Institute of Architecture of Application Systems

Choreography-based Consolidation of Multi-instance BPEL Processes

Sebastian Wagner[1], Oliver Kopp[2], Frank Leymann[1]

[1]IAAS, University of Stuttgart, Germany,
lastname@iaas.uni-stuttgart.de
[2]IPVS, University of Stuttgart, Stuttgart, Germany
oliver.kopp@informatik.uni-Stuttgart.de

BibTeX:


```
@inproceedings {INPROC-2014-17,
    author    = {Sebastian Wagner and Oliver Kopp and Frank Leymann},
    title     = {Choreography-based Consolidation of Multi-Instance BPEL
                 Processes},
    booktitle = {Proceedings of the 4th International Conference on Cloud
                 Computing and Services Science (CLOSER 2014);Barcelona, Spain,
                 April 3-5, 2014.},
    publisher = {SciTePress},
    pages     = {287-298},
    month     = {April},
    year      = {2014},
    keywords  = {BPEL; Choreography; Process Consolidation; Multi-Instance
                 Interactions}
 }
```

These publication and contributions were presented at CLOSER 2014
CLOSER 2014 Web site: http://closer.scitevents.org

Universität Stuttgart
Germany

Choreography-based Consolidation of Multi-Instance BPEL Processes

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*
{*wagnerse, kopp, leymann*}*@iaas.uni-stuttgart.de*

Keywords: BPEL, Choreography, Process Consolidation, Multi-Instance Interactions

Abstract: Interaction behavior between processes of different organizational units such as an enterprise and its suppliers can be modeled by choreographies. When organizations decide, for instance, to gain more control about their suppliers to minimize transaction costs, they may decide to insource these companies. This especially includes the integration of the partner processes into the organization's processes. Existing works are able to merge single-instance BPEL process interactions where each process model is only instantiated once during choreography execution. However, there exist different interaction scenarios where one process interacts with *several* instances of another process and where the number of instances involved is not known at design time but determined during runtime of the choreography. In this work we investigate these interaction scenarios and extend the process consolidation approach in a way that we can emulate the multi-instance interaction scenarios in the merged process model.

1 INTRODUCTION

To enable collaboration between companies, their processes have to interact with each other. The required interaction behavior can be specified with interconnection choreographies, where the communicating activities of the interacting business processes are connected by message links (Decker et al., 2008). In Figure 1 an interconnection choreography is shown where a travel agency queries a set of different airlines to check flight availability and price for the dates specified by a traveler (the traveler is not depicted in Figure 1). Then it selects the cheapest airline that has a flight available and orders a flight for the traveler.

In previous work (Wagner et al., 2011), a process consolidation approach has been proposed that merges all or a subset of complementing BPEL processes (OASIS, 2007) belonging to the same choreography into a new single process model. These processes can be either abstract or executable.

One motivation for the consolidation of interacting processes is that we want to reverse the fragmentation of BPEL processes proposed by Khalaf and Leymann (Khalaf and Leymann, 2006; Khalaf and Leymann, 2010). This fragmentation approach splits single BPEL processes into several interacting BPEL process fragments that keep the operational semantics of the original process. The consolidation of interacting process models can also lead to significant performance gains as the message transfers between processes are avoided and since the number of instances is decreased. Performance measurements conducted by Wagner et al. (Wagner et al., 2013b) have shown that the CPU load, required for executing a consolidated process that was created from a choreography con-

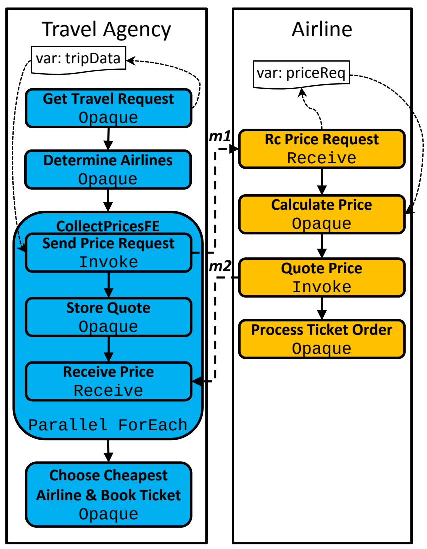

Figure 1: Example Choreography (adapted from (Decker et al., 2008))

sisting of 5 interacting processes, is reduced by 80% (compared to the CPU load required for executing the original choreography). As cloud providers charge for the resources used by their customers (pay-per-use) these performance gains do also lead to reduced process execution costs when the processes are enacted in cloud environments. A more technical reason for consolidating interacting BPEL processes is that available BPEL workflow engines are just capable to deploy and execute business processes but cannot enact choreographies. After a choreography was consolidated into a single process, it can be executed on these workflow engines. So far, process consolidation has been investigated just for one-to-one interactions (Barros et al., 2005) (Wagner et al., 2013a), where for each process participating at the choreography just one instance is created during choreography execution. However, the travel agency interacts with a set of multiple instances of the airline process (one-to-many interaction). The set of airlines involved is unknown during the design time of the choreography as it is determined by the travel agency during runtime.

In this work, we will extend the process consolidation approach to emulate these one-to-many multi-instance interaction scenarios in the merged process model P_{Merged} without introducing new BPEL language constructs or additional middleware. We will also show how multi-instance interactions where the number of instances is not known at design time can be emulated by the consolidated process P_{Merged}. Thereby, the sequential relations between the business activities A_{BA} in P_{Merged} must be the same as the sequential relations between business activities of the original choreography $\mathcal{C}$. The set of business activities A_{BA} includes all activities that implement a certain business function. In Figure 1 business activities are marked with the label `opaque`. Preserving the sequential relation between the business activities ensures that the originally modeled control flow of activities performing the actual business logic in $\mathcal{C}$ is approximated as far as possible in P_{Merged} and that at least all control flow constraints between these activities are kept in P_{Merged}.

