Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart
Germany

Studienarbeit Nr. 2459

# Konsolidierung mittels Event-Handler kommunizierender BPEL-Prozesse

Aleksandar Milutinovic

| | |
|---|---|
| **Studiengang:** | Informatik |
| **Prüfer:** | Prof. Dr. Frank Leymann |
| **Betreuer:** | Dipl.-Inf. Sebastian Wagner |
| **begonnen am:** | 07.01.2014 |
| **beendet am:** | 09.07.2014 |
| **CR-Klassifikation:** | H.4.1 |

## Kurzfassung

Unternehmen, die in einer Choreographie miteinander kommunizieren, sind zur Organisation einer einheitlichen Ablauforganisation verpflichtet. Dazu werden die Geschäftsprozesse der Unternehmen konsolidiert, wobei auch Fehlerbehandlungskonstrukte wie Event-Handler konsolidiert werden müssen [BER13]. Diese Arbeit ergänzt die Untersuchungen zur Konsolidierung von interagierenden BPEL-Prozessen von Berger [BER13] und Debicki [DEB13] um die Umwandlung von mittels Event-Handlern interagierender Prozesse innerhalb einer Choreographie. Dabei werden verschiedene Interaktionsszenarien der Choreographien betrachtet und im Hinblick auf die notwendige Nachbildung des Event-Handler-Verhaltens untersucht. Bei der Konsolidierung der Prozessmodelle mit beteiligten Event-Handlern können Grenzüberschreitungen von Kontrollfluss-Links entstehen. Durch die Nachbildung des Verhaltens der Event-Handler können diese auftretenden Grenzüberschreitungen aufgelöst oder vermieden werden. Es wird ebenfalls untersucht, wie detailgetreu das Verhalten eines Event-Handlers im konsolidierten Prozess mittels BPEL-Sprachkonstrukten emuliert werden kann. Einschränkungen und Unterschiede des emulierten Event-Handlers im Vergleich zum Event-Handler selbst werden hierbei ebenfalls genauer Betrachtet und zusammenfassend dargestellt.

# Inhaltsverzeichnis

## Abbildungsverzeichnis

## Abkürzungsverzeichnis

BPEL ..... Business Process Execution Language

BPEL4CHOR ..... Business Process Execution Language for Choreographies

EH ..... Event-Handler

FH ..... Fault-Handler

IAAS ..... Institut für Architektur von Anwendungssystemen

NMML ..... Non-Mergeable-Message-Link

OMG ..... Object Management Group

PBD ..... Participant Behavior Description

TH ..... Termination-Handler

WS-BPEL ..... Webservices Business Process Execution Language

XML ..... Extensible Markup Language

XPATH ..... XML Language Path

## Verzeichnis der Listings

# 1 Einleitung

Diese Studienarbeit wurde als weiterführende Arbeit zu [WKL11] und den Diplomarbeiten von Berger [BER13], Debicki [DEB13] und Cui [CUI12] verfasst und beschäftigt sich intensiver mit der Konsolidierung von Event-Handlern (nachfolgend auch als EH abgekürzt) in BPEL4Chor. In den genannten Arbeiten wurde schon grundlegend die Möglichkeit der Konsolidierung von interagierenden Prozessen innerhalb einer Choreographie untersucht. Hierbei wurden bereits interagierende Prozesse in [DEB13] betrachtet und im Speziellen Fehlerbehandlungskonstrukte wie Fault-, Compensation-, und Termination-Handler [BER13] untersucht. Aus der Diplomarbeit von Berger [BER13] geht allerdings nur ein Vorschlag hervor, wie Event-Handler möglicherweise konsolidiert werden können. Dieser Vorschlag wird in der vorliegenden Arbeit aufgegriffen und weitergehend untersucht. Dazu werden die verschiedenen Interaktionsszenarien von mittels Event-Handler kommunizierender Prozesse innerhalb von Choreographien betrachtet und deren Konsolidierung anhand von Schemata graphisch und theoretisch durchgespielt.

**Gliederung**

Die Gliederung dieser Arbeit ergibt sich wie folgt:

**Kapitel 2 – Grundlagen**: In diesem Kapitel werden die grundlegenden Informationen zu WS-BPEL 2.0 und den verwendeten Ausdrücken erläutert. Der Fokus liegt hierbei auf den verwendeten Ausdrücken, die für diese Arbeit notwendig sind. Anschließend wird die Motivation zur Konsolidierung von interagierenden BPEL Prozessen erläutert und anhand eines Beispiels aufgezeigt.

**Kapitel 3 – Konsolidierung von Prozessen mit Event-Handler**: Dieses Kapitel bildet den Kern der Arbeit. Nach vorausgegangener Motivation und Erläuterung der Konsolidierung von interagierenden Prozessen in Kapitel 2, wird hier ausschließlich auf die Konsolidierung von interagierenden Prozessen im Hinblick auf vorhandene Event-Handler eingegangen. Dabei wird untersucht, ob Prozesse mit Event-Handler konsolidiert werden können. Zudem wird auch betrachtet, welche Einschränkungen und Unterschiede des konsolidierten Prozesses mit emuliertem Event-Handler zum Konstrukt des Event-Handlers selbst bestehen.

**Kapitel 4 – Implementation**: Dieses Kapitel beschreibt die Implementation der gefundenen Konsolidierungsansätze aus dem vorangegangenen Kapitel im Choreography Merger Tool am IAAS.

**Kapitel 5 – Zusammenfassung und Ausblick**: Hier wird das Abschlussergebnis der Konsolidierung von Event-Handlern dargestellt und ein Ausblick auf noch mögliche zukünftige Konsolidierungsarbeiten gegeben.

# 2 Grundlagen

## 2.1 WS-BPEL 2.0 Grundlagen

Dieses Kapitel soll eine kurze Übersicht über die eingesetzte Sprache WS-BPEL 2.0 bieten, um dem Leser einen schnellen Einstieg in das Thema zu ermöglichen. Dabei werden nachfolgend der Aufruf und die Funktion der in dieser Arbeit verwendeten BPEL-Ausdrücke erläutert. Ebenfalls wird auf die Besonderheiten der für diese Arbeit relevanten Ausdrücke detaillierter eingegangen. Für ausführlichere Beschreibungen und weiterführende Informationen zu Konstrukten in WS-BPEL 2.0 wird auf die Spezifikation der OASIS (Organization for the Advancement of Structured Information Standards) [OAS07] verwiesen.

Für die Modellierung der BPEL-Prozesse wurde die Entwicklungsumgebung eclipse Kepler mit installiertem BPEL-Designer Plugin [ECL14] verwendet. Als Workflow-Engine kam Apache ODE 1.3.6 (Orchestration Director Engine) [AODE11] im Apache Tomcat 7.0.53 Server [ATOM14] zum Einsatz.

### 2.1.1 Process

Mittels `<process>` wird der Beginn eines BPEL-Prozesses markiert. Der Ausdruck umschließt den gesamten modellierten Prozess und kann nicht mehrfach verwendet werden.

### 2.1.2 Scope

Ein `<scope>` stellt einen Container für weitere Sprachkonstrukte dar. Im simpelsten Fall werden Aktivitäten in diesem `<scope>` sequentiell ausgeführt. Der `<scope>` bietet auch eine Möglichkeit, die Ausführung der eingebetteten Aktivitäten an die Lebensdauer des `<scope>` zu binden. Darüber hinaus besitzt ein `<scope>` weitere Fehlerbehandlungsmöglichkeiten wie Fault-, Compensation-, Termination-Handler, sowie die Ausnahmebehandlungsmöglichkeit über einen EH.

Die Lebensdauer hängt vom jeweiligen Zustand des Scopes ab. Wie Abbildung 1 vereinfacht zeigt, gibt es mehrere Zustände, in denen sich der `<scope>` befinden kann. Die für diese Arbeit untersuchten und wesentlichen Zustände sind: *Inactive*, *Running*, *Finished*, *Failed* und *Terminated*.

*Inactive*: Der `<scope>` ist inaktiv und zur Ausführung bereit.

*Running*: Der `<scope>` befindet sich in einem aktiven Zustand und wird ausgeführt.

*Finished*: Alle Aktivitäten des `<scope>` wurden erfolgreich beendet und es existiert keine weitere Aktivität oder kein weiteres Konstrukt, welches noch auf eine Ausführung wartet.

*Terminated*: Dieser zustand wird erreicht, wenn ein laufender `<scope>` unterbrochen werden muss. Beispielsweise durch eine `<exit>`-Aktivität oder wenn der Parent-`<scope>` durch einen Fehler beendet wird und damit der Child-`<scope>` terminiert.

*Failed*: Es ist ein Fehler bei der Ausführung aufgetreten und die Ausführung wurde mit einem Fehler gestoppt.

Die nachfolgende Abbildung stellt in einem simplifizierten Zustandsdiagramm die Abhängigkeiten und vereinfachten möglichen Zustände eines Scopes dar.

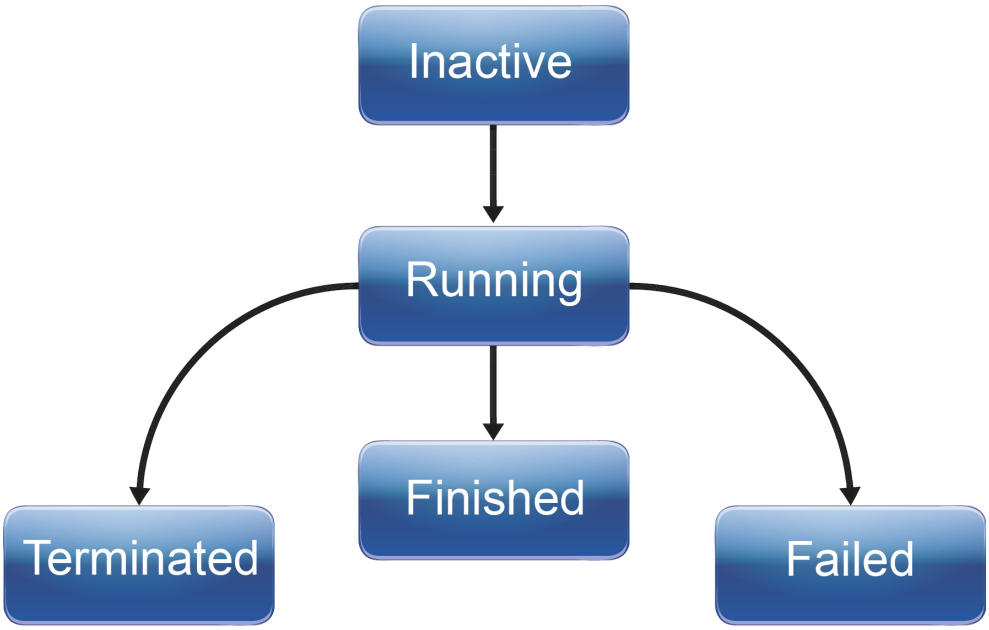


**Abbildung 1 - Mögliche Zustände eines BPEL-Scopes, vereinfacht**

### 2.1.3 Sequence

Der Ausdruck `<sequence>` beschreibt eine Folge von Aktivitäten innerhalb einer Sequenz, die nacheinander ausgeführt werden. Die `<sequence>` wird beendet, wenn die letzte Aktivität in der Folge beendet wurde.

### 2.1.4 Flow

Der Ausdruck `<flow>` wird verwendet, um Nebenläufigkeit in BPEL zu realisieren. Dabei werden Scopes innerhalb eines `<flow>` parallel ausgeführt.

### 2.1.5 Pick

Die `<pick>`-Aktivität dient dazu, abhängig von eingehenden Messages oder `<onAlarm>` Events dazugehörige Sequenzen zu starten. Dabei muss jede `<pick>`-Aktivität mindestens über ein `<onMessage>`-Konstrukt verfügen. Das `<pick>` wird erst beendet, wenn mindestens ein Event aufgetreten ist und die Sequenz des Events beendet wurde. Sollte die `<pick>`-Aktivität innerhalb eines `<scope>` laufen, so wird auch der `<scope>` erst beendet, wenn die `<pick>`-Aktivität beendet wurde.

### 2.1.6 Receive

Der Ausdruck `<receive>` empfängt Messages im Prozess und ist daher für den Empfang von Nutzdaten, die in einer Variablen gespeichert werden, relevant.

### 2.1.7 Reply

Mittels `<reply>` wird eine Message aus dem Prozess heraus als Antwort an den entsprechenden Partner versendet und der Inhalt einer Variablen übertragen.

### 2.1.8 Throw

Über `<throw>` kann ein beliebiger Fehler geworfen werden. Diese Aktivität ist besonders interessant, da während der Emulation von EH auch benutzerdefinierte Fehler geworfen werden müssen, um die Lebensdauer des EH nachzubilden. Dabei werden die benutzerdefinierten Fehler gesondert abgefangen.

### 2.1.9 Empty

Es besteht manchmal die Notwendigkeit die leere Aktivität `<empty>` auszuführen. Dies ist beispielsweise der Fall, wenn ein Fehler abgefangen und unterdrückt wird. Ein anderer Anwendungsfall ist das Verwenden von `<empty>` als Synchronisationspunkt innerhalb eines `<flow>`.

### 2.1.10 Assign

Die `<assign>`-Aktivität wird verwendet, um den Inhalt von Variablen mittels `<copy>` einander zuzuweisen. Dabei können die Variablen atomar zugewiesen werden – beispielsweise durch mehrere `<copy>`-Knoten, oder durch einen komplexen Ausdruck (XPATH).

### 2.1.11 Control Link

Kontrollfluss-Links werden mittels `<targets>` und `<sources>` eindeutig zwischen zwei Punkten modelliert. Dies sind Standardelemente und können in jedem Konstrukt zu Beginn definiert werden. In der Regel wird dadurch eine Aktivität getriggert, die sich außerhalb einer Sequenz befindet oder deren Ausführung an bestimmte Bedingungen geknüpft ist.

**Link Semantik:**

Ein Link hat drei mögliche Zustände: *undefiniert*, *wahr* und *falsch*. Ein Link befindet sich so lange im Zustand *undefiniert,* bis er evaluiert wurde, das bedeutet: bis die mit dem Link verknüpfte Bedingung überprüft und diese dann als *wahr* oder *falsch* evaluiert wurde. Dementsprechend ändert sich auch der Zustand des Links. Sollten mehrere Links auf eine Aktivität zeigen, so wird diese erst ausgeführt, wenn alle Links zu *wahr* evaluiert wurden. Damit sollen sogenannte Race-Conditions – also das mehrmalige Ausführen einer Aktivität bei nicht gleichzeitiger Link-Ausführung – verhindert werden.

### 2.1.12 Event-Handler

Da der Umgang mit Event-Handlern den Kern dieser Arbeit bildet, wird dieses Konstrukt nachfolgend etwas genauer betrachtet.

