Implementing Projections of Abstract Interorganizational Business Processes

Julius Köpke, Johann Eder, Markus Künstner

Department of Informatics-Systems, Alpen-Adria Universität Klagenfurt, Austria
{julius.koepke, johann.eder, markus.kuenstner}@aau.at
http://isys.uni-klu.ac.at

Abstract. Distributed interorganizational business processes are constituted by cooperating private processes of different partners. These private processes must be aligned to fulfill the overall goal of the (virtual) interorganizational process. Process views are considered as an excellent concept for modeling such interorganizational processes by providing a good balance between the information needs and privacy requirements of the partners. We follow a top-down development approach, where first an abstract global process is designed. Then each task of the global process is assigned to one of the partners. Finally a partitioning procedure is applied to generate views of the global process for each partner. The partners can then develop or adopt their private processes based on their views. We present a novel algorithm for the automatic partitioning of any block-structured process with any arbitrary assignment of partners and prove its correctness.

1 Introduction

Interorganizational cooperations between business partners are nowadays considered to be crucial for increasing productivity. In recent years many approaches were developed to support process cooperation with web technologies [7, 15, 2] in general and in particular with SOA-based protocols between business partners [1]. There are two main approaches: Orchestrations where a local (private) process is executed and controlled by a single orchestration engine and communicates with external services, that can be entry or exit points of partner processes; or choreographies which define a communication protocol (message exchanges) between collaborating business partners, which can be interpreted as a decentralized global process [13].

Process views [5] are an appropriate approach to represent the externally observable behavior of business processes and to balance the request for privacy and loose coupling between processes with the communication demands for collaboration. Most approaches for process-views such as [6, 2, 11] follow a bottom up or *global as view* approach. A view is derived from a private process definition which is actually instantiated and executed in a process engine at runtime. The integration of such cooperating views constitutes an interorganizational business process.

Views can also be used to distribute the steps of a global (interorganizational) process definition. In this top-down or *local as view* approach first an abstract process is defined as a global interorganizational process. The activities contained in this process definition are then distributed onto the involved partners. Since the process definition is abstract, it means that none of the steps are executed *globally* but that each step which is defined in the global process is executed by one of the participating processes. The projection of the global process onto a particular participant defines a view, i.e. a workflow derived from the abstract global process. The views of the global process specify the obligations of the partners (execution of steps defined in the global process) and the externally observable behavior of the private processes (choreography). The p2p approach presented in [19, 16] is an example for such a top-down modeling approach.

In this paper we present an automatic process partitioning algorithm that can be used in a top-down development approach. Starting with a global process each step of the process is assigned to one of the partners. In a next step the global process with partner assignments is partitioned fully automatically into views for each partner. We prove that the union of the generated views correctly realizes the fully distributed execution of the global process. While top-down process projection approaches have already been proposed in [16, 19, 17], we overcome limitations of existing approaches by supporting the correct partitioning of any block-structured global process with any arbitrary partner assignment and by explicitly addressing the distribution of decision variables to achieve deterministic rather than indeterministic processes.

2 Process Model

We provide the preliminaries for our approach with a meta-model. It is based on the capabilities of full-blocked workflow nets supporting sequence, PAR split/join, XOR split/join, and LOOP split/join as control flow patterns [18, 14]. We focus on fully blocked workflow-nets as they prevent typical flaws of unstructured business processes dealing with data [3] and are also in line with the WS-BPEL[12] standard. The meta-model in Figure 1 shows the essential components of this model in a simple way. The representation of algorithms in the following sections is based on this meta-model. Fully blocked workflow - nets can be represented either as graphs or as hierarchies of (abstract) activities. The meta-model captures both representations - the transformation between these two representations is straightforward. Therefore, we use the appropriate representation in our different algorithms.

A process is defined by activity-declarations, variable declarations and the control-flow between activities expressed by steps that refer to activities (activity-steps) or that define the control-flow in terms of sequences, XOR-, Parallel-, and Loop-constructs. We also introduce abstract steps which represent any abstract activity as black boxes. A step is associated with partners to define which partner is responsible for the execution. For activity-steps one partner can be assigned. For the control steps PAR, XOR, and $LOOPS$ two potentially different part-

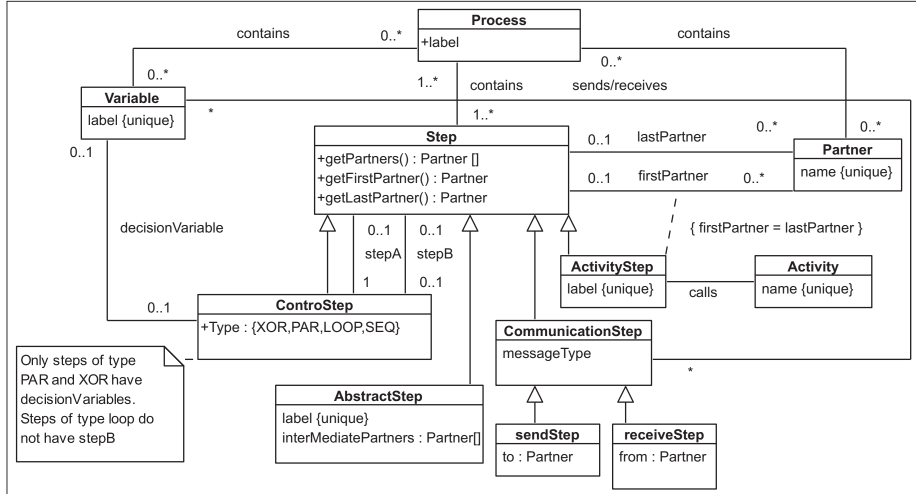

Fig. 1. Workflow Meta-Model

ners ($firstPartner$) and ($lastPartner$) can be assigned that are responsible for the split (decision or parallel invocation) and the join (synchronization) respectively. In this meta-model we abstract from data with the exception of decision variables. $XOR-$ and $LOOP-$ blocks are typically associated with a condition which is a boolean expression based on variables. In interorganizational P2P workflows these expressions are not viable since they would require that all variables in the condition expressions are synchronized between the partners. Therefore, we restrict the control flow decisions to single boolean variables. This approach can also be interpreted as one partner executes an expression and communicates the result to the involved partners. This restriction does not reduce the applicability of our model since any input process can be transformed to a process where all decisions are based on simple boolean variables by adding additional steps that write the results of the boolean expressions to decision variables.