The consolidation approach described in this and previous work focuses on BPEL as workflow language due to the following reasons: BPEL is still the de facto language for executable workflows (Leymann, 2010) and widely used in industry. BPEL provides a well-defined operational semantics and a variety of theoretical models are available for this language (van Breugel and Koshkina, 2006). In contrast to BPEL, BPMN (Object Management Group (OMG), 2011) is still underspecified and contains ambiguities (Kossak et al., 2012; Börger, 2012; Wohed et al., 2006). The well-defined operational semantics of BPEL is required to create a consolidated and portable process model that can be executed on any BPEL workflow engine. Another reason why we are focusing on BPEL is that, in the long run, we want to be able to reverse the fragmentation of processes described by Khalaf et al. (Khalaf and Leymann, 2006; Khalaf and Leymann, 2010), where they focus on the fragmentation of executable BPEL processes.

The remainder of this paper is structured as follows. Section 2 provides an overview on BPEL, the choreography language BPEL4Chor, and the consolidation of one-to-one interactions. The control flow properties of multi-instance interactions are discussed in Section 3. We suggest the consolidation approach for multi-instance interactions in Section 4. The results of the consolidation approach are discussed in Section 5 and the prototype implementing the approach is presented in Section 6. Related work is presented in Section 7 before Section 8 concludes this work and gives an outlook about future work.

2 PRELIMINARIES

Firstly, this section provides an overview about the choreography language BPEL4Chor and the language elements of BPEL relevant for realizing multi-instance interactions and their consolidation (Section 2.1). Then an overview about the process consolidation approach for one-to-one interactions is given (Section 2.2).

2.1 BPEL AND BPEL4CHOR

BPEL is a workflow language designed for enabling programming in the large (DeRemer and Kron, 1976) based on Web services. BPEL itself does not offer a capturing of multiple processes interacting with each other (Decker et al., 2009). This is enabled by BPEL4Chor (Decker et al., 2009), which provides a participant topology listing of all participants of a choreography and a list of message links linking communication activities of the participant behavior descriptions. As language for a participant behavior description, BPEL is used. BPEL enables describing the behavior of a participant without revealing the processes internal behavior. Thereby, the `opaque` activity allows to hide such behavior. It is replaced by actual business logic when the BPEL process is deployed and running (Aalst et al., 2008). A partner process is addressed by using correlation tokens or by an endpoint reference (Barros et al., 2007). In BPEL4Chor, a `forEach` activity may iterate on a set of participant references to enable a parallel interaction with multiple process instances of the same type. The `forEach` activity iterates its child activity exactly N+1 times, where N equals the `finalCounterValue` minus the `startCounterValue` (OASIS, 2007). The flag `parallel` can be used to specify whether this iteration should happen in parallel or sequentially. A `completionCondition` may be used within the `forEach` to allow the `forEach` activity to complete without executing or finishing all the branches specified: Remaining branches are terminated when the `completionCondition` evaluates to true. Although BPEL allows graph-based modeling with links connecting the activities (Kopp et al., 2009), it has the constraint, that links may not cross the boundary of a `forEach` activity (OASIS, 2007, SA00071). This constraint will get important when translating message links into control flow links.

We define as business activities those activities that perform the actual business logic. These activities interact with services outside the choreography, perform data manipulations, implement user interactions etc. Business activities do not contribute directly on intra-choreography communication (i. e., they are neither source nor target of a message link). Moreover, a business activity must not have any visible child activities (e. g., loops are no business activities). If a set of activities is indexed, it denotes the set the business activities of the indexed participant. For instance, $A_{BA,i}$ denotes the set of all business activities of participant i.

2.2 CONSOLIDATION OVERVIEW

In this section, we summarize the idea of the process consolidation by Wagner et al. (Wagner et al., 2011). The consolidation operation merges a set of n interacting processes of a choreography $\mathcal{C}$ into a single process P_{Merged}.

The consolidation has to keep the explicitly modeled control flow constraints between business activities of the same process, i. e., $\{(a,b) \mid a,b \in A_{BA,i}\}$ and the implicit control flow constraints between activities that originate from different processes, i. e., $\{(a,b) \mid a \in A_{BA,i}, b \in A_{BA,j}$, where $i \neq j\}$. Explicit control flow constraints are imposed by the BPEL control flow constructs such as control links between activities and structured activities (e. g., loops). Implicit control flow constraints are imposed by the interaction patterns between the processes that have to be merged. The initial asynchronous interaction between the "Travel Agency" and "Airline" implies for instance that the `receive` activity "Rcv Price Request" and its successor activities have to be started after the `invoke` activity "Send Price Request" completed. There exists, for instance, no implicit control flow constraint between the activity "Store Quote" and "Calculate Price" as "Calculate Price" may be started or completed even before "Store Quote" was started or completed. This may happen if the activity "Store Quote" is long running. The synchronous interaction, in turn, additionally implies that the successor activities of a sending synchronous `invoke` activity are not started until it received a response from the partner where it has sent a message to before (synchronous interactions are not depicted in Figure 1). Hence, if "Quote Price" would send a synchronous response to the "Send Price Request" activity, "Store Quote" would be executed after "Calculate Price" completed. To capture these implicit control flow constraints, the consolidation operation *materializes* them into explicit control flow relations. An example for control flow materialization for an asynchronous interaction is given in Figure 2. To derive the control flow from asynchronous interactions, the sending activity is replaced by a synchronization activity "syn_s" and the receiving activity is replaced by a second synchronization activity "syn_{rc}". The synchronization activity "syn_s" is an `assign` activity that emulates the former message transfer between the send/receive activity, i. e., it copies the former message payload data to the variable where the message payload was written to before. Between "syn_s" and "syn_{rc}" a control link is created that ensures that "syn_{rc}" and its successors are not started before "syn_s" has been completed. The incoming and outgoing links of the sending and receiving activity are mapped to the synchronization activities "syn_s" and "syn_{rc}" respectively. In our scenario, "Send Price Request" is for instance

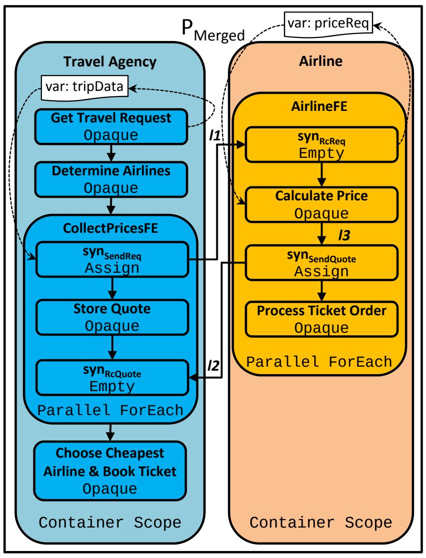

Figure 2: Consolidated Process Model Created from Example Choreography

replaced by "$syn_{SendReq}$" and "Rc Price Request" by "syn_{RcReq}". The latter activity could be also removed as it has no incoming links and the control link could point directly from "$syn_{SendReq}$" to "Calculate Price". A more formal description of deriving the control flow from interaction patterns is described by Wagner et al. (Wagner et al., 2012b).

