Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3566

Konsolidierung von BPEL Prozessmodellen im Kontext von Interaktionen über Fehlerbehandlungskonstrukte

Peter Berger

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Sebastian Wagner
begonnen am:	30. Mai 2013
beendet am:	29. November 2013
CR-Klassifikation:	D.3.4, H.4.1

Kurzfassung

Bei der Akquirierung von Unternehmen ist es notwendig eine einheitliche Ablauforganisation zu schaffen. Dies wird durch die Konsolidierung von Geschäftsprozessen der Unternehmen, welche in einer Choreographie miteinander kommunizieren, erreicht. Dabei können die Prozesse Fehlerbehandlungen enthalten, die ebenfalls mit konsolidiert werden müssen. Hierfür wird, in der vorliegenden Arbeit, die Umwandlung von synchronen und asynchronen Kommunikationsmuster in Synchronisationsaktivitäten anhand der BPEL Konstrukte Fault Handler, Termination Handler, Event Handler und Compensation Handler (FCTE-Handler) untersucht. Durch diese Umwandlung kann der Kontrollfluss in den BPEL Konstrukten sich verändern und muss korrigiert werden. Als weiterer Aspekt werden Kontrollflussbeschränkungen betrachtet. Hierzu werden die Grenzüberschreitungen von Links analysiert, die in die FCTE-Handler hinein- und aus diesen hinausgehen. Das Auftreten von Grenzverletzungen wird durch die Umstrukturierung der einzelnen FCTE-Handler gelöst.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

1. Einleitung

Wo einst die Kapazitäten durch die Ausstattung des eigenen Rechenzentrums begrenzt waren, sind diese heute nahezu grenzenlos durch das Cloud Computing verfügbar. Ungeachtet dessen, ob es sich hierbei um Bandbreite, Rechenleistung oder Speicherplatz handelt, der Vorteil des Cloud Computing ist die flexible Bereitstellung der vorher genannten Ressourcen, welche an die aktuell vorliegenden Bedürfnisse des Kunden angepasst werden. Neben dieser sogenannten Elastizität spielt das Pay Per Use eine wichtige Rolle. Während bei einem betriebseigenen Rechenzentrum fixe Kosten auch bei geringer Nutzung anfallen, so wird bei Pay Per Use nur der tatsächliche Ressourcenverbrauch in Rechnung gestellt.

Diese Vorteile haben die Attraktivität erhöht, die Geschäftsprozesse in eine Cloud auszulagern. Dies nutzen viele Unternehmen, um ihre Prozesse zur Verfügung zu stellen und mit ihren Geschäftspartnern zu interagieren. Jedoch könnten die Geschäftspartner anhand der Offenlegung des Prozesses die jeweiligen Kosten ermitteln, weshalb in [Khao8] das folgende Verfahren entwickelt wurde, um dieses Problem zu lösen.

- **Split:** Extraktion der Teilprozesse, welche an der Kommunikation mit dem Geschäftspartner beteiligt sind.
- **Merge:** Konsolidierung von Prozessen, welche in einer Choreographie miteinander interagieren. In Bezug auf das Split Verfahren werden die extrahierten Teilprozesse konsolidiert.

Um das Vorgehen des Verfahrens genauer zu erläutern, wird ein Beispiel aus der Automobilbranche verwendet [WKL11]. In Abbildung 1.1 ist eine Choreographie zur Fertigung eines Sportwagens zu sehen. Es ist zum einen der Sportwagenhersteller SWH und zum anderen ein Motorenhersteller MH an der Produktion beteiligt. Beide Hersteller interagieren über ihre Prozesse miteinander. Für die Herstellung des Sportwagens startet der SWH die Entwicklung des Fahrgestells und sendet einen Auftrag für die Motorentwicklung an den MH. Nachdem beide Entwicklungen abgeschlossen sind und der Motor von MH an SWH geliefert worden ist, wird in Kooperation mit MH ein Prototyp des Sportwagens erstellt und Tests für das Fahrgestell und den Motor gegebenenfalls wiederholt durchgeführt. Nach erfolgreichem Abschluss der Tests kann die Produktion des Sportwagens beginnen. Aufgrund der engen Kooperation von SWH und MH bei der Erstellung des Prototypen und der Tests bietet sich nun eine kostensparende Auslagerung in eine Cloud an. Hierzu werden nun die Teilbereiche Prototyp und Test bei beiden Geschäftspartner extrahiert (Split), in einem neuen Prozess zusammengefügt (Merge) und in der Cloud ausgeführt. Die daraus entstehende Choreographie kann Abbildung 1.2 entnommen werden. Durch diese Teilauslagerung entfällt nun auf beiden Seiten der Hersteller die Rechenleistung für die Ausführung der

Schleifen. Zusätzlich werden die entstehenden Kosten durch Pay Per Use in der Cloud auf beide Geschäftspartner verteilt, was im Endeffekt zu Einsparungen führt. Ein weiterer Vorteil besteht in der Reduzierung des Kommunikationsaufkommens und somit der Einsparung von Bandbreite, da die Kommunikationsnachrichten in der neuen Schleife nun durch interne Prozessnachrichten ersetzt werden. Beide Hersteller können somit durch die Kooperation und Anwendung dieses Verfahrens eine Reduzierung der Kosten erreichen.

Bei der Anwendung des genannten Verfahrens muss auf die Fehlerverarbeitung der Prozesse geachtet werden. Denn, existiert für beide Herstellerprozesse eine Fehlerbehandlung (Exception Flow) für die Schleifenkonstrukte, die untereinander Informationen austauschen, so müssen auch diese für den neuen Prozess konsolidiert werden. Die Zielsetzung der Diplomarbeit ist es, einen entwickelten Prototypen dahingehend zu erweitern, dass Prozessmodelle mit Fehlerbehandlungen konsolidiert werden können und die implizierten Kontrollflüsse der Choreographie erhalten bleiben.

Gliederung

Diese Arbeit gliedert sich in folgender Weise:

Kapitel 2 – Grundlagen: Hier findet die Vermittlung der Grundlagen für diese Arbeit statt. Darunter fallen unter anderem BPEL, WSDL, BPEL4Chor.

Kapitel 3 – Konsolidierung von BPEL Prozess Modellen: Analyse und Ermittlung von Lösungsansätzen bei der Konsolidierung von Fault Handler, Termination Handler, Event Handler und Kompensation Handler.

Kapitel 4 – Implementierung: Beschreibung der programmiertechnischen Umsetzung aus Kaptiel 3.

Kapitel 5 – Zusammenfassung und Ausblick stellt ein Abschlussergebnis der Fehlerbehandlung und einen Ausblick auf zukünftige Konsolidierungsarbeiten mit dem Prototyp zur Verfügung.

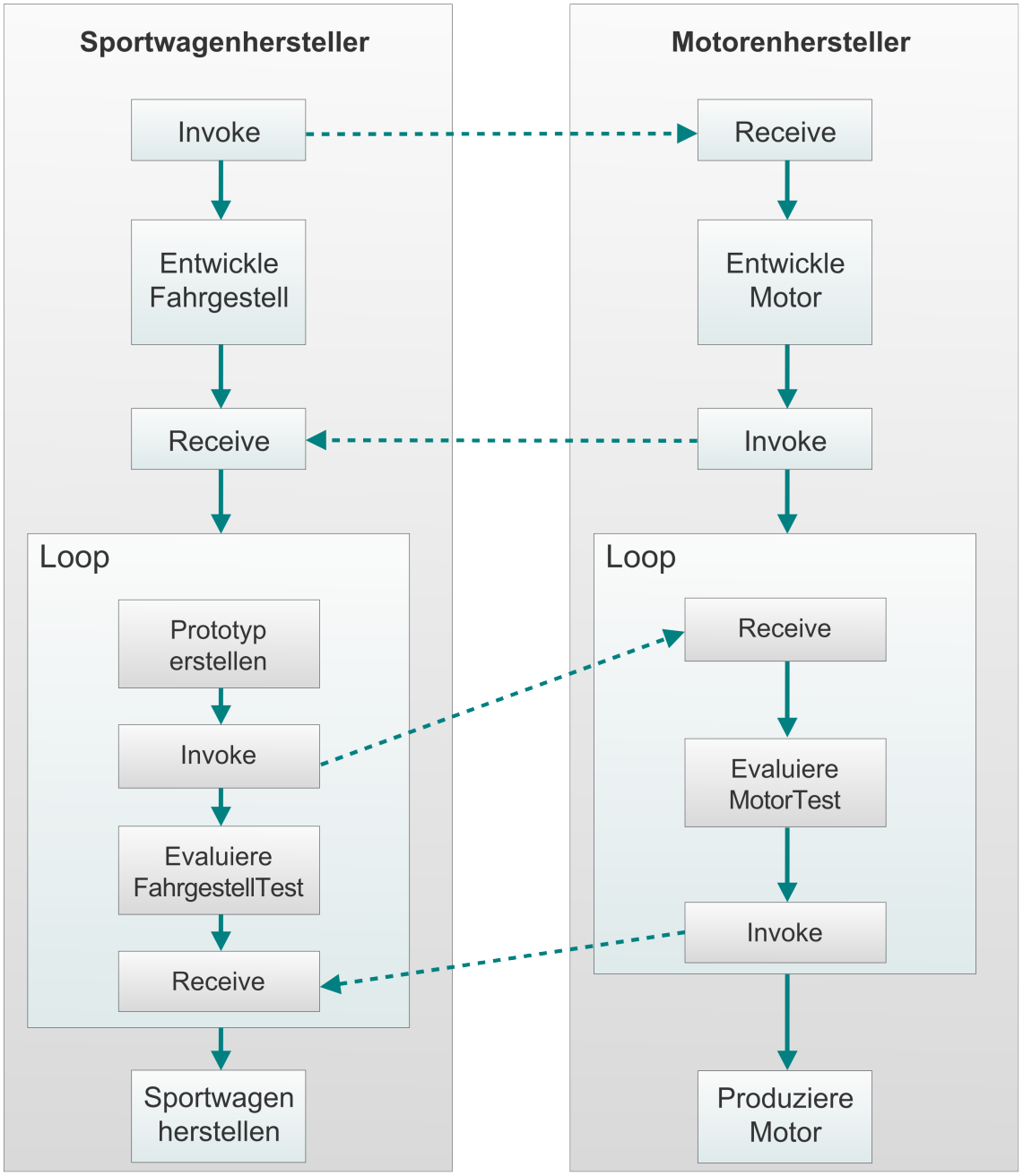


Abbildung 1.1.: Beispiel Autoentwicklung vor Split und Merge, adaptiert von [WKL11]

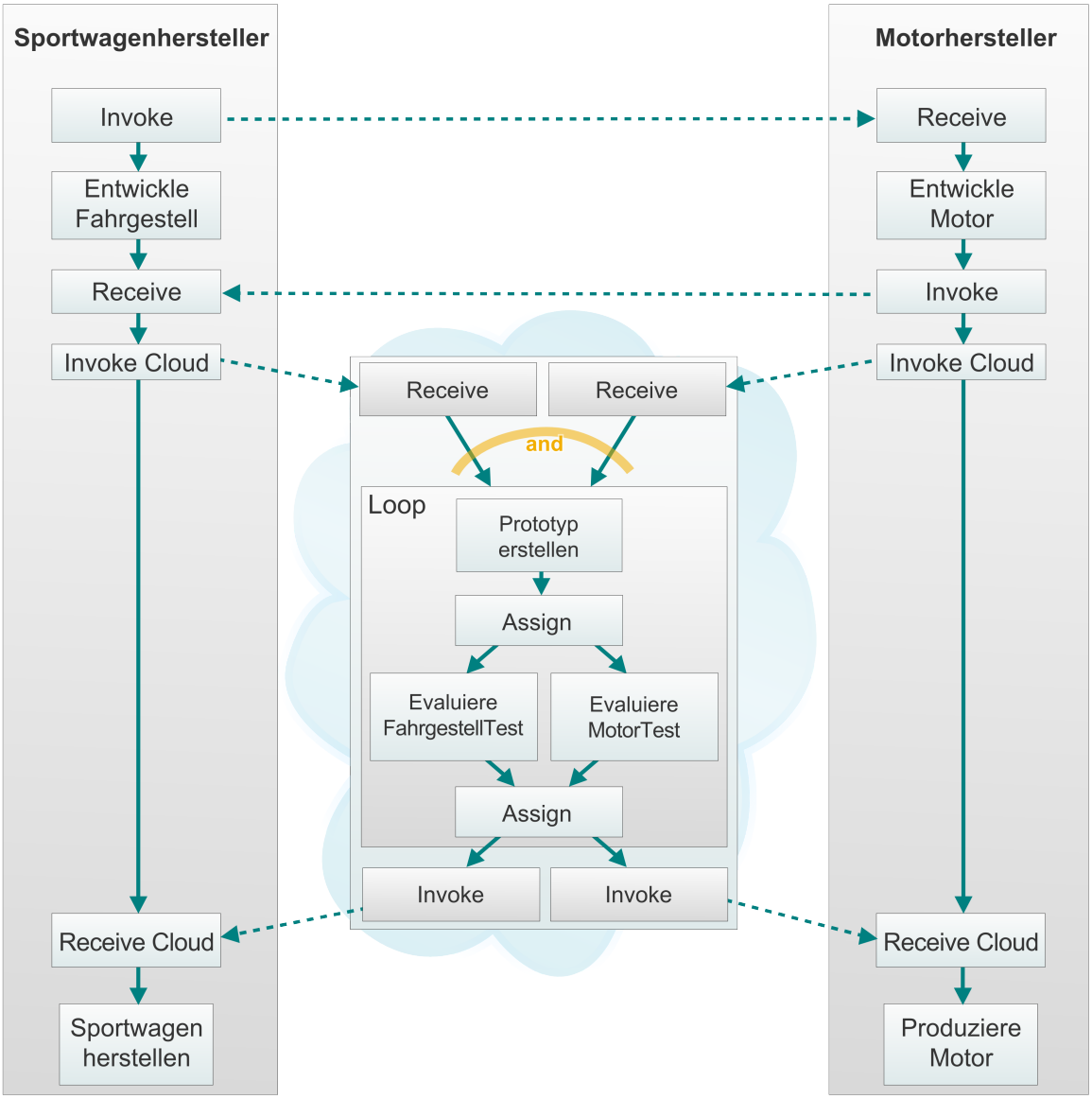


Abbildung 1.2.: Beispiel Autoentwicklung nach Split und Merge

2. Grundlagen

2.1. Web Services Description Language (WSDL)

Die Grundprinzipien einer Service Oriented Architecture (SOA) bestehen aus Publish, Find und Bind [WCL+05]. Zuerst wird der Service, der zur Verfügung gestellt werden soll, abstrakt definiert und in einem zentralen Register hinterlegt (Publish). Um nun den Zugriff auf einen Service zu erlangen, wird im zentralen Register nach einem Service gesucht, der den Suchkriterien am nächsten kommt (Find). Mit den zurückgelieferten Informationen kann nun auf den Service zugegriffen werden (Bind). Eine technische Umsetzung von SOA sind Web Services. Ihre funktionelle Beschreibung findet durch WSDL's statt. Die Abbildung 2.1 zeigt den groben strukturellen Aufbau einer WSDL 1.1, welche in der BPEL Spezifikation Version 2.0 verwendet wird.

Im Folgenden werden die Elemente einer WSDL genauer erläutert.

Abbildung 2.1.: Aufbau einer WSDL 1.1 [WCL+05]

- **Types**: Hier werden Datenstrukturen definiert.
- **Message**: Beschreibt den Aufbau der Nachricht, welche mit dem Web Service ausgetauscht wird.
- **PortType**: In PortType werden alle Operationen definiert, die ein Web Service anbietet. Für die Definition stehen vier Übertragungsmuster zur Verfügung [CCMW01]:
  - One-Way: Der Web Service erhält eine Nachricht.
  - Request-Response: Der Web Service erhält eine Nachricht und antwortet.
  - Solicit-Response: Der Web Service sendet eine Nachricht und erhält eine Antwort.
  - Notification: Der Web Service sendet eine Nachricht.
- **Binding**: Beschreibt in welchem Nachrichtenformat sowie mit welchem Transportprotokoll der Web Service aufgerufen werden kann.
- **Service**: Beschreibt wo man einen Web Service findet.

WSDL-Fehlerbehandlung

Es ist möglich, in WSDL's eigene Fehler für die Operationen zu definieren. Hierfür definiert man in Types oder in einem importierten Schema seinen Fehler. Als nächstes definiert man das Nachrichtenformat. Wie man in Listing 2.1 sehen kann, wird mit dem Attribut `element` auf den definierten Fehler in Types bzw. Schema referenziert. Nun muss man in PortType die Fault-`message` definieren. Hierfür fügt man einen Fault-Tag mit der Referenz auf die `message` hinzu, wie man in Listing 2.2 sehen kann. Als letzten Schritt ist noch die Angabe im `binding` notwendig. Mit diesen Deklarationen kann man nun den definierten Fehler verarbeiten, falls dieser bei der Kommunikation mit dem Web Service auftreten sollte. Die WSDL-Fehlerbehandlung gilt nur für die Übertragungsmuster Request-Response und Solicit-Response.

Listing 2.1 Abstrakte Beschreibung der Fehlernachricht.

```
<message name="MyException">
   <part name="fault" element="tns:MyException" />
</message>
```

Listing 2.2 Abstrakte Fehlerzuordnung zu einer Operation.

```
<portType name="HelloWorld">
       <operation name="...">
       <input message="..." />
              <output message="..." />
              <fault message="tns:MyException" name="MyException" />
       </operation>
</portType>
```

Listing 2.3 Konkrete Fehlerzuordnung.

```
<binding name="HelloWorldPortBinding" type="tns:HelloWorld">
       <operation name="sayHelloWorld">
              ...
              <fault name="MyException">
                     <soap:fault name="MyException" use="literal" />
              </fault>
       </operation>
</binding>
```

2.2. Business Process Execution Language (BPEL)

Mit BPEL (Version 2.0) lassen sich Geschäftsprozesse auf XML-Basis beschreiben. Zur Kommunikation und Beschreibung von Geschäftsprozessen werden Web Services (Abschnitt 2.1) eingesetzt. Hierbei wird BPEL als rekursives Aggregationsmodell für Web Services betrachtet. Es lassen sich ein oder mehrere Web Services zu einem neuen Web Service (Geschäftsprozess) zusammenfassen (Aggregation), der wiederum zur Definition eines weiteren Geschäftsprozesses dienen kann (Rekursion). [Ley07]

Die Definition eines Prozesses erfolgt über den `<process>` Tag, in welchem die Geschäftslogik implementiert wird. Zudem lässt sich ein Geschäftsprozess auf zwei Arten beschreiben, einen Ausführbaren und einen Abstrakten. Beim Ausführbaren Prozess werden alle notwendigen Informationen, die zur Ausführung des Prozesses benötigt werden, definiert. Der abstrakte Prozess hingegen ist unvollständig und nicht ausführbar, dadurch lässt er sich als Vorlage für weitere Prozesse nutzen. Weiterhin dient er dem Verbergen von Geschäftslogik, die beim Austausch mit anderen Geschäftspartnern nicht übertragen werden soll. [AAA+07]
Im Folgenden werden einige für diese Diplomarbeit wichtige BPEL-Konstrukte erklärt.

2.2.1. Scope & Isolated Scope

Ein Scope ist eine Aktivität, die eine Umgebung für die enthalten Aktivitäten zur Verfügung stellt, in welcher eigene Variablen, Fehlerbehandlungen, Terminierungsbehandlungen etc. definiert werden können. Eine Analogie zum Scope ist in Programmiersprachen, wie zum Beispiel Java, ein Block, welcher durch "{" und "}" gekennzeichnet wird. Das Listing 2.4 zeigt ein grobe Struktur eines Scopes. Vergleicht man diese Struktur mit der eines BPEL-Prozesses, so stellt man kaum Unterschiede fest. Ein Prozess jedoch ist keine Aktivität und enthält somit nicht die Attribute und Elemente einer Aktivität. Des Weiteren kann ein Prozess kein Isolated Scope (Abschnitt 2.2.1) sein sowie keinen Compensation Handler oder Termination Handler enthalten [AAA+07].

Listing 2.4 BPEL-Element Scope [AAA+07]

```
<bpel:scope name="MyScope">
       ...
       <variables>?...
       <faultHandlers>?...
       <compensationHandler>?...
       <terminationHandler>?...
       <eventHandlers>?...
       activity
</bpel:scope>
```

Variable

Im `<variables>` Tag können Variablen definiert werden, die nur für diesen Scope und seine verschachtelten Aktivitäten sichtbar sind. Bevor eine Variable verwendet werden kann, muss sie initialisiert werden. Die Initialisierung einer Variable findet entweder innerhalb des zugehörigen `<variable>` Tag oder in einem `<assign>` Tag statt. Ein `<assign>` Tag kann für Variablenzuweisungen oder Variableninitialisierungen verwendet werden.

Bei der Verwendung von Variablen ist dabei zu beachten, dass auch Variablenüberlagerung möglich sind. Enthält ein Scope S_A eine Variable V_A und einen Scope S_B. Sei in Scope S_B ebenfalls eine Variable V_B mit dem gleichen Namen wie von Variable V_A definiert. So gilt für alle Operationen in Scope S_B, die diesen Variablennamen verwenden, dass sie auf die Variable V_B und nicht auf die Variable V_A zugreifen.

Die Scope Zustände

Ein Scope kann folgende Zustände annehmen: [Sta05b, WSB]

- **Active**: Dieser Status wird einem Scope zugewiesen, sobald der Kontrollfluss den Scope erreicht und die verschachtelten Aktivitäten verarbeitet.
- **Completed**: Wurden alle Aktivitäten eines Scopes fehlerfrei verarbeitet, wechselt der Status auf Completed.
- **Compensated**: Befindet sich ein Scope im Status Completed und sein Compensation Handler wird ausgeführt, ändert sich der Status des Scopes auf Compensated.
- **Ended**: Dieser Status wird erreicht, wenn ein Scope aufgrund eines Fehlers sowie der Kontrollfluss beendet werden.
- **Faulted**: Tritt innerhalb eines Active Scopes ein Fehler auf, so wird der Fault Handler aufgerufen und der Scope wechselt in den Status Faulted.
- **Terminated**: Dieser Status wird gesetzt, wenn eine Scope aufgrund eines Fehlers, außerhalb seines Bereiches, terminiert wurde.

Fehlerbehandlung

Wie in anderen Sprachen existieren auch in BPEL Konstrukte für die Fehlerbehandlung. Diese sind der Termination Handler (Abschnitt 2.2.8), Fault Handler (Abschnitt 2.2.6) und Compensation Handler (Abschnitt 2.2.7). Beim Fault Handler ist es dabei möglich, eigene Fehler abzufangen, indem in einer WSDL Datei der Fehler definiert und in einem `<catch>` Block referenziert wird. Hierbei ist zu beachten, dass die Übertragungsmuster (Abschnitt 2.1) Solicit-Response und Notification nicht von BPEL Version 2.0 unterstützt werden und somit nur Request-Response zur eigenen Fehlerdefinition in einer WSDL zur Verfügung steht.

Isolated Scope

Isolated Scopes erlauben den parallelen Zugriff auf gemeinsame Resourcen wie zum Beispiel Variablen. Hierfür muss das Attribute `isolated="no"` auf `isolated="yes"` gesetzt werden. Darauffolgend dürfen keine verschachtelten Scopes mehr dieses Attribut auf `isolated="yes"` setzen. Greifen zwei parallele Scopes mit `isolated="yes"` auf Variablen außerhalb ihrer Scopes zu, so gewährleisten die Isolated Scopes, dass erst alle Zugriffe eines Scopes auf die Variablen abgeschlossen werden, bevor der andere Scope auf die gleichen Variablen zugreift. Dieses Verfahren ähnelt der Serialisierbarkeit bei Datenbanktransaktionen. Des Weiteren werden alle ausgehenden Links erst sichtbar, wenn der Isolated Scope auf Completed gesetzt wurde (siehe Abschnitt 2.2.1).

2.2.2. Invoke

Die Invoke Aktivität dient dem Aufruf von Web Services. Sie stellt die Übertragungsmuster One-Way und Request-Response zur Verfügung. Für Request-Response (siehe Abschnitt 2.1) werden die Attribute `inputVariable` und `outputVariable` sowie für One-Way nur die `inputVariable` angegeben. Werden für einen Web Service Aufruf keine Variablen benötigt, entfallen diese Attribute bei der Invoke Aktivität.
Sollte beim Aufruf eines Web Service ein Fehler auftreten, so besteht die Möglichkeit im `<invoke>` Tag (XML Repräsentation einer Invoke Aktivität) `<catch>` Blöcke oder einen `<catchAll>` Block zu definieren, um den Fehler abzufangen. Außerdem ist es möglich einen Compensation Handler (Abschnitt 2.2.7) zu installieren. Dies liegt an der Definition eines Invokes, denn für jedes Invoke existiert ein impliziter Scope.

2.2.3. Receive & Reply

Im Gegensatz zur Invoke Aktivität implementiert die Receive Aktivität Web Service Operationen, mit denen andere Geschäftsprozesse auf den eigenen Geschäftsprozess zugreifen können. Die Receive Aktivität blockiert hierbei so lange ihren Kontrollfluss, bis ein Aufruf eingeht. Wird die Receive Aktivität mit einem synchronen Kommunikationsaufruf gestartet, so kann mit einer Reply Aktivität eine synchrone Antwort an den Aufrufer gesendet werden.

Wird keine Antwort mittels der Reply Aktivität versendet, wird das Übertragungsmuster One-Way, ansonsten Request-Response, verwendet.

2.2.4. Correlation Sets

Die Ausführung von Prozessen findet in BPEL-Enines statt. Es wird für jeden Prozess eine oder mehrere Prozessinstanzen erzeugt und ausgeführt. Während seiner Laufzeit ist eine Prozessinstanz an einer Konversation mit einem oder mehreren Partnern beteiligt. Um eine Nachricht einer Prozessinstanz in einer BPEL-Engine eindeutig zuzuweisen werden Correlations Sets verwendet, in welchen sich CorrelationID's definieren lassen. Diese CorrelationID's werden bei der Kommunikation mit übertragen und lassen sich so einer Prozessinstanz eindeutig zuweisen.

2.2.5. Flow

Ein Flow dient der Parallelisierung und Synchronisierung von Aktivitäten. Es sollen z. B. zwei Aktivitäten A und B in einer dritten Aktivität C synchronisiert werden. Alle Aktivitäten sind in einem `<flow>` Tag, wie Listing 2.5 zeigt, definiert und werden somit parallel ausgeführt. Zur Synchronisierung wird dem Flow jeweils ein Link L_A, für Aktivität A nach Aktivität C, und ein Link L_B, für Aktivität B nach Aktivität C, hinzugefügt. Als nächstes werden die Links jeweils in einem Source, ausgehender Link, und einem Target, eingehender Link, gesetzt. Hierbei ist zu beachten, dass jeder Link nur in einem Source und einem Target definiert werden darf. Somit erhält Aktivität A ein `<source>` Tag mit Link L_A und Aktivität B ein `<source>` Tag mit Link L_B. In Aktivität C werden zwei `<target>` Tags mit jeweils Link L_A und Link L_B definiert. Dadurch erhält man die gewünschte Synchronisation der beiden parallel laufenden Aktivitäten A und B.

Listing 2.5 BPEL-Element flow [AAA+07]

```
<flow standard-attributes>
       standard-elements
       <links>?
              <link name="NCName">+
       </links>
       activity+
</flow>
```

Die Definition von `<source>` findet in der zugehörigen Aktivität im `<sources>` Tag und für `<target>` im jeweiligen `<targets>` Tag statt. Für jeden `<source>` Tag besteht dabei die Möglichkeit ein `<joinCondition>` Tag zu definieren, mit dem überprüft werden kann, ob der ausgehende Link mit `true` evaluiert werden soll. Für einzelne `<target>` Tags ist dies nicht möglich, jedoch kann eine `<joinCondition>` für alle `<target>` Tags im `<targets>` Tag angegeben werden. Mit diesem `<joinCondition>` Tag können alle eingehende Links in einer Bedingung über z. B. konjugierte oder disjunkte Operanden auf `true` oder `false` überprüft werden. Ist kein

<joinCondition> Tag im <targets> Tag definiert, so ist das Standardergebnis die Disjunktion aller eingehender Links.

