT \neq Activity$).

Note that this definition focuses on the control flow perspective. In particular, it can be applied to existing activity-oriented modeling languages (e.g. BPMN). However, the view operations presented in Section 3 consider other process perspectives as well (e.g., data elements, data flow). Furthermore, Proviado also considers loop structures. We exclude the latter in the paper due to lack of space (see [10] for details); i.e., we only consider acyclic process graphs.

We assume that a process schema has one start and one end node. Further, it has to be connected; i.e., each activity can be reached from the start node, and from each activity the end node is reachable. Finally, branches may be arbitrarily nested, but must be safe (e.g., a branch following a $XORsplit$ must not merge with an $ANDjoin$).

Definition 2. Let $P = (N, E, EC, NT)$ be a process schema and let $X \subseteq N$ be a subset of activity nodes. The subgraph P' induced by X is called *SESE* (Single Entry Single Exit) fragment iff P' is connected and has exactly one incoming and one outgoing edge connecting it with P.

Based on a process schema P related process instances can be created and executed at run-time. Regarding the process instance from Fig. 2, for example, activities A and B are completed, C and N are activated (i.e., offered as work items in user worklists), and H and K are running.

Definition 3. A *process instance* I is defined by a tuple $(P, NS, \mathcal{H})$ where

- P denotes the process schema on which I is running,
- $NS : N \to ExecutionStates := \{NotActivated, Activated, Running, Skipped, Completed\}$ describes the execution state of each node $n \in N$,
- $\mathcal{H} = \langle e_1, \ldots, e_n \rangle$ denotes the execution history of I where each entry e_k is related either to the start or completion of a particular process activity.

For an activity $n \in N$ with $NS(n) \in \{Activated, Running\}$ all preceding activities either must be in state *Completed* or *Skipped*, and all succeeding activities must be in state *NotActivated*. Further, there is a path π from the start node to n with $NS(n') = Completed \ \ \forall n' \in \pi$.

3. FUNDAMENTALS ON BUILDING VIEWS

We first introduce basic view-building operations and reason about properties of resulting view schemas. As first example consider the process instance from Fig. 3a. Assume that each of the activity sets $\{B, C, H, K\}$, $\{J, L\}$, and $\{T, U, V\}$ shall be aggregated, i.e., replaced by one abstract node. Assume further that activity sets $\{E, F, G\}$ and $\{R, S\}$ shall be hidden. Fig. 3b shows a possible view resulting from respective aggregations and reductions. Generally, process views exhibit an information loss when compared to the original process. As important requirement, view-building operations should have a precise semantics and be

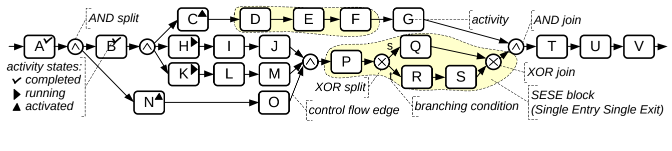

Figure 2: Example of a process instance

applicable to both process schemas and instances. Further, it should be possible to remove process nodes (*reduction*) or to replace them by abstracted ones (*aggregation*). When building views it is fundamental to preserve the structure of non-affected process regions. Finally, the effects of view-building operations should be parameterizable to meet application needs best and to be able to control the degree of information loss.

We first give an abstract definition of a *process view*. The concrete properties of such a view depend on the applied view operations and their parameterization.

Definition 4. Let $P = (N, E, EC, NT)$ be a process schema with activity set $A \subseteq N$. Then: A *process view* on P is a process schema $V(P) = (N', E', EC', NT')$ whose activity set $A' \subseteq N'$ can be derived from P by reducing and aggregating activities from $A \subseteq N$. Formally:

- $A_U = A \cap A'$ denotes the set of activities present in both P and $V(P)$,
- $A_D = A \setminus A'$ denotes the set of activities present in P, but not in $V(P)$; i.e., reduced or aggregated activities: $A_D \equiv AggrNodes \cup RedNodes$
- $A_N = A' \setminus A$ denotes the set of activities present in $V(P)$, but not in P. Each $a \in A_N$ is an abstract activity aggregating a set of activities from A:
  1. $\exists AggrNodes_i, i = 1, \ldots, n$ with $AggrNodes = \dot{\bigcup}_{i=1,\ldots,n} AggrNodes_i$
  2. There exists a bijective function *aggr* with: $aggr : \{AggrNodes_i | i = 1, \ldots, n\} \to A_N$

Using the notions from Def. 4 for a given process schema P and related view $V(P)$ we introduce function $VNode : A \to A'$, which maps each process activity $c \in A_U \cup AggrNodes$ to a corresponding activity in the view :

$$VNode(c) = \begin{cases} c & c \in A_U \\ aggr(AggrNodes_i) & \exists i \in \{1, \ldots, n\}: \\ & c \in AggrNodes_i \\ undefined & c \notin A_U \cup AggrNodes \end{cases}$$

For each view activity $c' \in A'$, $VNode^{-1}(c')$ denotes the corresponding activity or the set of activities aggregated by c' in the original schema. Finally more complex process views are created by composing a set of view operations, which also define the semantics of the view (cf. Sec. 4).