After the basic concepts of control flow materialization were explained, we summarize the different steps of the consolidation for one-to-one interactions. Deviations from these steps in multi-instance interactions will be discussed in Section 4.

(i) *Creation of Merged Process Model P_{Merged}:* The new process P_{Merged} is created that acts as container for the set of activities. All activities of all processes to be merged are put into P_{Merged}.

(ii) *Creating Participant Containers:* The activities $A_1, \ldots, A_n$ have to be added to P_{Merged} in an isolated fashion as they were also isolated from each other in $\mathcal{C}$. This preserves the control flow of $\mathcal{C}$ since uncaught faults in A_i do not cause $A_j (i \neq j)$ to fail. Assume, for instance, that the "Travel Agency" and "Airline" process were merged into a single process. If activity "Store Quote" is running and "Calculate Price" causes an uncaught fault, P_{Merged} is terminated. Thus, also the activity of "Store Quote" fails. This would cause an invalid execution trace during the execution of $\mathcal{C}$ as the activity "Store Quote" can still be performed even though "Travel Agency" might have crashed. Hence, containers have to be created to isolate the activities of the former processes from each other. In BPEL, the activities can be isolated from each other by placing each activity set A_i into a separate container `scope` CS_i that catches all faults.

(iii) *Control Flow Materialization:* Based on the interaction patterns, the control flow between all pairs of activity sets A_i and A_j is materialized.

(iv) *Resolving Language Violations:* The materialization may lead to control flow or data flow constructs in P_{Merged} violating certain language constraints. For instance, as a result of the materialization of the synchronous interaction between the "Travel Agency" and the "Airline", control links are created that cross the boundaries of the `forEach`. However, crossing loop boundaries is forbidden in BPEL. In this step, these violations have to be resolved.

(v) *Data Flow Adjustments:* To share data between activities within different container scopes CS_i and CS_j, the variables used by "syn_s" and "syn_{rc}" to emulate the message transfer between the scopes have to be globalized, i. e., they have to be lifted to the process `scope` P_{Merged}.

3 MULTI-INSTANCE INTERACTIONS

To describe the consolidation operation for multi-instance interactions in Section 4, this section discusses the control flow constraints implied by multi-instance interactions (Section 3.1) and the different approaches to initiate multi-instance interactions (Section 3.2).

3.1 CONTROL FLOW CONSTRAINTS

On process instance level, an instance of the processes $P_1, \ldots, P_n$ can create another instance of $P_1, \ldots, P_n$ (see Section 3.2) and synchronize itself with this or other instances via message exchanges. Apart from that, the instances run isolated from each other. If, for instance, one instance crashes, the other instances continue their operation. We refer to this as "instance autonomy". As a result, the different instances of one multi-instance process run completely isolated from each other if they do not exchange messages (Kopp et al., 2010). This also implies that there is no control flow relation between an instance i and an instance j of the same activity where instance i belongs to another process instance than instance j. In our example in Figure 1 an instance i of "Calculate Price" may be executed before, during, after etc. an instance j of "Calculate Price" during choreography execution. In this work, we will discuss how far this instance autonomy can be preserved in P_{Merged}.

3.2 MULTI-INSTANCE PROCESS INSTANTIATION

The only explicit influence a process P_A has on the lifecycle of another process P_B is when P_A creates an instance of process P_B. In BPEL, a process instance is created implicitly when an instance creating activity receives a message. In this paper, as instance creating activities, we treat the `receive` activity (flagged with `createInstance`) only and do not regard the 1:m choice activity `pick`. The example choreography contains the one instance-creating activity "Rc Price Request" that creates an instance of the "Airline" process after it received a message via message link $m1$. When a message link connects an invoke with an instance creating activity, we call the invoke "starting invoke"

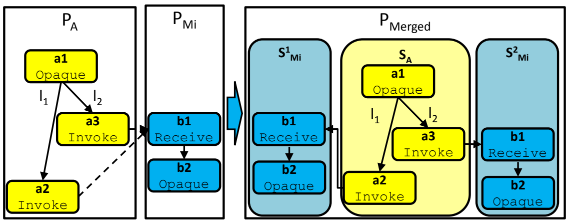

Figure 3: Static Multi-Instance Partner Instantiation (left) and Container Generation (right)

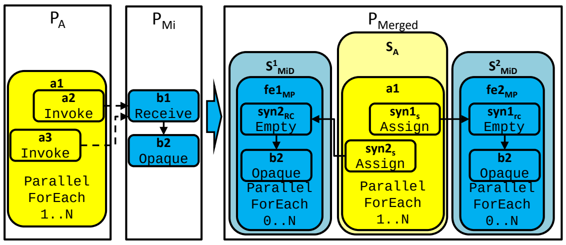

Figure 4: Dynamic Multi-Instance Partner Instantiation (left) and Container Generation (right)

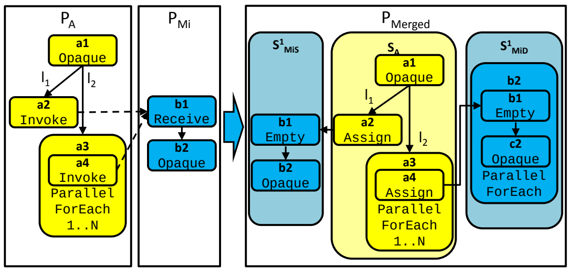

Figure 5: Hybrid Multi-Instance Partner Instantiation (left) and Container Generation (right)

(a_{start}) and the instance creating activity a_{init}. We distinguish between *static*, *dynamic* and *hybrid* instance creation scenarios that base on the multiple instance patterns described by Aalst et al. (van der Aalst et al., 2003).