Dead-Path Elimination

Mit Dead-Path Elimination (DPE) [AAA+07, Sta05a] lassen sich tote Pfade eliminieren. Ein solcher Pfad kann wie folgt entstehen. Ein Scope S_A enthält einen Fault Handler FH_A (siehe Abschnnitt 2.2.6). Scope S_A und Fault Handler FH_A synchronisieren mittels einem Link in Aktivität A. Jeder Link kann die Werte $S = \{true, false, undefined\}$ annehmen und hat bei Prozessbeginn den Wert `undefined`. Wechselt der Zustand von Scope S_A auf Completed (siehe Abschnitt 2.2.1), wird der Link zur Aktivität A vom Wert `undefined` auf `true` gesetzt. Da in Scope S_A kein Fehler auftritt, bleibt der Wert des Links vom Fault Handler FH_A `undefined`. Die Aktivität A kann durch diese Konstellation nicht ausgeführt werden, da ein toter Pfad vom Fault Handler FH_A zur Aktivität A entstanden ist. Hierdurch kann keine Validierung der <joinCondition> durchgeführt werden, da eine Validierung mit `undefined` Links nicht möglich ist. Durch das Setzen des Attributs `suppressJoinFailure=true` [AAA+07] wird DPE aktiviert. Tote Pfade werden daraufhin mit dem Wert `false` propagiert. Beide Links können nun im <joinCondition> Tag der Aktivität A validiert ($(true \vee false) = true$) und die Aktivität A ausgeführt werden.

Kontrollflussbeschränkung

Jeder Flow kann mehrere Aktivitäten enthalten, die miteinander über Links verbunden sein können. Jedoch gibt es hierbei folgende Restriktionen (siehe Abbildung 2.2):

- Links dürfen die Grenzen von wiederholbaren Konstrukten (<while>, <repeatUntil>, <forEach>, <eventHandlers>) oder des <compensationHandler> (siehe Abschnitt 2.2.7) nicht überschreiten.
- Die Grenzen von <catch>, <catchAll> (siehe Abschnitt 2.2.6) oder des <terminationHandler> (siehe Abschnitt 2.2.8) dürfen nur von innen (Quellaktivität) nach außen (Zielaktivität) überquert werden. Ein Link darf somit nur von der Quellaktivität (Source) zur Zielaktivität (Target) zeigen.
- Ein Link darf keinen Kontrollzyklus erzeugen.

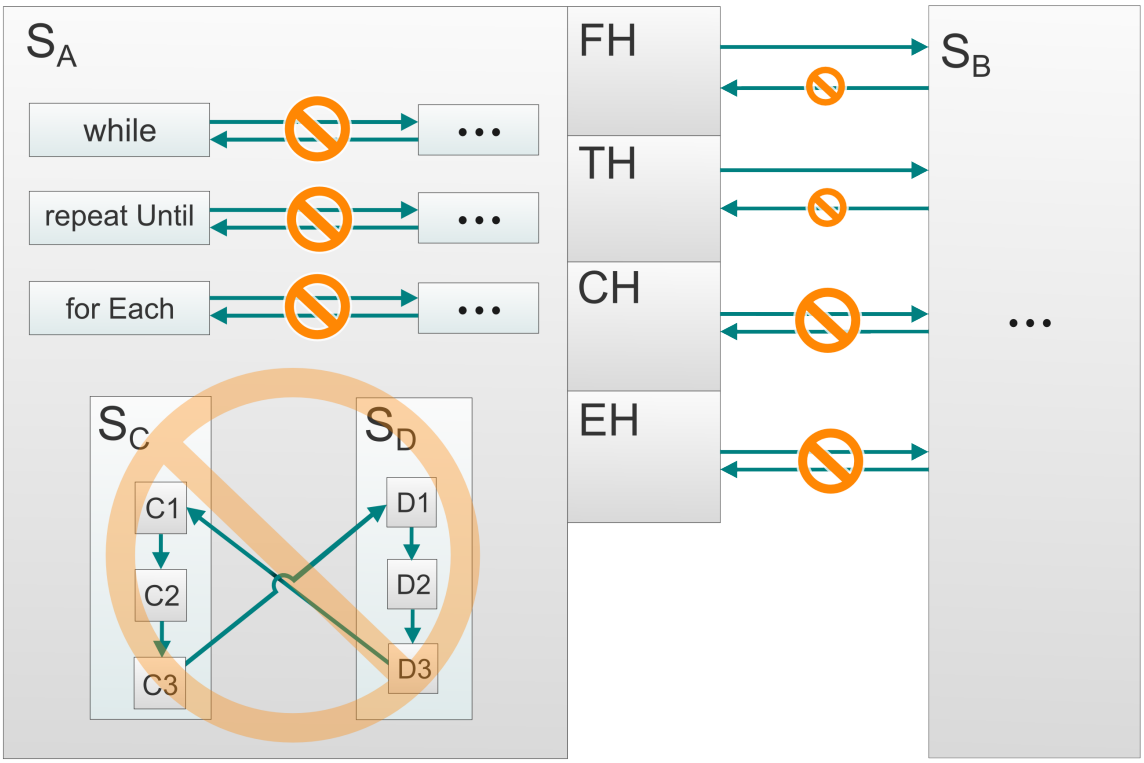


Abbildung 2.2.: Kontrollflussbeschränkungen

2.2.6. Fault Handler

Der Fault Handler stellt die Verarbeitung von Fehlern innerhalb eines Scopes zur Verfügung. Tritt in einem Scope ein Fehler auf, so wechselt der Status auf Faulted (Abschnit 2.2.1) und alle im Scope laufenden Aktivitäten werden über den Termination Handler deaktiviert. Aus diesem Status kann der zugehörige Compensation Handler nicht mehr erreicht werden, weil dieser mit dem Status Faulted vom Scope deinstalliert wird.

Für die Fehlerverarbeitung in BPEL steht zum einen der `<catchAll>` Tag und zum anderen der `<catch>` Tag zur Verfügung. Der `<catchAll>` Tag fängt alle in einem Scope auftretenden Fehler ab, während der `<catch>` Tag nur auf speziell definierte Fehler reagiert.

Ein Fehler wird wie folgt verarbeitet [AAA+07]:

- Suche `<catch>` Tag und verarbeite den zugehörigen Fehler. Hierfür werden die `<catch>` Tags nach folgenden Bedingungen, jeweils der Reihenfolge nach, abgearbeitet:
  1. `faultName` und `faultVariable` sind definiert und stimmen mit dem Fehler überein.
  2. `faultName`, `faultVariable` und `faultMessageType`/`faultElement` müssen zutreffen.

Listing 2.6 BPEL-Element faultHandlers [AAA+07]

```
<faultHandlers>
       <catch faultName="QName"? faultVariable="BPELVariableName"? (
           faultMessageType="QName" | faultElement="QName" )? >*
              activity
       </catch>
       <catchAll>?
              activity
       </catchAll>
</faultHandlers>
```

3. Nur der `faultName` ist im `<catch>` definiert.
4. Nur die `faultVariable` ist definiert. Übereinstimmung anhand des Datentyps.
5. `faultVariable` und `faultMessageType`/`faultElement` sind im `<catch>` definiert.

- Falls kein `<catch>` Tag für den geworfenen Fehler gefunden wird, führe die allgemeine Fehlerbehandlung im `<catchAll>` Tag aus.
- Wird ebenfalls kein `<catchAll>` Tag gefunden, wird der Standard Fault Handler ausgeführt.

Eine Weitergabe des Fehlers an einen umgebenen Scope ist mittels dem `<rethrow>` Tag möglich.

2.2.7. Compensation Handler

Die Kompensation von Geschäftslogik ist ein fundamentaler Bestandteil von BPEL, um erfolgreich ausgeführte Geschäftslogik rückgängig zu machen. Durch die Verwendung von `<compensationHandler>` Tags (siehe Listing 2.7) in Scopes oder in Invokes stellt man für diese Blöcke eine Kompensationslogik zur Verfügung. Über die Kompensationsaktivitäten `<compensate>` und `<compensateScope target="ScopeNameOderInvokeName">` kann die Kompensationslogik gestartet werden, vorausgesetzt die zu kompensierenden Scopes befinden sich im Status Completed. Mit dem `<compensate>` Tag startet man die Standard Kompensationsreihenfolge, während man mit dem `<compensateScope>` Tag die Kompensation eines speziellen Scopes/Invokes startet. Über das Attribut `target` wird dann auf die Aktivität mit dem `<compensationHandler>` Tag referenziert. Zudem dürfen die Kompensationsaktivitäten nur in den Tags `<catch>`, `<catchAll>`, `<compensationHandler>` und `<terminationHandler>` verwendet werden, um eine Kompensation zu starten.

Listing 2.7 BPEL-Element compensationHandler [AAA+07]

```
<compensationHandler>
       activity
</compensationHandler>
```

Kompensation innerhalb Fault Handler und Compensation Handler

Innerhalb eines Fault Handler, Compensation Handler oder Termination Handler (FCT-Handler) können gegebenenfalls Fehler auftreten. Tritt ein Fehler direkt in einem FCT-Handler auf, ohne durch einen umgebenden Scope und dessen Fault Handler abgefangen zu werden, so werden alle Aktivitäten innerhalb des FCT-Handler terminiert und der Fehler wird von den einzelnen Handlern wie folgt verarbeitet:

- **Fault Handler:** Propagiere Fehler an den Fault Handler des umgebenden Scopes
- **Termination Handler:** Ignoriere Fehler
- **Compensation Handler:** Propagiere Fehler an Aufrufer des Compensation Handler

Existiert hingegen ein umgebender Scope in dem FCT-Handler, wie in Abbildung 2.3, so kann mit einem speziellen Fault Handler der Fehler abgefangen und verarbeitet werden. Ist dies nicht der Fall, so wird der Standard Fault Handler aktiv, der eine Standard Kompensationreihenfolge der Aktivitäten startet und danach ein `<rethrow>` ausführt. Bei der Verwendung eines Scopes innerhalb eines FCT-Handler ist zu beachten, dass dieser Scope, auch als root-Scope bezeichnet, keinen Compensation Handler enthalten darf, weil dieser von keiner Stelle aus im Prozess erreichbar ist.

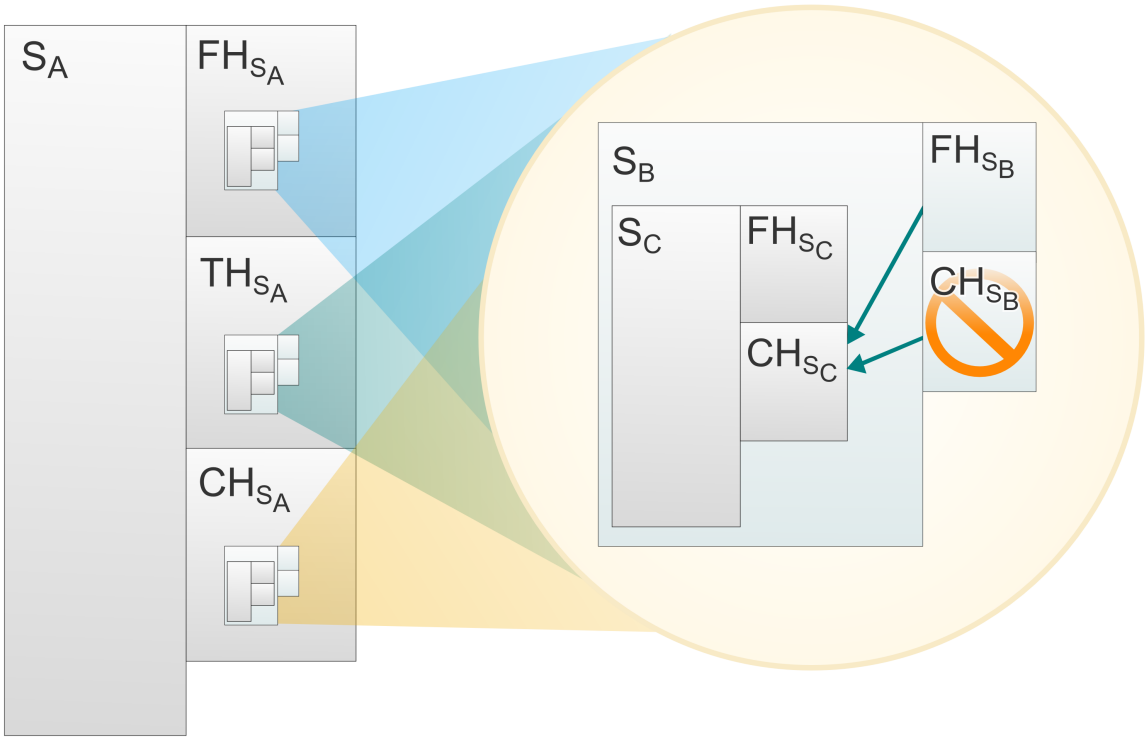


Abbildung 2.3.: Fehler in FCT-Handler abgeleitet von [AAA+07]

Abbildung 2.3 demonstriert wie ein root-Scope innerhalb eines FCT-Handler definiert werden muss. Scope S_A enhält FCT-Handler, die jeweils einen root-Scope beinhalten, der wie Scope S_B definiert werden kann. Scope S_B wird als root-Scope verwendet und darf infolgedessen keinen Compensation Handler enthalten. Innerhalb des Fault Handler FH_B von Scope S_B kann in `<catch>` oder `<catchAll>` die Kompensation von Scope S_C über `<compensate>` oder `<compensateScope>` gestartet weden. Wurden keine Fault Handler definiert, startet der Standard Fault Handler, welcher automatisch eine Kompensation aller Aktivitäten mit `<rethrow>` enthält und die Kompensation von Scope S_C im Fehlerfall auslösen kann.

Standard Kompensationsreihenfolge

Um die Standard Kompensationsreihenfolge eines Prozesses zu erzeugen, müssen zwei Regeln aus der BPEL Spezifikation [AAA+07] beachtet werden. Mit der ersten Regel wird die Reihenfolge in Bezug auf den Prozessablauf definiert. Regel zwei definiert das Verhalten von Scopes, die nicht Isolated Scopes (siehe Abschnitt 2.2.1) sind. Sie behandelt alle Scopes wie Isolated Scopes, wodurch keine Zyklen entstehen können. Dies wird dadurch verhindert, dass der Kontrollfluss erst dann vorgesetzt wird, wenn der Isolated Scope den Status auf Completed setzt. Ohne die zweite Regel könnten sonst Konstellationen entstehen, dass zum Beispiel eine Aktivität A in Scope S_A vor einer Aktivität B in Scope S_B ausgeführt wird und diese wiederum vor einer Aktivität C in Scope S_A. Für die Anwendung der zweiten Regel sind zusätzliche Definition notwendig [AAA+07], welche in Abbidlung 2.4 aufgelistet sind.

Definition (Control Dependency): Wenn eine Aktivität A beendet werden muss, bevor eine Aktivität B ausgeführt wird, aufgrund eines existierenden Kontrollpfades von Aktivität A nach Aktivität B in einer Prozessdefinition, dann sagt man, dass Aktivität B eine Control Dependency zu Aktivität A hat. Control Dependencies können durch Kontrolllinks in `<flow>` als auch durch Konstrukte wie `<sequence>` entstehen. Ein Kontrollfluss durch `<throw>` stellt keine Control Dependency dar.

Regel 1: Scope S_B habe eine Control Dependency zu Scope S_A und beide Scopes befänden sich im Status Completed (siehe Abschnitt 2.2.1). Außerdem müssten beide Scopes aufgrund eines Standard Kompensationsvorganges kompensiert werden. Dann gilt, dass der Compensation Handler von Scope S_B ausgeführt und erfolgreich beendet werden muss, bevor der Compensation Handler von Scope S_A gestartet werden kann.

Definition (Peer-Scopes): Zwei Scopes S_A und S_B werden als Peer-Scopes bezeichnet, wenn sie direkt vom gleichen Scope umschlossen werden (einschließlich des Prozess Scopes).

Definition (Scope-Controlled Set): Eine Aktivität A ist in dem Scope-Controlled Set von Aktivitäten des Scopes S, wenn Aktivität A selbst Scope S entspricht oder Aktivität A von Scope S umgeben wird, ungeachtet der Verschachtelungstiefe.

Definition (Peer-Scope Dependency): Scope S_A und Scope S_B seien Peer-Scopes. Es existiere eine Aktivität B innerhalb des Scope-Controlled Set von Scope S_B und eine Aktivität A

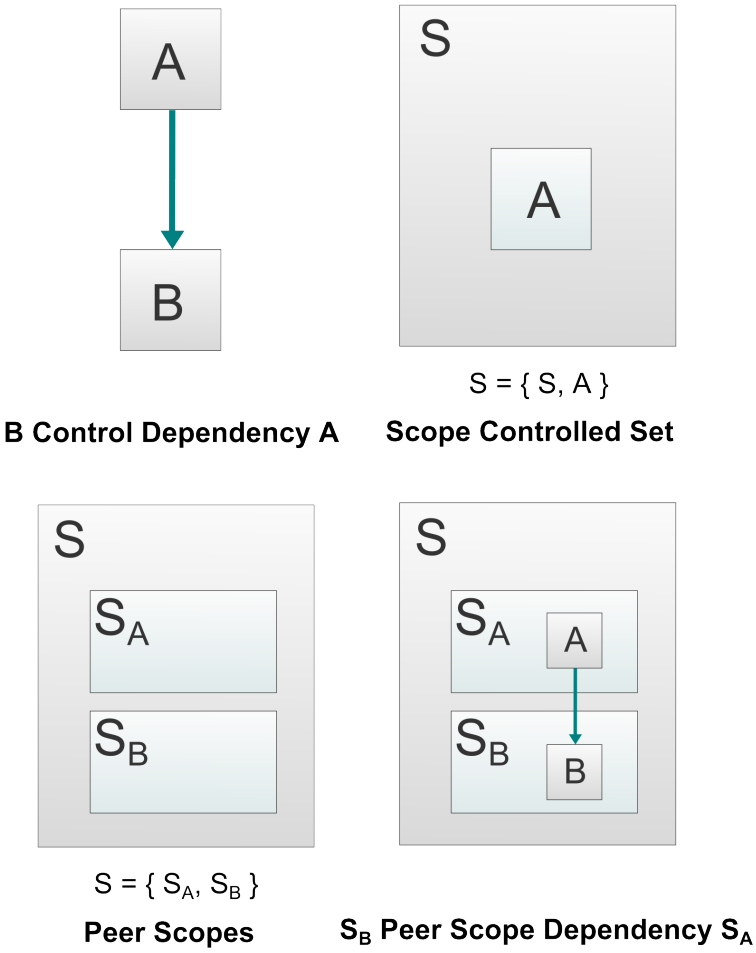


Abbildung 2.4.: Grafische Darstellung der Kompensationsdefinitionen

innerhalb des Scope-Controlled Set von Scope S_A. Es gilt, dass Scope S_B eine direkte Peer-Scope Dependency zu Scope S_A hat, wenn Aktivität B eine Control Dependency zu Aktivität A hat. Die Peer-Scope Dependency Relation ist die transitive Schlussfolgerung aus der direkten Peer-Scope Dependency Relation.

Regel 2: Die Peer-Scope Dependency Relation darf keinen Zyklus enthalten. Es dürfen somit im Prozess keine Peer-Scopes S_A und S_B existieren, bei denen Scope S_A eine Peer-Scope Dependency zu S_B und S_B eine Peer-Scope Dependency zu S_A hat.

2.2.8. Termination Handler

Der Termination Handler ist für die Deaktivierung von Event Handler und die Terminierung von laufenden Aktivitäten zuständig. Nach der Deaktivierung und Terminierung werden die definierten Aktivitäten im Termination Handler ausgeführt. Es können die selben Aktivitäten wie beim Fault Handler verwendet werden. Jedoch werden nicht abgefangene Fehler, die im Termination Handler auftreten, nicht wie im Fault Handler, an die übergeordnete Aktivität weitergeleitet, da diese die Terminierung beauftragt hat oder sich selbst in der Terminierung befindet.

Die Terminierung eines Scopes ist nur zulässig, wenn sich der Status des Scopes nicht in Faulted befindet. Bei der Terminierung werden Aktivitäten eines Scopes unterschiedlich behandelt:

- **dürfen beenden:** Kurzlebige Aktivitäten wie zum Beispiel `<assign>` oder `<rethrow>` dürfen ihre Tätigkeit beenden.
- **werden unterbrochen:** Aktivitäten wie `<invoke>` oder `<flow>` müssen hingegen unterbrochen werden.
- **darf nicht unterbrochen werden:** Die `<exit>` Aktivität darf nicht unterbrochen werden, sie beendet einen Prozess, jedoch nicht laufende Terminierungsverarbeitungen, Fehlerverarbeitungen oder Kompensationsverarbeitungen.
- **propagiert:** Die Aktivitäten `<compensate>` und `<compensateScope>` werden unterbrochen, indem die Terminierung zu den Compensation Handler propagiert wird und im Compensation Handler die Terminierung angestoßen wird.

Ein Termination Handler wird wie im Listining 2.8 definiert.

Listing 2.8 BPEL-Element terminationHandler [AAA+07]

```
<terminationHandler>
       activity
</terminationHandler>
```

2.2.9. Event Handler

Event Handler können in jedem Scope sowie auch in einem Prozess definiert werden. Sie können parallel laufen und reagieren auf eingehende Nachrichten von WSDL-Operationen (nachrichtengesteuert) oder auf definierte Alarm-Ereignisse (zeitgesteuert). Beim Auftreten eines dieser beiden Ereignisse wird der zugehörige Scope ausgeführt. Die Lebensdauer eines Event Handlers ist abhängig von dem Scope, in dem er definiert wurde. Solange dieser Scope den Status Active hat, ist der Event Handler erreichbar.

2.3. BPEL4Chor

BPEL ist eine Orchestrierungsprache und legt somit die Reihenfolge sowie Bedingungen für einen Nachrichtenaustausch für einzelne Prozesse fest. Man erhält hierdurch einen Überblick über einen individuellen Prozess und wie dieser mit seinen Partner kommuniziert. Eine Choreographie hingegen, liefert eine globale Sicht auf alle Prozesse und wie diese miteinander in Verbindung stehen. Ebenfalls werden, wie bei einer Orchestrierung, die Reihenfolge und Bedingungen für den Nachrichtenaustausch definiert. Um nun eine globale Sicht auf alle Prozesse zu erhalten, wurde BPEL4Chor geschaffen, welches die Orchestrierungssprache BPEL erweitert und somit eine Choreographiesprache erschafft.

BPEL4Chor ist in drei Artefakte [DKLW07, Dec09] untergegliedert (siehe Abbildung 2.5):

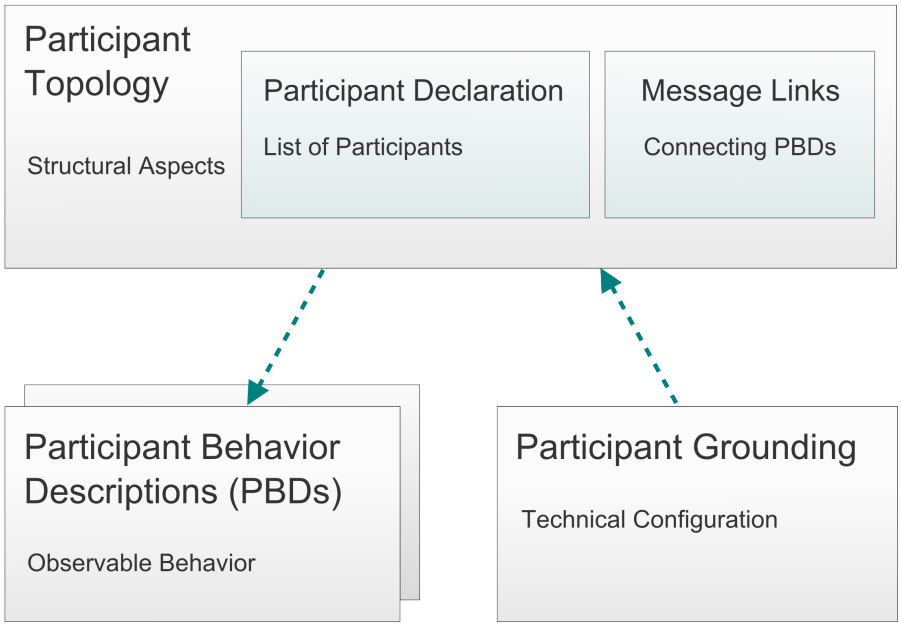


Abbildung 2.5.: BPEL4Chor Artefakte [Dec09]

2.3.1. Participant Topology

Die Participant Topology definiert die strukturellen Beziehungen einer Choreographie durch folgende Strukturen:

participant types: Definition aller an einer Choreographie beteiligten Prozesse (Participant Behavior Descriptions).

participants: Definition der Teilnehmer einer Choreographie sowie deren Verbindung untereinander. Sie werden als Instanzen von `participant types` gesetzt.

message links: In `message links` wird der Nachrichtenaustausch zwischen den `participants` definiert.

2.3.2. Participant Behavior Descriptions (PBDs)

Ein PBD definiert den Kontrollfluss und Datenfluss eines an der Choreographie beteiligten Prozesses. Die Basis für einen PBD basiert auf einen abstrakten Prozess aus BPEL. Durch die Verwendung von abstrakten Prozessen besteht die Möglichkeit der Abstrahierung der Datentypen sowie der Kommunikationspartner. Um diese Abstrahierung zu erreichen, ist es nötig, dass alle Aktivitäten eindeutig referenziert werden. Da jedoch nicht alle Aktivitäten in BPEL über ein `name` Attribut verfügen, zum Beispiel `onMessage`, wird für BPEL4Chor das Attribut `wsu:id` für alle Aktivitäten eingeführt. Mit diesem Attribut lassen sich nun alle Aktivitäten eindeutig in den PBDs identifizieren und in der Participant Topology unter `message links` für einen Nachrichtenaustausch referenzieren.

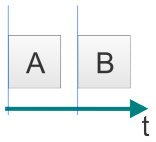
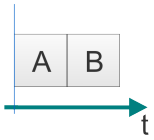
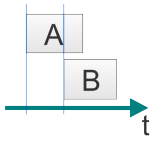
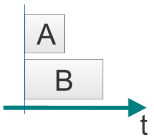
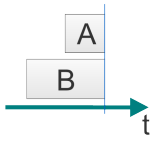
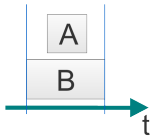
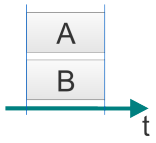
2.3.3. Participant Grounding

In der Participant Grounding finden alle technischen Zuweisungen, wie XML-Typen oder WSDL-Operationen, statt. Die PBDs und Participant Topology sind komplett abstrahiert und enthalten keine technischen Informationen. Hierdurch wird die Verbindung zwischen BPEL und WSDL aufgelöst und die Möglichkeit geschaffen andere Schnittstellenbeschreibungssprachen zu verwenden. Außerdem können dann plattformspezifische Informationen auf einfachem Wege erstellt und durch den Austausch der Participant Groundings verändert werden.

2.4. Allen Kalkül

Das Allen Kalkül beschäftigt sich mit den zeitlichen Zusammenhängen zweier Intervalle A und B. Hierzu wurden von Allen 13 Basisrelationen definiert ([WKL12]), welche aus der Tabelle 2.1 entnommen werden können.

