Service Outsourcing with Process Views

Rik Eshuis, Member, IEEE, Alex Norta, Oliver Kopp, and Esa Pitkänen

Abstract—Service outsourcing is a business paradigm in which an organization has a part of its business process performed by a service provider. Process views are pivotal to support this way of working. A process view shields secret or irrelevant details from a private business process, thus allowing an organization to reveal only public, relevant parts of its private business process to partner organizations. This paper introduces a specification framework to support service outsourcing using process views. To enable the construction of process views at various levels of detail, the framework defines several projection relations between process views and the underlying internal processes. To allow consumers and providers of process views to establish an outsourcing relation, the framework defines several matching relations between the respective views that are comprehensive and flexible for service outsourcing. A proof-of-concept prototype tool implements the framework, which is applied in a case study.

Index Terms—Inter-organizational processes, process trees, matching, B2B, process visibility

1 INTRODUCTION

The WAY companies collaborate with each other has changed significantly in recent years. With the emergence of service-oriented computing (SOC) [1], companies embrace the vision of using web services for engaging dynamically and flexibly in business-to-business (B2B) collaboration. Web services [2] are an important vehicle for enabling organizations to cooperate with each other. Cooperation based on web services results in inter-organizational business processes that span the boundaries of multiple organizations [3], [4].

Service outsourcing is a business paradigm in which a service-provider organization performs or coordinates parts of a business process of a service-consumer organization; these parts were previously performed or coordinated by the service-consumer organization itself. The main motivation for service outsourcing is that a specialized provider can perform the outsourced process parts more efficiently and effectively than the consumer. For example, in retail, the transportation process can be outsourced to a third-party logistics provider that coordinates the actual transportation. However, the consumer may require status information of the outsourced services to coordinate its own in-house business processes. Hence providers need to expose status details of outsourced services in public process views [5], [6], which abstract from private details of the provider processes performed on behalf of the consumer. Service consumers use process views to monitor and control the progress of service executions [7] by the provider.

While the notion of process view is generally recognized as pivotal to support service outsourcing [6], [8], to the best of our knowledge there has been no research on how a consumer organization that wishes to outsource a business process, finds and connects to a provider that can perform the process. Some approaches tackle the construction of process views [5], [9] from internal processes, but this is just one aspect of the problem and is not specific to service outsourcing [10]. In particular, each approach supports only one way to construct a process view from an internal process, ruling out the construction of other process views that are viable. Other approaches consider the problem of matching process descriptions of services [11], [12], [13], but do not relate this to the problem of constructing process views.

To fill this detected gap, this paper proposes a novel specification framework for service outsourcing, depicted in Fig. 1. The framework supports service outsourcing by enabling the flexible construction and matching of public, external process views that are fueled by private, internal processes. The framework hosts a comprehensive set of projection relations that enable the construction of different process views for the same internal process. Choosing between different projection relations allows collaborating parties to expose in flexible ways only those details of their internal processes they deem necessary. Next, the framework defines efficient matching relations that allow consumers and providers to quickly establish suitable collaborations. The framework also considers the interplay of projection and matching relations. To connect to industrial practice, we use BPEL [14] as base modeling language.

The rest of this paper is organized as follows. Section 2 introduces the problem of service outsourcing by presenting a running example that is revisited in the remaining sections. Section 3 formalizes BPEL specifications as process trees. Section 4 first defines projection rules on process trees for constructing a process view from an internal process. Based on the projection rules, we identify several useful projection relations between a private
process and a process view. Section 5 defines several matching relations between process views based on which consumers and providers find each other and establish an outsourcing relationship. Section 6 relates the projection and matching definitions by defining feasible combinations for an outsourcing collaboration. Section 7 presents a prototype implementation of the framework. Section 8 applies the projection and matching definitions in an industrial case study. Section 9 presents related work. Finally, Section 10 concludes this paper.

This paper revises and expands two conference papers [12], [15], which tackle projection [15] and matching [12] of process views in isolation. This paper analyzes their interplay and presents in addition a tool implementation.

2 Running Example

We introduce by means of an example adapted from Vonk and Grefen [16] the notion of service outsourcing using process views. As explained in the introduction, we use BPEL as specification language. However, BPEL does not have a standard graphical notation. We therefore use BPMN [17] to visualize BPEL processes [18].

Fig. 2 presents a telecom-sales process view that starts with picking a GSM and next, configuring it according to customer demand. Next, a subprocess (compound node) is performed to deliver the GSM-package, containing the following activities. First, a route is scheduled. Next, the GSM-package is delivered, which can be done in three different ways. Finally, the GSM-package is handed over to the customer who signs the receipt.

Fig. 2 uses specific notation for outsourcing. First, the same activity can occur in different models in the framework of Fig. 1. For instance, an activity can occur in a process-view request of a consumer and a process-view offer of a provider. We distinguish these activity occurrences by prefixing activity labels with a namespace indicator, which is either $c$: (for a consumer) or $p$: (for providers).

Next, we specify which party takes the initiative to start an activity. If the service provider initiates an activity, the activity is called observable [19], [20] (labeled $Ob$ in Fig. 2). If the service consumer initiates an activity, the activity is called invocable [19], [20] (labeled $In$). All observable and invocable activities are performed by the service provider, and monitored by the service consumer. Observable activities are the default for outsourcing: a service consumer can track the status of the execution at the provider side but cannot control it. By using invocable activities, a service consumer can exert control over the execution at the provider side. Distinguishing between observable and invocable activities thus allows fine-grained service outsourcing.

We annotate a choice construct to specify whether that choice is made externally or internally. The annotation $IX$ for $IXOR$ in Fig. 2 means that the service consumer expects the provider to decide for the branch of an exclusive-choice split during enactment. Conversely, an $EX$-annotation for $EXOR$ means the service consumer determines which branch of an OR-choice to take. Fig. 3 depicts an EXOR-construct with an event-based gateway.

The BPMN models in Figs. 3 and 4 show two private processes owned by service providers $X$ and $Y$ for matching to the consumer process in Fig. 2. Both processes share some activities and ordering constraints with the consumer process, for example $get\ \text{GSM}$. Still, both
provider processes contain unobservable and uninvocable activities that are not part of any process view.

The private process of provider \( X \) in Fig. 3 differs from the consumer view in Fig. 2 since it contains an event-based gateway where the service consumer decides what branch to take during enactment. The two message-flows in Fig. 3 from the service consumer’s in-house domain depict this decision. The provider process contains compared to the consumer view the additional activities \( px: \text{wrap envelope} \), \( px: \text{determine transportation} \) and \( px: \text{determine route} \). Since these activities are neither observable nor invocable, these activities have to become opaque for the service consumer during enactment. Furthermore, the consumer activity labeled \( c: \text{schedule route} \) does not exist in the private process of provider \( X \) in Fig. 3.

The private process of provider \( Y \) in Fig. 4 contains extra activities \( py: \text{check GSM} \) and \( py: \text{deliver priority} \) compared to Fig. 2. The latter activity leads to an additional third branch after the exclusive XOR-split. Again, both activities carry no textual annotation, i.e., they are opaque to the service consumer during enactment time.

To determine whether these two private provider processes can actually realize the requested consumer process view, both private processes need to be projected into public process views and these views have to be matched with the consumer view. Sections 4 and 5 present projection and matching definitions.