3.1 Building Process Views Based on Schema Reduction

Any view component should be able to remove activities in a process. For example, this is required to hide irrelevant

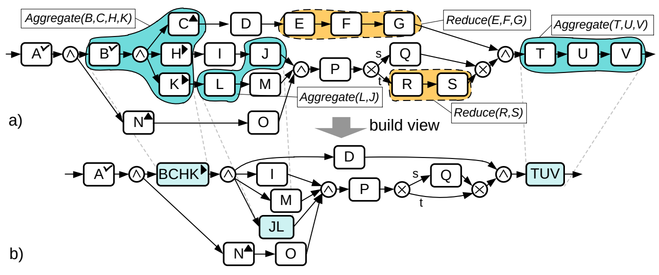

Figure 3: Example of a process view

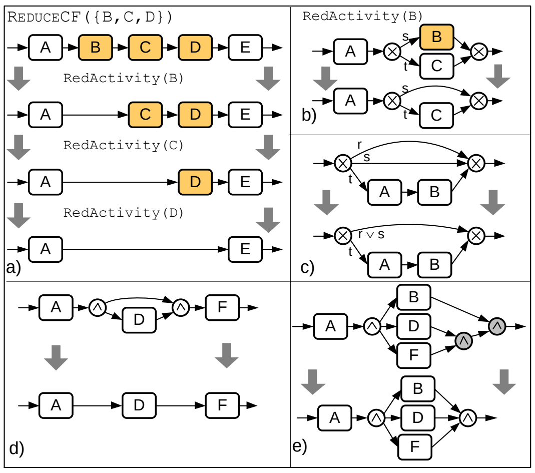

Figure 4: Recution and simplification operations

or confidential process details from a particular user group. For this purpose, Proviado provides an elementary reduction operation (cf. Fig. 4b). Based on it, higher-level reduction operations for hiding a set of activities at once can be realized. Reduction of an activity (`RedActivity`) is realized by removing it together with its incoming/outgoing edges from the schema. Then a new control edge is inserted between the predecessor and successor of the removed activity (cf. Fig. 4b). For reducing activity sets, the single-aspect view operation REDUCECF is provided. Single-aspect operations focus on elements of one particular process aspect. Reduction is performed stepwise, i.e., for all activities to be removed, `RedActivity` is applied (cf. Fig. 4a). Note that the algorithms we apply are context-free and may introduce unnecessary process nodes (e.g. empty branches). Respective nodes are purged afterwards by applying well-defined simplification rules to the created view schema. For example, when reducing a complete branch of a parallel branching, the resulting control edge may be removed as well (cf. Fig. 4d). In case of an XOR-/OR-branching, however, the empty path (i.e., control edge) needs to be preserved. Otherwise an inconsistent schema would result (cf. Fig. 4b). Similarly, when applying simplification rules to XOR-/OR-branches respective transition conditions must be recalculated (cf. Fig. 4c). Reduction of activities always comes along with an information loss, while preserving the structure of the non-reduced schema parts, i.e., the activities being present in both process and view schema. The latter can be expressed using the notion of *order preservation*. For this we introduce partial order relation $\preceq (\subseteq N \times N)$ on process schema P with $n_1 \preceq n_2 \Leftrightarrow \exists$ path π in P from n_1 to n_2.

Definition 5. Let $P = (N, E, EC, NT)$ be a process schema with activity set $A \subseteq N$ and let $V(P) = (N', E', EC', NT')$ be a view on P with activity set $A' \subseteq N'$. Then: $V(P)$ is called *order-preserving* iff $\forall n_1, n_2 \in A$ *with* $n_1 \neq n_2$ *and* $n_1 \preceq n_2 : n'_1 = VNode(n_1) \wedge n'_2 = VNode(n_2) \Rightarrow \neg(n'_2 \preceq n'_1)$.

This property reflects that the order of two activities in a process schema must not be reversed in a corresponding view. Obviously, the reduction operations depicted in Fig. 4ab are order preserving. Generally, this property is fundamental for ensuring the integrity of process schemas and corresponding views. A stronger notion is provided by Def. 6. As we will see later, in comparison to Fig. 4ab there are view operations which do not comply with Def. 6.

Definition 6. Let $P = (N, E, EC, NT)$ be a process schema with activity set $A \subseteq N$ and let $V(P) = (N', E', EC', NT')$ be a corresponding view with $A' \subseteq N'$. Then: $V(P)$ is *strong order-preserving* iff $\forall n_1, n_2 \in A$ *with* $n_1 \neq n_2$ *and* $n_1 \preceq n_2$: $n'_1 = VNode(n_1) \wedge n'_2 = VNode(n_2) \Rightarrow n'_1 \preceq n'_2$.