Im weiteren Verlauf wird das Allen Kalkül für die Überprüfung auf Änderungen zweier Prozessfragmente verwendet. Sei ein Prozessfragment *PF2* ein Ergebnis eines Algorithmus mit der Eingabe vom Prozessfragment *PF1* und es gilt die Bedingung, dass das Prozessfragment *PF2* nach Anwendung des Algorithmus seine zeitlichen Zusammenhänge zwischen den Aktivitäten beibehält. Betrachtet man die Abbildung 2.6, so erkennt man, dass sich die Reihenfolge der Aktivitäten verändert hat. Die Aktivität *A1* wird nach der Anwendung des Algorithmus nicht mehr vor den Aktivitäten *A2* und *A3* ausgeführt, sondern danach. Da eine Auswertung zweier Grafiken nicht immer so einfach ist, wie in diesem Beispiel, verwendet man Allen's Intervallmatrizen. Hierbei liefert jede Zeile die Relation zweier Intervalle und jede Spalte die inverse Relation dieser Intervalle. Eine reflexive Relation eines Intervalls ist nach dem Allen Kalkül nicht erlaubt und entspricht somit der leeren Menge in

Relation	Bedeutung	Grafische Darstellung
A **<** B B **>** A	A wird vor B ausgeführt B wird nach A ausgeführt	
A **m** B B **mi** A	A trifft auf B ohne Verzögerung B wird von A ohne Verzögerung getroffen	
A **o** B B **oi** A	A überlappt B B wird von A überlappt	
A **s** B B **si** A	A startet zur selben Zeit wie B B wird zur selben Zeit wie A gestartet	
A **f** B B **fi** A	A endet zur selben Zeit wie B B wird zur selben Zeit wie A beendet	
A **d** B B **di** A	A läuft während B Während B läuft, läuft auch A	
A **e** B	Beide Intervalle starten und enden zur selben Zeit	

Abkürzungen: i = invers, m = meets, o = overlaps, s = starts, f = finishes, d = during, e = equals

Tabelle 2.1.: Allen's Basis Intervallrelationen [WKL12]

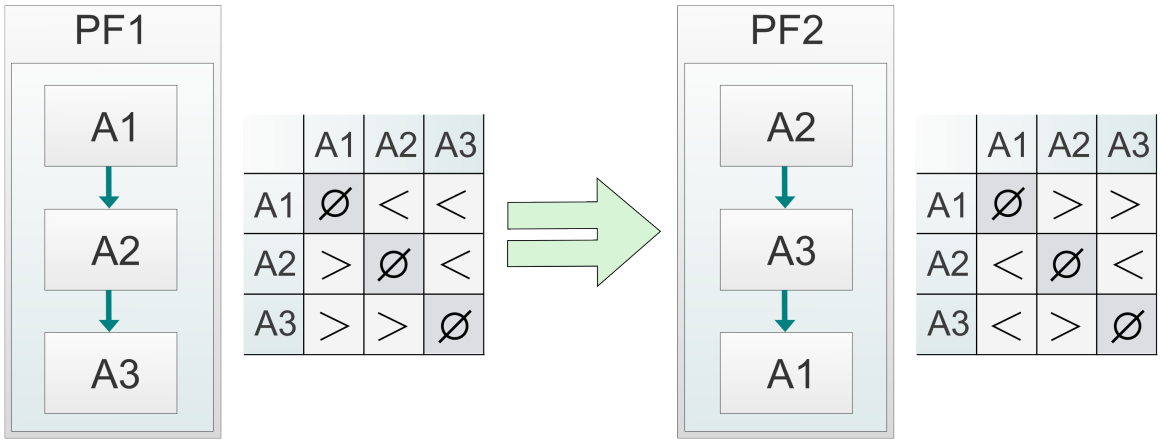


Abbildung 2.6.: Beispiel für die Anwendung des Allen Kalküls

der Intervallmatrix. Sollte für eine Relation zweier Intervalle alle Allen Intervallrelationen gelten, so verwendet man R={<,>,m,mi,o,oi,s,si,f,fi,d,di,e}. Wie man nun aus Abbildung 2.6 entnehmen kann, stimmen beide Intervallmatrizen nicht überein und somit ist die oben definierte Bedingung nicht erfüllt und der Algorithmus funktioniert nicht wie gewünscht.

2.5. Bestehendes System

Das bestehende System, basierend auf Cui [Cui12] und Debicki [Deb13], nimmt als Eingabe eine ZIP entgegen, welche eine Choreographie in BPEL4Chor und die zugehörigen WSDLs enthält, siehe Abbildung 2.7.

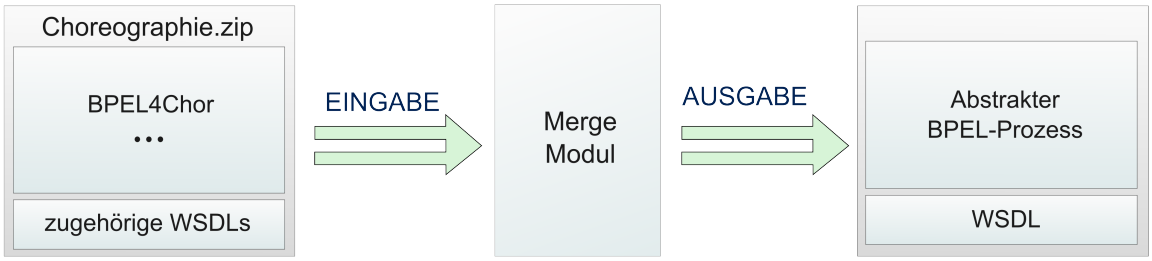


Abbildung 2.7.: Konsolidierung

Im Merge Modul findet dann die Umwandlung in einen abstrakten BPEL-Prozess und die Konsolidierung, der generelle Austausch der Kommunikationsaktivitäten durch Synchronisationsaktivitäten, statt. Hierzu wird ein Prozess erstellt, der einen `<flow>` enthält. Dieser `<flow>` wiederum enthält Scopes, welche die Logik der einzelnen PBDs enthalten. Im nächsten

Schritt werden alle `message links` (siehe Abschnitt 2.3.1) auf bestimmte Konsolidierungsmuster untersucht und anhand dieser umgewandelt. Alle `message links`, die mit keinem Muster übereinstimmen, werden nicht umgewandelt und in einer Liste NMML (Non-Mergeable-Message-Links) gespeichert. Im letzten Schritt werden die technischen Informationen aus der Participant Grounding (siehe Abschnitt 2.3.3) genommen und dem Prozess hinzugefügt. Das Ergebnis ist ein konsolidierter abstrakter Prozess mit seinen zugehörigen WSDLs.

3. Konsolidierung von BPEL Prozess Modellen

Unter Konsolidierung von BPEL Prozess Modellen wird das Zusammenführen (Merge) von Geschäftsprozessen in einen neuen Prozess verstanden. Die Umsetzung der Konsolidierung im bestehenden System erfolgt in vier Schritten [Wag13].

1. **Schritt:** Analyse der Kontrollflussrelationen
2. **Schritt:** Erstellung des Konsolidierungsvorgehens
3. **Schritt:** Umwandlung der Kommunikationsaktivitäten in Synchronisationsaktivitäten
4. **Schritt:** Kontrollflussverletzungen auflösen

In Schritt 1 werden alle synchronen und asynchronen Kommunikationsaktivitäten erfasst. Als nächstes, Schritt 2, wird der Konsolidierungsprozess erstellt. Hierfür werden die an der Choreographie beteiligten Prozesse jeweils als Scope einem Prozess P_{merged} hinzugefügt. Jeder Scope bekommt zusätzlich einen Fault Handler, damit sie im Fehlerfall die anderen Scopes, ehemals Prozesse, nicht beeinflussen und diese ihre Arbeit fortsetzen können. In Schritt 3 findet die Umwandlung der Kommunikationsaktivitäten in Synchronisationsaktivitäten statt (siehe Abbildung 3.1). Bei der Verwendung von asynchronen Kommunikationsaktivitäten ruft eine Invoke Aktivität eine Receive Aktivität auf und lässt dann den Kontrollfluss weiterlaufen. Um dieses Konstrukt auch nach der Konsolidierung beizubehalten wird die Invoke Aktivität durch eine Assign Aktivität und die Receive Aktivität durch eine Empty Aktivität ersetzt und ein Link von Assign nach Empty gezogen. Bei synchronen Kommunikationsaktivitäten erhält die Invoke Aktivität noch eine Antwort von einer Reply Aktivität und blockiert bis zum eintreffen der Antwort den Kontrollfluss. Wie bei den asynchronen Kommunikationsaktivitäten wird die Invoke und Receive Aktivität durch eine Assign Aktivität AN_A und eine Empty Aktivität E_B mit einem zugehörigen Link ersetzt. Die Reply Aktivität wird ebenfalls durch eine Assign Aktivität AN_B ersetzt. Zusätzlich wird nach der Assign Aktivität AN_A eine weitere Empty Aktivität E_A eingeführt und ein Link von AN_B nach E_A gesetzt, wodurch die Synchronisation erreicht wird. Die verwendeten Assign Aktivitäten simulieren dabei die Variablenübergabe, die zuvor von den Kommunikationsaktivitäten durchgeführt wurde. Im letzten Schritt, Schritt 4, findet die Korrektur von Kontrollflussverletzungen statt (siehe Abschnitt 2.2.5). Bei der Durchführung dieser Konsolidierungsschritte ist zu beachten, dass die Prozesse der Choreographie nur einmal instanziiert werden dürfen (One-to-One Interaktion). One-to-Many Interaktionen, wie in [Wag13] beschrieben, werden zur Zeit nicht unterstützt.

In diesem Kapitel findet die Konsolidierung von Fault Handler, Compensation Handler, Termination Handler und Event Handler, im Folgenden als FCTE-Handler bezeichnet, statt.

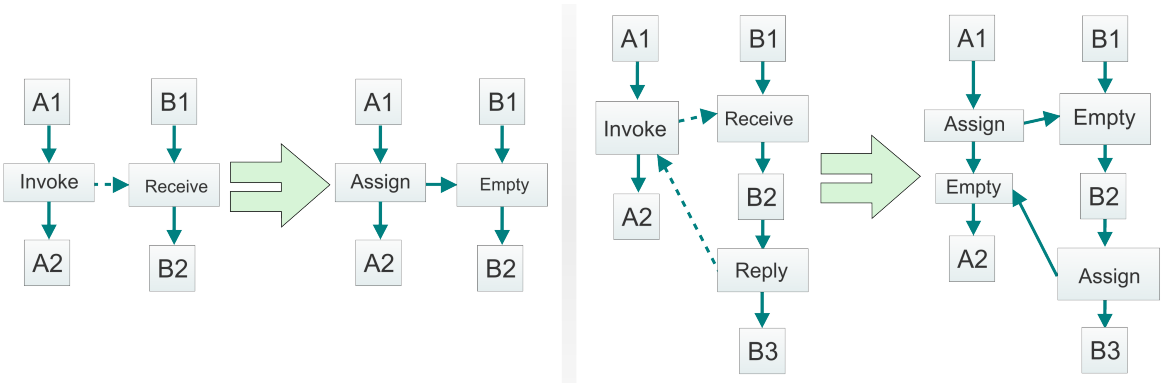


Abbildung 3.1.: Asynchrone und Synchrone Konsolidierung adaptiert von [WKL12]

Als Ausgangsbasis wird das bestehende System aus Abschnitt 2.5 verwendet. Um eine Verwendung der Konsolidierungsmuster in den FCTE-Handlern zu ermöglichen, sind folgende Bedingungen [WKL13] zu beachten, bevor eine Analyse begonnen und eine Konsolidierung durchgeführt werden kann.

- Die Choreographie basiert auf One-to-One Interaktionen.
- Die gegebenen Prozesse sind korrekt [DBKL08] und Deadlock [LKLR07] frei.
- Alle wiederholenden BPEL Konstrukte, wie zum Beispiel ForEach, enthalten keine Kommunikationsaktivitäten die über `message links` miteinander verbunden sind.

3.1. Fault Handler - Scope

In diesem Abschnitt wird auf die Fehlerbehandlung mittels Fault Handler in Scopes eingegangen. Die Analyse von Fehlerbehandlungen von Invokes sind in Abschnitt 3.2 zu finden. Im Nachfolgenden werden unterschiedliche Kommunikationsszenarien von Fault Handler bei einer Konsolidierung betrachtet.

3.1.1. Fault Handler ohne Kommunikationslink

In diesem Szenario enthält der Fault Handler keinen Kommunikationslink zu einem Partnerprozess und spiegelt damit den trivialsten Fall wider [WKL13]. In Abbildung 3.2 zeigt die Ausgangssituation, dass Prozess P_A einen Scope S_A mit dem Fault Handler FH_A enthält. Der FH_A enthält keinen Kommunikationslink. Um eine Choreographie zu ermöglichen, existiert in S_A eine Aktivität $A1$ die über den Kommunikationslink m mit einer Aktivität $B1$ aus Prozess P_B kommuniziert.

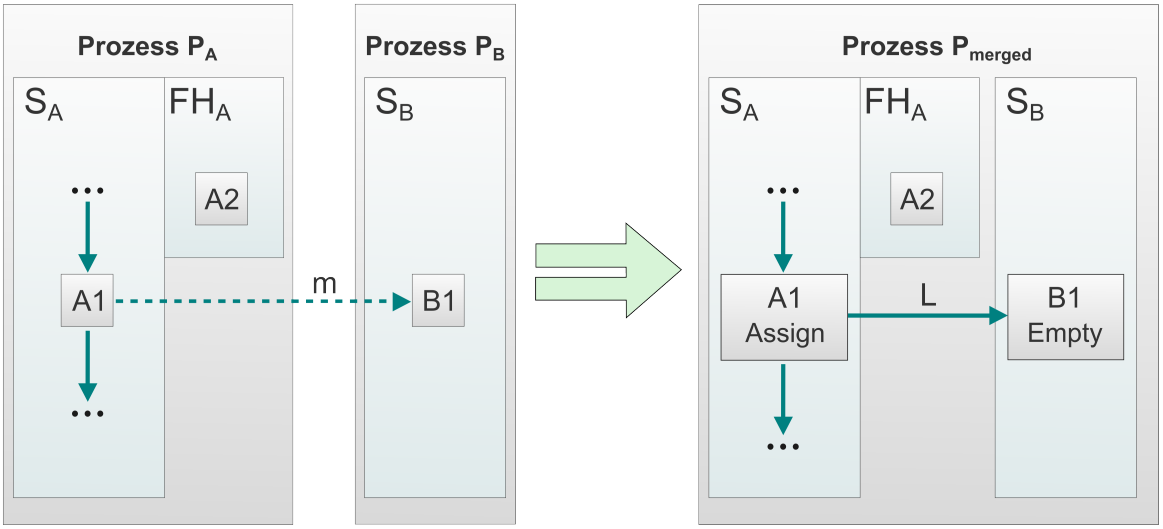


Abbildung 3.2.: Fault Handler ohne Kommunikationslink

Bei der Konsolidierung wird die Logik beider Prozesse P_A und P_B in einen jeweiligen Scope transferiert und m durch Synchronisationsaktivitäten ersetzt. Zur besseren Übersichtlichkeit wird in den Abbildungen auf die Darstellung der Prozess Scopes verzichtet. Durch die fehlende Kommunikationsaktivität in FH_A kann die Standard Konsolidierung vom bestehenden System aus Abschnitt 2.5 verwendet werden, da keine Grenzüberschreitungen von Links im Fault Handler stattfinden.

3.1.2. Fault Handler mit ausgehendem Kommunikationslink

Vergleicht man dieses Szenario (Abbildung 3.3) mit dem Szenario aus Abschnitt 3.1.1, so erkennt man, dass diesmal der Kommunikationslink nicht von Scope S_A zu Scope S_B zeigt, sondern vom Fault Handler FH_A über eine Invoke Aktivität eine Nachricht an eine Receive Aktivität in Scope S_B übermittelt wird. Bei der Übermittlung der Nachrichten werden die Daten aus der Variablen V_{in}, welche in Scope S_A definiert ist, entnommen und über die Nachricht an Scope S_B übermittelt, die die Daten in einer eigenen Variablen V_{rec} speichert.

Bei der Konsolidierung wird die Inovke Aktivität durch eine Assign Aktivität AN und die Receive Aktivität durch eine Empty Aktivität E ersetzt. Außerdem findet eine Synchronisation zwischen dem Assign AN und dem Empty E statt. Hierzu wird bei der Quellaktivität Assign AN der Link L im Source und bei der Zielaktivität Empty E der Link L im Target (siehe Abschnitt 2.2.5) gesetzt. Somit wird der Kommunikationslink m durch Synchronisationsaktivitäten ersetzt. Durch den Wegfall des Kommunikationslinks müssen nun die Daten, welche durch m übertragen wurden, durch die Zuweisung im Assign AN an den neuen Scope S_B übermittelt werden. Hierzu benötigt das Assign AN Zugriff auf die Variable V_{rec}. Dies erreicht man, indem die Variable V_{rec} aus dem Scope S_B entfernt und im Prozess P_{merged}

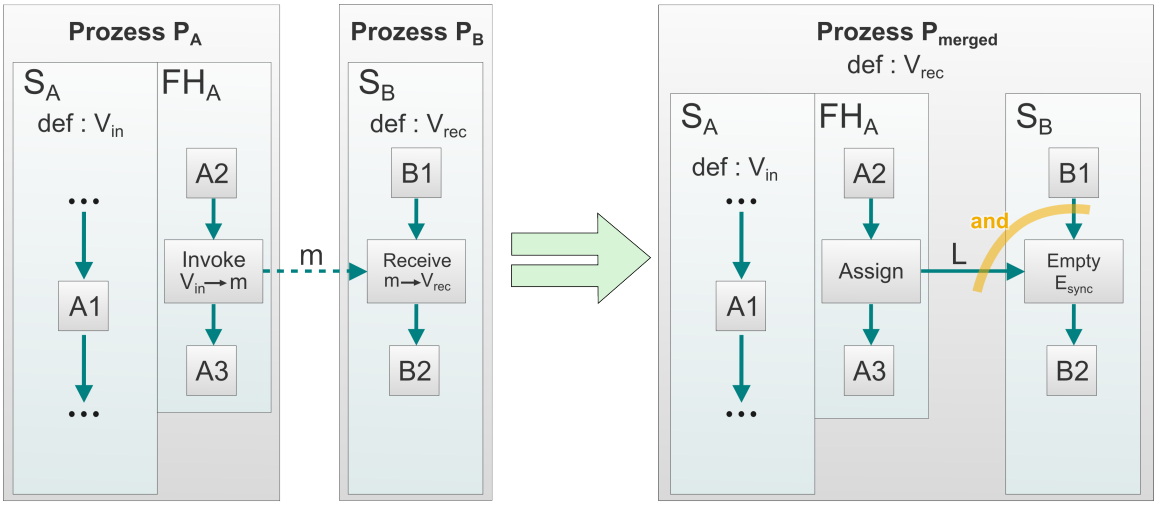


Abbildung 3.3.: Fault Handler mit ausgehendem Kommunikationslink

	A2	A3	B1	B2
A2	∅	<	R	<
A3	>	∅	R	R
B1	R	R	∅	<
B2	>	R	>	∅

	A2	A3	B1	B2
A2	∅	<	R	<
A3	>	∅	R	R
B1	R	R	∅	<
B2	>	R	>	∅

Abbildung 3.4.: Fault Handler mit ausgehendem Kommunikationslink: Allen Kalkühl Analyse

definiert wird. Da die Logik jedes Prozesses beim Konsolidieren in einen Scope transferiert wird, können diese Scopes auf die Variablen von Prozess P_{merged} zugreifen und somit die Zuweisung durchführen.

Die Abbildung 3.4 zeigt die Intervallmatrizen vor und nach der Konsolidierung. Man erkennt, dass die Änderung von Kommunikationsaktivitäten auf Synchronisationsaktivitäten hier keine Auswirkungen auf die einzelnen Aktivitäten hat. Die Aktivität $A2$ wird weiterhin vor der Aktivität $B2$ und $A3$ ausgeführt, während $A2$ zu $B1$ in jeder Relation stehen kann. Durch das Einfügen der Empty Aktivität ändert sich auch nichts an der Beziehung $B1 < B2$. $A3$ kann ebenfalls weiterhin Relation R zu $B1$ annehmen, auch nach dem Einführen der Assign Aktivität. Der Kontrolllink zwischen dem Assign und der Empty Aktivität gewährleistet außerdem die Relation $A2 < B2$. Die Betrachtung der Szenarien werden auf den Fault Handler

und den Prozess beschränkt, die miteinander kommunizieren, weswegen die Aktivität $A1$ nicht in der Intervallanalyse berücksichtigt wird.
Da in diesem Szenario eine Grenzüberschreitung der Links vom Inneren des Fault Handlers nach außen erfolgt und sich nach einer Konsolidierung keine Änderungen im Kontrollfluss ergeben, wird für diesen Fall ebenfalls die Standardimplementierung vom bestehenden System verwendet.

3.1.3. Fault Handler mit synchronem ausgehenden Kommunikationslink

In Abbildung 3.5 wird eine synchrone Kommunikation zwischen einem Fault Handler FH_A und einem Prozess P_B dargestellt. Für die Kommunikationen werden in Scope S_A die Variablen V_{in} und V_{out} sowie in Scope S_B die Variablen V_{rec} und V_{rep} definiert. Mit der Invoke Aktivität I_A werden die Daten der Variablen V_{in} über Nachricht m an die Receive Aktivität übermittelt und in die Variable V_{rec} gespeichert. Der Kontrollfluss des Fault Handlers FH_A wird bis zur Antwortnachricht m' der Reply Aktivität durch die Invoke Aktivität I_A blockiert. Der Prozess P_B arbeitet seinen Kontrollfluss bis zur Reply Aktivität ab und übermittelt die Daten der Variablen V_{rep} über die Nachricht m' zurück an die Invoke Aktivität I_A, welche die Daten aus der Nachricht m' in die Variable V_{out} übergibt. Damit ist der synchrone Kommunikationsaufruf abgeschlossen und der Fault Handler FH_A sowie der Prozess P_B können noch ihre verbleibenden Aktivitäten ausführen.

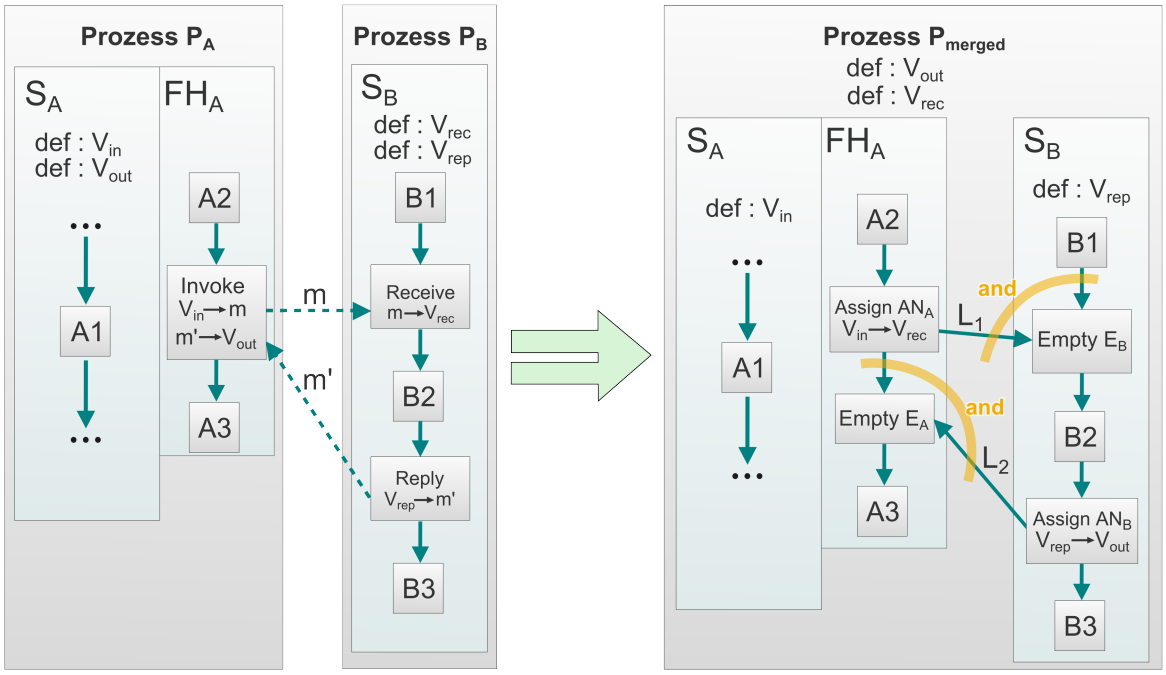


Abbildung 3.5.: Fault Handler mit synchronem ausgehenden Kommunikationslink

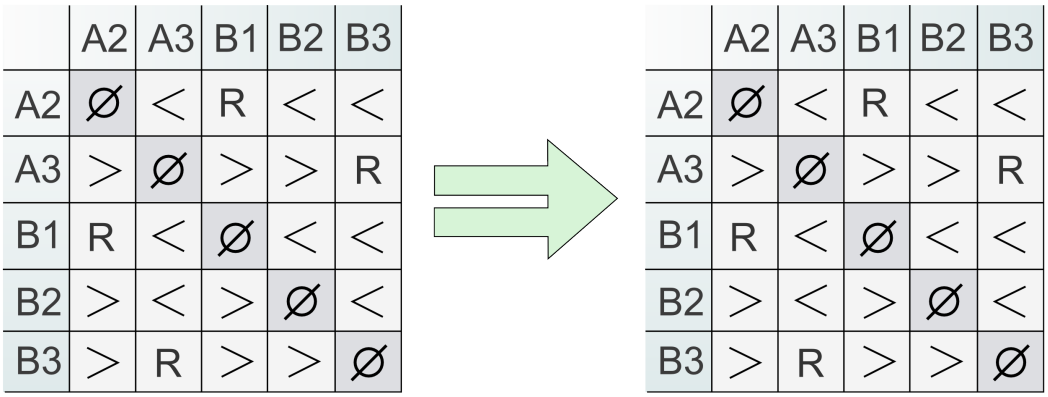


Abbildung 3.6.: Fault Handler mit synchronem ausgehenden Kommunikationslink: Allen Kalkühl Analyse

Diese Konsolidierung basiert auf dem synchronen Szenario in [WKL13]. Die Variablen V_{out} aus dem Scope S_A und die Variable V_{rec} aus dem Scope S_B werden in den Prozess P_{merged} transferiert. Die Invoke Aktivität wird durch eine Assign Aktivität AN_A und einer Empty Aktivität E_A ersetzt, die zueinander in Relation von AN_A **m** E_A stehen. Im Scope S_B, der die Logik von Prozess P_B enthält, wird die Receive Aktivität durch eine Empty Aktivität AN_B und die Reply Aktivität durch eine Assign Aktivität AN_B ersetzt. In AN_A findet die Zuweisung von V_{in} nach V_{rec} und in Assign AN_B die Zuweisung V_{rep} nach V_{out} statt, wodurch die Datenübertragung der Variablen gewährleistet wird. Um den Kontrollfluss beizubehalten, werden von AN_A nach E_B ein Link L_1 und von AN_B nach E_A ein Link L_2 gelegt.

Die Intervallmatrizen in Abbildung 3.6 geben die Ausgangssituation sowie das Ergebnis nach der Konsolidierung wieder. Die Intervallmatrizen zeigen, dass aufgrund der gesetzten Links sich der Kontrollfluss nicht verändert hat. Dabei ist zu beachten, dass der Link L_2 gegen die Kontrollflussbeschränkungen für Fault Handlers (Abschnitt 2.2.5) verstößt, da der Link L_2 in den Fault Handler FH_A hinein zeigt. Nach der Durchführung des Konsolidierungsmusters vom bestehenden System muss somit in diesem Bereich eine Korrektur vorgenommen werden, die in Abschnitt 3.1.8 zu finden ist.

3.1.4. Fault Handler mit synchronem eingehenden Kommunikationslink

In Abschnitt 3.1.3 wurde auf eine synchrone Kommunikation, ausgehend von dem Fault Handler, eingegangen, in diesem Abschnitt wird der Fault Handler von außerhalb synchron aufgerufen, wie in Abbildung 3.7 ersichtlich. Für die Kommunikation sind in Prozess P_A die Variablen V_{rec} und V_{rep} in Scope S_A sowie in Prozess P_B die Variablen V_{in} und V_{out} in Scope S_B definiert. Die Variablen V_{in} wird mittels Invoke im Prozess P_B über die Nachricht m an die Receive Aktivität des Fault Handlers übermittelt und in der Variablen V_{rec} abgelegt. Über die Reply Aktivität wird eine Antwort mit dem Variablenwert aus V_{rep} an die Invoke Aktivität zurückgesendet.

