Evolving Process Views

Rik Eshuis\textsuperscript{a}, Alex Norta\textsuperscript{b}, Raoul Roulaux\textsuperscript{a}

\textsuperscript{a}Eindhoven University of Technology, the Netherlands
\textsuperscript{b}Department of Informatics, Tallinn University of Technology, Estonia

Abstract

Context: Process views support the paradigm of Business Process Outsourcing, in which providers perform business processes on behalf of their clients. A public process view shields secret or irrelevant details from a private, internal business process, thus allowing a provider to reveal only relevant, non-confidential parts of its business process to its clients. Providers can change their internal business processes that may result in inconsistencies with the corresponding process views.

Objective: This paper aims to develop an approach for propagating changes from an internal, private process to its public process view, such that the internal process and its process view remain consistent.

Method: We develop the approach in a formal way. Definitions of process models and process views are based on BPEL, the standard language for realizing process models using state-of-the-art service-oriented technology. We validate the feasibility of the approach by showing how it can be supported by a conceptual system architecture.

Results: The approach relies on two key results. First, a formal characterization of the set of private changes to an internal process, i.e., changes that do not need to be propagated to the process view. Second, a characterization of the non-private changes that can safely be propagated from an internal process to its process view such that they remain consistent. Other non-private changes result in an internal process and a process view that are not consistent. The approach

\textsuperscript{Email addresses: } h.eshuis@tue.nl (Rik Eshuis), alex.norta.phd@ieee.org (Alex Norta)
is supported by a system architecture for process-based business collaboration.

Conclusion: The approach supports providers in deciding if and how changes to a private, internal process propagate to its public process view such that the process view and the internal process remain consistent. The approach allows clients to monitor a public process view such that they can safely track changes made to a private, internal process.

Keywords: Inter-organizational; business process management; process evolution; process visibility; smart contract; e-governance; decentralized autonomous organization

1. Introduction

Business Process Outsourcing is a paradigm in which a provider performs, or coordinates a process on behalf of its client, the consumer. The outsourced process is not a black box for the consumer, as the consumer needs status information to coordinate its other processes. Some parts of the performed process in the provider domain may be confidential, or irrelevant for the consumer. To solve this problem, the notion of process views has been proposed [1, 2, 3]. A process view is an abstraction of an internal process, thus allowing consumers to monitor the progress of process instances, while at the same time shielding private or irrelevant parts of the provider process from the consumer. Process views are also used to visualize different end-user perspectives on internal processes [4].

Process views can be defined in two different ways: as virtual or materialized entities. Virtual process views are projections of internal processes: they do not exist as entities independent from internal processes [1, 5]. However, in the context of outsourcing, it is natural to consider materialized process views [6, 7], which exist independent from the internal processes they abstract from. Materialized process views correspond to smart contracts [8, 9] that specify the work to be performed on behalf of the consumer [11]. A smart contract is a legally enforceable agreement in which two or more parties commit to certain obligations in return for certain rights [12]. Contracts are instruments
for organizing business collaborations. Smart contracts aim at using information technologies to significantly improve the efficiency and effectiveness of paper contracts, allowing companies to support newly emerging business paradigms such as for cyber-physical systems [10], while still being legally protected. Moreover, using materialized process views helps to establish outsourcing relationships by supporting matchmaking between offered and requested process views [13, 14]. Notably, materialized process views are an integral part of the eSourcing-framework [13, 15] for harmonizing on an external layer the intra-organizational business processes of a service consuming and one or many service providing organizations into a B2B supply-chain collaboration. Important elements of eSourcing are the support of different visibility layers of corporate process views for the collaborating counterpart and flexible mechanisms for service monitoring and information exchange.

A key feature of business processes in modern organizations is that they frequently change [16]. Process-oriented information systems are increasingly capable of supporting such changes [17]. If an internal process changes, the process view needs to be aligned with the change. If the process view is materialized, this means the view is changed based on the change in the internal process. Such a procedure is similar to amending a contract between two parties by explicitly mentioning the change applied to the old contract, allowing the client to safely track the change. Current process view approaches [18, 19, 4, 20, 21, 13, 2, 5, 3, 6, 7] do not support the propagation of changes made to internal processes to changes in materialized process views.

To fill this gap, this paper presents a formal approach that defines how changes to an internal process propagate to changes to a materialized process view such that both remain consistent (see Figure 1). Input to the approach is a change to an internal process that has a consistent process view, i.e., there exists an abstraction relation between them. The approach defines the resulting change to the process view and the resulting change to the abstraction relation such that a changed internal process view and the changed view are consistent again. Thus, the changed process view is an abstraction of the changed internal
process.

The approach distinguishes between private and public (non-private) changes. Private changes to an internal process do not affect the view, so these changes result in empty changes for the view. We formally characterize which changes are private and which are public. We show that for some of the public changes, the changed view is not an abstraction of the changed internal process. Such changes should be blocked or modified. Finally, we formally characterize the subset of public changes to an internal process that can be safely propagated to its process view. We validate the feasibility of the approach by showing how it can be supported by a conceptual system architecture.

We focus in this paper on structured processes [22], since they are close to BPEL [23], which is the standard language for expressing process views using state-of-the-art service-oriented technology. Moreover, structured processes allow for a concise and simple definition of the change operations as well as the abstraction relation between an internal process and its process view. Several approaches have been defined to convert an unstructured into a structured process while preserving its behavior [24, 25].

This paper is organized as follows. Section 2 discusses related work. Section 3 presents a motivating example that illustrates the problem of propagating changes from internal processes to process views. Section 4 formally defines
structured processes as process trees. We also define several change operations on process trees in a concise manner. Next, Section 5 formally defines process views. Section 6 formally characterizes private changes to an internal process that do not affect a process view, which abstracts the internal process. Section 7 characterizes under which conditions and how public (non-private) changes to an internal process can be propagated to a view. The other applied public changes result in an inconsistent view and therefore should be blocked. Section 8 validates the feasibility of the approach and Section 9 discusses the findings of this paper. Finally, Section 10 presents conclusions and future work.

2. Related work

Process views have received increased research attention in recent years in the context of inter-organizational business process management [19, 18, 20, 1, 2, 21, 26, 13, 27, 3, 6, 28, 14, 29, 7], where a provider performs a business process on behalf of a consumer. Most of these papers deal with the design-time aspects of process views [19, 18, 20, 2, 21, 26, 13, 27, 3, 28, 29, 7]. Most process view design approaches focus on deriving a public process view from a private, internal process [4, 20, 2, 21, 26, 13, 3, 28, 14, 29, 7] while others study the derivation of internal processes from public views [19, 18, 2]. Other papers [1, 6] focus on how to support a process view at run-time and do not address how to construct a process view. All these papers consider static process views, so the constructed process views, once deployed, do not change. Process views have also been applied to offer personalized, role-specific views on a common process [4, 5]. Smirnov et al. [30] provide a survey of different static abstraction approaches. All these papers study the relation between an internal process and a process view, but not consider change propagation from internal processes to process views, which is studied in this paper.

From a conceptual point of view, process views and their underlying internal process models are expressed in separate models. From an implementation point of view, process views can be implemented in separate models [31, 29, 7]. We call such process views materialized. Alternatively, process views can be implemented
as projections of internal processes [1, 5]. In that case, process views are virtual, as they are defined by abstraction relations defined on top of internal process models. In this paper, we focus on materialized process views, as we motivated in Section 1.

Other related approaches that deal with changes in the context of process views [32, 5, 33] all consider virtual process views. For a virtual process view, the abstraction relation defines the process view. Therefore, changing the abstraction relation results in a new virtual process view. However, we consider materialized process views. In that case, a private change to an internal process needs to be propagated to a corresponding public change to its materialized process view. This separation between private and corresponding public changes allows a consumer to safely track in the public process view public changes that abstract from private changes made to confidential parts of internal processes. A complication that arises for materialized process views is that an internal process and its process view may no longer be consistent after applying changes. This motivates a characterization of internal changes that cannot be propagated to process views, as we did in this paper. This issue is not applicable to virtual process views [32, 5, 33].

Weidlich et al. [34] study how to propagate changes between aligned process models. Given a change in a process model, they focus on identifying the relevant part of the aligned process model that needs to be changed. In our paper, an abstraction function identifies the part of the process view requiring change, without separate identification. Weidlich et al. ignore alignment between the changed processes as well as fail cases that we both cover with the approach in this paper.