3.2 Building Process Views Based on Schema Aggregation

The aggregation operation enables merging a set of activities into one abstracted node. Depending on the structure of the subgraph induced by the respective activities, different schema transformations may have to be applied. In particular, the aggregation of non-connected activities necessitates a more complex restructuring of the process schema. Fig. 5 shows the elementary operations for building aggregated views. The depicted operations follow the policy to substitute the activities in-place by an abstract one (if possible) while maintaining order-preservation (cf. Def. 5). Note that *in-place substitution* is always possible when aggregating a SESE fragment (cf. Def. 2). If none of the operations from Fig. 5ade can be applied, `AggrAddBranch` (cf. Fig. 5b) is used. It identifies the nearest common ancestor and successor of all activities to be aggregated and adds a new branch between them (cf. Fig. 5b). Alternatively, aggregation of non-connected activities can be handled by applying elementary operations of type `AggrSESE` (cf. Sec. 4). Finally, when aggregating activities directly following a split node there exist two alternatives (cf. Fig. 5e). The first one aggregates activities applying `AggrAddBranch`, the second one shifts activities to the position preceding the split node (`AggrShiftOut`).

Except `AggrAddBranch` the operations presented are strong order-preserving (cf. Def. 6). However, `AggrAddBranch` violates this property. For example, in Fig. 5b order relation $D \preceq E$ is not preserved after applying this operation.

Definition 7. Let $P = (N, E, EC, NT)$ be a process schema with activity set $A \subseteq N$. Then: $\mathbb{D}_P = \{(n_1, n_2) \in A \times A | n_1 \preceq n_2\}$ is denoted as *dependency set* and reflects all direct and transitive control flow dependencies between any two activities.

We are interested in the relation between the dependency set of a process and a related process view. For this purpose, let $\mathbb{D}_P$ be the dependency set of P and $\mathbb{D}_{V(P)}$ be the dependency set of view $V(P)$. We further introduce a projection of the dependencies from $V(P)$ on P denoted as $\mathbb{D}'_{V(P)}$. It can be derived by substituting the dependencies of the abstract activities by the ones of the original activities. As example consider `AggrShiftOut` in Fig. 5e. We obtain $\mathbb{D}_P = \{(A, B), (B, C), (C, F), (A, D), (D, E), (E, F)\}$ and $\mathbb{D}_{V(P)} = \{(A, BD), (BD, C), (BD, E), (C, F), (E, F)\}$. Further $\mathbb{D}'_{V(P)} = \{(A, B), (B, C), (D, C), (C, F), (A, D), (B, E), (D, E), (E, F)\}$ holds. As one can see, $\mathbb{D}'_{V(P)}$ contains additional dependencies. We denote this property as dependency-generating.

Calculation of $\mathbb{D}'_{V(P)}$: For $n_1 \in A_N$, $n_2 \in A'$: remove all $(n_1, n_2) \in \mathbb{D}_{V(P)}$; insert $\{(n, n_2) | n \in aggr^{-1}(n_1)\}$ instead (analogously for $n_2 \in A_N$); finally insert the dependencies between $AggrNodes$, i.e., $\mathbb{D}_P[AggrNodes] = \{d = (n_1, n_2) \in \mathbb{D}_P | n_1 \in AggrNodes \wedge n_2 \in AggrNodes\}$

Now we can classify effects on the dependencies between activities when building a view.

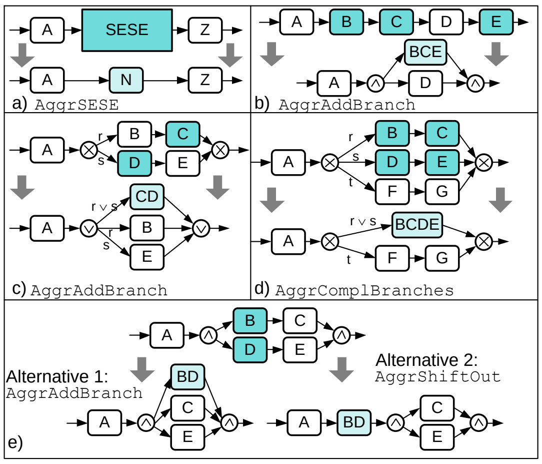

Figure 5: Elementary aggregation operations

Definition 8. Let $P = (N, E, EC, NT)$ be a process schema and let $V(P)$ be a corresponding view. Let $\mathbb{D}_P$ and $\mathbb{D}'_{V(P)}$ be the dependency sets as defined above.

- $V(P)$ is denoted as **dependency-erasing** iff there exist dependency relations in $\mathbb{D}_P$ not existing in $\mathbb{D}'_{V(P)}$ anymore.
- $V(P)$ is denoted as **dependency-generating** iff $\mathbb{D}'_{V(P)}$ contains dependency relations not contained in $\mathbb{D}_P$.
- $V(P)$ is denoted as **dependency-preserving** iff it is neither dependency-erasing nor dependency-generating.

