Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2101

Generierung einer BPEL4Chor-Beschreibung aus BPEL-Prozessen

Thomas Steinmetz

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Oliver Kopp
begonnen am:	01. März 2007
beendet am:	31. August 2007
CR-Klassifikation:	H.4.1, K.1

INHALTSVERZEICHNIS

KAPITEL 1

EINLEITUNG

Ein neues Architekturkonzept ist die Service-orientierte Architektur (SOA). *Web Services* sind eine Möglichkeit, eine SOA zu realisieren. Die Idee von *Web Services* ist es, einzelne Funktionalitäten in Form von *Services* über ein Netzwerk zur Verfügung zu stellen. Mit diesen lassen sich Geschäftsprozesse beschreiben, etwa mittels WS-BPEL 2.0 (Web Service Business Process Execution Language 2.0), einer Modellierungssprache für Geschäftsprozesse. WS-BPEL 2.0 (kurz BPEL) bietet dabei die Möglichkeit zur Orchestrierung der *Web Services*. Eine Orchestrierung beschreibt die Geschäftsprozesslogik aus der Sicht eines teilnehmenden Prozesses. Dies beinhaltet sowohl den Nachrichtenaustausch mit anderen Prozessen, als auch interne Aktionen oder Dienste.

BPEL bietet nicht die nötigen Sprachkonstrukte, um eine Choreografie zu erstellen. Ziel einer Choreografie ist es, das Zusammenwirken von *Web Services* aus einer globalen Sicht zu beschreiben. Dazu gehören etwa der Datenfluss, der Kontrollfluss, Nachrichtenkorrelationen und Transaktionsbedingungen, nicht aber interne Aktionen, deren Auswirkungen von außen nicht erkennbar sind (vgl. [BDO05]). Ein neuer Ansatz zur Beschreibung von Choreografien, der auch in dieser Arbeit verwendet wird, ist BPEL4Chor (Business Process Execution Language for Choreographies). BPEL4Chor verwendet BPEL als Grundlage, und erweitert dieses um eine weitere Schicht, um damit eine Sprache zur Beschreibung von Choreografien zu erhalten (vgl. [DKLW07a]).

Obwohl BPEL-Prozesse Orchestrierungen beschreiben, können sie choreographisch zusammenarbeiten. Dies ist der Fall, wenn beispielsweise ein Partnerservice eines BPEL-Prozesses auch in BPEL realisiert ist. BPEL selbst bietet keine Möglichkeit, diese implizit definierte Choreographie explizit zu beschreiben. Das Ziel dieser Studienarbeit ist es, eine Möglichkeit aufzuzeigen, wie aus zusammenarbeitenden BPEL-Prozessen eine in BPEL4Chor beschriebene Choreographie abgeleitet werden kann. Zunächst werden Paare von miteinander kommunizierenden Aktivitäten gebildet. Ebenso wird die Beziehung bestimmt, die zu diesem Zeitpunkt zwischen zwei BPEL-Prozessen besteht. Im Anschluss werden die Teilnehmer generiert. Dabei sollen möglichst viele und möglichst wahrscheinliche Fälle abgedeckt werden.

Im nächsten Kapitel dieser Arbeit werden zunächst WS-BPEL 2.0 sowie BPEL4Chor näher vorgestellt, um einen kurzen Überblick über diese beiden Themenkomplexe zu erhalten. Im Anschluss wird in Kapitel 3 festgelegt, wie wir aus gegebenen BPEL-Prozessen eine BPEL4Chor Beschreibung generieren können. Die vorgestellte Transformation wird zum großen Teil als Prototyp implementiert. Ein Fazit und ein Ausblick in Kapitel 4 schließen diese Arbeit ab.

KAPITEL 2

GRUNDLAGEN

Zunächst werden wir uns WS-BPEL 2.0 näher ansehen, da dieses als Grundlage für BPEL4Chor dient. Hierzu betrachten wir die existierenden Aktivitäten, sowie die wichtigsten Konstrukte von WS-BPEL 2.0. Im Anschluss daran wird BPEL4Chor vorgestellt.

2.1. WS-BPEL 2.0

BPEL ist eine weit verbreitete Sprache und ein de-facto Standard zur Komposition von *Web Services*. Ein BPEL-Prozess besitzt, wie andere *Web Services* auch, eine WSDL-Schnittstelle, mit der das Verhalten des Prozesses nach außen hin offen gelegt wird. Somit fungiert ein BPEL-Prozess ebenfalls als *Web Service* und kann von anderen Prozessen als solcher eingebunden werden (vgl. [JE07]).

Dieses Unterkapitel bezieht sich hauptsächlich auf [JE07], andere Quellen werden gesondert angegeben.

2.1.1. Aktivitäten

Die einzelnen Arbeitsabläufe eines BPEL-Prozesses sind in Aktivitäten unterteilt. Um einen Überblick über BPEL zu erhalten, betrachten wir zunächst diese Aktivitäten.

Die verschiedenen Aktivitäten lassen sich in Basisaktivitäten und in strukturierte Aktivitäten unterteilen. Basisaktivitäten sind dabei atomar, d. h. bestehen nicht aus weiteren Aktivitäten, während sich strukturierte Aktivitäten aus anderen Aktivitäten zusammensetzen, um diese etwa sequentiell oder parallel auszuführen.

Basisaktivitäten

BPEL besitzt folgende Basisaktivitäten:

- <receive>
  Die <receive>-Aktivität erlaubt es einem Prozess auf eine passende Nachricht zu warten. Die <receive>-Aktivität endet, wenn eine solche Nachricht ankommt.

- <reply>
  Mittels der <reply>-Aktivität kann ein Prozess eine Antwort auf eine Nachricht, die etwa mittels der <receive>-Aktivität empfangen wurde, senden. Mit der Kombination der <receive>- und <reply>-Aktivität kann ein BPEL-Prozess anderen Prozessen Operationen anbieten. Die <reply>-Aktivität ist dabei optional. Ohne diese Aktivität bietet der Prozess lediglich eine *One-Way-Operation* an. Mittels eines optionalen „messageExchange"-Attributs kann eine <reply>-Aktivität mit einer <receive>-Aktivität assoziiert werden.

- <invoke>
  Mittels der <invoke>-Aktivität kann ein BPEL-Prozess eine von einem anderen Prozess angebotene Operation einbinden. Die Operation kann sowohl eine asynchrone *One-Way-Operation* sein, als auch eine synchrone *Request-Response-Operation*, bei der der eingebundene Prozess mittels einer <reply>-Aktivität antwortet. Im Falle einer *Request-Response-Operation* endet die <invoke>-Aktivität mit dem Empfang der Antwort.

- <assign>
  Mit der <assign>-Aktivität lassen sich Variablen neue Werte zuweisen. Die Variablen dienen dem Nachrichtenaustausch zwischen Prozessen (etwa als Übergabeparameter) und dem Speichern von Daten. Variablen können in BPEL WSDL-Nachrichtentypen, XML-Schematypen oder auch XML-Schemaelemente sein.