For $XOR-$ and $LOOP-$ blocks the $firstPartner$ is responsible for the decision. Therefore, he must be able to compute the value of the decision variable. Sequences, PAR, and XOR have two sub-blocks, Loops have one sub-block. For $XOR-$blocks the sub-block *stepA* is executed if the decision variable is *true*, otherwise *stepB*. In case of loops the sub-block *stepA* is executed, if the decision variable is *true*, otherwise the loop is not entered or terminated. The sub-block *stepA* is the first block of a sequence, *stepB* is the second one. For all steps the method $getPartners()$ returns a list of all partners that are (recursively) involved in the step. Partners communicate with communication steps (*sendSteps* and *receiveSteps*) which realize the control flow and pass decision variables.

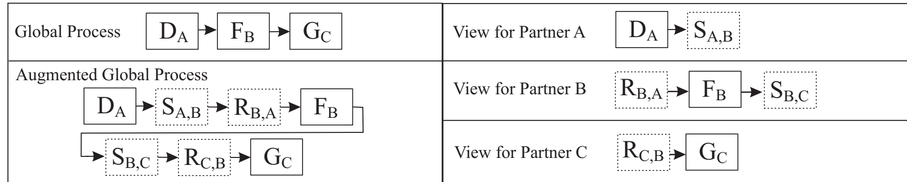

Fig. 2. Partitioning of Sequences

3 Projecting a Global Process to Views for each Partner

As in [16, 19] we propose the following procedure for defining interorganizational workflows: (1) define a global (abstract) workflow, (2) assign each step in the workflow to a partner (3) partition the process. (4) Create private processes based on the generated views. The global process is executed by the distributed execution of the private processes. While the work in [16, 19, 17] concentrates on the question whether a private process correctly implements a (public) view (is in accordance with) our aim is to provide a partitioning procedure that does not impose any restrictions on the global process and on the partner assignments. Additional design rationales are: (1) the resulting views should be as simple as possible, and (b) unnecessary message exchange should be avoided. E.g. a partner should only know about control structures and receive only messages that are absolutely required. In the following we discuss our partitioning method by examples for each control structure. Our approach operates in two phases: First the global process is augmented with communication steps that realize the interorganizational control-flow and variable passing. In a next step views for the partners are created by projection.

Sequence Blocks: In Figure 2 an example for the partitioning of a sequence is shown. The global process contains the steps D_A, F_B and G_C in a sequence. D is defined to be executed by A, F is executed by B and G is executed by C. In order to support a distributed execution the global process is first augmented with pairs of send- and receive- steps whenever a step is followed by another step which is assigned to a different partner. In particular in the example send- and receive- steps are inserted between D_A and F_B and between F_B and G_C in the augmented global process. The corresponding views for each partner are shown on the right. They are created by simply projecting only steps that are assigned to the specific target partner.

Parallel Blocks: An example for the augmentation and partitioning of a parallel block is shown in Figure 3. Partner A is responsible for the parallel split. The partners B and C execute their tasks F and G in parallel. Partner D is responsible for the synchronization. The process is augmented by send- and receive-steps between partner A and B, and A and C and finally between B and D and C and D. While the parallel split and join steps remain in the views of partner

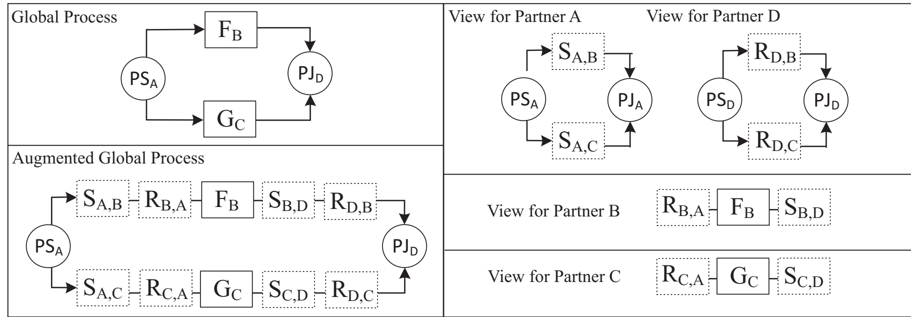

Fig. 3. Partitioning of Parallel Blocks

A and D, partners B and C do not need to know about the parallel execution and get only sequences in their views.

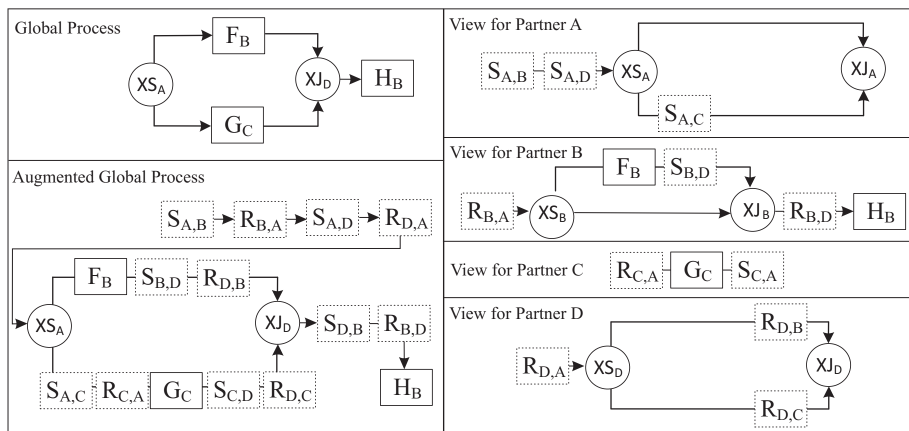

Fig. 4. Partitioning of XOR-Blocks

XOR-Blocks: XOR-blocks are based on some decision variable. The owner of the XOR-split takes the decision. All partners who need to execute the XOR split themselves are informed about the decision by a message transporting the decision variable. In the example in Figure 4 partner A is responsible for the XOR split and distributes the decision variable to B and D. Partner C does not need to know about the decision because he does not need to have a XOR-block in his view. After the XOR-split the control-flow must be forwarded. This is realized by a send- and receive- steps between A and C. There is no such communication required between A and B because B can proceed as soon as the value of the decision variable was received.. The XOR join is executed by