In *static multi-instance creation scenarios*, the number of instances of multi-instance process to be executed can be determined at design time. Static instantiation can be either implemented by performing exactly one instance of one or more possible starting invokes a_{start} or by performing a fixed number of instances of the same activity a_{start}, e. g., within a repeatable construct whose number of iterations is defined at design time. In Figure 3 on the left two instances of multi-instance process P_{Mi} are created when $a2$ and $a3$ are performed.

In *dynamic multi-instance creation scenarios*, the number of instances of a_{start} is unknown at design time but determined at runtime. Dynamic scenarios can be implemented by repeatable constructs whose number of iterations is only known at runtime such as loops or event handlers. In this work, we focus on `forEach` loops. The number of instances of the activity "Send Price Request" that create an instance of the "Airline" process is, for instance, determined by the set of available airlines that is created during runtime of activity "Determine Airlines". Another example of dynamic instance-creation is shown at the left side of Figure 4. There, each activity instance of $a2$ and $a3$ creates a process instance of P_{Mi}. The number of instances of P_{Mi} depends on the number of executed activity instances of $a2$ and $a3$. That, in turn, depends on the number of iterations of their surrounding `forEach` loop whose upper bound N is determined at runtime.

Also a combination of both instantiation approaches can be implemented, i. e., some instances of multi-instance process are created statically while others are created dynamically. An example for this *hybrid instantiation approach* is shown in Figure 5. Each execution of activities $a2$, $a3$ creates an instance of P_{Mi}, but the number of instances of $a4$ to be executed is not known at design time.

4 MULTI-INSTANCE PROCESS CONSOLIDATION

Based on the consolidation steps for single instance interactions described in Section 2.2, the control-flow properties and the different approaches to initiate multi-instance interactions described in Section 3, the consolidation approach for multi-instance interactions is proposed in this section.

4.1 CONTAINER GENERATION

As described in Section 2.2, in the second step of the consolidation, for each process model $P_1, \ldots, P_n$ to be merged a container $S_1, \ldots, S_n$ (`scope`) is created in P_{Merged}, where each container contains the respective activities of $P_1, \ldots, P_n$. This approach cannot be directly applied for multi-instance processes as a (potentially unknown) number of instances of activities have to be executed during the execution of P_{Merged} that run simultaneously and isolated from each other. We have to distinguish between the container generation for static multi-instance creation scenarios and for dynamic multi-instance creation scenarios as in the former scenario the number of activity instances to be isolated is known while in the other case it is unknown.

During the execution of C, one static starting invoke activity a_{start} can potentially create one instance of a multi-instance process by sending a message to an activity a_{init}. To emulate this behavior, "multi-instance process unrolling" is performed. That means, for each static starting invoke a^i_{start}, a separate container S^i_{MiS} is created as immediate child activity of P_{Merged}. Even though the invokes may be mutually exclusive, we generate a container for each invoke. BPEL's execution semantics (the dead path elimination) ensures that the container is only executed if the invoke itself had been executed. This also implies that for each activity a_{start} residing within a handler (fault handler, event handler, ...), a container CS_{MP} is generated as they may potentially be executed during the execution of $\mathcal{C}$. Thus, the number of static instantiation activities for multi-instance process determines the number of containers CS_{MiS} that are generated within P_{Merged}. The execution of the two static starting invoke activities for P_{Mi} $a2$ and $a3$ in Figure 3 could be mutually exclusive if the transition conditions of control links l_1 and l_2 always evaluate to different Boolean values or if both links evaluate to *false*. Nevertheless, due to the over-approximation the two containers S^1_{MiS} and S^2_{MiS} are created in P_{Merged}. Creating a separate container S_{MiS} for each multi-instance process instantiation also ensures the same instance-independence of the activities as in $\mathcal{C}$. To perform the control flow materialization correctly for each generated container CS_{MiS}, a new set of message links is created.

As the number of instances of a_{start} cannot be determined in dynamic multi-instance creation scenarios, the number of process instances of multi-instance process is not known at design time. Hence, the multi-instance process cannot be "unrolled" into different containers as in static multi-instance creation scenarios. To emulate these scenarios in P_{Merged}, a container for the activities of the multi-instance process is needed. This container has to realize the simultaneous and isolated execution of an at design time unknown number of instances of the activities of the multi-instance process. The only construct in BPEL supporting a simultaneous execution of a number of instances (branches) of its root activities (along with its children) in an isolated fashion is the parallel `forEach` activity. Therefore, for each dynamic a_{start}, a dynamic container `scope` S_{MiD} is created that contains a parallel `forEach` activity fe_{MP} that contains the root activity of the multi-instance process within its `scope`. The scope is required as BPEL enforces the immediate child of a `forEach` loop being a `scope`. This results in two levels of isolation: (i) the scope within the `forEach` isolates the root activity instances of the multi-instance process from each other by catching all faults that may be thrown by them; (ii) the scope S_{MiD} isolates fe_{MP} from the other containers.

Note that for each dynamic a_{start}, a separate dynamic container S_{MiD} is created. Even if n distinct dynamic a_{start} activities that send a message to the same instance-creating activity a_{init} are located within the same parent loop as shown in Figure 4. The reason for creating a separate dynamic container for each of the n activities $a^1_{init}, \ldots, a^n_{init}$ is that all or a subset of the a_{init} activities may be performed simultaneously.

The attribute values for fe_{MP} are defined as follows: It gets the start and the final counter value of the parent forEach of a_{start} and a unique id to distinguish it from other forEach loops. The counter values are required to assign data between the instances of the forEach loop as described in Section 4.4. It may happen that a_{start} is not executed during each iteration due to certain control flow conditions. Consequently, more instances of fe_{MP} may be created than necessary. Due to the synchronization activities that are created during the control flow materialization, it is ensured that the business activities in the "unused" instances are not activated. The parent forEach of an a_{start} may have a completion condition defined that specifies that the processing of the forEach maybe ended as soon as a subset of its branches completed successfully even though there are still running branches. This condition is not applied to fe_{MP}, because during the execution of $\mathcal{C}$ all instances of a multi-instance process that were created are allowed to complete.

In the hybrid instantiation scenarios of multi-instance processes, static and dynamic multi-instance containers are created in P_{Merged} that contain the activities of the multi-instance process. In Figure 5, the static container S^1_{MiS} is created and the dynamic container S^1_{MiD} for the instantiation activities $a2$ and $a4$ respectively.