### 3 Process Trees

As explained in the introduction, we base the outsourcing framework on BPEL [14]. We formalize the syntax of BPEL models as process trees (see Definition 1). Leaves specify execution of basic activities and internal nodes specify ordering constraints on their child nodes. A \( \text{SEQ} \)-node specifies sequential execution of children nodes. An \( \text{AND} \)-node specifies concurrent execution, corresponding to the BPEL flow construct. An \( \text{XOR} \)-node specifies the execution of one of its child nodes. There exist two kinds of XOR: an \( \text{XOR} \) depicted with an \( \text{In} \)-annotation in Fig. 3, where the system chooses a branch to enact and an \( \text{XOR} \) depicted with an \( \text{EX} \)-annotation in Fig. 3, where the environment of the system chooses a branch to enact. An external choice corresponds to a BPEL if-then-else or switch construct while an internal choice corresponds to a pick construct. Finally, a \( \text{LOOP} \)-node specifies structured repeat-until loops. To simplify the exposition we do not consider synchronization links here, but they can be included too [12].

We present a formal definition for process trees [5], [12].

**Definition 1.** A process tree \( P \) is a tuple \( (N, A, C, \text{parent}, \text{successor}) \) where:
- \( N \) is the set of nodes, partitioned into sets \( A \) and \( C \);
- \( A \) is the set of activities or tasks. As subsets of \( A \), we consider disjoint sets \( In, Ob \), so \( In \cap Ob = \emptyset \), where...
In is the set of invocable activities, and Ob is the set of observable activities;

- C is the set of control nodes, partitioned into sets $C_{SEQ}$, $C_{AND}$, $C_{EXOR}$, $C_{IXOR}$, and $C_{LOOP}$. Each node in C has subnodes, which are either control nodes or activities (see function parent below). If a node $n$ is in $C$, we call $n$ a node of type $t$ or a $t$-node for short. A SEQ-node specifies a sequential behavior, an AND-node parallel behavior, an IXOR- and an EXOR-node exclusive behavior, where the choice is made internally for IXOR and externally for EXOR-nodes, and a LOOP-node specifies iterative behavior;

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

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

We require that the start activities under an EXOR-node are invocable: the environment chooses a particular branch of the EXOR-node by invoking the start activity of that branch. We follow BPMN [17] by modeling invocations as enabling signals that are thrown by the consumer and caught by the provider via messages. For instance, the branches of the EXOR-node in Fig. 3 have start activities wrap envelope and configure GSM, both of which can only be invoked by the consumer.

For a node $n$, let $children(n) = \{ x | parent(x) = n \}$. By $children^*$ and $children^{+}$, we denote the reflexive-transitive closure and the irreflexive-transitive closure of $children$, respectively. If $n \in children^*(n')$, we say that $n$ is a descendant of $n'$ and that $n'$ is an ancestor of $n$. In particular, each node $n$ is ancestor and descendant of itself, so $n \in children^*(n)$, but $n \notin children^{+}(n)$. We require that the $parent$ function induces a tree, so each node has one parent, except one unique node $r$, which has no parent. Since $r$ is unique, $r$ is ancestor of every node in $N$. 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.

Next, we require that each SEQ-, AND-, EXOR-, or IXOR-node has more than one child, and that each LOOP-node has exactly one child. If while constructing a process view (projection), an internal node $i$ that is not a LOOP-node gets one child only, we remove $i$ and its child becomes a child of the parent of $i$.

To explain the relation with the BPMN models in Section 2, we show in Fig. 5 the abstract tree representations of the BPMN models presented in Section 2. Nodes with bold lines are invocable activities, e.g., service consumer initiates the invocation of activities $gG$ (get GSM) and $cG$ (configure GSM) in the service provider domain. All the other nodes in Fig. 5a are observable but not invocable by the service consumer. As before, the namespace-prefix $c$: indicates that activities reside in the service consumer domain. Nodes with dashed lines in Figs. 5b and 5c are neither invocable nor observable.

4 Projection Relations

When a party projects its internal business process to a process view at the external level, it reveals what a successful collaboration setup requires and conceals the remainder to protect business secrets that offer a competitive advantage. Still, the process view must be consistent with the underlying internal process. We define three projection rules for constructing a process view from an internal business process. The rules ensure consistency.
while safeguarding private details from the internal process and are useful for both provider and consumer side processes:

- Hiding: a set of nodes executed at the internal level are not shown in the process view at the external level.
- Omitting: a set of nodes that do not need to be executed at the internal level are not shown in the process view at the external level, e.g., activities of an EXOR-branch.
- Aggregation: a set of nodes executed at the internal level is shown as a single node in the process view at the external level.

First, we define and explain these three basic projection rules in Section 4.1. Next, Section 4.2 uses the rules to define different projection relations between a process view and an underlying internal process.

### 4.1 Definition of Projection Rules

Each rule takes as input a process tree $P$ and a node $n$, and returns an abstracted process tree $P'$ in which a set input nodes induced by $n$ are hidden, omitted, or aggregated into a new activity. For the aggregation rule, the new aggregate activity is an additional input. To construct a process view from an internal process, one or more of the rules can be applied consecutively.

#### 4.1.1 Rule 1 (Hiding)

Node $n$ is hidden from process tree $P$ if node $n$ and its descendants are not contained in the process view but still part of the internal process and executable at the internal level.

However, a hidden node cannot be an invocable activity, since a consumer cannot invoke an invisible activity. Moreover, the parent of $n$ must be a $SEQ$, $AND$, or $LOOP$-node. By constraint, each branch of an EXOR-node contains as initial node an invocable activity, and invocable activities cannot be hidden. If a branch of an IXOR-node would be hidden and is chosen during execution, the consumer observes that an internal choice has to be made but observes that none of the visible branches are chosen, while the provider does continue with the sequel of the process, counter the expectation of the consumer. Therefore, node $n$ cannot be hidden if the parent of $n$ is an EXOR-, or IXOR-node.

#### Definition 2 (Hiding)

Let $P$ be a process tree $(N, A, C, parent, succ)$. If $n \in N$ is a node to be hidden from process tree $P$, then we require that

- none of the descendants of $n$ are invocable, so $children^*(n) \cap In = \emptyset$; and
- $n$ is not a complete branch of an EXOR or IXOR-node, so $parent(n) \notin C_{EXOR} \cup C_{IXOR}$.

The resulting process tree $P'$ is defined as $(N', A', C', parent', succ')$ where:

- $N' = N \setminus children^*(n)$
- $A' = A \cap N'$
- $C' = C \cap N'$
- $parent' = parent \cap (N' \times N')$
- $succ' = (\text{succ} \cap (N' \times N')) \cup \{(x, y) | succ(x) = n \land succ(n) = y\}$

The definition for $succ'$ states that if node $n$ is child of a $SEQ$-node with predecessor $x$ and successor $y$, then the successor of $x$ becomes $y$ instead of $n$.

#### Example 1

In Fig. 5c, $py:chG$ can be hidden, since it satisfies the precondition for hiding. However, node $py:dp$ must not be hidden, since the parent is an $IXOR$-node. This means the provider decides internally whether to enact $py:dp$ or one of its sibling nodes: $py:dr$ or $py:de$. If $py:dp$ is not visible in the process view but executed at the internal level, none of its sibling nodes is chosen in the process view, but still the process view continues with $py:hp$, so then the consumer observes a gap in the execution of the process view.

#### 4.1.2 Rule 2 (Omitting)

Node $n$ is omitted from process tree $P$ if node $n$ and its successors in the same branch are not contained in the process view and not executed at the internal level, though the internal process contains the omitted nodes.

Omitting is only allowed if $n$ is invocable. Next, $n$ must be part of an EXOR-branch, that is, either the parent of $n$ or its grandparent is an EXOR-node. Node $n$ must be the first activity in the branch, as the consumer chooses which branch of the EXOR-node it executes, but the provider limits the possible options. If $n$ is not the first activity, the internal process deadlocks when executing the branch and arriving at $n$.