- <validate>
  Mit dieser Aktivität lässt sich testen, ob eine Variable bezüglich ihrer Definition mit einem zulässigen Wert belegt ist.

- <throw>
  Mit der <throw>-Aktivität wird eine Ausnahme innerhalb eines BPEL-Prozesses geworfen.

- <rethrow>
  Mit dieser Aktivität kann eine mit <throw> geworfene Ausnahme an einen *Fault Handler* eines äußereren Gültigkeitsbereichs weitergereicht werden.

- <exit>
  Mit dieser Aktivität kann eine Instanz eines BPEL-Prozesses beendet werden.

- <wait>
  Die <wait>-Aktivität erlaubt es einem Prozess, eine bestimme Zeitperiode oder bis zu einem bestimmten Zeitpunkt zu warten.

- <empty>
  Die <empty>-Aktivität ist eine Aktivität, die nichts tut. Dies wird etwa zur Synchronisation von nebenläufigen Aktivitäten benötigt.

- <compensateScope>
  Die <compensateScope>-Aktivität setzt alle Daten eines bestimmten inneren Scopes zurück, wenn dieser bereits erfolgreich ausgeführt und beendet wurde und wenn im äußeren Scope anschließend ein Fehler auftritt.

- <compensate>
  Die <compensate>-Aktivität macht dies für alle bereits erfolgreich ausgeführten inneren Scopes, in einer bestimmten Reihenfolge.

Die beiden Aktivitäten <compensateScope> und <compensate> dürfen nur innerhalb eines *Fault Handlers*, *Compensation Handlers* oder eines *Termination Handlers* genutzt werden. Die Handler selbst werden im Abschnitt 2.1.3 näher vorgestellt.

Strukturierte Aktivitäten

BPEL besitzt folgende strukturierte Aktivitäten:

- <sequence>
  Mittels der <sequence>-Aktivität wird eine Menge von Aktivitäten zusammengefasst, die sequentiell abgearbeitet werden, und zwar in der Reihenfolge, in der sie aufgeführt werden.

- <if>
  Die <if>-Aktivität erlaubt es, aus einer Menge von möglichen Aktivitäten genau eine auszuwählen. Mittels einer Bedingung (<condition>) wird jeweils die Bedingung für die Auswahl einer Aktivität festgelegt.

- <while>
  Mit der <while>-Aktivität wird eine Schleife definiert, d. h. die darin eingeschlossene Aktivität wird immer wieder ausgeführt, solange die Schleifenbedingung wahr ist. Diese Bedingung wird vor jedem Ausführen der inneren Aktivität überprüft.

- <repeatUntil>
  Mit der <repeatUntil>-Aktivität wird, wie bei der <while>-Aktivität, eine Schleife definiert. Nach der Ausführung der sich in der Schleife befindenden Aktivität wird geprüft, ob die Schleifenbedingung (<condition>) wahr ist. Die Schleife wird solange wiederholt, bis die Bedingung wahr wird. Im Unterschied zur <while>-Aktivität wird diese innere Aktivität auf jeden Fall einmal ausgeführt.

- <forEach>
  Mittels der <forEach>-Aktivität kann eine eingeschlossene <scope>-Aktivität mehrmals ausgeführt werden, wobei mit Hilfe von <startCounterValue> und <finalCounterValue> genau festgelegt ist, wie oft die Schleife durchlaufen werden kann. Mit einer *completion-Condition* kann angegeben werden, ob es eine Bedingung gibt, anhand derer man bereits vor der Gesamtanzahl der Durchläufe der Schleife diese vorzeitig verlassen kann.

  Die einzelnen Schleifendurchläufe können sowohl sequentiell als auch parallel ausgeführt werden.

- <pick>
  Die <pick>-Aktivität wird genutzt, um auf eine von vielen möglichen Nachrichten (<onMessage>) oder auf einen *TimeOut* (<onAlarm>) zu warten. Tritt solch ein Fall ein, wird die entsprechende Kind-Aktivität ausgeführt. Die <pick>-Aktivität endet, wenn diese ausgewählte Kind-Aktivität endet.

  Wie bei der <receive>-Aktivität kann einem <onMessage>-Ereignis eine <reply>-Aktivität zugeordnet werden. Über das optionale „messageExchange“-Attribut kann eine <reply>-Aktivität mit einem <onMessage>-Ereignis assoziiert werden.

  Jede <pick>-Aktivität muss mindestens ein <onMessage>-Ereignis enthalten.

- <flow>
  Die <flow>-Aktivität spezifiziert eine oder mehrere Aktivitäten, die nebenläufig ausgeführt werden sollen. Es können mittels des <links>-Konstrukts Links definiert werden, die die Abhängigkeiten zwischen einzelnen Kindaktivitäten festlegen. Durch Angabe der <source>-Elemente bei Aktivitäten kann angegeben werden, für welche Links diese Aktivität die Quelle ist. Dabei gilt ein Link als gesetzt, wenn die Aktivität, die die Quelle dieses Links ist, beendet wurde. Mit der Angabe einer *transitionCondition* kann eine Bedingung angegeben werden, die erfüllt sein muss, damit der Link auf den Wert „true“ gesetzt wird. Mittels der <target>-Elemente bei Aktivitäten kann angegeben werden, welche Links gesetzt sein müssen, damit die Aktivität begonnen werden kann. Dabei genügt es, dass einer der mittels <target>-Element definierten eingehenden Links auf den Status „true“ gesetzt ist, es sei denn, es ist eine *joinCondition* angegeben und damit ein boolscher Ausdruck, der über eine Menge von Links definiert ist. Die Aktivität kann nur begonnen werden, wenn all die Links so gesetzt sind, dass der boolsche Ausdruck wahr wird.

  Damit eine Aktivität begonnen werden kann, müssen alle eingehenden Links gesetzt sein, d. h. der Status eines jeden eingehenden Links muss bekannt sein.

- <scope>
  Die <scope>-Aktivität kann genutzt werden, um eingebettete Aktivitäten mit eigenen Variablen, *Partner Links*, Korrelationsmengen, Handlern und eigenen „messageExchange“-Attributen zu definieren, die in diesem Bereich gültig sind. Der äußerste Gültigkeitsbereich ist dabei der BPEL-Prozess selbst.

2.1.2. Partner Links

Wie bereits erwähnt, wird mittels WSDL die Schnittstelle eines *Web Service* beschrieben. Ein *Port Type* definiert dabei die Menge von zusammengehörenden Operationen eines Dienstes. Um definieren zu können, welche Verknüpfungen zwischen interagierenden BPEL-Prozessen bestehen, wird ein flexibles Instrument benötigt: die *Partner Links*.