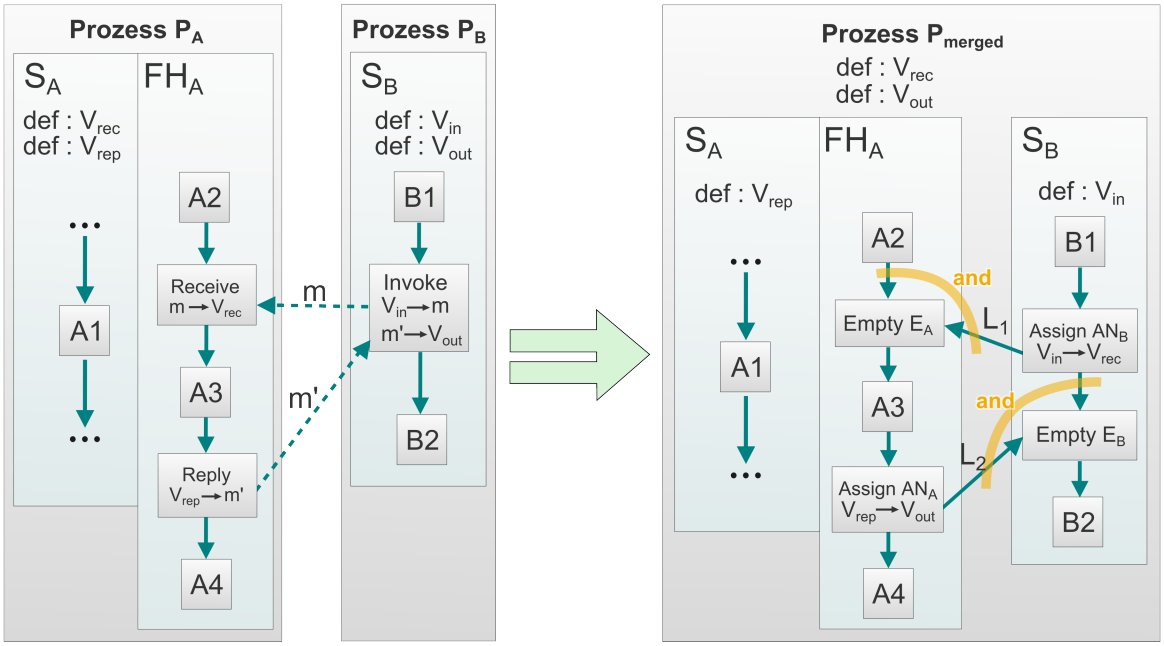
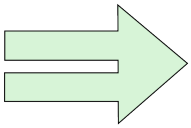


Abbildung 3.7.: Fault Handler mit synchronem eingehenden Kommunikationslink

	A2	A3	A4	B1	B2
A2	∅	<	<	R	<
A3	>	∅	<	>	<
A4	>	>	∅	>	R
B1	R	<	<	∅	<
B2	>	>	R	>	∅



	A2	A3	A4	B1	B2
A2	∅	<	<	R	<
A3	>	∅	<	>	<
A4	>	>	∅	>	R
B1	R	<	<	∅	<
B2	>	>	R	>	∅

Abbildung 3.8.: Fault Handler mit synchronem eingehenden Kommunikationslink: Allen Kalkühl Analyse

Das Ergebnis der Konsolidierung ähnelt dem aus Abschnitt 3.1.3. Die Invoke Aktivität wird durch ein Assign AN_B und ein Empty E_B ersetzt, außerdem wird die Receive Aktivität durch ein Empty E_A und die Reply Aktivität durch ein Assign AN_A ersetzt. Der Verstoß gegen die Grenzüberschreitung von Links findet diesmal von L_1 statt, weil dies der eingehende Link in den Fault Handler ist. (Die Korrektur ist in Abschnitt 3.1.8 zu finden.)

Betrachtet man die Intervallmatrix in Abbildung 3.8, so ist zu erkennen, dass diese vor und nach der Konsolidierung kongruent ist. Da in diesem Szenario, im Vergleich zum Szenario aus 3.1.3, die Aktivitäten gespiegelt worden sind, ist ebenfalls eine Spiegelung in den Intervallmatrizen ersichtlich.

3.1.5. Fault Handler mit asynchroner Kommunikation

In diesem Szenario wird auf die asynchrone Kommunikation eingegangen, die in Abbildung 3.9 dargestellt ist. Zuerst findet ein asynchroner Aufruf vom Fault Handler FH_A aus Prozess P_A nach Prozess P_B statt und danach erfolgt eine asynchrone Antwort von Prozess P_B in den Fault Handler FH_A. Beim Aufruf von Invoke I_A wird eine CorrelationID (siehe Abschnitt 2.2.4) sowie der Variablenwert V_{in} über die Nachricht m an die Receive Aktivität R_B übermittelt. Nach dem Empfang der Nachricht m kann nun der Prozess P_B über Invoke I_B einen asynchronen Aufruf an R_A senden. Hierbei wird dieselbe CorrelationID verwendet, um die zugehörige Prozessinstanz zu finden.

Bei der Konsolidierung werden die Invoke Aktivitäten durch Assign Aktivitäten und die Receive Aktivitäten durch Empty Aktivitäten ersetzt. Außerdem wird von AN_A nach E_B sowie von AN_B nach E_A jeweils ein Synchronisationslink gesetzt. Die Variablen V_{out} und V_{rec} werden für die Zuweisung in den Assign Aktivitäten benötigt und müssen somit in den

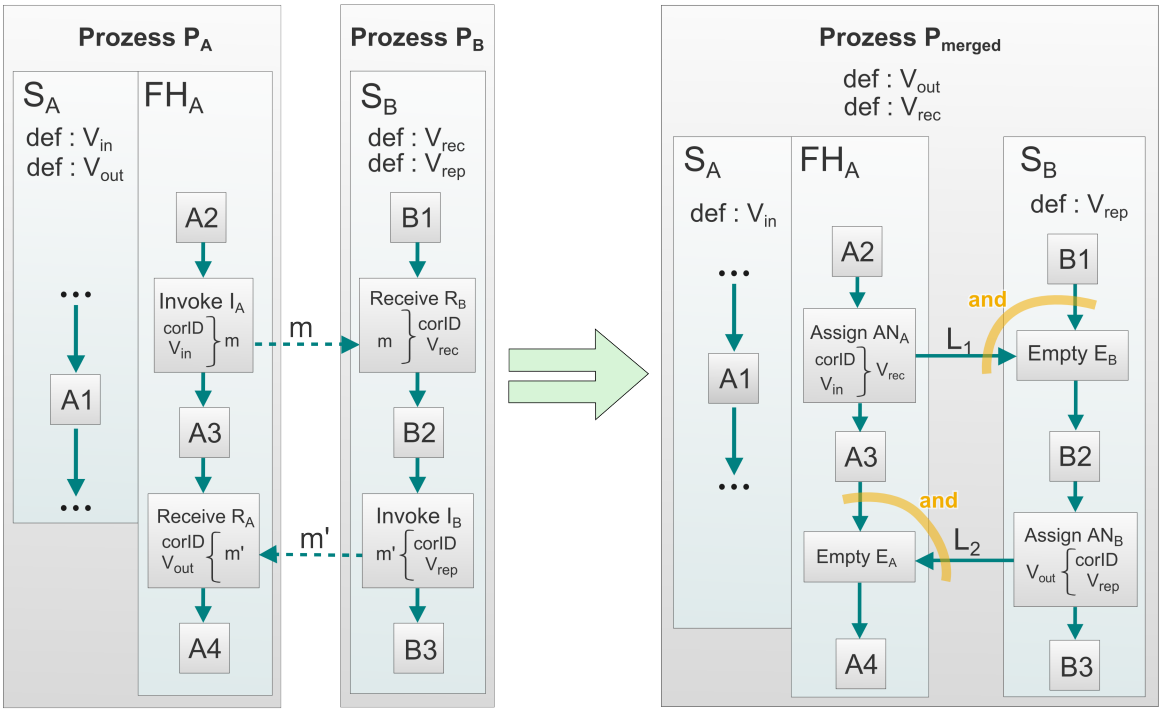


Abbildung 3.9.: Fault Handler mit asynchroner Kommunikation

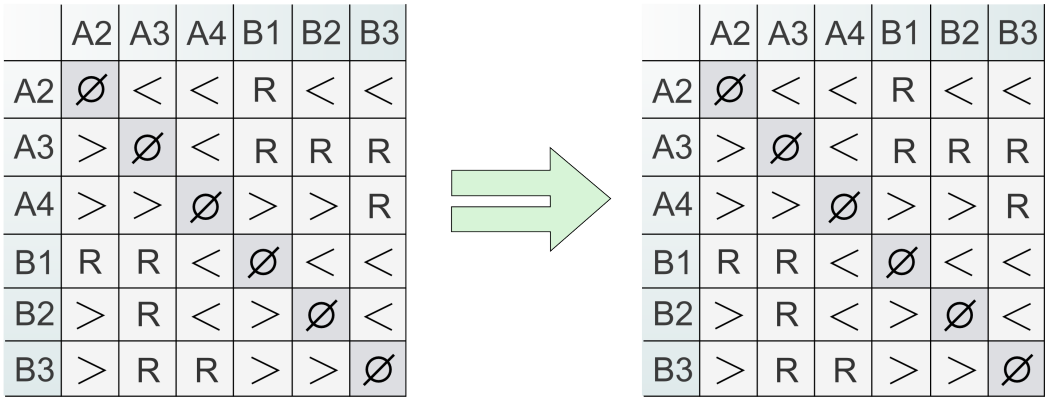


Abbildung 3.10.: Fault Handler mit asynchroner Kommunikation: Allen Kalkühl Analyse

Prozess P_{merged} ausgelagert werden. Die CorrelationID's werden nach der Konsolidierung für den Kontrollfluss nicht mehr benötigt und könnten entfernt werden. Es besteht jedoch die Möglichkeit, dass die CorrelationID weiterhin als Wert in der Geschäftslogik verwendet wird. Die Entfernung der CorrelationID würde somit nach der Konsolidierung keinen korrekten Prozess zurückliefern. Dies widerspräche den definierten Bedingungen, siehe Kapitel 3, weswegen erst nach einer ausreichenden Analyse die CorrelationID entfernt werden darf.

Wie man in Abbildung 3.10 erkennt, stimmen beide Intervallmatrizen überein, was als Anhaltspunkt dient, dass die Konsolidierung korrekt ablaufen würde. Das Problem in diesem Szenario besteht jedoch wieder in der Grenzüberschreitung des Links L_2 von Scope S_B in den FaultHandler FH_A. (Die Korrektur ist in Abschnitt 3.1.8 zu finden.) Die Spiegelung der Aktivitäten von FH_A und des Prozesses P_B sowie die Umgekehrte Aufrufreihenfolge der Kommunikationsaktivitäten, anstatt $I_A < R_A$ erfolgt der Aufruf $R_A < I_A$ in FH_A, sind durch die asynchrone Kommunikation bereits inbegriffen und werden nicht mehr explizit analysiert.

3.1.6. Verschachtelte Fault Handler

Eine weitere Möglichkeit Fault Handler zu definieren, ist über die Verschachtelung. In Abbildung 3.11 findet eine Verschachtelung im Fault Handler FH_A statt. Es wird in FH_A ein root-Scope S_{FH} mit einem weiteren Fault Handler FH_{root} definiert. Mit dieser Konstellation können Fehler innerhalb des Fault Handlers FH_A abgefangen und verarbeitet werden, solange die Fehler in S_{FH} auftreten. Nach einem synchronen Aufruf von S_{FH} nach S_B wird in S_{FH} ein Fehler geworfen und vom installierten Fault Handler FH_{root} verarbeitet, welcher wiederum einen synchronen Aufruf mit Prozess P_B über Scope S_C ausführt.

Die Konsolidierung folgt in diesem Beispiel dem gleichen Muster wie der im Abschnitt 3.1.3. Die Aktivitäten Invoke werden durch Assign und Empty, die Receive Aktivitäten jeweils durch ein Empty E_B und ein Empty E_C sowie die Reply Aktivitäten durch ein Assign AN_B

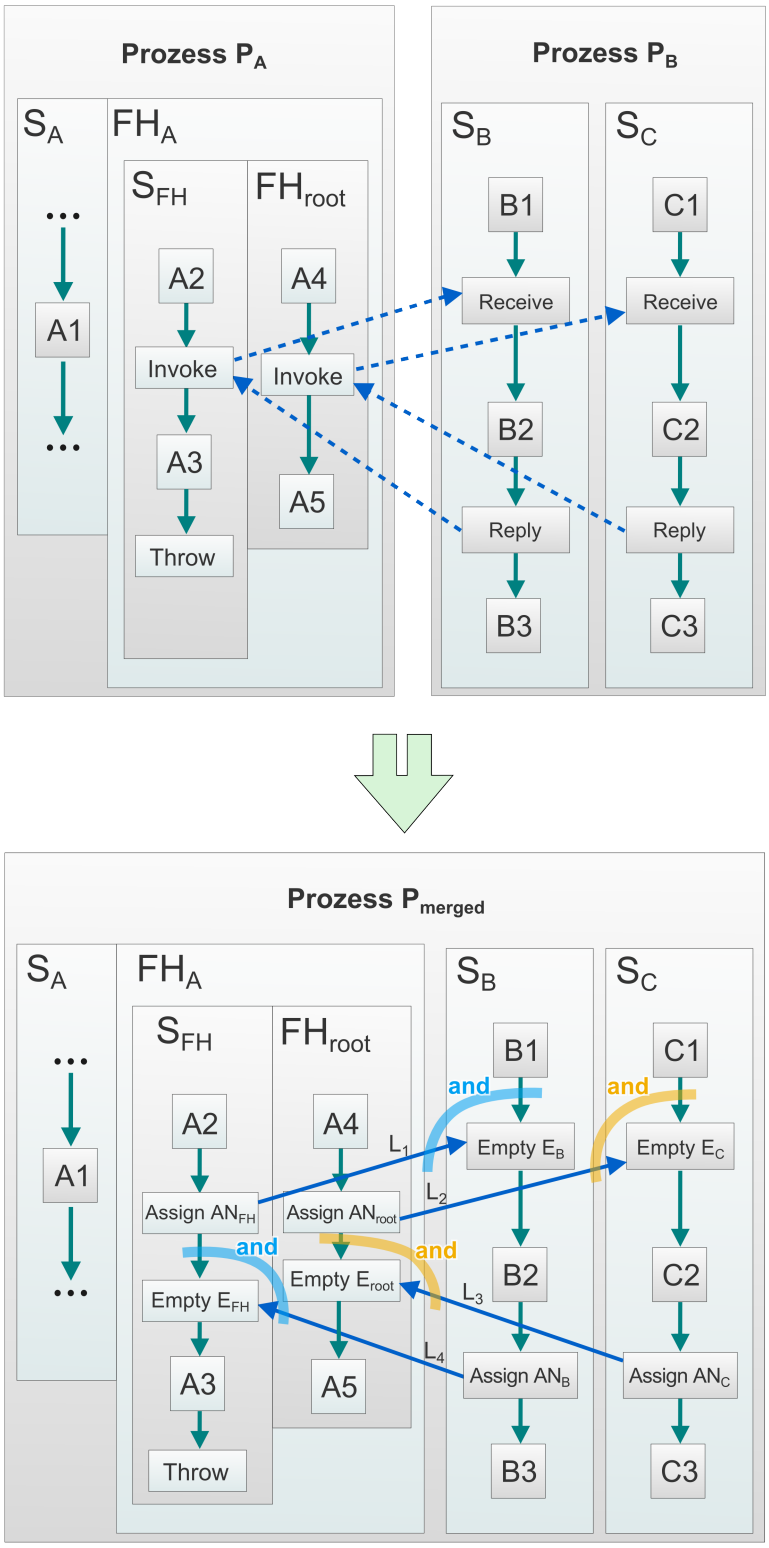
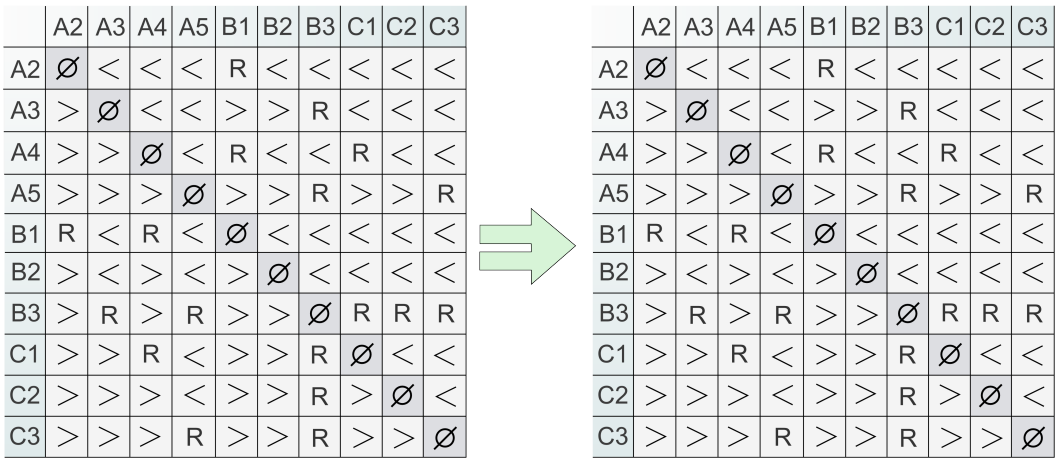


Abbildung 3.11.: Verschachtelung von Fault Handlern



	A2	A3	A4	A5	B1	B2	B3	C1	C2	C3
A2	∅	<	<	<	R	<	<	<	<	<
A3	>	∅	<	<	>	>	R	<	<	<
A4	>	>	∅	<	R	<	<	R	<	<
A5	>	>	>	∅	>	>	R	>	>	R
B1	R	<	R	<	∅	<	<	<	<	<
B2	>	<	>	<	>	∅	<	<	<	<
B3	>	R	>	R	>	>	∅	R	R	R
C1	>	>	R	<	>	>	R	∅	<	<
C2	>	>	>	<	>	>	R	>	∅	<
C3	>	>	>	R	>	>	R	>	>	∅

	A2	A3	A4	A5	B1	B2	B3	C1	C2	C3
A2	∅	<	<	<	R	<	<	<	<	<
A3	>	∅	<	<	>	>	R	<	<	<
A4	>	>	∅	<	R	<	<	R	<	<
A5	>	>	>	∅	>	>	R	>	>	R
B1	R	<	R	<	∅	<	<	<	<	<
B2	>	<	>	<	>	∅	<	<	<	<
B3	>	R	>	R	>	>	∅	R	R	R
C1	>	>	R	<	>	>	R	∅	<	<
C2	>	>	>	<	>	>	R	>	∅	<
C3	>	>	>	R	>	>	R	>	>	∅

Abbildung 3.12.: Verschachtelte Fault Handler: Allen Kalkühl Analyse

und ein Assign AN_C ersetzt. Ebenfalls werden von der Aktivität Assign AN_{FH} nach Empty E_B, von Assign AN_{root} nach E_C, von Assign AN_C nach Empty E_{root} und von Assign AN_B nach Empty E_{FH} Links definiert. Hierdurch wird die Synchronisation sowie der Erhalt des Kontrollflusses nach der Konsolidierung erreicht. Außerdem liegen die Quellaktivität von den Links L_1 und L_2 in einem Fault Handler während die Zielaktivitäten außerhalb der Fault Handler definiert sind. Im Gegensatz dazu sind die Quellaktivitäten von Link L_3 und Link L_4 außerhalb und die Zielaktivitäten innerhalb eines Fault Handlers definiert. Dies verstößt gegen die Kontrollflussbeschränkungen (siehe Abschnitt 2.2.5).

Die Intervallmatrix 3.12 zeigt, dass der Kontrollfluss nach der Konsolidierung weiterhin bestehen würde, da sich an der Aufrufstruktur nichts verändert hat. Eine Überprüfung der anderen Kommunikationsmuster ist nicht notwendig, da der Kontrollfluss bei der Verschachtelung die Endaktivität `<throw>` aus dem Scope S_{FH} mit der Startaktivität A_{start}, in diesem Beispiel ist $A_{start} = A4$, aus dem Fault Handler FH_{root}, mittels der Intervallrelation `<throw>` **m** A_{start}, verbindet. Wird z. B. der zweite synchrone Aufruf durch einen asynchronen Aufruf ersetzt, bleibt die Beziehung `<throw>` **m** A_{start} weiterhin bestehen.

3.1.7. Geschäfts- und Laufzeitfehler

In einem Fault Handler können zwei unterschiedliche Fehler auftreten [VBP08].

- **Geschäftsfehler:** Hierunter fallen alle selbst definierten Fehler. Diese Fehler werden in der WSDL deklariert und können durch die Throw Aktivität oder durch eine Fehlerantwort auf eine Invoke Anfrage auftreten.
- **Laufzeitfehler (Runtime Exception):** Runtime Exceptions treten zur Laufzeit eines Prozesses auf und sind in keiner WSDL vom Benutzer definiert worden. Dies können z. B. Fehler beim Kopieren einer Variablen oder unbekannte Fehler sein.

Listing 3.1 Definition eines Geschäftsfehler

```
<faultHandlers>
       <catch faultName="ns:MyServiceException" faultVariable="faultVar"
                                    faultMessageType="ns:MyServiceException">
              activity
       </catch>
</faultHandlers>
```

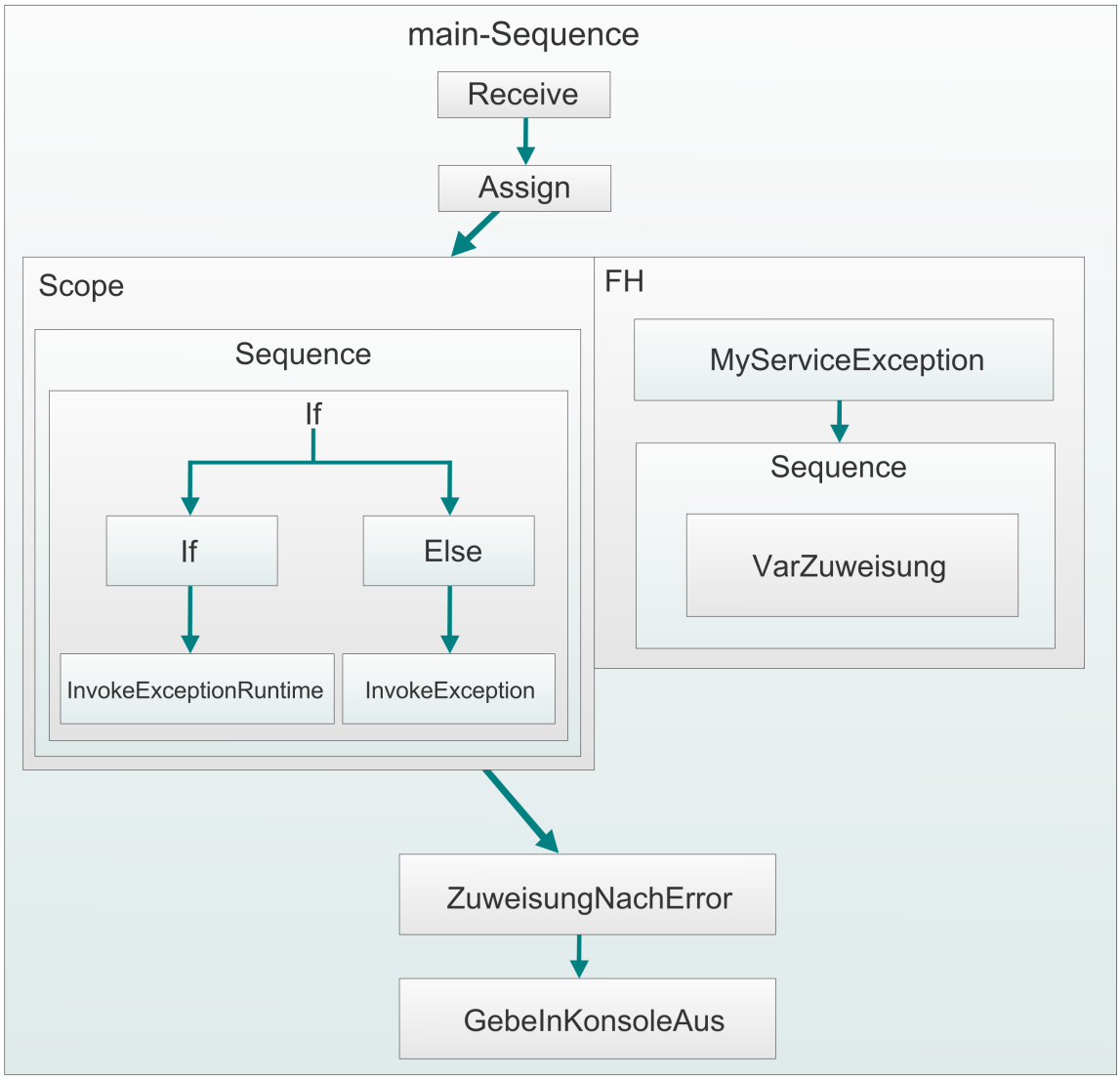


Abbildung 3.13.: Beispielprozess für Geschäfts- und Laufzeitfehler

Um nun die Auswirkung beider Fehler zu analysieren, wird das Beispiel in Abbildung 3.13 betrachtet. Der Prozess legt einen Scope mit einem speziellen Fehler fest, der in der zugehörigen WSDL definiert ist. Bei der Instanzerzeugung kann über einen Übergabewert $W = \{1, 0\}$ die If Aktivität gesteuert werden, die jeweils einen Web Service aufruft. Mit der Invoke Aktivität `InvokeExceptionRuntime` wird ein Service aufgerufen, der einen Laufzeitfehler wirft (Runtime Exception). Die Invoke Aktivität `InvokeException` hingegen wirft einen Fehler `MyServiceException`, der zuvor in der zugehörigen WSDL definiert wurde. Durch die Verarbeitung des Geschäftsfehlers mit dem `<catch>` Block, Listing 3.1, kann der Kontrollfluss fortgesetzt werden. Dabei wird der Fehler durch die Attribute `faultName`, `faultVariable` und `faultMessageType` erkannt und dem jeweiligen `<catch>` Block zugewiesen. Das `faultElement` Attribut ist der Datentyp des Fehlers und ist in der WSDL festgelegt. Die `faultVariable` gilt in diesem `<catch>` Block als Variablendefinition und wird von der BPEL-Engine automatisch, mit den Fehlerdaten, initialisiert. Im Fall des Laufzeitfehlers erkennt die BPEL-Engine den Fehler nicht, siehe Listing A.1 bzw. Listing A.2 zum Vergleich, und führt somit den Kontrollfluss nicht weiter.

Dieses Beispiel zeigt, dass eine Konsolidierung keinen Einfluss auf Laufzeitfehler hat. Bei Geschäftsfehlern hingegen können die eingesetzten Variablen bei der Fehlerverarbeitung verwendet werden und müssen deshalb bei der Korrektur der Grenzüberschreitung der Links beachtet werden, hierzu mehr in Abschnitt 3.1.8.

3.1.8. Konsolidierungsergebnis für Fault Handler

Die Tabelle 3.1 zeigt das Ergebnis der Analyse der Szenarien. Es ist ersichtlich, dass alle Szenarien mit verbotenen Grenzüberschreitungen in den Fault Handler die Konsolidierungsbedingungen aus Abschnitt 3 nicht erfüllt haben. Die Basis einer Lösung dieses Problems

Szenario	Kontrollfluss	Grenzüberschreitung von Links
FH ohne Kommunikationslink (KL)	●	●
FH mit ausgehendem KL	●	●
FH mit synchronem ausgehenden KL	●	○
FH mit synchronem eingehenden KL	●	○
FH mit asynchroner Kommunikation	●	○
Verschachtelte Fault Handler	●	○

Legende:

- ● – Kontrollfluss bleibt auch nach der Konsolidierung **erhalten** oder die grenzüberschreitenden Links **verstoßen nicht** gegen die Bedingungen
- ○ – Kontrollfluss bleibt nach der Konsolidierung **nicht erhalten** oder die grenzüberschreitenden Links **verstoßen** gegen die Bedingungen

Tabelle 3.1.: Ergebnis der Fault Handler (FH) Szenarien

bietet [WKL13], in welchem die Logik des Fault Handlers in eine neue Scope Aktivität ausgelagert wird. In Abbildung 3.14 ist die modifizierte Lösung der Konsolidierung eines Fault Handler abgebildet. Dabei ist zu beachten, dass die Lösung nur auf jeden einzelnen Fault Handler mit eingehendem Link angewendet wird.