Note that if $n$ is part of an IXOR-branch, the provider decides to execute $n$ and therefore, omitting is not allowed. For example, if from Fig. 5b node $p:dR$ were omitted, then the provider-process view does not contain this node, but the provider internal process does contain $p:dR$ and therefore could decide to execute it, contradicting the definition of omitting. If $n$ has an EXOR-parent, the consumer, not the provider process, decides which branch to choose, without being able to choose omitted nodes.

#### Definition 3 (Omitting)

Let $P$ be a process tree $(N, A, C, parent, fssucc)$. If $n \in N$ is a node to be omitted from process tree $P$, then we require that

- $n$ is invocable, so $n \in In$;
- $n$ has an EXOR-parent or grandparent: either $type(parent(n)) = EXOR$ or $type(parent(parent(n))) = EXOR$;
- $n$ is the first activity in its branch, so there does not exist a node $n' \in N$ such that $succ(n') = n$.

The resulting process tree $P'$ is defined as $(N', A', C', parent', succ')$ where:

- $N' = N \setminus \{(n' | \exists n'' \in succ^*(n) : n' \in children^*(n'')\} \cup \{(n' | parent(n) = n' \land type(n') = \text{SEQ})\}$
- $A' = A \cap N'$
- $C' = C \cap N'$
- $parent' = parent \cap (N' \times N')$
- $succ' = succ \cap (N' \times N')$
Example 2. Omitting node px:we in Fig. 5a results in a process tree in which node EXOR has one child only. As explained in Section 3, internal node EXOR is therefore eliminated and px:wp becomes child of SEQ1.

4.1.3 Rule 3 (Aggregation)
A set of neighboring nodes $X \subseteq N$ is aggregated from process tree $P$, if a single new activity $n_{agg}$ in the process view condenses all nodes in $X$ and their descendants. The aggregated nodes are still part of the internal process and executable at the internal level. Elsewhere [5], we explain how to aggregate a set of non-neighboring nodes.

However, we only allow aggregation of observable nodes. Aggregating an invocable node means the consumer is not able to invoke the node at the external level and therefore, the execution deadlocks. This implies that external choices cannot be part of an aggregate.

Definition 4 (Aggregation). Let $P$ be a process tree $(N, A, C, \text{parent}, \text{succ})$. If $X \subseteq N$ is a set of nodes to be aggregated from process tree $P$, then we require that

- none of the descendants of any node $n \in X$ are invocable, so $\text{children}(n) \cap \text{In} = \emptyset$.
- none of the descendants of any node $n \in X$ are external choices, so $\text{children}(n) \cap C_{\text{EXOR}} = \emptyset$.

If $agg$ is the new aggregate activity, the resulting process tree $P'$ is defined as $(N', A', C', \text{parent}', \text{succ}')$ where:

- $N' = (N \setminus \bigcup_{n \in X} \text{children}(n)) \cup \{agg\}$
- $A' = (A \setminus (N' \cap A)) \cup \{agg\}$, where $Obl = (Ob \cap A') \cup \{agg\}$.
- $C' = C \setminus (N' \cap C)$
- $\text{parent}' = \{\text{parent} \cap (N' \times N')\} \cup \{(agg, l) | \text{parent}(n) = l\}$
- $\text{succ}' = (\text{succ} \cap (N' \times N')) \cup \{(n', agg), (agg, n') | \text{succ}(n') = n \land \text{succ}(n) = n'\}$

Example 3. Applying the aggregation rule to nodes px:dt and px:dR0 in Fig. 5b with new activity sR (schedule route) results in a process tree in which the subtree containing the last four activities is isomorphic to the corresponding subtree in Fig. 5a.

Having explained and illustrated the projection rules, we revisit the example processes in Fig. 5, where Fig. 5a contains the consumer process view and Figs. 5b and 5c internal processes of two providers.

Example 4. Omitting px:we and aggregating SEQ3 in Fig. 5b yields a process view that is identical (modulo prefixes) to the process view in Fig. 5a. So provider X and the consumer can collaborate with each other.

However, the internal process of provider y contains activity deliver priority (dp) which is part of an Ixor-branch. Hiding, omitting, and aggregation are not applicable to this node. Consequently, it is not possible to derive a process view from Fig. 5c that satisfies the process view in Fig. 5a.

Finally, note that the projection rules are also useful for deriving an internal process from a process view. By inserting activities and control-flow to the process view, we derive an internal process. The implicitly applied projection rules relate the resulting internal process to the process view. For example, Fig. 5b can be derived from Fig. 5a since omitting and aggregating the extensions of the process model in (b) results in the process view in (a).

Next, we analyze the role played by the projections in the setup of an inter-organizational collaboration configuration.

4.2 Projection Relations between Process View and Internal Process
In the previous section, we define general rules for transforming an internal process to a process view and vice versa. In this section, we look at possible projection relations between a process view and an internal process.

Each projection relation uses a combination of the projection rules and Table 1 lists the extreme projection relations we consider. Black box, glass box, and open box stem from a Web-service outsourcing example by Grefen et al. [7] while Norta [21] identifies gray-box and white-box projection in a Petri-net study. All other possible projection relations are hybrid forms of these extreme relations. In the remainder of this section, we explain the projection relations listed in Table 1.

**Black-box projection** occurs if the process view contains only a single observable activity. A process tree that results from black-box projection contains a single node that aggregates or hides a set of nodes from the internal process. It is not possible to hide or aggregate invocable activities, so the internal process does not contain any invocable activities. Moreover, since the external process tree cannot contain any EXOR-node with invocable activities as descendants, omitting is no abstraction option. The service consumer has no options to monitor and control the progress of the internal process.

**Glass-box projection** occurs if the process view only contains observable activities; the consumer cannot invoke any of the provider-activities. Hiding and aggregating activities from an internal-level process results in a glass-box view. Since the process view does not contain any invocable activities, omitting is not used. Black-box projection is a special case of glass-box projection. The service consumer can monitor the progress of the internal process through the process view, but has no control options.

**Gray-box projection** occurs if the process view comes into existence through hiding and omitting from the internal process. The process view optionally contains both observable and invocable activities while there is no aggregation. The service consumer can monitor the internal process using observable activities in the process view. The service consumer can control the progress of the internal process at the provider side through invocable activities in the process view.

**Open-box projection** occurs by establishing the process view through hiding, omitting, and aggregation from the internal process. The process view contains both observable and invocable activities, allowing a service consumer to monitor and control the progress of the internal process. Thus,
the service consumer can partially influence the progress of the internal process in the provider-domain.

Finally, with a white-box projection the process view is identical to the internal process. There is no application of abstraction rules and the service consumer has a direct view on the internal process of the provider. Thus, a white-box is a special case of an open-box projection. The service consumer has now full control over the internal process at the provider.

**Example 5.** The consumer-process view in Fig. 5a is white-box related to the internal-level consumer process if that process is identical to Fig. 5a. Next, Fig. 5a relates as an open-box projection to Fig. 5b by applying both omitting and aggregation rules. Both provider processes in Figs. 5b and 5c can not relate to Fig. 5a using black-box or glass-box projections, since both processes contain an invocable activity.

In summary, the projection relations support a collaboration scenario in which collaborating parties optionally expose in flexible ways only those details of their internal processes as they deem necessary. However, as we show in Section 6, not every combination of projection relations for consumer and provider-side is suitable for achieving external-level harmonization between consumer and provider.

