# Choreography-based Consolidation of Interacting Processes Having Activity-based Loops

Sebastian Wagner[1], Oliver Kopp[1,2] and Frank Leymann[1]
[1]*IAAS, University of Stuttgart, Universitaetsstr. 38, Stuttgart, Germany*
[2]*IPVS, University of Stuttgart, Universitaetsstr. 38, Stuttgart, Germany*
*firstname.lastname@iaas.uni-stuttgart.de*

Keywords: BPEL, Choreography, Process Consolidation, Loops.

Abstract: Choreographies describe the interaction between two or more parties. The interaction behavior description might contain loops. In case two parties want to merge their behavior to gain competitive advantage, the contained loop constructs also have to be merged. This paper presents a language-independent discussion on loop-structure pairing in choreographies and possible merging strategies. Thereby, the focus is turned on loops grouping child activities to be iterated. We show the feasibility of the merging strategies by applying them to BPEL-based choreographies.

## 1 INTRODUCTION

Business process consolidation (also called "process merge") integrates two or more complementing and hence often interacting business processes into a single process. From a business perspective, process consolidation is applied by companies to regain control of outsourced business functions ("business process insourcing"). In the scenario in Fig. 1, for instance, a manufacturer integrates the process of its supplier in its own process. Beside the business perspective, there exist also technical drivers for consolidating processes. Especially in instance-intensive interaction scenarios, where hundreds or thousands of process instances interact with each other, consolidating the interacting processes may lead to significant performance gains (Wagner et al., 2013). They result from avoiding the costly message transfer steps, i. e., the message serialization at the sender side, the actual message transfer and the message deserialization at the receiver side. Since usually complex XML-based protocols such as SOAP are used to exchange messages between processes the message transfer becomes even more resource intensive (Ng et al., 2004). Another advantage of consolidating interacting processes is the decreased number of process instances that have to be managed by the workflow engines. As typically the pay-per-use model is applied in Cloud environments, these performance savings result also in lower costs for enacting a choreography on a workflow engine being hosted in the cloud.

To facilitate process consolidation an approach (Wagner et al., 2012) was developed that automatically consolidates complementing acyclic processes, whose interaction behavior is specified by a choreography, into a single process. The approach ensures that the consolidated process, called $P_{Merged}$ in the following, generates the same set of traces of basic activities as the original choreography. To accomplish that, the approach also adds additional control links to $P_{Merged}$ to relate activities originating from the different processes to be consolidated. So far, the consolidation approach is just capable to merge acyclic processes. If processes are merged that interact via activity-based loops, i. e., loops that contain activities to be iterated in their loop body, the consolidated process $P_{Merged}$ becomes invalid. This is due to the fact, that the additional control links created by the current consolidation approach may cross loop boundaries. However, workflow languages that support activity-based loops, such as BPEL (OASIS, 2007) and BPMN (Object Management Group (OMG), 2011), do not allow control links crossing loops boundaries. For instance, in Fig. 1 the consolidation created an invalid process because the generated control link connects the activities "Syn3" and "Syn4" that are located in different loops.

This work extends the consolidation approach to support the consolidation of processes that interact via activity-based loops. For this purpose, we discuss different interaction patterns involving activity-based loops communicating with other loops (e. g., graph-

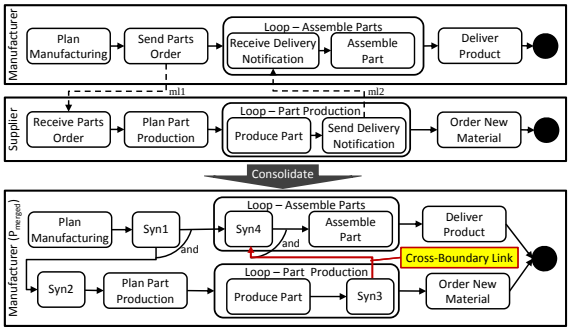


Figure 1: Consolidation of Interacting Processes.

based loops) by means of a language-independent workflow meta-model. For each pattern it is discussed how a consolidation can be performed that keeps the execution order between basic activities that was defined in the original choreography. We developed a tool for consolidating interacting BPEL processes. Therefore, to validate the approach, we will show to what extend the language-independent patterns can be mapped to BPEL in order to implement them in the consolidation tool.

The remainder of the paper is structured as follows. In Sect. 2 we give a brief overview about the process consolidation. In Sect. 3 the meta-model of the workflow language used in this work is defined. Section 4 describes patterns to resolve the cross-boundary violations of activity-based loops. In Sect. 5 the patterns are validated to BPEL and the prototype that implements the patterns is presented. Section 6 gives an overview about the related work and in Sect. 7 the work is concluded.

## 2 PROCESS CONSOLIDATION APPROACH

The actual business functions of processes, e. g., human tasks, data manipulations etc. is implemented by basic activities, i. e., by activities that do not contain other activities. The possible set of execution traces of basic activities during choreography runtime is determined by the control flow constructs (e. g., control links, loops etc.) and interaction patterns defined in the choreography. Thus, for being correct, $P_{Merged}$ must be able to generate the same set of traces of basic activities (without communication activities that are removed during consolidation) during runtime as the original choreography, where $P_{Merged}$ was created from. The same set of traces can be only generated if $P_{Merged}$ keeps the execution orders between the basic activities. The execution order between two basic activities $a_i$ and $a_j$ defines that $a_i$ must be either performed before, after or parallel to $a_j$. To preserve the execution order, the consolidation operation performs the following steps:

At first, a single process named $P_{Merged}$ is created. Then the activities of all interacting processes of the choreography along with their incoming and outgoing control links are copied into $P_{Merged}$. The basic activities are left in their parent activities (e. g., "Produce Part" stays in "Part Production"). This ensures that in $P_{Merged}$ the originally modeled execution order between the activities originating from the same process is preserved.

$P_{Merged}$ still contains communication activities used by the processes to interact with each other. These activities are replaced by synchronization activities that inherit the control links from the communication activities replaced by them. For instance, in Fig. 1 "Send Parts Order" and "Receive Parts Order" are replaced by "Syn1" and "Syn2" respectively. If the data flow of the workflow language is modeled by control flow constructs such as in BPEL, the synchronization activities can be used to emulate the message transfer. For instance by copying the former message content from the data object that was read by the sending activity, to the data object where the message content was copied to by the receiving activity. In BPMN the synchronization activities just act as sources or targets for the materialized control links but they do not perform any operations. In a choreography the execution order between basic activities originating from different processes is implicitly defined by the interaction specification, i. e., by the message links ($ml1$ and $ml2$ in the example). The asynchronous interaction between "Send Parts Order" and "Receive Parts Order" via message link $ml1$ implies that activity "Plan Manufacturing" is always performed before "Plan Part Production". To keep this execution order "control-flow materialization" is performed, i. e., based on the interaction type new control links are created. To replace an asynchronous interaction the link originates at the synchronization activity that replaced the sending activity and ends at the synchronization activity that replaced the receiving activity. These new control links may cause cross-boundary violations, i. e., they cross the boundaries of the activity-based loops, as shown in the example in Fig. 1, which is not permitted in BPMN or BPEL.

## 3 PRELIMINARIES

**Definition 1** (Process)**.** *A process is defined as a directed single entry single exit (SESE) graph* $P = (A, E)$.

*The set A denotes the set of activities and set E denotes the set of control links of the graph. The set of control links is defined as $E \subseteq A \times A \times C$. C denotes the set of link conditions. Conditions are logical expressions that can be evaluated at runtime of P to true or false.*

An activity of a process has a set of incoming control links $E^{\rightarrow}(a) = \{(a_i,a,c) |\ (a,a_i,c) \in E\}$ and a set of outgoing control links $E^{\leftarrow}(a) = \{(a,a_i,c) |\ (a,a_i,c) \in E\}$. The activity where $|E^{\rightarrow}(a)| = 0$ is called "entry activity" $a_{entry}$ of $P$ and the activity where $|E^{\leftarrow}(a)| = 0$ is called "exit activity" $a_{exit}$ of $P$. The set of directly preceding activities of an activity $a$ is denoted by $\bullet a$ and the set of directly succeeding activities of $a$ are denoted by $a\bullet$.