partner D. Therefore, like for parallel join nodes the last partner in each branch needs to pass the control-flow to the join partner. However, the join partner requires the decision variable from partner A to know whether the incoming message will arrive from B or from C.
In the example partner C has a sequence of steps in his view rather than an XOR-Block. This simplification is possible because C takes exclusively part in one branch of the XOR-block (and in no other step before, after, or parallel to this XOR-block). Therefore, C does not need to know about the XOR-block at all. Such a simplification is not possible for partner B because B must know whether step F needs to be executed before step H or not.

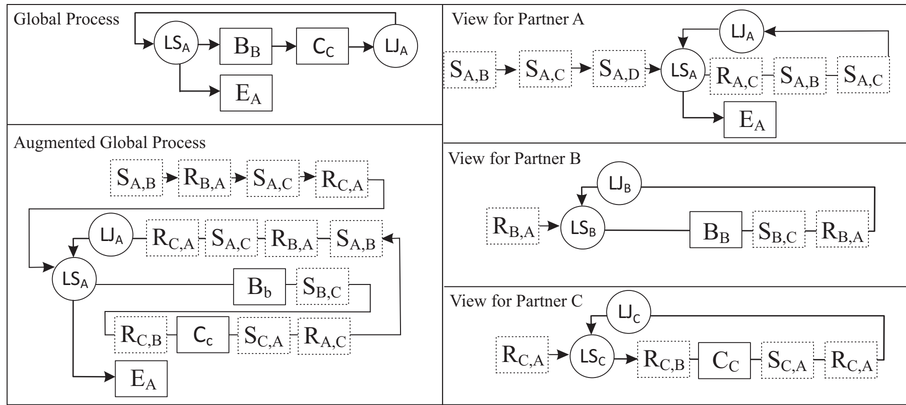

Fig. 5. Partitioning of Loop-Blocks

LOOP-Blocks: Loops are partitioned into views in analogy to XOR-blocks. The first partner in the loop (assigned to the loop-split node) is responsible for the decision. Therefore, the first partner distributes the decision variable to the partners that take part in the loop. An example is shown in Figure 5. As for $XOR-$blocks, the first partner in the loop body (B) does not need to be explicitly called by A because he receives the decision variable from A and can directly execute the loop split and the activity step. After each iteration of the loop the current version of the decision variable is distributed to all relevant partners by a sequence of send- and receive- steps that are added directly before the loop-join.

4 Automatic Generation of Views

After we have introduced the general idea of the partitioning processes we will now present an algorithm that realizes the discussed methods fully automatically. As in the examples the approach is based on two phases: Augmentation and projection.

4.1 Augmentation

The augmentation algorithm is shown in Algorithm 1. It directly operates on the hierarchic process representation as discussed in Section 2. It is called with the root block of a process (*inBlock*) and recursively traverses the process tree. Depending on the type of the input block (*inBlock*) the different augmentation patterns introduced in Section 3 are applied. In particular, in case of sequences send- and receive steps are inserted between two subsequent elements $< e1, e2 >$, if $e1.lastPartner \neq e2.firstPartner$ and if the steps are not communication steps. In case of $XOR-$ and $LOOP-$blocks the helper function $addConditionMessages()$ is used to distribute the decision variable from the partner that executes the split ($block.firstPartner$) to all partners that are different from $block.firstPartner$ and that take part in the block (directly or indirectly). In case of an $XOR-$block the conditional messages are only added if the partner takes part in both branches, or when the partner is responsible for the join (is the last partner), or when the partner takes part in only one branch and he also takes part in some other block which is not a child of the current block). Finally the control flow needs to be correctly passed not only between elements in a sequence but also between different partners in $PAR-$, $XOR-$ and $LOOP-$ blocks. This is realized by the helper procedures $augmentSplit()$ and $augmentJoin()$ that add the required send- and receive- steps between the partner responsible for the split (*block.firstPartner*) and the first partner of *stepA* (*block.stepA.firstPartner*) and the first partner of *stepB* (*block.stepB.firstPartner*, not for loops) and between the last partner of the block and the last partner of *stepA* (*block.StepA.lastPartner*) and the last partner of *stepB* (*block.StepB.LastPartner*, nor for loops) respectively. The called procedures *addConditionMessages(), addSendReceive(), augmentSplit(), and augmentJoin()* add the required send- and receive steps such that the triple *(sendNode.to, receiveNode.from, messageType)* is always unique.

4.2 Projection

After the global process is augmented with send- and receive- steps the views for each partner are created by simply projecting the steps to the specific partners. In order to project the augmented global process to some partner p, the following steps are performed: (1) Blocks that do not contain any direct or indirect partner assignment to p are removed. (2) Control steps (XOR, LOOP, PAR), where p takes part in are rewritten to p. Thus, the first and the last partner (split and join node in graph representation) are assigned to p. (3) A special case are abstract steps that may contain multiple partners. For such blocks the block itself is projected to p by removing all partner assignments (*firstPartner, lastPartner, interMediatePartners*) that are not equivalent to p. All these operations can easily be realized in a simple traversal of the view. The corresponding procedure *projectProcess()* is shown in algorithm 2. It is called with an augmented global workflow *inBlock* and a partner p. The output of the procedure is the view for partner p.