Generally reduction operations are dependency-erasing. When aggregating activities, however, there exist elementary operations of all three types. In Fig. 5, for instance, `AggrSESE` is dependency-preserving, while `AggrAddBranch` is dependency-erasing as $(B, C) \notin \mathbb{D}'_{V(P)}$ holds; finally, `AggrShiftOut` is dependency-generating: $(B, E) \in \mathbb{D}'_{V(P)}$. Theorem 1 explains the relation of dependency properties (cf. Def. 8) and order-preservation (cf. Def. 5).

Theorem 1. Let $P = (N, E, EC, NT)$ be a process schema and let $V(P)$ be a corresponding view. Then:

- $V(P)$ is dependency-erasing $\Rightarrow$ $V(P)$ is not strong order-preserving
- $V(P)$ is dependency-preserving $\Rightarrow$ $V(P)$ is strong order-preserving

The proof of Theorem 1 can be based on the definition of the properties and dependency sets, and is omitted here. In summary, we have presented a set of elementary aggregation operations. Each of them fits to a specific ordering of the activities to be aggregated. In Section 4.1 we combine these operations into more complex ones utilizing the discussed properties.

Generally, additional aspects have to be covered including data elements, data flow, and attributes of process elements. Regarding data elements, for example, aggregation operations (e.g., to create abstract business objects) and reduction operations (e.g., to hide technical data elements) are provided by the Proviado framework. Fig. 6a shows an example of a simple aggregation. Here, data elements $\{D1, D2\}$ are aggregated and data edges connecting activities with data elements are re-linked when aggregating $\{B, C, D, E\}$ to preserve a valid process model.

Further, Proviado offers functions enabling attribute aggregations. Fig. 6b, for example, applies a MIN-function to the aggregation set in order to determine start time, a MAX-function for calculating end time, and a SUM-function for aggregating cost attributes.

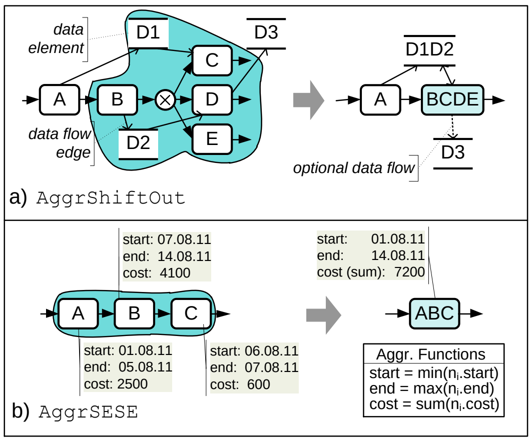

Figure 6: Aggregation of (a) data elements, (b) attributes

3.3 Applying Views to Process Instances

So far, we have only considered views on process schemas. This section additionally introduces views on process instances (cf. Def. 3). When building respective instance views their execution state (i.e. states of concrete and abstract activities) must be determined and the trace relating to the instance view logically needs to be adapted. Examples are depicted in Fig. 7a (reduction) and Fig. 7b (aggregation). Function VNS calculates the state of an abstract activity based on the states of the aggregated activities.

$$VNS(X) = \begin{cases} NotActivated & \forall x \in X: NS(x) \notin \{Activated, Running, Completed\} \wedge \\ & \exists x \in X: NS(x) = NotActivated \\ Activated & \exists x \in X: NS(x) = Activated \ \wedge \forall x \in X: \\ & NS(x) \notin \{Running, Completed\} \\ Running & \exists x \in X: NS(x) = Running \ \vee \\ & \exists x_1, x_2 \in X: NS(x_1) = Completed \ \wedge \\ & (NS(x_2) = NotActivated \vee NS(x_2) \\ & = Activated) \\ Completed & \forall x \in X: NS(x) \notin \{NotActivated, \\ & Running, Activated\} \wedge \\ & \exists x \in X: NS(x) = Completed \\ Skipped & \forall x \in X: NS(x) = Skipped \end{cases}$$

Definition 9. Let $I = (P, NS, \mathcal{H})$ be an instance of schema $P = (N, E, EC, NT)$ and $V(P) = (N', E', EC', NT')$ be a view on P with corresponding aggregation and reduction sets $AggrNodes$ and $RedNodes$ (cf. Def. 4). Then: $V(I)$ on I is a tuple $(V(P), NS', \mathcal{H}')$ with:

- $NS' : N' \rightarrow ExecutionStates$ with $NS'(n') = VNS(VNode^{-1}(n'))$ assigns to each view activity a corresponding execution state.
- $\mathcal{H}'$ is the reduced/aggregated history of the instance. It is derived from $\mathcal{H}$ by (1) removing all entries e_i related to activities in $RedNodes$ and (2) for all j: replacing the first (last) occurrence of a start event (end event) of activities in $AggrNodes_j$ and remove the remaining start (end) events.

Examples are depicted in Fig. 7. Fig. 7c shows the scenario from Fig. 5e with execution states added. Note that applying **`AggrShiftOut`** yields an inconsistent state as two subsequent activities are in state *Running*.