The function $\mathsf{PreDom} : A \rightarrow 2^A$ returns all activities that are (pre-)dominated by activity $a$ and $a$ itself (Koehler et al., 2005). An activity $a$ dominates another activity $b$ if every path from the entry activity to $b$ goes through activity $a$. All activities that are post-dominated by activity $a$ and $a$ itself are returned by the function $\mathsf{PostDom} : A \rightarrow 2^A$ (Koehler et al., 2005). An activity $a$ post-dominates another activity $b$ if every path from $b$ to the exit activity goes through activity $a$.

The control flow of a process model follows the token semantics of BPMN (Object Management Group (OMG), 2011). The entry activity of $P$ propagates a token to each of its outgoing control links. A link that receives a token consumes it and evaluates its link condition. If the link condition evaluates to *true* the link is activated, i. e., it produces a token and passes it to its subsequent target activity. An activity is started (consumes a token) when at least one of its incoming links is activated and no more upstream tokens may reach the activity. Informally, this also holds for OR-joins. Formally, OR-joins have to be Q-enabled to start. Q-enabledness is defined by Völzer (Völzer, 2010). After the activity is completed, one token is passed on to each of its outgoing links. The exit activity just consumes tokens. A process is completed when there are no other upstream tokens.

**Definition 2** (Choreography and Message Links)**.** *A choreography $\mathcal{C} = (\mathcal{P}, \mathcal{ML})$ consists of a set of processes $\mathcal{P}$ and message links $\mathcal{ML} \subseteq A \times A$. Each message link $ML \in \mathcal{ML}$ connects two activities $a_i$ and $a_j$ from different processes $P_1, P_2 \in \mathcal{P}$. In a message link $ML = (a_i, a_j)$ the source activity $a_i$ is the sending activity and the target activity $a_j$ the receiving activity of a message. An activity must be only source or target of exactly one message link. A message link is activated, when $a_i$ is started and $a_j$ cannot complete until the link is activated , i. e., $a_j$ "hangs" until the link is activated. Note, that $a_i$ sends a message m in a send and forget manner, i. e., $a_i$ completes, even if $a_j$ was not started yet. We refer to all activities that are source or target of a message link as communication activities.*

In the following different types of loops are defined that are provided by the most workflow languages (van der Aalst et al., 2003). These loop types are used, to define the patterns for solving cross-boundary violations introduced in 4.

**Definition 3** (Activity-based Loop)**.** *An activity-based loop L is a special type of activity defined as $L = (A_L \subset A, E_L \subset E, c \in C, eval_c = \{\texttt{pre}, \texttt{post}\})$. The loop body is a SESE graph consisting of the activities $A_L$ and the control links $E_L$. No control link must cross the boundary of the loop, i. e., $\forall e \in E_L : \pi_1(e), \pi_2(e) \in A_L$, where $\pi_i$ projects to the ith element of a tuple. $eval_c$ is set to* `pre` *if the loop condition must be evaluated before the first iteration of the loop body (pre-test loop) or set to* `post` *if the loop condition must be evaluated after the first iteration of the loop body (post-test loop). The function* Body*: $L \rightarrow 2^A \times 2^E$ returns the graph in the loop body. The loop condition is returned by function* Cond*: $L \rightarrow C$.*

Activity-based loops can be subdivided in "static activity-based loops" and "dynamic activity-based loops". For static activity-based loops, the maximum possible number of iterations can be determined during design time by using data-flow analysis techniques (Heinze et al., 2012; Kopp et al., 2008). For dynamic activity-based loops, this is not possible at design time but only at runtime. The function Max: $L \rightarrow \mathbb{N} \cup \{\perp\}$ returns the maximum number of iterations of $L$ and returns "$\perp$" in the case of dynamic activity-based loops.

The loop body of an activity can be thought of as a subprocess because it has the same operational semantics as a process. An activity-based loop that is started, passes a single token to its entry activity and the loop completes after all produced tokens were consumed by its exit activity.

**Definition 4** (Graph-based Structured Loop)**.** *A graph-based structured loop $L = (A_L \subseteq A, E_L \subseteq E)$ is a subgraph of P, such that there is an entry node $a \in A_L$ and an exit node $b \in A_L$ (which can be the same), such that every path starting from a visits b and may visit the loop entry a again and thus forms a cycle. Hence, all nodes in $A_L$ are reachable from a.*

In this paper, we focus on structured loops and do not tackle unstructured loops. Finally, we provide a definition for interacting loops.

**Definition 5** (Interacting Loop)**.** *Two loops L1 and L2*

*are called interacting loops, if*

$$\begin{aligned}\exists ml \ &\in \mathcal{ML}: \\ &(\pi_1(ml) \in \Pi_1(\mathsf{Body}(L1)) \\ &\wedge \pi_2(ml) \in \Pi_1(\mathsf{Body}(L2))) \\ \vee &(\pi_2(ml) \in \Pi_1(\mathsf{Body}(L1)) \\ &\wedge \pi_1(ml) \in \Pi_1(\mathsf{Body}(L2)))\end{aligned}$$

*Informally, this means that $L1$ contains a communication activity $a_i$ and that is related to a communication activity $a_j$ in $L2$ via a message link $ml$. Note, that $\Pi_i$ returns the set of ith elements from the given set of tuples, here the activities of the body of $L1$ and $L2$.*

# 4 SOLVING CROSS-BOUNDARY VIOLATIONS IN ACTIVITY-BASED LOOPS

This section describes different patterns to solve cross-boundary violations created in $P_{Merged}$ by the control-flow materialization while keeping the execution order between basic activities. The patterns focus on scenarios with two interacting loops $L1$ and $L2$, however, they can be also applied to more interacting loops, as discussed at the end of this section. One of the loops must be an activity-based loop as cross-boundary violations do only occur if activity-based loops are involved.

In the following, we refer to the set of links that are crossing boundaries of a loop as $E_{CB} \subset E$. The source and target activities of a link $e_{CB} \in E_{CB}$ are referred to as synchronization activities. For interacting graph-based loops $L1$ and $L2$ $P_{Merged}$ does not have to be adapted, as graph-based loops do not have a loop body. Hence, cross-boundary violations cannot occur. The pattern to be applied depends on the type of $L1$ and $L2$ receptively and also on the types of loops supported by the workflow language.

The *context* of a pattern defines, for what types of interacting loops it can be applied. The *solution* describes how the cross-boundary violation can be resolved. *Variations* discusses different variants of the pattern. The *discussion* describes how the pattern preserves the control flow order between the atomic activities that was originally defined in $\mathcal{C}$. The patterns can be only applied to choreographies that are deadlock free.

## Pattern 1: Activity-based Loop Unrolling

**Context.** A static activity-based loop $L1$ is related to another static activity-based loop $L2$ via one or more cross-boundary links as shown in Fig. 2. $L1$ and $L2$ are not forced to have the same number of iterations. The workflow language does not support graph-based structured loops.

**Solution.** As the number of iterations of $L1$ and $L2$ is known at design time *loop unrolling* (also called loop unwinding) can be performed on the two loops to resolve the cross-boundary violations. Algorithm 1 implements loop unrolling for an activity-based loop $L$. In line 3 the first iteration of $L$ is unrolled into $P_{Merged}$. All other iterations of $L$ (if any) are unrolled in line 7.