Static and dynamic multi-instance containers are started when P_{Merged} becomes active. However, the business activities within the containers are not executed until the incoming links are activated.

4.2 CONTROL FLOW MATERIALIZATION

The basic concepts to materialize the control flow from the message flow for single instance scenarios have been described in Section 2.2. These concepts can be also applied to multi-instance scenarios since on the instance level also one-to-one communication takes place between the interacting processes. For instance, in the example choreography in Figure 1, an instance of the "Send Price Request" activity communicates with exactly one instance of the "Rc Price Request" activity. Therefore, we do not need to modify the control flow materialization for multi-instance scenarios, neither for static, dynamic, nor hybrid instantiation scenarios. Activities that were used in $\mathcal{C}$ to assign the endpoint references (EPR) of the processes to be called from the communication activities can be removed.

For instance, the activity "Determine Airlines" returns a set of EPRs. In each iteration of "Collect Prices FE" another airline EPR is assigned to "Send Price Request". This assignment is not shown in Figure 1. The consolidated process model P_{Merged} created from the example choreography is depicted in Figure 2.

4.3 RESOLVING LINK VIOLATIONS

If dynamic multi-instance containers are created, the control flow materialization causes cross-boundary link violations as the consolidation always generates control links between synchronization activities that cross boundaries of forEach loops. For instance, the link $l1$ from activity "$\text{syn}_{\text{SendReq}}$" to activity "$\text{syn}_{\text{RcReq}}$" in Figure 2. To resolve these violations, the link status value ("true" if the link is enabled, "false" if it is disabled and "undef." if undetermined) of "syn_s" to "syn_rc" could be written to a new variable v_{ls} that can be also accessed from within the loop to check if "syn_rc" can be started, i. e., if the link is enabled. However, as from within the loop it is unknown when v_{ls} is set, another loop preceding "syn_rc" is needed that constantly polls the value of v_{ls} until it is set from "undef." to "true" or to "false". As the permanent polling stresses the workflow engine and the underlying resources, this solution is not an option. Of course, one could adjust the polling interval but a useful polling interval would have to be determined for each business scenario to ensure the overall process execution time is not negatively affected. Instead, we propose a *forEach loop fragmentation approach* to resolve the cross-boundary link violations. We also describe how link status values can be propagated between the created forEach loop fragments.

4.3.1 FOREACH LOOP FRAGMENTATION

Each pair of activities "syn_s" and "syn_rc" violating the cross-boundary link constraint is moved from its original forEach loop into a new forEach loop fe_{syn}. By convention, fe_{syn} is always placed into the container scope CS of "syn_s" to ensure that no data are copied by "syn_s" if another activity fails within CS. All opaque activities directly or indirectly preceding "syn_s" (but no other synchronization activity) are moved from their original forEach loop to a new FE-fragment fe_{pred} that is also created in the container scope of "syn_s". The direct and indirect predecessor activities of "syn_rc" (if any) that do not precede another synchronization activity and that were contained in the same original forEach as "syn_rc", are moved to a new FE-fragment fe_{rc}. This FE-fragment must be created in the container scope of "syn_rc" as the opaque activities preceding "syn_rc" originate from this container. The FE-fragments fe_{syn} and fe_{rc} are connected via a control link $l(fe_{rc}, fe_{syn})$. This maintains the original execution order that defined that "syn_rc" is not started until its predecessors completed. Accordingly, FE-fragment fe_{pred} is connected to fe_{syn} via control link $l(fe_{pred}, fe_{syn})$ to ensure that "syn_s" is only started after its predecessor activities completed. If "syn_s" had no predecessor activities in its original forEach, fe_{syn} is connected to the predecessor activities of the original forEach of "syn_s". FE-fragment fe_{syn} is either connected to the FE-fragment that contains the direct successor activities of "syn_s" or directly to its successor activities in case it does not reside within a forEach. Additionally, fe_{syn} is connected to the forEach hosting the direct successor activities of "syn_rc" or directly to the successor activities of "syn_rc" that do not reside within a forEach. All opaque activities that do not precede any synchronization activity are left in their corresponding original forEach loops and connected to the fe_{syn} fragment that contains their preceding "syn_s" and "syn_rc" respectively. Note, that all FE-fragments that host opaque activities inherit the start and end counter values of the forEach loop where the opaque activities originate from. Also the fault handlers and termination handlers are adapted from the original forEach. FE-fragments fe_{syn} inherit the attributes and handlers from `forEach` syn_s. The control links between the activities within a FE-fragment are kept to maintain the execution order between these activities. All incoming links of activities whose predecessors reside within another FE-fragment are removed. When the successor activities of an activity were moved to another FE-fragment its outgoing links are removed. Figure 6 shows the fragmented version of P_{Merged} of our example scenario. In the "Travel Agency" container the two FE-fragments "CollectPricesFE_1" and "CollectPricesFE_2" are created. The "Airline" container hosts the activities "AirlineFE_1" to "AirlineFE_3". The control links between the FE-fragments and also their preceding and succeeding activities are created as described above.

4.3.2 LINK STATUS PROPAGATION

The split of the original forEach loops of P_{Merged} into several FE-fragments causes control links to break if activities a_{src} and a_{trg} that were connected via link $l(a_{src}, a_{trg}, tc)$ (tc represents the transition condition of the link) were placed into different FE-fragments fe_m and fe_n. Khalaf and Leymann (Khalaf and Leymann, 2006) describe an approach to split a single process into several individual process fragments. As this also causes control links to break if the source and target activity of a link reside within different process fragments, they described a technique to prop-

