Consolidation of Interacting BPEL Process Models with Fault Handlers

Sebastian Wagner, Oliver Kopp, and Frank Leymann

Institute of Architecture of Application Systems, University of Stuttgart, Germany
{wagnerse,kopp,leymann}@iaas.uni-stuttgart.de

Abstract The interaction behavior between processes of organizations and their suppliers can be modeled by using choreographies. When an organization decides to gain more control about their suppliers and to minimize transaction costs they may decide to insource these companies. This also requires the integration of the partner processes into the organization. In previous work we proposed an approach to merge (consolidate) interacting BPEL process models of different partners into a single process model by deriving control flow links between the process models from their interaction specification. In this work we are detailing this consolidation approach. Thereby, special attention is turned on extending the consolidation operations in a way that process models with fault handlers can be merged.

1 Introduction

To reduce transaction costs or to gain more control companies often decide to integrate suppliers into their organization (in-sourcing, mergers, and acquisitions). This requires the integration of the processes and the organizational structure of the two companies. In this work we focus on the integration on the process-level. More precisely, we want to merge (or consolidate) complementing process models whose interaction behavior is described by a choreography.

Process modeling languages such as BPEL [10] or BPMN [11] offer different language constructs to raise and handle faults that work similar to *throw-catch* constructs in traditional programming languages such as Java. As fault handling constructs can also cause message exchanges between interacting processes they are also affected by the consolidation. In this paper we want to describe a technique to merge BPEL process models that communicate via fault handlers. Moreover, we describe an extension of the merge operations proposed in [13] and [14].

We use BPEL4Chor [2] to model BPEL choreographies as this language provides a means to define message links between the communication activities of the interacting BPEL processes.

We assume that the reader is familiar with BPEL. Nevertheless, we give a brief overview about BPEL's fault handling concepts in Sect. 2. In Sect. 3 an overview on the merge operations is provided and extensions to them are discussed. Then, Sect. 4 presents a technique to merge processes that communicate via BPEL fault handlers. After discussing related work in Sect. 5, Sect. 6 concludes this paper and provides an outlook on future work.

2 BPEL Fault Handling Basics

BPEL offers three language constructs to repair faulty situations during process execution, namely *fault handlers*, *compensation handlers* and *termination handlers*. If a fault occurs within a `scope` all running activities within this `scope` are terminated and its fault handlers are called. A fault handler is represented by a `catch` or `catchAll` block. Thereby, multiple `catch` blocks can be defined for a `scope`. Each `catch` block catches a particular fault that may be thrown during execution of the `scope` and contains BPEL activities to handle this fault. A `catchAll` block contains logic to catch all other faults that do not match to a particular `catch` block. If no explicit fault handlers are defined for a `scope` it has an implicit default fault handler attached to it. If any kind of fault occurs during the execution of the `scope` the default fault handler triggers compensation handling for its child `scopes` (see below) and finally rethrows the fault to its parent `scope`. If this `scope` does not provide a fault handler for this fault either, it is propagated up to its parent `scope` and so on until the process `scope` is reached. If the process `scope` cannot catch the fault the process fails and is terminated.

Compensation handlers contain activities to undo work that was successfully performed by a `scope` they are attached to, e. g., canceling a flight that was booked by the activities of the `scope`. Hence, they are only executed if their associated `scope` has completed successfully.

To control the termination of a `scope` that is still running a termination handler can be attached to it. Within the termination handler activities can be defined that are performed before the actual termination of the `scope`. If no explicit termination handler was defined for a `scope` its default termination handler compensates its child `scopes`. A more detailed discussion about fault and compensation handling concepts in BPEL was provided by Khalaf et al. [4].