To perform the actual duplication of the loop body graph the loop unrolling algorithm calls the function Duplicate that is shown in Algorithm 2. The function creates one copy $a'$ of each activity of the given graph $G$ (line 5). The incoming and outgoing links of each original activity $a$ are also duplicated. These link copies become the incoming and outgoing links of the corresponding copy of $a$, i. e., $a'$ (lines 10 to 15). This also includes the cross-boundary links. In lines 4 and 8 Algorithm 1 adds the created activity and link copies to the process graph. As the duplication is performed $n$ times, where $n$ denotes the max. number of iterations of $L$, $n$ subgraphs $G_1, \ldots, G_1$ are created in $P_{Merged}$. Subgraph $G^1_{L1}$ represents the first iteration of $L1$, $G^2_{L1}$ the second iteration, etc. For instance, the example loop $L1$ in Fig. 2 is unrolled into two subgraphs $G^1_{L1}$ and $G^2_{L1}$. Hence, the function Duplicate is called twice by the loop unrolling algorithm. The first call creates the activity copies $a2^1$ - $a5^1$ and second call $a2^2$ - $a5^2$ along with the corresponding link copies.

To preserve the control flow order between the unrolled iterations of $L$, $G_1, \ldots, G_n$ have to be linked sequentially with each other by a new set of $n-1$ control links (lines 11 to 14 in Algorithm 1) . Therefore, each exit activity of the subgraphs $G_1, \ldots, G_{n-1}$ is connected to an entry activity of $G_2, \ldots G_n$ by a new control link $e_{next}$. To emulate the behavior, that another iteration of $L$ is only performed if the loop condition of $L$ evaluates to *true*, the loop condition of $L$ is also added to each link $e_{next}$. To skip all other iterations if the loop condition evaluates to *false* after the execution of an iteration $G_i$, each exit activity of $G_1, \ldots, G_{n-1}$ is linked to all direct successor activities of $L$ via a set of links $e_{skip}$ (lines 11 - 14). This means, that each of these links replaces a link from the set of outgoing links of $L$ ($E^{\leftarrow}(L)$). Note, that each of the links from $E^{\leftarrow}(L)$ may have also a link conditions assigned. Hence, each link $e_{skip}$ must be only activated if the loop condition evaluates to *false* and if the link condition of the link from $E^{\leftarrow}(L)$ it replaces, evaluates to *false*. The last iteration, i. e., $G_n$ is related to the successor activities of $L$ in the same way by calling Algorithm 4.

To ensure that the direct predecessor activities of

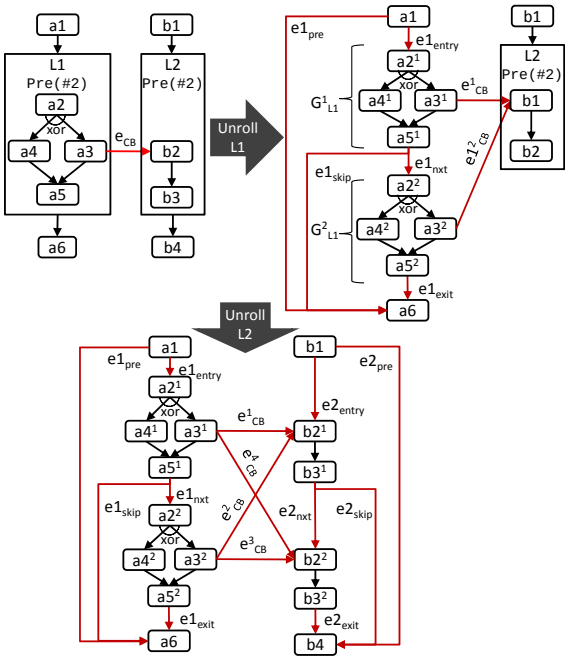


Figure 2: Unrolling of Activity-based Loops.

$L$ become the predecessors of the unrolled iterations of $L$ $G_1,\ldots,G_n$ Algorithm 3 is called. The algorithm links the entry activity of $G_1$ to the direct predecessor activities of $L$. For each former incoming link of $L$ (i. e., $E^{\rightarrow}(L)$) a new entry link $e_{entry}$ is created (lines 4 and line 11 in Algorithm 3). If $L$ is a post-test loop, the link conditions of the entry links inherit the link conditions of the incoming links $E^{\rightarrow}(L)$ (line 11 in Algorithm 3). This ensures the originally modeled behavior, that the first iteration is always started, if at least one of the incoming links of $L$ is activated.

If $L$ is a pre-test loop the first iteration of the loop body must be only started if at least one of the incoming links of $L$ is activated and if the loop condition evaluates to *true*. To emulate this behavior, the link conditions of the entry links are concatenated with the loop condition of $L$. To skip the execution of the unrolled loop if the pre-test loop condition evaluates to *false* another set of links $E_{pre} \subset E$ is created between all direct predecessor and successor activities of $L$

To guarantee that an activity, target of a copy of a cross-boundary link $e^i_{CB}$, is performed at most once a Boolean flag is added to $P_{Merged}$ and accessed by the link condition of all copies $e^i_{CB}$ of a cross-boundary link $e_{CB}$ (due to space reasons not shown in the presented algorithms). This flag carries the name of the original cross-boundary link and it is set from *true* to *false* when a copy $e^i_{CB}$ was activated. Thus, any another copy $e^j_{CB}$ of $e_{CB}$ $(i \neq j)$ cannot be activated anymore, which prevents its target activity from being executed again. For instance, if in Fig. 2 the path $\langle a2^1, a3^1, a5^1\rangle$ is taken, $e^1_{CB}$ is activated and $b2^1$ is executed which causes the flag to be set to *false*. If the execution continues on the path $\langle a2^2, a3^2, a5^2\rangle$ the link condition of $e^2_{CB}$ deactivates this link and prevents $b2^1$ from being started again.

**Algorithm 1:** Loop Unrolling.

1: **procedure** UNROLL-LOOP($L$)
2: $i \leftarrow 1$
3: $G_i \leftarrow$ DUPLICATE(Body($L$))
4: $P_{Merged} \leftarrow P_{Merged} \cup G_i$
5: ADD-TO-LOOP-PREDECESSORS($G_i$,$L$)
6: **while** $i <$ Max($L$) **do**
7: $G_{i+1} \leftarrow$ DUPLICATE($G_i$)
8: $P_{Merged} \leftarrow P_{Merged} \cup G_{i+1}$
9: $e_{next} = (\mathsf{Exit}(G_i), \mathsf{Entry}(G_{i+1}), \mathsf{Cond}(L))$
10: ADD-LINK($P_{Merged}, e_{next}$) ▷ Add link to $P_{Merged}$
11: **for all** $e_{succ} \in E^{\leftarrow}(L)$ **do**
12: $e_{skip} = (\mathsf{Exit}(G_i), \pi_2(e_{succ}), (\pi_3(e_{succ}) \wedge \neg \mathsf{Cond}(L)))$
13: ADD-LINK($P_{Merged}, e_{skip}$)
14: **end for**
15: $i \leftarrow i+1$
16: **end while**
17: RELATE-TO-LOOP-SUCCESSORS($G_{i-1}$,$L$)
18: **end procedure**

**Algorithm 2:** Graph Duplication.

1: **function** DUPLICATE($G$)
2: $A = \Pi_1(G)$ ▷ Original activities
3: $A' = \{\}; E' = \{\}$ ▷ Activity and link copies
4: **for all** $a \in A$ **do**
5: $a' =$ DUPLICATE-ACTIVITY($a$)
6: $A' \leftarrow A' \cup a'$
7: **end for**
8: **for all** $a' \in A'$ **do**
9: $a =$ GET-ORIGIN($a'$) ▷ Get original activity
10: **for all** $e_{in} \in E^{\rightarrow}(a)$ **do**
11: $E' \leftarrow E' \cup$ ADD-LINK($\pi_1(e_{in}), a', \pi_3(e_{in})$)
12: **end for**
13: **for all** $e_{out} \in E^{\leftarrow}(a)$ **do**
14: $E' \leftarrow E' \cup$ ADD-LINK($a', \pi_2(e_{out}), \pi_3(e_{out})$)
15: **end for**
16: **end for**
17: **return** $G' = (A', E')$
18: **end function**

**Variations.** For resolving cross-boundary violations between two interacting static activity-based loops $L1$ and $L2$, both loops have to be unrolled. The order in which $L1$ and $L2$ are unrolled by Algorithm 1 is

not relevant because the loop unrolling technique neither considers nor changes the structure of the other loop. Also if $L1$ and $L2$ have a different number of maximal iterations the unrolling can be performed as the cross-boundary links are duplicated for each iteration. Which, in turn, keeps the control flow relations between each unrolled iteration of $L1$ and $L2$.