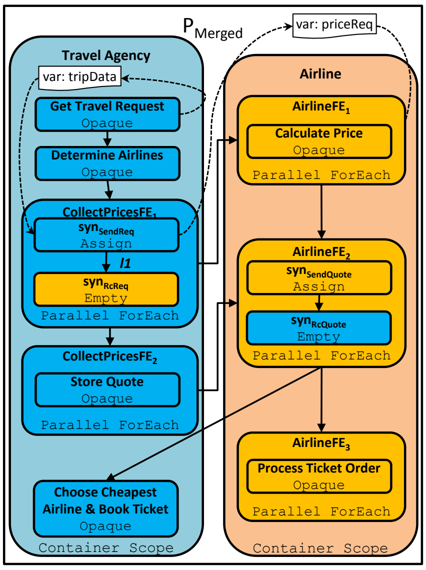

Figure 6: Fragmented Version of Consolidated Process Model P_{Merged}

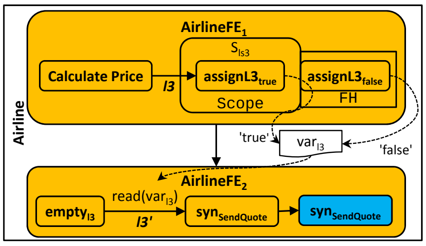

Figure 7: Passing Link Status Values Between FE-Fragments

agate the link status from one process fragment to another via message exchanges. We adapt this approach to propagate the link status from one FE-fragment to another by using variables instead of message exchanges. In the FE-fragment hosting a_{src} a new scope S_{ls} with a fault handler FH is added. To S_{ls} the assign activity $assign_{true}$ and to FH the assign activity $assign_{false}$ is added. $assign_{true}$ writes the link status value "true" to a newly introduced variable var_{ls} that resides in the parent scope of fe_m and fe_n. $assign_{false}$ writes the link status value "false" to var_{ls}. A new link $l_{src}(a_{src}, assign_{true}, tc)$ is created. The attribute *suppressJoinFailure* within S_{ls} is set to "no", i. e., when the transition condition tc evaluates to "false" a *bpel:joinFailure* is thrown. This failure is caught by FH and $assign_{false}$ is executed that writes the link status value "false" to a newly created variable var_{ls}. If tc evaluates to "true" $assign_{true}$ writes "true" to var_{ls}. In the FE-fragment that hosts the link target a_{trg} a new `empty` activity *em* is created that is connected to a_{trg} via the new link $l_{trg}(em, a_{trg}, read(var_{ls}))$. Thereby, the function $read(var_{ls})$ reads the link status of var_{ls}. An instance of the `empty` activity *em* is started as soon as the corresponding `forEach` branch becomes active. As the execution order between activities a_{src} and a_{trg} is preserved by the execution order between their hosting FE-fragments, the value of var_{ls} is always set before *em* is started. Figure 7 shows exemplary the FE-fragments "AirlineFE$_1$" and "AirlineFE$_2$" with the link propagation logic. To propagate the status of link $l3$ from "Calculate Price" to "syn$_{\text{SendQuote}}$" as in the original unfragmented "AirlineFE" the status of the outgoing link of "Calculate Price" is written to var_{l3} by "assignL3$_{\text{true}}$" and "assignL3$_{\text{false}}$" respectively, depending on the status of $l3$ in "AirlineFE$_1$". This status is then applied to the new link $l3'$ in $AirlineFE2$ by reading var_{l3}. The link propagation approach does also ensure that death-path elimination can be performed. If, for instance, the incoming link of "Calculate Price" evaluates to "false" and "syn$_{\text{SendQuote}}$" had any outgoing links (which is not the case in the example), the outgoing links of "syn$_{\text{SendQuote}}$" are automatically set to "false".

So far, we regarded one variable var_{ls}, where the link status is written to and read from. However, since multiple instances of the same link are created, also multiple instances of link status values must be hold. Therefore, var_{ls} has to be extended to hold multiple status values of the same link and also assign activities have to be modified accordingly to access the correct instance of a link status. This is described in Section 4.4.

4.4 DATA FLOW

As described in Section 2.2, the variables have to be globalized to the process scope of P_{Merged} for sharing data between different container scopes CS. To share data and link status values between different FE-fragments that originate from the same forEach, these variables have to be lifted to the parent container scope CS_{MiD} of the FE-fragments. In contrast to the one-to-one interaction scenarios during choreography runtime, several instances of the same variable are created, e. g., for each "Airline" partner, one instance of the variable "priceReq". We refer to these variable as "multi-instance variables" V_{MI}. To store multiple instances of a variable v_{MI}, it is replaced by a map mp_{MI}. Each entry of mp_{MI} represents an instance of v_{MI}, i. e., the entries of mp_{MI} inherit the data type of the original v_{MI} and can be uniquely identified by a key key_{IID} (instance id) of type `<xsd:id/>` that represents an instance of v_{MI}.

This raises the question how the key_{IID} is composed, how a branch (instance) of a FE-fragment can be related to a certain entry in mp_{MI} via key_{IID} and how to ensure that data can be shared between different FE-fragments. Intuitively, EPRs could be used to compose keys as they are also employed in $\mathcal{C}$ to identify an instance of a multi-instance process. However, there is technically no way to inject a key such as an EPR into a forEach branch from outside to a specific instance when it starts. Thus, a forEach branch is not aware of what key it is related to. Therefore, a convention is needed to enable the branch to determine its associated key by itself. The only information that uniquely identifies a forEach branch among the other branches and that does not change during its lifetime is the value of the instance counter variable. As different forEach loops are contained within P_{Merged}, their instance counter values are not globally unique but only within the respective forEach. To ensure global uniqueness and to share data between the same branches (i. e., branches with the same counter value) of FE-fragments that originate from the same forEach loop, the static id *feid* (defined at design time) and the dynamic instance counter value *iid* (defined at runtime) are concatenated. Note, that all FE-fragments that originate from the same forEach share the same *feid*. Assume, for instance, that the execution of three instances of the "Airline" process has to be emulated. The first instance would get the instance key "AirlineFE_1", the second "AirlineFE_2" and so on. Similarly, three instances of the forEach "CollectPricesFE" would get the instance keys "CollectPricesFE_1" to "CollectPricesFE_3". As described in Section 4.1, fe_{MI} inherits the start and end counter value of the parent forEach of a_{start} (e. g., "AirlineFE" has the same start and end counter values as "CollectPricesFE"). Also the different FE-fragments inherit the values of the forEach loops they were created from. Since all of these FE-fragments share the same start and end counter variables, their branches can be logically related to each other by using their instance id *iid*, e. g., branch "CollectPricesFE_1" can be related to "AirlineFE_1", "CollectPricesFE_2" to "AirlineFE_2" etc. Data can be passed between all logical related FE-fragments by rewriting assign statements and transition conditions accessing multi-instance variables v_{MI}. As these variables are always located within FE-fragments, data accesses to v_{MI} can be rewritten during the consolidation to data accesses to those entries of mp_{MI} that share the same *iid*.