3 Asynchronous and Synchronous Consolidation

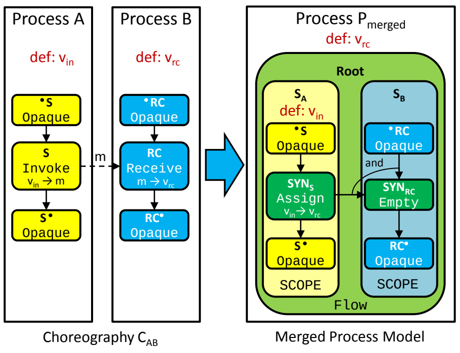

Figure 1. Asynchronous Merging Operation

We introduced the consolidation operations to merge asynchronous and synchronous communicating process models in [13]. The aim of the consolidation is that the atomic activities of the different participants in the merged process model have the same control flow relations as in the original choreography. The basic idea behind the consolidation algorithm is that the message links imply control flow relations between the activities of the communicating process models. The message link *m* in Fig. 1 implies for instance that the successor *RC*• of the `receive` activity is always performed after the predecessor activity •*S* of the `invoke` activity *S* in process model *A* as *RC*• cannot be performed before *S* completed. However, no statement can be made about the execution sequence between $S\bullet$ and $RC\bullet$, e. g., if they have to be performed simultaneously or if $S\bullet$ is performed before $RC\bullet$ and so on. This is different from the synchronous scenario depicted in Fig. 2. There $RC\bullet$ is always performed before $S\bullet$. As activity S does not complete until it received a response message from RP that is performed after $RC\bullet$. Given these implicit control flow dependencies, the sending and receiving activities can act as merge points. Therefore the consolidation operation *materializes* the implicit control flow to explicit control flow relations between the activities.

The consolidation algorithm to merge an arbitrary number of process models is described in the following. As a prerequisite we assume that the choreography is modeled correctly [3] and deadlock free [8]. Moreover, we assume there exists just one instance of each participant per choreography instance, i. e., interaction patterns involving multiple instances of one participant such as *one-to-many send/receive* [1] are not supported yet by the consolidation algorithm. Another restriction we make is that a repeatable constructs such as a BPEL `ForEach` loop do not contain any communication activities that are replaced by control flow links between the process models to be merged. As this would violate the BPEL restriction that repeatable constructs must not be crossed by control flow links [SA00070] [1].

In a first step a new process model P_{merged} is created that contains a `flow` as root activity. For each of the process models P_1 to P_n to be merged a separate `scope` is created in the `flow` activity of P_{merged}. This ensures that the `scope` activities are performed simultaneously. Each scope contains the root activity (along with its child activities) of one of the process models P_1 to P_n. The purpose of the `scope` is to isolate the activities of the process models from each other as they were also isolated in the original choreography. To avoid that an uncaught fault thrown in one scope causes the other scope to crash (as uncaught faults are propagated up to the process scope) a `catchAll` fault handler is added to the scopes as shown in Fig. 2. The `catchAll` contains a `compensate` activity to emulate the default compensation that would have been triggered in an original process without an explicit fault handler. In case an explicit fault handler was defined in a `catchAll` block on the original process scope nothing is changed. If a process scope of a process to be merged has already a a `catch` block defined simply the `catchAll` block is added to this fault handler.

Then the message links are materialized to control flow links. In the asynchronous case the `invoke` activity S is replaced by an `assign` activity SYN_S and the `receive` activity RC by an `empty` activity SYN_{RC}. SYN_S emulates the message transfer between between the former `invoke` and `receive` activity, i. e., it copies the message from the input variable v_s of the `invoke` to the variable v_{rc} of the `receive` activity where the message was copied to before. To perform the assignment the declaration of variable v_{rc} is lifted to the parent `scope` that encloses the two `scopes` that contain the participant activities. Otherwise, SYN_S could not access v_{rc}. The `empty` activity SYN_{RC} replaces the former `receive` RC. To avoid name clashes between variables it might be necessary to adapt the variable names accordingly during the consolidation. The incoming and outgoing links of S and RC are mapped to SYN_S and SYN_{RC}, respectively. An additional link from SYN_S to SYN_{RC} is created. This link ensures that SYN_{RC} is not started before SYN_S was executed.

[1] Static Analysis (SA) Fault Codes defined in the BPEL specification [10]