Zunächst werden bereits in WSDL die Partner-Link-Typen (*Partner Link Types*) definiert. Jeder Typ besteht aus einer oder zwei Rollen, die die beiden interagierenden Prozesse einnehmen können, die wiederum jeweils genau einem *Port Type* zugeordnet werden. Es kann manchmal sinnvoll sein, dass nur eine Rolle angegeben ist. In einem solchen Fall empfängt ein BPEL-Prozess von anderen Prozessen etwas, ohne selbst etwas an den anderen Prozess zu senden. In BPEL selbst werden die *Partner Links* definiert. Ein *Partner Link* besteht aus einem Namen, der diesen *Partner Link* identifiziert, einem Partner-Link-Typ sowie den beiden Rollen (oder auch nur der einen Rolle). Die eigene Rolle wird durch das Attribut „myRole“ angegeben, die Rolle des Kommunikationspartners durch das „partnerRole“-Attribut. Bei dem Partnerprozess, falls dieser überhaupt existiert, wird ebenfalls ein *Partner Link* definiert, der auf den selben Partner-Link-Typ verweist und dessen Rollenbezeichnungen vertauscht sind. Es ist nun also genau definiert, welche Operation ein BPEL-Prozess über einen *Partner Link* beim jeweiligen Partnerprozess aufrufen kann.

Der *Partner Link* wird, eventuell zusammen mit dem optional anzugebenden *Port Type*, bei Aktivitäten angegeben, die der Kommunikation von Prozessen dienen, also <invoke>, <receive>, <reply>, im <onMessage>-Zweig einer <pick>-Aktivität, sowie im <onEvent>-Zweig des *Event Handlers*.

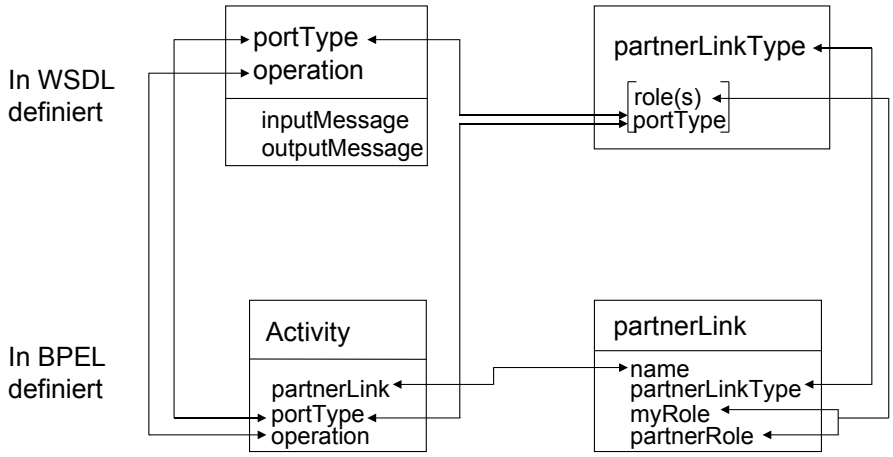

Abbildung 2.1.: *Partner Links* (vgl. [Wic06])

Allerdings kann unter Umständen ein gegenwärtiger, konkreter Partner erst dynamisch innerhalb eines Prozesses bestimmt werden. BPEL erweitert hierzu das Konzept der *Partner Links* um Endpunktreferenzen. Die Daten, die benötigt werden, um einen Endpunkt zu beschreiben, werden mittels *Service Reference Container* (sref:service-ref) gespeichert und können an andere Prozesse weitergereicht werden, die damit mit diesem konkreten Partner kommunizieren können.

2.1.3. Handler

Für Prozesse und bestimmte Aktivitäten können verschiedene Handler definiert werden, die wir hier kurz betrachten.

Event Handler

Für jeden BPEL-Prozess (und jede <scope>-Aktivität) können optional *Event Handler* definiert werden. Ein *Event Handler* besteht aus beliebig vielen <onEvent>- und <onAlarm>-Zweigen. Dabei muss gelten, dass mindestens ein <onAlarm>- oder <onEvent>-Zweig auftritt. Die Kindaktivität in einem <onEvent>- oder <onAlarm>-Element des *Event Handlers* muss dabei eine <scope>-Aktivität sein, die ausgeführt wird, wenn ein bestimmtes Ereignis auftritt (das Eintreffen einer Nachricht bei <onEvent> oder der Auftritt eines *TimeOuts* bei <onAlarm>).

Das zu <receive> bzw. zum <onMessage>-Ereignis der <pick>-Aktivität Gesagte lässt sich im Wesentlichen auf das <onEvent>-Ereignis übertragen. Mittels <messageExchange> kann eine <reply>-Aktivität mit einem <onEvent>-Ereignis assoziiert werden. Auch das <onAlarm>-Ereignis ähnelt im wesentlichen dem <onAlarm>-Ereignis der <pick>-Aktivität. Der Unterschied ist hier das optionale <repeatEvery>-Element, mit dem jedesmal nach Ablauf einer gewissen Zeitspanne das <onAlarm>-Ereignis ausgeführt wird.

Fault Handler

Für jeden BPEL-Prozess und jede <scope>-Aktivität kann ein *Fault Handler* definiert werden. Mit den *Fault Handlern* können Ausnahmen, die etwa mit der <throw>-Aktivität geworfen werden, mittels einer <catch>-Anweisung abgefangen und anschließend abgearbeitet werden.

Compensation Handler

Für <scope>-Aktivitäten oder innerhalb einer <invoke>-Aktivität können *Compensation Handler* definiert werden. Ein *Compensation Handler* beinhaltet Aktivitäten, die ausgeführt werden sollen, wenn es zu einer Kompensation (Compensation) kommt, d. h. wenn etwa ein transaktionsbedingter Rollback durchgeführt wird (vgl. [Bac06]).

Termination Handler

Innerhalb einer <scope>-Aktivität kann ein *Termination Handler* definiert werden. Damit erhalten <scope>-Aktivitäten die Möglichkeit, auf eine erzwungene Terminierung (etwa mittels der <exit>-Aktivität) Einfluss zu nehmen. Nach der erzwungenen Beendigung aller Aktivitäten des Scopes wird der *Termination Handler* ausgeführt.

2.1.4. Korrelationsmengen

Eine Nachricht muss nicht nur an den richtigen *Port Type* gesendet werden, sondern auch an die richtige Instanz eines BPEL-Prozesses, da zu einem Zeitpunkt mehrere Instanzen eines BPEL-Prozesses existieren können. Diese Aufgabe können bei BPEL die Korrelationsmengen (*Correlation Sets*) übernehmen. Eine Korrelationsmenge kann innerhalb eines Prozesses oder innerhalb einer <scope>-Aktivität definiert werden. Der Name einer Korrelationsmenge muss dabei eindeutig innerhalb seines Gültigkeitsbereiches sein.

Innerhalb der Lebenszeit einer Prozessinstanz kann eine Korrelationsmenge höchstens einmal instantiiert werden. Eine Korrelationsmenge wird mit einer Nachricht instantiiert und behält dann diese Werte. Diese Werte identifizieren die Prozessinstanz.

In einem zusammengehörenden Nachrichtenaustausch fungiert ein teilnehmender Prozess entweder als Initiator des Nachrichtenaustausches oder als nachfolgender Teilnehmer. Der Initiator sendet die erste Nachricht und legt somit die Werte der Korrelationsmenge fest, mit denen der zusammengehörende Nachrichtenaustausch markiert wird, wobei mittels der WSDL-Beschreibung eines *Web Services* festgelegt wird, welche Werte angenommen werden können. Alle nachfolgenden Teilnehmer erhalten durch diese instantiierende Nachricht die Werte für die Korrelationsmenge und können somit dieselbe Korrelationsmenge erzeugen.