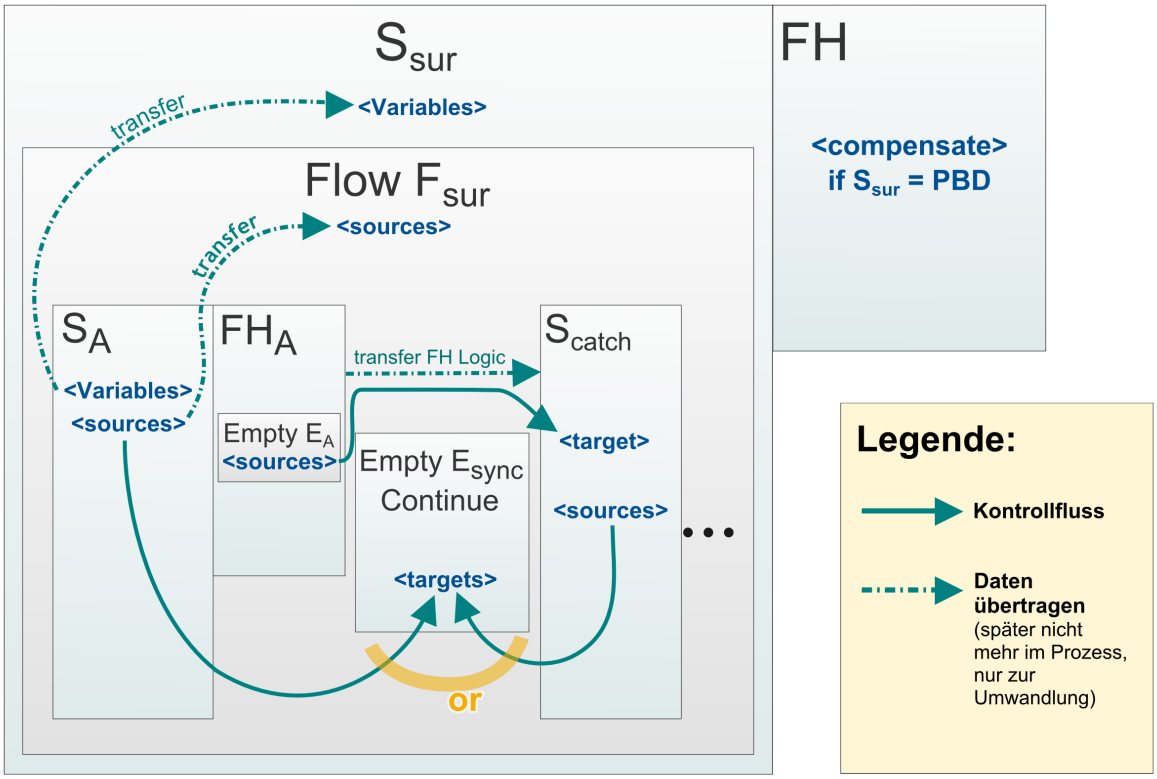


Abbildung 3.14.: Algorithmus zur Behebung der Grenzüberschreitung

Der Fault Handler von Scope S_A soll konsolidiert werden, hierzu wird im ersten Schritt ein Scope S_{sur} erstellt und die Variablen von Scope S_A nach Scope S_{sur} übertragen. Dies dient dem Zweck der Erreichbarkeit. Denn sollte innerhalb eines Fault Handlers FH_A auf eine Variable von Scope S_A zugegriffen werden und die Logik würde im Laufe der Konsolidierung in einen Peer-Scope ausgelagert, dann wäre der Zugriff auf diese Variable nicht mehr möglich. Als Nächstes wird ein Flow F_{sur} dem Scope S_{sur} hinzugefügt, welches der Synchronisation der Aktivitäten dient. Dem Flow F_{sur} werden unter anderem das Scope S_A, eine Synchronisationsaktivität Empty E_{sync} und für jeden `catch` und `catchAll` Fall ein neuer Scope S_{catch} hinzugefügt. Danach werden die Sources aus Scope S_A in Flow F_{sur} übertragen, damit der ehemalige Kontrollfluss nicht unterbrochen wird, und mit einem Link $L_{AtoEsync}$ von Scope S_A nach Empty E_{sync} überschrieben. Darauffolgend wird die Logik aus den `<catch>` und `<catchAll>` in die zugehörigen Scopes S_{catch} transferiert und jeweils eine Empty Aktivität E_A im `<catch>` oder `<catchAll>` erzeugt, die per Link mit dem zugehörigen Scope S_{catch} verbunden ist. Im Scope S_{catch} wird der Compensation Handler durch eine Empty Aktivität deaktiviert,

da der root Scope eines Fault Handlers keinen Compensation Handler haben darf (siehe Abschnitt 2.2.7) und ein Fault Handler nach dem Beenden auch nicht mehr kompensiert werden kann. Außerdem wird von Scope S_{catch} nach Empty E_{sync} ein Link gezogen. Der neue Scope S_{catch} und der alte Scope S_A synchronisieren nun in der Empty Aktivität E_{sync}. Sollte der Scope S_A den Status Completed erreichen, so kann der Kontrollfluss trotzdem noch durch Dead-Path Elimination (siehe Abschnitt 2.2.5) fortgesetzt werden, indem der Link vom S_{catch} nach E_{sync} als `false` propagiert wird. Bevor zum Abschluss das Scope S_{sur} an die Stelle von S_A gesetzt werden kann, muss erst noch geprüft werden, ob S_A ein Prozess Scope war. Trifft dies zu, so wird in S_{sur} der Standard Fault Handler überschrieben und nur ein `<compensate>` im `<catchAll>` gesetzt. Dies dient dem Schutz der anderen PBD Scopes. Sollte ein PBD einen Fehler erzeugen, den er nach oben weiterreicht, so wird dies im Scope S_{sur}, durch die Entfernung des Standard Fault Handlers, gestoppt. Die Peer-Scope PBDs laufen dadurch ungestört weiter. Tritt ein Fehler nach der Konsolidierung in einem Fault Handler auf, für den kein definierter Fehlerfall existiert, so wird der Fehler an den umgebenden Scope S_{sur} propagiert. Mit den Standard Fault Handler von Scope S_{sur} wird der Fehler an den übergeordneten Scope weiter propagiert. Somit verändert sich der Propagierungsablauf auch nach der Konsolidierung nicht.

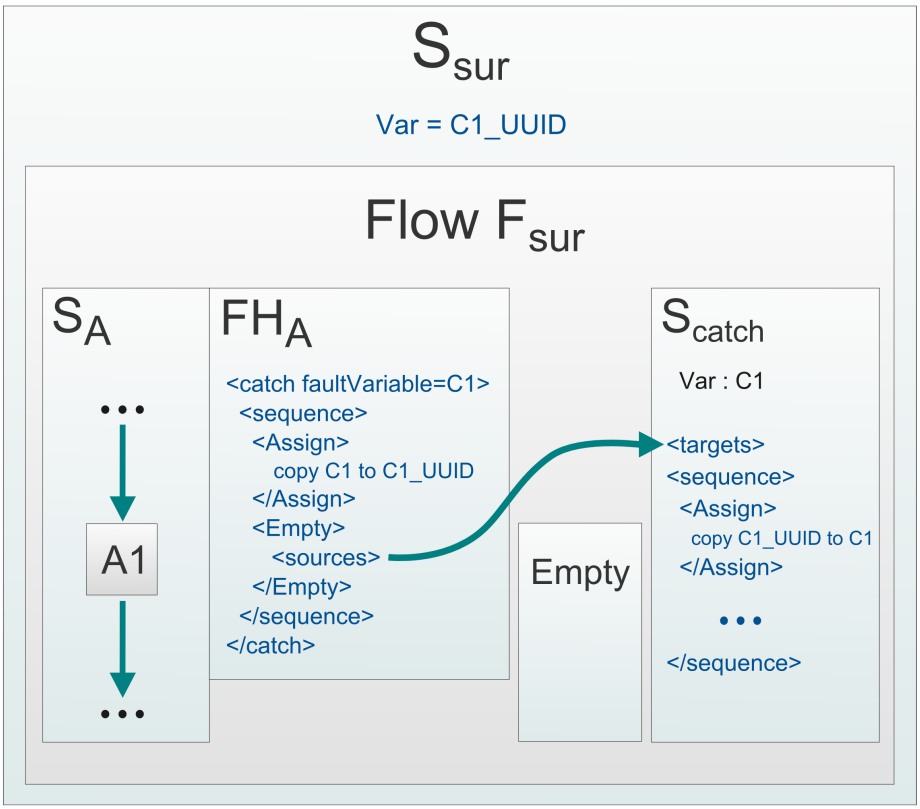


Abbildung 3.15.: Lösung für den Zugriff auf Fehlervariablen

Durch diese Lösung werden die vom bestehenden System erzeugten Probleme bei der Grenzüberschreitung von Links umgangen. Jedoch wird mit dieser Lösung auch ein weiteres Problem erzeugt, und zwar das der Geschäftsfehler (siehe Abschnitt 3.1.7). Die definierten Variablen von Geschäftsfehlern könnten bei der Verarbeitung verwendet werden. Hierzu wurde die Lösung, wie in Abbildung 3.15 zu sehen, erweitert. Enthält ein Fault Handler in Scope S_A eine `faultVariable=C1` so wird in Scope S_{sur} eine Variable `C1_UUID` vom gleichen Typ angelegt. Mit dem Suffix UUID (Universally Unique Identifier) kann jede Variable eindeutig identifiziert werden. Im `<catch>` oder `<catchAll>` wird daraufhin eine Sequence angelegt und die Variable `C1` der Variablen `C1_UUID` zugewiesen. Im zugehörigen Scope S_{catch} wird ebenfalls eine Sequence angelegt und eine Zuweisung von `C1_UUID` nach `C1` durchgeführt. Die Variable `C1` wurde dabei ebenfalls im S_{catch} neu erstellt. Hierdurch können die Variablen auch nach der Auslagerung der Logik mit demselben Namen weiterhin verwendet werden, da der Wert der Variablen `C1` aus dem Fault Handler in Scope S_A in die Variable `C1` in Scope S_{catch} übertragen wurde. Damit wäre das Problem der Geschäftsfehler ebenfalls gelöst und der Algorithmus vollständig.

Anwendungsbeispiel

Anhand dem in Abbildung 3.16 dargestellten Beispiel wird gezeigt, dass durch das Auslagern der Aktivitäten eines Fault Handlers in einen Peer-Scope die Kontrollflussbeschränkungen aus Abschnitt 2.2.5 nicht mehr verletzt werden. Als Ausgangsbasis wird das Beispiel aus Abschnitt 3.1.2 verwendet. In diesem Szenario existiert ein ausgehender und eingehender Kommunikationslink, wodurch die anderen Szenarien impliziert werden.

Bei der Konsolidierung werden die Prozesse P_A und P_B in den Flow F_{merged} des Prozesses P_{merged} als Scopes übertragen. Diese werden zu Gunsten der Übersichtlichkeit in Abbildung 3.16 nicht dargestellt. Die Kommunikationsaktivitäten Invoke, Receive und Reply werden, wie in Abschnitt 3.1.2 beschrieben, umgewandelt. Die Logik vom Fault Handler FH_A wird in einen Peer-Scope S_{catch} übertragen. Der Fault Handler FH_A erhält eine Empty Aktivität, die mit einem Link auf den Peer-Scope S_{catch} referenziert. Beide Scopes S_A und S_{catch} werden in ein Flow F_{sur}, welches in einem Scope S_{sur} liegt, übertragen. Der Scope S_{sur} liegt im Scope von Prozess P_A und bekommt die Variablen von Scope S_A zugewiesen, damit Scope S_{catch} weiterhin auf die Variablen zugreifen kann. Scope S_A und S_{catch} synchronisieren in einer Empty Aktivität $E_{continue}$. Wird beim fehlerfreien Durchlauf der Scope S_{catch} nicht aufgerufen, so wird durch Dead-Path Elimination der Kontrollfluss von Scope S_{catch} zur Empty Aktivität $E_{continue}$ auf `false` gesetzt und der Kontrollfluss kann fortgesetzt werden. Tritt ein Fehler auf, ruft der Fault Handler FH_A den Scope S_{catch} auf und führt seinen Kontrollfluss zur Aktivität Empty $E_{continue}$ weiter. Der Scope S_{catch} verarbeitet nun seine Aktivitäten und synchronisiert mit der Empty Aktivität $E_{continue}$. Dadurch wird der Kontrollfluss vom Ausgangsbeispiel erst fortgesetzt, wenn auch die Fehlerverarbeitung abgeschlossen ist. Ebenfalls findet keine Kontrollflussverletzung von der Aktivität Assign AN_B nach Empty E_A statt, da die Empty Aktivität E_A sich nicht mehr im Fault Handler FH_A, sondern im Scope S_{catch} befindet.

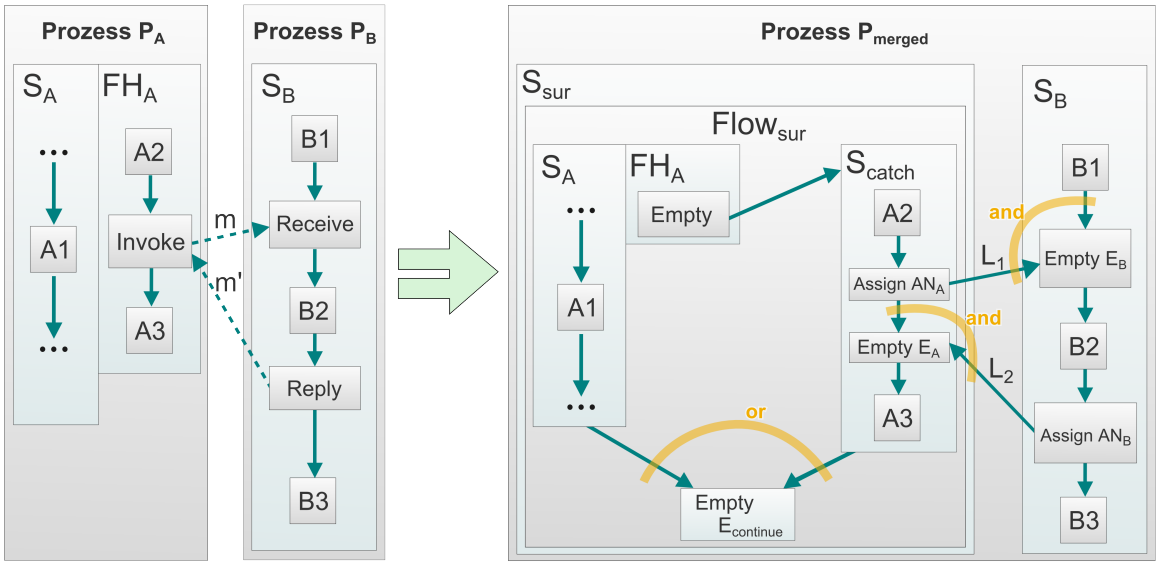


Abbildung 3.16.: Anwendungsbeispiel für den Konsolidierungsalgorithmus

	A2	A3	B1	B2	B3
A2	∅	<	R	<	<
A3	>	∅	>	>	R
B1	R	<	∅	<	<
B2	>	<	>	∅	<
B3	>	R	>	>	∅

	A2	A3	B1	B2	B3
A2	∅	<	R	<	<
A3	>	∅	>	>	R
B1	R	<	∅	<	<
B2	>	<	>	∅	<
B3	>	R	>	>	∅

Abbildung 3.17.: Anwendungsbeispiel: Allen Kalkühl Analyse

Ein Vergleich von Abbildung 3.17 und Abbildung 3.4 gibt ein Indiz dafür, dass sich der Kontrollfluss nicht verändert hat. Es finden zwei Umwandlungen statt. Die erste Umwandlung ersetzt die Kommunikationsmuster durch Synchronisationsaktivitäten. Dies verändert den Kontrollfluss nicht, wie in Abbildung 3.17 zu sehen ist. Die zweite Umwandlung verändert den strukturellen Aufbau der Scopes, so dass keine Kontrollflussverletzungen im Fault Handler stattfinden. Hierzu wird Scope S_A von S_{sur} ersetzt. Da der Scope S_{sur} die Logik von Scope S_A weiterhin erhält, bleibt der Kontrollfluss somit vor und nach Scope S_{sur} unverändert bestehen. Innerhalb des Scopes S_{sur} wird Scope S_A parallel zu Scope S_{catch} ausgeführt. Durch den Link zwischen Empty E_A und Scope S_{catch} wird Scope S_{catch} erst nach Empty E_A abgearbeitet. Alle Aktivitäten A_{all} in Scope S_{catch} stehen somit zum Fault Handler

FH_A in Relation FH_A **<** A_{all} und nicht mehr in der Relation A_{all} **d** FH_A. Darauffolgend findet die Synchronisation, mit der Disjunktion aller eingehenden Links (siehe Abschnitt 2.2.5), über die Aktivität Emtpy $E_{continue}$ statt, mit der der Kontrollfluss fortgesetzt wird. Im Ausgangsbeispiel wurden die Aktivitäten A_{all} im Fault Handler ausgeführt. Im konsolidierten Kontrollfluss wird der Fault Handler beendet, der Scope S_{catch} ausgeführt und dann beide in der Empty Aktivität $E_{continue}$ synchronisiert. Die Aktivitäten A_{all} werden somit wie im Ausgangsbeispiel vor der Fortsetzung des Kontrollfluss ausgeführt, womit der Kontrollfluss im Allgemeinen erhalten bleibt.

Dieses Beispiel ist ebenfalls auf die anderen Szenarien anwendbar. Durch Substituieren der Aktivitäten, im Fault Handler durch die Fault Handler der einzelnen Szenarien, entsteht dieselbe Situation. Sobald ein eingehender Kontrollfluss vorliegt, wird der Algorithmus angewendet. Wie bereits beschrieben, werden dabei die Aktivitäten aus dem Fault Handler FH_A der einzelnen Szenarien in Scope S_{catch} ausgelagert und erhalten somit die gleiche Struktur wie in Abbildung 3.16 dargestellt.

3.2. Fault Handler - Invoke

Ein Invoke kann laut Spezifikation ebenfalls `<catch>` oder `<catchAll>` Blöcke enthalten. Um diese Blöcke bei der Konsolidierung zu berücksichtigen, werden Invoke Aktivitäten mit solchen Blöcken in equivalente Scopes umgewandelt [AAA+07]. Hierzu wird ein Scope S_A erzeugt und ein Fault Handler hinzugefügt, der die `<catch>` und `<catchAll>` Blöcke des Invokes I_{Start}, welches umgewandelt wird, zugewiesen bekommt. Ebenfalls wird der Compensation Handler vom Invoke I_{Start} auf das Scope S_A übertragen. Das Invoke I_{Start} enthält daraufhin keinen Fault Handler und auch keinen Compensation Handler mehr. Im nächsten Schritt wird das Invoke I_{Start} in den Scope S_A übertragen. Die gesamte Umwandlung, welche Abbildung 3.18 entnommen werden kann, muss ebenfalls vor der Konsolidierung des bestehenden Systems erfolgen, damit die Algorithmen auf das neue Scope reagieren können. Was jedoch noch zu beachten ist, sind die `CompensateScope` Aktivitäten. Diese können auf Scope sowie auf Invoke Aktivitäten zeigen und müssen entsprechend an den neuen Scope S_A angepasst werden. Dies wird durch das Umschreiben des `target` Attributs vom Invoke I_{Start} auf das Scope S_A erreicht. Mit dieser Umwandlung kann nun die Konsolidierung ohne Probleme auch für Invoke Aktivitäten durchgeführt sowie das SyncPattern1.5 von Debicki [Deb13] entfernt werden.

Die Abbildung A.1, im Anhang, zeigt das SyncPattern1.5. Das Pattern geht davon aus, dass ein synchrones Invoke in einem FCTE-Handler in Prozess P_A einen zweiten Prozess P_B aufruft. Es wird vorausgesetzt, dass dies die einzigen Kommunikationslinks von Prozess P_B sind. Davon ausgehend kann das Pattern angewendet werden. Hierzu wird im FCTE-Hander ein Flow erzeugt und die Aktivitäten zuvor in den Flow transferiert. Der Process P_B wird in den zugehörigen Scope S_B umgewandelt und ebenfalls in den Flow verschoben. Dies ist jedoch nur möglich, da der P_B keine weiteren Kommunikationslinks besitzt. Hierdurch wurde die Grenzüberschreitung von Links von außen nach innen in einen Fault Handler nicht verletzt. Durch die Umwandlung und die Verarbeitung von FCTE-Handlern wird

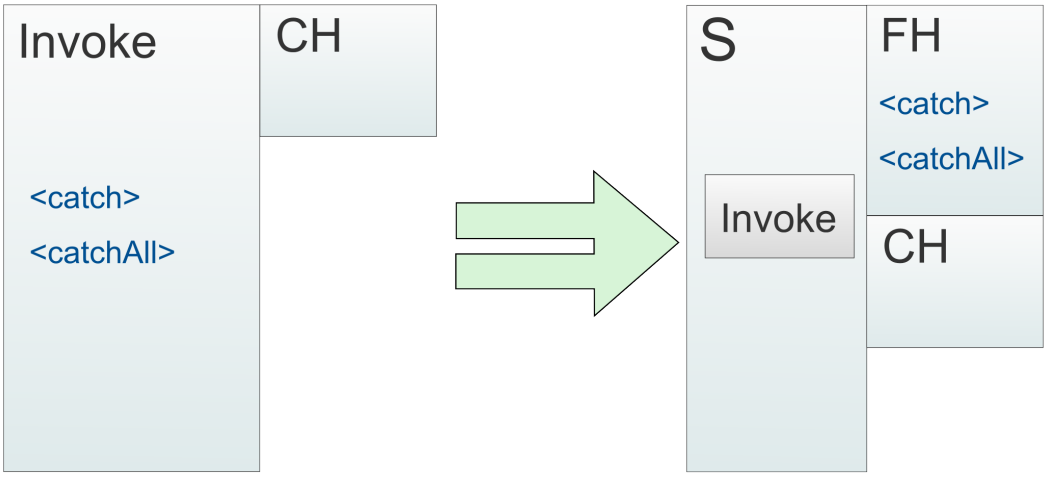


Abbildung 3.18.: Umwandlung eines Invokes in einen equivalenten Scope

dieses SyncPattern1.5 nicht mehr benötigt und wurde infolgedessen aus dem bestehenden System entfernt.

3.3. Termination Handler

Für den Termination Handler gilt, wie beim Fault Handler, dass die Grenzüberschreitungen von Links nur von innen nach außen erlaubt ist. Somit kann auf den Termination Handler, mit Hilfe kleiner Änderungen, der Korrekturalgorithmus vom Fault Handler angewandt werden.

Vergleicht man den Termination Handler Listing 2.8 mit dem Fault Handler Listing 2.6 ist ersichtlich, dass der TerminationHandler keine `<catch>` und `<catchAll>` Blöcke enthält und somit auch keine Variablendefinitionen. Infolgedessen muss das Auslagern von Variablen (siehe Abbildung 3.15) beim Termination Handler nicht berücksichtigt werden.

Abbildung 3.19 zeigt die Konsolidierung eines Termination Handlers mit synchronem ausgehenden Kommunikationslink. Die Aktivitäten aus dem Termination Handler werden in ein Scope S_{TH} ausgelagert. Da jeder Scope nur einen Termination Handler besitzt, muss der Name des neuen Scopes S_{TH} nicht mit einer Universally Unique Identifier (UUID) ergänzt werden, um den Scope eindeutig zu identifizieren, so wie es bei der Umwandlung von `<catch>` Blöcken im Fault Handler notwendig ist. Die restliche Struktur folgt dem Aufbau aus dem Abschnitt 3.1.8.

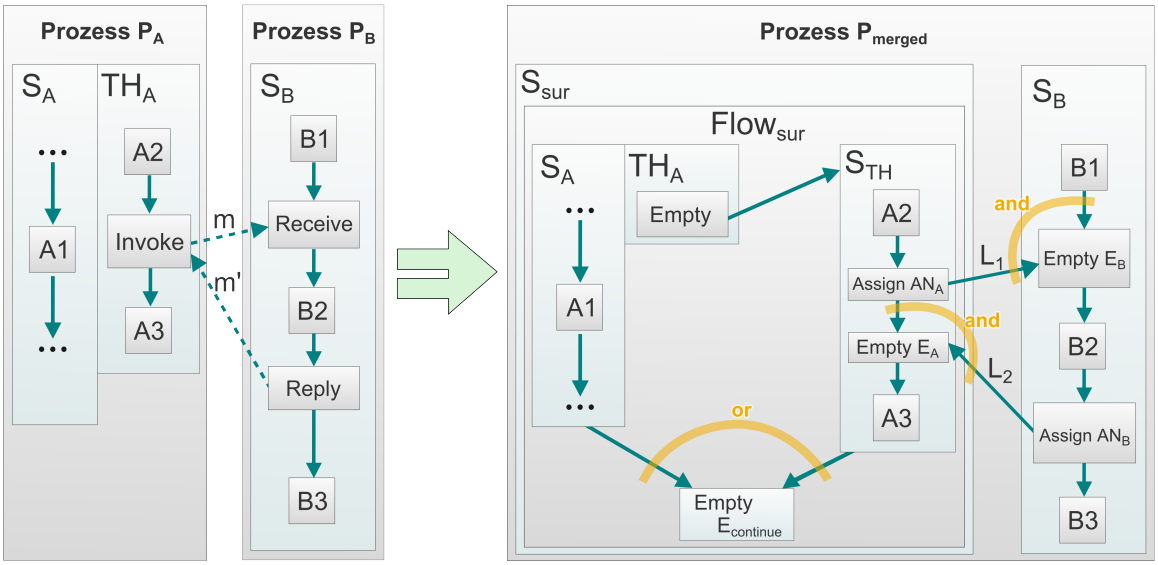


Abbildung 3.19.: Termination Handler mit synchronem ausgehenden Kommunikationslink

3.4. Event Handler

In diesem Abschnitt wird ein Lösungsansatz für die Konsolidierungen von Event Handlern, anhand dem Beispiel von Abbildung 3.20, erörtert. Das Listing 3.2 zeigt die grobe Struktur für die Definition eines Event Handlers. Der `<onEvent>` Tag ist für ein nachrichtengesteuertes und der `<onAlarm>` für ein zeitgesteuertes Ereignis zuständig. Beim Eintritt eines Ereignisses werden die zugehörigen Scopes ausgeführt.

Sei Prozess P_A für eine Autoherstellung zuständig, siehe Abbildung 3.20. Für die Produktion wird ein Tag benötigt. Der Prozess P_B erhält alle sechs Stunden eine synchrone Statusmeldung von Prozess P_A über den Event Handler EH_A.

Listing 3.2 BPEL-Konstrukt EventHandler [AAA+07]

```
<eventHandlers>?
        <onEvent ...>*
                ...
                <scope ...>...</scope>
        </onEvent>
        <onAlarm>*
                ...
                <scope ...>...</scope>
        </onAlarm>
</eventHandlers>
```

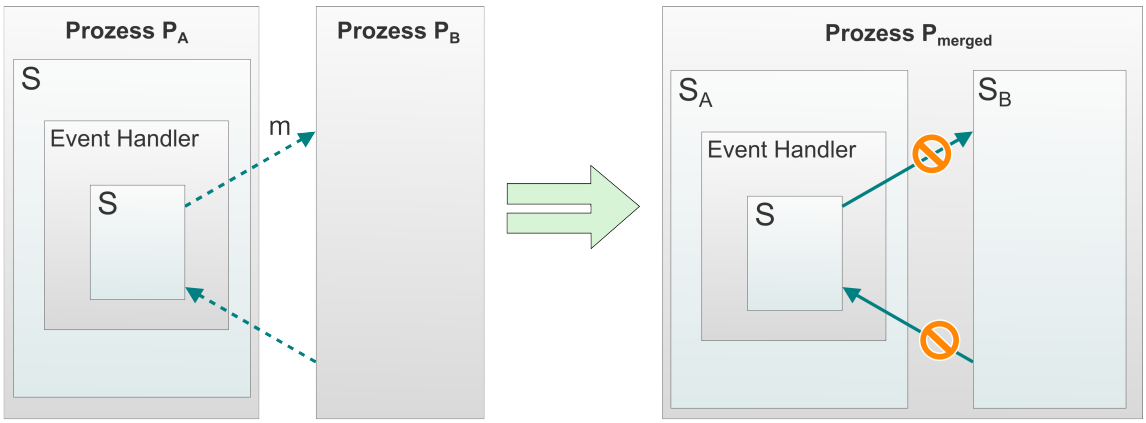


Abbildung 3.20.: Beispiel Konsolidierung für Event Handler

Bei einer Konsolidierung würde die Umwandlung wie in Abschnitt 3.1.3 stattfinden. Die Kommunikationslinks werden somit durch Synchronisationslinks ersetzt und der Event Handler verstößt somit gegen Grenzüberschreitungen von Links (Abschnitt 2.2.5). Der Grund hierfür liegt in Scope S_B in Prozess P_{merged}. Bei der Ausführung von Prozess P_{merged} würde alle sechs Stunden der Scope S_B einmal aufgerufen. Dieser würde jedoch bereits nach dem Aufruf den Status Completed (siehe Abschnitt 2.2.1) erreichen und könnte somit kein weiteres Mal aufgerufen (und den Status in Active wechseln) werden. Dies bedeutet, dass die iterative Ausführung von Event Handlern verboten ist. Das Scope im Event Handler EH_A kann jedoch durch weitere Instanziierungen mehrmals aufgerufen werden.

Um das Problem der Grenzüberschreitung zu lösen, könnte man versuchen, den entwickelten Algorithmus vom Fault Handler auf das Beispiel anzuwenden. Dies würde zwar die Grenzüberschreitungen innerhalb des Event Handlers lösen, jedoch nicht die Grenzüberschreitung im Prozess P_{merged}. Ein möglicher Lösungsansatz könnte wie folgt aussehen. Der Prozess P_A enthalte einen Event Handler EH_A. Es gelte die Bedingung, dass alle an der Konsolidierung beteiligten Prozesse keinen Zyklus zum Prozess P_A bilden dürfen. Der Event Handler EH_A greift über seinen Scope auf den Prozess P_B zu. Hieraus folgt, dass der Prozess P_B, sowie alle Prozesse die mit P_B kommunizieren, keinen Kommunikationslink zu Prozess P_A enthalten dürfen.