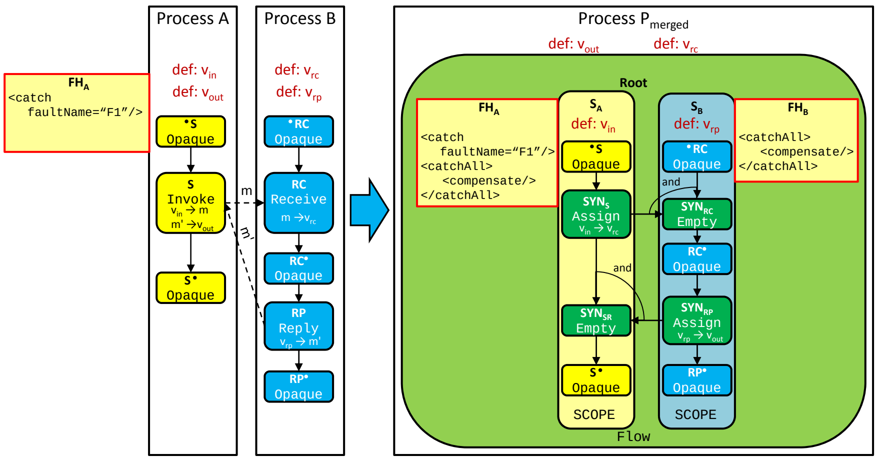

Figure 2. Synchronous Merging Operation

The synchronous merge is sketched in Fig. 2. There additionally the `reply` activity *RP* is replaced by the `reply` activity SYN_{RP} to emulate the transfer of the response message sent via message link m'. SYN_{RP} copies the value of the former `reply` variable v_{rp} to the output variable of the former `invoke` activity v_{out}. The declaration of v_{out} has to be lifted to the parent scope as well to make this variable accessible for SYN_{RP}. The `empty` activity SYN_{SR} is added for the same reason SYN_{RC} was added. The control links of *RC* are mapped to SYN_{RP} and the outbound links of *S* are mapped to SYN_{RC}. Moreover, a new link is created to connect SYN_{S} and SYN_{SR} and another one between SYN_{RP} and SYN_{SR} to ensure that the successors of the former `invoke` activity are not started before.

4 Consolidation in the Context of Fault Handlers

In this section we discuss the challenges that arise when materializing the control flow from message links between communication activities that reside within BPEL fault handlers. Thereby, we focus on the *cross boundary link* constraint imposed by the BPEL specification [SA00071]. This constraint specifies that no control link must point to a target activity within a fault handler from outside the fault handler, i. e., no link must point into a `catch` or `catchAll` block.

In the following we distinguish between three different scenarios (i) fault handlers without communicating activities (ii) fault handlers with only outgoing message links and (iii) fault handlers with at least one incoming message link.

The first scenario is trivial as there is no communication between the fault handlers of the two process models that have to be merged. Consequently, they can be simply merged with the consolidation operations introduced in 3.

The second is scenario is depicted in Fig. 3. In process model *A* the fault handler FH_{A} is attached to the `scope` S_{A}. FH_{A} contains an asynchronous `invoke` activity *A*4 that is related to the corresponding `receive` activity *B*2 in process model *B* via message link m. Note, that for simplicity reasons the `flow` activity containing S_A and S_B is not explicitly depicted in Fig. 3 and in the following figures.

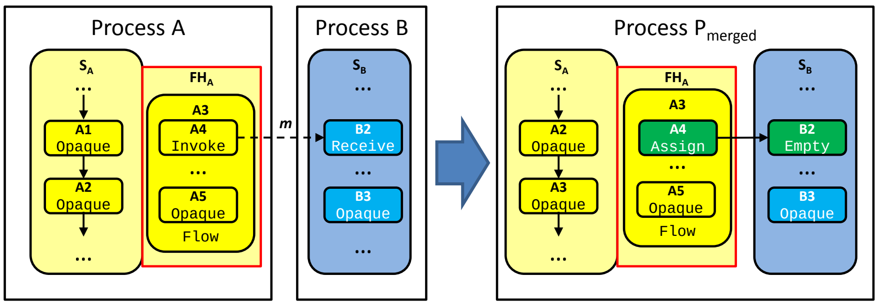

Figure 3. Scenario 2: Message Link pointing from a Fault Handler

To merge the process models in a first step the merge operations introduced in Sect. 3 are applied. This results in process model P_{merged}. As the new control flow links materialized from the message links leave the fault handler boundaries outbound only, the cross boundary link constraint is not violated.

The scenario in Fig. 4 is very similar to the previous one except that `invoke` activity $A4$ is synchronous, hence, a second message link from the `reply` activity points back to $A4$. The synchronous consolidation operation creates from the message link m_2 the control flow link l_2. This link crosses the fault handler boundary of FH_A inbound in order to realize that $A5$ is performed after $B2$ or $B3$ respectively. This violates the cross boundary link constraint.