Mit dem <correlation>-Konstrukt können Aktivitäten, die der Kommunikation dienen, also <invoke>, <receive>, <reply>, die <onMessage>-Zweige der <pick>-Aktivität sowie die <onEvent>-Zweige des *Event Handlers*, einer Prozessinstanz zugeordnet werden. Bei der Verwendung der definierten Korrelationsmengen in der <invoke>-Aktivität muss noch festgelegt werden, wann welche Korrelationsmenge genutzt wird, wenn die eingebundene Operation eine *Request-Response-Operation* ist. Dies geschieht über das „pattern“-Attribut. Dieses kann die Werte „request“, „response“ und „request-response“ annehmen. Gilt etwa „pattern = request“,

so wird die Korrelationsmenge nur bei den ausgehenden Nachrichten benutzt. Somit können für eingehende und ausgehende Nachrichten zwei verschiedene Korrelationsmengen genutzt werden.

2.1.5. Abstrakte Prozesse

Da wir später den Fall berücksichtigen, dass ein Prozess abstrakt sein kann, betrachten wir kurz, was für abstrakte Prozesse gelten muss.

Abstrakte Prozesse bieten eine abstrakte Sicht auf die Geschäftsprozesse, während ausführbare Prozesse, die wir mit der bisher beschriebenen BPEL-Syntax modellieren können, der konkreten Implementation von Geschäftsprozessen dienen. Ein abstrakter Prozess beschreibt dabei eine Klasse von BPEL-Prozessen. Er baut auf die beiden Teile *Common Base* und *Profile* auf.

Die *Common Base* legt die syntaktische Basis fest, der alle abstrakten Prozesse entsprechen müssen. Die syntaktischen Charakteristika der *Common Base* sind folgende:

- das *abstractProcessProfile* muss existieren
- alle Konstrukte der ausführbaren Prozesse sind erlaubt
- das „createInstance“-Attribut muss nicht spezifiziert sein
- es existieren zusätzliche Konstrukte zum expliziten bzw. impliziten Verbergen von Informationen

Das *abstractProcessProfile* enthält dabei die URI, die das Profil eines abstrakten Prozesses identifiziert.

Mit Hilfe der *Opaque Language Extension* lassen sich Informationen explizit verbergen. Es existieren vier verschiedene Platzhalter: Platzhalter für Aktivitäten (*Opaque Activities*), Platzhalter für Ausdrücke der Entscheidungslogik (*Opaque Expressions*), Platzhalter für Zuweisungen versteckter Werte (*Opaque Assignments*) und Platzhalter für Attribute (*Opaque Attributes*).

Implizites Verbergen geschieht über das Weglassen von Elementen. So können Elemente weggelassen werden, die syntaktisch erforderlich wären. Somit sind in diesem Fall das Weglassen des Elements und das Belegen des Elements mit einem Standardwert äquivalent.

Ein abstrakter Prozess ist nur dann gültig, wenn es mindestens eine *Basisvervollständigung* (*Basic Executable Completion*) gibt. Bei einer Basisvervollständigung werden unter anderem die Platzhalter durch ausführbare Elemente ersetzt.

In einem Profil wird unter anderem festgelegt, welche Opaque-Platzhalter überhaupt zulässig sind und wie der abstrakte Prozess zu einem ausführbaren Prozess erweitert werden kann. Die WS-BPEL 2.0 Spezifikation enthält bereits die beiden Profile *Abstract Process Profile for Observable Behavior* und *Abstract Process Profile for Templates*.

Im *Abstract Process Profile for Observable Behavior*, das der Modellierung des Interaktionsverhaltens eines Prozesses dient, gilt etwa, dass z. B. Elemente, die der Datenmanipulation dienen, verborgen sein dürfen, nicht aber *Partner Links*.

2.2. BPEL4Chor

Dieses Unterkapitel basiert auf den Quellen [DKLW07a] und [DKLW07b].

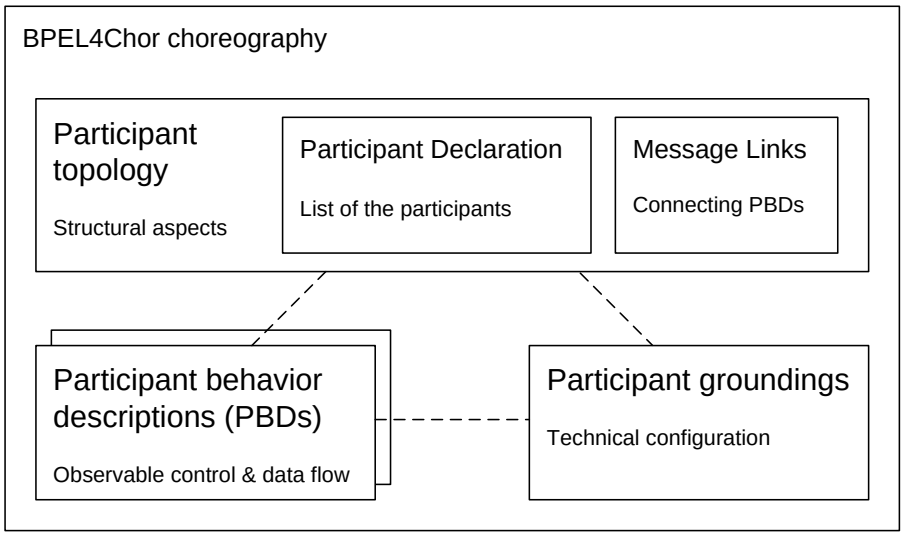

Abbildung 2.2.: Artefakte von BPEL4Chor (vgl. [DKLW07a])

Wie bereits erwähnt, eignet sich BPEL dafür, Orchestrierungen zu erstellen. Manchmal kann es auch sinnvoll sein, eine Choreografie zu erstellen, z. B. wenn mehrere Dienste verknüpft werden sollen oder eine globale Sicht auf die Teilnehmer gewünscht wird, wofür BPEL alleine nicht ausreicht. Hier kommt BPEL4Chor ins Spiel.

BPEL4Chor besteht aus drei verschiedenen Bestandteilen (siehe Abbildung 2.2), auf die in den folgenden Abschnitten näher eingegangen wird:

- *Participant Behavior Descriptions*
  definieren die Kontrollflussabhängigkeiten.

- *Participant Topology*
  legt die strukturellen Aspekte einer Choreografie fest, indem Teilnehmer (*Participants*) und Teilnehmertypen (*Participant Types*), Teilnehmerreferenzen (*Participant References*) sowie *Message Links* spezifiziert werden. Teilnehmer desselben Typs müssen die gleichen, der Kommunikation dienenden Aktivitäten anbieten, d. h. diese Teilnehmer folgen derselben *Participant Behavior Description*. Diese Aktivitäten von verschiedenen Teilnehmern werden über *Message Links* miteinander verknüpft.