**Listing 2.1** – Beschreibung des Event-Handlers in WS-BPEL [OAS07]

```
<eventHandlers>
   <onEvent partnerLink="travelAgency"
      portType="ns:agent"
      operation="travelUpdate"
      messageType="ns:travelStatsUpdate"
      variable="travelUpdate">
      <correlations>
         <correlation set="travelCode" initialize="no" />
         <correlation set="updateCode" initialize="yes" />
      </correlations>
      <scope name="EH-Scope">
         ...
         <correlationSets>
         <correlationSet name="updateCode"
               properties="ns:updateCode" />
         </correlationSets>
         ...
      </scope>
   </onEvent>
</eventHandlers>
```

Die erste Child-Aktivität innerhalb von `<onEvent>` beziehungsweise `<onAlarm>` des EH muss und darf exakt nur ein einzelner `<scope>` sein. Darüber hinaus muss der EH über mindestens ein `<onEvent>` oder `<onAlarm>` Element verfügen. Über `<onEvent>` wird der Empfang einer Message durch den EH ermöglicht, wodurch dann nach Erhalt dieser eine neue Instanz des EH erzeugt und der `<scope>` innerhalb des `<onEvent>` ausgeführt wird. Bei `<onAlarm>` wird nach Ablauf eines Timers oder durch einen definierten Zeitpunkt ebenfalls der EH instanziiert und der <scope> innerhalb des <onAlarm> ausgeführt.

Die Lebensdauer des EH hängt direkt vom zugehörigen Scope, an den der EH angeheftet wird, ab. Sobald dieser aktiv ist, ist auch der EH aktiv und wartet auf den Empfang einer Message über `<onEvent>` oder den Eintritt des `<onAlarm>`-Events.

Sobald eine Instanz des EH erzeugt wurde, darf diese auch completen, auch wenn der zugehörige Scope bereits erfolgreich completed wurde und damit nicht mehr aktiv ist. Sollte der zugehörige Scope des EH jedoch terminieren oder durch einen Fehler beendet werden, wird auch der zugehörige Zustand unmittelbar an den EH-Scope weitergegeben und die Ausführung beendet.

### 2.1.13 Fault-Handler

Der Fault-Handler führt die Aktivitäten innerhalb des Handlers aus, sobald der zugehörige `<scope>` in den Zustand eines Fehlers kommt.

### 2.1.14 Termination-Handler

Der Termination-Handler führt die Aktivitäten innerhalb des Handlers aus, sobald der zugehörige `<scope>` in den Zustand *terminating* kommt.

### 2.1.15 ForEach

Der Scope innerhalb eines `<forEach>` wird mehrmals ausgeführt (1..N Iterationen). Dabei kann gewählt werden, ob die Ausführung parallel oder sequentiell stattfindet. Bei paralleler Ausführung werden sofort alle Iterationen gleichzeitig gestartet.

### 2.1.16 RepeatUntil

Mittels `<repeatUntil>` wird eine Schleife realisiert, dessen Inhalt so lange ausgeführt wird, bis die Schleifenbedingung innerhalb des `<repeatUntil>` zu *wahr* evaluiert wird. Da die Bedingung erst am Ende der Schleife geprüft wird, wird der Schleifeninhalt mindestens ein Mal ausgeführt (1..N Iterationen).

### 2.1.17 While

Im Gegensatz zu `<repeatUntil>` ist die Ausführung des Schleifeninhalts bei `<while>` an die vorherige Evaluierung der Schleifenbedingung zu *wahr* geknüpft. Die Schleife wird solange ausgeführt, bis die Schleifenbedingung zu *false* evaluiert wird. Im Falle der Evaluierung der Schleifenbedingung zu false wird nichts ausgeführt (0..N Iterationen).

### 2.1.18 Wait

Die `<wait>`-Aktivität wird verwendet, wenn vor der weiteren Ausführung eine bestimmte Dauer abgewartet oder ein bestimmter Zeitpunkt erreicht werden soll. Dabei kann eine `<wait>`-Aktivität jeweils nur genau eine dieser Optionen abbilden.

## 2.2 Konsolidierung von interagierenden BPEL Prozessen

Unter der Konsolidierung von interagierenden BPEL Prozessen versteht man das Zusammenführen (Merge) der interagierenden Prozesse einer Choreographie zu einem einzelnen Prozess [BER13]. Dabei wird angenommen, dass die Prozesse der Choreographie nur einmal instanziiert werden dürfen (One-to-One Interaktion). One-to-Many Interaktionen können zurzeit nicht konsolidiert werden [WAG13].

Das grundsätzliche Vorgehen bei der Konsolidierung besteht aus fünf Schritten [WKL14]:

1. *Erstellen eines neuen Prozess-Modells* $P_{Merged}$

   Der neue Prozess $P_{Merged}$ agiert hier als Container für alle zu konsolidierenden Prozesse. Dabei werden alle zu konsolidierenden Aktivitäten in diesen Container abgelegt.

2. *Erstellen der Container für die beteiligten Prozesse*

   Der Inhalt jedes zu konsolidierenden Prozesses wird in einen Scope kopiert. Dadurch wird die Isolation gegenüber den anderen zu konsolidierenden Prozessen erreicht. Anschließend werden die Kontrollfluss-Relationen untersucht. Hierbei werden die Kontrollflüsse analysiert, um dann im nächsten Schritt das Konsolidierungsvorgehen zu erstellen.

3. *Umwandeln der Message-Links in Kontrollfluss-Links*

   In diesem Schritt werden die Kommunikationsaktivitäten umgewandelt. Dabei werden Message-Links in Kontrollfluss-Links umgewandelt und die Aktivitäten Invoke und Receive in Assign beziehungsweise Empty umgewandelt. Der Austausch der Informationen, die sonst mit der Message gesendet würden, erfolgt dann durch das Kopieren in eine Variable mittels Assign.

4. *Beheben der Verstöße gegen die BPEL-Spezifikation*

   Grenzüberschreitungen der Kontrollfluss-Links werden analysiert und aufgelöst. Diese können beispielsweise entstehen, wenn Message-Links aus einem EH in einen zu konsolidierenden Prozess zeigen. Dies wird nach der Konsolidierung problematisch, da die nun zu Kontrollfluss-Links umgewandelten Links außerhalb des EH-Scopes zeigen. Dies ist allerdings per Spezifikation nicht erlaubt.

5. *Anpassen des Datenflusses*

   Durch Anwenden verschiedener Konsolidierungsmuster kann es vorkommen, dass die Sichtbarkeit von Variablen zwischen einzelnen Scopes nicht mehr gegeben ist. In diesem Schritt wird dies beispielsweise durch globalisieren der Variablen behoben, sodass die Variablen wieder für die entsprechenden Scopes sichtbar sind.

Nachfolgend ist der dritte Schritt sowohl für den asynchronen, als auch den synchronen Fall in Abbildung 2 und Abbildung 3 illustriert. Dabei muss beim synchronen Fall nicht nur das Invoke getauscht werden, sondern im Anschluss auch ein Empty eingefügt werden, welches als Synchronisationspunkt für den Empfang der Message benötigt wird, damit sich die Ausführungsreihenfolge des Prozesses nicht ändert. Dadurch wird vermieden, dass B2 direkt nach B1 ausgeführt wird, obwohl die Antwort noch nicht empfangen wurde und somit der ursprüngliche Kontrollfluss sichergestellt.

**Konsolidierung asynchroner Fall:**

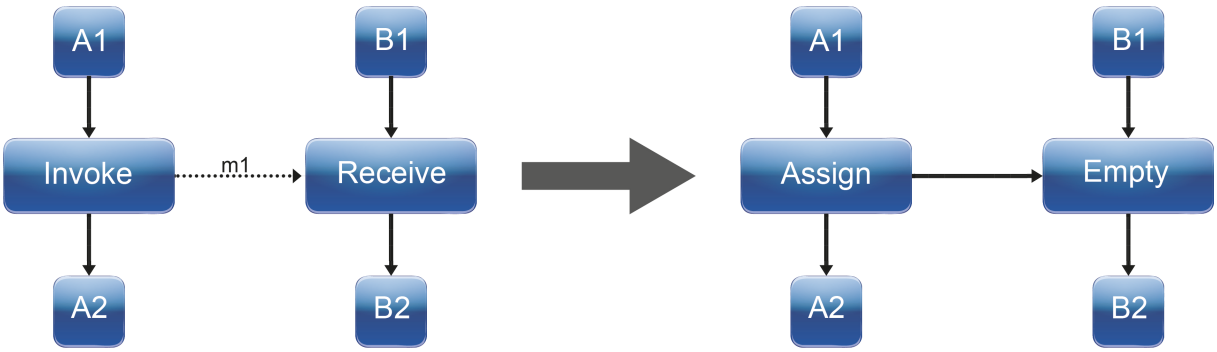


**Abbildung 2 - Asynchrone Konsolidierung von Prozessen, adaptiert aus [WKL11]**

**Konsolidierung synchroner Fall:**

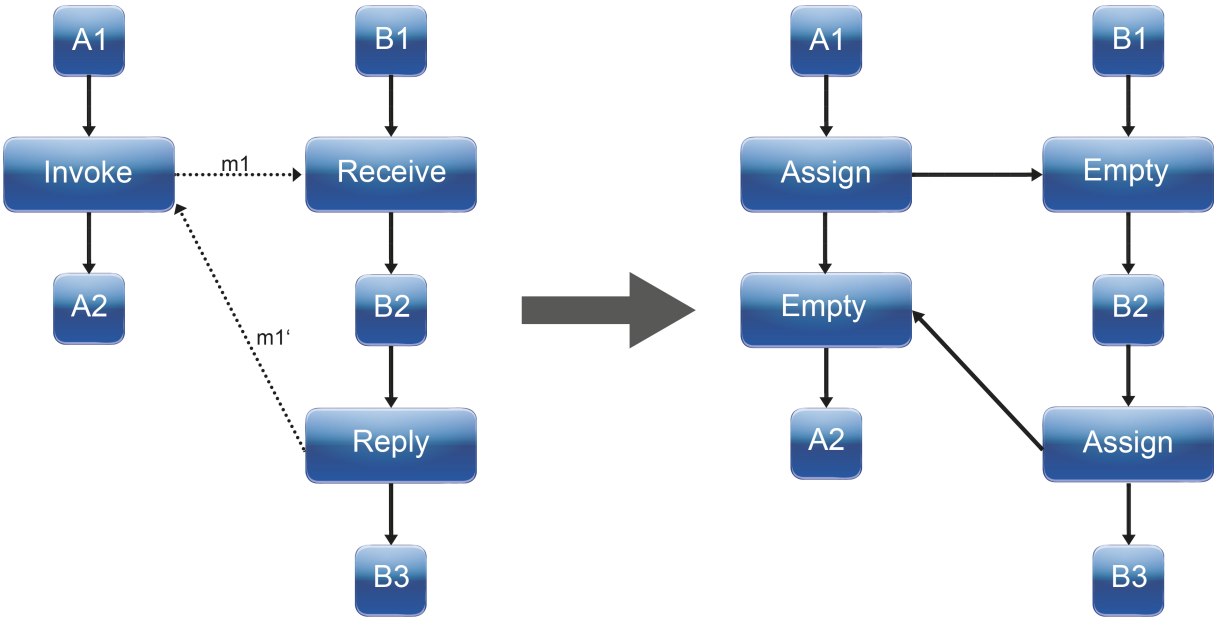


**Abbildung 3 - Synchrone Konsolidierung von Prozessen, adaptiert aus [WKL11]**

# 3 Konsolidierung von Prozessen mit Event-Handler

Wendet man den in Abschnitt 2.2 beschriebenen Ansatz zur Konsolidierung von Prozessen auch bei solchen an die einen EH beinhalten, so werden Message-Links, die in und/oder aus dem Scope eines EH zeigen, in entsprechende Kontrollfluss-Links umgewandelt. Dies widerspricht allerdings der BPEL-Spezifikation und stellt eine Grenzüberschreitung von Links dar. Die nachfolgende Abbildung 4 aus der Arbeit von Berger [BER13] veranschaulicht dieses Problem schematisch:

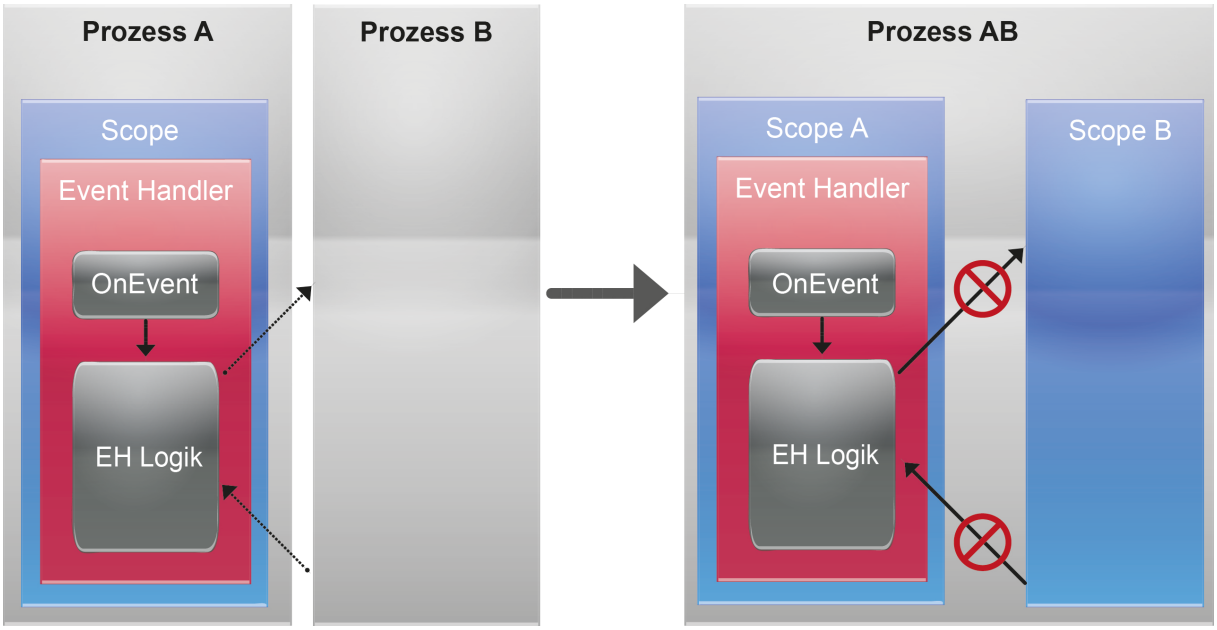


**Abbildung 4 - Beispiel Konsolidierung für EH, adaptiert aus [BER13]**

Es muss also nach einer anderen Lösung gesucht werden, sodass keine Grenzüberschreitung von Links nach der Konsolidierung der beiden Prozesse entsteht.

## 3.1 Aufrufszenarien durch Prozesse mit EH

Das Ziel ist, das Verhalten des EH mit vorhandenen Sprachkonstrukten in BPEL nachzubauen, beziehungsweise anzunähern und somit den Einsatz eines EH vermeiden zu können. Da die Message-Links durch die Konsolidierung in Kontrollfluss-Links umgewandelt werden, ist der Fall von herauszeigenden Links praktisch unvermeidbar und muss dementsprechend nachgebildet oder durch eine andere Verwendung von Sprachkonstrukten vermieden werden.

Um zwei miteinander interagierende Prozesse erfolgreich zusammenzuführen, ist es daher notwendig, den Einsatz von EH durch andere Sprachkonstrukte zu emulieren.

Je nach Szenario der interagierenden Prozesse, sind die Aktionen der Mergeoperation um die Prozesse zusammenzuführen unterschiedlich und jeweils anzupassen. Im Folgenden werden die Szenarien im Einzelnen dargestellt und die resultierende Mergeoperation analysiert.