A dynamic activity-based loop $L1$ without synchronization activities on alternative paths in its loop body can be unrolled, if it interacts with a static activity-based loop $L2$ that does not have synchronization activities on alternative paths either. The number of unrolled iterations of $L1$ is implied by the maximum number of iterations of $L2$. Each execution of a synchronization activity $a$ from $L1$ connected to a synchronization activity $b$ from $L2$ via a link $e_{cb}(a,b,true)$ results in an execution of $b$ and vice versa. Hence, the maximum number of iterations of $L2$ implies the maximum number of iterations of $L1$.

**Algorithm 3:** Add Loop Predecessors.

```
1:  procedure ADD-TO-LOOP-PREDECESSORS(G, L)
2:    for all e_pred ∈ E^→(L) do
3:      if eval_C(L) = pre then
4:        e_entry = (π1(e_pred), Entry(G),
                (π3(e_pred) ∧ Cond(L) = true))
5:        ADD-LINK(P_Merged, e_entry)
6:        for all e_succ ∈ E^←(L) do
7:          e_pre = (π1(e_pred), π2(e_succ),
                (π3(e_pred) ∧ π3(e_succ) ∧ ¬Cond(L)))
8:          ADD-LINK(P_Merged, e_pre)
9:        end for
10:     else
11:       e_entry = (π1(e_pred), Entry(G), π3(e_pred))
12:       ADD-LINK(P_Merged, e_entry)
13:     end if
14:   end for
15: end procedure
```

**Algorithm 4:** Add Loop Successors.

```
1: procedure ADD-TO-LOOP-SUCCESSORS(G, L)
2:   for all e_succ ∈ E^←(L) do
3:     e_exit = (Exit(G), π2(e_succ),
4:         (π3(e_succ) ∧ ¬Cond(L))
5:     ADD-LINK(P_Merged, e_exit)
6:   end for
7: end procedure
```

**Discussion.** The consecutive execution of the iterations of an unrolled loop $L$ is emulated by duplicating the loop body of $L$ $n$ times to the subgraphs $G_L^1,\ldots,G_L^{n-1}$ and by linking these subgraphs sequentially. The control relations within the activities of the duplicated loop bodies are preserved as no new activities or control links are introduced in $G_L^i$. The behavior, that no further iteration of $L$ is performed when its loop condition evaluates to *false*, is emulated by the set of additional control links $E_{skip}$ connecting the exit activity of each unrolled subgraph $G_L^i$ with the successor activities of $L$.

The loop unrolling technique keeps also the control flow relations between the activities of $L1$ and $L2$ implied by $\mathcal{C}$ as the links $E_{CB}$ are also duplicated. Assume, for instance, that $L1$ in Fig. 2 can be iterated up to six times and $L2$ only up to two times. Hence, $L1$ has to be unrolled six times and $L2$ two times. If $\mathcal{C}$ is deadlock free, the path $\langle a2,a3,a5\rangle$ in $L1$ must be taken exactly in two iterations to execute $b2$ and $b3$. However, it can not be determined at design time, which iterations take this path. The duplication of the cross-boundary link $e_{CB}$ ensures, that taking this path is possible in each unrolled iteration $G_{L1}^i$. At the same time, multiple executions of the same activity are avoided by using the flag that tracks, if a copy of a cross-boundary link was already performed. This also implies that a target activity of one or many cross-boundary links does not run into deadlocks. If it ran into a deadlock, none of the source activities of their incoming cross-boundary link is performed. As each cross-boundary link represents a former message exchange in $\mathcal{C}$, this, in turn, would mean that the target activity would wait forever for an incoming message. Hence, $\mathcal{C}$ would not be deadlock free which contradicts to our prerequisite.

## Pattern 2: Transforming Activity-based Loops to Graph-based Structured Loops

**Context.** A dynamic activity-based loop $L$ is related to another static or dynamic graph-based structured loop via a set of cross-boundary links as shown in Fig. 3. The workflow language supports graph-based structured loops.

**Solution.** To resolve the cross-boundary violations, $L$ can be transformed to a graph-based structured loop $L_G$ by copying the loop body $G_L$ of $L$ to $P_{Merged}$ and by creating a control flow cycle between the exit and entry activity of $G_L$. Algorithm 5 describes the transformation in detail.

The actual duplication of the loop body of $L$ is done in line 2. To realize the repetitive execution of $G_L$ a control flow cycle between the exit activity $a_{exit}$ and the entry activity $a_{entry}$ is created in line 5 by adding the new control link $e_{loop}$. For instance in Fig. 3, the

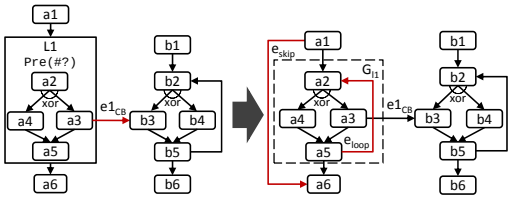


Figure 3: Activity-based Loops to Graph-based Structured Loops.

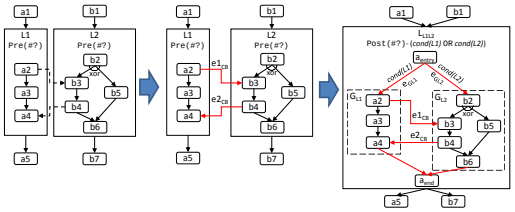


Figure 4: Merging Two Dynamic Activity-based Loops.

link $e_{loop}$ connects the exit activity $a5$ with the entry activity $a2$. To incorporate $G_L$ into the process graph of $P_{Merged}$ Algorithm 5 calls Algorithm 3 and 4.

As described in pattern 1, Algorithm 3 is used to ensure that the set of direct predecessor activities of $L$ become predecessors of the entry activity of $G_L$ by creating a set of entry links (denoted as $e_{entry}$). This means, that if $L$ is a post-test loop, the entry links inherit the link conditions of the original entry links of $L$. If $L$ has a pre-test condition, it has to be guaranteed, that the first iteration of $G_L$ is only performed if the loop condition evaluates to *true*, otherwise $G_L$ must be skipped.

Algorithm 4 is called, to relate the direct successor activities of $L$ with the exit activities $G_L$ by creating for each outgoing link of $L$ ($e \in E^{\leftarrow}(L)$) a corresponding exit link $e_{exit}$. To start the successor activities if and only if all iterations of $G_L$ completed, the link condition of each exit link is concatenated with a negation of the loop condition of $L$.

**Algorithm 5:** Loop Transformation.

```
1: procedure LOOP-TRANSFORM(L)
2:   G_L ← DUPLICATE(Body(L))
3:   P_Merged ← P_Merged ∪ G_L
4:   ADD-TO-LOOP-PREDECESSORS(G_L,L)
5:   e_loop = (Exit(G_L), Entry(G_L), Cond(L))
6:   ADD-LINK(P_Merged, e_loop)
7:   ADD-TO-LOOP-SUCCESSORS(G_L,L)
8: end procedure
```

**Variations.** Transforming a static structured loop $L$ to an unstructured loop instead of unrolling it (as described in Sect. 4), may be useful if $L$ has a high maximum of iterations. This avoids $P_{Merged}$ to be "polluted" with unrolled iterations of $L$.