The BPEL standard [23] distinguishes between abstract and executable processes, where abstract processes correspond to process views. Still, there are no concrete guidelines for relating abstract and executable processes, let alone changing a process. Instead, the results of this paper are directly applicable to BPEL as we show by implementing a prototype tool that reads BPEL protocols (see Section 8).
The concept of process views originates from views in databases [35]. The ANSI/SPARC reference architecture for databases [36] distinguishes between conceptual database schemas, internal schemas that realize conceptual schemas and external schemas called views that are user specific and therefore omit certain details of the conceptual schema. This distinction inspires the initial work on process views [1, 2]. The notion of materialized view also originates from databases [37, 36]. The data of a materialized database view is stored separately from the data in the underlying database, whereas the data of a non-materialized database view is queried directly from the database. The problem how to keep the data of a materialized view up-to-date with the underlying database is called view maintenance [37]. The related problem of keeping the schema of a materialized view up-to-date after changing the underlying database schema is called view synchronization [38]. The approach in this paper can be seen as view synchronization for process models.

Finally, there are approaches in other domains that also consider abstractions for structures and their associated operations. In category theory [39], a homomorphism is a function that maps a structure and its operations into another structure with operations while preserving characteristics of the original structure and operations in the image. Structures relate to process models, operations to changes on process models, and homomorphisms are similar to abstraction functions. Figure 1 resembles a commutative diagram in category theory [39], meaning that for an internal process to which a change is applied, the derived process view found by applying the derived change to the process view is identical to the derived process view found by abstracting the changed internal process. A more concrete approach is abstract interpretation [40]. In order to perform static analysis on programs, abstract interpretation defines abstract executions for these programs that approximate the actual program executions. Though these other approaches share some underlying principles with the approach developed in this paper, they do not apply to the domain of business processes.
3. Motivating example

We introduce the main concepts and problem by means of a simple example that we revisit in the sequel of this paper. The running example is about a process from a headhunter company for handling job applications on behalf of service consumers that wish to hire personnel (Figure 2). For other, more elaborate examples about industry business-to-business (B2B) collaborations, we refer the reader to results from the EU-project CrossWork [41, 42] that uses also externally projected process-views for matching. These examples stem from truck production where an original equipment manufacturer (OEM) outsources process parts to small- and medium sized enterprise (SME) service providers that are part of an automotive cluster. Example application domains for business-process outsourcing are for instance logistics (freight forwarding), manufacturing (compliance and regulatory reporting), or finance and accounting (processing of accounts payable and receivable) [43].

We use BPMN (Business Process Model and Notation) 2.0 [44] to graphically visualize processes. BPMN uses basic modeling elements and has become the de-facto standard for modeling business processes in industry. In Figure 2, the included legend explains the BPMN notation elements for the running case. The start event is the initial process event and the end event terminates the process. An exclusive gateway evaluates the state of the business process and based on the condition, splits the control-flow into one of the two or more mutually exclusive paths. A parallel gateway does not evaluate any condition or event and instead represents two or more concurrent process branches. Finally, a normal task is a single action that occurs in a business process. Since we consider structured processes, gateways come in split-join pairs [45].

In the running example, the headhunter company has an internal process for processing job applications (Figure 2(a)). Based on this internal process, the company offers a personnel-evaluation service to hiring service consumers (process view in Figure 2(b)). The assumption in this example is that not all details of the internal hiring-process are of interest to the service consumer.
Alternatively, the service provider may hide process details from the service consumer, for instance about screening applications. These details are key to the headhunter company to gain a competitive advantage over competitors and therefore should remain secret.

The client of the provider sees the projected public process view in Figure 2(b) based on which he decides to be a service consumer. We consider activities of the process view and the corresponding internal process sharing the same label to be identical. Shaded activities in Figure 2(a) are private and either omitted (Screen application) in the process view, or aggregated in the process view into opaque activities Detailed screening and Make proposal of Figure 2(b), as the dashed lines indicate. The internal process and the public process view agree on the ordering of the non-omitted activities and are therefore consistent.

Suppose the provider company introduces two changes to the internal process (Figure 3(a)). First, a selected candidate always goes through an assessment. The Perform assessment activity is private. Second, the company consults references of a candidate before preparing and sending a proposal. Activity Consult references is public and therefore visible to the client.

Now the problem arises how the change affects the process view in Figure 2(b).
In particular, the following questions need to be answered: Is the original view consistent with the changed internal process in Figure 3(a)? If the original view is not consistent, can the view be changed such that it is consistent with the changed internal process? If the original view can be changed, how does the internal change translate to a change at the view?

In the sequel, we develop a framework that answers these concrete questions for our running example. In general, the framework characterizes and defines:

- The set of internal changes that cause a change in the process view.
- The translation of internal changes into external changes of the view.
- The set of internal changes that result via the corresponding external change in a changed view that is consistent with the changed internal process.

The next section formally defines process models and change operations.
4. Preliminaries

This section introduces definitions for process models (Section 4.1) and change operations on process models (Section 4.2). Section 4.1 also introduces behavioral relations on process models that we use for relating process views with their underlying internal processes.

4.1. Process trees

The introduction considers structured processes such as BPEL [23]. To simplify the definitions in the sequel, we formalize the syntax of structured process models as process trees [13] (see Definition 1). Leaves specify the execution of basic activities and internal nodes specify ordering constraints on their child nodes. There are several types of internal nodes. A SEQ-node specifies sequential execution of children nodes. An AND-node specifies concurrent execution, corresponding to the BPEL-flow construct. A XOR-node specifies a choice: one of its child nodes is executed. Finally, a LOOP-node specifies structured repeat-until loops. To simplify the exposition, we do not consider synchronization links here, while they can be included too [46].

We now present a formal definition for process trees [46, 21].

Definition 1 (Process tree). A process tree $P$ is a tuple $(N, A, C, type, parent, succ)$ where:

- $N$ is the set of nodes, partitioned into sets $A$ and $C$;
- $A$ is the set of activities or tasks;
- $C$ is the set of control nodes;
- $type : C \rightarrow \{ SEQ, AND, XOR, LOOP, OPAQUE \}$ is a function that specifies the type for each control node. If $type(n) = t$, we write that $n$ is a $t$-node. A SEQ-node specifies a sequential behavior, an AND-node parallel behavior, a XOR-node exclusive behavior, where the choice is made internally, and a LOOP-node specifies iterative behavior; an OPAQUE-node is
only used for process views and specifies that the node aggregates multiple
nodes from an underlying internal process (see the next section).

- **parent**: \( A \cup C \rightarrow C \) is a partial function that specifies for a node \( n \in N \) its
  super node \( \text{parent}(n) \). An activity cannot be a parent node;

- **succ**: \( N \rightarrow N \) is a partial function that specifies sequential ordering within
  SEQ-nodes. If \( n' = \text{succ}(n) \) then \( n \) and \( n' \) have the same SEQ parent and
  \( n \) is directly succeeded by \( n' \).

We show in Fig. 4 a process tree that is visualized in Fig. 2(a) as a BPMN2 [44]
model. Each node of the tree is visualized as a BPMN fragment with a single
entry and a single exit node. Each SEQ-node with child nodes \( c_1, \ldots, c_n \) is
represented by the BPMN fragments of its child nodes \( c_i \) with an arrow from the
exit node from \( c_i \) to the entry node of \( c_{i+1} \) for \( 1 \leq i < n \). Each other internal
node is visualized as a BPMN fragment in which the entry and exit nodes are
gateways of the same type (AND/XOR).

We require that the **parent** function induces a tree, so each node has one
parent, except one unique root node \( r \) that has no parent. These constraints
ensure that nodes form a tree structure with root \( r \). Leaves of the tree are
activities while the internal nodes are control nodes. By **parent**\(^*\) we denote the
reflexive transitive closure of **parent**. If \( n \in \text{parent}^*(n') \) then either \( n \) is a direct
or indirect parent of \( n' \) or \( n = n' \). In that case, \( n \) is **ancestor** of \( n' \).

To define the consistency notion between process models, we introduce
behavioral relations on nodes [13]. We first define auxiliary functions on the
syntax of process trees.
Definition 2 (Lowest common ancestor). For a set $X$ of nodes, the lowest common ancestor of $X$, denoted $\text{lca}(X)$, is the unique node $l$ such that (i) $l$ is ancestor of each node in $X$, and (ii) every other node $y$ that is ancestor of each node in $X$ is ancestor of $l$ too.