In Abbildung 3.21 ist dieser Lösungsansatz anhand eines Beispiel modelliert. Ein Prozess P_A enthält einen Event Handler mit einem Scope S_{EH}. Dieser Scope kommuniziert asynchron mit einem Prozess P_B, der wiederum mit einem Prozess P_C synchron interagiert. Die Kommunikationslinks $m2$ und $m3$ sind laut der oben definierten Bedingung verboten. Wenn diese Kommunikationslinks nicht existieren, so kann die Konsolidierung durchgeführt werden. Hierzu kommen die Prozesse P_B und P_C als Scope zusammen mit dem Scope S_{EH} in ein Flow, welches wiederum in einem Scope S_{sur} liegt. Dieses Scope S_{sur} wird nun an die ursprüngliche Position des Scope S_{EH} gesetzt. Hiermit hat man eine Konsolidierung mit Einschränkungen erreicht.

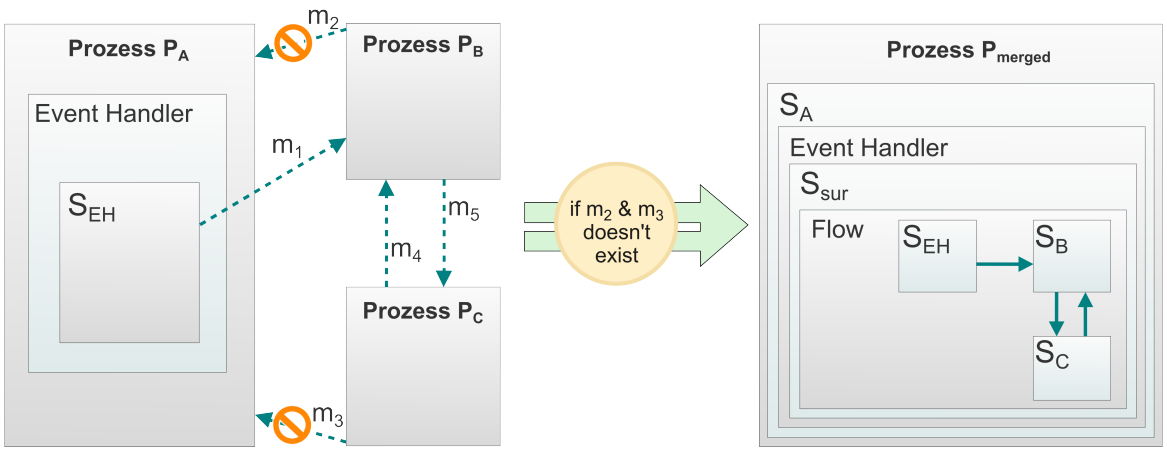


Abbildung 3.21.: Lösungsansatz für Event Handler

Ein zu berücksichtigender Fall ist, dass nach der Konsolidierung der Prozess Scope S_A gehindert wird, einen Zustandswechsel auf Completed durchzuführen. Vor der Konsolidierung wurde der Prozess P_B durch einen asynchronen Kommunikationsaufruf aus dem Scope S_{EH} im Event Handler gestartet. Der Event Handler ist somit nicht zeitlich an der Ausführung des Prozesses P_B gebunden. Dies trifft nach der Konsolidierung nicht mehr zu. Der Event Handler muss auf den Scope S_B warten. Laut Spezifikation ist die Lebensspanne eines Event Handlers an die seines umgebenden Scopes gebunden. Beim Beenden eines Scopes werden seine Event Handler informiert, ihre Arbeit ebenfalls zu beenden. Alle aktiven Instanzen eines Event Handlers dürfen jedoch ihre Arbeit abschließen. Hieraus folgt, dass der Prozess Scope S_A auf seinen Event Handler warten muss, der wiederum auf Scope S_B wartet, wodurch eine Verzögerung bei der Ausführung von Scope S_B entstehen kann.

Betrachtet man das vorangegangene Beispiel, ist zu erkennen, dass dies für Fälle gilt, bei denen ein Prozess von einem Event Handler aufgerufen wird. Ruft jedoch ein Prozess einen Event Handler auf, so müsste ein anderer Lösungsansatz verfolgt werden. Ruft ein Prozess P_B einen Event Handler EH_A in Prozess P_A auf, so wird der Prozess P_B nur einmal ausgeführt. Der Event Handler hingegen kann zusätzlich noch von anderen Prozessen aufgerufen werden. Ein Lösungsansatz könnte sein, dass der Scope vom Event Handler EH_A vor Konsolidierung auf dasselbe Level (Peer-Scopes) kopiert wird wie der Prozess P_B. Der Event Handler EH_A liegt somit redundant vor, jedoch ist die Kopie an einen bestimmten Prozess gekoppelt und wird nur ein einziges Mal ausgeführt.

3.5. Compensation Handler

In diesem Abschnitt wird auf die Konsolidierung von Compensation Handler eingegangen. Es wird anhand des Beispiels Abbildung 3.22 gezeigt, wie Fault Handler, Termination Handler und Compensation Handler (FTC-Handler) untereinander in Beziehung stehen [KRL09]. Dann wird die Auswirkungen einer Konsolidierung mit dem bestehenden System in Bezug auf den Compensation Handler betrachtet und ein Lösungsansatz vorgestellt.

3.5.1. Beziehung der FCT-Handler

In Abbildung 3.22 findet eine Autoproduktion Scope S_A statt. Der Scope S_B ist für den Zusammenbau von Motor und Fahrgestell zuständig. Dabei bereitet Scope S_C das Fahrgestell für den Zusammenbau vor und darauffolgend findet der Motoreinbau in Scope S_D statt. Die Abnahme des Zusammenbaus erfolgt parallel durch einen KFZ-Meister, welches durch die Aktivität B repräsentiert wird. Die Aktivität A läuft ebenfalls parallel zur Produktion und ist für die Einholung von Genehmigungen zuständig. Sei nun der Zusammenbau abgeschlossen. Die Scopes S_C und S_D befinden sich somit im Status Completed (siehe Abschnitt 2.2.1). Die Abnahme durch den KFZ-Meister läuft noch, Aktivität B, daraus folgt, dass der Scope S_B noch läuft und der Fault Handler und Termination Handler noch installiert sind.

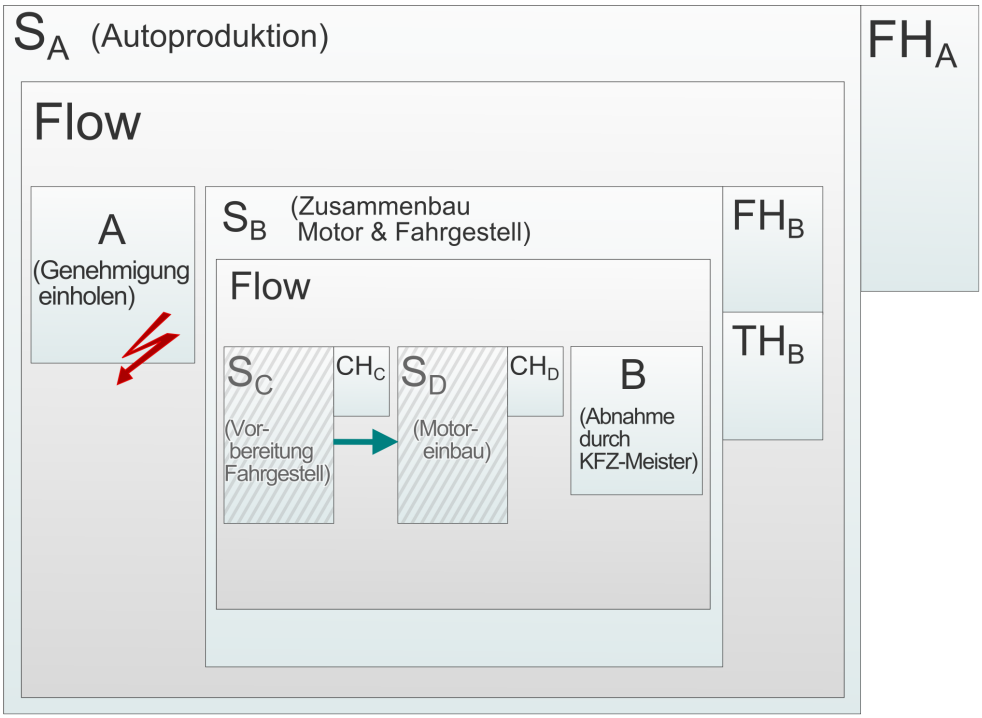


Abbildung 3.22.: Beispiel für die Beziehung von FTC-Handler

Nun verursacht Aktivität A einen Fehler, wofür der Fault Handler FH_A zuständig ist. Es wird davon ausgegangen, dass alle FCT-Handler die Standard Handler sind. Es wird somit der Standard Fault Handler aufgerufen, der eine Kompensation startet. Zuerst werden alle laufenden Aktivitäten beendet, hierzu werden die Fault Handler der laufenden Aktivitäten deinstalliert und deren Termination Handler, hier TH_B, gestartet. Der Termination Handler TH_B beendet die Aktivität B und startet daraufhin die Kompensation von Scope S_C und S_D in der Standard Kompensationsreihenfolge S_D **m** S_C.

3.5.2. Konsolidierung des Compensation Handlers

Während beim Fault Handler und Termination Handler die Grenzüberschreitung von Links nur von innen nach außen erlaubt ist, verbietet der Compensation Handler jegliche Grenzüberschreitung von Links. Bei der Konsolidierung mit dem bestehenden System können aber genau solche Grenzüberschreitungen stattfinden. Ein möglicher Lösungsansatz könnte wie in Abbildung 3.23 aussehen. Es existieren zwei Scopes S_A und S_D, die miteinander über den Compensation Handler CH_A asynchron kommunizieren. Nach einer Konsolidierung würde der Compensation Handler CH_A einen Kontrolllink zu Scope S_D haben. Um nun die Grenzverletzung zu beseitigen, wird für den Compensation Handler CH_A ein weiterer Scope S_{CHA} angelegt und die Geschäftslogik von CH_A nach S_{CHA} übertragen. Zusätzlich wird der Compensation Handler CH_A deaktiviert. Hierzu setzt man im Compensation Handler CH_A eine Empty Aktivität ein. Im Laufe eines Kompensationsvorganges kann so der Status des jeweiligen Scopes auf Compensated (siehe Abschnitt 2.2.1) wechseln. Für jeden Scope wird daraufhin eine alternative Kompensation erzeugt, indem die neu erzeugten Scopes S_{CH} in einer Flow Aktivität in der Kompensationsreihenfolge definiert werden. Die Flow Aktivität ist über Links, mit disjunkter `<joinCondition>` (siehe Abschnitt 2.2.5), mit dem übergeordneten Fault Handler und Termination Handler verbunden. Im Fault Handler und im Termination Handler wird vor dem `<compensate>` Aufruf eine Empty Aktivität hinzugefügt, welche die Quellaktivität für den Link zum jeweiligen Flow ist. Das Starten der Kompensation für die Aktivitäten in Scope S_A findet durch den Fault Handler und den Termination Handler statt, welche den Flow F_1 aufrufen. Flow F_1 enthält die Kompensationslogik von den Scopes S_B und S_C. Um die Kompensation vom Compensation Handler CH_A zu starten, muss der Fault Handler oder Termination Handler vom übergeordneten Scope S_{PBDA} den Flow F_2 aufrufen, welcher die Kompensationsreihenfolge S_{CHC}, S_{CHB} nach S_{CHA} ausführt. Es ist zu beachten, dass bei der Nachbildung der Kompensationsreihenfolge redundante Kompensations-Scopes enstehen, die jedoch zu vernachlässigen sind, da hierdurch die Grenzverletzungen beseitigt werden. Weiterhin muss der alternativen Kompensation über Scope mitgeteilt werden, ob ein Scope überhaupt kompensiert werden kann, also auf dem Status Completed ist. Hierzu wird für jeden Scope eine neue Variable `isScopeCompleted=false` eingeführt, welche in jedem Scope am Anfang auf `isScopeComleted=true` gesetzt wird. Tritt innerhalb eines Scopes ein Fehler auf, wird der Fault Handler aktiviert und setzt die Variable wieder auf `isScopeCompleted=false`. Beim Aufruf des Termination Handler wird ebenfalls die Variable `isScopeCompleted=false` gesetzt, da der Scope beendet und die Termination Handler Aktivitäten ausgeführt werden. Die Zuweisung der Variablen findet am Anfang eines Scopes statt, da am Ende des Scopes Fehler bei der Zuweisung auftreten könnten und somit

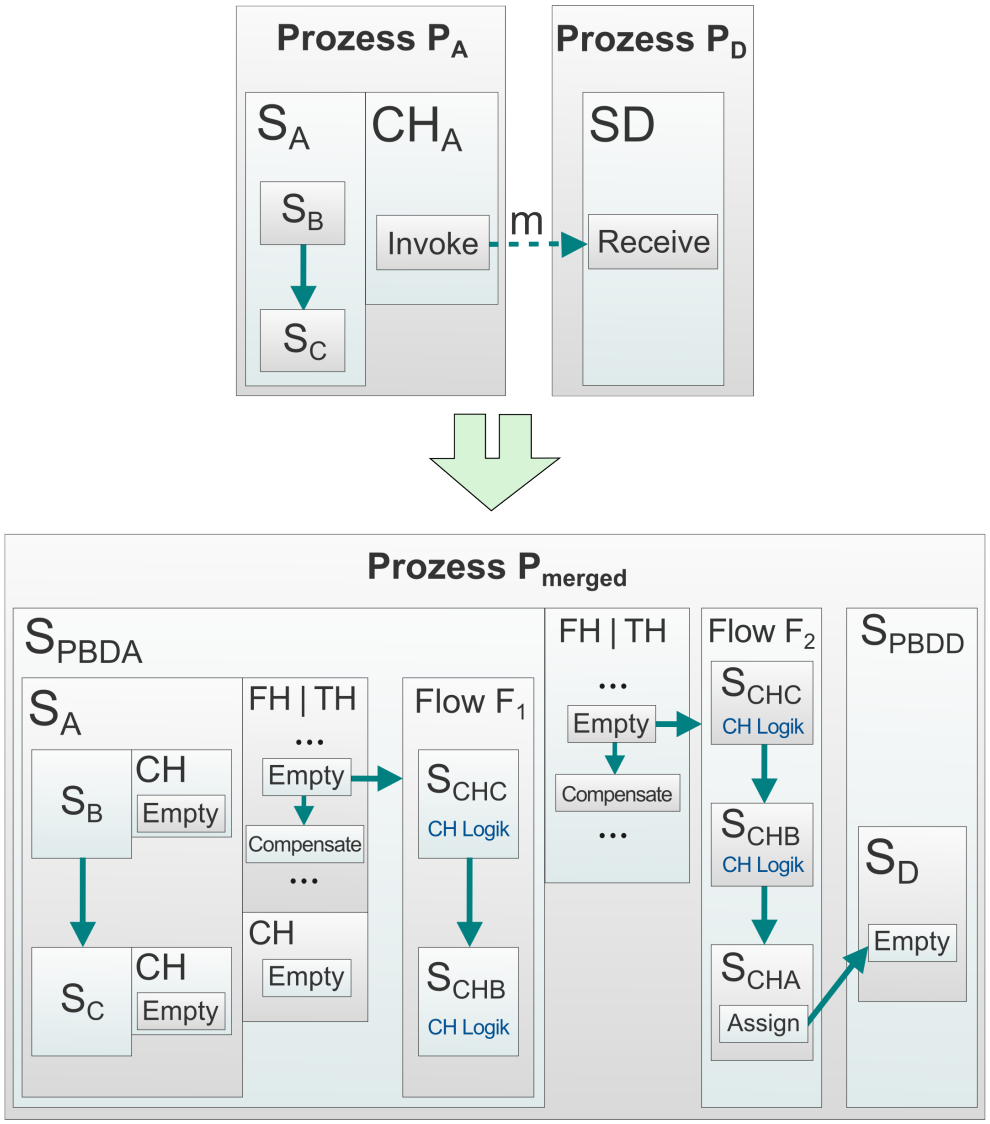


Abbildung 3.23.: Beispiel Konsolidierung

ein Scope mit Status Completed auf Faulted gesetzt werden würde. Bei der Kompensation wird die zugehörige Variable `isScopeCompleted` auf `true` geprüft. Stimmt die Variable mit dem Wert überein, werden die Aktivitäten ausgeführt. Ist dies nicht der Fall, werden die Kompensationsaktivitäten für den jeweiligen Scope nicht ausgeführt und der nächste Kompensations-Scope überprüft. Außerdem ist bei der Erstellung der Kompensationsreihenfolge zu beachten, dass alle FCT-Handler in den neuen Kompensations-Scopes deaktiviert werden müssen. Bei der Kompensation eines Scopes mit der CompensateScope Aktivität, muss die CompensateScope Aktivität durch eine Empty Aktivität ersetzt und mit dem zugehörigen Kompensations-Flow verlinkt werden.

3.5.3. Algorithmus für den CompensationHandler

Um die aus Abschnitt 3.5.2 vorgestellte Konsolidierung umzusetzen, wird ein Algorithmus benötigt, der die Kompensationsreihenfolge eines bestehenden Prozesses nachbildet. Als Ausgangsbasis wird der Prozess aus Abbildung 3.25 genommen. Der Tabelle 3.2 können Beschreibungen von Funktionen und Variablen entnommen werden, welche der Algorithmus verwendet. Der Algorithmus ist ein Prototyp und unterliegt somit einigen Beschränkungen.

Beschränkungen:

- Wiederholende Strukturen sind ausgeschlossen.
- Kompensationsreihenfolge innerhalb von FCTE-Handlern wird nicht beachtet.
- Die Veränderung der Kompensationsreihenfolge durch Anomalien wird, durch Restrukturierung des Prozesses, ausgeschlossen. [KRL09]
- Die gegebenen Prozesse sind korrekt (siehe Abschnitt 2.2.7).

Tabelle 3.2.: Pseudocode: Funktions- und Variablenbeschreibung

Funktion/Variable	Bedeutung
TreeNode	Ist ein Knotenpunkt in einem Baum. Er enthält eine BPEL Aktivität, eine BPEL Kompensationsaktivität, seine Nachfolgeraktivität, die ebenfalls ein TreeNode ist, eine Liste seiner Kinder (ebenfalls TreeNodes) sowie eine Liste seiner Linknachfolger (ebenfalls TreeNodes).
rootTn	Ein TreeNode Startknoten, von dem aus durch den Prozess traversiert wird.
resultFlow	Flow Aktivität, welche die alternative Kompensation enthält.
TN	Die Menge aller erstellten TreeNodes.
iterateTN	Variable, welche den letzten verwendeten Knoten speichert.
tmpTN	Temporäre Variable für TreeNodes.
createCompensationScope (*rootTn*,*TN*)	Traversiert durch eine Scope Aktivität und bereitet mit Hilfe von TreeNodes die Kompensationsreihenfolge vor.
createTreeIterator (*rootTn*)	Liefert einen Iterator zurück, mit welchem durch die BPEL Aktivität traversiert wird.
hasMoreElements (*treeIterator*)	Liefert *true* zurück, wenn ein weiteres Element im Iterator existiert, ansonsten $false$.

Fortsetzung auf der nächsten Seite

Tabelle 3.2 – Fortsetzung von vorheriger Seite

Funktion/Variable	Bedeutung
getNextTreeIteratorElement (*treeIterator*)	Liefert das nächste Element im Iterator zurück.
prune (*treeIterator*)	Unterbricht die Traversierung in tiefere Ebenen und signalisiert dem Iterator, dass diese Ebenen übersprungen werden sollen.
getType(a)	Liefert den Typ der übergebenen Aktivität zurück.
createTN(a)	Erzeugt einen neuen TreeNode und setzt die Knotenaktivität auf a.
addTNtoSetTN (*tmpTN*,*TN*)	Fügt die TreeNode *tmpTN* zur Menge *TN* hinzu.
setSuccessorForIterateTN (*iterateTN*,*tmpTN*)	Im TreeNode *iterateTN* wird die Nachfolgeraktivität auf *tmpTN* gesetzt.
getLeafTN (*tmpTN*)	Liefert vom Knotenpunkt *tmpTN* durch Rekursion den Blattknoten zurück.
createCompensationFlow (*tmpTN*,*TN*)	Traversiert durch eine Flow Aktivität und bereitet mit Hilfe von TreeNodes die Kompensationsreihenfolge vor.
addChildtoFlowTN (*tmpTN*,*rootTN*)	Der TreeNode *tmpTN* wird zur Liste der Kinder von *rootTN* hinzugefügt.
getAllLinks (*rootTn*)	Holt aus dem TreeNode *rootTn* die Knotenaktivität und liefert von dieser allee Links zurück.
getTnFromSet (*sourceA*,*TN*)	Liefert den TreeNode zurück, bei dem die Knotenaktivität mit der *sourceA* Aktivität übereinstimmt.
getImmediateEnclosingSrcSoF (*l*)	Von Link *l* wird die Source Aktivität ermittelt und von dieser der erste umgebende Scope oder Flow zurückgeliefert.
getImmediateEnclosingTarSoF (*l*)	Von Link *l* wird die Target Aktivität ermittelt und von dieser der erste umgebende Scope oder Flow zurückgeliefert.
setSuccessorToSourceLeaf (*sourceTn*,*targetTn*)	Ermittelt den Blattknoten über getLeafTN(*sourceTn*) und fügt in dessen Liste die Linknachfolger des TreeNodes *targetTn* ein.
createCompensationOrder (*TN*, *resultFlow*)	Erzeugt im Flow *resultFlow* die Kompensationsreihenfolge.
getActivityFromTN (*tn*,*TN*)	Liefert die Knotenaktivität vom TreeNode *tn* zurück.
Fortsetzung auf der nächsten Seite	

Tabelle 3.2 – Fortsetzung von vorheriger Seite

Funktion/Variable	Bedeutung
getCompensationFlow (tn)	Liefert die Kompensationsaktivität vom TreeNode tn zurück.
addAllChildsToFlow ($cflow$,tn)	Fügt alle Kinder aus tn dem Kompensationsflow $cflow$ hinzu.
addScope (tn)	Erstellt einen Kompensations-Scope und setzt diesen auf die Kompensationsaktivität von tn. Außerdem werden die Links zwischen TreeNode und Nachfolger in umgekehrter Reihenfolge erstellt.
addCompensationActivity (tn, $resultFlow$)	Fügt dem Flow $resultFlow$ die Kompensationsaktivität aus tn hinzu.
createFlow()	Erzeugt eine neue Flow Aktivität.

Funktionsweise des TreeIterators

Bei der Anwendung des Algorithmus ist es notwendig durch die Baumstruktur einer Scope Aktivität zu traversieren. Der TreeIterator stellt diese Funktion bereit. Anhand der Abbildung 3.24 wird die Funktionsweise erläutert. Der Scope S_A dient als Startknoten für die Iteration. Mit der Funktion `hasMoreElements` wird überprüft, ob der aktuelle Knoten über weitere Elemente verfügt.

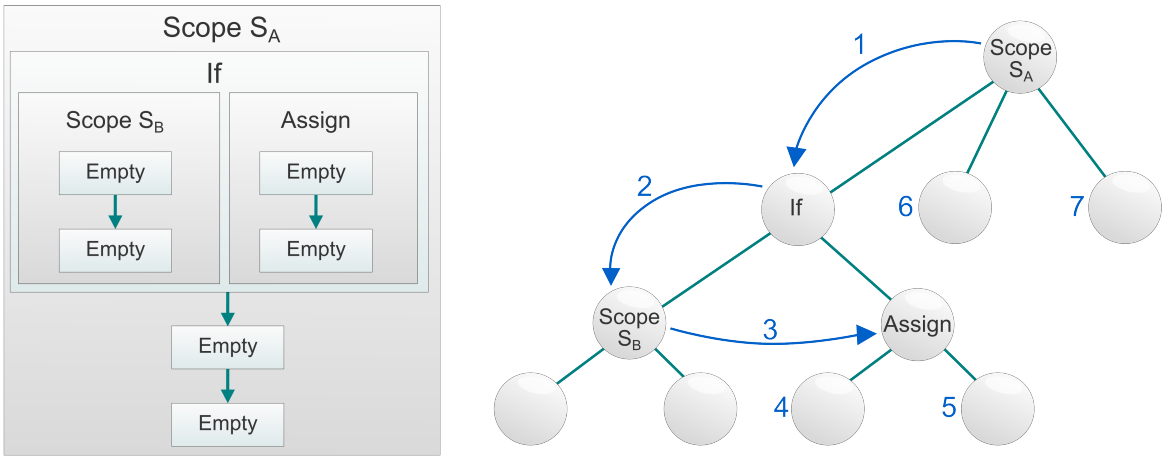


Abbildung 3.24.: Beispiel für TreeIterator

Da im Beispiel der Startknoten der aktuelle Knoten ist, verfügt dieser über drei Kindelemente. Daraufhin wird das erste Element, dies wäre die If Aktivität, mit der Funktion

`getNextTreeIteratorElement` von dem TreeIterator zurückgeliefert. Für die Auswahl des ersten Elementes orientiert sich der TreeIterator an der XML Datei und wählt dabei aus der Definitionsreihenfolge aus. Ein weiterer Aufruf von `getNextTreeIteratorElement` liefert den Scope S_B zurück. Wird dann die Funktion `prune` aufgerufen, wird die Traversierung zu weiteren Kindelementen von Scope S_B unterbrochen und das nächste Element verarbeitet. Im Beispiel wäre dies die Assign Aktivität. Wird die Funktion `prune` nicht mehr aufgerufen, so werden die Kindelemente von der Assign Aktivität und darauffolgend die letzten zwei Kindelemente von Scope S_A verarbeitet.

Anwendung des Algorithmus

Um aus den Prozess in Abbildung 3.25 eine Kompensationsreihenfolge zu erhalten, traversiert der Algorithmus durch die Baumstruktur des Prozesses und baut diesen mit Hilfe von TreeNodes (siehe Tabelle 3.2) nach. Beim Aufbau der Baumstruktur muss der Prozess unter Berücksichtigung von `<flow>` und `<sequence>` Aktivitäten analysiert werden (siehe Abschnitt 2.2.7). Durch die gegebene Baumstruktur des Prozesses, aufgrund der Basis von XML, wird eine Kombination aus Breitensuche und Tiefensuche angewandt.

Mit der Funktion `createCompensationOrderForScope(scope)` (siehe Algorithmus 3.1) wird der Algorithmus gestartet. Als Eingabe wird eine Scope Aktivität erwartet. Als Beispiel wird der Prozess aus Abbildung 3.25 in einen Scope S_{PBD} umgewandelt und der Funktion übergeben. Zu Beginn des Algorithmus existieren noch keine TreeNodes. Es wird ein Startknoten `rootTN` erstellt, in welchem die BPEL Aktivität mit Scope S_{PBD} referenziert wird. Anschließend wird mit den Funktionen `createCompensationScope` (siehe Algorithmus 3.2) und `createCompensationFlow` (siehe Algorithmus 3.3) die TreeNode Baumstruktur erzeugt und damit die Menge TN gefüllt. Mit der Funktion `createCompensationOrder` (siehe Algorithmus 3.4) wird abschließend der Kompensations-Flow erzeugt und als Ergebnis zurückgeliefert (siehe Abbildung 3.26).