5 DISCUSSION

In this section the properties of a process P_{Merged} created by applying the multi-instance process consolidation approach proposed in Section 4 are discussed.

The consolidation maintains the control flow constraints defined in $\mathcal{C}$. On the one hand, the original control links between activities moved to the same FE-fragment (e. g., "$\text{syn}_{\text{SendQuote}}$" and "$\text{syn}_{\text{RcQuote}}$" in FE-fragment "$\text{AirlineFE}_2$") are maintained. One the other hand, the execution order between consecutive activities a_i and a_j moved to different FE-fragments is kept by connecting the FE-fragments based on the control relation between a_i and a_j. The proposed propagation technique for link status values guarantees that activities are only executed if their incoming links evaluate to "true" even if link source activity resides within another FE-fragment. Since the fault handler and termination handler from the original `forEach` are attached to the FE-fragments and since the FE-fragments are executed sequentially the fault handling and semantics of BPEL is also kept, i. e., if within a certain FE-fragment branch an activity fails, all running activities within this branch are terminated and activities within the succeeding FE-fragment branches that represent the same instance (i. e., branches with the same instance counter value *iid*) are not started anymore.

The consolidation keeps the control flow constraints modeled in $\mathcal{C}$. However, P_{Merged} cannot generate all execution traces between business activities that could be generated by $\mathcal{C}$. The reason is that the multi-partner instances lose their instance autonomy from each other. The instance autonomy is lost, because of the FE-fragmentation. Assume, for instance, the execution time of activity "Calculate Price" in the choreography in Figure 1 depends on the input data, i. e., the price request. Then, depending on the request, the instance "Calculate Price^{i}" is long running while in another instance "Calculate Price^{j}" is short running. Therefore, the succeeding activity instances of "Calculate Price^{i}" would be started earlier as those of "Calculate Price^{j}". If all succeeding activity instances i and j would need the same execution time, all activity instances i would be completed before activity instances j. The FE-fragmentation, however, imposes also control flow relations between different instances that are emulated. This is due to the fact, that all branches of a forEach, i. e., an FE-fragment, have to be completed until the succeeding FE-fragment can be started. Hence, no instance of "$\text{syn}_{\text{SendQuote}}$" is started until all instances of "Calculate Price" completed as the emulated instances have to "wait" for each other. This can significantly increase the time until a business goal is reached, especially, if the execution times of individual instances of the same activity are highly different and if only a subset of instances is required for fulfilling the business goal (e. g., the "Travel Agency" may only need the response from three out of 10 airlines). However, if $\mathcal{C}$ and P_{Merged} are started with the same input data, their overall execution time is the same, as in both case all business activity instances (that are activated via a path) have to be executed. If we have just static multi-instance scenarios where a_{start} does not reside within a forEach loop, we do not encounter the problem of losing instance autonomy that was described above. That is the reason why we distinguish between these scenarios.

The multi-instance consolidation may lead to a complex process P_{Merged}. This is because the multi-instance process unrolling and the forEach fragmentation create many new activities and the corresponding control links in P_{Merged}. Hence, the maintainability of P_{Merged} is decreased, i. e., the process structure is difficult to understand. This, in turn, makes it very difficult to change the process. For that reason we recommend that the consolidation should be applied when the choreography is not changed anymore. For instance at deployment time the choreography can be consolidated to save resources (refer to Section 1). To monitor the state of the original choreography state propagation rules can be used that derive the state of the choreography from from the runtime state of P_{Merged} (Wagner et al., 2012a).

6 IMPLEMENTATION

In previous work (Debicki, 2013) a Java prototype was developed that gets a BPEL4Chor choreography as input and returns the consolidated process model in BPEL. The consolidated process model can then be deployed on a workflow engine (Wagner et al., 2013b).

Initially the prototype implemented the consolidation steps for one-to-one interactions described in Section 2.2. For evaluation of the multi-instance process consolidation approach described in Section 4 it was extended (Dadashov, 2013) to create multi-instance containers in P_{Merged}, to perform the forEach loop fragmentation and to create multi-instance variables.

Internally, the prototype performs the consolidation steps on a BPEL4Chor EMF[1] object model[2] which is an extension of the Eclipse BPEL EMF model[3]. The EMF models provide Java object serializations of BPEL processes and BPEL4Chor choreographies respectively. To consolidate interacting processes the prototype reads a ZIP file that contains the XML representations of the participant topology and the participant behavior descriptions. These XML representations are transformed into a BPEL4Chor EMF model and a new BPEL EMF model for P_{Merged} is generated. Then the prototype inspects the BPEL4Chor EMF model to determine the container scopes to be generated in P_{Merged} and to perform the control flow materialization. After the consolidation operations completed P_{Merged} is transformed back into a BPEL schema-compliant XML. A process modeler has to replace the `opaque` activities in P_{Merged} (if any) by executable activities (executable completion (Aalst et al., 2008)). As deployment descriptors a workflow engine specific the process modeler also has to add them manually after the consolidation completed. Then P_{Merged} can be deployed.

[1]http://www.eclipse.org/modeling/emf/

[2]https://github.com/IAAS/BPEL4Chor-model

[3]http://www.eclipse.org/bpel/developers/model.php

7 RELATED WORK

Mendling and Simon (Mendling and Simon, 2006) propose an approach where semantically equivalent events and functions of Event Driven Process Chains (Scheer et al., 2005) are merged. Küster et al. (Küster et al., 2008) describe how change logs can be employed to merge different process variants that were created from the same original process. These approaches merge processes that are semantically equivalent or that are different variants of the same original process. Our approach focuses on the consolidation of collaborating processes into a single process model. Moreover, none of these approaches deals with multi-instance interactions.

An alternative way to generate a BPEL orchestration of a BPEL4Chor choreography is using an intermediate format. However, there is currently no approach keeping the structure of the generated orchestration close to the structure of the original choreography. For instance, Lohmann and Kleine (Lohmann and Kleine, 2008) do not generate BPEL scopes out of Petri nets, even if the formal model of Lohmann (Lohmann, 2007) generates a Petri net representation of BPEL scopes.