Definition 10. Let $I = (P, NS, \mathcal{H})$ be a process instance and let $V(I)$ be a corresponding view on I. Then:

- $V(I)$ is **strong state-consistent** iff for all paths π (cf. Sec. 2) in $V(I)$ from start to end and not containing activities in state *Skipped*, there exists exactly one activity in state *Activated* or *Running*.
- $V(I)$ is **state-consistent** iff for all paths π in $V(I)$ from start to end and not containing activities in state *Skipped*, there does not exists more than one activity in state *Activated* or *Running*.

As indicated in Fig. 7a, reducing activities from a process instance may result in a "gap" during instance execution, if no activity is *Running* or *Activated*. Hence, reduction is not strong state-consistent. A similar gap occurs if the process instance has entered a deadlock. Since we solely use our view mechanism for visualizing processes executed by heterogeneous source systems and run-time information, deadlock issues can be excluded here. Theorem 2 shows how state inconsistency is correlated with dependency relations (cf. Def. 8):

Theorem 2. Let I be a process instance and $V(I)$ a corresponding view. $V(I)$ is dependency-generating $\Rightarrow$ $V(I)$ is not state-consistent.

Proof. Let $I = (P, NS, \mathcal{H})$ be a process instance of process $P = (N, E, EC, NT)$ and $V(I) = (V(P), NS', \mathcal{H}')$ be a view on I with $V(P) = (N', E', EC', NT')$ being dependency-generating. Then, there exists $n'_1, n'_2 \in N'$ in V which generates a dependency, i.e. $n'_1 \preceq n'_2$. Let $n_1 = VNode^{-1}(n'_1) \in N$ and $n_2 = VNode^{-1}(n'_2) \in N$. Then: $n_1 \npreceq n_2$. Further there are two paths in P from start to end containing n_1 and n_2. Therefore, there exists a state of instance I with $NS(n_1) = Running$ and $NS(n_2) = Running$. Thus, $NS(n'_1) = Running$ and $NS(n'_2) = Running$. Since there is a path from start to end in $V(P)$, containing both n'_1 and n'_2, the claim follows. $\square$

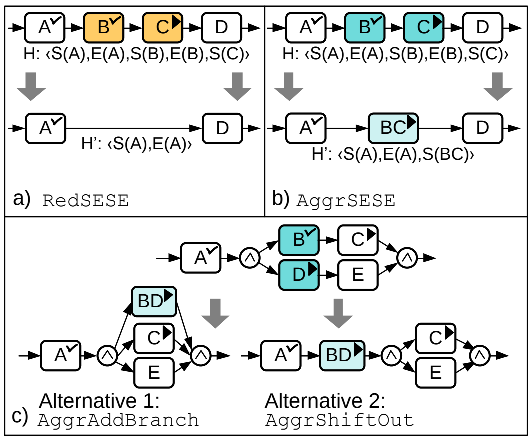

Figure 7: View operations for process instances

Table 1: Overview of operations properties

	Properties						
Operation	str. order preserving	order preserving	str. state consistent	state consistent	depend. preserv.	depend. erasing	depend. generat.
`RedActivity`	+	+	-	+	-	+	-
`AggrSESE`	+	+	+	+	+	-	-
`AggrComplBranches`	+	+	+	+	+	-	-
`AggrShiftOut`	+	+	-	-	-	-	+
`AggrAddBranch`	-	+	+	+	-	+	-

Theorem 2 shows that `AggrShiftOut` always causes an inconsistent execution state, whereas applying `AggrAddBranch` maintains a consistent state.

Concerning aggregation of process instances, Proviado allows aggregating collections of instances; i.e. multiple instances of the same process schema are condensed to an aggregated one to provide abstracted information about their progress or to aggregate key performance data.

4. ADVANCED VIEW-BUILDING CONCEPTS

To enable more sophisticated view-building concepts, the presented elementary operations may be combined. Table 1 gives an overview of view properties we can guarantee for the different elementary operations. Based on this, we can also reason about properties of views resulting from the combined use of elementary operations. For any process visualization component, however, manual selection of the elementary operations to be applied is not convenient for users since this would require indepth knowledge of these operations and their semantics. Thus, Proviado additionally provides single-aspect view operations on top of the elementary ones for coping with more complex use cases. Single-aspect operations analyze the context of the activities to be reduced or aggregated in a process schema, and determine the appropriate elementary operations to build the view with the desired properties.

4.1 Parameterizable View-building Operations

The primary use case for our view-building approach is process visualization. However, other use cases (e.g., process modeling) can be covered as well. Thus, different requirements regarding the consistency of the resulting view schema exist. Our case studies have shown that for the visualization of large processes minor inconsistencies or information loss will be tolerated if an appropriate visualization can be obtained. Obviously, inconsistencies will be not acceptable if process updates based on views shall be enabled.

To deal with these varying goals, Proviado expands single-aspect operations with a parameter set. This allows specifying the properties the resulting view schema shall have. For example, the parameters of operation AGGREGATECF are summarized in Table 2; e.g., when aggregating activities directly succeeding an ANDsplit (cf. Fig. 5e) and requiring that the view has to be state-consistent, Alt. 1 (i.e. `AggrAddBranch`) is chosen.