Algorithmus 3.1 Pseudocode: Startfunktion des Kompensationsalgorithmus

```
1: procedure createCompensationOrderForScope(scope)
2:     TN = ∅
3:     rootTn = createTN(scope)
4:     createCompensationScope(rootTn,TN)
5:     resultFlow = createCompensationOrder(TN)
6:     return resultFlow
7: end procedure
```

Für den Aufbau der TreeNode Baumstruktur wird im ersten Schritt der Algorithmus 3.2 ausgeführt. Als Eingabeparameter werden `rootTn`, der Startknoten und `TN` übergeben. Mit dem TreeIterator wird durch den Scope traversiert. Hierbei werden alle Elemente auf Flow und Scope Aktivitäten geprüft.
Entspricht ein Element einem Scope, so wird ein TreeNode mit der Funktion `createTN` angelegt und mit der Funktion `addTNtoSetTN` der Menge `TN` hinzugefügt. Dies dient beim

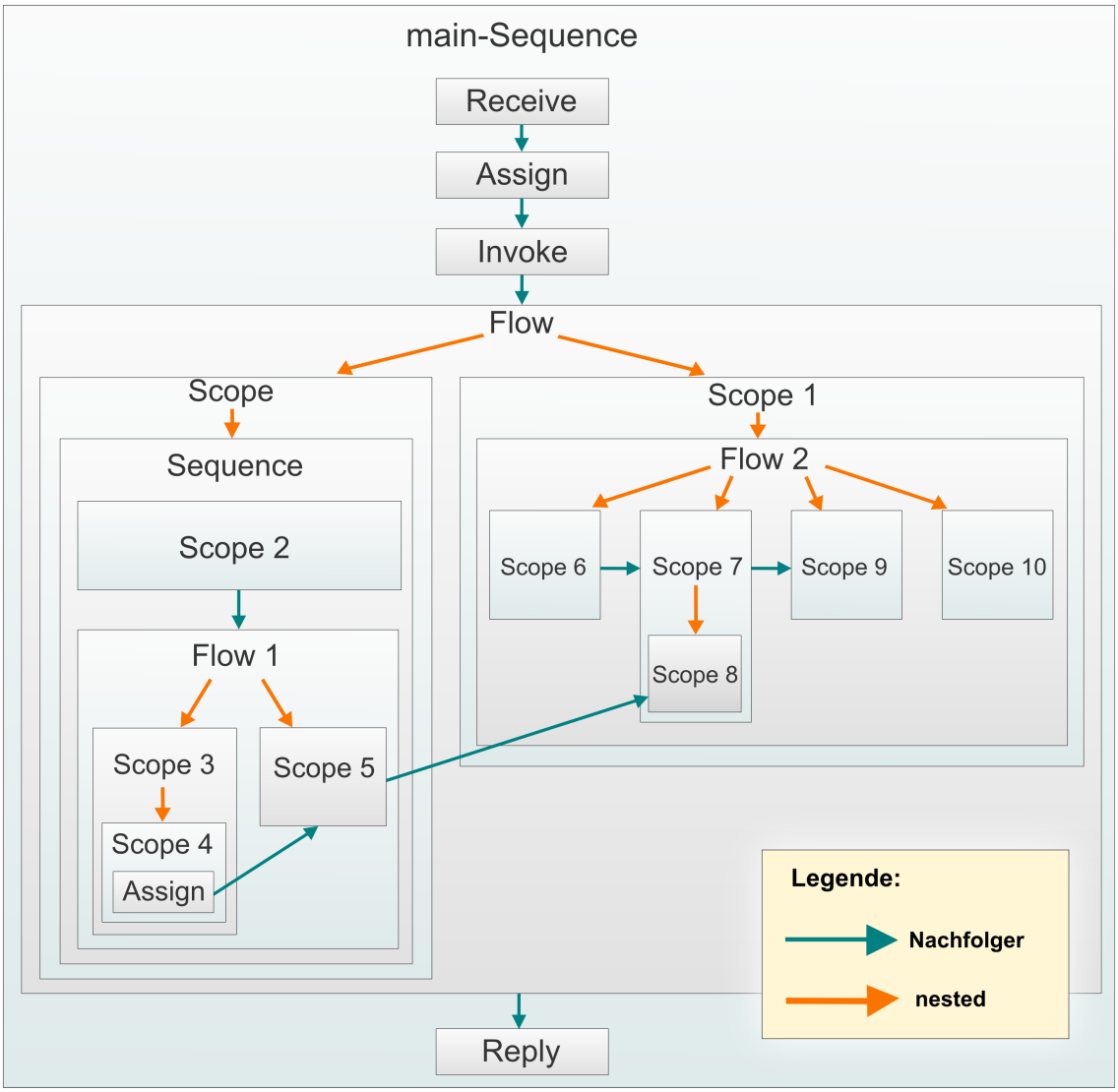


Abbildung 3.25.: Beispielprozess für Kompensation

späteren Durchlauf von `TN` im Algorithmus 3.4 dem Aufbau der umgekehrten Kompensationsreihenfolge. Nachfolgend wird dieser Scope durch den rekursiven Aufruf der Funktion `createCompensationScope` ebenfalls analysiert. Daraufhin wird der Nachfolger mit der Funktion `setSuccessorForIterateTN` sowie die Variable `iterateTN` mit der Funktion `getLeafTN` gesetzt. Die Funktion `getLeafTN(tmpTN)` (siehe Tabelle 3.2) liefert dabei den benötigten Blattknoten (tiefster Nachfolgerknoten des übergebenen TreeNodes), da durch die Rekursion von Scope und Flow Aktivitäten dieser Blattknoten als nächstes in der Kompensationsreihenfolge auftritt. Mit der Funktion `prune` wird die Scope Aktivität nicht weiter durchlaufen, da dies bereits durch den rekursiven Aufruf erfolgte. Entspricht der aktuelle TreeIterator-Knoten einer Flow

Algorithmus 3.2 Pseudocode für den Aufbau der Kompensationsreihenfolge: Scope

```
1:  procedure createCompensationScope(rootTn,TN)
2:      iterateTN = rootTn
3:      treeIterate =createTreeIterator(rootTn)
4:      while hasMoreElements(treeIterator) do          // Durchlaufe alle Aktivitäten
5:          a =getNextTreeIteratorElement(treeIterator)     // Beziehe nächstes Element
6:          if getType(a) ∈ A_restricted then       // Überspringe alle nicht erlaubten Elemente
7:              prune(treeIterator)
8:          else if getType(a) ∈ A_scope then
9:              tmpTN =createTN(a)
10:             addTNtoSetTN(tmpTN,TN)
11:             createCompensationScope(tmpTN,TN)
12:             setSuccessorForIterateTN(iterateTN,tmpTN)
13:             iterateTN =getLeafTN(tmpTN)
14:             prune(treeIterator)
15:         else if getType(a) ∈ A_flow then
16:             tmpTN =createTN(a)
17:             addTNtoSetTN(tmpTN,TN)
18:             createCompensationFlow(tmpTN,TN)
19:             setSuccessorForIterateTN(iterateTN,tmpTN)
20:             iterateTN =getLeafTN(tmpTN)
21:             prune(treeIterator)
22:         end if
23:     end while
24: end procedure
```

Aktivität, so wird genauso verfahren wie bei der Scope Aktivität mit dem Unterschied, dass anstatt der Funktion `createCompensationScope` die Funktion `createCompensationFlow` aufgerufen wird.
Ausgehend vom Beispiel in Abbildung 3.25 würden die Aktivitäten Receive, Assign und Invoke übersprungen und der Flow `Flow` als Nachfolger von Scope S_{PBD} gesetzt. Daraufhin würde die Funktion `createCompensationFlow` ausgeführt.

Die Funktion `createCompensationFlow` traversiert, wie die Funktion `createCompensationScope`, durch die übergebene Aktivität, welche sich in der Variablen `rootTn` befindet. Ebenso wird zwischen Scope und Flow unterschieden und jeweils TreeNodes erstellt, der Menge TN hinzugefügt sowie die rekursiven Funktionen aufgerufen. Zusätzlich werden die Scope und Flow Aktivitäten in die Liste der Kinder von Variablen $rootTn$ hinzugefügt.

Nach dem Traversieren durch den Flow werden alle Links durchlaufen und von den Quellaktivitäten $sourceA$ und den Zielaktivitäten $targetA$ die umgebenden Scopes oder Flows mit den Funktionen `getImmediateEnclosingSrcSoF` und `getImmediateEnclosingTarSoF` ermittelt. Die Ermittlung der umgebenen Scopes und Flows ist notwendig, da ein Link nicht nur von einer Scope oder Flow Aktivität ausgehen kann. Die Assign Aktivität (siehe Abbbildung 3.25) in

Scope S_4 zeigt auf den Scope S_5. Um nun diesen Link in der Kompensationsreihenfolge zu beachten, wird auf den übergeordneten Scope oder Flow referenziert. Von den Aktivitäten *sourceA* und *targetA* werden mit der Funktion `getTnFromSet` die jeweiligen TreeNodes *targetTn* und *sourceTn* ermittelt. Der TreeNode *targetTn* wird daraufhin mit der Funktion `setSuccessorToSourceLeaf` zur Liste der Linknachfolger des Blattknotens von TreeNode *sourceTn* hinzugefügt. Die Liste der Linknachfolger wird benötigt, damit keine Nachfolger überschrieben werden. In Abbildung 3.25 wäre dies bei S_7 der Fall, wobei S_9 den Nachfolger S_8 in S_7 überschreiben würde.

Die Menge TN enthält die TreeNodes $TN = \{TN_{PBD}, TN_{Flow_0}\}$. Die Funktion `createCompensationFlow` ruft für beide Kindelemente, Scope S_0 und Scope S_1, die Funktion `createCompensationScope` auf. Im Folgenden wird der Scope S_0 betrachtet, da der Scope S_1 implizit verarbeitet werden kann. Der Scope S_0 setzt als Nachfolger den Scope S_2. Dieser wiederum erhält als Nachfolger Flow F_1. Die Scopes S_3 und S_5 werden zur Liste der Kinder von Flow F_1 hinzugefügt. Der Nachfolger von Scope S_3 wird Scope S_4. Danach wird die Liste der Links von Flow F_1 durchlaufen. Hierbei wird der umgebende Scope S_4 sowie der Scope S_5 ermittelt. Somit wird Scope S_5 in die Liste der Linknachfolger von Scope S_4 aufgenommen. Hierdurch wurde eine TreeNode Baumstruktur aus Scope und Flow Aktivitäten erzeugt.

Nun muss anhand der Menge TN die Kompensationsreihenfolge aufgebaut werden. Hierzu werden alle TreeNodes durchlaufen. Für Flow Aktivitäten werden ebenfalls Flow Aktivitäten in der Kompensationsreihenfolge angelegt, da die Aktivitäten, die nicht miteinander verbunden sind, parallel ausgeführt werden können. Die Kinder werden ebenfalls zur Kompensations-Flow Aktivität hinzugefügt (`addAllChildsToFlow(cflow,tn)`). Ist ein TreeNode ein Scope, so wird für dieses ein Kompensations-Scope erstellt. Innerhalb dieses Scopes findet eine Prüfung auf die Variable `isScopeCompleted=true` statt, die, beim Zutreffen der Bedingung, die Logik aus dem ehemaligen Compensation Handler ausführt. Zudem werden der Nachfolger und die Linknachfolger in umgekehrter Reihenfolge verlinkt (`addScope(a)`). Zum Schluss werden alle Kompensationsaktivitäten aus den TreeNodes in den Flow *resultFlow* übertragen. In Abbildung 3.26 ist das Ergebnis der Anwendung des Algorithmus zu finden.

Algorithmus 3.3 Pseudocode für den Aufbau der Kompensationsreihenfolge: Flow

```
1:  procedure createCompensationFlow(rootTn,TN)
2:      treeIterate =createTreeIterator(rootTn)
3:      while hasMoreElements(treeIterator) do                  // Durchlaufe alle Aktivitäten
4:          a =getNextTreeIteratorElement(treeIterator)         // Beziehe nächstes Element
5:          if getType(a) ∈ A_restricted then                   // Überspringe alle nicht erlaubten Elemente
6:              prune(treeIterator)
7:          else if getType(a) ∈ A_scope then
8:              tmpTN =createTN(a)
9:              addTNtoSetTN(tmpTN,TN)
10:             addChildtoFlowTN(tmpTN,rootTN)
11:             createCompensationScope(a,TN)
12:             prune(treeIterator)
13:         else if getType(a) ∈ A_flow then
14:             tmpTN =createTN(a)
15:             addTNtoSetTN(tmpTN,TN)
16:             addChildtoFlowTN(tmpTN,rootTN)
17:             createCompensationFlow(a,TN)
18:             prune(treeIterator)
19:         end if
20:     end while
21:     L =getAllLinks(rootTn)
22:     for all l ∈ L do                                        // Prüfe alle Links
23:         sourceA =getImmediateEnclosingSrcSoF(l)
24:         sourceTn =getTnFromSet(sourceA,TN)
25:         targetA =getImmediateEnclosingTarSoF(l)
26:         targetTn =getTnFromSet(targetA,TN)
27:         setSuccessorToSourceLeaf(sourceTn,targetTn)
28:     end for
29: end procedure
```

Algorithmus 3.4 Pseudocode für den Aufbau der Kompensationsreihenfolge

```
1: procedure createCompensationOrder(TN)
2:     resultFlow =createFlow()
3:     for all tn ∈ TN do                                   // Verarbeite alle tn
4:         a =getActivityFromTN(tn,TN)
5:         if a ∈ A_flow then
6:             cflow =getCompensationFlow(tn)
7:             addAllChildsToFlow(cflow,tn)
8:         end if
9:         if a ∈ A_scope then
10:            addScope(tn)          // Erstelle Scope mit isScopeCompleted und setze Link
   (Kind,Vater)
11:        end if
12:        addCompensationActivity(tn, resultFlow)
13:    end for
14:    return resultFlow
15: end procedure
```

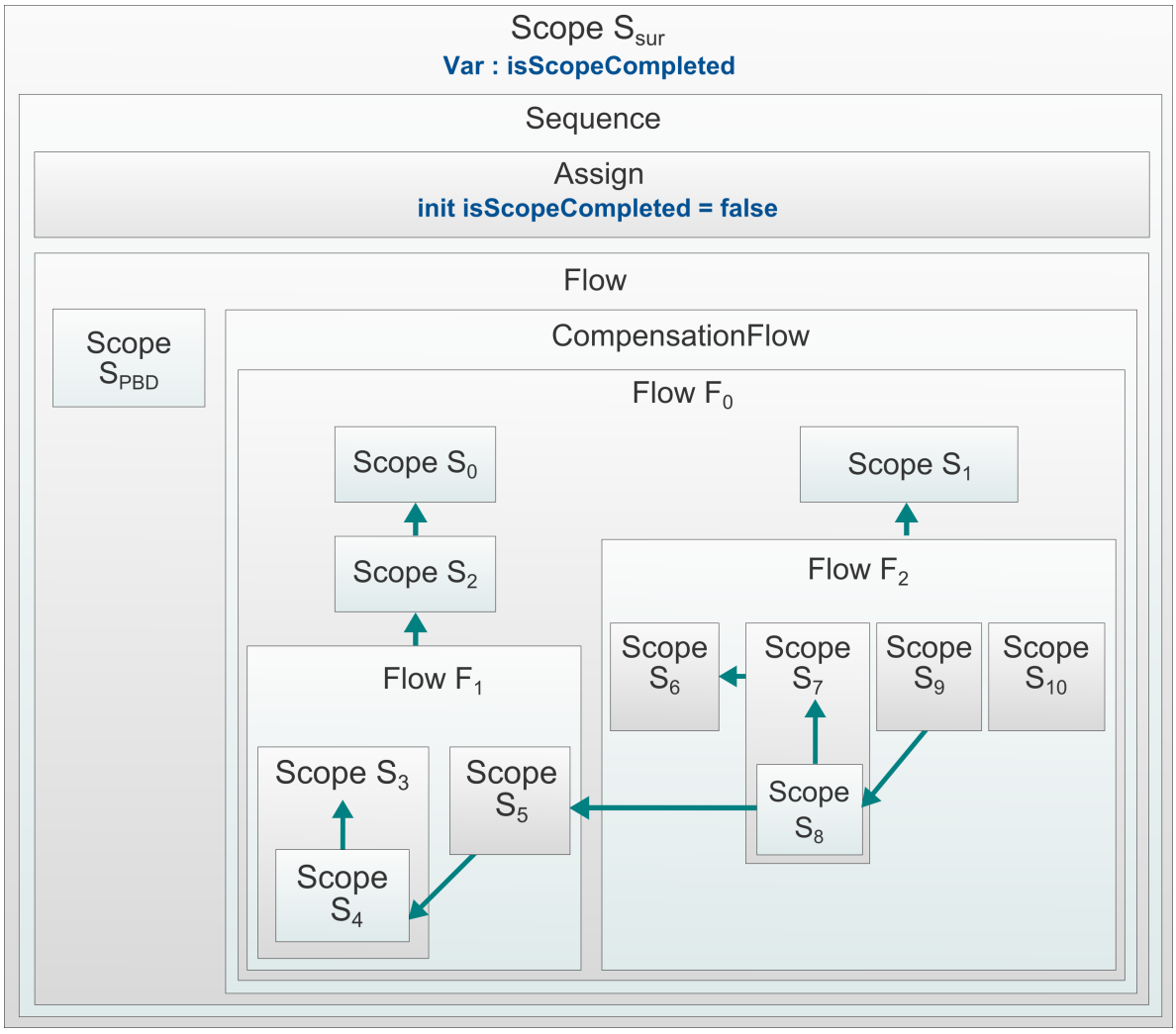


Abbildung 3.26.: Ergebnis des Algorithmus für die Kompensationsreihenfolge

4. Implementierung

Die nachfolgende Implementierung basiert auf dem System, welches in Kapitel 2.5 beschrieben wurde. Als Entwicklungsumgebung wurde Eclipse Juno mit Java 7 (64bit) verwendet. Als BPEL-Engine kam die Apache ODE v.1.3.6 (Orchestration Director Engine) zum Einsatz.

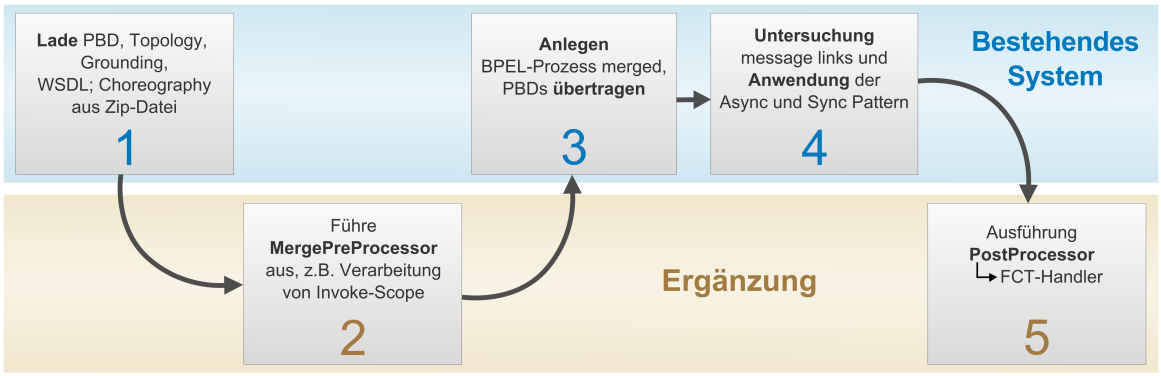


Abbildung 4.1.: Vorgehensweise bei Konsolidierung einer Choreographie

Die Konsolidierung einer Choreographie gliedert sich, wie in Abbildung 4.1 zu sehen, in fünf Schritte. Zuerst werden die Choreographie-Dateien, PBD, Topology, Grounding und WSDL, aus der zugehörigen ZIP-Archivdatei gelesen. Im nächsten Schritt wird der `MergePreProcessor` ausgeführt (siehe Abschnitt 4.1). Daraufhin wird der Prozess P_{merged} angelegt und die PBDs in Scopes transformiert. Als nächstes findet die Konsolidierung anhand definierter Konsolidierungsmuster statt. Alle `message links`, die keinem Muster entsprechen, werden in der NMML (Non-Mergeable-Message-Links) Liste gespeichert (siehe Abschnitt 2.5). Hierunter fielen ebenfalls der Fault Handler, Termination Handler und Compensation Handler (FCT-Handler), welche nun im letzten Schritt `MergePostProcessor` (siehe Abschnitt 4.2) verarbeitet werden.

4.1. MergePreProcessor & Invoke Umwandlung

Der `MergePreProcessor` führt notwendige Vorarbeiten durch. Hierunter fällt z. B. die Invoke Umwandlung (siehe Abschnitt 3.2). Bei der Umwandlung müssen die `<catch>` und `<catchAll>` Blöcke in den neuen Scope übertragen und der neue Scope um die Invoke Aktivität herum erzeugt werden. Für solche Transfervorgänge wäre der `PBDFragmentDuplicatior` zuständig, der

das Kopieren der Kommunikationsaktivitäten aus dem PBD in die PBD Scopes unterstützt. Bei diesem Vorgang werden alle Attribute, die nicht der BPEL Spezifikation entsprechen, entfernt. Deswegen würden bei der Übertragung notwendige Attribute, wie `wsu:id` (siehe Abschnitt 2.3.2), aus den kopierten Aktivitäten verloren gehen. Der `MergePreProcessor` wird vor der Konsolidierung ausgeführt. Würde dieser den `PBDFragmentDuplicatior` verwenden, wäre somit keine Verbindung mehr zwischen `message link` und der entsprechenden Aktivität im PBD möglich, da die `wsu:id` als Verbindungsstück durch den `PBDFragmentDuplicatior` entfernt worden wäre. Um nun trotzdem eine Übertragung der `<catch>` und `<catchAll>` Tags zu erreichen, wird die Klasse `EcoreUtil` vom Eclipse Modeling Framework (EMF), Version 2.8.3, anstatt des `PBDFragmentDuplicatior` verwendet. Diese Klasse lässt mittels der Methode `copy(object)` ein Deep Copy zu, indem alle Daten, Attribute und Kindelemente aus dem Fault Handler von der Invoke Aktivität in den Fault Handler des Scopes kopiert werden. Eine andere Möglichkeit wäre die direkte EMF-Objektzuweisung mit Referenzauflösung, welche durch EMF gegeben ist, gewesen. Da sich EMF an der BPEL Spezifikation orientiert, würde bei einer Zuweisung der `<catch>` und `<catchAll>` Blöcke ein Fault Handler ohne `<catch>` und `<catchAll>` Block entstehen. Dies widerspricht der BPEL Spezifikation und würde zu einem Fehler im EMF führen, da ein Fault Handler stets mindestens einen `<catch>` oder `<catchAll>` Block enthalten muss. Eine Zuweisung der Fault Handler Objekte ist ebenfalls nicht möglich, da die verwendeten Objekte auf EMF basieren und den oben genannten Fehler bei der Zuweisung verursachen. Somit muss ein Deep Copy mit der Klasse `EcoreUtil` für diesen Fall durchgeführt werden.

4.2. MergePostProcessor

Im `MergePostProcessor` finden Nacharbeiten statt. Hierzu gehört die Beseitigung der Grenzüberschreitungen von Links nach der Konsolidierung. Dazu wird der Prozess P_{merged} analysiert und abhängig von den einzelnen Handlern die Klassen `FaultHandlerUtil`, `TerminationHandlerUtil` und `CompensationHandlerUtil` ausgeführt. Vor dem Aufruf der Klassen werden alle existieren Scopes in der `FCTEUtil` Klasse auf verbotene Grenzüberschreitungen untersucht. Wird ein Scope S gefunden, bei dem dies zutrifft, wird ein Scope S_{sur} und ein Flow F_{sur} , wie in Abschnitt 3.1.8 beschrieben, angelegt. Darauffolgend wird die jeweilige `HandlerUtil` Klasse ausgeführt. In der `FaultHandlerUtil` Klasse werden die Scopes für die `<catch>` und `<catchAll>` Fällen angelegt und bei Bedarf die Variablen zugewiesen. Hierzu wurde zusätzlich die `ChoreographyPackage`, zuständig für das Einlesen und der Objekt-Abbildung der Choreographie Dateien, angepasst, damit einer PBD mehrere WSDLs zugeordnet werden können. Hierdurch kann z. B. ein Callback Client, für einen asynchronen Aufruf, der in einer weiteren WSDL definiert ist, einer PBD zugeordnet werden. Bei der `TerminationHandlerUtil` Klasse hingegen findet eine Zuweisung der Aktivitäten aus dem Termination Handler in den neu erzeugten Scope S_{TH} statt, da keine Unterscheidung zwischen `<catch>` und `<catchAll>` notwendig ist. In der `CompensationHandlerUtil` ist der Algorithmus aus Abschnitt 3.5.3 für PBD Scopes umgesetzt. Vom Lösungsansatz wurden die umgebenen Scopes und Flows, die Erzeugung der Variablen `isScopeCompleted`, die Zuweisung `isScopeCompleted=true` in jedem Scope, die Abfrage in den neuen Compensation Handler Scopes und die Auslagerung der

Kompensationsaktivitäten in die neuen Compensation Handler Scopes integriert. Das Setzen der Links von Fault Handler und Termination Handler zu den Kompensations-Flows, das Ersetzen der CompensateScope Aktivität, das Erzeugen der Kompensationsstruktur für jeden einzelnen Scope, die Kompensation von FCTE-Handlern und wiederholbaren Strukturen stehen noch aus.

4.3. Beziehungen der Komponenten

Die Abhängigkeiten der Klassen untereinander ist in dem Komponentendiagramm in Abbildung 4.2 dargestellt. Das Hauptprojekt ist `org.bpel4chor.mergeChoreography` mit der Startklasse `ChoreographyMerger`, die die Klassen `MergePreProcessor`, `ChoreographyPackage` und `MergePostProcessor` für die Steuerung der Konsolidierung nutzt. Die `ChoreographyPackage` wird für das Einlesen, Speichern und das Initialisieren des Prozesses P_{merged} genutzt. Beim Einlesen werden die BPEL4Chor Dateien in BPEL4Chor Objekten aus dem Projekt `org.bpel4chor.model` gespeichert, welches von Cui [Cui12] entwickelt wurde. Für die Konsolidierung muss durch die BPEL und BPEL4Chor Objekte traversiert werden, hierzu nutzt `ChoreographyPackage` Hilfsfunktionen aus dem Projekt `de.uni_stuttgart.iaas.bpel.model.utilities`. Außerdem werden für die Objektrepräsentation und Verarbeitung von BPEL Objekten die EMF Projekte `org.eclipse.bpel.model` und `org.eclipse.bpel.common.model` verwendet. Im Projekt `org.bpel4chor.mergeChoreography.test` findet der Test der Komponenten statt (siehe Abbildung 4.3). Hier befindet sich auch das `CheckMergeResult` Programm (siehe Abschnitt 4.5), welches für die Validierung von konsolidierten Geschäftsprozessen zuständig ist. Es verwendet die Schnittstellen von `org.bpel4chor.mergeChoreography` zur Erzeugung der Prozesse P_{merged}.