There are other ways to describe inter-organizational collaboration. For instance, the lifecycle of a business entity can be put into the center of modeling (Hull et al., 2011). In this paper, we did not follow that approach, but used the interconnection model choreography approach.

8 CONCLUSION AND OUTLOOK

In this paper, we introduced an approach to consolidate BPEL processes interacting with multi-instance processes in a choreography into a single process model that emulates the multi-instance behavior of the original choreography. To perform the consolidation, we distinguished three instantiation scenarios for the multi-instance processes. In static instantiation scenarios, the number of instances can be determined during the consolidation. To emulate the static multi-instance scenarios, for each possible instance, a separate container containing activities of the multi-instance process is created within the consolidated process. In dynamic instantiation scenarios, the number of instances cannot be determined during the consolidation. To emulate this scenario, the activities of multi-instance processes are copied into a parallel forEach loop whose number of iterations is determined during runtime. Each branch of this loop emulates one instance of the multi-instance process. The control flow materialization may create control links from the message links that cross the boundaries of the forEach loop what is not permitted in BPEL. To resolve these violations we proposed a technique to sequentially split the forEach loop into different fragments. We also discussed how the data flow within the consolidated process model has to be modified to store and access multiple instances of the same variable.

Currently, we do only support consolidation of multi-instance processes that are instantiated by an instance creating activity that resides in a forEach loop. In the future also the consolidation of multi-instance processes is supported that are instantiated from within nested and condition-controlled loops such as while loops.

Reference passing is an additional major aspect of multi-instance interactions, e. g., the travel agency could pass the endpoints of the three cheapest airlines to a traveler process who then decides by itself what airline is booked. Our approach has to be extended accordingly to support these interaction scenarios as they are very common in the context of multi-instance processes. We also have to discuss more in depth how BPEL's compensation handling mechanism is affected by the split of a single forEach into different fragments.

As shown by Wagner et al. (Wagner et al., 2013b), the execution time and performance overhead of a choreography execution can be significantly reduced if its processes are consolidated into a single process model, because message serialization, message transport, and message de-serialization is avoided. In future work, we have to analyze the performance differences between consolidated process models that emulate multi-instance processes and their original choreography.

As BPMN becomes more and more important and since it shares a lot of concepts and language constructs with BPEL and BPEL4Chor, we have to investigate how our consolidation approach can be applied to merge BPMN collaboration diagrams.

References

Aalst, W. M. P. v. d., Lohmann, N., Massuthe, P., Stahl, C., and Wolf, K. (2008). Multiparty contracts: Agreeing and implementing interorganizational processes. *Comput. J.*, 53(1):90–106.

Barros, A., Decker, G., Dumas, M., and Weber, F. (2007). Correlation patterns in service-oriented architectures. In *FASE*, volume 4422. Springer.

Barros, A., Dumas, M., and ter Hofstede, A. (2005). Service Interaction Patterns. In *BPM*. Springer.

Börger, E. (2012). Approaches to modeling business processes: a critical analysis of bpmn, workflow patterns and yawl. *Softw. Syst. Model.*, 11(3):305–318.

Dadashov, E. (2013). Choreography-based Business Process Consolidation in One-To-Many interactions. Master thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany.

Debicki, P. (2013). Choreographie-basierte Konsoliedierung von BPEL Prozessmodellen. Diploma thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany.

Decker, G., Kopp, O., and Barros, A. (2008). An Introduction to Service Choreographies. *Information Technology*, 50(2):122–127.

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

DeRemer, F. and Kron, H. (1976). Programming-in-the-large versus programming-in-the-small. *Software Engineering, IEEE Transactions on*, SE-2(2):80 – 86.

Hull, R., Damaggio, E., Fournier, F., Gupta, M., Heath, F. T., Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P., et al. (2011). Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In *7th International Workshop, WS-FM 2010*, LNCS. Springer-Verlag.

Khalaf, R. and Leymann, F. (2006). Role-based Decomposition of Business Processes using BPEL. In *ICWS 2006*. IEEE.

Khalaf, R. and Leymann, F. (2010). Coordination for Fragmented Loops and Scopes in a Distributed Business Process. In *BPM*. Springer.

Kopp, O., Eberle, H., Leymann, F., and Unger, T. (2010). The Subprocess Spectrum. In *Proceedings of the Business Process and Services Computing Conference: BPSC 2010*. GI e.V.

Kopp, O., Martin, D., Wutke, D., and Leymann, F. (2009). The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages. *Enterprise Modelling and Information Systems*, 4(1):3–13.

Kossak, F., Illibauer, C., and Geist, V. (2012). Event-based gateways: Open questions and inconsistencies. In Mendling, J. and Weidlich, M., editors, *Business*

Process Model and Notation, volume 125 of *Lecture Notes in Business Information Processing*, pages 53–67. Springer Berlin Heidelberg.

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*.

Leymann, F. (2010). Bpel vs. bpmn 2.0: Should you care? In *BPMN*, pages 8–13.

Lohmann, N. (2007). A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In *WS-FM*.

Lohmann, N. and Kleine, J. (2008). Fully-automatic Translation of Open Workflow Net Models into Simple Abstract BPEL Processes. In *Modellierung*. GI e.V.

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

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.

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

van Breugel, F. and Koshkina, M. (2006). Models and Verification of BPEL. `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf`.

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.

Wagner, S., Fehling, C., Karastoyanova, D., and Schumm, D. (2012a). State Propagation-based Monitoring of Business Transactions. In *Proceedings of the 2012 IEEE International Conference on Service-Oriented Computing and Applications*. IEEE Xplore.

Wagner, S., Kopp, O., and Leymann, F. (2011). Towards Choreography-based Process Distribution In The Cloud. In *CCIS*.

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

Wagner, S., Kopp, O., and Leymann, F. (2013a). Consolidation of Interacting BPEL Process Models with Fault Handlers. In *Proceedings of the 5th Central-European Workshop on Services and their Composition (ZEUS 2013)*, pages 1–7, Rostock. CEUR Workshop Proceedings.

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

Wohed, P., van der Aalst, W. M. P., Dumas, M., ter Hofstede, A. H. M., and Russell, N. (2006). On the suitability of bpmn for business process modelling. In *Business Process Management*, pages 161–176.