- *Participant Groundings*
  enthalten die tatsächliche technische Konfiguration einer Choreografie. Erst hier wird die Verknüpfung mit den WSDL-Definitionen verwirklicht.

Diese Entkopplung der Aktivitäten, ihrer Abhängigkeiten und ihrer Verknüpfungen von den technischen Details, wie etwa den *Port Types*, erlaubt eine höhere Wiederverwendbarkeit eines Choreografie-Modells. Dies stellt einen großen Vorteil etwa gegenüber WS-CDL dar, das eng an WSDL gekoppelt ist.

2.2.1. Beispiel

Ein Beispiel für eine Choreografie findet sich in Abbildung 2.3.

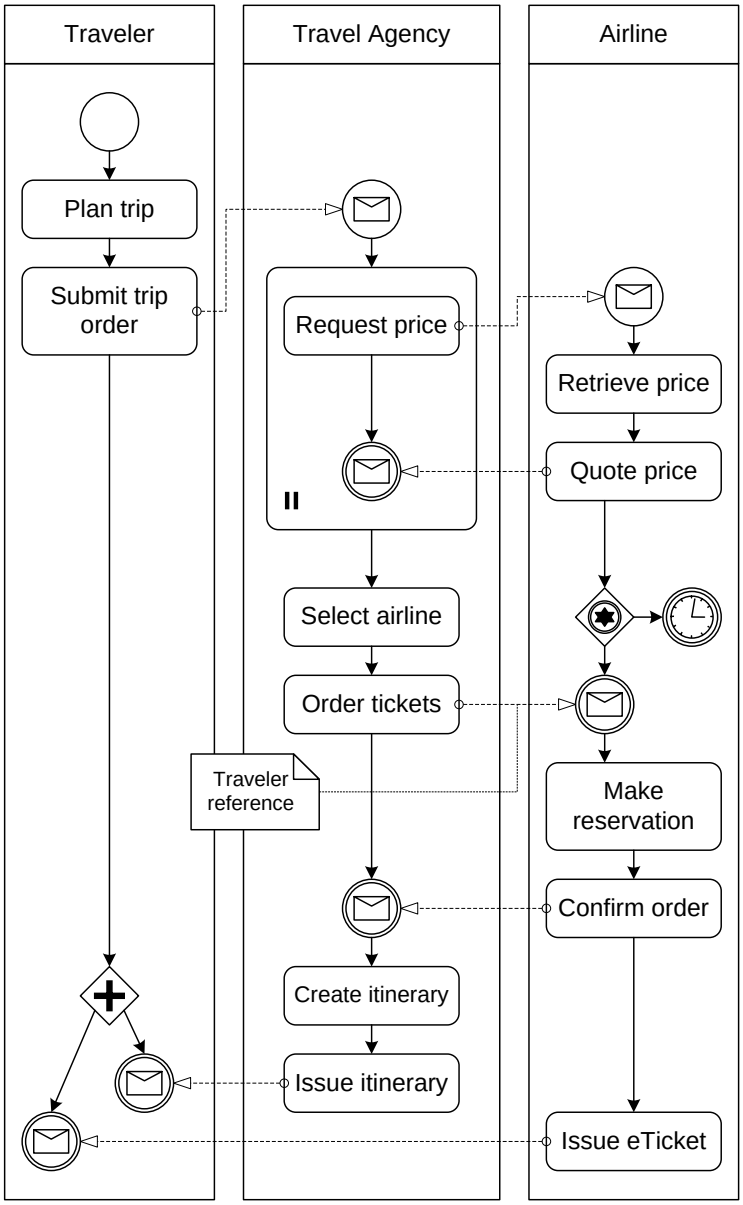

Abbildung 2.3.: Beispiel einer Choreografie [DKLW07a]

Ein Kunde (Traveler) will einen Flug buchen und kontaktiert hierzu ein Reisebüro (Travel Agency). Das Reisebüro kontaktiert verschiedene Fluglinien (Airline), um den Preis der jeweiligen Fluglinie für die vom Kunden gewünschte Route und das gewünschte Datum zu erfahren. Hat jede Fluglinie geantwortet, so wählt das Reisebüro die Fluglinie, die den günstigsten Preis anbietet, d. h. bestellt dort das Ticket und teilt der Fluglinie die EMail-Adresse des Kunden mit. Alle anderen Fluglinien, die eine Anfrage erhalten haben, aber nicht ausgewählt wurden, hören nach einer gewissen Zeit auf, auf eine Bestellung zu warten. Sobald die ausgewählte Fluglinie die Buchung bestätigt hat, sendet das Reisebüro dem Kunden einen Reiseplan und die Fluglinie dem Kunden das Ticket in elektronischer Form.

2.2.2. Participant Behavior Descriptions

Wie in obiger Definition bereits geschrieben, definieren die *Participant Behavior Descriptions* die Kontrollflussabhängigkeiten zwischen Aktivitäten, die der Kommunikation dienen. BPEL enthält bereits eine Menge an Konstrukten für den Kontrollfluss sowie zur Manipulation von Daten. Diese werden unverändert bei BPEL4Chor weiterverwendet.

Eine *Participant Behavior Description* entspricht im Wesentlichen einem abstrakten Prozess (siehe Abschnitt 2.1.5). Das verwendete Profil trägt den Namen *Abstract Process Profile for Participant Behavior Descriptions* und ist im Grunde eine Variante des *Abstract Process Profile for Observable Behavior*. Die Forderungen sind, dass jede der Kommunikation dienende Aktivität innerhalb des Namensraumes des Prozesses einen eindeutigen Namen hat und dass das Belegen der „partnerLink“-, „portType“- und „operation“-Attribute in solchen Aktivitäten verboten ist. Die Namen dieser Aktivitäten dienen später der Verknüpfung der Aktivitäten interagierender Teilnehmer. Erlaubt ist dagegen das Verwenden von Platzhaltern für Aktivitäten, bei denen nicht genau definiert ist, was sie tun, die aber etwa bei der Dokumentation hilfreich sein können. Da <onMessage>-Zweige der <pick>-Aktivitäten, ebenso wie <onEvent>-Zweige der *Event Handler*, nicht über „name“-Attribute verfügen, wird das neue Attribut „wsu:id“ eingeführt, das in den *Participant Behavior Descriptions* bei allen Konstrukten, die der Kommunikation dienen, Verwendung findet. Der Wert des „wsu:id“-Atrributs einer Aktivität dient uns als identifizierender Bezeichner dieser Aktivität.

Anders als bei ausführbarem BPEL muss bei BPEL4Chor nicht für jeden Prozess dessen Instantiierung festgelegt werden. Bei BPEL geschieht dies durch das „createInstance“-Attribut bei empfangenden Aktivitäten. Ist dieses auf „yes“ gesetzt, so wird eine neue Instanz generiert. Das heißt, es ist bei BPEL4Chor nicht unbedingt festgelegt, wann eine Instanz eines Prozesses erzeugt wird. Verzweigende Bedingungen müssen nicht spezifiziert werden, sondern können natürlichsprachlich angegeben werden. Die Bedingungen müssen erst später genau angegeben werden.