### 5 Matching Relations Between Process Views

In the previous sections, we assume that a consumer and a provider view are identical. In practice, however, they may differ, since both parties develop process views independently. For instance, a provider offers a commoditized view that can be accessed by different consumers. If the provider view differs slightly from the consumer view and is cheap, the consumer can decide to adjust its intended way of working by changing its process view. In these cases, the consumer requires a process view that is similar but not necessarily identical.

In this section, we introduce various matching definitions for comparing a consumer and a provider process view. The definitions guide providers and consumers to find process views that are similar to a requested process view. Each matching definition uses its own notion of similarity, which is useful for a specific purpose.

In the remainder, Section 5.1 introduces different types of matching, motivated by real-world scenarios. Section 5.2 introduces the auxiliary notion of process type that Section 5.3 uses to formalize different types of matching between process views.

#### 5.1 Matching Scenarios

Based on our experience in projects on inter-organizational process integration [22] and existing literature on matching semantic web services [23], [24], [25], [26], we list sample scenarios that motivate requirements on matching definitions:

- A manufacturing organization requires the just-in-time delivery of a high-precision system service for its integration into a custom made production unit. The complexity of the end-product demands strict compliance of the service provider to the specified request of the service consumer to ensure a timely completion of production.

- An organization interacts with a service provider, for example to arrange logistics. A need exists for cross-synchronization between the main process and the process view. To prevent a deadlock, the provider process must be able to replace the projected consumer-process view. However, there is no need for an exact match between provider and consumer view. For example, the provider offers an extra quality check that the consumer chooses to ignore.
An organization outsources part of its business process to a supplier. The outsourcing organization does not require synchronization with the supplier, but wishes to monitor the progress at the provider side. Therefore, the outsourcing organization does not require an exact matching provider process; a close match (in combination with a low price) is sufficient for a provider selection. The outsourcing organization aligns its process view subsequently to the provider view.

As illustrated by the scenarios, various types of matching are useful.

Exact matching of the external process views means both process views are identical, so the provider view strictly complies with the consumer view. Exact matching supports the first scenario.

Plug-in matching means the provider view is a substitute for the consumer view. That is, the provider view inherits from the consumer view. Using plug-in matching instead of exact matching increases the probability of finding a collaborating counter-party since plug-in matching is less restrictive. Plug-in matching supports the second scenario.

Inexact matching means the consumer view only contains the main features of the desired process view and the service consumer accepts process views that are not equivalent but do resemble the query process view. Inexact matching is useful for the third scenario, when the service consumer is willing to adjust its own way of working. After obtaining an inexact match, the consumer creates an aligned process view that is an exact or plug-in match with the provider view. Alternatively, the provider decides to align its process view to the consumer view to obtain an exact or plug-in match.

Failure occurs when no inexact matching is possible, even though the two processes may share some activities. In that case, the process views do not match.

5.2 Process Types

We define the matching relations in terms of process types rather than process trees. The process type of a process tree specifies the behavioral relations between the interactions. The advantage of a process type over a process tree for matching is that process types abstract from irrelevant syntactical details [12].

To define process types, we use auxiliary functions on the syntax of process trees. For a set \( X \) of nodes, the least common ancestor of \( X \), denoted \( \text{lea}(X) \), is the node \( x \) such that \( x \) is ancestor of each node in \( X \), and every other node \( y \), which is ancestor of each node in \( X \) is ancestor of \( x \):

- \( X \subseteq \text{children}^*(x) \)
- For every \( y \in N \) such that \( X \subseteq \text{children}^*(x) \), we have that \( x \in \text{children}^*(y) \).

Since nodes are arranged in a tree, every set of nodes has a unique least common ancestor. For example, in Fig. 5a the \( \text{lea} \) of \( cG \) and \( csR \) is \( \text{SEQ}I \) whereas the \( \text{lea} \) of \( cR \) and \( cE \) is \( \text{IXOR} \).

Based on the notion of \( \text{lea} \), we define behavioral relations on nodes. \( \text{SEQ} \)-nodes order their children and this way induce the before \(< \) relation on their descendants. Given two nodes \( n, n' \in N \), we have \( n < n' \) if and only if

- node \( l = \text{lea}(\{n, n'\}) \) is a \( \text{SEQ} \)-node, and
- node \( l \) has children \( c_n, c_{n'} \) such that \( c_n \) is ancestor of \( n \), so \( n \in \text{children}^*(c_n) \), \( c_{n'} \) is ancestor of \( n' \), \( n' \in \text{children}^*(c_{n'}) \), 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. Given two nodes \( n, n' \in N \), we have

- \( n \cap n' \) if and only if node \( l = \text{lea}(\{n, n'\}) \) is an \( \text{IXOR} \)-node.
- \( n \cap n' \) if and only if node \( l = \text{lea}(\{n, n'\}) \) is an \( \text{EXOR} \)-node.
- \( n \cap n' \) if and only if node \( l = \text{lea}(\{n, n'\}) \) is an \( \text{AND} \)-node.

These three relations are symmetric, so if for instance \( n \cap n' \) then by definition also \( n' \cap n \).

The four defined relations abstract from loops, since they do not take \( \text{LOOP} \)-nodes into account. To remedy this, we superscribe the four relations with multiplicity constraints as follows. For instance, consider two nodes \( n, n' \) of which the least common ancestor is a \( \text{SEQ} \)-node. If the parent of the least common ancestor of \( n \) and \( n' \) is a \( \text{LOOP} \)-node, we write \( n < n \), and we write \( n < n' \) otherwise. In a similar way, we superscribe the \( \text{\lor}, \text{\land}, \text{\&} \)-relations.

This way, we can distinguish a process with a loop from the same process without the loop. Unsubscripted relations can have any multiplicity, so for example \( n \cap n' \) means either \( n < n' \) or \( n' < n \). We assume each \( \text{LOOP} \)-node has no \( \text{LOOP} \)-parent.

In the sequel, we restrict the relations \( <^1, <^*, \text{\lor}^1, \text{\lor}^*, \text{\land}^1, \text{\land}^*, \text{\&}^1, \text{\&}^* \), from nodes to activities, i.e. the leaves of the process tree. By definition of process trees, for any distinct \( a, a' \in A \) we either have \( a < a' \), \( a < a' \), \( a \text{\lor} a' \), \( a \text{\&} a' \), and \( a \text{\lor} a' \) or \( a \text{\lor} a' \). Moreover, a \( \text{op}_a a' \) and a \( \text{op}_a a' \) implies \( \text{op}_a a' \). So for example \( a < a' \) and \( a \text{\lor} a' \) is impossible for the same process tree.

The process type of a process \( P = (N, A, C, \text{parent}, \text{succ}) \), denoted \( PT(P) \), is a tuple containing the different behavioral relations:

\[
PT(P) = (\langle^1, \langle^*, \text{\lor}^1, \text{\lor}^*, \text{\land}^1, \text{\land}^*, \text{\&}^1, \text{\&}^*).
\]

If \( L \subseteq A \), then \( \text{op}_L \) is \( \text{op} \) limited to \( L \), so \( \text{op}_L = \text{op} \cap (L \times L) \). Then \( PT_L(P) \) is obtained from \( PT(P) \) by limiting each \( op \) of \( PT(P) \) to \( L \).

**Example 6.** The consumer process view in Fig. 2 has among others the following behavioral relations for activity deliver regular: get GSM < deliver regular, configure GSM < deliver regular, schedule route < deliver regular, deliver regular \( \text{\lor} \) deliver express, and deliver regular < handover parcel.