Algorithm 1 Augmentation of a Global Process

```
1: Procedure augmentProcess
Input: RootBlockOfGlobalProcess inBlock
2: if (inBlock.type = ActivityStep or inBlock.type = Abstract) then
3:   return
4: end if
5: if (inBlock.type = SEQ or inBlock.type = XOR or inBlock.type = PAR) then
6:   augmentProcess(inBlock.StepA); augmentProcess(inBlock.StepB);
7: end if
8: if (inBlock.type = SEQ) then
9:   if ((inBlock.StepA.lastPartner != inBlock.StepB.firstPartner) and
     !(stepOf(inBlock.StepA.lastPartner).type = CommunicationStep and
     stepOf(inBlock.StepB.firstPartner).type = CommunicationStep)) then
10:    addSendReceive(inBlock)
11:  end if
12: end if
13: if (inBlock.type = XOR) then
14:   for all (partner ∈ inBlock.getPartners()) do
15:     if partnerOnlyInOneBranch(partner,inBlock) then
16:       augmentSplit(inBlock)
17:     end if
18:   end for
19:   augmentJoin(inBlock)
20:   for all (partner ∈ inBlock.getPartners()) do
21:     if (partner ≠ inBlock.firstPartner and (partnerInBoth-
       Branches(partner,inBlock) or partner = inBlock.lastPartner or
       (partnerOnlyInOneBranch(partner,inBlock) and !partnerOnlyInThis-
       Block(partner,inBlock)))) then
22:       addConditionMessages(inBlock,partner)
23:     end if
24:   end for
25: end if
26: if (inBlock.type = PAR ) then
27:   augmentSplit(inBlock); augmentJoin(inBlock);
28: end if
29: if (inBlock.type = LOOP ) then
30:   for all (partner ∈ {inBlock.getPartners()}) do
31:     if (partner ≠ inBlock.firstPartner) then
32:       addConditionMessages(inBlock,partner)
33:     end if
34:   end for
35:   augmentSplit(inBlock); augmentProcess(inBlock.StepA);
36: end if
```

4.3 Cleanup

After the views are created for each partner some cleanup steps are executed using an additional traversal of the views. The following cleaning actions are

Algorithm 2 Create a View for a Specific Partner

```
1: Procedure projectProcess
Input: RootBlockOfAugmentedGlobalProcess inBlock, Partner p
2: if ({p} ∩ inBlock.getPartners() == {}) then
3:    hide(inStep)
4:    return
5: end if
6: if (inBlock.type = Simple}) then
7:    return
8: end if
9: if (inBlock.type = Abstract}) then
10:   projectAbstractStep(inBlock,p)
11:   return
12: end if
13: if (inBlock.type = SEQ) then
14:   projectProcess(elem.StepA,p)
15:   projectProcess(elem.StepB,p)
16: end if
17: if (inBlock.type ∈ {XOR, AND, LOOP}) then
18:   projectBlock(inBlock,p)
19:   projectProcess(inBlock.stepA,p)
20:   if (inBlock.type ≠ LOOP) then
21:      projectProcess(inBlock.stepB,p)
22:   end if
23: end if
```

performed for each view:
(1) If a $PAR - Block$ has one empty branch ($StepA$ or $StepB$ is null) in the view, then the $PAR - Block$ is replaced by a sequence. (2) If a $XOR - Block$ has one empty branch ($StepA$ or $StepB$ is null) and the first and last partner of the corresponding XOR-Block in the global process is not assigned to the partner of the view (the partner of the view is not responsible for the split and join), and the partner of the view is not assigned to any other block that is executed before or after the XOR-block, then the $XOR - block$ is replaced by a sequence. (3) Sequences that are containing only one element are eliminated.
The procedure *cleanupView()* shown in Algorithm 3 is called with an input view *inBlock* and a view partner *p*. The algorithm returns a cleaned and optimized view for partner *p*. The procedure traverses the hierarchical process tree recursively and applies the patterns stated above depending on the block type. If *inBlock* is of type *SEQ*, the helper function *replaceSequenceByStep()* is called if the sequence contains exactly one element. The method *replaceSequenceByStep()* fulfills requirement number(3) and replaces the sequence by the contained block. If *inBlock* is of type *AND*, the helper function *replacePARBySequence()* will be called if exactly one branch of the block is empty. This method fulfills the requirement number (1) by removing the PAR block and replacing it by a sequence. If

Algorithm 3 Cleanup of a Private View

```
1: Procedure cleanupView
Input: View inBlock, Partner p
2: cleanupView(inBlock.stepA)
3: if inBlock.type != LOOP then
4:    cleanupView(inBlock.stepB)
5: end if
6: if inBlock.type = SEQ and inBlock.size = 1 then
7:    replaceSequenceByStep(inBlock)
8: else if inBlock.type = AND then
9:    if (inBlock.stepA = null or inBlock.stepB = null) and !(inBlock.stepA = null
      and inBlock.stepB = null) then
10:      replacePARBySequence(inBlock)
11:   end if
12: else if inBlock.type = XOR then
13:   if (inBlock.stepA = null or inBlock.stepB = null) and !(inBlock.stepA = null
      and inBlock.stepB = null) and partnerOnlyInOneBranch(inBlock, partner) and
      !partnerIsOriginalOwnerOfSplitOrJoin(inBlock, partner) then
14:      replaceXORBySequence(inBlock)
15:   end if
16: end if
```

inBlock is of type *XOR*, the helper function *replaceXORBySequence()* is called to fulfill requirement number (2) if the conditions stated above hold.

5 Simple Merge of Views

While the process partitioning approach is already finished after augmentation and projection we will now discuss how a set of views $V = v_{p1} \ldots v_{pn}$ of all n partners that adhere to some global process can be merged in order to reconstruct the global process. We can assume that the generated views are structurally equivalent in the sense that no equivalence transformations [4] of the views need to be applied in order to merge them. We will discuss a simple process merge method that exploits this property and operates on the graph representation of the views here. Each view $v \in V$ can be represented as a graph (V, E) (in particular a directed, attributed, acyclic graph). Each node in v has a type $\in$ {XOR-Split, XOR-Join, PAR-Split, PAR-Join, LOOP-Split, LOOP-Join, ACTIVITY-STEP, Send-Step, Receive-Step, AbstractStep}. The nodes of each view are connected by the set of directed edges $v.E$. Outgoing edges of XOR-Split and LOOP-Split nodes are annotated with $true$ and $false$. All further properties of the graph representation are equivalent to those discussed in the meta-model as discussed in Section 2: All nodes have an attribute *node.partner* that represents the partner that executes the corresponding step. For each *Send-Step* the target partner is represented with the attribute *node.to*. For receive steps the sending partner is defined by $node.from$. In addition for send-and receive steps the message type $node.messageType$ and the send variable $node.sendVar$ are known. According

to our process partitioning procedure the triple *(sendNode.to, receiveNode.from, messageType)* is always unique. Each abstract step has an attribute for the first partner $node.first$, the last partner $node.last$ and a set of partners in between $node.partners$. Activity-steps and abstract steps have a label defined by $node.label$. Steps with the same label in different views define the same node.