### 3.1.1 Aufruf des EH durch einen Prozess innerhalb einer Choreographie

In diesem Fall wird der EH durch eine Message von Prozess B an Prozess A instanziiert. Um die Prozesse A und B aus einer Choreographie zusammenzuführen, müsste aus dem Invoke aus Prozess B ein Assign werden. Zudem sollte das Assign mit einem Kontrollfluss-Link auf den Ausführungsteil des EH zeigen. Wie allerdings zu Beginn des Kapitels gezeigt, dürfen keine Links in und aus dem EH zeigen. Somit muss eine andere Lösung in Betracht gezogen werden.

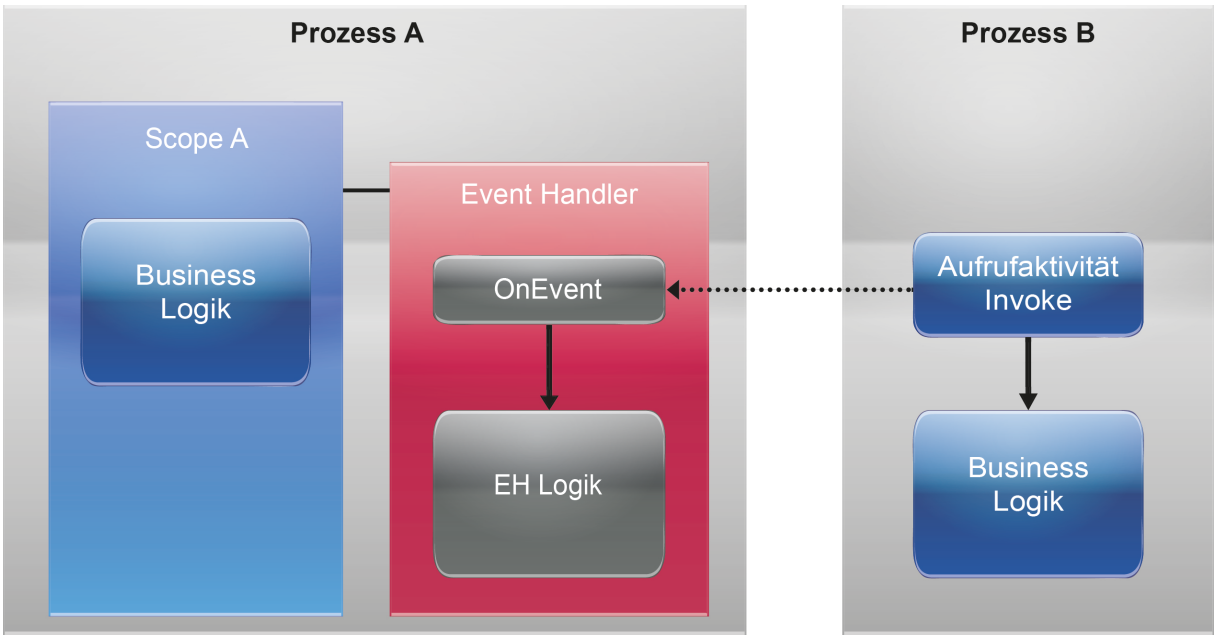


**Abbildung 5 - Aufrufbeispiel EH mit zwei Prozessen**

Eine Lösung für diesen Fall wäre, den Teil der EH-Logik mit in den konsolidierten Prozess zu kopieren, sodass dieser über einen Link ausgeführt werden kann sobald die entsprechende Aktivität aus Prozess B ausgeführt wird. In Abbildung 6 wird diese Lösung stark vereinfacht dargestellt:

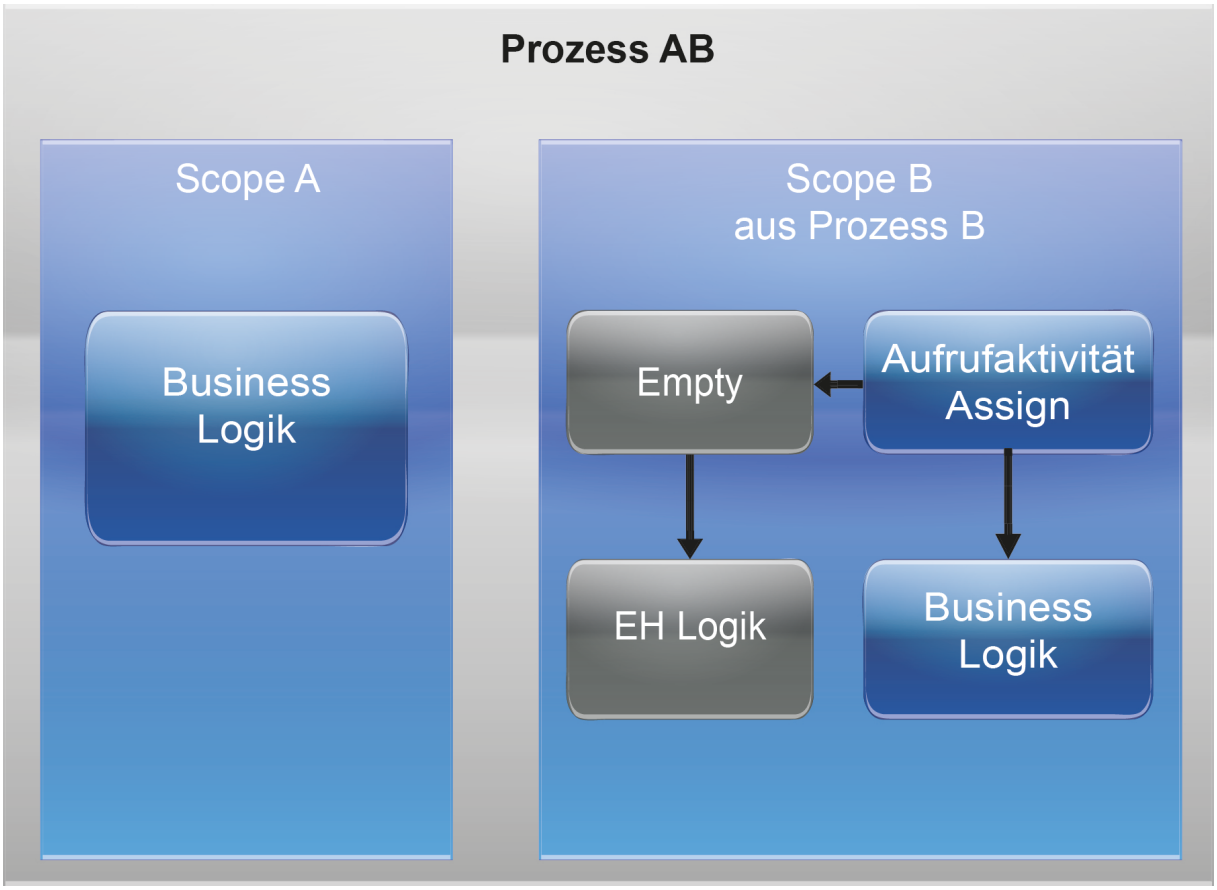


**Abbildung 6 - Aufrufbeispiel EH mit zwei Prozessen konsolidiert - vereinfacht**

Nach erfolgter Konsolidierung wurde der Prozess B in einen weiteren Scope innerhalb des konsolidierten Prozesses AB verschoben. Um die Instanziierung des EH zu emulieren, wurde die Logik des EH mit in den Scope kopiert. Aus dem Invoke wurde, wie in Abschnitt 3.1, gezeigt ein Assign. Von diesem Assign zeigt nun ein Link auf das Empty als Synchronisationspunkt um die Ausführungsreihenfolge beizubehalten, welches dann die Ausführung der EH-Logik startet.

Ebenfalls wie beim EH ist auch hier im emulierten Fall die Bedingung erfüllt, dass der instanziierte EH bis zur completion ausgeführt werden können muss. Da sich der Logik-Teil des EH nun im Scope B befindet, kann dieser auch problemlos bis zum Ende ausgeführt werden, denn spätestens dann wird der Scope B auch als *finished* gesetzt und kann beendet werden. Weitere Betrachtungen der Lebensdauer des EH-Scopes folgen in Abschnitt 3.1.1.2.

#### 3.1.1.1 Betrachtung des Datenflusses

Der EH kann auf alle Variablen und Daten des Scopes zugreifen, an den er angefügt ist. Durch das Kopieren des EH-Scopes innerhalb des konsolidierten Prozessmodells läuft der EH-Scope parallel zum Scope mit der Ausführungslogik, nachfolgend auch als Scope A referenziert, des zugehörigen Prozesses innerhalb eines Flows. Dadurch kann der EH-Scope nicht mehr auf die benötigten Variablen zugreifen. Um dieses Problem zu lösen, ist es notwendig die Variablen in den gemeinsamen Parent-Scope des EH- und Prozess-Scopes zu verschieben. Durch die Vererbung haben dann wieder beide Scopes Zugriff auf die Daten.

#### 3.1.1.2 Lebensdauer des EH und des zugehörigen Scopes

Um den EH komplett abzubilden und die Eigenschaften zu emulieren, sollte auch die Lebensdauer des EH betrachtet werden. Grundsätzlich ist es so, dass der EH zusammen mit dem zugehörigen Scope aktiv und an dessen Lebensdauer gekoppelt ist. In den möglichen Zuständen des Scopes (vgl. Abschnitt 2.1.2) *running*, *finished*, *failed*, *terminated* stellt sich der EH wie folgt dar:

- *Running*: Der Scope A, der den EH beinhaltet ist aktiv und läuft. Der EH ist aktiv und wartet auf ein Event oder Alarm, oder läuft bereits

- *Finished*: Der Scope A ist bereits beendet. Damit wird der EH inaktiv, allerdings ist es so, dass ein bereits gestarteter EH completen können muss.

- *Terminated*: Der Scope A wurde terminiert, dadurch der EH inaktiv. Ebenfalls wird auch die eventuelle Ausführung des EH terminiert.

- *Failed*: Der Scope A wurde mit einem Fehler beendet, der Fehler wird an einen ausgeführten EH weiter gegeben und dieser wird ebenfalls beendet.

Damit ist ersichtlich, dass der EH-Logik-Teil im konsolidierten Prozess AB (vgl. Abbildung 6) von einen weiteren Scope umschlossen werden muss, damit die Lebensdauer entsprechend emuliert werden kann. Die eben erwähnten vier Zustände bedürfen dann einer weiteren Anpassung:

- *Running*: Scope A läuft und der EH-Scope ist ebenfalls aktiv. Hier besteht auch beim konsolidierten Prozessmodell keine Einschränkung, da beide Scopes innerhalb eines Flows parallel (und möglicherweise mehrfach) laufen können.

- *Completed*: Scope A ist bereits beendet. Der EH-Scope kann innerhalb des Flows ebenfalls komplett durchlaufen, da der Flow erst beendet wird, wenn alle Scopes innerhalb beendet sind.

- *Terminated*: Scope A terminiert. Dies stellt den ersten Sonderfall dar, da der EH-Scope innerhalb des Flows bisher keine Information erhalten hat, dass der aufrufende Scope A terminiert. Dies kann mit Hilfe eines Termination-Handlers gelöst werden. Scope A erhält einen Termination-Handler, welcher wiederum mittels Throw einen Fehler innerhalb des EH-Scopes wirft, sodass dieser terminiert wird. Der EH-Scope befindet sich dann zwar nicht im Zustand *terminated*, dennoch wird die Ausführung gestoppt und die Lebensdauer des EH damit entsprechend emuliert.

- *Failed*: Der Fehler wird über einen Fault-Handler angefangen und wie im Beispiel beim Termination-Handler an den EH-Scope weiter gegeben, sodass dieser auch in den Zustand *failed* übergeht und damit beendet wird.

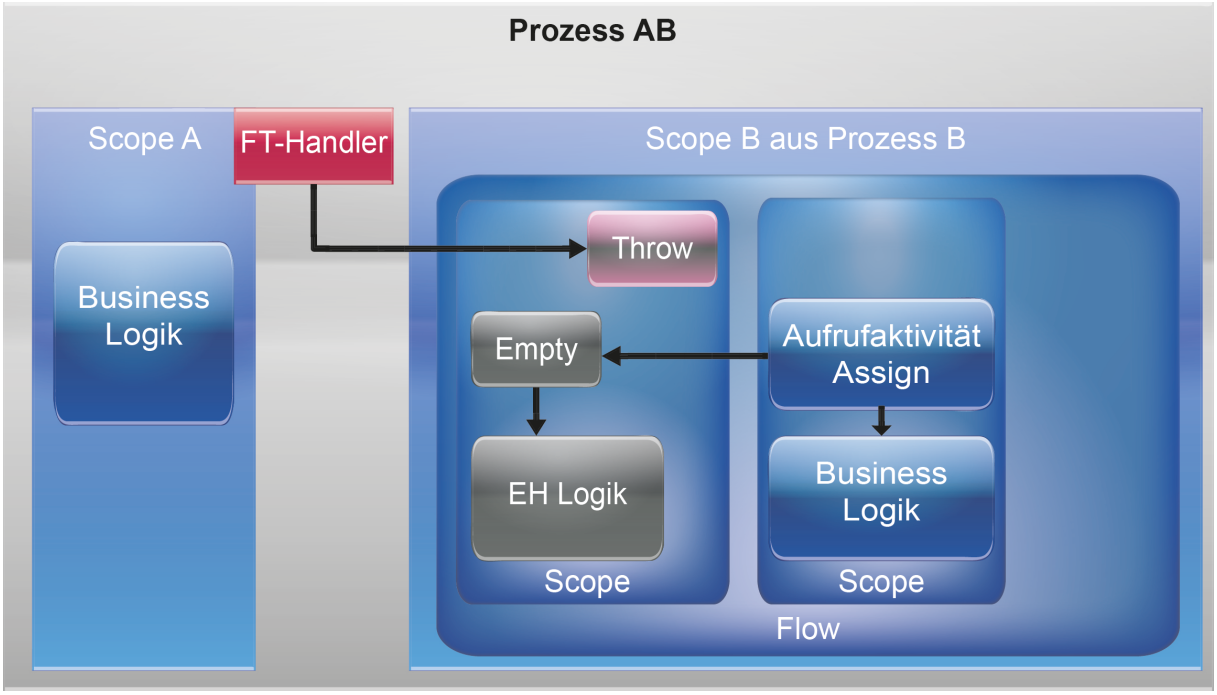


**Abbildung 7 - Aufrufbeispiel EH mit zwei Prozessen konsolidiert - Lebensdauer Scope**

Wie man in obiger Abbildung erkennen kann, ist das Schema nun erweitert worden, um die Lebensdauer des emulierten EH abzubilden. Dabei überwacht jeweils der Fault- und Termination-Handler (in Abbildung mit FT-Handler vereinfacht) den Zustand des Scopes A. Sobald dieser in einen der beiden Zustände *failed* oder *terminated* gelangt, wird der zugehörige EH-Scope über das Werfen eines benutzerdefinierten Fehlers über Throw ebenfalls beendet. Eine Einschränkung hierbei ist, dass für den EH-Scope zwischen den Zuständen *failed* und *terminated* nicht unterschieden wird. Um allerdings die Ausführungssemantik nachzubilden, reicht es, wenn der EH-Scope gemeinsam mit Scope A beendet wird. Dadurch wurde eine Lösung für den Zustandswechsel der Scopes, wie auf Seite 78 aus [BER13] angemerkt, gefunden. Der Fault-, bzw. Termination-Handler erzeugt dabei keine Grenzüberschreitungen von Links (Cross-Boundary-Links) [BER13], sodass diese nicht gesondert aufgelöst werden müssen, da sich alle neu entstehenden Links innerhalb des konsolidierten Prozesses befinden.