Since nodes are arranged in a tree, every set of nodes has a unique least common ancestor.

Based on the notion of $\text{lca}$, we define behavioral relations on nodes.

Definition 3 (Behavioral relations). The before relation $<$ is induced by $\text{SEQ}$-nodes. Given two nodes $n, n' \in N$, we have $n < n'$ if and only if

- node $l = \text{lca}\{n, n'\}$ is a $\text{SEQ}$-node, and
- node $l$ has children $c_n, c_{n'}$ such that $c_n$ is ancestor of $n$, so $c_n \in \text{parent}^*(n)$, $c_{n'}$ is ancestor of $n'$, $c_{n'} \in \text{children}^*(n')$, and $c_{n'}$ is a successor of $c_n$ according to $\text{succ}$, so $c_{n'} \in \text{succ}^*(c_n)$.

Next, we define relations for choice and parallelism. For two nodes $n, n' \in N$,

- $n \smallvdash n'$ if and only if node $l = \text{lca}\{n, n'\}$ is an $\text{XOR}$-node.
- $n \& n'$ if and only if node $l = \text{lca}\{n, n'\}$ is an $\text{AND}$-node.

Finally, we define a containment relation: $n \triangleleft n'$ if and only if $n'$ is ancestor of $n$ and $n \neq n'$.

It is easy to show that for every pair of nodes $n_1, n_2$ in a process, there is exactly one behavioral relation $\otimes \in \{<, \smallvdash, \&\}$ such that $n_1 \otimes n_2$. Furthermore, $\smallvdash$ and $\&$ are symmetric while $<$ and $\triangleleft$ are asymmetric.

Example 1. For the process tree in Fig. 4, we have among others $\text{Screen application} < \text{Invite for interview}$, $\text{Screen application} < \text{Collect feedback}$, $\text{Screen application} < \text{Reject immediately}$, $\text{Invite for interview} < \text{Collect feedback}$, $\text{Invite for interview} \smallvdash \text{Reject immediately}$, and $\text{Collect feedback} \triangleleft \text{Reject immediately}$.
4.2. Change operators

We use basic change operations on process models for insertion and deletion of an activity. Every insert operation on a process tree $P$ is in the context of an existing node $n$ of $P$. Each insert operation adds control flow relative to $n$ and may also add an activity $x$. An insert operation can add an activity $x$ before, after, in parallel, or alternatively to $n$. An insert operation that does not add an activity can make $n$ optional or repeating. A delete operation removes an activity from the process model. Below we formalize both types of operation in the context of process trees. In future work we will consider more advanced operations as proposed for instance by Weber et al. [47].

**Definition 4 (Inserting activity).** Let $P$ be a process tree, let $n$ be a node in $P$ and let $x$ be an activity that does not occur in $P$, and let $insertType \in \{\text{before}, \text{after}, \text{parallel}, \text{choice}\}$. Then $insertActivity(P, n, x, insertType) = (N', A', C', type', parent', succ')$ where:

- $N' = N \cup \{x, \text{compound}_x\}$;
- $A' = A \cup \{x\}$;
- $C' = C \cup \{\text{compound}_x\}$;
- $type' = \begin{cases} \text{SEQ} & \text{if } insertType = \text{before} \\ \text{SEQ} & \text{if } insertType = \text{after} \\ \text{AND} & \text{if } insertType = \text{parallel} \\ \text{XOR} & \text{if } insertType = \text{choice} \end{cases}$;
- $parent' = parent \oplus \{x \mapsto \text{compound}_x, n \mapsto \text{compound}_x, \text{compound}_x \mapsto parent(n)\}$;
- $succ' = succ \oplus \begin{cases} \{x \mapsto n\} & \text{if } insertType = \text{before} \\ \{n \mapsto x\} & \text{if } insertType = \text{after} \\ {} & \text{otherwise}, \end{cases}$

where $\text{compound}_x$ is a fresh node not in $N$ and operator $\oplus$ denotes function overriding.
Figure 5: Inserting (a)-(e) and deleting (f) activity C in Process P.

Figure 5(a)-(e) illustrates different variations of insertActivity. To the very left in Fig. 5(a), we show a base process P in which fresh activity C is to be inserted in the context of node B (cf. Definition 4). The next four parts of Fig. 5(b)-(e) show the results of carrying out the insertActivity operation with insertType variations after, before, parallel and choice respectively.

The next definition specifies how n can become optional or repeating.

**Definition 5 (Inserting control flow).** Let P be a process tree, let n be a node in P, and let insertType ∈ \{choice, loop\}. Then insertControl(P, n, insertType) = (N', A', C', type', parent', succ') where:

- N' = N ∪ \{compoundn\};
- A' = A;
- C' = C ∪ \{compoundn\};
- type' = type ⊕ \begin{cases} \{compoundn \mapsto XOR\} & \text{if insertType=choice} \\ \{compoundn \mapsto LOOP\} & \text{if insertType=loop} \end{cases};
- parent' = parent ⊕ \{n \mapsto compoundn, compoundn \mapsto parent(n)\};
- succ' = succ,

where compoundn is a fresh node not in N.

**Definition 6 (Deleting node).** Let P be a process tree, and let n be a node in P. Then deleteNode(P, n) = (N', A', C', type', parent', succ', link') where:
\[ N' = N \setminus \{ n' \mid \text{parent}^*(n') = n \}; \]
\[ A' = A \setminus \{ n' \mid \text{parent}^*(n') = n \}; \]
\[ C' = C \setminus \{ n' \mid \text{parent}^*(n') = n \}; \]
\[ \text{type}' = \text{type} \cap (N' \times \{ \text{SEQ}, \text{AND}, \text{XOR}, \text{LOOP}, \text{OPAQUE} \}); \]
\[ \text{parent}' = \text{parent} \cap (N' \times N'); \]
\[ \text{succ}' = (\text{succ} \cap (N' \times N')) \cup \{ x \mapsto y \mid \text{succ}(x) = n \land \text{succ}(n) = y \}; \]

Operation deleteNode is illustrated with the example in Fig. 5(f), in which activity C is marked for deletion at the lefthand side, and has been removed at the righthand side.

5. Process views

Several approaches have been proposed for constructing process views from internal processes [48, 21, 4, 7]. We do not select any of these approaches, since for the purpose of this paper, the construction of the process view from the internal process is not relevant. Instead, we focus on the declarative abstraction relation that defines the relation between an internal process and its view. The abstraction relation is the implicit effect of applying any of the existing approaches for constructing process views and has the following characteristics.

Since a process view has a higher abstraction level than its underlying internal process, an activity in the view can decompose into multiple, connected activities in the internal process. Then the activity in the view is an aggregate and has type OPAQUE. This means the abstraction relation is a function that maps internal activities to activities in the process view. If some activities in the internal process have no counterpart in the view, the abstraction function is partial, meaning that those activities are omitted in the view. Furthermore, the abstraction function must be surjective, so cover all activities from the process view, to ensure that each activity in the view is realized by one or more internal activities.
Next, to ensure that the internal process and its view are consistent, the abstraction function must preserve the ordering relations from the internal process in the process view. For example, if in the internal process activity $A$ is performed before $B$ and both activities are part of different aggregates in the process view, then in the process view, the aggregate for $A$ needs to be performed before the aggregate for $B$. In terms of process views, the process view preserves the relevant behavioral relations from the internal process.

We next formally define process views in terms of this abstraction function.

**Definition 7 (Process view).** Let $V, P$ be two process trees. Then $V$ is a **process view** on $P$ if and only if there exists a surjective, partial abstraction function $\alpha : N_P \to N_V$ such that for each $n \in N_P$

- $\alpha(n)$ is either identity and type preserving ($\alpha(n) = n$ and $\text{type}_V(\alpha(n)) = \text{type}_P(n)$), or an aggregate ($\alpha(n) \in A_V$ and $\text{type}_V(\alpha(n)) = \text{OPAQUE}$); and

- for every pair of nodes $n_1, n_2 \in \text{dom}(\alpha)$, $n_1 \otimes_P n_2$ if and only if either $n_1$ and $n_2$ are in the same aggregate, so $\alpha(n_1) = \alpha(n_2)$, or $\alpha(n_1)$ and $\alpha(n_2)$ are distinct and have the same behavioral relation in $V$ as $n_1$ and $n_2$ in $P$: $\alpha(n_1) \neq \alpha(n_2)$ and $\alpha(n_1) \otimes_V \alpha(n_2)$ such that $\otimes_V = \otimes_P$.