5.1 Observations of the properties of a single view

We will now discuss some properties of a single view V_{px} of some partner px with regard to the merged global process V_{union}.

1. Any edge $(n1, n2) \in V_{px}.E$ where $n1.type = send \wedge n2.type \neq send \wedge (n1.sendsVar \neq n2.decisionVariable$ cannot be part of V_{union}.
2. Any edge $(n1, n2) \in V_{px}.E$ is impossible in V_{union} if $n2.type = abstract \wedge n2.first \neq px$ or $n1.type = abstract \wedge n1.last \neq px$

A send-step is connected to some receive step of another process during merging. If it is additionally connected to some local step this results in unwanted token multiplication in the merged global process. However, this behavior does not happen in the distributed execution of the private process because the processes are synchronized with a succeeding receive step. Token multiplication / parallelism is only allowed for the distribution of decision variables (1). Incoming control-flow to abstract steps can only exist to the first partner, outgoing control-flow can only exist for the last partner of an abstract step. All other incoming and outgoing control flows of abstract steps are only valid in a specific view (2).
Therefore, we first remove all edges that match properties (1) or (2) before we proceed with our merge method. The corresponding algorithm for the preparation of views for merge is shown in Algorithm 4.

5.2 Merging of prepared views

After the views are cleaned based on the observations of section 5.1 we can merge the views. View merging basically operates in three stages: (1) A merged graph V_{union} that contains all the edges and all the nodes of all input-views is created. (2) In a next step matching send- (s) and receive nodes (r) where ($s.to = r.from$ *and* $s.messageType = r.messageType$) are merged. In particular, whenever a match is found send- and- receive nodes are eliminated and the previous node of the send-node is connected to all successors of the send-node as well as to all successors of the receive-node. We also need to preserve the annotations of edges which is relevant for XOR and loops. (3) In a last step all abstract nodes with the same label are merged and all *XOR-Split*, *Loop-Split* and *Loop-Join* nodes are merged, when they have the same previous node. Finnaly, unconnected elements (subgraphs with no connection to the merged graph) will be removed of the resulting graph.

Algorithm 4 Prepare a View for Merge

1: *Procedure prepareForMerge*
Input: View inView
Output: View without globally impossible edges, inView
2: **for all** $((n1, n2) \in inView.E)$ **do**
3:   **if** $(((n1.type = send \wedge n2.type \neq send \wedge n1.sendsVar \neq n2.decisionVariable)))$) **then**
4:     $inView.E = inView.E \setminus \{(n1, n2)\}$
5:   **end if**
6:   **if** ($(n1.type = AbstractStep \wedge inView.partner \neq n1.lastPartner)$ **then**
7:     $inView.E = inView.E \setminus\{(n1, n2)\}$
8:   **end if**
9:   **if** $((n2.type = AbstractStep \wedge inView.partner \neq n2.firstPartner)$ **then**
10:     $inView.E = inView.E \setminus\{(n1, n2)\}$
11:   **end if**
12: **end for**
13: return inView

An algorithm that realized the discussed merging approach is shown in Algorithm 5. It gets a set of prepared views as an input and returns a merged view V_{union}. As a first step of the procedure the union of all prepared views (edges and vertices) is computed and stored in $V_{union}.E$ and $V_{union}.V$ (1). For merging send-receive-pairs an iteration over all matching send (s) and receive (r) nodes is executed if the message types as well as the communicating partners are equal ($s.to = r.from$). In the loop the predecessor of the send $s.prev$ is connected to each successor of send ($getSucceedingNodes(s)$) and each successor of receive ($getSucceedingNodes(r)$) by adding the edges to $V_{union}.E$. As a last step the send and receive nodes are removed from $V_{union}.V$ (2). The second loop iterates over all abstract steps $c1$ and $c2$ and merges them if they have the same label. The helper function $mergeAbstractSteps()$ is responsible for merging the partner assignments (first, last and in between) as well as moving incoming and outgoing edges of $c2$ to $c1$. The last loop iterates over all nodes $n1$ and $n2$ of type *XOR-Split*, *Loop-Split* and *Loop-Join* to merge them. For each pair $(n1, n2)$ another iteration over all predecessors of $n1$ and $n2$ is executed in order to check which nodes must be merged. The are merged if $n1$ is different to $n2$ and they have the same type and the same annotation of incoming edges ($getAnnotation(prev1, n1) = getAnnotation(prev2, n2)$). The helper function $mergeSteps()$ is then used to move incoming and outgoing edges of $n2$ to $n1$ and delete $n2$ from $V_{union}.V$ (3). As a very last step of the algorithm the method $clean()$ is be called to remove remaining fragments of the graph (subgraphs which are not connected to the merged graph) in order to get the final merged global workflow.