**Discussion.** Transforming an activity-based loop $L$ to a graph-based structured loop $L_G$ removes the loop boundaries of $L$ while preserving all control flow constraints implied by $\mathcal{C}$. The control link $e_{loop}$ enables iterations of the loop body $G_L$ to be consecutively executed, as long as the loop condition evaluates to *true*. As the loop condition is not changed, $L_G$ is iterated as often as $L$ (under the same data assignment). If the loop condition evaluates to *false*, link $e_{loop}$ is deactivated and the exit links $E^{\leftarrow}(exit)$ are activated. This ensures the original behavior, that the successors of $L$ are started after all iterations of $L$ completed. The original control relations between $L1$ and other loops are also preserved since the cross-boundary links between are not changed either.

## Pattern 3: Merging Two Dynamic Activity-based Loops

**Context.** A dynamic activity-based loop $L1$ is related to another dynamic activity-based loop $L2$ via a set of cross-boundary links as shown in Fig. 4. The workflow language does not support graph-based structured loops.

**Solution.** Loop unrolling cannot be performed as the number of iterations of $L1$ and $L2$ is unknown at design time. To resolve the violations, the source and target activities of these links are moved into the same loop, referred to as $L_{L1L2}$. Thus, the activity graphs $G_{L1}$ and $G_{L2}$ have to be merged into a new single loop $L_{L1L2}$ as shown in Algorithm 6.

The algorithm creates the new loop $L_{L1L2}$ in line 4. The loop body $G_{L1L2}$ of this loop, which consists of of the loop bodies of $L1$ and $L2$, is created in line 5. Additionally, an entry activity $a_{entry}$ and an exit activity $a_{exit}$ is added to the loop body. These activities precede and succeed the entry and exit activities of $G_{L1}$ and $G_{L2}$. They are added to keep the SESE property and to ensure that $G_{L1}$ is only performed, if the loop condition of $L1$ becomes *true* and that $G_{L2}$ is only performed, if the loop condition of $L2$ becomes *true*. For this purpose, the transition condition of the entry link pointing from $a_{entry}$ to $G_{L1}$ gets the loop condition of $L1$ assigned and the the entry link pointing from $a_{entry}$ to $G_{L2}$ gets the join condition of $L2$ assigned. Moreover, the link condition of the entry link is conjoined by an additional flag $firstIt \mapsto \{true, false\}$ if the loop body $G_{L1}$ or $G_{L2}$ originates from a post-test loop (lines 10 and 13). If the first iteration of $L_{L1L2}$ is executed the flag is set to *true*, otherwise it is set to *false*. For instance, in Fig. 4 the entry link for $e_{GL2}$) would be

defined as $(a_{entry}, b2, \mathsf{Cond}(L2) \vee' firstIt = true')$. The value of the flag is hold in the variable *firstIt* that is declared in $P_{Merged}$ (line 8). After the first iteration *firstIt* must be set to *false* (not depicted in Algorithm 6).

$L_{L1L2}$ is always a post-test loop (see below), no matter if $L1$, $L2$ or both are pre-test loops. The loop condition of $L_{L1L2}$ is set to the logical disjunction of the loop conditions of $L1$ and $L2$ (line 16). This ensures that $G_{L1L2}$ is also executed if the condition of $L1$ evaluates to *false* while the condition of $L2$ evaluates to *true* (or vice versa). In Fig. 4, for instance, under a certain data assignment $L1$ may iterate two times and $L2$ five times. To emulate this behavior, $L_{L1L2}$ must be iterated five times which ensured by its loop condition. Since the link condition of the entry link $(a_{start}, a2, \mathsf{Cond}(L1))$ is set to the loop condition of $L1$ its body $G_{L1}$ is just performed twice. $L_{L1L2}$ is wired into $P_{Merged}$ by connecting it to the predecessor and successor activities of $L1$ and $L2$ (lines 17 and 18).

**Algorithm 6:** Merging Activity-based Loops.

```
 1: function MERGE-LOOPS(L1, L2)
 2:   G_L1 ← DUPLICATE(Body(L1))
 3:   G_L2 ← DUPLICATE(Body(L2))
 4:   L_L1L2 = ADD-LOOP(∅, ∅, eval_C ← post)
 5:   Body(L_L1L2) ← G_L1 ∪ G_L2 ∪ ({a_exit},
        {(Exit(G_L1), a_exit, true), (Exit(G_L2), a_exit, true)})
 6:   condEntry_L1 ← Cond(L1)
 7:   condEntry_L2 ← Cond(L2)
 8:   DECLARE(P_Merged, firstIt)          ▷ Adds variable
 9:   if eval_C(L1) = post then
10:     condEntry_L1 ← condEntry_L1 ∨'$firstIt = true'
11:   end if
12:   if eval_C(L2) = post then
13:     condEntry_L2 ← condEntry_L2 ∨'$firstIt = true'
14:   end if
15:   Body(L_L1L2) ← Body(L_L1L2) ∪ ({a_entry},
        {(a_entry, Entry(G_L1), condEntry_L1),
        (a_entry, Entry(G_L2), condEntry_L2)})
16:   Cond(L_L1L2) ← (Cond(L1) ∨ Cond(L2))
17:   E→(L_L1L2) ← E→(L1) ∪ E→(L2)
18:   E←(L_L1L2) ← E←(L1) ∪ E←(L2)
19:   return L_L1L2
20: end function
```

**Variations.** If a dynamic activity-based loop, with synchronization activities on alternative paths in its loop body, is related to a static activity-based loop, its number of iterations cannot be determined from the max. number of iterations of the static loop (refer to pattern 1). Hence, the dynamic loop cannot be unrolled and has to be merged with the static loop in the same way as described in the solution.

**Discussion.** Merging $L1$ and $L2$ into $L_{L1L2}$ keeps the original control flow order between the activities of $G_{L1}$ and $G_{L2}$ as the control links are not changed. The loop condition of $L_{L1L2}$ and the link conditions of the entry links ensure that the same number of iterations of $G_{L1}$ and $G_{L2}$ are executed as in $\mathcal{C}$ (under the same data assignment). However, the activities of $G_{L1}$ and $G_{L2}$ become *iteration-dependent* on each other, i. e., another iteration $i+1$ of the activities in $G_{L1}$ cannot be performed until all activity iterations $i$ in $G_{L2}$ completed (and vice versa). This becomes especially an issue, if the execution times of the activities within $G_{L1}$ and $G_{L2}$ are very different from each other. It also postpones the execution of the successor activities of $L_{L1L2}$. For instance, in 4 $a5$ cannot be started until all iterations of $L_{L1L2}$ completed, i. e., compared to $\mathcal{C}$ its execution is postponed until all iterations of $G_{L1}$ completed. Note, that the postponed execution resulting from the loop merge may increase the time until the business outcome is reached, but it does not affect the overall completion time of $P_{Merged}$ compared to $\mathcal{C}$. The successful completion of all activities of an instance of $P_{Merged}$ takes as long as completing all activities of $\mathcal{C}$ (assuming the same data are used for an instance of $P_{Merged}$ and $\mathcal{C}$).

## Combining the Patterns

In the following two algorithms are proposed that make use of the patterns to solve cross-boundary violations in $P_{Merged}$. If graph-based structured loops are supported by the workflow language, the set of all dynamic or static activity-based loops being source or target of a cross-boundary link (denoted as $\mathcal{L}_{\mathsf{CB}}$) are transformed into graph-based structured loops by Algorithm 7. The transformation preserves all sets of basic activity traces implied by the original loop and it just adds the two new links $e_{skip}$ and $e_{loop}$ to $P_{Merged}$. The runtime of the algorithm is $O(n)$, where $n = |\mathcal{L}_{\mathsf{CB}}|$.