We define the union \( \cup \) and intersection \( \cap \) operator on two process types \( PT(P) \) and \( PT(Q) \) by applying these operators to the elements of the tuples. For instance, \( PT(P) \cup PT(Q) \) is the tuple \( t \) such that each element \( t_i \) is the union of the corresponding elements in the individual tuples, \( t_i = PT_i(P) \cup PT_i(Q) \).
5.3 Matching Definitions

We formalize the different matching types in terms of process trees. We use heuristics to define matching relations \[12\]: they are efficiently computable but require additional semantical analysis in the case of exact and plug-in matching.

5.3.1 Exact Matching

Two process trees \(P_1, P_2\) match exactly, denoted \(P_1 \equiv P_2\), if and only if

- they have the same observable and invocable activities, so \(I_{P_1} = I_{P_2}\) and \(O_{P_1} = O_{P_2}\);
- they have the same process type, so \(P(T)(P_1) = P(T)(P_2)\), and
- the multiplicity of the activities is the same, so for each \(a \in A_{P_1}(= A_{P_2})\), the parent of \(a_1\) is a LOOP-node if and only if the parent of \(a_2\) is a LOOP-node.

An exact match does not imply that the process-models are fully equivalent, since the process tree abstracts from atomic actions such as \textit{assign} and \textit{wait}. However, if a user targets a fully equivalent process, still exact matching is useful. For example, a matching engine uses exact matching to filter out relevant candidate processes and then applies to these candidates the full equivalence decision procedures, which are computationally more expensive.

5.3.2 Plug-In Matching

Process tree \(P_1\) is a plug-in for process tree \(P_2\), denoted \(P_1 \triangleright P_2\), if \(P_1\) contains all activities of \(P_2\) and has the same process type as \(P_1\) for these shared activities. Still \(P_1\) might contain other activities that are not in \(P_2\). The definition ensures that \(P_1\) successfully replaces \(P_2\) in every possible context. In particular, the invocations made on \(P_1\) can be made in the same order on \(P_2\). However, \(P_1\) and \(P_2\) must agree on internal and external choices, to avoid that \(P_1\) bypasses by means of a choice construct an obligatory part of \(P_2\) (see Example 7). This relation resembles process inheritance, so \(P_1\) inherits from \(P_2\). Formally, \(P_1 \triangleright P_2\) if and only if

- \(P_1\) offers all activities of \(P_2\), so each activity of \(P_2\) is an activity of \(P_1\): \(A_2 \subseteq A_1\);
- for the common activities, which are in \(A_{P_2}\), \(P_1\) and \(P_2\) agree on the process types and multiplicity: \(P(T)(P_1) = P(T)(P_2)\) and for each \(a \in A_{P_2}\), parent\(_1\)(\(a\)) is a LOOP-node if and only if parent\(_2\)(\(a\)) is a LOOP-node;
- \(P_1\) and \(P_2\) agree on internal and external choices, i.e., \(\nabla_{P_1} = \nabla_{P_2}\) and \(\cap_{P_1} = \cap_{P_2}\).

Note that each exact match is also a plug-in match, but not vice versa. As in the previous case, plug-in matching requires additional heuristics to check at the semantic level and decide whether a service replaces another one in all possible states. Plug-in matching is useful to filter out irrelevant processes before applying such a semantic check.

5.3.3 Inexact Matching

We distinguish two types of inexact matching. The first one is loop sensitive, taking multiplicities into account. Since multiplicity 1 is a special case of * (executing a loop only once), business cases exist where it is useful to ignore the difference between multiplicities. Therefore, we define a second variant of inexact matching that abstracts from multiplicities.

- \(a)\) Loop-sensitive matching: For two process trees \(P_1, P_2\) to match inexactly in the loop sensitive case, written \(P_1 \sqcap P_2\), \(P_1\) and \(P_2\) must have some overlap in process types, thus respecting multiplicities:
  - \(P_1\) and \(P_2\) have activities in common, so \(A_1 \cap A_2 \neq \emptyset\).
  - For the overlapping activities, the process types have an overlap as well. Let \(L = A_1 \cap A_2\). Then \(P(T)(P_1) \cap P(T)(P_2) \neq \emptyset\).
  - The multiplicities of the overlapping activities are the same: for each \(a \in A_1 \cap A_2\), parent\(_1\)(\(a\)) is a LOOP-node if and only if mult\(_2\)(\(a\)) is a LOOP-node.

We rank inexactly matched process trees according to their degree of similarity. For this, we use the simple metric:

\[
\frac{|P(T)(P_1) \cap P(T)(P_2)|}{|P(T)(P_1) \cup P(T)(P_2)|}.
\]

Note that in the case of an exact match, the metric yields 1. Furthermore, if there is no match at all, the metric yields 0. To ensure that a plug-in match yields a number smaller than 1, the denominator considers all possible relations, and not just those in common between \(P_1\) and \(P_2\). For instance, if \(P_1 \triangleright P_2\) and \(P_1\) extends with \(P_2\) with one extra activity, then the common activities \(P_1\) and \(P_2\) have the same behavioral relations, and therefore \(|P(T)(P_1) \cap P(T)(P_2)| = |P(T)(P_1) \cup P(T)(P_2)|\).

- \(b)\) Loop-insensitive matching: For inexact matching, it is useful to ignore the difference between multiplicities. For example, \(a <\cdot b\) matches \(a < b\), since there is only one loop execution in the first case. Therefore, process types must abstract from multiplicities. Given a process \(P\), its abstract process type \(A\text{PT}(P)\) we define as the tuple \((<, \nabla, \&.)\).

For loop insensitive inexact matching, the abstract process ignores multiplicities while types must overlap. Processes \(P_1\) and \(P_2\) match inexactly in the loop insensitive case, written \(P_1 \sqcap P_2\), if and only if

- \(P_1\) and \(P_2\) have activities in common, so \(A_1 \cap A_2 \neq \emptyset\).
- For the overlapping operations, the abstract process types have an overlap as well. Let \(L = A_1 \cap A_2\). Then \(A\text{PT}(P_1) \cap A\text{PT}(P_2) \neq \emptyset\).

A metric similar to the one defined for the loop-sensitive case, ranks different candidate process trees.

Example 7. Consider the process view for provider \textit{Y} obtained from the internal process in Fig. 4 by hiding \textit{py: check GSM}. Compared to the process view of the consumer in Fig. 2, the process view for provider \textit{Y} contains one additional activity, \textit{py: deliver priority}. Though the common activities have equal behavioral relations in both process views, there is no plug-in match, since both process views have different \(\cap\)-relations due to activity \textit{py: deliver priority}. Thus, the process view for \textit{Y} allows that \textit{deliver regular} and \textit{deliver
express are not executed, which is not possible in the consumer view.

Comparing Figs. 2 and 4, we have an inexact match. The consumer view has 14 tuples in the \( \prec \)-relation and 2 in the \( \sqcap \)-relation; all other behavioral relations are empty. The process view for \( Y \) has 18 tuples in the \( \prec \)-relation and 6 in the \( \sqcap \)-relation. Since the consumer view has no additional behavioral relations, \( \preceq_c \subset \prec \) and \( \sqcap_c \subset \sqcap \) and \( |\preceq_c| = 14 \) while \( |\sqcap_c| = 2 \). Therefore, the match is \((14 + 2)/(18 + 6)\) which is 2/3.

6 COLLABORATION CONFIGURATION FOR OUTSOURCING

After laying a foundation by identifying extreme projection relations between an internal process and an external process view, we now turn to establishing a collaboration-configuration between a service consumer and a service provider. The critical issue is the detection of projection and matching constellations so that the overall collaboration configuration is sound, i.e., enacts in synchrony to the end as intended by the collaborating counter-parties. For this purpose, collapsing the consumer- and provider processes is useful. Additionally, collapsing is useful to show the actual outsourcing enactment.