In the definition, $\text{dom}(\alpha)$ is the domain of function $\alpha$, so the set of nodes from the internal process that are represented by the nodes from the process view. As explained above, function $\alpha$ is surjective, since every node in the view must be realized by one or more nodes from the internal process.

If $\alpha(n) = n$ then $n$ is a public activity in the internal process. Otherwise, if $\alpha(n) \neq n$ then $n$ is aggregated into opaque activity $\alpha(n)$. The abstraction function $\alpha$ may be partial: in that case, there are internal nodes $n$ that are not defined by $\alpha$, so $n \notin \text{dom}(\alpha)$. Such a node $n$ is not represented by any node in the process view, for instance **Screen application** in Fig. 2(a). If $n \notin \text{dom}(\alpha)$ and therefore $\alpha(n)$ is not defined, we write $\alpha(n) = \bot$. 

17
Example 2. Figure 2(b) is a process view of Fig. 2(a). The abstraction function $\alpha$ is defined as follows: $\alpha($Screen application$) = \perp$, $\alpha($Invite for interview$) = \alpha($Collect feedback$) = $Detailed screening, $\alpha($Prepare proposal$) = \alpha($Send proposal$) = $Make proposal and for all other nodes $n$, $\alpha(n) = n$.

The behavioral relations in (a) and (b) are consistent for the nodes in $\text{dom}(\alpha)$: for instance for the process in Fig. 2(a), among others Invite for interview $\triangledown$ Reject immediately while for (b) Detailed screening $\triangledown$ Reject immediately, and for (a) Invite for interview $<$ Collect feedback, which belong to the same aggregate Detailed screening in (b).

We next analyze for which changes to internal business processes their public process views remain consistent.

6. Private changes

This section defines which changes to an internal process do not affect its process view. Denote with $\delta$ an arbitrary change operator as defined in Section 4.1. Applying $\delta$ to internal process $P$ results in a modified process $P'$ and a modified abstraction function $\alpha'$. We study in this section for which changes $\delta$ the process view $V$ remains valid for $P'$ using modified abstraction function $\alpha'$ (see Fig. 6). As explained in Section 4.1, we consider insertion and deletion of activities. For insertion, we consider a set $F$ of fresh activities, partitioned into public activities $F_{public}$ and private activities $F_{private}$.

The next definition details how a change in an internal process that has a process view leads to a change in the abstraction function $\alpha$ that relates the
internal process and the process view. The abstraction function only changes with respect to nodes that are added to, or removed from the internal process.

**Definition 8 (Change of abstraction function $\alpha$).** Let $V, P$ be two process trees such that $V$ is a view on $P$ using abstraction function $\alpha$. Let $\delta$ be a change operation that changes $P$ into $P'$.

If $\delta$ inserts fresh activity $x$ in the context of existing node $n$, abstraction function $\alpha'$ is defined as follows: for every $y \in N_P \cup \{x, \text{compound}_x\}$,

$$
\alpha'(y) = \begin{cases} 
\alpha(y) & \text{, if } y \in N_P \\
\alpha(n) & \text{, if } y \in \{x, \text{compound}_x\} \text{ and } x \in F_{\text{private}} \text{ and } \alpha(n) \neq n \\
\bot & \text{, if } y \in \{x, \text{compound}_x\} \text{ and } x \in F_{\text{private}} \text{ and either } \alpha(n) = n \text{ or } \alpha(n) = \bot \\
y & \text{, if } y \in \{x, \text{compound}_x\} \text{ and } x \in F_{\text{public}}
\end{cases}
$$

If $\delta$ inserts additional control flow in the context of existing node $n$, abstraction function $\alpha'$ is defined as follows: for every $y \in N_P \cup \{\text{compound}_n\}$,

$$
\alpha'(y) = \begin{cases} 
\alpha(y) & \text{, if } y \in N_P \\
\alpha(n) & \text{, if } y = \text{compound}_n \text{ and } (\alpha(n) \neq n \text{ or } \alpha(n) = \bot) \\
\text{compound}_n & \text{, if } y = \text{compound}_n \text{ and } \alpha(n) = n
\end{cases}
$$

If $\delta$ deletes node $n$, then $\alpha' = \alpha \cap ((N_P \setminus \{n\}) \times N_V)$.

The definition states that if a fresh private activity $x$ is inserted in the context of $n$, then if $n$ belongs to an aggregate ($\alpha(n) \neq n$) then $x$ belongs to the same aggregate. Otherwise, if $n$ does not belong to an aggregate, $x$ is omitted in the view $V'$. If $x$ is a fresh public activity, $x$ always is inserted in the view.

If additional control flow is inserted in the context of node $n$, then the new node $\text{compound}_n$ is treated similarly as $n$ by $\alpha'$: if $n$ is aggregated, then $\text{compound}_n$ belongs to the same aggregate. Otherwise, $\text{compound}_n$ is either revealed or omitted from the view.

**Example 3.** Consider the two changes made in Figure 3(a) to Figure 2(a): the insertion of Perform assessment (see Figure 7) and Consult references (see
Figure 8). Since Perform interview is private and part of aggregate Detailed screening, \(\alpha'(\text{Perform assessment}) = \text{Detailed screening}\). Activity Consult references is public. Therefore \(\alpha'(\text{Consult references}) = \text{Consult references}\).

Note that the definition only specifies how a change of an internal process affects the abstraction function. The definition does not state whether or not a change is valid. To answer that question, we first need to identify the effect of the change on the view.

The easiest case is if the modified abstraction function is still valid for the original process view \(V\), so \(V\) is a valid process view for the changed internal process \(P'\). In that case, the change \(\delta\) is private and trivially valid. We define private changes declaratively in terms of the abstraction function.

**Definition 9 (Private change).** Let \(V, P\) be two process trees such that \(V\) is a view on \(P\) using abstraction function \(\alpha\). Let \(\delta\) be a valid change operation on \(P\) leading to \(P'\). If \(V\) is a valid view on \(P'\) using abstraction function \(\alpha'\), then \(\delta\) specifies a private change, i.e., the change is not visible in the public view.

We now formally characterize which type of concrete changes in internal processes are private.
Theorem 1. Let $V, P$ be two process trees such that $V$ is a view on $P$ using abstraction function $\alpha$. Let $\delta$ be a valid change operation on $P$. Then $\delta$ is a private change if and only if the following three conditions for $\delta$ hold:

- if $\delta$ inserts activity $x$ in the context of node $n$, then $x \in F_{private}$;
- if $\delta$ inserts additional control flow in the context of node $n$, then either $\alpha(n) \neq n$ or $\alpha(n) = \bot$;
- if $\delta$ deletes node $n$ and $\alpha(n) \neq \bot$, then there is another node $n' \in N_P$ such that $\alpha(n) = \alpha(n')$.

The proof is in the Appendix of this paper. As to be expected, the theorem asserts that inserting activity $x$ is a private change if and only if activity $x$ is private. Inserting additional control flow is only private if the context node is hidden or aggregated in the view. Furthermore, deletion is only allowed if the view contains a corresponding aggregate that after the deletion still refers to activities from the internal process.

Example 4. (Continued from Example 3) The insertion of activity Perform assessment (see Figure 7) is a private change, but the insertion of public activity Consult references (see Figure 8) is not. Consequently, the process in Figure 2(b) is not a view of the internal process in Figure 3(a).
7. Public changes

This section studies public changes, i.e. changes that are not private. We define how a public change \( \delta \) applied to an internal process \( P \) translates into a change \( \delta' \) applied to process view \( V \) (see Fig. 9). Definition 8 already specifies how any change \( \delta \) applied to \( P \) leads to a modified abstraction function \( \alpha' \). However, we show in this section that the resulting view is not always valid for \( P' \), i.e., there can be a mismatch between \( \alpha' \) and \( \delta' \). We formally characterize which public changes can be translated properly into changes applied to the process view \( V \) such that the changed internal process and changed view are consistent.

**Definition 10 (Public change).** Let \( V, P \) be two process trees such that \( V \) is a view on \( P \) using abstraction function \( \alpha \). Let \( \delta \) be a valid change operation on \( P \) leading to \( P' \) that is not a private change. Then \( \delta \) specifies a public change.