Algorithm 5 Merge Views

```
1:  Procedure mergeViews
Input: Set of prepared Views of each partner V = V_p1...V_pn
Output: Merged View, V_union
2:  V_union.V = V_p1.V ∪ V_p2.V... ∪ V_pn.V
3:  V_union.E = V_p1.E ∪ V_p2.E... ∪ V_pn.E
4:  // Remove corresponding send-receive nodes and connect nodes directly
5:  while (∃ (s, r) ∈ {{V_union.V | type = send } × {V_union.V | type = receive } |
    s.MessageType = r.MessageType ∧ s.to = r.from}) do
6:    //Connect all succ. of send- and receive nodes to s.prev and save annotations
7:    an1 = getAnnotation(s.prev,s)
8:    for all (s2 ∈ getSucceedingNodes(s)) do
9:      V_union.E = V_union.E ∪ {(s.prev, s2)}
10:     addAnnotation(getAnnotation(s,s2),s.prev,s2);
11:     addAnnotation(an1,s.prev,s2);
12:   end for
13:   for all ((r2) ∈ getSucceedingNodes(r)) do
14:     ... // Do the same as above for the succeeding nodes of r
15:   end for
16:   V_union.V = V_union.V \ {s, r} // Remove matching send and receive nodes
17: end while
18: // Match and merge abstract nodes
19: for all ((c1, c2) ∈ {V_union.V|type = abstrtact} × {V_union.V|type = abstract}) do
20:   if (c1 ≠ c2 ∧ c1.label = c2.label) then
21:     mergeAbstractSteps(c1, c2)
22:   end if
23: end for
24: // Match and Merge equivalent XOR-Split, LOOP-Split, LOOP-Join steps
25: relevantNodes = {V_union.V|type ∈ {XOR − Split, LOOP − Split, LOOP − Join}
26: for all ((n1, n2) ∈ relevantNodes × relevantNodes) do
27:   for all (prev1, prev2) ∈ getPreviousNodes(n1) × getPreviousNodes(n2)) do
28:     if (n1 ≠ n2∧n1.type = n2.type∧prev1 = prev2∧getAnnotation(prev1, n1) =
        getAnnotation(prev2, n2)) then
29:       mergeSteps(n1, n2)
30:     end if
31:   end for
32: end for
33: V_union = Clean(V_union);
34: return V_union)
```

5.3 Correctness of merged views

The merging method as discussed in the previous section may produce a global process that is not behaviorally equivalent to the distributed execution of the input views. By splitting the input views into pieces during preparation and reconnecting them by matching send- and receive steps the order of steps in the views is not guaranteed to be preserved in the merged result. Additionally, the condition for the predecessor relation between any two steps may be different in

the the merged result and the views. E.g. In some view a step a may be followed by a step b in all cases but in the merged result b is only executed under a specific condition.

Definition 1. A merge-result is correct for the set of input views V, iff:

$\forall\ v \in V : \forall\ (n1,n2) \in prec(V)\ \exists\ (g1,g2) \in prec(V_{union})$ such that $n1 = g1 \wedge n2 = g2 \wedge condPrec(n1, n2, v) = condPrec(g1, g2, V_{union})$
$prec(wf)$ defines the predecessor relation between all activity- or abstract steps in the process wf. $condPrec(n1, n2, wf)$ returns the set of conditions under which $n2$ is a predecessor of $n1$ in the process wf.

An algorithm that checks the correctness of a merged global process with respect to its input views is shown in Algorithm 6. The procedure *correctMerge()* gets a merged graph *mergedGraph* and a set of non-prepared, original views V as an input and returns *true* if the first input is a correct merge of the views, *false* otherwise. The step matrix represents the predecessor relation between all activity- and abstract steps and the conditions under which the predecessor relation between the steps holds. In order to check the correctness of a merge the global step matrix is compared with each view step matrix. We realize the step matrix as a two dimensional array with indexes specified by steps (vertices of the graph) and values containing matrix entries. A matrix entry is a pair of *relation* (in our case we only use the predecessor relations *PRE*) and *condition*. Moreover, a condition is a set of pairs holding variable names and values (*true* or *false*). By using this data structure we can express both the predecessor relationship of each pair of steps as well as the conditions under which the predecessor relationship holds. To achieve the checking of the merged graph two methods are used.
The helper function *createStepMatrix()* is responsible for computing the step matrices. It iterates over all possible pairs of steps *(v1,v2)* that only contain steps of type *Simple* and *Abstract* and computes the matrix entries. A matrix entry is set if *v1* is a predecessor of *v2*. Beside the entry *relation* also the condition under which the predecessor relationship holds is added using the additional helper function *getCondition(v1,v2)*. After iterating through all combinations of steps the matrix is returned.
The next helper function *compareStepMatrices()* is responsible to compare the global process with one of the view matrices. To achieve that an iteration over the global step matrix is used to compare each matrix entry with the corresponding entry in the view matrix. The method returns *false* if one matrix entry is different between the global matrix and the view matrix.

6 Evaluation of the Automatic Partitioning Approach

A partitioning is correct, if the global process and the set of communicating local processes are behaviorally equivalent, i.e. if the distributed execution of the ensemble of communicating process generated by the partitioning algorithm

Algorithm 6 Checking the Correctness of a Merge

```
1: Procedure correctMerge
Input: Merged workflow graph mergedGraph, Set of original view graphs V = Vp1...Vpn
Output: Boolean result
2: StepMatrix stepMatrixMerged = createStepMatrix(mergedGraph)
3: for all (view ∈ V) do
4:    StepMatrix stepMatrixView = createStepMatrix()
5:    if (!compareStepMatrices(stepMatrixMerged, stepMatrixView)) then
6:       return false
7:    end if
8: end for
9: return true
10:
11: Procedure createStepMatrix
Input: WorkflowGraph graph
Output: StepMatrix stepMatrix
12: for all (v ∈ graph.V) do
13:    if ( v.type != Simple and v.type != Abstract ) then
14:       graph.V = graph.V \ {v}
15:    end if
16: end for
17: StepMatrix stepMatrix[][]
18: for all ((v1,v2) ∈ graph.V ) do
19:    if (v1.isPredecessor(v2)) then
20:       Relation relation = PRE
21:       Condition condition = getCondition(v1,v2)
22:       stepMatrix[v1][v2] = (relation,condition)
23:    end if
24: end for
25: return stepMatrix
26:
27: Procedure compareStepMatrices
Input: StepMatrix global, StepMatrix view
Output: Boolean result
28: for all ((g1,g2) ∈ global ) do
29:    if (view.partner ∈ g1.partners and view.partner ∈ g2.partners) then
30:       MatrixEntry viewEntry = getCorrespondingViewEntry(g1.label,g2.label,
          view);
31:       if (global[g1][g2] != viewEntry) then
32:          return false
33:       end if
34:    end if
35: end for
36: return true
```

admits the same traces of activity invocations as a centralized execution of the global process would. A sufficient condition for this behavioral equivalence is

when a correct merging of the set of derived processes is identical to the global process.