First, Section 6.1 analyzes the possible configuration options that define how the internal processes of the consumer and provider and their shared process views relate to each other. Next, Section 6.2 maps a collaboration configuration to a tree formalization and shows how a collapsing results in a deployment version that is suitable for verification and enactment.

6.1 Configuration Options

The service consumer on the left and service provider on the right of Fig. 6 relate their process views with one another using the matching relations of Section 5. They also relate their process views to their underlying internal processes using the projection relations of Section 4.2. Consequently, only certain combinations of projection and matching relations are possible for process views at the external level.

The internal levels of Fig. 6 show on the left the consumer process and on the right the provider process. For both parties, the figure depicts the useful projection relations near the projection arrow. In the middle of Fig. 6, the tuples with projection combinations represent the meaningful options for establishing process views on the external level. We now explain and motivate these combinations.

If the service consumer performs a black-box projection, the service provider only may use a black-box projection. All other projections result in processes having more than one activity, and thus, the resulting process view of the provider does not equal the consumer-process view. Likewise, if the service consumer performs a glass-box projection, the service provider can only use a glass-box projection, since that is the only projection relation resulting in an external process view with only observable activities. If a service consumer performs a white-box projection, the service provider may respond with either a gray-box, open-box, or white-box projection. Since the consumer-view may contain invocable activities, black-box and glass-box projections do not apply.

Finally, a service consumer cannot use a gray-box or open-box projection. To see why, suppose the outsourced internal process of the service consumer contains an invocable node that is omitted in the consumer-process view. Since the invocable node is not in the process view, the provider process at the internal level does not need to have a corresponding invocable node. However, since the original node at the consumer-side is invocable, that node can be invoked by another internal process of the consumer that interacts with the outsourced process. Consequently, the provider process cannot replace the outsourced consumer process. Therefore, only white-box projection is applicable if the internal process of the consumer contains an invocable node.

All projection options are only usable for exact matching and plug-in matching. Inexact matching is useful to identify a consumer and provider view that are similar, followed by a process-view alignment to yield an exact or plug-in match. This may require a change of the underlying internal processes. An exact or plug-in match can be used with any of the possible projection combinations. In all cases, during actual execution of the provider view, its status gets mirrored in the consumer view. If the match is exact, the views of the consumer and provider are identical. Otherwise, the provider contains additional activities that are not monitored in the consumer view.

6.2 Tree-Formalization Mapping

The outsourced consumer view can interact with other internal consumer processes that are not outsourced. To ensure that the collaboration configuration is deadlock free, collapsing the consumer and provider processes is useful. We show in Fig. 7 the application of an existing collapsing method [21] with a process-tree formalization for which the collaboration configuration is white-box projection for the consumer and gray-box projection for the provider. The outsourced consumer...
process on the top left of Fig. 7 is part of a larger consumer process. The subtree below OSPROC is the outsourced consumer process at the internal level. Consumer activity cOD invokes outsourced activity p:gG and ccN invokes p:cG.

The resulting collapsed configuration on the right side of Fig. 7 requires evaluation with tool support for correctness issues, e.g., deadlocks or lack of synchronization. By representing the resulting process tree in BPEL, it is possible to verify properties, e.g., by mapping to place/transition-nets [27].

Since the collapsing method requires disclosure of the internal-level processes of all collaborating parties, an independent trusted third party must perform the replacement. Otherwise, leaking of business secrets is likely, which endangers competitive advantages. The collapsing is correct by following the projection options identified in Section 6.1; deadlocks do not occur because invocable activities of the process outsourced by the consumer are preserved at the provider side.

Currently, BPEL [14] lacks the language constructs for specifying the linking of respective processes for monitoring enactment progress and for allowing a service consumer to trigger the enactment of activities in the service-provider domain. Potentially, the service provider, does not want to be monitored extensively during enactment. In Fig. 7, links start and terminate the provider process. However, if the collaborating parties agree, cross-platform linking constructs must be available to observe the enactment progress of only specific basic activities in a process view. More information about monitorability can be found elsewhere [21], [28].

7 TOOL IMPLEMENTATION

To validate the feasibility of the framework, we have implemented the projection and matching definitions in a prototype tool. First, we present an algorithm that takes as input a process tree, together with projection rules defined on nodes of the process tree, and returns the projection of the process tree as output. Next, we present the actual implementation. We have applied the prototype tool to the case study discussed in the next section.

7.1 Algorithm

We present a depth-first tree traversal algorithm that performs a projection of a given process tree, given a set of projection rules as defined above. Input to the algorithm are a process tree \( P \) and a set \( M \) of projection specifications for the nodes in \( P \). For each node \( n \), its projection specification \( M(n) \in \{ \text{omit}, \text{hide}, \text{aggregate}, \text{none} \} \) specifies what projection rule applies to the node. We assume that given a node \( n \), the projection specification \( M(n) \) satisfies the preconditions for the corresponding projection rule; an algorithm to check this is omitted due to space limitations. A depth-first traversal of \( M \)-labeling is downward closed, so if \( \text{parent}(n) = n' \) and \( M(n') = l' \) then \( M(n) = l \). Furthermore, if \( M(n) = \text{"hide"} \) and \( \text{parent}(n) \) is a SEQ-node, then for \( n' = \text{succ}(n) \) also \( M(n') = \text{"hide"} \). Given a process tree \( P \) and projection rules \( M \) for nodes of \( P \), we next give Algorithm 1 that returns a process view obtained via projection such that

1. Each node labeled with “omit” or “hide” is removed.
2. For a node \( n \) that is labeled “aggregate” such that \( n \) is the root of an aggregated subtree, we collapse the subtree into \( n \) and replace \( n \) with \( n_{\text{agg}} \).
3. A labeled node may not be the only child of any node. In such a case we delete the parent node and the parent of parent is connected to the child.
Algorithm 1 project(P,M).

1: {Steps 1) and 2) for node n of P in DepthFirstSearchPostOrder do
2: if M(n) = “omit” or M(n) = “hide” or M(n) = “aggregate” and n ≠ root and M(parent(n)) = “aggregate”) then
3: remove node n from P
4: remove any link to/from n
5: if n ≠ root then
6: remove edge (parent(n), n)
7: else if M(n) = “aggregate” [subtree root] then
8: remove edge (parent(n), n)
9: replace node n with node nagg
10: {Step 3}
11: for node n of P in DepthFirstSearchPostOrder do
12: if |children(n)| = 1 then
13: let c be the child of n
14: if n ≠ root then
15: add edge (parent(n), c)
16: remove edge (parent(n), n)
17: remove edge (n, c)
18: remove node n
19: return P

7.2 Implementation

An Eclipse-based prototype, available at https://github.com/koppor/outsourcing, implements the projection and matching definitions, including the algorithms mentioned in Section 7.1. The prototype uses the Eclipse BPEL model (http://www.eclipse.org/bpel/developers/model.php) and the JGraphT library (http://jgrapht.org) to represent process trees. Input to the prototype are two BPEL files plus two text files specifying the projection rules that have to be applied. The projection rules use XPath expressions to identify the BPEL nodes that need to be projected. A projection rule for aggregation in addition specifies the new activity.

Class Projections performs these projections on a process tree. Afterwards, the process type of the process tree is determined. Thereby, class LCA determines the least common ancestor, which is computed in polynomial time [29]. Class ProcessTree implements the compare methods matchesExactly(ProcessTree), isPluginForProcessTree(ProcessTree), and getDegreeOfInexactMatching(ProcessTree), which implement the matching definitions of Section 5.1.