By Theorem 1, we can derive the following properties for public changes. Each public change that is an insert operation inserts a public activity. Consequently, we ignore insertion of private activities in the sequel. Moreover, if a public change inserts additional control flow in the context of node \( n \), then \( \alpha(n) = n \). Finally, for a public change that deletes node \( n \), \( \alpha(n) \neq \bot \) and there is no other node \( n' \) such that \( \alpha(n) = \alpha(n') \).

We now define how a public change \( \delta \) applied to \( P \) translates into a change \( \delta' \) applied to process view \( V \).
Definition 11 (Change operations for process view). Let \( P, V \) be two process trees such that \( V \) is a view on \( P \) using abstraction function \( \alpha \). Let \( \delta \) be a public change operation that changes \( P \) into \( P' \) leading to modified abstraction function \( \alpha' \). The change operation \( \delta' \) for the process view \( V \) is defined as follows.

If \( \delta \) inserts public activity \( x \) in the context of existing node \( n \) such that \( \alpha(n) \neq \bot \), then \( \delta' \) is the same operation on \( V \) but inserting \( \alpha'(x) \) in the context of \( \alpha'(n) \).

If \( \delta \) inserts additional control flow in the context of existing node \( n \) such that \( \alpha(n) = n \), then \( \delta' \) is the same operation on \( V \). If \( \delta \) deletes existing node \( n \) such that \( \alpha(n) \neq \bot \) and there is not another node \( n' \) such that \( \alpha(n) = \alpha(n') \), then \( \delta' \) deletes node \( \alpha(n) \). Otherwise, \( \delta' \) is not defined.

This definition explains how \( V' \) can be derived from \( V \), based on the public change \( \delta \) applied to \( P \), provided the context node \( n \) has a counterpart in the view. However, it might be that \( V' \) is not a view of \( P' \) based on \( \alpha' \) (cf. Fig. 9). In that case, the change cannot be propagated to \( V \) since \( V' \) and \( P' \) are not in a view relation, and therefore the change to \( P \) should be blocked. Before we can characterize these cases, we first need to define which public changes are valid.

Definition 12 (Valid public changes). Let \( P, V \) be two process trees such that \( V \) is a view on \( P \) using abstraction function \( \alpha \). Let \( \delta \) be a public change operation that changes \( P \) into \( P' \) leading to modified abstraction function \( \alpha' \). Let \( \delta' \) be the change operation, based on \( \delta \), that is applied to \( V \), resulting in view \( V' \). Change \( \delta \) is valid if and only if \( V' \) is a view on \( P' \) using abstraction function \( \alpha' \) (Def. 7).

We illustrate the definition with several examples. First, we revisit the example of Section 3.

Example 5. For Fig. 3(a), the insertion of public activity Consult references before Prepare proposal (see Fig. 8) is valid. By Def. 11 the insertion translates into an insertion of Consult references before Make proposal in Fig. 3(b). The changed view in Fig. 3(b) is consistent with the changed internal process in
Fig. 3(a). In particular, Consult references < Send proposal in Fig. 2(a) while Consult references < Make proposal in Fig. 3(b).

The next example, slightly modified from the example discussed in Section 3, illustrates an invalid change.

Example 6. Figure 10(a) contains two insert operations. The insertion of Consult references is private, and therefore valid by default. Note that according to Def. 8, $\alpha(\text{Consult references}) = \bot$.

However, the insertion of activity Perform assessment in parallel to Perform interview is public and not valid. By Def. 11 the change of Fig. 10 translates into the insertion of activity Perform assessment (see Fig. 7) in parallel to Detailed screening for the original view in Fig. 2(b). The resulting process in Fig. 10(b) is not a view of the changed internal process in Fig. 10(a) according to Def. 7: Perform assessment < Collect feedback in Fig. 10(a) while in the changed view of
The following theorem formally characterizes which public changes $\delta$ are valid, so which public changes to $P$ can be propagated to the view $V$. The proof is in the Appendix.

**Theorem 2.** Let $P, V$ be two process trees such that $V$ is a view on $P$ using abstraction function $\alpha$. Let $\delta$ be a public change operation that changes $P$ into $P'$ leading to modified abstraction function $\alpha'$. Let $\delta'$ be the change operation based on $\delta$ that changes $V$ into $V'$. Then change $\delta$ is valid if and only if an insertion of activity $x$ by $\delta$ in the context of node $n$ implies either

- $\alpha(n) = n$; or
- $\alpha(n) \neq n$ and for every $y \in N_P$ such that $\alpha(y) = \alpha(n)$ and $y \neq n$:
  - if $\delta$ inserts $x$ before $n$, then $n <_P y$;
  - if $\delta$ inserts $x$ after $n$, then $y <_P n$;
  - if $\delta$ inserts $x$ in parallel to $n$, then $n \&_P y$;
  - if $\delta$ inserts $x$ as choice to $n$, then $n \lor_P y$.

The theorem states that a public change is always valid if the change is a deletion or an insertion of additional control flow. If the public change is an insertion of some activity $x$ in the context of node $n$, then the change is valid if either $n$ is public: $\alpha(n) = n$, so $n$ is visible in the view, or if $n$ is part of an aggregate $a$ such that all nodes in $a$ are in the same behavioral relation to $x$ as $n$ after insertion. This is ensured if $n$ is a representative for all the other nodes in $a$, i.e., the behavioral relation of $x$ with $n$ is the same as with all other nodes in $a$. The exact behavioral relation between $n$ and $x$ depends on the type of insertion.

For instance, for Fig. 3(a) the insertion of public activity $x = \text{Consult references}$ before $n = \text{Prepare proposal}$ is valid, since $\text{Prepare proposal}$ is before the other
node in the aggregate, **Send proposal**. But for Fig. 10(a), inserting node **Perform assessment** in parallel to **Perform interview** is not valid since **Perform interview** is not in parallel to the other node in the aggregate, **Collect feedback**.

Invalid changes can be repaired in various ways. The natural repair is to modify the change itself. For instance, if the insert type of the change for Fig. 10(a) is **before** or **after**, the change is valid. Another repair is to let the new activity be private instead of public, since then the change becomes private and private changes are valid by default.

An alternative, more radical repair, is destroying the aggregate in the original view, thus changing the original abstraction function \( \alpha \). For instance, aggregate **Perform assessment** can be destroyed for the view in Fig. 2(b) to make the change in Fig. 10(a) valid. However, such a change will violate the privacy policy of the provider, which caused him to aggregate activities **Perform interview** and **Collect feedback** in the first place. We therefore not believe that such type of change is useful in practice.

Next, we perform a multi-faceted feasibility study by showing which system features and existing information technologies support evolution of process-view based business collaborations.

### 8. Feasibility validation

We validate the feasibility of our evolving process-view approach by answering the following two questions:

**Q1** How can the approach be supported by a conceptual system architecture?

**Q2** How can the conceptual system architecture be realized?

We answer Q1 by adapting an existing system architecture that enables the evolution of collaborating business processes based on process views [15]. Section 8.1 presents the architecture from a structural point of view while Section 8.2 presents the lifecycle that shows how the architecture supports business collaborations in which the process views evolve. We answer Q2 by
8.1. System-architecture feasibility

To enable the setup and enactment of process-view based collaboration evolution, a system must meet a set of requirements. First, there must exist a service that facilitates the matching of service offers from collaborating parties and service requests from consuming organizations. Second, the collaborating parties house internally a component for the distributed binding and enactment of emergency cases. Third, with tool support, the parties must rapidly develop service offers and concrete services. Fourth, each collaborating party is capable of orchestrating its own internal legacy system for automating the collaboration. Finally, due to the heterogeneity of the collaboration, a translation service must exist for bridging the differences (technical, syntactic, semantic, pragmatic) between collaborating parties.