Korrelationsmengen (siehe Abschnitt 2.1.4) werden ebenfalls in den *Participant Behavior Descriptions* spezifiziert, wie in BPEL mit dem <correlationSets>-Konstrukt. Allerdings werden die Nachrichteneigenschaften (die *message properties*) hier nicht als Referenzen auf in WSDL-Dokumenten definierte Eigenschaftswerte verwendet, sondern nur als Namen. Somit werden diese Eigenschaften in den *Participant Behavior Descriptions* nicht typisiert. Dies geschieht erst in den *Participant Groundings*.

Das folgende Listing zeigt die *Participant Behavior Description* für das Reisebüro aus dem Beispiel im Abschnitt 2.2.1:

```
<process name="agency"
        targetNamespace="urn:booking:agency"
        abstractProcessProfile=
        "urn:HPI_IAAS:choreography:profile:2006/12">
        <sequence>
                <receive wsu:id="ReceiveTripOrder"
                        createInstance="yes" />
                <forEach wsu:id="RequestPriceFE" parallel="yes">
                        <scope>
                                <sequence>
                                        <invoke wsu:id="RequestPrice" />
                                        <receive wsu:id="ReceivePrice" />
                                </sequence>
```

```
                    </scope>
                </forEach>
                <opaqueActivity name="SelectAirline" />
                <invoke wsu:id="OrderTickets" />
                <receive wsu:id="ReceiveOrderConfirmation" />
                <opaqueActivity name="CreateItinerary" />
                <invoke wsu:id="IssueItinerary" />
        </sequence>
</process>
```

Listing 2.1: *Participant Behavior Description* [DKLW07a]

Szenarien, in denen verschiedene Teilnehmer des gleichen Teilnehmertyps teilnehmen, sind recht häufig. Interaktionen mit diesen Teilnehmern finden üblicherweise parallel statt. Die Anzahl dieser Teilnehmer ist oft erst zur Laufzeit bekannt. Im Beispiel ist dies bei den Fluglinien der Fall. Wird in einer in der *Participant Behavior Description* verwendeten *forEach*-Schleife kein *counterName* spezifiziert, so ändert sich die Semantik dieses forEach. Diese forEach-Schleife läuft nicht über einen Integer Zähler, sondern iteriert über eine Menge von Teilnehmern. Teilnehmer und Teilnehmermengen werden in der *Participant Topology* angegeben.

2.2.3. Participant Topology

Eine *Participant Topology* legt die strukturellen Aspekte einer Choreografie fest. Dazu werden Teilnehmertypen, Teilnehmer und *Message Links* definiert. Ein Teilnehmertyp wird dabei von genau einer *Participant Behavior Description* repräsentiert.

Das folgende Listing zeigt die *Participant Topology* für das Beispiel aus Abschnitt 2.2.1:

```
<topology name="bookingtopology"
        targetNamespace="urn:booking"
        xmlns:agency="urn:booking:agency">
        <participantTypes>
                <participantType name="Agency"
                        participantBehaviorDescription="agency:agency" />
                <participantType name="Traveler" ... />
                <participantType name="Airline" ... />
        </participantTypes>
        <participants>
                <participant     name="traveler" type="Traveler" selects="agency" />
                <participant     name="agency" type="Agency" selects="airlines" />
                <participantSet name="airlines" type="Airline"
                        forEach="agency:RequestPriceFE">
                        <participant name="currentAirline"
                                forEach="agency:RequestPriceFE" />
                        <participant name="selectedAirline" />
                </participantSet>
        </participants>
        <messageLinks>
                <messageLink name="tripOrderLink"
                        sender="traveler"
                        sendActivity="SubmitTripOrder"
                        receiver="travelagency"
                        receiveActivity="ReceiveTripOrder"
                        messageName="tripOrder" />
```

```
            <!-- ... -->
            <messageLink name="ticketOrderLink"
                    sender="travelagency"
                    sendActivity="OrderTickets"
                    receiver="selectedAirline"
                    receiveActivity="ReceiveOrder"
                    messageName="ticketOrder"
                    participantRefs="traveler" />
            <messageLink name="eTicketLink"
                    sender="selectedAirline"
                    sendActivity="IssueETicket"
                    receiver="traveler"
                    receiveActivity="ReceiveETicket"
                    messageName="eTicket" />
        </messageLinks>
</topology>
```

Listing 2.2: *Participant Topology* [DKLW07a]

Ein Teilnehmer trägt einen Namen, der diesen eindeutig identifiziert, und verweist mittels dem „type“-Attribut auf genau einen Teilnehmertyp. Es kann allerdings sein, dass mehrere Teilnehmer desselben Teilnehmertyps vorkommen, wie etwa die Fluglinien in unserem Beispiel. Dies wird durch die Definition von Teilnehmermengen (*participant sets*) unterstützt. Das optionale Attribut „selects“ eines Teilnehmers gibt an, welchen anderen Teilnehmer ein Teilnehmer auswählt. Innerhalb einer Teilnehmermenge können Teilnehmerreferenzen oder Teilnehmermengen dieses Typs definiert werden.

Dies ermöglicht es uns, aus einer Menge von Teilnehmern desselben Typs einen auszuwählen, wie in unserem Beispiel die „selectedAirline“ , die der vom Reisebüro ausgewählten Fluglinie entspricht. Weiter werden, etwa für parallel ablaufende Äste, weitere Teilnehmerreferenzen benötigt. In obigem Beispiel ist das der Fall für das forEach-Konstrukt. Eine Teilnehmerreferenz repräsentiert den Teilnehmer der Teilnehmermenge, der in einem der parallel ausgeführten Äste ausgewählt wird. In unserem Beispiel ist diese Teilnehmerreferenz „currentAirline“.

Mittels des „forEach“-Attributs ist diese Teilnehmermenge, und entsprechend ebenso die Teilnehmerreferenz „currentAirline“, auf diesen Gültigkeitsbereich beim Partnerprozess beschränkt.

Mittels eines optionalen „scope“-Attributs kann eine Teilnehmerreferenz auf eine Menge von Gültigkeitsbereichen beim Partnerprozess beschränkt werden.

Message Links geben an, welche Teilnehmer potentiell mit welchem anderen Teilnehmer kommunizieren können. Die Namen der dort angegebenen Aktivitäten entsprechen dabei den in den *Participant Behavior Descriptions* angegebenen Aktivitäten. Die Einschränkungen, die sich aus der Anordnung der Aktivitäten ergeben, werden nicht in der *Participant Topology* angegeben, da dies schon in den *Participant Behavior Descriptions* geschieht. Das Attribut „sender“ gibt an, welche Teilnehmermenge oder welche Teilnehmerreferenz der Sender ist, das Attribut „receiver“ entsprechend, welche Teilnehmerreferenz der Empfänger ist. Das „sendActivity“-Attribut gibt an, welche Aktivität des Teilnehmers sendet, während das „receiveActivity“-Attribut angibt, welche Aktivität des empfangenden Teilnehmers die Nachricht empfängt. Das „sender“-Attribut kann durch das „senders“-Attribut ersetzt werden. Gibt es zu einer empfangenden Aktivität mehrere potentielle Sender, so enthält das „senders“-Attribut die Menge dieser Sender. Gibt es zu einer sendenden Aktivität mehrere empfangende Aktivitäten bei einem Partnerprozess,

so erhalten diese beim „wsu:id“-Attribut denselben Wert, womit hierfür nur noch ein *Message Link* benötigt wird. Entsprechend verfahren wir in dem Fall, in dem zu einer empfangenden Aktivität mehrere sendende Aktivitäten beim Partnerprozess existieren.