Auch hier ist eine weitere Einschränkung beim Timingverhalten im Bezug auf die gleichzeitige Aktivierung der Scopes A und B zu erwarten. So kann Scope B vor Scope A aktiviert werden, sodass korrekt betrachtet der emulierte EH aktiv ist, bevor der ursprünglich angehängte Scope A aktiv ist, was per Spezifikation so nicht möglich ist.

### 3.1.1.3 Mehrfache Aufrufe des EH

Im Fallbeispiel aus Abschnitt 3.1.2 wird davon ausgegangen, dass die Aufrufaktivität (vgl. Abbildung 9) aus Prozess B nur eine Message an den EH sendet und damit nur eine einzige Instanz erzeugt. Angenommen Prozess B kann die Message mehrfach senden, so kann dies auf unterschiedliche Weise der Fall sein:

- Schleifenkonstrukte: parallel mittels forEach oder seriell über While / repeatUntil

- Parallele Pfade mit mehreren Invoke-Aktivitäten

#### 3.1.1.4 Schleifenkonstrukte

Die Multi-Instance-Eigenschaft des EH muss gegeben sein, damit dieser potentiell mehrfach instanziiert werden kann. Hierbei ist es daher wichtig zu wissen, wie mehrfache Messages an den EH gesendet werden. In diesem Abschnitt wird der Fall betrachtet, wenn sich die Aufrufaktivität innerhalb einer Schleife befindet. Es gibt vier verschiedene Schleifenkonstrukte, welche die Aufrufaktivität beinhalten können:

- *While*
- *RepeatUntil*
- *Serielles forEach*
- *Paralleles forEach*

Bei allen vier Konstrukten besteht die Möglichkeit, dass zum Start der Laufzeit die Anzahl der Ausführungen unbekannt ist. Eventuell kann sich auch die Anzahl der Aufrufe durch die Instanziierung eines EH verändern, wobei wir hier Endlosschleifen von der Betrachtung ausschließen.

Eine Möglichkeit dieses Problem der unbekannten Anzahl der Aufrufe zu lösen könnte sein, den EH-Logik-Scope mit in das Schleifenkonstrukt zu kopieren. Dabei ist das Timingverhalten des emulierten EH zu beachten: Im Falle des eigentlichen EH wird bei jeder ankommenden Message eine Instanz des EH erzeugt, die dann parallel zu den bereits aktiven Instanzen ausgeführt wird. Bei Schleifenkonstrukten kommt es bei serieller Ausführung dazu, dass jede Iteration des in den Schleifenblock kopierten Logik-Teil des EH erst beendet sein muss, bevor die nächste Iteration gestartet werden kann. Somit verlängert sich die Dauer der Ausführung.

Während vor der Konsolidierung im Schleifenblock bei sequentieller Ausführung bei jeder Iteration eine Message an den EH verschickt wird und damit auch eine Instanz, welche parallel zu den anderen läuft erzeugt wurde, muss nun bei jedem Durchlauf auch gewartet werden, bis die EH-Logik komplett ausgeführt wurde, bevor die nächste Iteration ausgeführt wird. Dies führt dazu, dass bei gleichbleibender Ausführungszeit die Ausführungsdauer verlängert wird, da die Emulation der EH-Instanzen unter Umständen nicht parallel ablaufen kann.

Um die Eigenschaft der möglichen parallel laufenden Instanzen des EH so genau wie möglich nachzubilden, muss der EH-Scope innerhalb eines parallelen forEach laufen. Dies ist allerdings nur dann möglich, wenn zu Beginn der Laufzeit die exakte Anzahl der Iterationen bekannt ist und sich diese nicht ändert, oder sich die Aufrufaktivität in einem parallelen forEach befindet. Der Grund hierfür ist, dass bei Aufruf des parallelen forEach alle Iterationen parallel gestartet werden und die entsprechenden Variablen für die Bestimmung der Anzahl der Iterationen nicht mehr verändert werden dürfen.

Nachfolgende Abbildung zeigt eine mögliche Konsolidierung der Choreographie bei mehrfach ausgeführten Aufrufaktivitäten mittels parallelem forEach. Hierzu wird der EH-Scope mit in die Schleife kopiert und somit mehrfach ausgeführt.

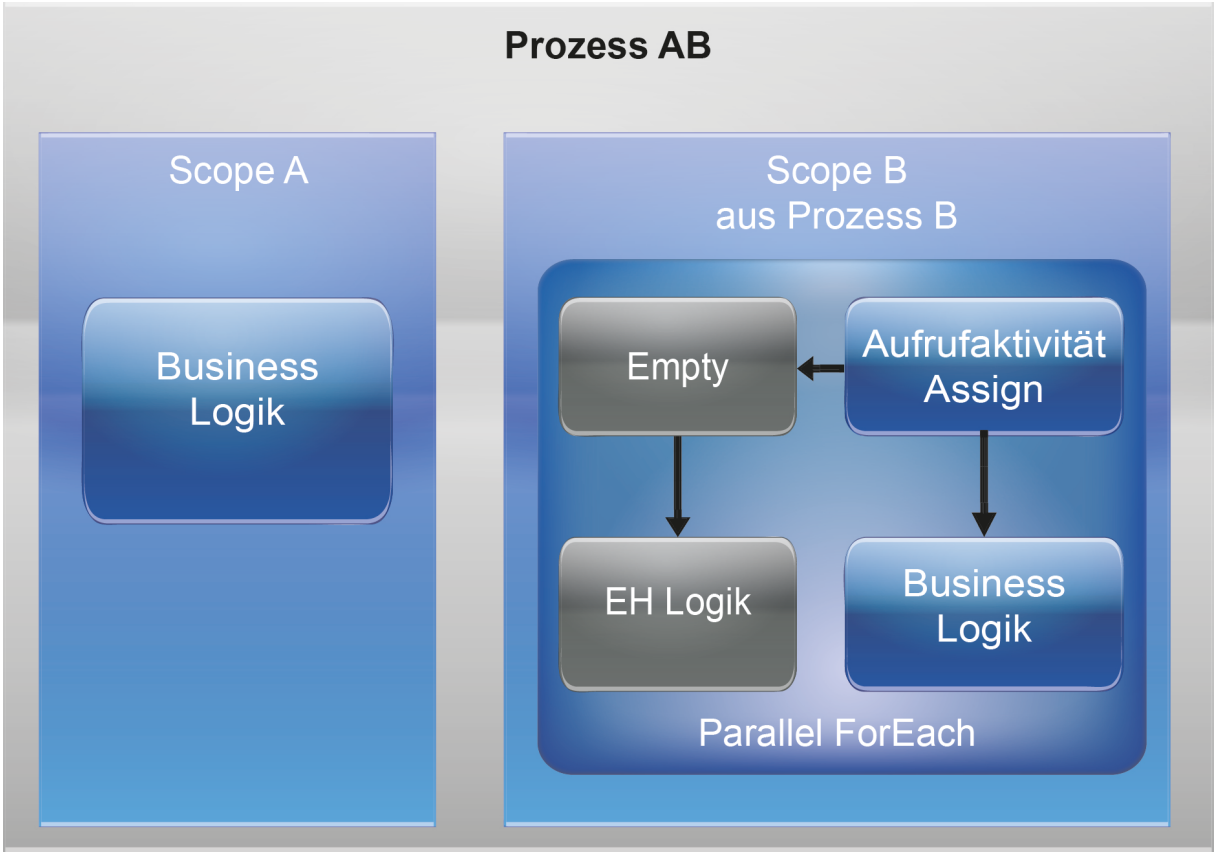


**Abbildung 8 - Aufrufbeispiel EH mit zwei Prozessen konsolidiert - MultiInstance**

### 3.1.1.5 Parallele Pfade

Durch Invoke-Aktivitäten, die auf parallelen Pfaden ausgeführt werden, können ebenfalls mehrere Messages an den EH gesendet und somit mehrere Instanzen des EH erzeugt werden. Die naheliegende Lösung, von beiden Assign-Aktivitäten aus dem konsolidierten Prozess jeweils einen Link zur EH-Logik zu ziehen, wird allerdings nicht funktionieren. Der Grund hierfür ist die Link-Semantik von BPEL: Um sogenannte Race-Conditions zu vermeiden, wird eine Aktivität auf die ein Link zeigt erst ausgeführt, wenn alle Links evaluiert werden können. Wie in Abschnitt 2.1.11 erläutert, besitzt ein Link einen von drei möglichen Zuständen: *wahr*, *falsch* und *undefiniert*. Solange die Assign-Aktivität noch nicht ausgeführt wurde, ist der Zustand des Links also undefiniert. Dies führt hier dazu, dass die in den Scope kopierte EH-Logik nur ausgeführt wird, wenn beide Assign-Aktivitäten abgearbeitet wurden – und dann auch nur genau ein Mal. Daher muss die EH-Logik für jede Assign-Aktivität auf dem Pfad in den Scope kopiert und mit dem Assign verlinkt werden.

Somit zeichnet sich hier bei multiplen Instanzen bereits eine Einschränkung der emulierten EH ab, was das Timingverhalten betrifft (vgl. hierzu Abschnitt 3.4) [WKL14].

### 3.1.2 Mehrfacher Aufruf des EH durch Schleifen

EH dürfen grundsätzlich mehrfach aufgerufen werden. Für jeden Aufruf wird eine eigene Instanz des EH erzeugt. Es gibt allerdings mehrere Möglichkeiten den EH mehrfach aufzurufen. Im Nachfolgenden Fall wird von einem Prozess aus einem wiederholbaren Konstrukt (Schleife) eine Message pro Iteration an den EH versendet.

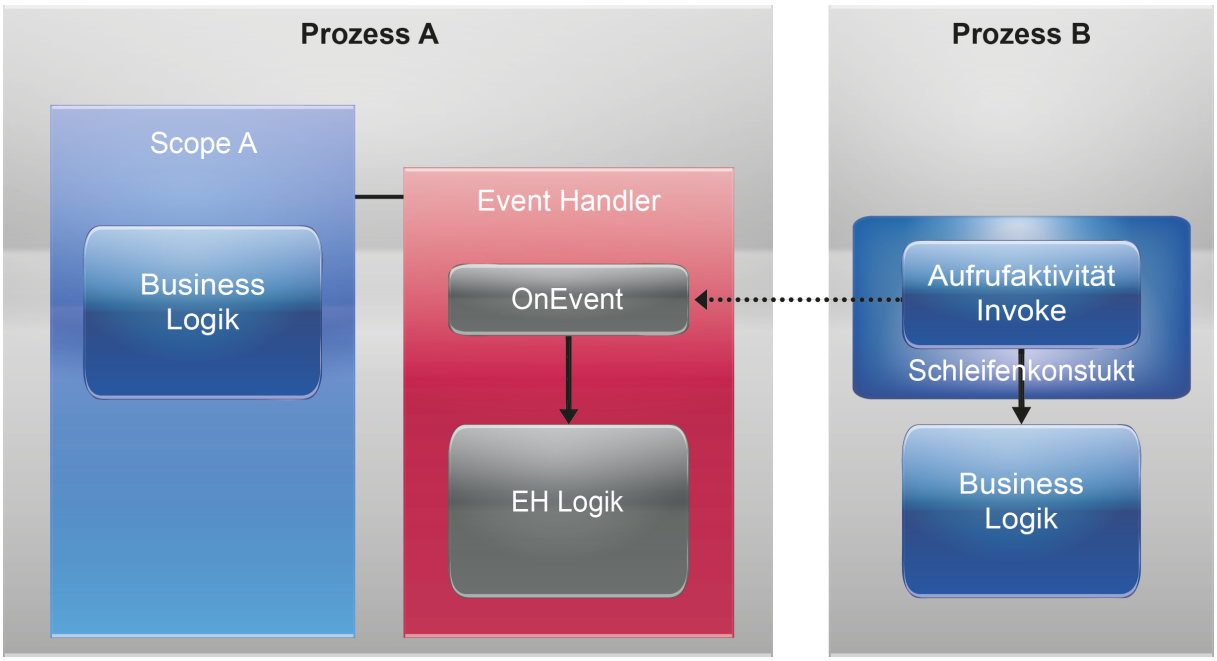


**Abbildung 9 - Mehrfacher Aufruf des EH durch Schleifen**

Wie bereits in Abschnitt 3.1.1.4 erläutert, kommt es auf die Art des Schleifenkonstrukts, welches die Aufrufaktivität umgibt, an um zu bestimmen wie multiple Instanzen des EH aufgerufen werden sollen.

Da die detaillierte Betrachtung von Multiplen Instanzen innerhalb zu konsolidierender Choreographien den Umfang dieser Arbeit sprengen würde, wird in den nachfolgenden Szenarien nicht weiter darauf eingegangen. Ebenfalls wird, bis auf den einfachen Fall aus Abschnitt 3.1.1.4, indem zur Laufzeit die Anzahl der Iterationen fest steht und der EH-Scope nicht mit anderen Prozessen interagiert (vgl. Abbildung 8), auf eine Implementierung der Fälle für Multiple Instanzen verzichtet.

### 3.1.3 Mehrfacher Aufruf des EH durch Prozesse

Eine andere Art den EH mehrfach aufzurufen ist der einfache Aufruf von Prozessen. In diesem Fall rufen mehrere Prozesse jeweils dasselbe OnEvent innerhalb des EH auf und erzeugen somit jeweils eine Instanz des EH.

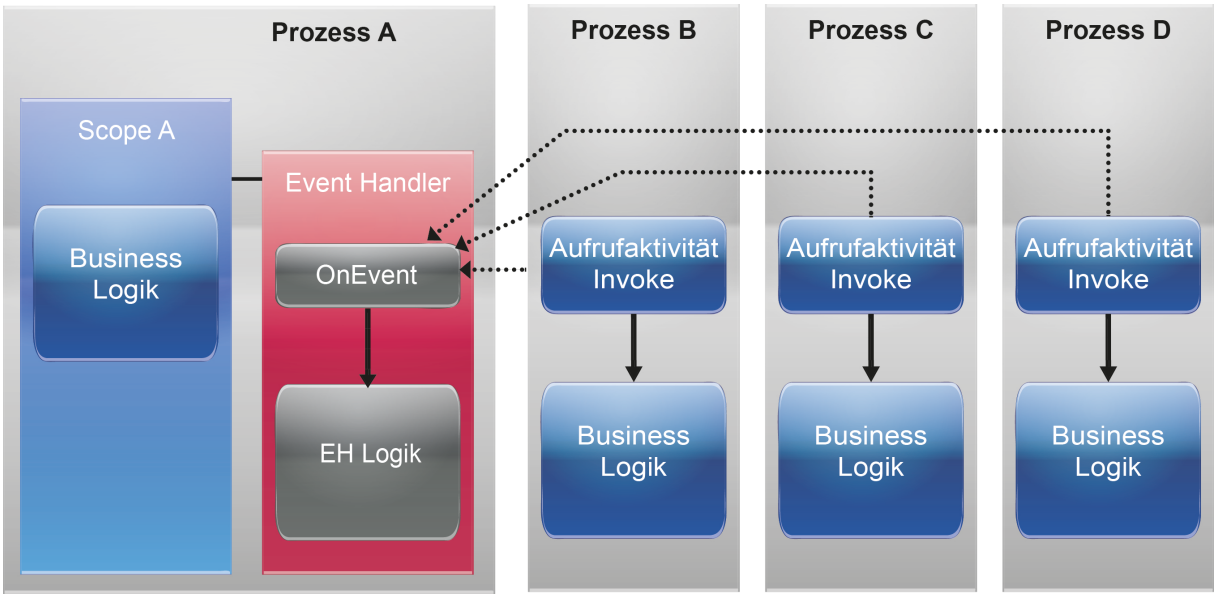


**Abbildung 10 - Mehrfacher Aufruf des EH durch Prozesse**

Damit der Aufruf auch im konsolidierten Prozess klappt, wird der Scope mit der beinhaltenden EH-Logik für jeden Prozess B, C und D jeweils in den entsprechenden Scope nach der Konsolidierung kopiert und steht dadurch jedem zur Verfügung. Dabei werden die Vorgehensweisen aus Abschnitt 3.1.1 angewendet.