The eSourcing Reference Architecture [15] supports these requirements. Fig-
Table 1: Overview of requirements and components of Fig. 11.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Realized by component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matching service offers and requests</td>
<td>Trusted_Third_Party</td>
</tr>
<tr>
<td>Distributed enactment of emergency cases</td>
<td>eSourcing_Middleware</td>
</tr>
<tr>
<td>Rapid development of services and requests</td>
<td>eSourcing_Setup_Support</td>
</tr>
<tr>
<td>Orchestrating legacy system</td>
<td>Legacy_Management</td>
</tr>
<tr>
<td>Dealing with heterogeneity</td>
<td>Translator</td>
</tr>
<tr>
<td>Dealing with process evolution</td>
<td>all components</td>
</tr>
</tbody>
</table>

Figure 11 depicts the resulting architecture in UML-component diagram notation that takes into account the above listed requirements. The Trusted_Third_Party in the middle satisfies the first requirement and is suitable for the rapid setup phase between collaborating parties. Each party has on an external layer an eSourcing_Middleware for the technical binding after a successful setup that satisfies the second requirement. During the distributed collaboration-enactment, the eSourcing_Middleware exchanges data with the other parties via a security-ensuring gateway. Thus, the eSourcing_Middleware also comprises external workflow- and rules-enactment services that coordinate each other not only internally but also via the gateway with other parties. We assume there exists in each party a conceptual layer with a service for Setup_Support that satisfies the third requirement and comprises tools for not only rapidly internally designing services and rules with the help of pattern libraries [49], but also includes a local verification- and simulation service. Next, each party has an internal layer with a service for Legacy_Management that satisfies the fourth requirement and comprises local workflow- and rules enactment services that coordinate each other for the orchestration of Web-service wrapped internal legacy systems. Finally, the external- and internal enactment services exchange data via a Translator service on the conceptual layer of each party to bridge the heterogeneous collaboration aspects. The Translator satisfies the final requirement and also connects on the conceptual layer with the Setup_Support service. Table 1 summarizes the requirements and components.
8.2. Lifecycle of collaboration setup and process-view evolution

The generalized lifecycle in Figure 12 depicts conceptually on the one hand the initial steps for setting up a cross-organizational collaboration configuration with the components of the system architecture in Figure 11. On the other hand, a subset of the lifecycle steps allows for a process-view based evolution of a collaboration configuration (see Table 2).

We first explain below the elementary steps of the lifecycle in Figure 12 related to the architecture in Figure 11. Alphabetic letters relate the explanations to the respective lifecycle steps. Support for the conceptual formulation (a) of business processes and their accompanying rules involves the eSourcing_Setup_Support component. The latter comprises functionality for re-using business rules and process patterns for rapid conceptual-layer business process formulation that corresponds to, e.g., the internal job-application process of Figure 2(a).

Mapping details from the conceptual layer business-process to the internal layer (b) pertains to binding the tasks of a conceptual-layer process involving the eSourcing_Setup_Support to the Web-service ports that wrap legacy systems in the Legacy_Management so that an enactment-time orchestration of the latter is
possible. The mapping also involves the Translator component from a conceptual- to an internal level. Projecting from the conceptual layer business-process details to the external layer (c) involves the Translator component that creates different notation-formats to cater for business-process heterogeneity, e.g. from BPEL on the conceptual layer to BPMN on the external layer. Furthermore, the eSourcing_Middleware receives on an external layer the projected processes after their creation with the eSourcing_Setup_Support.

Brokering capability of projected business processes (d) for both the service consumer- and the service provider must be able to place their projected process views into a broker environment of a Trusted_Third_Party component. This functionality is important for collaborations in an anonymous environment. The process views must be searchable for potential business partners. Bidding capability for projected processes views (e) is also part of the Trusted_Third_Party component. The collaborating counter-party evaluates and chooses the subjectively best bid.

Negotiation support for setting up a collaboration configuration with known collaborating parties (f) is relevant after collaborating parties have found each other, they need a Trusted_Third_Party component for starting the contracting negotiations on the external layer of a collaboration configuration. This negotiation involves also the eSourcing_Middleware where the projected process views must be matched into a collaboration consumption.

Verifying perspectives of a collaboration configuration (g) from a control-flow point of view is important to verify a collaboration configuration for correct termination [50]. A verification must ensure that a service provisioning internally adheres to the externally promised collaboration behavior. Simulation of a collaboration configuration (h) addresses that despite verification, errors may still occur during service enactment. Hence, a simulation component for business processes must be available for a-priori enactment simulation.

Distribution of business processes (i) to the external- and internal layer we cater for in the Translator component on the conceptual-layer. The recipients are the eSourcing_Middleware and the Legacy_Management component.
Table 2: Relating life cycle actions to components of Fig. 11.

<table>
<thead>
<tr>
<th>Life cycle action</th>
<th>Trusted Third Party</th>
<th>eSourcing Middleware</th>
<th>eSourcing Setup Support</th>
<th>Legacy Management Translator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conceptual formulation of processes and rules</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping conceptual process to an internal layer</td>
<td>x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Projecting from a conceptual to external layer</td>
<td>x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Brokering of projected processes</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bidding for processes</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negotiation support for setup</td>
<td>x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verifying collaboration</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distribution of processes</td>
<td>x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shielding of processes and legacy systems</td>
<td>x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enactment of a configuration</td>
<td>x x x x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Shielding of business processes and legacy systems on concern-separating layers (j) must ensure the legacy systems that are Web-service wrapped and part of the internal layer, are safeguarded by data-monitoring functionality. The concerned components are the eSourcing Middleware, Translator and the Legacy Management component.

Finally, the enactment of a ready collaboration configuration (k) takes place with distributed rules- and process engines that are on the one hand part of the external layer’s eSourcing Middleware component and on the other hand, the Legacy Management component of the internal layer.

After a completed setup phase, the enactment of a collaboration configuration commences. The actual enactment components must be present on an internal layer for orchestrating legacy systems. Additional enactment components on the external layer need to choreograph the internal components of the respective collaborating parties.

Finally, besides the full collaboration-setup lifecycle in Figure 12 that culminates in the enactment stage, there is a subset-lifecycle embedded denoted
by dashed arrows for supporting evolution of process-view based collaborations. The start is from (k) to (a) for first performing changes to the existing internal business process following the Section 4.2 change operators. This involves the \textit{eSourcing\_Setup\_Support} for modifying the internal process and subsequently the \textit{Translator} for communicating the changes on the one hand to the \textit{Legacy\_Management} component and also on the other hand to the \textit{eSourcing\_Middleware}. The latter checks if the changes must be propagated to the process views and the domain of the counterparty in accordance with the rules stipulated in this paper.

Next, the internal-process projection (c) to the external-layer process view culminates in a verification (g) and also optional simulation (h). The component of concern for the verification is on the one hand the \textit{eSourcing\_Setup\_Support} on the conceptual layer and on the other hand, also the \textit{Trusted\_Third\_Party} comprises an extra component for verification on the external layer. To bridge between the layers, again the \textit{Translator} serves for communicating a projection to the external layer. Additionally, for simulation of the evolved processes, the \textit{eSourcing\_Setup\_Support} comprises a corresponding component.

The collaboration-configuration validation either results in no changes of the latter in accordance with Section 6, or according to Section 7. Alternatively, the validation result requires changes to the process view that must be propagated into the domains of collaborating counterparties. In the latter case, a verification (g) and simulation (h) assures the re-established soundness of the overall business collaboration. Distribution (i) and shielding (j) precede a continued enactment (k) of the changed business collaboration. Again, the distribution involves the \textit{Translator} while shielding concerns the \textit{eSourcing\_Middleware} on the external layer and also the \textit{Legacy\_Management} component of the internal layer. Finally, a dashed, bi-directional arrow in Figure 12 denotes that a faulty exception occurring at a specific lifecycle stage leads to a rollback into a previous lifecycle stage.

As an additional step to show feasibility, we have implemented the \textit{Translator} component of Figure 11, which is pivotal in supporting the process view approach.
The component extends an existing Java-based prototype for constructing and matching BPEL process views [13]. Given an internal BPEL process and its process view, the extended prototype computes how a change to the internal BPEL process translates to a change to the process view. Also, the prototype reports if a change is private, in which case the view change becomes empty, or infeasible. In the latter case, feedback occurs because of the actual inconsistency introduced if both the internal process and the process view are changed.

8.3. Technological feasibility validation

In accordance with the components in Section 8.1, there exist several technical instantiation options for concrete architectures (Table 3). Starting with the assumption for the architecture of Figure 11 that the enactment takes place in a distributed P2P manner, we refer the reader to [51] for a distributed multi-engine workflow system that is scalable and supports strong coupling between process management and business application using Web services. A distributed workflow in [52] targets redundancies in virtual enterprises in terms of carrying out the same operations, job, or functions. Furthermore, a distributed rules engine named ViDRE [53] separates the implementation logic with exposing rules as Web services for accessibility across various rule engines.