Für *Message Links* gelten folgende Bedingungen:

1. Eine <receive>-Aktivität, ein <onMessage>-Zweig, ein <onEvent>-Zweig sowie, im Falle einer *Request-Response-Operation*, die <invoke>-Aktivität sind in *Message Links* als *receive-Activity* gültig.

2. Wird mit einer <invoke>-Aktivität eine *One-Way-Operation* realisiert, so darf diese Aktivität nicht als *receiveActivity* in den *Message Links* vorkommen.

3. Gültig als *sendActivity* bei den *Message Links* sind die <invoke>- und <reply>-Aktivitäten. Dabei gilt, dass für jede <invoke>- und <reply>-Aktivität genau ein *Message Link* existiert, bei dem diese Aktivität die *sendActivity* ist. Daraus folgt, wenn zu einer dieser sendenden Aktivitäten mehrere empfangende Aktivitäten existieren, so müssen diese denselben Namen tragen. Ansonsten würden für eine sendende Aktivität mehrere *Message Links* in der *Participant Topology* definiert werden.

4. Für jede <receive>-Aktivität und jeden <onMessage>- bzw. <onEvent>-Zweig existiert genau ein *Message Link*, bei der diese Aktivität die *receiveActivity* ist. Tritt der Fall ein, dass es zu einer empfangenden Aktivität mehrere potentielle Sender gibt, so wird in einem *Message Link* die Menge dieser Sender angegeben. Allerdings gilt, dass diese Sender vom gleichen Teilnehmertyp sein müssen.

```
<messageLink name="NCName"
        (sender="NCName" | senders="NCNames")
        sendActivity="NCName"
        receiver="NCName"
        receiveActivity="NCName"
        bindSenderTo="NCName"?
        messageName="NCName"
        (participantRefs="NCNames"
        copyParticipantRefsTo="NCNames"?)?
/>
```

Listing 2.3: Schema eines *Message Links* [DKLW07a]

Manchmal kann es nötig sein, dass eine Referenz auf einen Teilnehmer weitergereicht wird. Dies geschieht über das Attribut „participantRefs“ in den *Message Links*. Dieses Vorgehen wird *link passing mobility* genannt. Im Beispiel übergibt etwa das Reisebüro der Fluglinie bei der Buchung des Fluges eine Referenz auf den Kunden, so dass die betreffende Fluglinie weiß, welchem Kunden sie das Ticket schicken muss. Das „copyParticipantRefsTo“-Attribut wird genutzt, wenn ein Teilnehmer eine Teilnehmerreferenz empfängt und ihr einen anderen Namen geben will.

Wird beim „sender“-Attribut eine Teilnehmermenge oder mehrere Teilnehmerreferenzen angegeben, so kann das „bindSenderTo“-Attribut Verwendung finden. Hier wird eine Teilnehmerreferenz angegeben, in die der Empfänger den Teilnehmer schreibt, der ihm die Nachricht geschickt hat, da eine einzelne Nachricht nur von genau einem Sender kommen kann.

2.2.4. Participant Groundings

Bei den *Participant Groundings* kommen die technischen Details ins Blickfeld. Hier wird der Bezug zu den WSDL-Definitionen und den XML-Schematypen hergestellt. Somit wird die Choreografie Web-Service-spezifisch.

Das folgende Listing zeigt die *Participant Groundings* für das Beispiel aus dem Abschnitt 2.2.1:

```
<grounding topology="top:bookingtopology"
        xmlns:top="urn:booking" xmlns:...>
        <messageLinks>
                <messageLink name="tripOrderLink"
                        portType="ag1:travelAgency_pt"
                        operation="getTripRequest" />
                <messageLink name="ticketOrderLink"
                        portType="lhx:web_pt"
                        operation="getOrder" />
                <!-- ... -->
        </messageLinks>
        <participantRefs>
                <participantRef name="traveler"
                        WSDLproperty="msgs:travelerProp" />
        </participantRefs>
</grounding>
```

Listing 2.4: *Participant Groundings* [DKLW07a]

Für jeden Link wird eine „*Port Type*/Operation“-Kombination angegeben. Über das „name“-Attribut wird auf einen entsprechenden *Message Link* in der *Participant Topology* verwiesen. Somit kann ein Teilnehmer durch verschiedene *Port Types* realisiert werden.

Da bei den *Participant Behavior Descriptions* bei Korrelationsmengen nur die Namen der Eigenschaften verwendet werden, müssen diese in den *Participant Groundings* noch typisiert werden.

Das Attribut „participantRefs“ ermöglicht die *link passing mobility*. Analog zu Korrelationen werden *WSDL Properties* referenziert, d. h. es wird auf Eigenschaftswerte verwiesen, so dass eine Referenz auf einen Teilnehmer eindeutig ist.

KAPITEL 3

TRANSFORMATION VON BPEL-PROZESSEN IN EINE BPEL4CHOR-BESCHREIBUNG

Gegeben sind eine Menge von BPEL-Prozessen und WSDL-Definitionen. Daraus sind die drei Bestandteile einer BPEL4Chor-Beschreibung zu generieren: die *Participant Topology*, die *Participant Groundings* und zu jedem gegebenen BPEL-Prozess die passende *Participant Behavior Description* (PBD). Abbildung 3.1 zeigt das Ziel der Transformation.