4.4. Konsolidierungsablauf

In Abbildung 4.4 wird anhand eines Sequenzdiagramms der Konsolidierungsablauf gezeigt. Über die Methode `merge(fileName)` in der Klasse `ChoreographyMerger` wird der Konsolidierungsvorgang gestartet. Im ersten Schritt finden Vorarbeiten über den Aufruf `startPreProcessing(choreographyPackage)` statt. In der Methode `mergeChoreography()` wird der zweite und dritte Schritt (Abbildung 4.1)ausgeführt, die Initialisierung des Prozesses P_{merged} sowie die Anwendung der Konsolidierungsoperationen und das Befüllen der NMML (Non-Mergeable-Message-Links) Liste. Für die Initialisierung ist die Methode `initMergedProcess` zuständig. In der Schleife werden alle `message links` durchlaufen und auf Konsolidierungsmuster überprüft und umgewandelt. Zum Schluss der Methode `mergeChoreography()` werden alle nicht verarbeiteten `message links` in der NMML Liste über die Methode `configureNMMLActivities` durchlaufen und die zugehörigen Kommunikationsaktivitäten mit technischen Informationen aus der Grounding Datei gefüllt sowie dem Prozess P_{merged} hinzugefügt, da eine Konsolidierung für diese Kommunikationsaktivitäten nicht möglich ist. Im nächsten Schritt

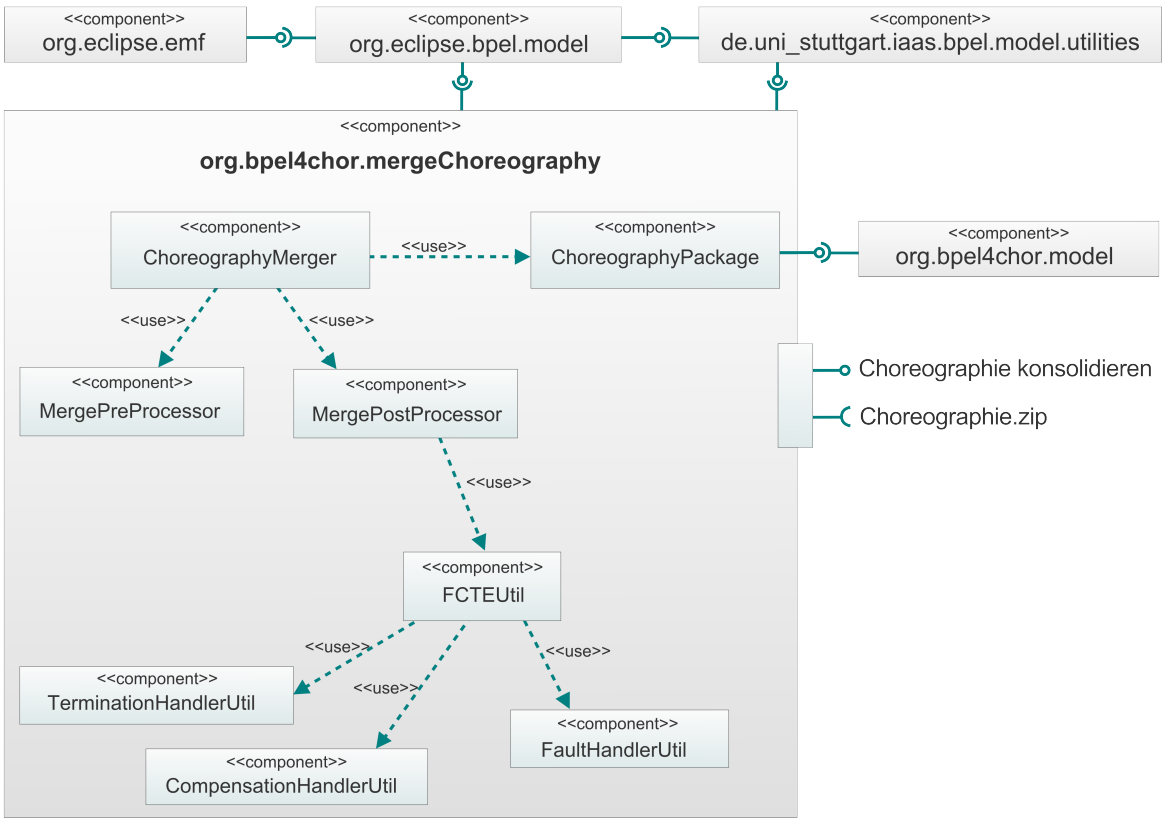


Abbildung 4.2.: Komponentendiagramm für die Konsolidierung einer Choreographie

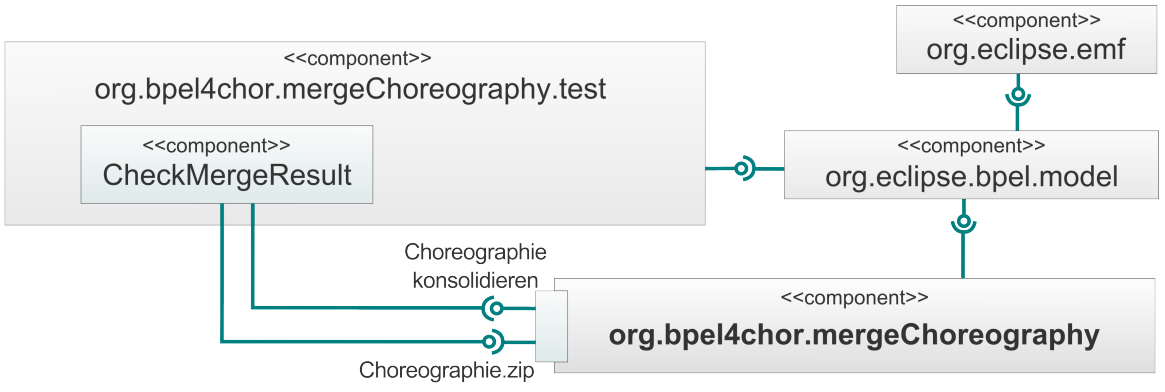


Abbildung 4.3.: Komponentendiagramm für die Überprüfung eines Konsolidierung

wird der `MergePostProcessor` über die Methode `startPostProcessing(choreographyPackage)` aufgerufen (Abbildung 4.5). Zum Schluss findet die Speicherung des neuen Prozesses sowie der zugehörigen WSDL Dateien statt (`saveMergedChoreography(fileName)`).

Durch die von Debicki [Deb13] zur Verfügung gestellte Mustererweiterung, ist es möglich, anhand der Kontrollflussstruktur und den Interaktionsszenarien bestimmte Konsolidierungsmuster auszuführen und neue Mustererkennungen zur Verfügung zu stellen. Nach Entfernen der FCTE-Handler aus der NMML Liste werden auch Konsolidierungsmuster in den FCTE-Handlern erkannt und umgewandelt, wodurch verbotene Grenzüberschreitungen entstehen. Um diese Grenzverletzungen aufzulösen, wurde der `MergePostProcessor` eingeführt. In Abbildung 4.5 ist ein Sequenzdiagramm für den `MergePostProcessor` abgebildet. Über die Methoden `processCompensationHandler(mergedProcess)` und `processScopesFT(mergedProcess)` wird in der Klasse `FCTEUtil` zum einen der Kompensationsalgorithmus und zum anderen die Fault Handler und Termination Handler Logik ausgeführt. Der Prototyp der Kompensationslogik generiert dabei für jeden PBD Scope eine Kompensationsreihenfolge anhand dem Algorithmus aus Abschnitt 3.5.3 über den Aufruf der Methode `processCompensation(Scope)`. Für den Fault Handler und Termination Handler hingegen werden alle Scopes eines Prozesses P_{merged} durchlaufen und die entsprechende Logik über `processFaultHandler(Scope)` und `processTerminationHandler(Scope)` aufgerufen.

4.5. Überprüfung der erzeugten Prozesse

Für die Überprüfung der erzeugten Prozesse P_{merged} wurde ein Programm *CheckMergeResult* entwickelt, welches als Eingabe einen erzeugten Prozess P_{merged} sowie ein abstrakten Prozess $P_{expected}$ erwartet. Der Prozess $P_{expected}$ wird zum Abgleich von Prozess P_{merged} benötigt. Es wird überprüft, ob der Prozess P_{merged} den strukturellen Aufbau von Prozess $P_{expected}$ entspricht und ob die Aktivitätsnamen übereinstimmen. Bei den Aktivitätsnamen besteht die Möglichkeit eine Überprüfung mittels Java Regex durchzuführen. Es ist zu beachten, dass bei der Definition von Prozess $P_{expected}$ keine der Spezifikation widersprechenden Konstrukte erzeugt werden dürfen. Hierunter fällt z. B. die Definition von zwei Aktivitäten in einem `<catch>` Block, da laut Spezifikation nur eine Aktivität in einem `<catch>` Block definiert werden darf.

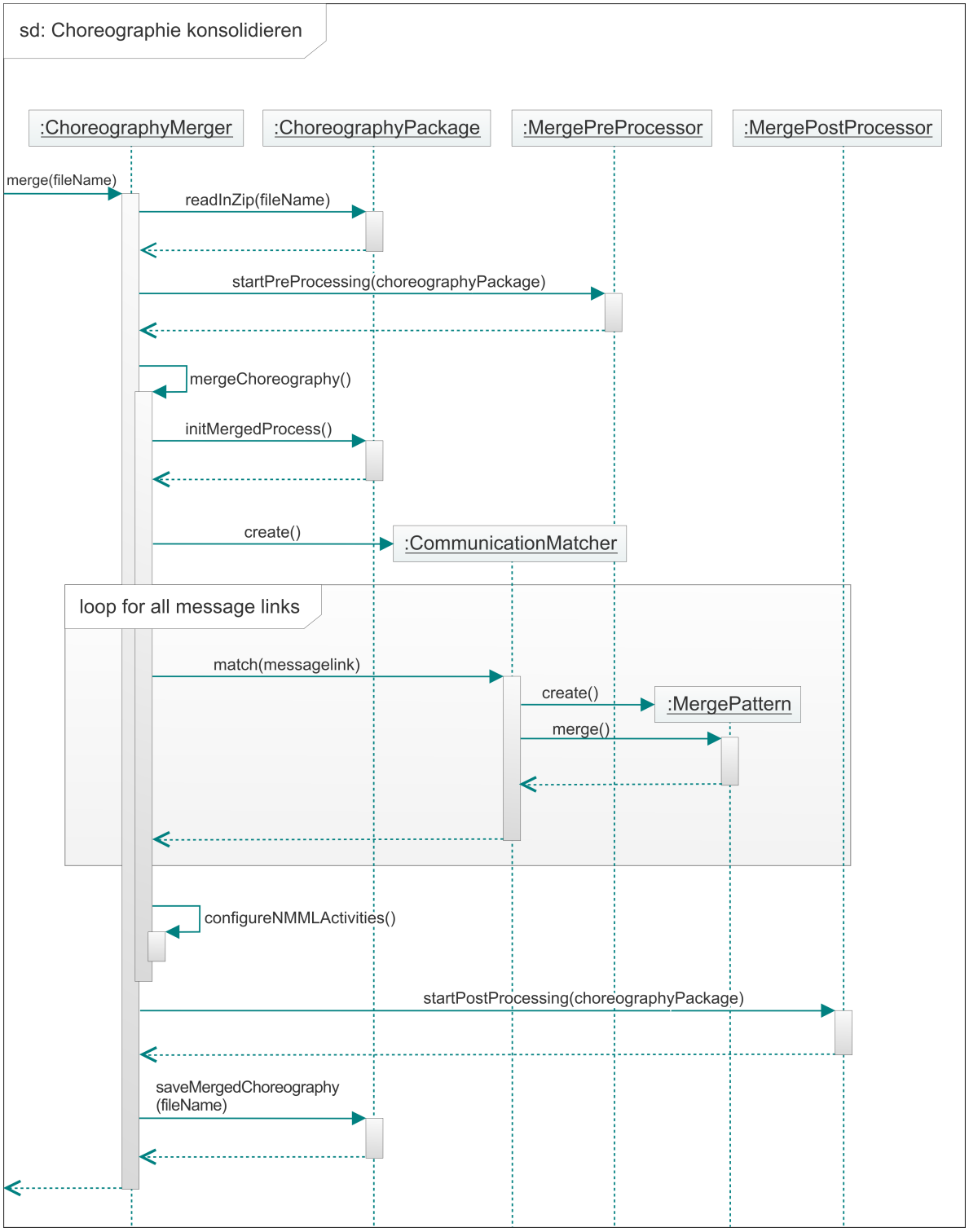


Abbildung 4.4.: Sequenzdiagramm zur Konsolidierung

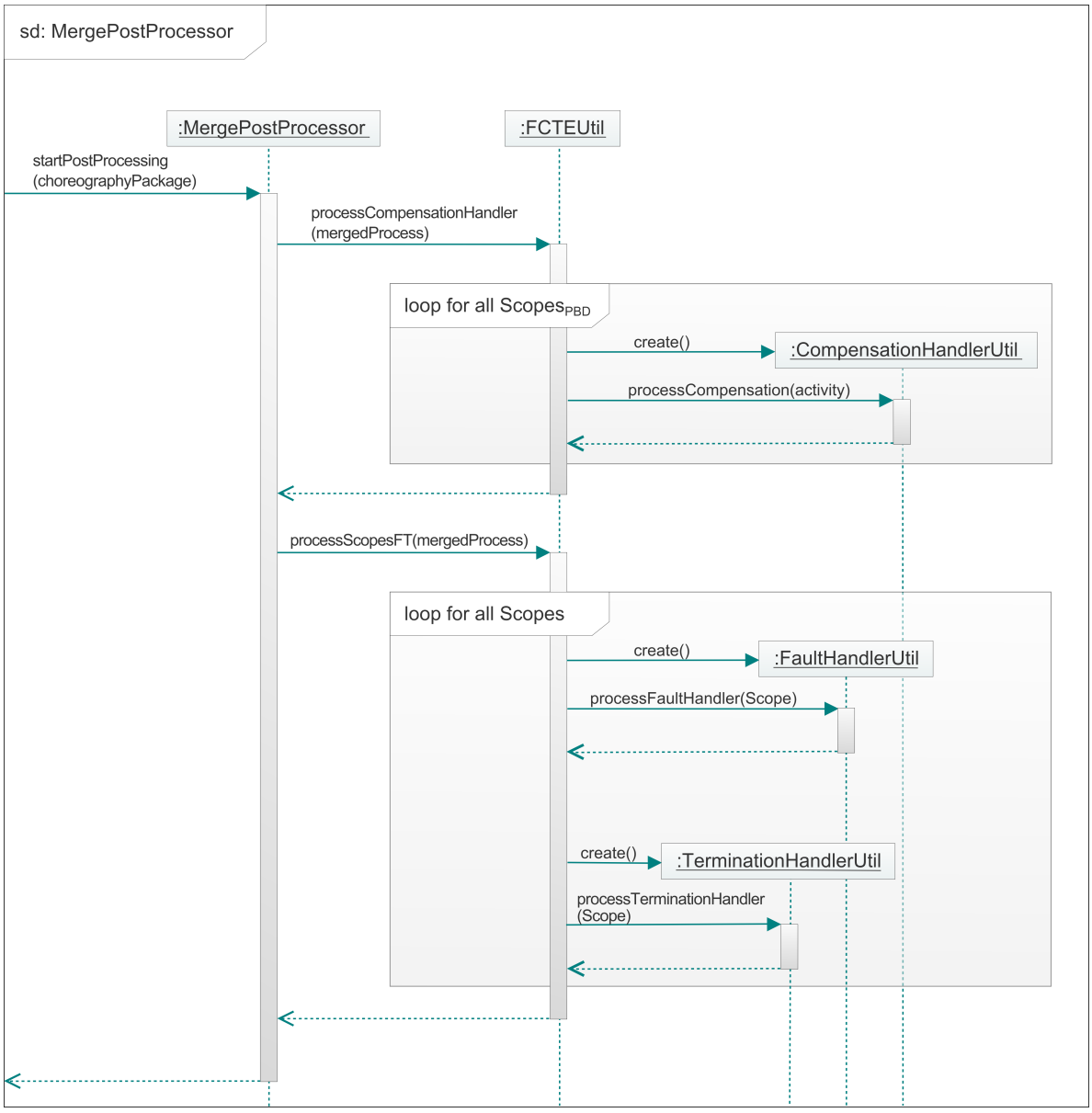


Abbildung 4.5.: Sequenzdiagramm zur Ausführung der FCT-Handler Implementierungen

5. Zusammenfassung und Ausblick

Das Ziel der Diplomarbeit war, die korrekte Verarbeitung der Konsolidierung einer Choreographie mit Fehlerbehandlung in einen Prototypen Debicki [Deb13] zu integrieren. Diese Zielsetzung wurde durch das Erstellen von Szenarien, welche die bestehenden Konsolidierungsmuster auf einen Fault Handler anwenden, und das Analysieren der Auswirkungen auf den Geschäftsprozess sowie durch das Implementieren eines Lösungsverfahrens erreicht. Darüber hinaus wurde die Konsolidierung des Termination Handlers umgesetzt und ein Lösungsansatz für den Event Handler und Compensation Handler entwickelt. Zusätzlich wurde ein Prototyp für die Umsetzung des Compensation Handlers entwickelt und in das bestehende System integriert.

In Kaptiel 3 fand die Analyse der Fault Handler, Termination Handler, Event Handler und Compensation Handler (FCTE-Handler) statt. Hierzu wurden unterschiedliche synchrone und asynchrone Szenarien in Bezug auf die FCTE-Handler entwickelt und ihre Auswirkung nach einer Konsolidierung überprüft. Die synchrone und asynchrone Kommunikation nach der Konsolidierung weist den gleichen Kontrollfluss wie die Kommunikationsaktivitäten auf, die nicht im FCTE-Handler aufgerufen werden (Anwendung des Allen Kalküls). Es wurden jedoch verbotene Grenzüberschreitungen von Links ermittelt, Tabelle 3.1, für deren Beseitigung speziell für den Fall des Fault Handler und Termination Handler eine Lösung sowie hinsichtlich des Compensation Handler ein Lösungsansatz entwickelt werden konnte. Die Lösung für den Fault Handler und Temination Handler wurde vollständig und der Lösungsansatz für den Compensation Handler teilweise (siehe Abschnitt 4.2) in das bestehende System integriert.

In Kaptiel 4 wurde die Umsetzung der Korrekturen des bestehenden Systems vorgestellt und die Beziehung der einzelnen Komponenten untereinander betrachtet. Es wurden der `MergePreProcessor` und der `MergePostProcessor` vorgestellt, die die Eingabedateien aufbereiten und die Ausgabedateien nachbearbeiten. Weiterhin wurde die Klasse `FCTEUtil` vorgestellt und gezeigt, dass sie für die Koordination der FCTE-Handler zuständig ist. Für die Überprüfung der erzeugten Prozesse wurde das Programm *CheckMergeResult* vorgestellt.

Ausblick

Beim Lösungsansatz für den Event Handler, werden die Prozesse, welche mit dem Event Handler kommunizieren, in den Event Handler integriert oder eine Kopie des Event Handler erstellt, der mit dem jeweiligen Prozess interagiert. Hierbei wurden die beiden Fälle betrachtet, bei denen zum einen der Event Handler aufgerufen wird und zum anderen selbst einen Prozess aufruft. Beide Fälle zeigen, dass durch synchrone Aufrufe sowie durch Aufruf mehrerer Instanzen eines Event Handlers unterschiedliche Lösungsansätze entstehen können. Somit wäre eine Analyse weiterer synchroner und asynchroner Kommunikationsmuster sowie die Kommunikation zwischen einem oder mehreren Event Handler interessant. Zudem erwies sich in dem Fall der Integration von Prozessen in den Event Handler, dass diese den umgebenden Scope des Event Handlers am Zustandswechsel hindern konnten. Hierfür muss noch eine Lösung gefunden werden.

Die Implementierung des Lösungsansatzes für den Compensation Handler ist aktuell noch ein Prototyp und muss entsprechend angepasst und erweitert werden (siehe Abschnitt 4.2). Bei einer Weiterentwicklung des Compensation Handlers wäre interessant, wie Snapshots [AAA+07] von Scopes sich auf die normale und alternative Kompensationsreihenfolge auswirken.

In BPEL und BPEL4Chor ist es möglich, dass ein Invoke, je nach Fall, mit mehreren Reply oder Receive Aktivitäten in Verbindung stehen kann. Das bestehende System jedoch geht momentan davon aus, dass für jeden `message link` nur eine aufrufende und eine empfangende Kommunikationsaktivität existiert. Zugehörige Definition in BPEL4Chor werden aktuell ignoriert.

Gegenwärtig ist das bestehende System auf One-to-One Interaktionen ausgerichtet. Dies bedeutet, jeder Prozess wird nur einmal instanziiert. Bei der Ausführung von One-to-Many Interaktionen führt dies zu Konflikten, da ein Prozess mehrere Instanzen eines anderen Prozesses aufrufen kann. Da nach der Konsolidierung ein Prozess durch ein Scope ersetzt wird, können diese Scopes nur noch einmal aufgerufen werden. Das System benötigt somit eine Erweiterung auf One-to-Many Prozessrelationen.

A. Anhang

In diesem Kapitel sind weitere Illustrationen sowie Listing's zu finden die dem Verständnis dienen.

A.1. Weitere Illustrationen

Abbildung A.1 ein von Debicki [Deb13] entwickeltes Kommunikationsmuster, dass infolge der Umstrukturierung des Invokes nicht mehr vom System aufgerufen wird und somit entfernt werden konnte.

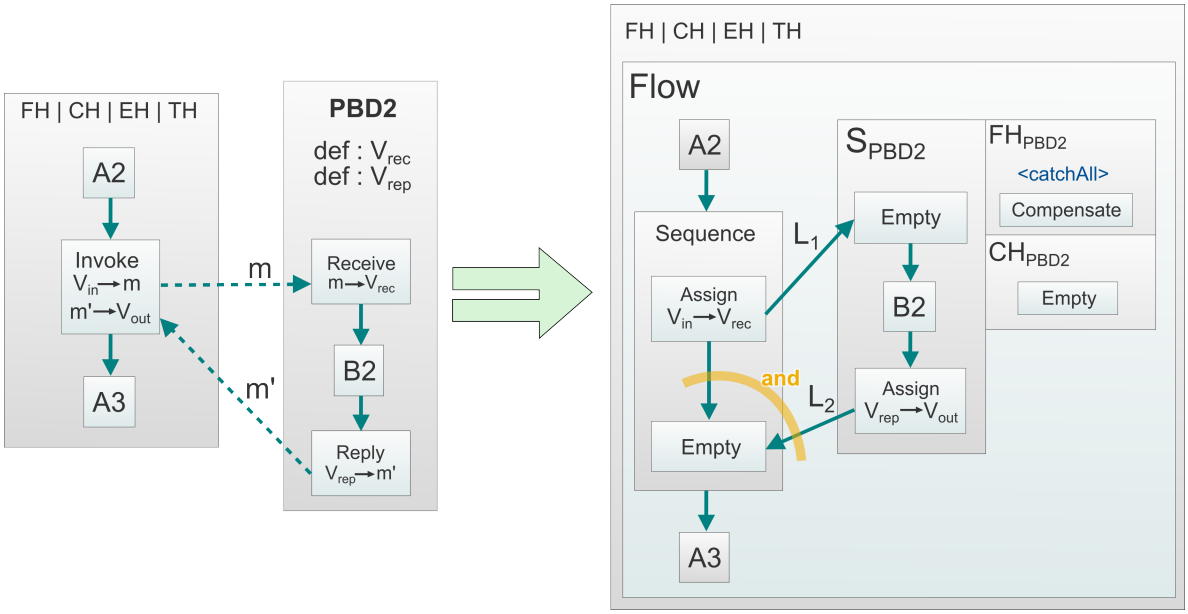


Abbildung A.1.: SyncPattern1.5 von Debicki [Deb13]

A.2. Weitere Listing's

Listing A.1 und Listing A.2 zeigen die Antwortnachrichten der Geschäfts- und Laufzeitfehler.

Listing A.1 SOAP-Envelope eines Geschäftsfehler

```
Fault response: faultName=MyServiceException
    faultType={http://webservice.iaas.uni_stuttgart.de}MyServiceException

<?xml version="1.0" encoding="UTF-8"?>
<message>
       <MyServiceException>
              <MyServiceException xmlns="http://webservice.iaas.uni_stuttgart.de"
                  xmlns:ns1="http://webservice.iaas.uni_stuttgart.de"
                  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
                     <faultInfo>
                            <faultCode>1234</faultCode>
                            <faultString>My Service Error</faultString>
                     </faultInfo>
              </MyServiceException>
       </MyServiceException>
</message>
```

Listing A.2 SOAP-Envelope eines Laufzeitfehler

```
Fault response: faultType=(unkown)

<?xml version='1.0' encoding='utf-8'?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
       <soapenv:Body>
              <soapenv:Fault>
                     <faultcode>soapenv:Server.userException</faultcode>
                     <faultstring>java.lang.RuntimeException</faultstring>
                     <detail>
                            <ns1:hostname
                                xmlns:ns1="http://xml.apache.org/axis/">virtualdipl</ns1:hostname>
                     </detail>
              </soapenv:Fault>
       </soapenv:Body>
</soapenv:Envelope>
```

Literaturverzeichnis

[AAA+07] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guizar, N. Kartha, C. Liu, R. Khalaf, D. Koenig, M. Marin, V. Mehta, S. Thatte, D. Rijn, P. Yendluri, A. Yiu. Web Services Business Process Execution Language Version 2.0 (OASIS Standard), 2007. URL WS-BPELTCOASIS,http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html. (Zitiert auf den Seiten 7, 8, 17, 18, 20, 21, 22, 23, 24, 25, 27, 50, 52 und 78)

[CCMW01] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. Web Services Description Language (WSDL) 1.1. World Wide Web Consortium, Note NOTE-wsdl-20010315, 2001. URL http://www.w3.org/TR/2001/NOTE-wsdl-20010315. (Zitiert auf Seite 16)

[Cui12] D. Cui. *Splitting BPEL Processes*. Diplomarbeit, Universität Stuttgart, Institut für Architektur von Anwendungssystemen, Deutschland, 2012. URL http://elib.uni-stuttgart.de/opus/volltexte/2012/7605/. (Zitiert auf den Seiten 31 und 71)

[DBKL08] G. Decker, A. P. Barros, F. M. Kraft, N. Lohmann. Non-desynchronizable Service Choreographies. In *ICSOC*, Band 5364 von *Lecture Notes in Computer Science*, S. 331–346. 2008. (Zitiert auf Seite 34)

[Deb13] P. Debicki. *Choreographie-basierte Konsolidierung von BPEL Prozessmodellen*. Diplomarbeit, Universität Stuttgart, Institut für Architektur von Anwendungssystemen, Deutschland, 2013. URL http://elib.uni-stuttgart.de/opus/volltexte/2013/8263/. (Zitiert auf den Seiten 8, 31, 50, 73, 77 und 79)

[Dec09] G. Decker. *Design and Analysis of Process Choregraphies*. Dissertation, Business Process Technology Group Hasso Plattner Institute, University of Potsdam Potsdam, Germany, 2009. (Zitiert auf den Seiten 7 und 28)

[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. doi:10.1109/ICWS.2007.59. (Zitiert auf Seite 28)

[Kha08] R. Y. Khalaf. *Supporting business process fragmentation while maintaining operational semantics: a BPEL perspective*. Dissertation, University of Stuttgart, 2008. URL http://elib.uni-stuttgart.de/opus/volltexte/2008/3514/. Http://d-nb.info/997164751. (Zitiert auf Seite 11)

[KRL09] R. Khalaf, D. Roller, F. Leymann. Revisiting the Behavior of Fault and Compensation Handlers in WS-BPEL. In *OTM '09: Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems*, S. 286–303. Springer-Verlag, 2009. doi: 10.1007/978-3-642-05148-7_20. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2009-140&engl=1. (Zitiert auf den Seiten 55 und 58)

[Ley07] F. Leymann. OASIS Webinar, 2007. URL https://www.oasis-open.org/committees/download.php/23070/. (Zitiert auf Seite 17)

[LKLR07] N. Lohmann, O. Kopp, F. Leymann, W. Reisig. Analyzing BPEL4Chor: Verification and Participant Synthesis. In *WS-FM*, Band 4937 von *Lecture Notes in Computer Science*, S. 46–60. Springer, 2007. (Zitiert auf Seite 34)

[Sta05a] C. Stahl. Einführung in BPEL Teil 2, 2005. URL http://www2.informatik.hu-berlin.de/top/lehre/WS05-06/taskforce/bpel-einfuehrung_2.pdf. (Zitiert auf Seite 21)

[Sta05b] C. Stahl. *A Petri net semantics for BPEL*. Professoren des Inst. für Informatik, 2005. (Zitiert auf Seite 18)

[VBP08] R. D. M. K. Virginia Beecher, Deanna Bradshaw, A. Prazma. *Developer's Guide for Oracle SOA Suite*. Oracle, 11g release 1 (11.1.1) Auflage, 2008. URL http://download.oracle.com/otndocs/products/soa/e10224.pdf. (Zitiert auf Seite 43)

[Wag13] S. Wagner. Choreography-based BPEL Process Consolidation. Elektronische Übergabe des Posters, Universität Stuttgart, Institut für Architektur von Anwendungssystemen, Deutschland, 2013. (Zitiert auf Seite 33)

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005. (Zitiert auf den Seiten 7 und 15)

[WKL11] S. Wagner, O. Kopp, F. Leymann. Towards Choreography-based Process Distribution In The Cloud. In *Proceedings of the 2011 IEEE International Conference on Cloud Computing and Intelligence Systems*, S. 490–494. IEEE Xplore, Bejing, China, 2011. doi:10.1109/CCIS.2011.6045116. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2011-58&engl=1. (Zitiert auf den Seiten 7, 11 und 13)

[WKL12] S. Wagner, O. Kopp, F. Leymann. Towards Verification of Process Merge Patterns with Allen's Interval Algebra. In *Proceedings of the 4th Central-European Workshop on Services and their Composition (ZEUS 2012)*, S. 1–8. CEUR Workshop Proceedings, Bamberg, 2012. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2012-10&engl=1. (Zitiert auf den Seiten 7, 8, 29, 30 und 34)

[WKL13] S. Wagner, O. Kopp, F. Leymann. Consolidation of Interacting BPEL Process Models with Fault Handlers. In *Proceedings of the 5th Central-European Workshop on Services and their Composition (ZEUS 2013)*, S. 1–7. CEUR Workshop Proceedings, Rostock, 2013. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2013-08&engl=1. (Zitiert auf den Seiten 34, 38 und 46)

[WSB] Web Services Business Activity Version 1.1. URL http://docs.oasis-open.org/ws-tx/wstx-wsba-1.1-spec/wstx-wsba-1.1-spec.html. (Zitiert auf Seite 18)

Alle URLs wurden zuletzt am 17. 11. 2013 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.

Ort, Datum, Unterschrift