For using legacy systems to be part of a larger service-based application system, loosely coupled services facilitate the establishment of highly dynamic business relations using service-oriented computing [66]. A pattern repository in [42, 49] describes the storage of conceptually formulated patterns organized in a taxonomy belonging to different perspectives such as control-flow, data-flow, exception management. The knowledge base also stores implementations of respective patterns in industry standards such as BPEL [23], BPMN [44] for service specification, and so on. For housing business processes as a service, we propose a (BPaaS-HUB) [57] on which web services are search- and evaluable for matching with service requests in a master-client collaboration context, or as the discussion in the subsequent section shows, for matching service types with service offers. For brokering services, BROSEMWEB [58] is an example from
<table>
<thead>
<tr>
<th>Information technology</th>
<th>Trusted Third-Party</th>
<th>eSourcing Middleware</th>
<th>eSourcing Setup</th>
<th>Support</th>
<th>Legacy Management</th>
<th>Translator</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPEL [23]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPMN [44]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pattern repository [42, 49]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ViDRE [53]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hadoop [54]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hive [55]</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impala [56]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPaaS-HUB [57]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BROSEMWEB [58]</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negotiation Broker [59]</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Game Theory [60]</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>ProM [61]</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>YAWL [62]</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signavio [63]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed WF engine [51, 52]</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drools [64]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mandarax [65]</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Service-oriented computing [66]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aristaflow [67]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Websphere [68]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>BizTalk [69]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Event Translator [70]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Xalan [71]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Nikse [72]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

Table 3: Relating information technologies to components of Fig. 11.
the domain of e-Procurement. We assume that bidding and negotiation follows
game-theoretical [60] principles while the Negotiation Broker [59] example for
negotiation in a service-oriented context. Verification algorithms are ever more
integrated in tools such as ProM [61], YAWL [62], Signavio [63], and so on.

Examples for internal workflow enactment exist from research and industry.
A candidate example from research for a local WFMS is YAWL [62] that uses
a Petri-net based business-process modeling notation as a foundation. With
such semantic clarity, it is possible to verify processes for runtime soundness [73]
in advance before enactment, i.e., one can check for control-flow issues such as
deadlocks. The industrial AristaFlow system [67] is another example originating
from research. With the AristaFlow system it is possible to verify control
flow before enactment. Additionally, the formal notation allows to change the
control flow of business processes at runtime if the enactment context requires so.
Examples for local workflow management systems in the Legacy_Management
component from industry are WebSphere [68] and BizTalk [69]. The former
WFMS enacts business processes specified in BPEL [23], which is an industry
standard for composing Web-service orchestration. BizTalk enacts business
processes formulated in XLANG [74], which is BPEL-compatible. For local rules
enactment, again the earlier mentioned ViDRE [53] is a candidate from research
while there also exist several Java-based open-source projects such as Drools [64]
or Mandarax [65].

For realizing the translations of data-, events-, rules-, and process transfers,
several technical realizations exist assuming that XML use dominates most trans-
fers. First, for the translation of data, several options exist such as Hadoop [54]
that is an open-source project for the processing of large and distributed data
sets. The Hadoop Distributed File System provides high-throughput access
to application-systems data and Hadoop MapReduce enables reliable parallel
processing of large data sets. Many additional open-source projects on top of
Hadoop enhance that data-translation power, e.g., Hive [55], a data warehouse
system for easy data summaries, ad-hoc queries in a SQL-like language called
HiveQL to analyze large datasets. Also Impala [56] is an open source query
engine for massively parallel data processing for Hadoop. Next, the open-source initiative Event Translator [70] comprises four features, namely first, an automated discovery of network services, secondly, receive events from many different network protocols and also generate events combined with notification management, thirdly, monitoring of service-level agreement assurance with service monitors and finally, performance measures for thresholds or value changes. Assuming that rules- and process specifications use XML, transforming to other rules- and process-notations is possible with versions of extensible stylesheet language transformations (XSLT) such as Xalan [71]. Finally, XML content in rules and processes can be translated with an open-source XML content translator [72] called Nikse.

8.4. Feasibility-validation outcome

To show feasibility of our evolving process-view approach, we present a supporting conceptual system architecture. One of the key components in this architecture, the Translator, implements the formal definitions of this paper presented in Section 5, 6, and 5. Next, we also discuss relevant technologies that are instrumental to realize concrete application-implementations according to the conceptual system architecture.

9. Discussion

The developed approach supports providers in deciding if and how changes to a private, internal process propagate to a corresponding public process view such that the process view and the internal process remain consistent. The approach allows clients to monitor a public process view such that they can safely track changes made to a private, internal process.

These results apply to the evolution of smart contracts [75], a recent significant development for implementing process views in B2B collaborations. Smart contracts have been developed to enable decentralized autonomous organizations (DAO)[76] to engage in the formation of electronic communities. A smart contract is a computerized transaction protocol [77] to execute contract terms
that are machine readable. For achieving non-repudiation and fact-tracking of a consensual smart-contract agreement, blockchain technology [78, 79] is suitable. A blockchain is a distributed database for independently verifying the chain of ownership of artifacts [80, 79] in hash values that result from cryptographic digests. As a further means to realize electronic communities of DAOs with evolving long-lasting collaborations, service-oriented cloud computing (SOCC) [81] enables companies an acceleration of seamless, ad-hoc integration and coordination of information- and business-process flows [13] to orchestrate and choreograph [15] heterogeneous legacy-system infrastructures.

Figure 13: P2P-collaboration using the eSourcing framework [82].

Pertaining to evolving DAO-collaborations, Figure 13(a) conceptually depicts a configuration. The blueprint for an electronic-community formation is a so-called business-network model (BNM) [83]. The latter captures choreographies that are relevant for a business scenario and it contains legally valid template contracts that are service types with affiliated organizational roles. The BNMs are available in a collaboration hub that houses business processes as a service (BPaaS-HUB) [57] in the form of process views [13]. The latter enable a fast and semi-automatic discovery of collaboration parties for learning about their identity, services, and reputation.
On the external layer of Figure 13(a), service offers identically match with service types contained in the BNM with the contractual sphere of collaborating parties. Additionally, a collaborating partner must match into the specific partner roles associated with a respective service type. We refer the reader to [13] for details about the tree-based process-view matching to establish a DAO-configuration into a contract-based collaborations. Note, while Figure 13(a) uses Petri net [50] notation, mapping the depicted collaboration scenario to the tree-formalization as per this paper is straightforward.

The top-level structure shows a smart-contract language termed eSourcing Markup Language (eSML) [8] in Figure 13(b). The bold typed definitions in the eSML-structure are extensions and modifications that are not part of the Electronic Contracting Markup Language (ECML) [84] foundation. We refer to [8] for more information about the smart-contract ontology.

Long-lasting automated DAO-collaboration must be able to manage exceptions [85] that result in changes on a process level. Exceptions and adjustments to contextual changes that do not result in a collapse of an overall collaboration configuration must have a set of change rules available that achieve a sound transition. A smart contract can be seen as a particular kind of process view. The change rules of this paper therefore provide a basis for managing evolution of smart contracts based on changes in underlying internal processes. The results of this paper therefore contribute towards achieving flexibility in DAO-collaborations.

10. Conclusion

The main contribution of this paper is the definition of a formal approach to propagate consistency-preserving changes from an internal process to its process view. This allows consumers to safely track in process views the effect of internal changes made to the internal processes underlying the process views.

The approach is independent from and actually complementary to existing approaches for constructing process views. The approach allows process views to evolve in order to remain consistent with the internal processes they are based
on. If the process view cannot evolve as a change creates an inconsistency, the change should be blocked and modified.

We have shown feasibility of the approach in several steps. We have analyzed how an existing system architecture for process-based business collaboration supports the approach, and identified technologies that are relevant for building actual systems. The key component of the system architecture, which translates internal processes plus change requests to changes on process views, has been implemented.

As future work, we plan to automatically generate different repair options for fail cases of change propagations. An extension of the set of change operators is also planned. Additionally, it is possible to generalize the currently considered process language by incorporating more operators. We also plan to extend the approach to unstructured processes in future work. Finally, we plan to study evolution of process views for data-centric processes, building upon earlier research [27] that studies outsourcing of data-centric process models.

Acknowledgement

The work reported in this paper is partly supported by the research project SF0140013s10 ”Model-based Creation and Management of Evolutionary Information Systems” by the Estonian Ministry of Education and Research.