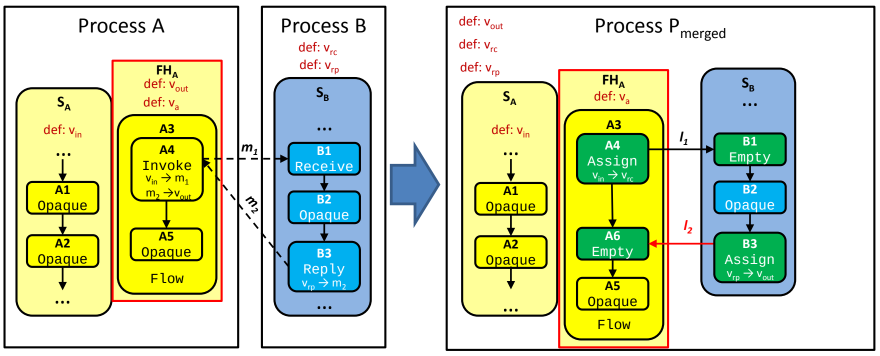

Figure 4. Scenario 3: Message Link pointing into a Fault Handler

To develop a solution to overcome this problem control flow links pointing into a fault handler have to be avoided, hence, the control flow has to be modified accordingly. To resolve the incoming link issue we can either remove the fault handler completely or we move the activities with incoming links out of fault handler. Removing the fault handler completely is not an option as a fault handler can be activated any time during runtime of a scope if a fault is thrown. This behavior cannot be emulated by using other BPEL constructs. Hence, we suggest a solution where the fault handler is kept and the fault handling activities are factored out of it.

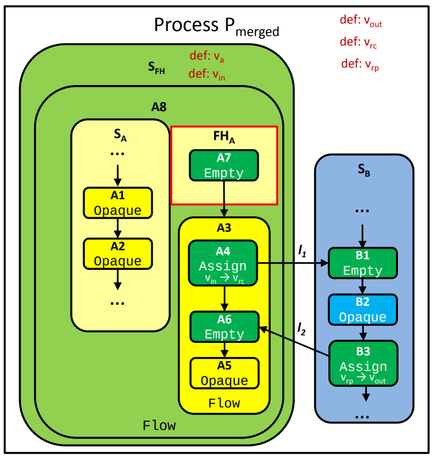

Figure 5. Merged Process Model with outfactored Fault Handler Logic

A scope S is given that has several fault handlers $FH^1{}_S$ to $FH^n{}_S$ represented by `catch` or `catchAll` blocks. After asynchronous and synchronous merges were performed for each fault handler $FH^i{}_S$, it is checked if it contains activities that are target of a link originating from outside of $FH^i{}_S$. If this is the case for at least one fault handler $FH^i{}_S$, a new `scope` S_{FH} is created that contains a `flow` activity. S_{FH} is required to ensure that the activities have still access to the data context of the fault handler they were moved from (see below). The `flow` activity within S_{FH} serves as container for the scope S and its fault handlers. Then for each fault handler $FH^i{}_S$ with an incoming link its root activity $root^i{}_{FH}$ is moved to the scope S_{FH} and replaced by a new `empty` activity $e^i{}_{FH}$. Between $e^i{}_{FH}$ and $root^i{}_{FH}$ a control link is created where $e^i{}_{FH}$ acts as source. This ensures that the fault handling logic contained in $root^i{}_{FH}$ is always performed when $FH^i{}_S$ is activated. All fault handlers that have no incoming control flow link remain unchanged. Figure 5 shows the merged process model P_{merged} where the root activity $A3$ of the fault handler was factored out. The new `empty` activity $A7$ acts as source for the control link pointing to $A3$.

The activities and control flow links that were moved out of the fault handler cannot access the local data context (local variable, partner link declarations etc.) that was either defined in scope S or in the `catch` block anymore as they reside in the parent scope S_{FH}. To make the data context visible again, the defined data have to be moved to the parent scope S_{FH} of the fault handling activities. In Fig. 5 this affects the variables v_a and v_{in}, hence, they are lifted from scope S to S_{FH}.

The solution described above keeps the control flow relations between the (non-communicating) basic activities as defined in the choreography. If S_A in Fig. 5 completes successfully the fault handler FH_A is uninstalled and all links originating from activity $A7$ are marked as dead. Thus, activity $A3$, its child activities and also all activities within S_B are not activated either (dead path elimination). This is the same behavior as modeled in the choreography. In case a fault is thrown and caught by FH_A activity $A7$ is executed and its outgoing links are activated. This causes the former root activity of FH_A $A3$ and its children to be executed. This in turn causes all activities in S_B to be performed. When S_A completes successfully but S_{FH} was not completed yet by the process engine (which usually happens immediately after S_A was completed) and S_FH terminated due to an error in its parent `scope` the default termination handler compensates all successfully executed child `scopes` of S_FH. In case a fault is thrown during the execution of the fault handler FH_A this fault is simply thrown to its parent `scope`. Also this behavior is kept as S_FH does not catch any fault, i. e., it simply rethrows the fault to its parent `scope` that used to be the parent scope `scope` of S_A. This also happens when S_A throws a fault that is not caught by fault handler FH_A.