In certain cases, the specified parameters may be too strict; i.e., no elementary view operations can be found to realize the desired properties. We provide two strategies for addressing such scenarios. The first one subdivides the set of activities until elementary operations can be applied. The second strategy expands the activity set. Regarding reduction, in turn, view generation is always possible due to the way activity sets are split into single activities and the application of `RedActivity`.

Fig. 8 illustrates the use of the single-aspect operation AGGREGATECF. It depicts a process schema together with the set of activities to be aggregated. The view operation analyzes the structure of the activities and determines which elementary operation shall be applied. If the application of these operations results in a view schema complying with the properties defined by the desired parameters (***dependencies***, ***execution states***) it is applied as shown in Alt. 1. If parameter ***strategy*** forces us to process the set of activities as it is, and an appropriate operation cannot be found, view generation is aborted with an error message. Alt. 2 shows the result when expanding the activity set to be aggregated to the minimum SESE-block containing all activities to be aggregated. This strategy has been proposed in literature as well [11, 12]. Generally, for visualization purpose it is not always acceptable to aggregate activities originally not contained in the aggregation set.

Alt. 3ab+ and 4 can be derived by subdividing the set of activities to be aggregated. This is done stepwise: First, all connected fragments are identified (Alt. 3). If the aggregation of these fragments does not meet the required properties, the fragments are further subdivided until each subset constitutes a SESE (Alt. 4).

Altogether, parameterization of view operations significantly increases the flexibility of our view-building approach and allows defining exactly the view properties to be preserved. Considering reduction, a parameterization at the level of single-aspect operations is not useful since reduction of a complex set of activities is realized by calling `RedActivity` repeatedly as explained in Sec. 3.1.

Table 2: Overview of parameters for AggregateCF

Parameter	Values[1]	Description
dependencies	preserving, non-erasing, non-generating, **any**	The view operation applied should be dep.-preserving, not dep.-erasing, or not dep-generating. Otherwise no restrictions regarding dependencies are considered.
exec. states	**inconsist.**, consistent	The view operation applied should be state consistent or may be state inconsistent.
strategy	**as-is**,subdivide, expand	Activity set should be aggregated as-is, may be subdivided or expanded.

[1]default values are printed in bold face

4.2 A Leveled Operational Approach for Realizing Views

So far, we have presented a set of elementary and single-aspect view buliding operations. Additionally, Proviado offers high-level operations hiding as much complexity from end-users as possible. As configuration parameter the single-aspect view operations take the sets of activities to be reduced or aggregated, and then determine appropriate elementary operations. What is still needed are view operations allowing for a predicate-based specification of the re-

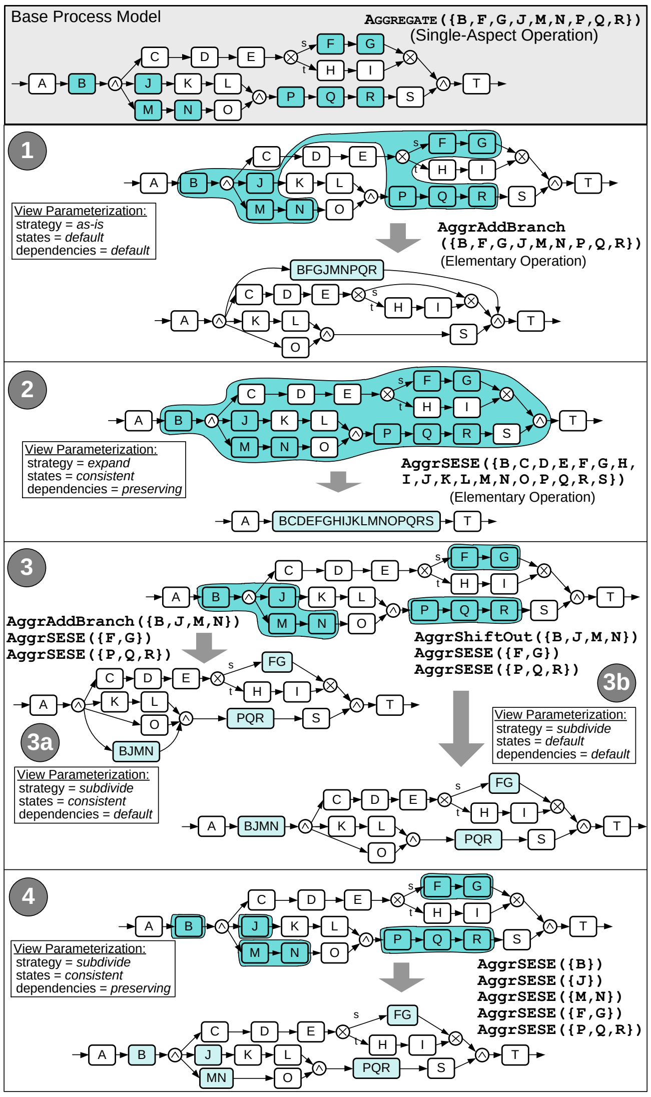
Base Process Model
AGGREGATE({B,F,G,J,M,N,P,Q,R}) (Single-Aspect Operation)
AggrAddBranch ({B,F,G,J,M,N,P,Q,R}) (Elementary Operation)
View Parameterization: strategy = as-is states = default dependencies = default
AggrSESE({B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S}) (Elementary Operation)
View Parameterization: strategy = expand states = consistent dependencies = preserving
AggrAddBranch({B,J,M,N}) AggrSESE({F,G}) AggrSESE({P,Q,R})
AggrShiftOut({B,J,M,N}) AggrSESE({F,G}) AggrSESE({P,Q,R})
View Parameterization: strategy = subdivide states = consistent dependencies = default
View Parameterization: strategy = subdivide states = default dependencies = default
View Parameterization: strategy = subdivide states = consistent dependencies = preserving
AggrSESE({B}) AggrSESE({J}) AggrSESE({M,N}) AggrSESE({F,G}) AggrSESE({P,Q,R})