[75] N. Szabo, Formalizing and securing relationships on public networks, First Monday 2 (9).

[76] V. Butterin, A next-generation smart contract and decentralized application platform (2014).


47
Appendix A. Proofs

This appendix contains the proofs of Theorem 1 and 2.

Proof (Theorem 1). \( \Rightarrow \): Suppose \( V \) is a valid view for \( P' \) using abstraction function \( \alpha' \) derived from \( \alpha \). This implies that \( \alpha \) and \( \alpha' \) have the same range. There are three cases.

- If \( \delta \) is an insert operation in the context of \( n \), then \( x \) is not a public activity, so \( x \) is private.

- If \( \delta \) insert additional control flow in the context of \( n \), then since \( \text{compound}_n \) not in range \( \alpha' \), we have \( \alpha(n) \neq n \).

- If \( \delta \) is a delete operation for node \( n \) and \( \alpha(n) \neq \bot \), then since \( V \) is a view for \( P' \) there is another node \( n' \) such that \( \alpha(n) = \alpha(n') \).

\( \Leftarrow \): We enumerate the cases for \( \delta \). If \( \delta \) inserts a private activity \( x \) in the context of \( n \), there are two cases.

- If \( \alpha(n) \neq n \), there are two subcases. (i) If \( \alpha(n) = \bot \) then \( \alpha(x) = \bot \). (ii) If \( \alpha(n) = y \) where \( y \in N_V \) then \( \alpha(x) = y \). Therefore \( V \) is a valid view for \( P' \).
• If $\alpha(n) = n$ then $\alpha(x) = \bot$. Therefore $V$ is a valid view for $P'$.

If $\delta$ inserts additional control flow in the context of $n$ and $\alpha(n) \neq n$, then $\alpha(\text{compound}_n) = \alpha(n)$ and $\alpha(\text{compound}_n) \neq \text{compound}_n$. Therefore $V$ is a valid view for $P'$.

If $\delta$ deletes node $n$ where $\alpha(n) = y$ for $y \in N_V$, then there is another node $n'$ such that $\alpha(n') = y$. Therefore $y$ remains in $V$ and $V$ is valid view for $P'$.

**Proof (Theorem 2).** $\Rightarrow$: Since $\delta$ is valid, $V'$ is a view on $P'$ using abstraction function $\alpha'$. If $\delta$ deletes node $n$ or inserts additional control flow in the context of $n$, the claim is true.

Let therefore $\delta$ insert activity $x$ in the context of $n$ where $\alpha(n) \neq \bot$. Since $\delta$ is a public change, by Theorem 1 $x \notin F_{\text{private}}$ and by Definition 8 $\alpha'(x) = x$. Since $x$ not in $P$ and $\alpha(n) = \alpha'(n)$, this implies that $\alpha'(n) \neq \alpha(x)$. Since $\alpha(n) \neq \bot$, either $\alpha(n) = n$ or $\alpha(n) \neq n$. For $\alpha(n) = n$ the claim is true.

For $\alpha(n) \neq n$, we reason as follows. First, $\alpha(n) \otimes_{V'} x$, where $\otimes_{V'} \in \{<, \triangledown, \&\}$. Since $V'$ is a view for $P'$ using $\alpha'$, by Def. 7 all nodes in the aggregate for $n$ have the behavioral relation $\otimes_{V'}$ with $x$ in $P$: for every $y \in N_P$ such that $\alpha(y) = \alpha(n)$, $y \otimes_{V'} x$. Behavioral relation $\otimes_{V'}$ depends on the insertType (before, after, parallel, choice) for $\delta$.

We analyze each of these four cases:

If insertType=before, then $\otimes_{P'} = >$. For all nodes $y \neq n$ in the aggregate for $n$, $y >_{P'} x$ if and only if $n <_P y$.

If insertType=after, then $\otimes_{P'} = <$. For all nodes $y \neq n$ in the aggregate for $n$, $y <_{P'} x$ if and only if $n >_P y$.

If insertType=parallel, then $\otimes_{P'} = \&$. For all nodes $y \neq n$ in the aggregate for $n$, $y \&_{P'} x$ if and only if $n \&_P y$.

If insertType=choice, then $\otimes_{P'} = \triangledown$. For all nodes $y \neq n$ in the aggregate for $n$, $y \triangledown_{P'} x$ if and only if $n \triangledown_P y$.

$\Leftarrow$: Operation $\delta$ is by definition either an insert or a delete operation.

$\delta$ inserts fresh activity $x$ in the context of node $n$ and $\delta'$ inserts $x$ before $\alpha(n)$. By definition of the insert operation, compared to $P$, $P'$ contains additional
behavioral relations referencing $x$. And compared to $V$, $V'$ contains additional behavioral relations referencing $x$.

To prove that $V'$ is a view on $P'$ using $\alpha'$, it suffices to prove that these additional behavioral relations between $x$ and the other nodes in $P'$ are preserved in the view $V'$ (the other behavioral relations are valid since $V$ is a view on $P$ using $\alpha$). Consider an arbitrary node $y$ from $N_P$ with $x \otimes_{P'} y$. By definition, $\alpha'(x) \neq \alpha'(y)$. Therefore, we have to prove that $\alpha'(x) \otimes_{V'} \alpha'(y)$ such that $\otimes_{P'} = \otimes_{V'}$.

We prove the claim for the before insertType. The reasoning for the other insertTypes is similar and therefore omitted. For $P'$, by definition of $\delta$, $x <_{P'} n$. For $V'$ by definition of $\delta'$, $\alpha'(x) <_V \alpha'(n)$. Consider an arbitrary node $y$ from $N_P$, $y \neq x$. There are two cases

- If $\alpha(n) = \alpha(y)$, then $n <_{P} y$ by assumption. Since $x <_{P'} n$, we have $x <_{P'} y$. Since $\alpha(n) = \alpha(y)$, for $V'$ we have $\alpha(x) <_{V'} \alpha(y)$.
- If $\alpha(n) \neq \alpha(y)$, then $x \otimes_{P'} y$ where $n \otimes_{P} y$ and $\otimes_{P} = \otimes_{P'}$. For $V'$, $\alpha'(x) <_{V'} \alpha'(n)$. For $V$, we have $\alpha(n) \otimes_{V} \alpha(y)$ and $\otimes_{V} = \otimes_{P}$. Therefore for $V'$, $\alpha'(n) \otimes_{V} \alpha'(y)$. By definition of the insert operation, $\alpha'(x) \otimes_{V'} \alpha'(y)$ and $\otimes_{V} = \otimes_{V'}$.

$\delta$ inserts additional control flow in the context of node $n$ and $\alpha(n) = n$. By definition of the operation, extra node $\text{compound}_n$ has been inserted in both $P'$ and $V'$. Consider an arbitrary node $y$ from $N_P$ with $\text{compound}_n \otimes_{P'} y$ and $\text{compound}_n \otimes_{V'} y$. We have to prove that $\text{compound}_n \otimes_{V'} y$ such that $\otimes_{P'} = \otimes_{V'}$.

By definition of $\delta$ applied to $P$, $n \otimes_{P} y$ and $\otimes_{P} = \otimes_{P'}$, so $n$ has the same behavioral relation to $y$ as $\text{compound}_n$ to $y$ in $P'$. Since $V$ is a view of $P$, $\alpha(n) \otimes_{V} \alpha(y)$ and $\otimes_{V} = \otimes_{P}$. By definition of $\delta$ applied to $V$, $\alpha(n) \otimes_{V} \alpha(y)$ and $\otimes_{V} = \otimes_{V'}$. Therefore $\otimes_{V'} = \otimes_{P'}$.

$\delta$ deletes node $n$ and $\delta'$ deletes $\alpha(n)$. Again, we show that $V'$ resulting from applying $\delta'$ is a view on $P'$ resulting from $\delta$.

First, observe that there is not another node $n'$ such that $\alpha(n) = \alpha(n')$. By definition of the delete operation, compared to $P$, in $P'$ all behavioral relations
incident to $n$ are removed. And compared to $V$, in $V'$ all behavioral relations incident to $\alpha(n)$ are removed. By Def. 8, tuple $(n, \alpha(n))$ is removed from $\alpha$ to obtain $\alpha'$, and there is no other tuple referencing $\alpha(n)$. Therefore $V'$ is a view on $P$ using $\alpha'$. 