8 CASE STUDY

The case study stems from the domain of distributed, inter-organizational development processes in the automotive industry as studied in the CrossWork research project [22], [30]. Observing the complexity of such B2B-collaborations from a business and a technical point of view reveals a scenario as described in Fig. 9. An original equipment manufacturer (OEM) rests on top of the depicted B2B-pyramid and is responsible for engineering a product and setting up the machinery and plant construction for production. On the first tier, producers assemble systems and modules stemming from suppliers of the second tier. Finally, suppliers of raw materials and standardized parts are at the bottom of the supply-chain pyramid.

In such supply chains, the OEM as a service consumer pushes the responsibility for accurate service provisioning down the pyramid to first-tier suppliers as service providers while the first tries to concentrate tight control at the top tier. The suppliers must perform a mirroring of the particular outsourced parts of the OEM’s internal process. In an extreme case, the service consumer dictates the specified control-flow, data-flow, resources, and so on, in the services of a provider. However, in other industry
domains, the opposite extreme is thinkable where a service consumer does not impose any restrictions on steps that create the desired service provisioning. The remainder of this section addresses the mentioned problems inherent to supply-chain hierarchies as depicted in Fig. 9. Based on the collaboration framework of Fig. 1, we populate the internal and external levels with models of processes for the inter-organizational production of a water tank that is part of a truck. For brevity, we visualize the BPEL-based complex processes from the CrossWork case study in BPMN.

In the remainder below, Section 8.1 describes how the service consumer outsources a business-process sphere from the internal domain to a cluster of small and medium-sized service providers. Section 8.2 details external-level collaboration specifics of several clusters.

### 8.1 Consumer Service

In the case study, a truck-producing OEM has several suppliers of services united in a regional automobile cluster of service providers. We assume the existence of internal processes in the domains of the service consumer and service provider. The service consumer proposes the creation of a service-outsourcing configuration. The negotiations about creating a common external-level process follows an integration of the business processes in the respective domains. Next follows a verification of the service-outsourcing configuration and a negotiation of the extent of monitoring. Finally, the enactment of the completed service-outsourcing configuration commences.

### 8.2 External-Level Process View

Fig. 10 depicts the in-house process setup for the OEM. On top, the internal level shows the OEM’s business process that orchestrates meaningful automation of human action. Fig. 10 depicts the internal-level process for producing a truck in which unlabeled boxes represent activities of which we omit unnecessary details. Contained in the in-house process is a subprocess for producing a water tank that the OEM plans to outsource to its suppliers. The OEM uses a process view to select the right provider. The process view depicted in Fig. 11 shows the external-level details for the subprocess in Fig. 10 from the perspective of the OEM.

The OEM projects the outsourced sphere to the external level as a process view that is visible for all potential service providers (see Fig. 11). Since a single service provider is not capable of handling a request by itself, a group of providers collaborate as a team in a cluster to fulfill the request. Clusters in the case study are formed ad-hoc, based on a specific request by the OEM. Fig. 12 shows the internal process of one cluster. Different activities in the process are performed by different providers in the cluster, but this is not shown here. By aggregating the grommet-related activities of the internal process, an open-box process view can be constructed that matches exactly with the requested consumer view.

Several clusters may engage in bidding for servicing a particular outsourced sphere from the external level, which allows the service consumer to pick the “best deal”. Figs. 13 and 14 show two process view bids of two alternative clusters A and B. If the exactly matching offer of the cluster in Fig. 12 is not available, the OEM can choose for one of these two alternative clusters. The process view of cluster A in Fig. 13 does not match exactly, since it contains one extra compound activity, assemble watertank body. The inexact match with the process view in Fig. 11 is $(9 + 12)/(9 + 3 + 12 + 6) = 21/30 = 7/10$ since Fig. 11 has 9 tuples in the $<\text{-relation}$ and 12 tuples in the $\&\text{-relation}$, while Fig. 13 has 3 additional tuples in the $<\text{-relation}$ and 6 extra tuples in the $\&\text{-relation}$.

The process view of cluster B in Fig. 14 does not match exactly since the ordering between producewatertankbody
and \texttt{producedispenser} is different from the one in Fig. 11. There is also no plug-in match, since the ordering is different. The inexact match we compute as follows. The process views have 9 shared tuples in their $<$-relations and 10 shared tuples in their $\&$-relations. The process view in Fig. 11 contains two additional tuples in its $\&$-relation while Fig. 14 contains one additional tuple in its $<$-relation. The inexact match is therefore $\frac{9 + 10}{9 + 1 + 10 + 2} = 19/22$.

Whichever process view is selected, further alignment is needed to establish a valid collaboration configuration (cf. Section 6), so either the consumer process or the provider process view is selected.
process needs to be changed such that an exact or plug-in match can be made for their process views. Since the bid cluster B matches best with the requested view, alignment for that process view is probably easiest. However, it might be that the bid of cluster A is that cheap or the quality of the cluster that high that the OEM decides for A.

8.3 BPEL Example

We show in Fig. 15 the BPEL model for the service consumer process view in Fig. 11. We omit the other BPEL models due to space limitations but all BPEL models are available for download at https://github.com/koppor/outsourcing/tree/master/processes/CrossWorkCase-Study. The complete process is an abstract BPEL process (line 2) that is not executable by an engine, since it represents a process view. BPEL does not support the outsourcing features considered in this paper, but can be easily extended [31]. Lines 4 to 8 declare the BPEL extension supporting the outsourcing framework presented in this paper. The process starts with a sequence (line 9) of activities. First, the preparation of the water tank specification (line 10) is executed. Next, a flow construct (line 11) comprises four parallel branches invoking the production services (lines 12 to 15). Finally, the water tank is assembled (line 17). We use the esrc:status attribute to specify that a node is invocable or observable (line 10, lines 12 to 15, and line 17). The BPEL model does not contain the message flows expressed in Fig. 11. These flows can be expressed in BPEL4Chor [32], which supports the specification of interacting BPEL processes.

8.4 Discussion

The case study shows the feasibility of the service outsourcing framework. We have applied the prototype tool discussed in Section 7 to the BPEL models presented in this section; the construction and matching of process views completes within a few seconds. The framework uses polynomial algorithms and therefore scales well. The framework is therefore useful in scenarios in which speed is important, for example in the case of dynamic business networks [30]. As we discuss in Section 9, there is no comparable framework that covers both the construction and matching of process views. Related matching approaches only support inexact matching. The framework supports the specification of fine-grained monitoring and control properties, which significantly extends current state of the art.

The framework focuses on process aspects of business collaborations, captured in process views. It assumes the parties already agree on the structure and meaning of
The service outsourcing framework supports inter-organizational business-process collaboration. Related work in this area is too extensive to discuss here. We focus on the two most related subareas: process views and matching.

9.1 Process Views

Earlier research [7], [34], [35] recognizes the importance of public process views for service outsourcing. However, these approaches focus on how to support a process view at run-time and do not address how to construct a process view. Chebbi et al. [36] consider the construction of process views, but use only one projection relation. The framework in this paper is most closely related to the work of Grefen et al. [7], which defines several of the projection options identified in Section 4.2. But the framework can be combined with these other papers.

Schumm et al. [37] present a categorization of process-viewing patterns. Other papers study process views [5], [6], [9], [10], [38]. Van der Aalst and Weske [38] define how to derive private local processes from a global public process view. The projection relations they identify correspond to hiding and omitting. Bobrik et al. [10] study the construction of personalized role-based public process views. Eshuis and Grefen [5] and Liu and Shen [9] focus on how to derive a process view from an internal process. Aggregation is a key abstraction principle in both approaches, while Eshuis and Grefen also use a form of hiding. None of these process-view approaches focus on service outsourcing, and consequently none identify projection options and meaningful configurations for partners in an outsourcing collaboration.