Figure 8: Views depending on quality parameters

spective activity sets. Besides, operations with built-in intelligence are useful, e.g. "show only activities of a certain user role". To meet these requirements, Proviado organizes view operations in four layers (cf. Fig. 9a). Thereby, high-level operations may access lower-level ones. For defining a view, operations from all layers can be used.

Elementary operations are tailored for a specific ordering of the selected activity set within the process schema (cf. Sec. 3). *Single-aspect operations* receive a set of activities as input to be processed. They analyze the structure of the activities in the process schema and select the appropriate elementary operations based on the chosen parameterization. *Multi-aspect operations* consider elements of different type (e.g., activities, data elements) and delegate their processing to single-aspect operations. *High-level operations* abstract from the aggregations or reductions neccessary to build a particular view: *ShowMyActivities* extracts exactly those parts the user is involved in. *AggrExecutedPart* only shows those parts of the process which still may be executed, and aggregates already finished activities. Fig. 9b gives an example illustrating the way high-level operations are translated into a combination of single-aspect and elementary operations, respectively.

Proviado supports additional operations at the different levels for considering data flow and for aggregating attribute values; e.g., to handle adjacent data elements when aggregating activities (remove, aggregate, or maintain) [10].

5. RELATED WORK

IEEE 1471 recommends user-specific viewpoints for software architectures [13]. These viewpoints are templates from which individual views are created for a concrete software architecture. Since this standard does not define any methods, tools or processes, Proviado could provide therefore a powerful framework in the context of PAIS.
Some view-building approaches deal with inter-organizational processes and apply views to create abstractions of private processes just hiding implementation details [14, 15, 16, 17]. Thereby views are specified by the designer.

[18] presents an approach with predefined view types (i.e. human tasks, collaboration view). As opposed to Proviado, it is limited to the specified view types and it is not possible to define own user-specific views. [19] applies graph reduction to verify structural properties of process schemas. Proviado accomplishes this via aggregation and high-level operations. [20] uses SPQR-tree decomposition for abstracting process models. This approach neither provides high-level abstractions nor does it take other process aspects (e.g. data flow) into account.

[21] determines semantic similarity between activities by analysing the structural information of a process model. The discovered similarity is used to abstract the given process model. However, the approach neither distinguish between different user perspectives on a process model nor provides concepts to manually create process views.

An approach for building aggregated views is provided by [22]. It proposes a two-phase procedure for aggregating parts of a process model that must not be exposed to public. However, it focuses on block-structured graphs and neither data flow nor attributes are considered.

Implementations of view models for monitoring purposes are presented in [23, 24]. These approaches focus on the mapping between original instance and process view at runtime. Respective views have to be pre-specified manually by the designer.

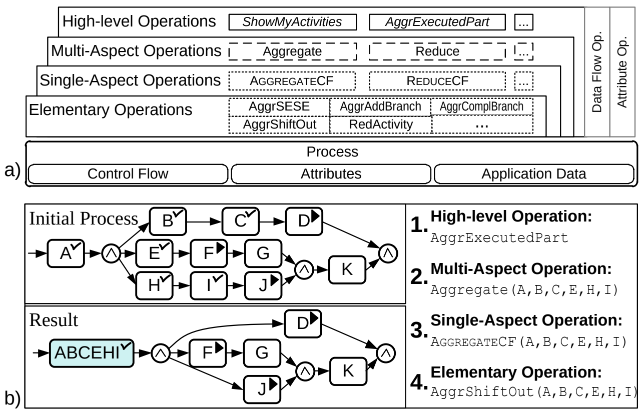

Figure 9: Multi-Layer view operations

Proviado provides a holistic framework for user-centric view-building with elementary and highl-level operations on both control and data flow. Additionally, it takes run-time information into account. None of the existing approaches covers all these aspects. Further, existing approaches for building views are based on rigid constraints not taking practical requirements into account. For example, our first design of a visualization-oriented view mechanism was based on reduction and aggregation techniques for block-structured process graphs [10]. Presenting this solution to business users, however, we figured out that block-structured aggregation does not always meet the practical requirements coming with the visualization of large processes. For this reason Proviado allows to flexibly specify the acceptable degree of imprecision. A validation of Proviado was conducted in the automotive domain, where users are confronted with with complex, long-running development processes [10].