Definition 2. Correctness of a Partitioning:

A partitioning $P(p) = \{v_{p1}, ..., v_{pn}\}$ of a process p for n partners is correct, iff: The representation of p as a graph is structurally equivalent to the correct merge of the views in graph representation:
$toGraph(p) \equiv mergeViews(\{toGraph(v_{p1}), ..., toGraph(v_{pn})\}) \wedge$
$correctMerge(mergeViews(\{toGraph(v_{p1}), ..., toGraph(v_{pn})\}), P(p))$.

This narrow definition of equivalence is applicable because we know that our algorithms will not require to apply any equivalence transformations [4] such as changing the structure of the hierarchy before the graphs are merged or compared.

Theorem 1 (Correct Partitioning)**.** Any global process is correctly partitioned into views for k partners using the presented partitioning approach.

Proof 1. We prove the theorem by induction over the nesting depth of the global process. A process W of nesting depth 0 can only be one activity-step and the partitioning is equal to the process and is trivially correct.

We assume that processes of nesting levels up to n are partitioned correctly and show that under this assumption the processes of nesting level $n+1$ are correctly partitioned. A process of nesting level $n+1$ can be a *XOR-, LOOP-, SEQUENCE-, PAR-* Block with a sub-block of nesting level n.

We use abstract steps as place-holders for the sub-blocks. This is possible because they have exactly the same properties as any other block: They are defined by a first partner, a last partners and an optional set of intermediate partners (see Figure 1).

We prove by exhaustive analysis of cases, i.e. for all four types of control steps and all possible assignment configurations of partners to steps that the partitioning is correct in the following way: We generate a global process, partition the process substituting the abstract step with its projections (which are correct according to the induction assumption) and merge the generated views and compare the result with the global process.

In Figure 6 the generic cases for nesting level n+1 for sequences, parallel and loop blocks are shown. Each generic case is annotated with a set of slots #1, ... #n that are placeholders for partners. For example a loop has 5 slots that can be filled with up to 5 different partners. Slots #1,#2,#4,#5 are single valued slots (they must be filled with exactly one partner), while the slot #{3} can be filled with sets of partners including the empty set. The possible partner

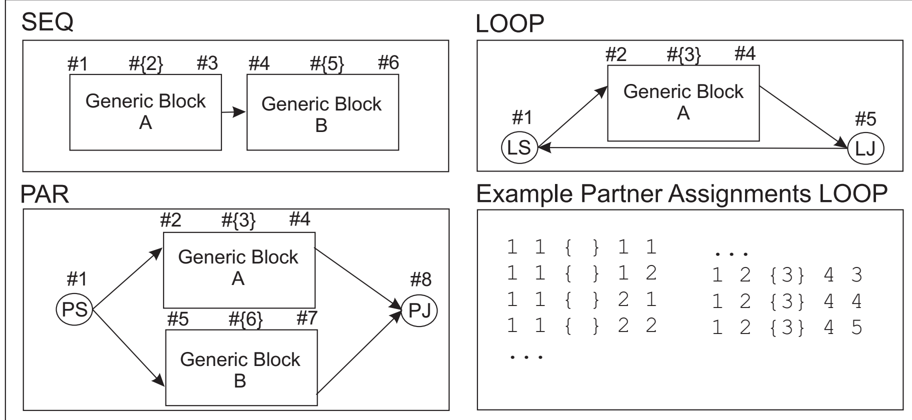

Fig. 6. Generic Cases for Sequence-, Parallel- and Loop- blocks

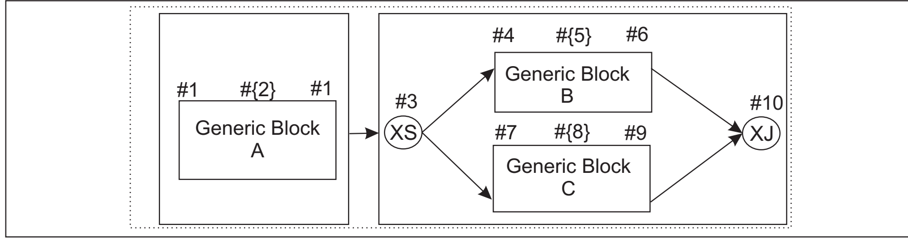

Fig. 7. Generic Cases for $XOR-$Blocks

assignments range from all slots $(1, 1, \{1\}, 1, 1)$ are filled with the same partner to each slot has a different partner $(1, 2, \{3\}, 4, 5)$. We include multi-valued slots with the empty set and any set of partner of size 1. Sets bigger than one do not constitute an additional assignment configuration for the partitioning algorithm and were therefore not tested.

While sequences, $PAR-$ and $LOOP-$ blocks are not context dependent (the partitioning only depends on the current block), the augmentation and projection of $XOR-$blocks is context-dependent. An XOR-block is transformed into a sequence in the view of a partner p and the corresponding decision variable is not sent to p, if p does not take part in the other branch of the XOR-block and if p does not take part in any other block that is not a parent of the XOR block. See line 21 in Alg. 1 and line 13 in Alg. 3. While the condition that a partner does not occur in both branches of a XOR *(partnerInBothBranches(partner,inBlock))* is local to the current block, the condition that a partner only takes part in this block *(partnerOnlyInThisBlock(partner,inBlock))* is dependent on the context. The two possible non local context instantiations are that the partner either takes part in another block or that a partner does not take part in another

block. Both non local context instantiations must be covered by the analysis of $XOR-$blocks. According to the augmentation and projection algorithm it makes no difference in what block type outside of the current XOR-block the partner in question p takes part. Therefore, it is sufficient to analyze all possible cases, when an XOR-block is nested into a sequence of two steps to simulate all possible context instantiations. The case of a XOR nested in a sequence including its 10 slots for partner assignments is shown in Figure 7.