### 3.1.4 Mehrfacher Aufruf des EH durch multiple OnEvents

Auch hier sind, wie in Abschnitt 3.1.3, mehrere Prozesse am Aufruf beteiligt.

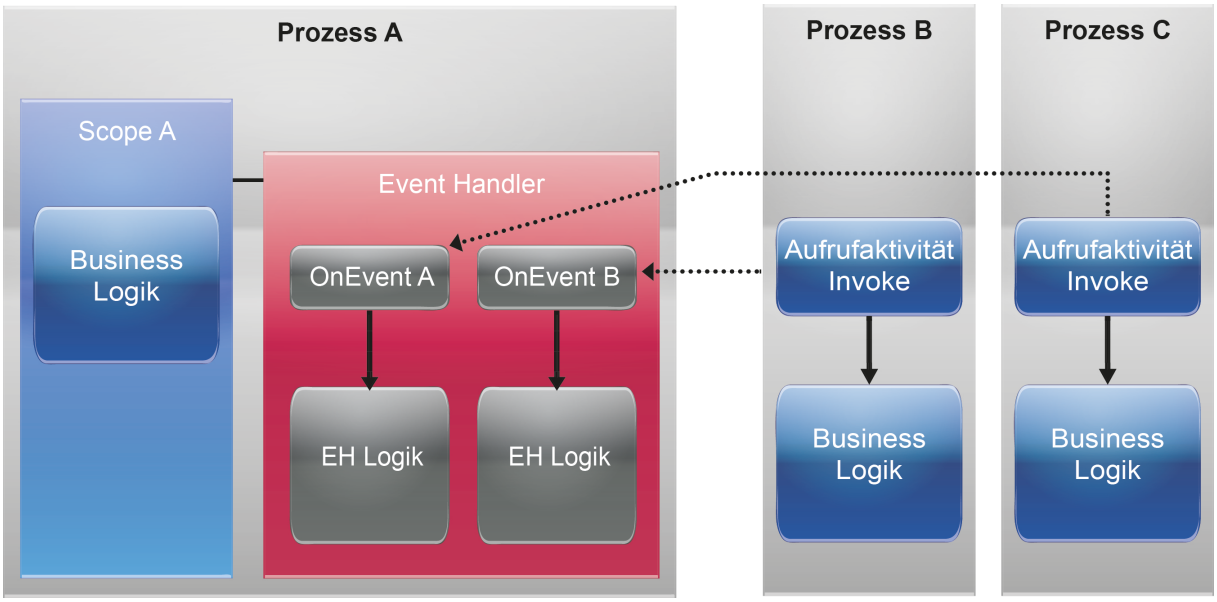


**Abbildung 11 - Mehrfacher Aufruf des EH durch multiple OnEvents**

Im Gegensatz zum erläuterten Fall aus 3.1.3 rufen die beteiligten Prozesse nicht alle dasselbe OnEvent innerhalb des EH auf. Daher muss bei der Konsolidierung darauf geachtet werden, dass der entsprechende Scope aus OnEvent A bzw. OnEvent B zum Prozess B bzw. C, wie in Abschnitt 3.1.1 erläutert, kopiert wird.

## 3.2 Interaktionsszenarien mit EH und Prozessen

### 3.2.1 Aufruf des EH durch externen Partner

Prozess C interagiert mit dem EH aus Prozess A, allerdings wird Prozess C nicht innerhalb der Choreographie gemeinsam mit den Prozessen A und B konsolidiert. Damit stellt Prozess C einen externen Partner dar.

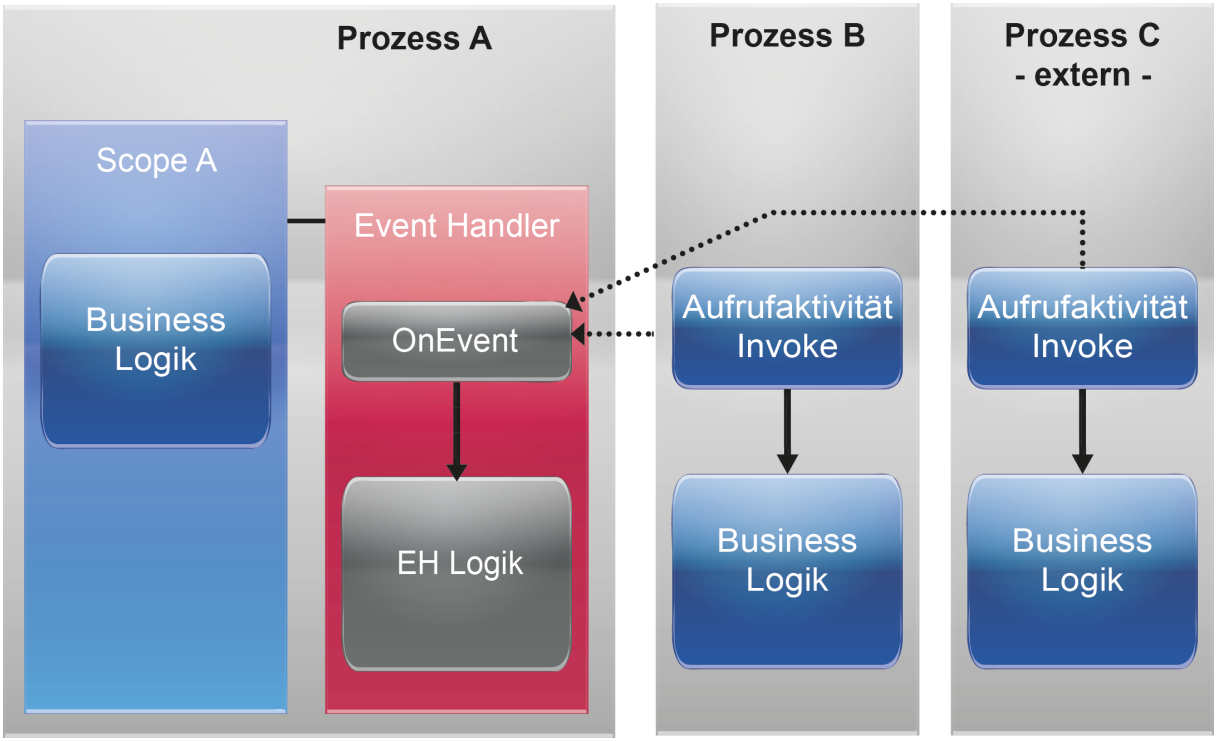


**Abbildung 12 - Aufruf des EH durch externen Prozess**

Damit der EH aus Prozess A auch im konsolidierten Prozess für externe Partner weiterhin aufrufbar bleibt, wird dieser an den konsolidierten Scope aus Prozess A wieder angehängt. Die Lebensdauer dieses EH bleibt wie beim Original erhalten. Für Prozess B wird, wie in Abschnitt 3.1.1. der Scope als Child des OnEvents kopiert.

Damit sind die Verhaltensweisen des EH für den Fall, dass der EH nicht mit den konsolidierten Prozessen innerhalb der Choreographie kommuniziert, mit den der originalen Choreographie identisch geblieben. Im folgenden Abschnitt 3.2.2 wird dieser Fall betrachtet.

### 3.2.2 Aktivierter EH versendet Message und aktiviert weiteren EH

Dieses Szenario lässt sich in zwei Unterfälle aufteilen, wobei der Fall mit externen Partnern einen Sonderfall darstellt.

Im ersten Fall versendet ein EH, der durch einen konsolidierten Prozess innerhalb der Choreographie instanziiert wurde, eine Message an einen EH eines ebenfalls konsolidierten Prozesses und erstellt dadurch eine neue Instanz des EH. Da der emulierte EH des ersten Prozesses ein Scope ist, wurde durch die Konsolidierung die Invoke-Aktivität in ein Assign umgewandelt und mit einem ausgehenden Kontrollfluss-Link versehen. Da hierbei nur Scopes beteiligt sind tritt keine Grenzüberschreitung der Links auf. Der Fall ist vereinfacht gesehen einem Merge von zwei interagierenden Prozessen gleichzusetzen und daher trivial.

Betrachten wir aber nun die Beteiligung eines externen Partners. In diesem Fall sind zwei Prozesse mit EH beteiligt, wobei ein bereits aktiver EH den EH des anderen Prozesses aktiviert. Dabei sollen sowohl Prozess A und auch B konsolidiert werden und der EH aus Prozess A soll auch durch externe Partner aufrufbar sein. Siehe Abbildung 12.

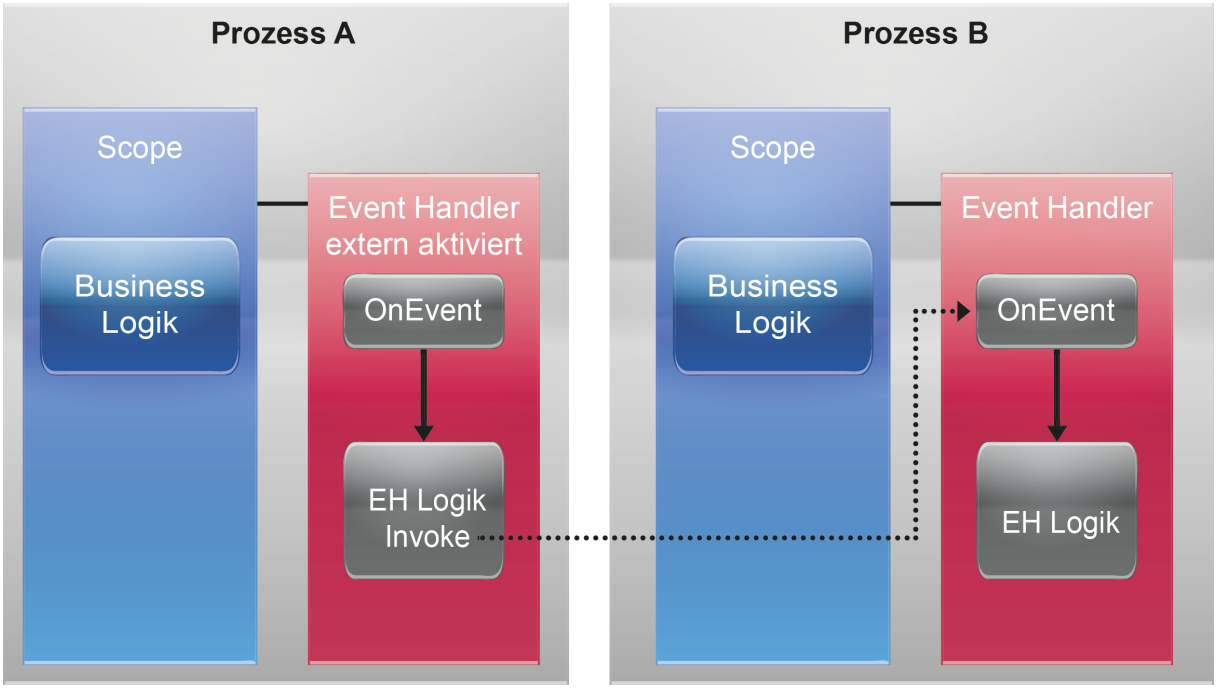


**Abbildung 13 - Extern instanziierter EH instanziiert weiteren EH**

Hier erkennen wir eine Einschränkung bei der Konsolidierung. Der EH aus Prozess A kann nicht wie in den vorangegangenen Abschnitten gezeigt durch kopieren in den konsolidierten Process-Scope emuliert werden, da dieser sonst nicht vom externen Partner aufgerufen werden kann. Ebenfalls kann der EH nicht am zu konsolidierenden Prozess-Scope A angeheftet bleiben, da sonst Grenzüberschreitungen der Kontrollfluss-Links entstehen, die nicht aufgelöst werden können. Dadurch ist dieses Interaktionsmuster zwischen extern aktiviertem EH und innerhalb der Choreographie zu konsolidierenden Prozessen mit den bisherigen Untersuchungen nicht zu Konsolidieren. Es bedarf weiterer Betrachtung dieses Falles um eine mögliche Lösung für die Konsolidierung zu finden.

Bei vollständiger Betrachtung stellt man fest, dass der EH aus Prozess A auch durch ein On-Alarm aktiviert werden kann. Vorgegriffen lässt sich sagen, dass es hierzu eine Lösung gibt und dieses Interaktionsmuster konsolidiert werden kann. In Abschnitt 3.3 wird detailliert darauf eingegangen.

### 3.2.3 Extern aktivierter EH und Messages

Dieser Fall besitzt die gleiche Einschränkung wie das Szenario aus Abschnitt 3.2.2. Der durch einen externen Partner instanziierte EH kommuniziert synchron mit den zu konsolidierenden Prozessen C und D. Hierbei treten bei der Konsolidierung die gleichen Grenzüberschreitungen der Kontrollfluss-Links auf, wie in Abschnitt 3.2.2 gezeigt.