6. SUMMARY AND OUTLOOK

We introduced the Proviado view mechanism and its formal foundation. Further, high-level view building operations provide the required flexibility since process schemas can be adapted to specific user groups. Reduction operations provide techniques hiding irrelevant parts of the process, whereas aggregation operations allow abstracting from process details by aggregating arbitrary sets of activities in one node. Finally, parameterization of the respective operations allows specifying the quality level the resulting view schema must comply with. This enables adaptable process visualization not feasible with existing approaches. We have implemented large parts of the described view mechanism in a prototype. For usability reasons it is important to provide appropriate methods for defining and maintaining process views. Therefore, we are working on a comprehensive set of user-oriented, high-level operations as well as on a view definition language. Both will be evaluated in a user experiment.

7. 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** (2009) 47–65

[2] Weber, B., Reichert, M.: Refactoring Process Models in Large Process Repositories. In: Proc. 20th CAiSE, Montpellier, France (2008) 124–139

[3] Weber, B., Reichert, M., Mendling, J., Reijers, H.A.: Refactoring Large Process Model Repositories. Computers in Industry **62** (2011) 467–486

[4] Streit, A., Pham, B., Brown, R.: Visualization support for managing large business process specifications. In: Proc. BPM '05. Volume 3649 of LNCS. (2005) 205–219

[5] Bobrik, R., Reichert, M., Bauer, T.: Requirements for the visualization of system-spanning business processes. In: Proc. DEXA Workshop. (2005) 948–954

[6] Bobrik, R., Bauer, T., Reichert, M.: Proviado - personalized and configurable visualizations of business processes. In: Proc. EC-Web '06. (2006) 61–71

[7] Reichert, M., Bassil, S., Bobrik, R., Bauer, T.: The Proviado Access Control Model for Business Process Monitoring Components. EMISA International Journal **5** (2010) 64–88

[8] Bobrik, R., Reichert, M., Bauer, T.: View-based process visualization. In: International Conference on Business Process Management (BPM'07). (2007)

[9] Rinderle, S., Bobrik, R., Reichert, M., Bauer, T.: Business process visualization - use cases, challenges, solutions. In: Proc. ICEIS. (2006) 204–211

[10] Bobrik, R.: Konfigurierbare Visualisierung komplexer Prozessmodelle. Phd thesis, Ulm University (2008) (in German).

[11] Liu, D.R., Shen, M.: Workflow modeling for virtual processes: an order-preserving process-view approach. Information Systems **28** (2003) 505–532

[12] Shen, M., Liu, D.R.: Discovering role-relevant process-views for recommending workflow information. In: DEXA'03. Volume 2736 of LNCS. (2003) 836–845

[13] IEEE 1471-2000: Recommended Practice for Architectural Description for Software-Intensive Systems. IEEE Standards Association (2000)

[14] Chebbi, I., Dustdar, S., Tataa, S.: The view-based approach to dynamic inter-organizational workflow cooperation. Data Knowl. Eng. **56** (2006) 139–173

[15] Chiu, D.K.W., Cheung, S.C., Till, S., Karlapalem, K., Li, Q., Kafeza, E.: Workflow view driven cross-organizational interoperability in a web service environment. Inf. Techn. and Mgmt. **5** (2004) 221–250

[16] Kafeza, E., Chiu, D., Kafeza, I.: View-based contracts in an e-service cross-organizational workflow environment. In: Techn. E-Services (TES '01). (2001)

[17] Schulz, K.A., Orlowska, M.E.: Facilitating cross-organisational workflows with a workflow view approach. Data Knowl. Eng. **51** (2004) 109–147

[18] Tran, H.: View-Based and Model-Driven Approach for Process-Driven, Service-Oriented Architectures. TU Wien, PhD thesis (2009)

[19] Sadiq, W., Orlowska, M.E.: Analyzing process models using graph reduction techniques. Information Systems **25** (2000) 117–134

[20] Polyvyanyy, A., Smirnov, S., Weske, M.: The Triconnected Abstraction of Process Models. In: Proc. 7th Int'l Conf. Business Process Management. (2009)

[21] Smirnov, S., Reijers, H., Weske, M.: A Semantic Approach for Business Process Model Abstraction. In: Advanced Information Systems Engineering. Volume 6741. (2011) 497–511

[22] Eshuis, R., Grefen, P.: Constructing customized process views. Data Knowl. Eng. **64** (2008) 419 – 438

[23] Shan, Z., Yang, Y., Li, Q., Luo, Y., Peng, Z.: A light-weighted approach to workflow view implementation. In: Proc. APWeb. (2006) 1059–1070

[24] Schumm, D., Latuske, G., Leymann, F., et al.: State Propagation for Business Process Monitoring on Different Levels of Abstraction. In: Proc. 19th ECIS, Helsinki, Finland (2011)