**Algorithm 7:** Transformation to Graph-based Loops.

```
1: procedure SOLVE-CB-VIOLATIONS(L_CB)
2:   for all L_CB ∈ L_CB do
3:     LOOP-TRANSFORM(L_CB)
4:     REMOVE-ACT(P_Merged, L_CB)
5:   end for
6: end procedure
```

If graph-based loops are not supported Algorithm 8 must be applied. It tries to unroll as many loops within $\mathcal{L}_{\mathsf{CB}}$ as possible (pattern 1). All loops that cannot be unrolled, i. e., all interacting dynamic activity-based loops and all static activity-based loops that interact with dynamic activity-based loops, are merged into activity-based loops (pattern 3). Applying pattern 1 decreases the readability of $P_{Merged}$ as it may signif-

icantly increase the number of activities and control links in $P_{Merged}$. Especially if a large number of iterations is unrolled. However, as pattern 3 causes the iteration-dependency issue, pattern 1 is always preferred to pattern 3.

**Algorithm 8:** Merging and Unrolling.

1: **procedure** SOLVE-CB-VIOLATIONS($E_{CB}, \mathcal{L}_{\mathsf{CB}}$)
2: **for all** $L1_{CB} \in \mathcal{L}_{\mathsf{CB}} \mid \text{MAX}(L1_{CB}) = \perp$ **do**
3: $A_L = \Pi_1(\text{BODY}(L1_{CB}))$
4: **for all** $e_{CB} \in E_{CB}$
5: $\mid (\pi_1(e_{CB}) \cup \pi_2(e_{CB})) \cap A_L \neq \emptyset$ **do**
6: $L1_{CB} = \text{PARENT}(\pi_1(e_{CB}))$
7: $L2_{CB} = \text{PARENT}(\pi_2(e_{CB}))$
8: $L_{merged} = \text{MERGE-LOOPS}(L1_{CB}, L2_{CB})$
9: $\text{ADD-ACT}(P_{Merged}, L_{merged})$
10: $\mathcal{L}_{\mathsf{CB}} \leftarrow (\mathcal{L}_{\mathsf{CB}} \cup L_{merged}) - (L1_{CB} \cup L2_{CB})$
11: $A_{L1} = \Pi_1(\text{BODY}(L1_{CB}))$
12: $A_{L2} = \Pi_2(\text{BODY}(L2_{CB}))$
13: $E_{CB}^{L1L2} = \{e_{CB}^{L1L2} \mid \forall e_{CB}^{L1L2} \in E_{CB} :$
$((A_{L1} \cup A_{L2}) \cap \pi_1(e_{CB}^{L1L2})) \neq \emptyset$
$\wedge((A_{L1} \cup A_{L2}) \cap \pi_2(e_{CB}^{L1L2})) \neq \emptyset\}$
14: $E_{CB} \leftarrow E_{CB} - E_{CB}^{L1L2}$
15: $\text{REMOVE-ACT}(P_{Merged}, L1_{CB})$
16: $\text{REMOVE-ACT}(P_{Merged}, L2_{CB})$
17: $L1_{CB} \leftarrow L_{merged}$
18: $A_L = \Pi_1(\text{BODY}(L1_{CB}))$
19: **end for**
20: $\mathcal{L}_{\mathsf{CB}} \leftarrow \mathcal{L}_{\mathsf{CB}} - L_{merged}$
21: **end for**
22: **for all** $e_{CB} \in E_{CB}$ **do**
23: $a_{L1} = \pi_1(e_{CB})$; $a_{L2} = \pi_2(e_{CB})$
24: $L1_{CB} = \text{PARENT}(a_{L1})$
25: $L2_{CB} = \text{PARENT}(a_{L2})$
26: $\text{UNROLL-LOOP}(L1_{CB})$
27: $\text{UNROLL-LOOP}(L2_{CB})$
28: $\mathcal{L}_{\mathsf{CB}} \leftarrow \mathcal{L}_{\mathsf{CB}} - (L1_{CB} \cup L2_{CB})$
29: **end for**
30: **end procedure**

Algorithm 7 is trivial and not further discussed. Algorithm 8 is explained by using the example scenario in Fig. 5. In this scenario $P_{Merged}$ contains four dynamic and two static activity-based loops. The control flow materialization created six cross-boundary links. To resolve the violations the algorithm is called with the parameters $E_{CB}$ and $\mathcal{L}_{\mathsf{CB}}$. Thereby, the set $E_{CB}$ denotes the set of cross-boundary links, here $e1_{CB}$ to $e4_{CB}$. $\mathcal{L}_{\mathsf{CB}}$ contains those loops of $P_{Merged}$ that are source or target of one or more cross-boundary links, i. e., in the scenario $L1$ to $L4$. In a first step, those loops within $\mathcal{L}_{\mathsf{CB}}$ that are related to a dynamic activity-based loop via a cross-boundary link are merged. For this purpose the algorithm selects a dynamic activity-based

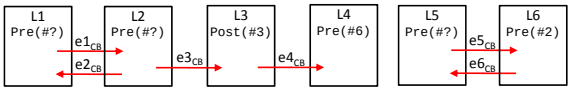

Figure 5: Multiple Interacting Activity-based Loops.

loop from $\mathcal{L}_{\mathsf{CB}}$ (line 4), e. g., $L1$. All loops containing activities that are related to activities within the selected loop via a cross-boundary link (line 4) are pairwise merged with the selected loop. For instance, $L1$ is first merged with $L2$ and the resulting loop $L_{merged}$ is added to $P_{Merged}$ (line 9) while $L2$ and $L1$ are removed from $P_{Merged}$ (lines 15 and 16). As the cross-boundary violations between the merged loops are resolved, all cross-boundary links between them are removed from the set $E_{CB}$ in line 13 (but not from $P_{Merged}$). Hence, after $L1$ and $L2$ were merged $e1_{CB}$ and $e2_{CB}$ are removed. Then all loops that are related to the merged loop via cross-boundary links are merged with $L_{merged}$, i. e., in our example the static loop $L3$ is merged with $L_{merged}$. The new merged loop consisting of the loop bodies of $L1$, $L2$ and $L3$ is then, in turn, merged with $L4$. As this new loop has no cross-boundary links to other loops, it is removed from the set $\mathcal{L}_{\mathsf{CB}}$ (line 20) and another dynamic loop whose activities are source or target of cross-boundary links is selected (if any). In our example this would be $L5$ or $L6$, which are also merged with each other.

After the dynamic loops were merged with other dynamic or static loops, $P_{Merged}$ contains only cross-boundary links between static loops. These loops are unrolled and added to $P_{Merged}$. Our example does only contain static loops that are transitively connected to dynamic loops. Hence, no loop unrolling is performed here. The runtime of the algorithm is also $O(n)$ (where $n = |\mathcal{L}_{\mathsf{CB}}|$), even though it has two nested for-loops. However, the for-loop in line reduces the iterations of its parent for-loop (line ) by removing cross-boundary loops from the set $\mathcal{L}_{\mathsf{CB}}$.

# 5 VALIDATION: CONSOLIDATION OF BPEL PROCESSES WITH INTERACTING LOOPS

To validate the process consolidation approach we developed a tool (Dadashov, 2013) that merges interacting BPEL processes being part of a BPEL4Chor choreography (Decker et al., 2009) into a single process. So far, the prototype was not capable to merge BPEL processes with interacting loops. To support these interaction scenarios, we applied the patterns of Sect. 4 to BPEL and extended the prototype ac-

cordingly. The prototype gets a ZIP file as input, that contains an XML representation of BPEL4Chor choreography along with the processes to be merged. To perform the consolidation, the prototype creates $P_{Merged}$ and copies all activities of the processes within the BPEL4Chor choreography into $P_{Merged}$. Based on the message links and the communication activities in the BPEL4Chor choreography the prototype performs the control flow materialization.