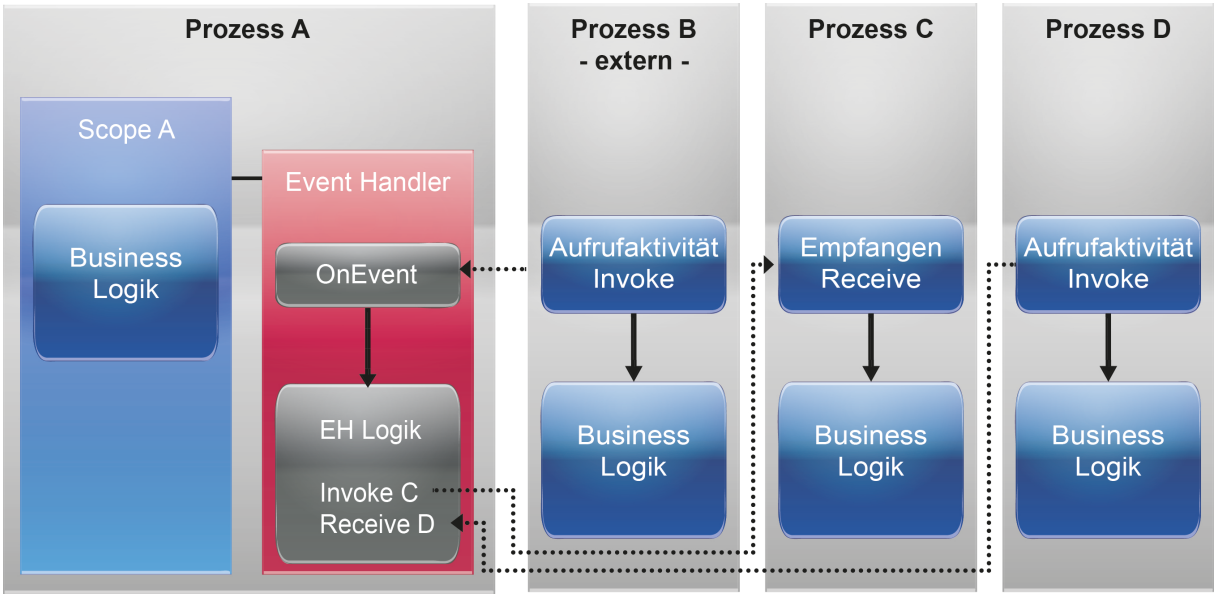


**Abbildung 14 - Synchrone Kommunikation durch extern instanziierten EH**

Diese Grenzüberschreitungen können nicht aufgelöst werden. Dabei ist es unerheblich, ob der EH synchron oder asynchron mit den Prozessen C und D kommuniziert. Somit können Choreographien, bei denen der am Scope A angeheftete EH für die externe Instanziierung bestehen bleibt nicht Konsolidiert werden. Diese Einschränkung führt dazu, dass die Merge-Operation nur für Closed-World-Betrachtungen durchgeführt werden kann (vgl. hierzu Abschnitt 3.4).

### 3.2.4 Intern aktivierter EH und synchrone Messages

In diesem Fall versendet der bereits aktivierte EH eine Message an einen Prozess und wartet auf eine Antwort.

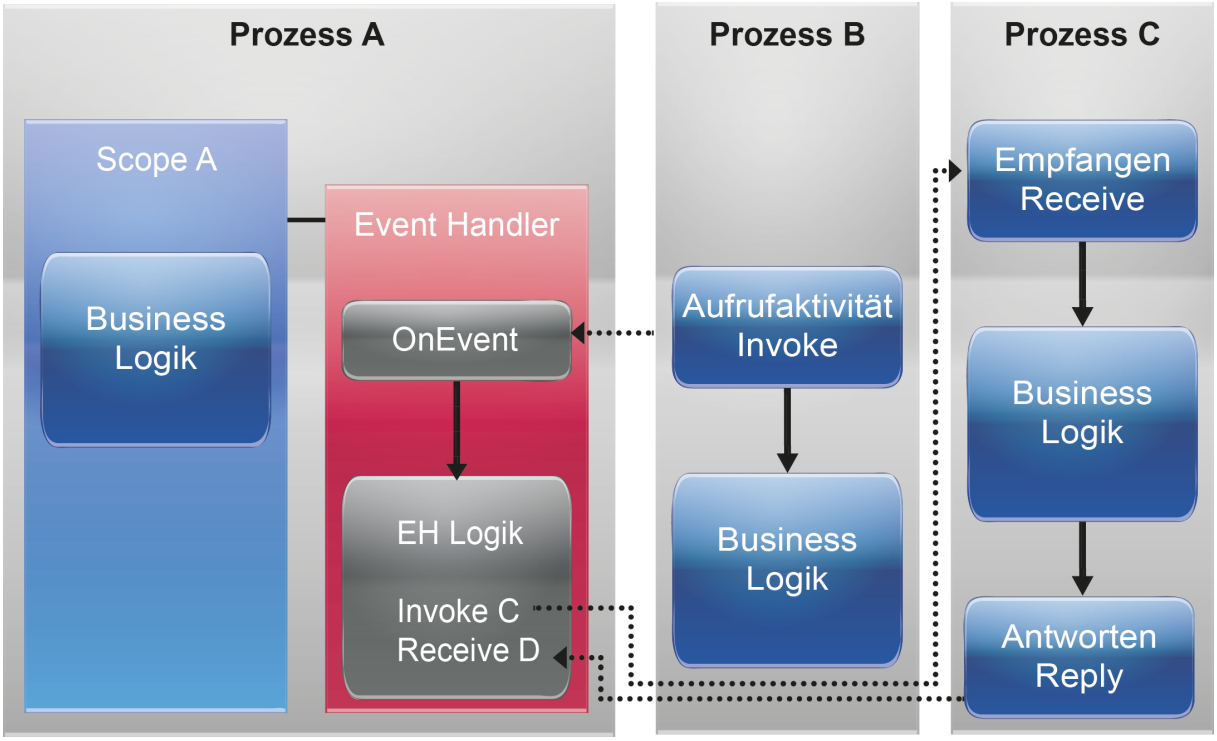


**Abbildung 15 - Synchrone Kommunikation mit EH**

Im Gegensatz zu vorherigem Abschnitt können die hier entstehenden Grenzüberschreitungen aufgelöst werden. Da sich die EH-Logik in einem Scope befindet, welcher von einem Flow und einem Parent-Scope umschlossen ist, können ein- und ausgehende Kontrollfluss-Links zu den konsolidierten Prozess-Scopes B und C zeigen. Hier greift ebenfalls die bisherige Umwandlung von Kommunikationsaktivitäten in Synchronisationsaktivitäten (vgl. Abschnitt 2.2).

### 3.2.5 Intern aktivierter EH sendet asynchrone Messages

In diesem Fall versendet der durch Prozess B aktivierte EH Messages an die innerhalb der Choreographie zu konsolidierenden Prozesse C und D. Nachfolgende Abbildung stellt das Interaktionsszenario vor der Konsolidierung dar.

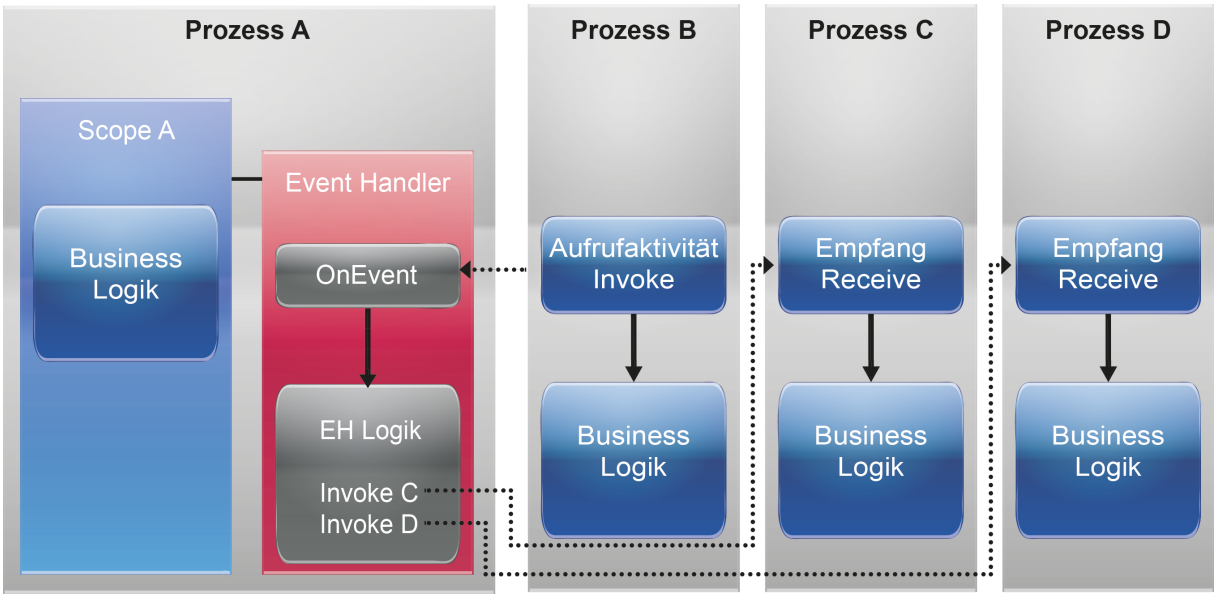


**Abbildung 16 - Versenden asynchroner Messages durch EH innerhalb Choreographie**

Hierbei ist erkennbar, dass im konsolidierten Prozess die send-and-forget-Aktivitäten innerhalb der Logik des EH mittels Assign emuliert werden können. Es entstehen auch keine Grenzüberschreitungen von Kontrollfluss-Links, da diese alle zwischen Scopes bestehen und der EH-Scope in den Scope von Prozess B kopiert wurde.

### 3.2.6 Intern aktivierter EH empfängt asynchrone Messages

Entsprechend umgekehrt zur Richtung des Szenarios aus Abschnitt 3.2.5 empfängt der aktivierte EH Messages von Prozessen innerhalb der Choreographie.

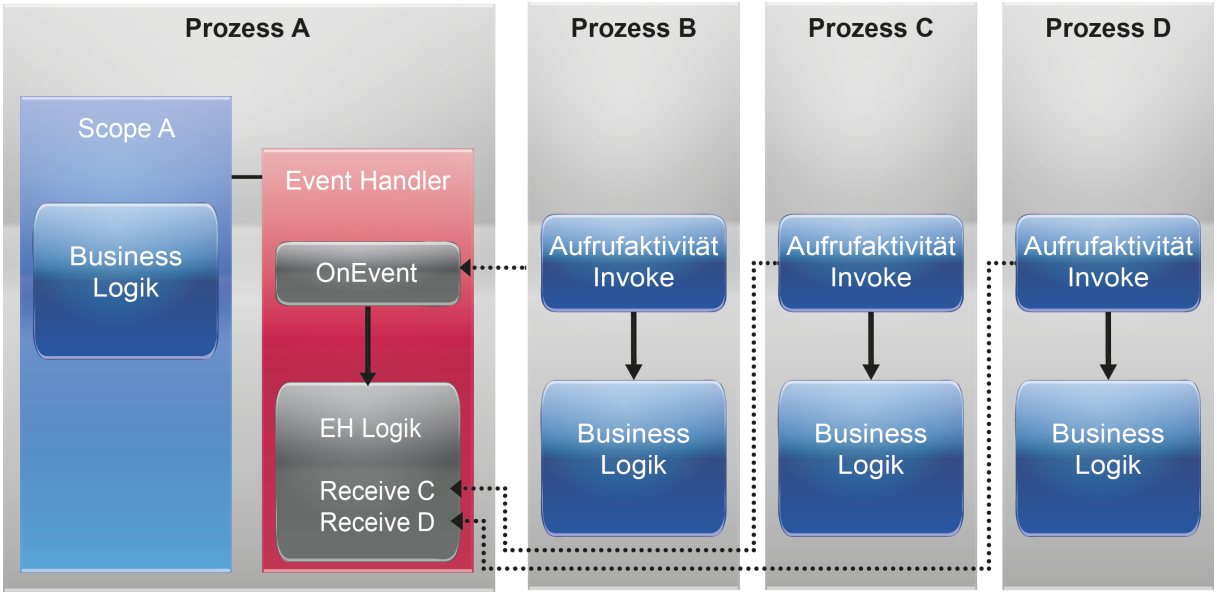


**Abbildung 17 - Empfangen asynchroner Messages durch EH innerhalb Choreographie**

Analog zum Fall aus Abschnitt 3.2.5 können auch hier die Kommunikationsaktivitäten durch Synchronisationsaktivitäten ersetzt werden.

## 3.3 Interaktionsszenarien mittels OnAlarm aktivierter EH

Neben dem senden einer Message an das OnEvent eines EH, kann der EH auch über ein OnAlarm instanziiert werden. Dabei wird das OnAlarm-Event nach Ablauf eines Timers oder zu einem über XPATH festgelegtem Zeitpunkt aktiviert und erzeugt eine neue Instanz des EH.

Sofern der EH-Scope innerhalb des OnAlarm nicht mit anderen Prozessen aus der Choreographie interagiert, muss dieser nicht umgebaut werden und der EH kann dem zugehörigen Scope angefügt werden.

Sollte der über OnAlarm instanziierte EH mit den Prozessen aus der Choreography synchron oder asynchron kommunizieren (vgl. Abbildung 18), muss der EH emuliert werden.

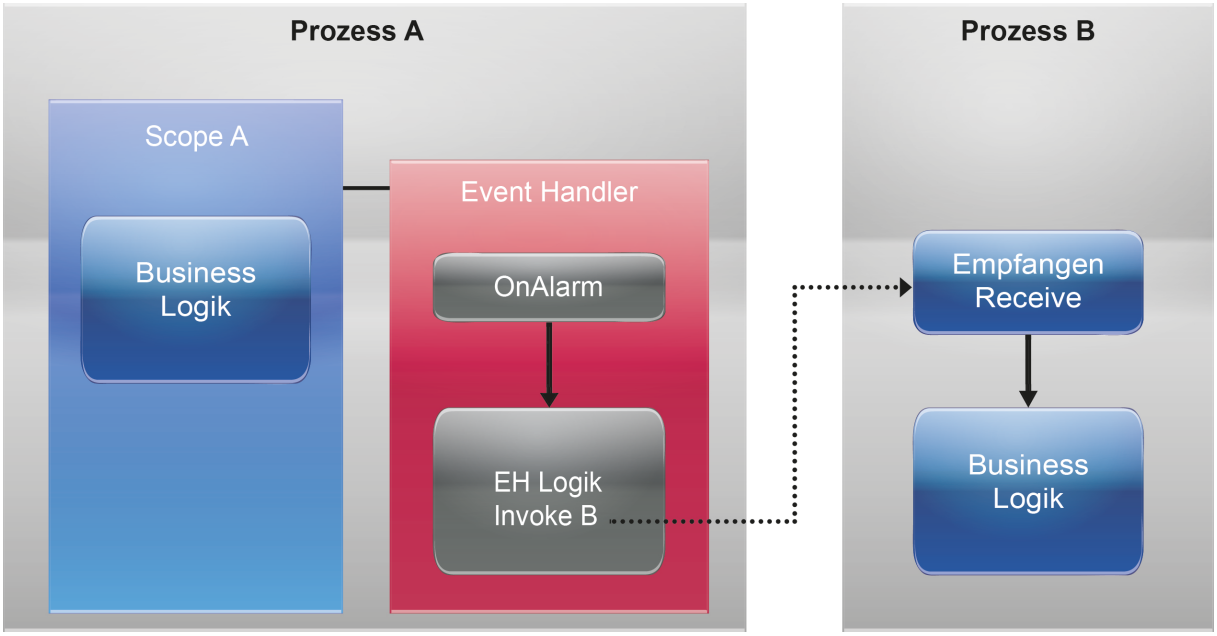


**Abbildung 18 - Interaktion des durch OnAlarm aktivierten EH**

Analog zu Abschnitt 3.1.1 besteht die Lösung darin, den Child-Scope des OnAlarms in den zu konsolidierenden Prozess zu kopieren und mit der bereits beschriebenen Funktionalität auszustatten. Der hierbei wesentliche Unterschied zu Abschnitt 3.1.1 ist jedoch, dass das OnAlarm durch ein Wait ersetzt wird. Die Wait-Aktivität erhält dabei den identischen XPATH-Ausdruck aus dem OnAlarm.

Ebenfalls gilt hier die Einschränkung des Timingverhaltens aus Abschnitt 3.1.1, da der EH-Scope nicht zwingend gleichzeitig zum Logik-Scope des konsolidierten Prozesses gestartet wird. Dies könnte für Timer, die erst ab der Aktivierung des Scopes laufen relevant sein.