We have implemented all required algorithms and could show that all cases are correctly partitioned by our partitioning algorithm. □

7 Related Work

Workflow View mechanisms typically allow to hide and aggregate elements of a process in order to provide a good balance between the information that needs to be shared for the cooperation and privacy concerns of the partners. Most process view approaches such as [6, 2, 11] allow to define views on private processes which is especially useful for outsourcing or producer/consumer scenarios. In contrast to the aforementioned view approaches we have presented a top-down approach that allows to derive views from a global process. This scenario is also addressed by the p2p approach to interorganizational workflows [19, 20] and more recently with multi party contracts [17]. Both approaches are based on extensions of petrinets and therefore abstract from the data perspective using indeterminism. This is especially problematic, when decisions need to be synchronized between different partners. In contrast to our approach the partitioning procedure presented in [19, 20] does not guarantee that the resulting partitions are valid for any assignments of partners. Instead they define that a partitioning is only correct if all resulting partitions are valid interorganizational workflow nets. This restriction was partly addressed in [17], where open workflow nets are used for modeling. However, a valid partitioning is still restricted: Interfaces places between partners must always be bilateral. This results in problems for possible partitions. For example the outgoing flow of a XOR (a place with two outgoing transitions) cannot relate to different partners. The same holds for XOR joins. Therefore, it is up to the designer to create a global process that can be partitioned correctly based on the requirements. We do not need to impose such restrictions and we operate on deterministic models. We achieve this by an augmentation phase that automatically injects the required communication steps for decisions and control-flow between the partners. This allows us to guarantee that any full-blocked process, can be partitioned automatically based on any arbitrary partner assignment.
Another field of related research is the top-down interaction modeling of choreographies including data (message contents) which is addressed in approaches such as [9, 10]. However, they have a different scope: In interaction modeling the entities of concern for projection are limited to messages while ignoring tasks. In our case not messages but processes are partitioned and the exchanged messages

are generated automatically.
An alternative top-down process partitioning method for global processes was presented in [8]. It translates XOR and LOOP constructs in the global process to deferred choice/deferred loop constructs in the views. This is an elegant solution that minimizes message exchanges in some cases. However, it does not comply with non-local constraints which can result in wrong projections. Another problem is that the approach requires broadcast-messages for the termination of loops. Both problems are solved by our approach.

8 Conclusions

We have presented a novel process partitioning method for the top down development of distributed interorganizational processes. The general idea is that the cooperating partners first agree on a global business process. In a next step each step of the global process is assigned to one of the partners. Finally our partitioning algorithm is used to automatically derive process views of each partner. The views contain the (abstract) tasks that should be realized by the partners and the required control structures and communication steps that specify the distributed execution (choreography) of the global process. In contrast to existing approaches we can guarantee that every full-blocked process can correctly be partitioned for any arbitrary partner assignment and we generate deterministic processes by explicitly addressing the distribution of decision variables. The partitioning algorithm is an essential module in designing P2P processes.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications.* Springer, Berlin, Germany, October 2003.
2. I. Chebbi, S. Dustdar, and S. Tata. The view-based approach to dynamic interorganizational workflow cooperation. *Data Knowl. Eng.*, 56(2):139–173, Feb. 2006.
3. C. Combi and M. Gambini. Flaws in the flow: The weakness of unstructured business process modeling languages dealing with data. In *On the Move to Meaningful Internet Systems: OTM 2009*, volume 5870 of *Lecture Notes in Computer Science*, pages 42–59. Springer, 2009.
4. J. Eder and W. Gruber. A meta model for structured workflows supporting workflow transformations. In *Proceedings of the 6th East European Conference on Advances in Databases and Information Systems*, ADBIS '02, pages 326–339, London, UK, UK, 2002. Springer.
5. J. Eder, N. Kerschbaumer, J. Köpke, H. Pichler, and A. Tahamtan. View-based interorganizational workflows. In *Proc. 12th Int. Conf. Computer Syst. and Tech. (CompSysTech'11)*, pages 1–10. ACM, 2011.
6. R. Eshuis and P. Grefen. Constructing customized process views. *Data Knowl. Eng.*, 64(2):419–438, Feb. 2008.
7. H. Groiss and J. Eder. Workflow systems for inter-organizational business processes. *ACM SIGGroup Bulletin*, 18:23–26, 1997.

8. N. Kerschbaumer. *View-Based Interorganizational Workflows*. Phd-thesis, Alpen Adria Universitaet Klagenfurt, Universitaetsstrasse 65-67, 9020 Klagenfurt, November 2011.
9. D. Knuplesch, R. Pryss, and M. Reichert. Data-aware interaction in distributed and collaborative workflows: Modeling, semantics, correctness. In *CollaborateCom*, pages 223–232. IEEE, 2012.
10. H. Nguyen, P. Poizat, and F. Zaidi. Automatic skeleton generation for data-aware service choreographies. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 320–329, Nov 2013.
11. A. Norta and R. Eshuis. Specification and verification of harmonized business-process collaborations. *Information Systems Frontiers*, 12(4):457–479, Sept. 2010.
12. OASIS. OASIS Web Services Business Process Execution Language (WSBPEL) TC. Technical report, "OASIS", Apr. 2007.
13. C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, October 2003.
14. W. M. P. van der Aalst. Verification of workflow nets. In *ICATPN*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer, 1997.
15. W. M. P. van der Aalst. Process-oriented architectures for electronic commerce and interorganizational workflow. *Information Systems*, 24(8):115–126, 1999.
16. W. M. P. van der Aalst. Inheritance of interorganizational workflows: How to agree to disagree without loosing control? *Information Technology and Management*, 4(4):345–389, 2003.
17. W. M. P. van der Aalst, N. Lohmann, P. Massuthe, C. Stahl, and K. Wolf. Multi-party contracts: Agreeing and implementing interorganizational processes. *Comput. J.*, 53(1):90–106, 2010.
18. W. M. P. van der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, July 2003.
19. W. M. P. van der Aalst and M. Weske. The p2p approach to interorganizational workflows. In *Advanced Information Systems Engineering*, pages 140–156. Springer, 2001.
20. W. M. P. van der Aalst and M. Weske. The p2p approach to interorganizational workflows. In *Seminal Contributions to Information Systems Engineering*, pages 289–305. Springer, 2013.