Preuner and Schrefl [20] define an approach for combining several process-based services into a compound process. They define two consistency relations between the composed process view and the underlying processes: observability consistency and invocability consistency. Observability consistency resembles hiding, whereas invocability consistency resembles omitting. They do not consider aggregation, different projection options or collaboration configurations.

Next, the BPEL standard [14] distinguishes between abstract and executable processes, where abstract processes correspond to process views. However, there are no concrete guidelines for relating abstract and executable processes. Khalaf et al. [39] discuss patterns for relating an abstract process to an executable process. The patterns related to the framework in our paper are export (creating an abstract process from executable process) and import (creating an executable process from an abstract process). The three projection rules in Section 4 are instrumental in combination with the export and import pattern. Martens et al. [40] proposes a Petri net-based approach to check the consistency of an abstract and an executable BPEL process. König et al. [41] and Zhao et al. [42] define several syntactic guidelines for transforming an abstract BPEL process into an executable one with hiding and omitting. None of these papers distinguishes between different projection relations and none consider aggregation.

Theoretical work exists on the problem of compatibility checking of Petri-net based services [27]. In principle, this work serves to check the compatibility of a consumer and a provider process. However, any combination of services that does not deadlock is correct, which is not suitable for outsourcing. The requirements for outsourcing are stricter since a process view must mirror the provider process. Thus, even though a consumer process and provider process are compatible, they may not be in an outsourcing relation, for example, if the provider removes an observable activity that the consumer needs to monitor.

Contractual-visibility patterns for inter-organizational business-process [21], [28] also assume the projection of a partitioned internal-level process to an external level, leading to different levels of visibility that resemble the projection relations presented in Section 4.2. The main difference between contractual-visibility patterns and projection relations is that the latter also incorporate the accessibility of projected nodes i.e., invocable or observable. Whereas the contractual-visibility patterns only focus on the relationship between the sets of nodes in the processes of the internal and external level and address the accessibility with separate so-called monitorability patterns [21]. By combining the contractual-visibility with monitorability patterns all projection relations can be realized. Another difference is that the contractual-visibility patterns are expressed in Petri-nets, whereas the framework uses BPEL.

In sum, the service outsourcing framework contributes a comprehensive set of projection relations between internal processes and process views that allow a service consumer to monitor and control the outsourced process view. Related work on process views focuses mostly on one projection relation and does not support monitoring and control of process views. Another novel aspect of the framework compared to related work is that it considers the interplay with matching process views by analyzing meaningful collaboration configurations.

9.2 Process Matching

There are a few papers that address the problem of matching and retrieving BPEL processes [11], [13], [43].
Corrales et al. [13], [43] describe an approach for inexact matching of BPEL processes based on the theory of graph matching. Each BPEL process translates into a graph. Two BPEL graphs are compared using the notion of edit distance, expressed in terms of the number of edit operations needed to transform one graph into the other. Matching is done entirely on the syntax level, ignoring behavioral aspects like choice and parallelism which are considered by the inexact matching definitions proposed in Section 5. Moreover, Corrales et al. [13], [43] do not consider other matching definitions, so they cannot support all scenarios listed in Section 5.

Beeri et al. [11] define a language for querying BPEL processes. In addition, they define an abstract syntax for BPEL, which consists of a set of process graphs and an implementation relation that specifies for each compound node of a process graph which other process graph implements the node. Each query is expressed as a structural pattern that is evaluated on this abstract syntax. The query language allows navigation along two dimensions: the path-dimension inside process graphs and the aggregation-dimension between process graphs. Again, matching of a query pattern with a BPEL process structure ignores behavioral aspects as it is done on the syntax level, and only one kind of matching is defined, so not all scenario outlined in Section 5 can be supported.

Next, there are matching approaches that use sequential automata [44], [45], [46], [47] or Petri nets [48] to formalize behavioral aspects of web services. However, in the context of BPEL and OWL-S, the translation of constructs exhibiting parallelism leads to a state explosion, because the cartesian product of all parallel branches has to be constructed. Thus, such a translation is not scalable, and leads to an inefficient matching approach. The matching approach of this paper is scalable, since process trees can be derived in linear time from BPEL models and the matching algorithms in Section 7 are polynomial.

In sum, the service outsourcing framework contributes a set of matching definitions for BPEL processes that support multiple scenarios, capture multiple behavioral aspects like sequence, choice, and parallelism, and can be efficiently checked. Approaches in related work support only one type of matching and focus either on syntactic similarity of BPEL processes [11], [13], or are not efficient in the context of BPEL [45], [46], [47], [48]. Another contribution of this paper is an analysis of the interplay between the construction and matching of process views.

10 Conclusion

Key contribution of this paper is a collaboration framework that supports outsourcing of business processes between parties based on process views. The separation of public process views from private internal processes is vital to support B2B-collaboration while maintaining privacy. The distinction between observable and invocable activities allows a service consumer to exert different levels of control over an outsourced activity. The framework considers both the construction of public process views by projection of internal processes, and the matching of consumer and provider views to measure their similarity.

The projection rules of the framework use the abstraction principles of hiding, omitting and aggregation, which allow for a flexible relationship between process views and internal processes. Based on the abstraction rules, the framework defines several extreme projection relations that can exist between a process view and an internal process. We showed that existing approaches for process views from literature mostly focus only on one of these projection relations, while the business-collaboration framework of this paper is more comprehensive. Also, existing process view approaches do not consider matching of process views.

The matching relations defined by the framework indicate the level of similarity between a consumer and provider process view, which is useful to find a similar process view or to establish a collaboration between consumer and provider. The matching relations employ heuristics, which allow for an efficient implementation. Due to its efficiency, the approach is useful in scenarios in which speed is important, for example for dynamic business networks [30].

We analyzed the interplay between projection and matching relations by identifying valid collaboration configurations for a service consumer and service provider. In particular, we show that not every combination of projection and matching relations for provider and consumer-side processes is meaningful. The projection relation used by a party to construct a process view, limits the matching definitions that the party can use to find related process views.

Open issues for future research focus on supporting applications to set up and enact collaboration configurations. As process description language, we consider BPEL. However, BPEL requires extensions with additional language constructs that allow a service consumer to start and stop the enactment of provider processes and to remotely observe the enactment progress of the provider process. For a distributed setup and enactment of a collaboration configuration, it is necessary to develop a reference architecture for supporting application systems. We are currently evaluating the BPEL4Chor ecosystem [32] as runtime infrastructure. Another topic for future research is applying process matching to ontology languages like OWL-S.

Acknowledgment

This work was partially funded by the BMWi project Cloud-Cycle (01MD11023) and the TIVIT project Cloud Software.

References

Oliver Kopp received the diploma degree from University of Stuttgart, Stuttgart, Germany, in 2005. He is a Research Assistant and PhD student at this University. In his PhD thesis, Oliver focuses on distributed transactions and global fault handling in service choreographies. His past projects include the Tools4BPEL project, where analysis tools for BPEL processes have been developed and the EU-project COMPAS, where compliance-driven models, languages, and architectures for services have been researched. Currently, he is part of the CloudCycle project, where the focus is on provisioning and management of portable cloud-services with guaranteed security and compliance throughout their complete lifecycle.

Esa Pitkänen received the PhD degree in computer science from the University of Helsinki in 2010. He is a postdoctoral researcher at the University of Helsinki in Aaltonen lab, Genome-Scale Biology Program.