Sollten innerhalb des Prozesses A multiple OnAlarm existieren, so muss jeder EH-Scope in das den konsolidierten Scope des Prozesses A kopiert werden.

Damit schließlich die Grenzüberschreitung der Kontrollfluss-Links des EH im OnAlarm-Scope behoben wird, ist es notwendig aus dem an Prozess A angehefteten EH alle betreffenden OnAlarm zu entfernen.

Abbildung 19 zeigt den erläuterten Fall nach erfolgreicher Konsolidierung. Zur einfacheren Darstellung wurde auf den ursprünglich an Prozess A angehefteten EH verzichtet.

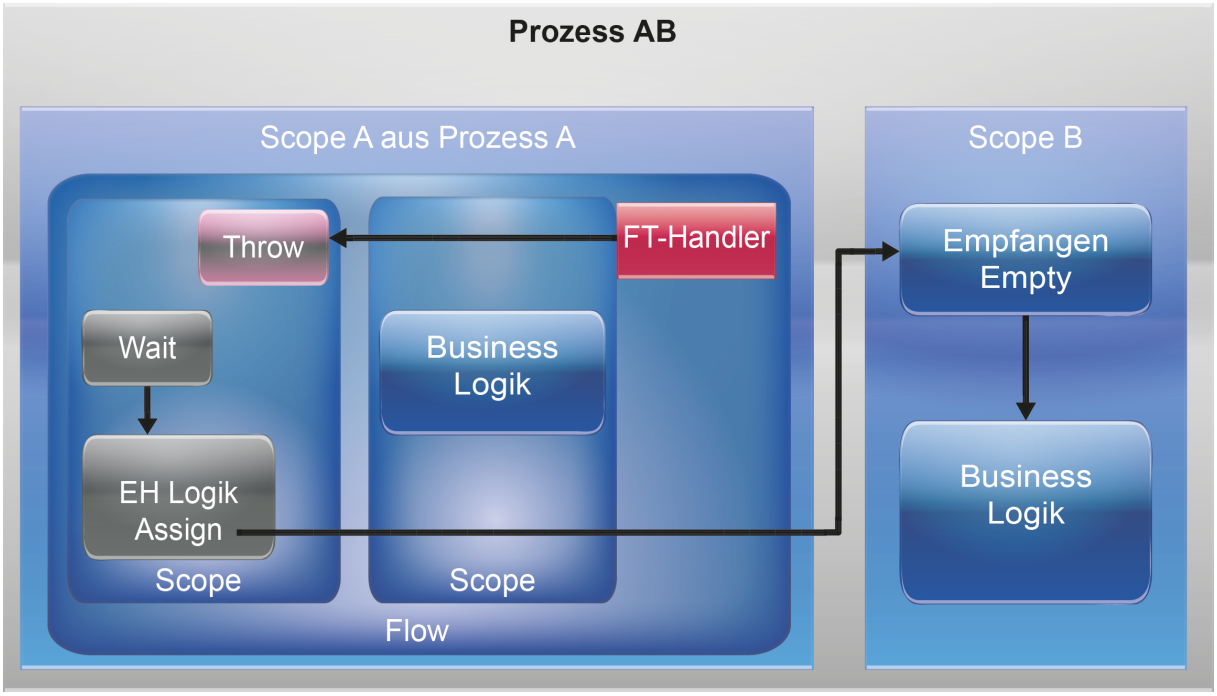


**Abbildung 19 - OnEvent aktivierter EH konsolidiert**

## 3.4 Eigenschaften, Einschränkungen und Unterschiede

Im vorangegangenen Kapitel wurde gezeigt, dass es grundsätzlich möglich ist das Verhalten eines EH zu Emulieren, sodass miteinander interagierende Prozesse konsolidiert werden können, sofern man die Aktivierung des EH durch externe Partner ausschließt und dadurch eine in sich geschlossene Betrachtung der Choreographie zu Grunde legt. Dieses Kapitel fasst die Eigenschaften, bestehende Einschränkungen und Unterschiede bezüglich des EH und dessen emulierter Version zusammen.

### 3.4.1 Nachbilden der Eigenschaften des EH

Die grundlegenden Eigenschaften eines EH lauten wie folgt:

- Der einem Scope zugehörige EH wird aktiv, sobald dieser Scope Instanziiert wurde.
- Der EH wird durch OnMessage oder OnAlarm instanziiert.
- Der EH kann über mehrfach Instanziiert werden. So wird beispielsweise bei jedem Erhalt einer Message eine neue Instanz erzeugt
- Der EH ist nur solange aktiv, bis der zugehörige Scope, an dem der EH angeheftet ist, beendet wird.
- Ein instanziierter EH darf alle Aktivitäten innerhalb des EH-Scopes ausführen, auch wenn der zugehörige Scope schon nicht mehr aktiv ist.

Diese Eigenschaften müssen folglich auch vom emulierten EH erfüllt sein um diesen korrekt Nachzubilden. Wie bereits in vorangegangenen den Fallbeispielen gezeigt, können die Aufruf- und Lebensdauer-Eigenschaften des EH mit kleinen Abstrichen, was das Timingverhalten betrifft, auch vom emulierten EH erfüllt werden. Einschränkungen gibt es bezüglich der Eigenschaften des EH bei der multiplen Instanziierung des EH, was in weiteren, noch folgenden, Arbeiten zu diesem Thema untersucht werden müsste.

### 3.4.2 Einschränkungen des emulierten EH

Eine mehrfach aufgetretene Einschränkung ist die bezüglich des Timingverhaltens. Hierbei ist das Verhalten stark von der benutzten Workflow-Engine abhängig. Die Einschränkungen bezüglich des Timings können gleich in mehreren Bereichen auftreten. Nachfolgend werden diese aufgezählt und die möglichen Auswirkungen des veränderten Timings näher erläutert.

**Timingverhalten bei Scopes:**

Der EH wird genau dann aktiv, wenn der zugehörige Scope, an dem der EH angeheftet ist, aktiv wird. Der emulierte EH wird erst dann aktiv, wenn das Flow-Konstrukt, in das der EH mit seinem Scope eingebettet ist, aktiv wird. Einen Parent-Scope um das Flow-Konstrukt anzulegen ist zwar zur Emulation sinnvoll um die Isolation und Abgrenzung zwischen den einzelnen Prozessen herzustellen, aber eben nicht zu 100-Prozent identisch. Mit dem Aktivieren des Parent-Scopes wird der EH nicht sofort aktiv, sondern erst wenn die Child-Scopes durch den Flow aktiv werden. Dadurch kann beispielsweise das Szenario auftreten, dass der mit dem emulierten EH interagierende Scope vor dem Scope, an dem der EH ursprünglich angeheftet

gewesen ist, aktiv wird und damit der EH vor seinem Ursprungs-Scope aktiv ist. Dies dürfte laut der Spezifikation so nicht vorkommen. Dadurch besteht die Möglichkeit, dass der EH zeitversetzt zum Scope mit der Ausführungslogik (In den Beispielen als Scope A bezeichnet) aktiv ist und somit eine eventuell zu früh eintreffende Message eines externen Partners den EH nicht instanziiert. Die BPEL Spezifikation macht hier keine Aussage darüber, was mit der Message passiert, wenn der Empfänger noch nicht gestartet ist. Es ist jedoch vorstellbar, dass die Message innerhalb der Workflow-Engine zwischengespeichert wird, bis der Empfänger verfügbar ist.

**Timingverhalten bei Schleifenkonstrukten:**

Hier kann es durch Serialisierung statt parallelem Ausführen der multiplen EH-Instanzen zu einer verlängerten Ausführungsdauer kommen. Zwar bleibt die Ausführungszeit ingesamt gleich, jedoch wird die Dauer durch die fehlende Parallelisierung der Instanzen verlängert (vgl. Abschnitt 3.1.1.4).

**Timingverhalten bei parallelen Pfaden:**

Ähnlich wie bei Schleifenkonstrukten kann durch das mehrfache Einfügen der EH-Logik auf den parallelen Pfaden die Ausführungsdauer verlängert werden. Grund in diesem Fall ist die nicht zwingend gegebene Parallelität.

**Timingverhalten bei emuliertem OnAlarm:**

Wie in Abschnitt 3.3 erläutert kann die Dauer einer Wait-Aktivität des emulierten OnAlarms aufgrund der verspäteten Aktivierung des Scopes später beginnen als beim Originalkonstrukt, was zu unerwünschtem Verhalten bei zu spätem Auslösen des OnAlarms führen könnte.

**Interaktion des extern aktivierten EH mit zu konsolidierenden Prozessen:**

Eine weitere, wesentliche Einschränkung ist die nicht mögliche Konsolidierung von Choreographien, bei denen ein extern instanziierter EH mit den intern konsolidierten Prozessen interagiert (vgl. Abschnitte 3.2.2 und 3.2.3). Hierbei treten Grenzüberschreitungen von Kontrollfluss-Links auf, wenn aus dem EH Messages versendet oder solche empfangen werden. Da die Message-Links während der Merge-Operation in Kontrollfluss-Links und Synchronisations-, beziehungsweise Aufrufaktivitäten umgewandelt werden, zeigen Kontrollfluss-Links in und aus dem EH. Dies ist laut Spezifikation nicht zulässig.

Ein erster Ansatz um dieses Problem zu lösen könnte die Emulation des EH-Verhaltens durch eine Pick-Aktivität sein. Hierbei muss jedoch geprüft werden, ob das Modell aus Abbildung 20 auch alle Eigenschaften eines EH erfüllen kann.

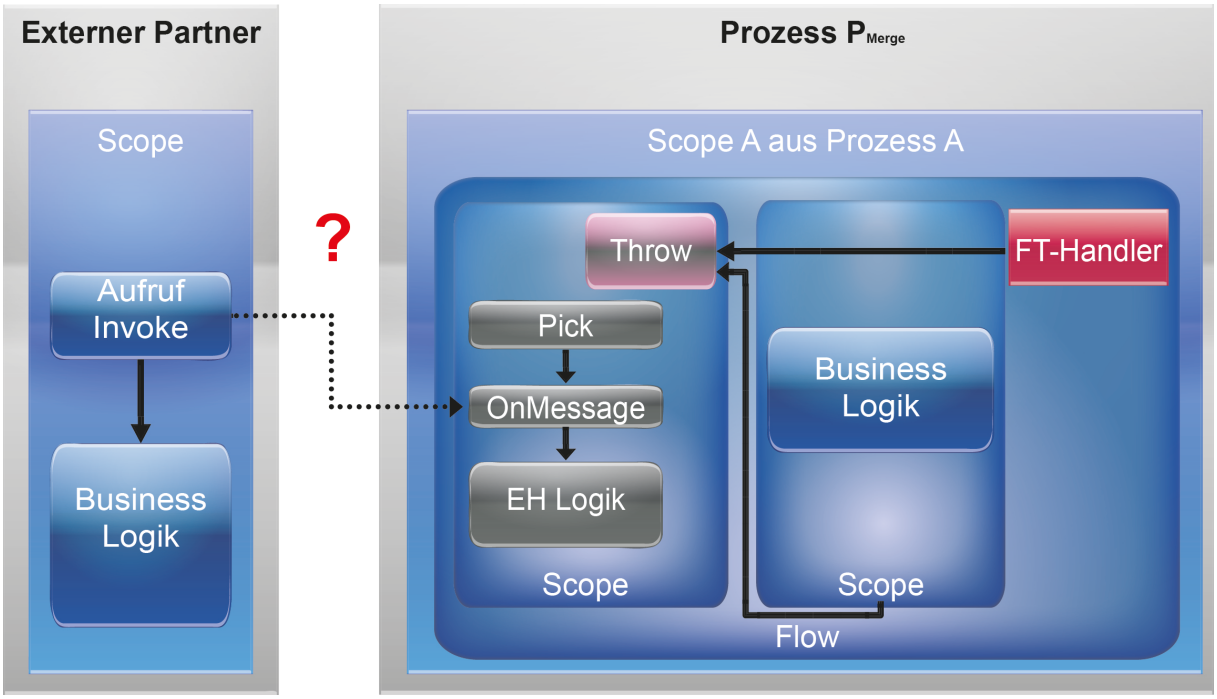


**Abbildung 20 - Zu untersuchende Emulation des EH für externe Partner**

### 3.4.3 Unterschiede des emulierten EH zum EH

Neben den als Einschränkung aufgeführten Unterschieden bezüglich des Timingverhaltens gibt es noch zwei weitere Unterschiede zwischen dem emulierten EH und dem Originalkonstrukt.

**Zustände des EH-Scopes:**

Um die Lebensdauer des EH-Scopes zu emulieren, muss in den Zuständen *failed* und *terminated* des Scopes mit der Ausführungslogik ein Fehler geworfen werden um den EH-Scope zu beenden. Dafür werden am Scope mit der Ausführungslogik ein Fault- und ein Termination-Handler angebracht, die dann im jeweiligen Zustand ein Throw innerhalb des EH-Scopes aufrufen. Dabei wird ein benutzerdefinierter Fehler geworfen, der gesondert abgefangen werden muss, damit nicht der ganze Prozess beendet wird. Durch das werfen des Throw auch im Zustand *terminated*, wird zwar der EH-Scope beendet, befindet sich aber semantisch betrachtet im falschen Zustand *failed*.

**Datenfluss betrachten:**

Während beim Originalkonstrukt des EH dieser Zugriff auf die Variablen innerhalb des Scopes, an dem der EH angeheftet ist, besitzt ist beim emulierten EH ohne Anpassung keine Variable sichtbar. Der Grund hierfür ist, dass sich sowohl der Scope mit der Ausführungslogik als auch der EH-Scope innerhalb eines Flows befinden und damit isoliert sind. Die Lösung hierfür ist, wie in Abschnitt 2.2 erläutert, das Globalisieren der Variablen. Befinden diese sich im Parent-Scope des Flows so sind die Variablen für beide Scopes innerhalb des Flows Sichtbar [WKL14].

# 4 Implementierung

Das bereits vorliegende Choreography-Merge-Tool des Instituts für Architektur von Anwendungssystemen (IAAS), welches im Rahmen der Diplomarbeiten von Cui [CUI12], Berger [BER13] und Debicki [DEB13] entwickelt wurde, soll zur Behandlung von Event-Handlern erweitert werden. Das Tool nimmt eine ZIP-Datei entgegen, die eine Choreographie in BPEL4Chor und die zugehörigen WSDLs enthält. Siehe Abbildung 21.