## 5.1 Control Flow Semantic of BPEL

Besides control links BPEL uses join conditions to determine if an activity can be started. A join condition specifies which links have to be activated to start this activity. This requires the status of all links to be known before the join condition is evaluated. Moreover, BPEL is using a dead path elimination control flow semantics (DPE) that determines all activities in the control flow that cannot be executed anymore. Due to the DPE semantics and the fact that all links of an activity have to be evaluated before it can be started, BPEL does not support graph-based loops.

To enable repetitive execution of activities, BPEL offers three types of activity-based loops, the pre-test loops `while` and `forEach` and the post-test loop `repeatUntil`. `while` and `repeatUntil` loops are defined in the same way as the activity-based loops defined in Sect. 3, i. e., they consist of a Boolean loop condition and an activity graph in their loop body. For simplicity reasons, we assume that the loop body is a SESE graph, even though this not stipulated by BPEL . The `forEach` activity has no Boolean expression as loop condition but a From and To attribute, representing the start and end value of the iteration counter. The attribute Counter provides the name of the counter variable that is increased by one in each iteration. The attribute values can be determined at runtime but they must be constant during the execution of the `forEach`. All iterations have to be executed in order to complete the `forEach` loop. The `completionCondition` attribute for modeling at-least-n-out-of-m semantics is not considered here.

## 5.2 Applying the Patterns to BPEL

This section describes how patterns 1 and 3 from Sect. 4 are applied to BPEL. Pattern 2 is not further considered here as BPEL does not support graph-based structured loops.

**Activity-based Loop Unrolling.** To determine the maximum iterations of a BPEL loop the data flow analysis techniques introduced by Heinze et al. (Heinze et al., 2012) are used. However, the pattern cannot be applied to arbitrary static activity-based loops. Because BPEL's link semantic requires that all incoming links of an activity are evaluated before it is started. However, if a loop is unrolled also the cross-boundary links $E_{CB}$ are duplicated. This results in $n$ multiple copies of $e_{CB}$ targeting the same activity, where each copy has its source in one of the unrolled iterations $G_L^1, \ldots, G_L^n$ of the loop $L$. Hence, the source activity of each copy of $e_{CB}$ must be performed before the target activity of $e_{CB}$ can be executed. In Fig. 2, for instance, activity $b2^1$ is not started until $a3^1$ and $a3^2$ completed. In the example, this just leads to a postponed execution of $b2^1$ compared to $C$ where the first iteration of $b2$ can complete after the first iteration of $a3$ is performed (if we assume that messages are instantly delivered). If $L2$ would contain an activity that is source of a link $e2_{CB}$ targeting an activity within the unrolled loop $L1$, this would even lead to a deadlock.

This issue is circumvented by ensuring that the activities $A_{L1}$ and $A_{L2}$ in the unrolled loops $G_{L1}$ and $G_{L2}$ have at most one incoming cross-boundary link $e^i_{CB}$:

$$\forall a \in A_{L1} \cup A_{L2} : |(E^{\rightarrow}(a) \cup E^{\leftarrow}(a)) \cap E_{CB}| \leq 1$$

This, in turn, requires the source and target activities of $E_{CB}$ to be performed during each iteration. Thus, there must be no potential alternative paths in $L1$ or $L2$ preventing synchronization activities from being executed during an iteration of $L1$ or $L2$:

$$\forall a_{L1} \in A_{L1} \mid (E^{\rightarrow}(a_{L1}) \cup E^{\leftarrow}(a_{L1})) \in E_{CB} : \mathsf{PreDom}(a_{L1}) \cup \mathsf{PostDom}(a_{L1}) = A_{L1}$$

$$\forall a_{L2} \in A_{L2} \mid (E^{\rightarrow}(a_{L2}) \cup E^{\leftarrow}(a_{L2})) \in E_{CB} : \mathsf{PreDom}(a_{L2}) \cup \mathsf{PostDom}(a_{L2}) = A_{L2}$$

If the aforementioned prerequisite is fulfilled, a copy of a cross-boundary link has to be connected to a source and target activity part of the subgraph $G^i_{L1}$ and $G^i_{L2}$ in the same iteration, i. e., the source and target activities must dominate the same number of sync. activities $A_{syn}$:

$$\forall e^i_{CB} \in E_{CB} : |\mathsf{PreDom}(\pi_1(e^i_{CB})) \cap A_{CB}| = |\mathsf{PreDom}(\pi_2(e^i_{CB})) \cap A_{CB}|$$

$$A_{CB} = \bigcup_{e \in E_{CB}} \pi_1(e) \cup \pi_2(e)$$

The set $A_{CB}$ denotes the set of activities of $L1$ and $L2$ having an incoming or outgoing cross-boundary link.

Figure 6 shows a `while` loop $L1$ ($\mathsf{Max}(L1) = 2$) interacting with a `repeatUntil` loop $L2$ ($\mathsf{Max}(L2) = 2$) that meet the aforementioned properties (in the figure

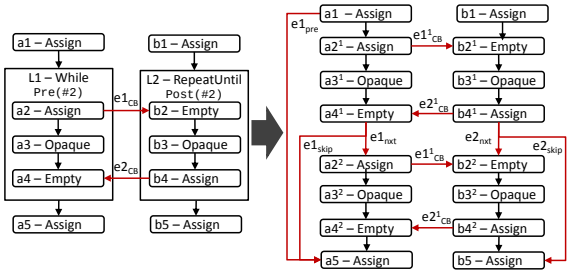


Figure 6: Unrolling Two Static BPEL Loops.

the `opaque` activities encapsulate some business logic). As there is no alternative flow in $L1$ and $L2$ they have to exchange messages during each iteration. This also implies that they always perform the same number of iterations. Hence, the copies of the cross-boundary links ($e1^1_{CB}$, $e1^2_{CB}$ and $e2^1_{CB}$, $e2^1_{CB}$) between the unrolled loop $G_{L1}$ and $G_{L2}$ must only connect activities from the same iterations. As shown in Figure 6, `while` or `repeatUntil` loops are unrolled in the same way as described in the pattern, i. e., the loop bodies are unrolled and added to the container activity of $L1$ and $L2$ (not depicted here). The set of links $E_{nxt}$, $E_{skip}$, $E_{entry}$, $E_{exit}$ connects the unrolled iterations with each other and with the direct predecessor and successor activities of $L$. The link $e_{pre}$ has to be only added for the pre-test loops `while` or `forEach`. In contrast to the pattern, for an unrolled `forEach` loop no loop condition is assigned to $e_{nxt}$ as a sequential `forEach` cannot be interrupted, i. e., the maximum number of iterations is always performed. Thus, the set of links $E_{skip}$ is not required either.

**Merging Two Dynamic Activity-based Loops.** If the interacting loops $L1$ and $L2$ are `while` loops the pattern can be directly applied. Then $L_{L1L2}$ is also a `while` loop whose loop condition is the disjunction of the loop conditions of $L1$ and $L2$. The loop body of $L_{L1L2}$ is created as described in the pattern. If one of the entry links of $G_{L1}$ or $G_{L2}$ evaluates to $false$, DPE ensures that the activities within $G_{L1}$ or $G_{L2}$ are not performed.

If one of the loops $L1$ or $L2$ is a `repeatUntil` loop the variation of the pattern for post-test loops has to be applied, i. e., $L_{L1L2}$ must be also a `repeatUntil` loop.