5 Related Work

Compared to many other techniques that merge processes that are semantically equivalent such as different variants of the same process, we aim to merge collaborating processes. Mendling and Simon [9] propose for instance an approach where semantically equivalent events and functions of Event Driven Process Chains [12] are merged. Küster et al. [5] describe how change logs can be employed to merge different process variants that were created from the same original process.

Instead of directly generating a BPEL orchestration out of a BPEL4Chor choreography, an intermediate format may be used. 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 [7] do not generate BPEL scopes out of Petri nets, even if the formal model of Lohmann [6] generates a Petri net representation of BPEL `scopes`.

6 Conclusion and Outlook

In this work we extended the process consolidation approach presented in [13] and [14]. We have shown, how to isolate the activities of the different partners from each other by using BPEL `scopss` and we also extended the asynchronous and synchronous merge operations to reduce the number of control flow links that may be created during the consolidation operation. The main contribution of this work is a technique to merge process models that interacted via fault handlers before they were merged. To satisfy the constraint that no control links must point into a fault handler we have shown a technique to factor the fault handling activities out of the handler.

In future works we also have to propose a way to merge process models that interact via compensation handlers and event handlers. This is even more challenging as they allow neither inbound nor outbound control flow links. Another issue we have to address is that our current merge operations create process models that violate the *peer-scope-dependency* rule. Basically, this rule states that two `scopes` enclosed within the same parent `scope` must have no cyclic control-flow dependencies, otherwise the compensation order of these `scopes` cannot be determined. However, in practice this rule is not enforced by engines such as the Apache ODE[2] or BPEL-g[3].

[2] `http://ode.apache.org/`

[3] `http://code.google.com/p/bpel-g/`

In this paper, we informally argued that the consolidation approach is correct. A first approach to provide a more formal validation has been presented in [14]. Our ongoing work is to evaluate the Petri net formalizations with respect to formal foundations for our merging approach.

Acknowledgments This work was partially funded by the BMWi project Migrate! (01ME11055) and the BMWi project CloudCycle (01MD11023).

References

1. Barros, A., Dumas, M., ter Hofstede, A.: Service Interaction Patterns. In: BPM. Springer (2005)
2. Decker, G., Kopp, O., Leymann, F., Weske, M.: Interacting services: From specification to execution. Data & Knowledge Engineering 68(10), 946–972 (Apr 2009)
3. Decker, G., et al.: Non-desynchronizable Service Choreographies. In: ISCOC 2008
4. Khalaf, R., Roller, D., Leymann, F.: Revisiting the Behavior of Fault and Compensation Handlers in WS-BPEL. In: OTM 2009
5. Küster, J., Gerth, C., Förster, A., Engels, G.: A Tool for Process Merging in Business-Driven Development. In: Proceedings of the Forum at the CAiSE (2008)
6. Lohmann, N.: A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In: WS-FM'07: Web Services and Formal Methods, 4th International Workshop (2007)
7. Lohmann, N., Kleine, J.: Fully-automatic Translation of Open Workflow Net Models into Simple Abstract BPEL Processes. In: Modellierung. Gesellschaft für Informatik e. V. (2008)
8. Lohmann, N., Kopp, O., Leymann, F., Reisig, W.: Analyzing BPEL4Chor: Verification and Participant Synthesis. In: WS-FM'07: Web Services and Formal Methods, 4th International Workshop (2007)
9. Mendling, J., Simon, C.: Business Process Design by View Integration. In: BPM Workshops. Springer (2006)
10. OASIS: Web Services Business Process Execution Language Version 2.0 – OASIS Standard (2007)
11. Object Management Group (OMG): Business Process Model and Notation (BPMN) Version 2.0 (2011), OMG Document Number: formal/2011-01-03
12. Scheer, A.W., Thomas, O., Adam, O.: Process Aware Information Systems: Bridging People and Software Through Process Technology, chap. Process Modeling Using Event-Driven Process Chains. Wiley-Interscience (2005)
13. Wagner, S., Kopp, O., Leymann, F.: Towards Choreography-based Process Distribution In The Cloud. In: Proceedings of the 2011 IEEE International Conference on Cloud Computing and Intelligence Systems (2011)
14. Wagner, S., Kopp, O., Leymann, F.: Towards Verification of Process Merge Patterns with Allen's Interval Algebra. In: ZEUS. CEUR, Bamberg (2012)