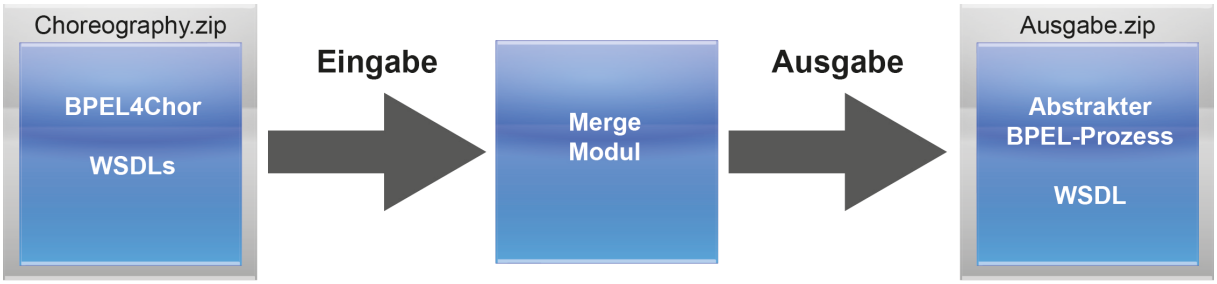


**Abbildung 21 - Funktionsweise Choreography Merge Tool, adaptiert aus [BER13]**

Nach dem Einlesen der Dateien aus der ZIP-Datei werden dann Schritte aus Abbildung 22 innerhalb des Merge-Moduls ausgeführt. Hierbei findet dann die eigentliche Konsolidierung der Choreographie statt. Die Kommunikationsaktivitäten, wie `<invoke>` und `<receive>`, werden durch Synchronisationsaktivitäten, wie `<assign>` und `<empty>`, ersetzt. Es wird ein neuer Prozess erstellt, der dann innerhalb eines `<flow>` die Scopes der zu konsolidierenden Prozesse enthält. Im vierten Schritt werden dann alle Message-Links auf bestimmte Konsolidierungsmuster untersucht und anhand dieser ebenfalls umgewandelt [BER13]. An dieser Stelle erweitert die Implementierung dieser Arbeit das vorhandene Choreography Merge Tool um die Erkennung von Konsolidierungsmustern mit Event-Handlern.

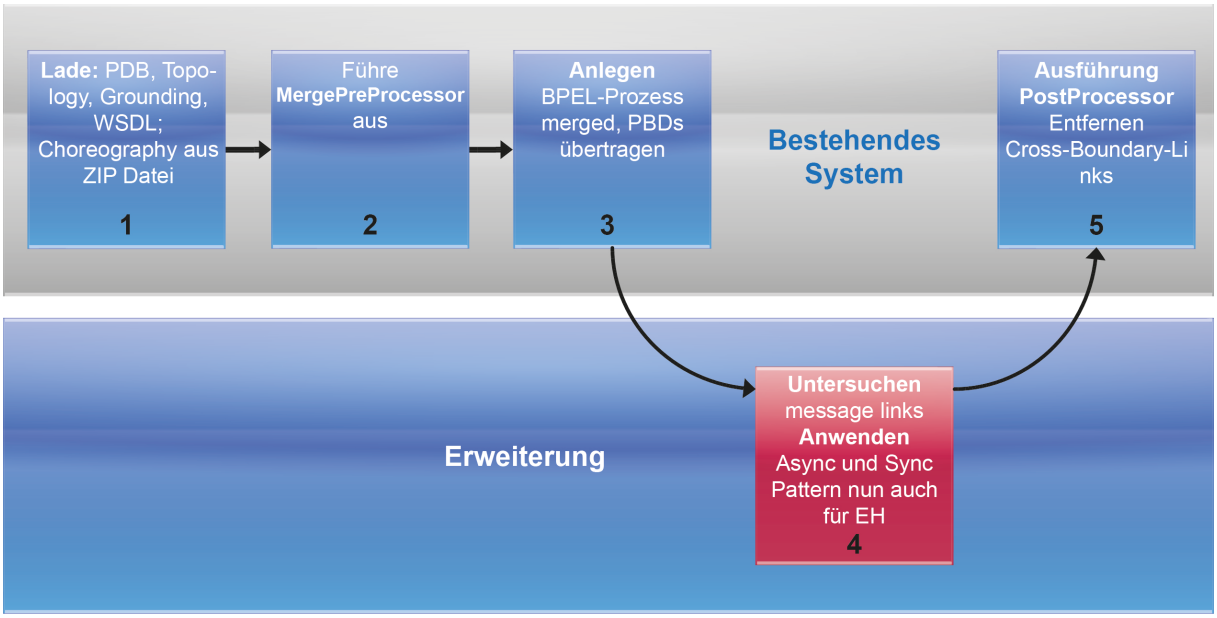


**Abbildung 22 - Erweiterung Choreography Merge Tool, adaptiert aus [BER13]**

Während der Anwendung der Konsolidierungsmuster für Event-Handler können zwischenzeitlich Grenzüberschreitungen bei Kontrollfluss-Links entstehen, da eventuell Fault- und Termination-Handler zu den Scopes hinzugefügt werden. Allerdings werden diese im letzten Schritt innerhalb des `MergePostProcessings` untersucht und entfernt. Als Ergebnis des Merge Moduls steht ein konsolidierter abstrakter Prozess mit den passenden WSDLs zur Verfügung [BER13].

Das `EventHandlerUtil` wird als weitere Komponente zum `FCTEUtil` hinzugefügt und realisiert so die Konsolidierung von Prozessen mit beteiligtem Event-Handler.

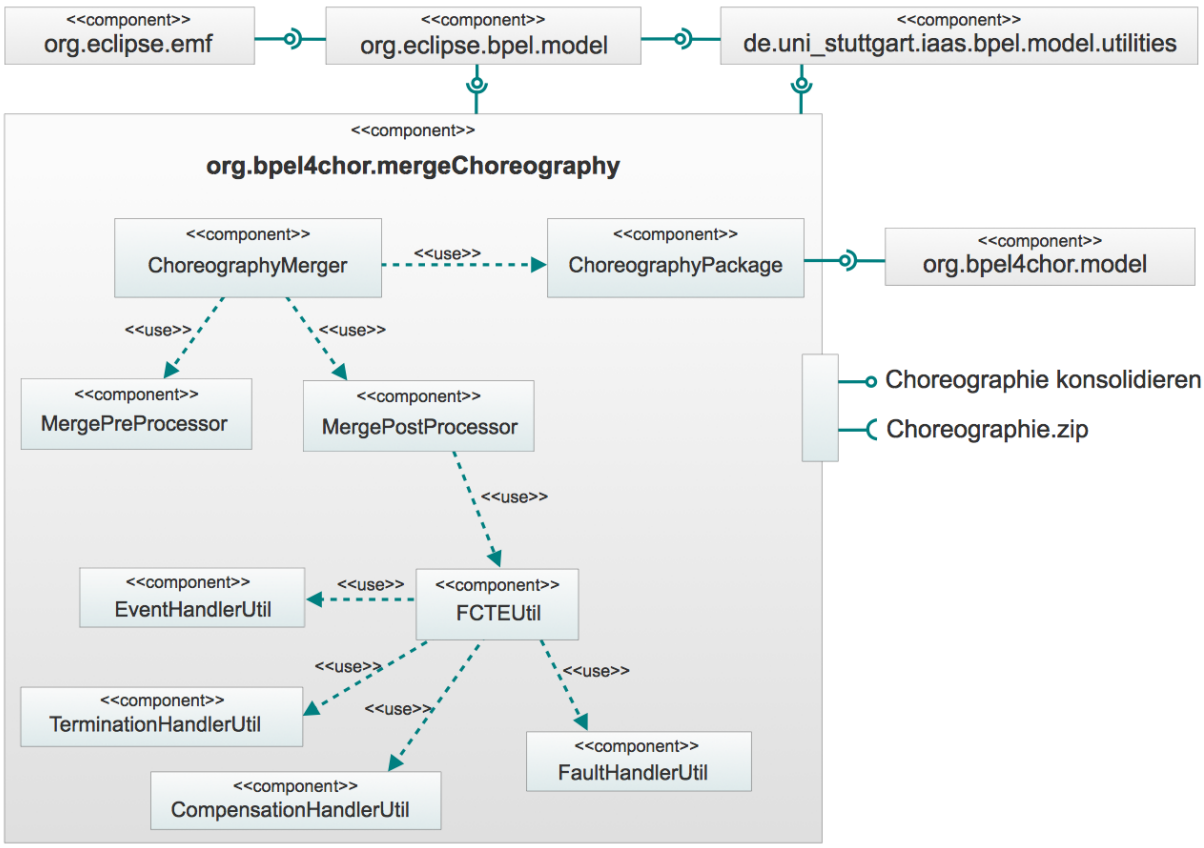


**Abbildung 23 - Komponentendiagramm für die Konsolidierung aus [BER13]**

Abbildung 23 zeigt das nun erweiterte Komponentendiagramm aus [BER13]. Die Komponente `EventHandlerUtil` wurde zum `FCTEUtil` hinzugefügt und stellt damit die Erweiterung in Schritt vier aus Abbildung 22 dar.

Das `EventHandlerUtil` untersucht ebenfalls wie die verwandten Handler-Utils ein- und ausgehende Links eines Scopes und erkennt damit eine Interaktion mit einem Event-Handler. Gefundene Scopes werden entsprechend markiert, dann mit den neu erstellten Konsolidierungsmustern (Patterns) verglichen und entsprechend über die Matcher umgewandelt. Je nachdem, ob der Event-Handler auch mehrfach aufgerufen werden kann, wird die aufrufende Aktivität in ein paralleles forEach gesetzt. Dabei wird die Variable für die Anzahl der Aufrufe entsprechend gesetzt. Die Event-Handler werden bei den Scopes jedoch beibehalten, damit diese auch mit eventuell nicht erkannten externen Partnern weiterhin kommunizieren können.

# 5 Zusammenfassung und Ausblick

Betrachtet man die zu konsolidierende Choreographie als solche, die nicht mit Partnern außerhalb der Choreographie interagiert, so können alle Szenarien, bis auf die mit multiplen Instanzen der Event-Handler, mit dem Choreography-Merger des IAAS konsolidiert werden. Bei multiplen Instanzen besteht die Einschränkung, dass die Eigenschaft der mehrfachen Instanziierung des Event-Handlers nicht erhalten werden kann.

Die Einschränkung bei multiplen Instanzen entsteht durch den Aufruf des Event-Handlers innerhalb von Schleifenkonstrukten. Da zur Design-Zeit nicht immer fest steht, wie viele Instanzen des Event-Handlers aufgerufen werden, kann das Verhalten von parallelen Instanzen des Event-Handlers nicht mittels eines parallelen forEach emuliert werden und bedarf daher weiterer Betrachtung. Zudem besteht beim Einsatz von Schleifenkonstrukten die Einschränkung der verlängerten Ausführungsdauer. Eine Optimierungsmöglichkeit zur Verbesserung der Ausführungsdauer bei multiplen Instanzen von Event-Handlern könnte das Ausrollen der Schleife bei einer geringen Anzahl von Iterationen sein. Hierbei werden die Schleifen entfernt und die entsprechenden Aktivitäten hintereinander ausgeführt. Voraussetzung dafür ist die Kenntnis der Anzahl der Iterationen zur Laufzeit [WKL14].

Abhängig von der Workflow-Engine können auch Einschränkungen im Zusammenhang mit dem Timingverhalten von verwendeten Flows zur Emulation der Event-Handler auftreten. Hierbei könnte der Aktivierungszeitpunkt der einzelnen, innerhalb des Flows parallel modellierten, Scopes untereinander differieren, sodass der Scope mit der Ausführungslogik erst nach dem Event-Handler aktiv ist. Zudem kann es bei der Emulation von OnAlarm-getriggerten Event-Handlern zu unterschiedlichen Wartezeiten bei einer Wait-Aktivität kommen, sofern der EH-Scope nicht gleichzeitig mit dem Scope der Ausführungslogik aktiviert wird.

Da aber nicht alle Choreographien als in sich geschlossen betrachtet werden können, ist es sehr wünschenswert die Interaktion von externen Partnern mit den zu konsolidierten Prozessen ebenfalls abbilden zu können. Wie in Abschnitt 3.4 vorgeschlagen, könnte die Verwendung eines Modells mit einer Pick-Aktivität ein möglicher Weg sein, sofern man eine Lösung für die mehrfache Instanziierung dieses emulierten Event-Handler-Modells findet. Die bei der Konsolidierung auftretenden Grenzüberschreitungen der Kontrollfluss-Links des, durch den externen Partner instanziierten, Event-Handlers mit den konsolidierten interagierenden Prozessen könnten so aufgehoben werden und daher die Konsolidierung auch für diesen Fall ermöglichen.

Das Choreography-Merger-Tool am IAAS wurde um die Funktionalität des Konsolidierens von mittels Event-Handler kommunizierender Choreographien erweitert. Wie bereits erwähnt können nicht alle Interaktionsszenarien mit Event-Handlern vom Tool konsolidiert werden. Durch den modularen und erweiterbaren Aufbau des Choreography-Merger-Tools ist es problemlos möglich weitere Funktionalitäten hinzuzufügen.

## Literaturverzeichnis


[AODE11] Apache ODE, BPEL 1.1 und WS-BPEL 2.0 konforme OpenSource BPEL-Engine. Online: http://ode.apache.org.

[ATOM14] Apache Tomcat, *Java Servlet and JavaServer Pages Technologies Server*, Version 7.0.53. Online: http://tomcat.apache.org.

[CUI12] D. Cui, *Splitting BPEL Processes. Diplomarbeit, Universität Stuttgart*, Institut für Architektur von Anwendungssystemen, Deutschland. Online: http://elib.unistuttgart.de/opus/volltexte/2012/7605

[BER13] P. Berger, *Konsolidierung von BPEL Prozessmodellen im Kontext von Interaktionen über Fehlerbehandlungskonstrukte*. Diplomarbeit, Universität Stuttgart, Institut für Architektur von Anwendungssystemen, Nov. 2013.

[DEB13] P. Debicki, *Choreographie-basierte Konsolidierung von BPEL Prozessmodellen*. Diplomarbeit, Universität Stuttgart, Institut für Architektur von Anwendungssystemen, Feb. 2013.

[DKLW07] G. Decker, O. Kopp, F. Leymann, M. Weske. *BPEL4Chor: Extending BPEL for Modeling Choreographies*. In ICWS 2007, S. 296–303. IEEE Computer Society, 2007

[ECL14] Eclipse *BPEL Designer Plugin*, Version 1.0 M6, Online: http://www.eclipse.org/bpel/

[OAS07] OASIS, *Web Service Business Process Execution Language Version 2.0*, 11 April 2007. Online: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[WAG13] S. Wagner. *Choreography-based BPEL Process Consolidation*. Elektronische Übergabe des Posters, Universität Stuttgart, Institut für Architektur von Anwendungssystemen, 2013.

[WKL11] S. Wagner, O. Kopp, F. Leymann. *Towards Choreography-based Process Distribution In The Cloud*. Proceedings of the 2011 IEEE International Conference on Cloud Computing and Intelligence Systems. pp. 490-494, 2011.

[WKL12] S. Wagner, O. Kopp, F. Leymann. *Towards Verification of Process Merge Patterns with Allen's Interval Algebra*. Proceedings of the 4th Central-European Workshop on Services and their Composition (ZEUS 2012). pp. 1-8, 2012.

[WKL13] S. Wagner, O. Kopp, F. Leymann. *Consolidation of Interacting BPEL Process Models with Fault Handlers*. Proceedings of the 5th Central-European Workshop on Services and their Composition (ZEUS 2013), S. 1–7. CEUR Workshop Proceedings, Rostock, 2013

[WKL14] S. Wagner, O. Kopp, F. Leymann, *Choreography-based Consolidation of Multi-Instance BPEL Processes*, Universität Stuttgart, Institut für Architektur von Anwendungssystemen, 2014

Alle Online-URLs wurden zuletzt am 02.07.2014 geprüft.

## Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.

Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Stuttgart, den 09. Juli 2014 _____