If a `forEach` loop interacts with a `while` loop the merged loop $L_{L1L2}$ must be a `while` loop because the loop condition of a `forEach` loop cannot specify complex logical expressions such as disjunctions. To specify the loop condition of $L_{L1L2}$, the interval defined by the From and To attributes of the `forEach` is transformed to a logical expression on the counter variable. In a `forEach` the counter variable is automatically increased with each iteration. This has to emulated in

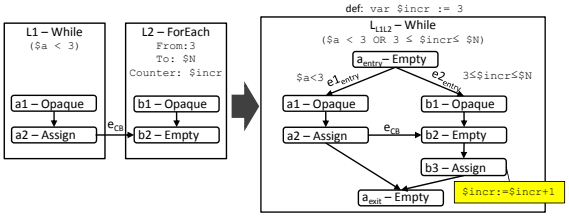


Figure 7: Merging While Interacting with ForEach.

$L_{L1L2}$ by defining the counter variable before $L_{L1L2}$ is started and by initializing it with the value of the From attribute. The counter can be increased by using an `assign` activity that increments the counter when $L_{L1L2}$ completes. Figure 7 shows how the `while` loop $L1$ and the `forEach` loop $L2$ are merged into $L_{L1L2}$. The counter variable *incr* is initialized with the value 3 from the From attribute. The loop condition of the `forEach` is transformed to the expression $3 \leq \$incr \leq N$ and forms a disjunction with the loop condition of $L1$. `assign` $b3$ increments the counter variable *incr* at the end of the $L_{L1L2}$. As the name of the counter variable is kept the activities of the loop body of the former `forEach` accessing the counter variable do not have to be adapted. Analogously, $L_{L1L2}$ must be a `repeatUntil` if the `forEach` interacts with a `repeatUntil` loop.

For two interacting `forEach` loops $L1$ and $L2$ the values of the From and To attributes may be unknown at design time and the name of the counter variables used may be different. Hence, they cannot be merged into a `forEach` loop as only one counter variable can be declared there. Instead they can be also merged into a `while`. Thereby, two counter variables have to be defined before $L_{L1L2}$, one for counting the iterations of $G_{L1}$ and another one for counting the iterations of $G_{L2}$. In BPEL, a single `assign` activity can perform multiple assignments, i. e., an `assign` following $G_{L1}$ and $G_{L2}$ can iterate both counter variables.

# 6 RELATED WORK

Existing approaches focus on merging semantically equivalent processes, which is different from our approach that merges *complementing* processes into a single process. For instance, Küster et al. (Küster et al., 2008) discuss how different variants of the same original process can be merged into a single process by employing change logs. Mendling and Simon (Mendling and Simon, 2006) describe an approach for merging Event Driven Process Chains (EPC) (Scheer et al., 2005) where semantically equivalent elements of an EPC have to be defined manually and based on this semantic mapping, the EPCs are merged.

Loop unrolling and merging is widely discussed in the area of compiler theory (Muchnick, 1997), especially, for optimizing parallel or embedded systems (Qian et al., 2002). Similar to our approach, in these works also loop conditions have to be combined and data analysis has to be performed to determine if a loop can be unrolled or merged (Darte, 1999). In contrast to our approach loop unrolling and merging in this context is employed for optimizing the runtime of short running programs in embedded systems and the language constraints of a programming language are different from those of a workflow language.

Kiepuszewski et al. (Kiepuszewski et al., 2013) also discuss activity and graph-based loops. However, in their work they focus on the transformation of graph-based loops into activity-based loops to structure unstructured workflows.

We investigated how cross-boundary link violations can be solved in parallel BPEL `forEach` loops (Wagner et al., 2014). But there we neither considered sequential `forEach` activities nor other BPEL loop types and the approach described there focuses only on BPEL workflows.

# 7 CONCLUSION AND OUTLOOK

In this work we extended the existing approach to support the automatic consolidation of processes interacting via activity-based loops. The focus was turned on activity-based loops, as the boundaries of these loops must not be crossed by the control links created by the control-flow materialization. To be universally applicable, the patterns for merging the loops were described independently of a concrete workflow language and different types of loops were considered that might be supported by a workflow language. All merge patterns keep the originally modeled execution order between the basic activities of the processes to be merged. However, when two dynamic activity-based loops are merged (pattern 3), additional control flow constraints are implicitly added to the merged process. The new constraints adhere to execution order defined the choreography but they may increase the time until a business outcome is reached. Hence, if the workflow language supports graph-based loops, interacting dynamic activity-based loops should be always transformed to graph-based structured loops. This also prevents the resulting process from getting too complicated in terms of number of activities and links (e. g., if loop unrolling is performed).

We also discussed the applicability of the patterns to the executable workflow language BPEL. As BPEL has a different control flow semantics compared to the workflow meta-model, we used to describe the patterns, the loop unrolling pattern had to be restricted in order to avoid deadlocks in BPEL processes.

In future works we have to investigate how to solve the link violations for activity-based loops interacting with activities in an acyclic graph. We also have to discuss how nested activity-based loops have to be treated. This includes the description of a formal algorithm that applies the patterns to activity-based loops interacting with several (nested) loops.

# ACKNOWLEDGEMENTS

This work was partially funded by the BMBF project ECHO (01XZ13023G) and the BMWi project NEMAR (03ET40188).

# REFERENCES

Dadashov, E. (2013). Choreography-based Business Process Consolidation in One-To-Many interactions. Master thesis, University of Stuttgart.

Darte, A. (1999). On the complexity of loop fusion. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pages 149–157.

Decker, G., Kopp, O., Leymann, F., and Weske, M. (2009). Interacting services: From specification to execution. *Data & Knowledge Engineering*, 68(10):946–972.

Heinze, T., Amme, W., and Moser, S. (2012). Control flow unfolding of workflow graphs using predicate analysis and SMT solving. In *ZEUS*.

Kiepuszewski, B., ter Hofstede, A. H. M., and Bussler, C. (2013). On structured workflow modelling. In *Seminal Contributions to Information Systems Engineering*, pages 241–255.

Koehler, J., Hauser, R., Sendall, S., and Wahler, M. (2005). Declarative techniques for model-driven business process integration. *IBM Systems Journal*, 44(1):47–65.

Kopp, O., Khalaf, R., and Leymann, F. (2008). Deriving Explicit Data Links in WS-BPEL Processes. In *IEEE International Conference on Services Computing*. IEEE.

Küster, J., Gerth, C., Förster, A., and Engels, G. (2008). A tool for process merging in business-driven development. In *Proceedings of the Forum at the CAiSE*.

Mendling, J. and Simon, C. (2006). Business process design by view integration. In *BPM Workshops*. Springer.

Muchnick, S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

Ng, A., Chen, S., and Greenfield, P. (2004). An Evaluation of Contemporary Commercial SOAP Implementations. In *AWSA*.

OASIS (2007). *Web Services Business Process Execution Language Version 2.0 – OASIS Standard*.

Object Management Group (OMG) (2011). *Business Process Model and Notation (BPMN) Version 2.0*. OMG Document Number: formal/2011-01-03.

Qian, Y., Carr, S., and Sweany, P. H. (2002). Loop fusion for clustered vliw architectures. In *LCTES-SCOPES*, pages 112–119.

Scheer, A.-W., Thomas, O., and Adam, O. (2005). *Process Aware Information Systems*, chapter Process Modeling Using Event-Driven Process Chains. Wiley-Interscience.

van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., and Barros, A. P. (2003). Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51.

Völzer, H. (2010). A new semantics for the inclusive converging gateway in safe processes. In *BPM 2010*.

Wagner, S., Kopp, O., and Leymann, F. (2012). Towards Verification of Process Merge Patterns with Allen's Interval Algebra. In *ZEUS*, Bamberg. CEUR.

Wagner, S., Kopp, O., and Leymann, F. (2014). Choreography-based Consolidation of Multi-Instance BPEL Processes. In *CLOSER*. SciTePress.

Wagner, S., Roller, D., Kopp, O., Unger, T., and Leymann, F. (2013). Performance optimizations for interacting business processes. In *IC2E*. IEEE.