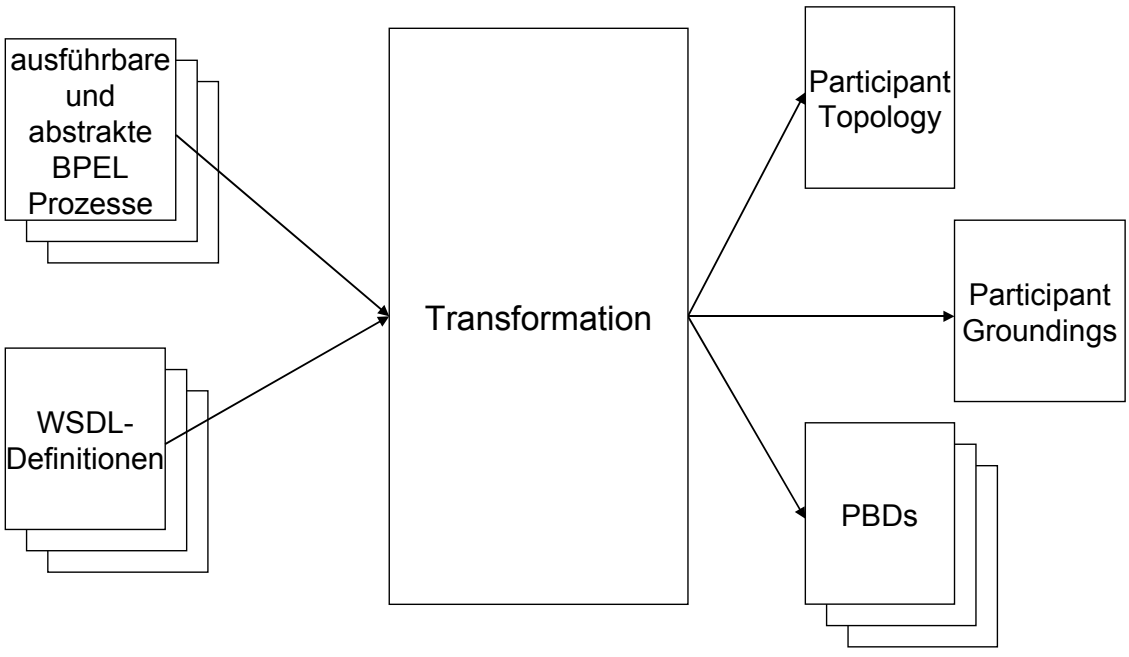

Abbildung 3.1.: Ziel der Transformation

Zunächst sind aus den BPEL-Prozessen und den WSDL-Definitionen die Informationen herauszulesen und die Informationen hinzuzufügen, die benötigt werden, um diese drei Bestandteile einer BPEL4Chor-Beschreibung generieren zu können.

Aufgrund der Komplexität von BPEL werden einige der dort möglichen Fälle von vorneherein ausgeschlossen (siehe Abschnitt 3.1). Auch bei den einzelnen Schritten der Informationsgewinnung kann es vorkommen, dass einige Fälle ausgeschlossen und Einschränkungen vorgenommen werden müssen. Diese sind bei einer Nutzung und einer etwaigen, späteren Erweiterung dieses Ansatzes zu berücksichtigen. Eine übersichtliche Zusammenfassung dieser Fälle findet sich im Anhang A.

Ein zentraler Punkt der Transformation ist das Finden von *Message Links*, d. h. zunächst das Finden von Paaren miteinander kommunizierender Aktivitäten. Dieser Themenkomplex wird in Abschnitt 3.2 behandelt.

Für die Bestimmung der Teilnehmer in der *Participant Topology* (siehe Abschnitt 2.2.3) ist es notwendig zu wissen, welche Beziehung zwischen zwei Prozessmodellen zum Zeitpunkt eines *Message Links* herrscht, d. h. ob ein Prozess mit mehreren Instanzen eines anderen BPEL-Prozesses kommuniziert. Dies untersuchen wir in Abschnitt 3.3.

Der folgende Abschnitt 3.4 befasst sich mit der Erstellung dieser Teilnehmer und Teilnehmermengen.

In Abschnitt 3.5 vereinigen wir Aktivitätsverbindungen miteinander und erzeugen neue Teilnehmerreferenzen, die jeweils eine Menge von Teilnehmerreferenzen ersetzen können. Somit gleichen wir unsere bisher erzeugten Informationen den BPEL4Chor-Vorgaben an.

In den PBDs dürfen, wie in BPEL, Korrelationsmengen vorkommen. Allerdings werden statt QNames hier NCNames verwendet. Wie wir dabei vorgehen müssen, sehen wir im Abschnitt 3.6.

Die Abschnitte 3.7, 3.8 und 3.9 befassen sich mit der Generierung der *Participant Topology*, den *Participant Groundings* und den PBDs.

Im Folgenden bezeichnen wir <invoke>-, <reply>- und <receive>-Aktivitäten als kommunizierende Aktivitäten. Aus Gründen der Lesbarkeit fallen die <onMessage>-Konstrukte einer <pick>-Aktivität ebenfalls unter die Bezeichnung kommunizierende Aktivitäten.

Im Folgenden bedeutet „instanzerzeugend“, dass die betreffende Aktivität ein „createInstance“-Attribut besitzt, das mit dem Wert „yes“ belegt ist. Im Falle eines <onMessage>-Zweigs einer <pick>-Aktivität bedeutet „instanzerzeugend“, dass die <pick>-Aktivität selbst das mit dem Wert „yes“ belegte „createInstance“-Attribut besitzt.

Wir betrachten jeden gegebenen BPEL-Prozess als XML-Baum. Dabei ist das Konstrukt <process> das Wurzelelement eines solchen Baumes. Eine <sequence>-Aktivität etwa ist ein Element des Baumes, dessen eingefassten, sequentiell abzuarbeitenden Aktivitäten dessen Kinder sind. Jedes Element des Baumes können wir als eigenständiges Objekt auffassen.

3.1. Annahmen und ausgeschlossene Fälle

Die *link passing mobility*, d. h. das Weiterreichen von Endpunktreferenzen, wird in dieser Arbeit nicht behandelt.

Handler (siehe Abschnitt 2.1.3) und die <extensionActivity>-Aktivität sind nicht Bestandteil dieser Arbeit und werden im Folgenden nicht weiter behandelt. Die in dieser Arbeit betrachteten BPEL-Prozesse dürfen folglich keine Handler und keine <extensionActivity>-Aktivitäten beinhalten.

Es werden keine Fälle behandelt, in denen zu einer Aktivität mehrere passende Aktivitäten bei verschiedenen BPEL-Prozessen existieren. Diese Fälle werden von BPEL4Chor bisher nicht abgedeckt. Ein Beispiel findet sich in Abbildung 3.2. Im dort dargestellten Beispielfall tritt der Fall auf, dass der eingebundene Prozess über eine <reply>-Aktivität verschiedenen BPEL-Prozessen antworten kann. Ein derartiger Fall wird von BPEL4Chor bisher nicht abgedeckt, da

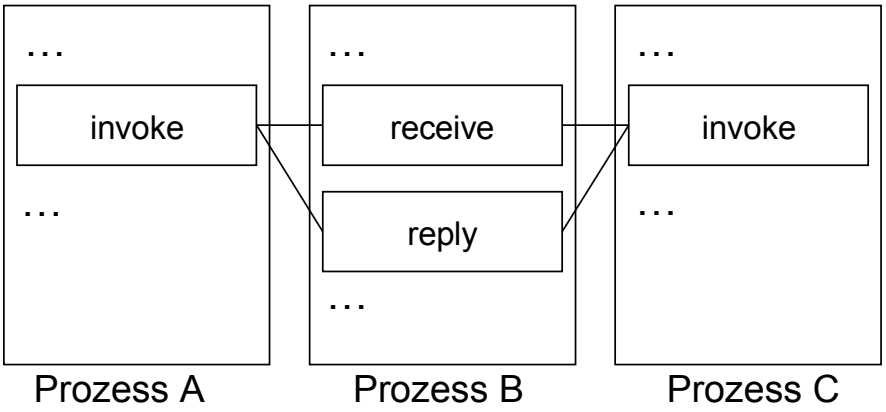

Abbildung 3.2.: Invoke-Aktivitäten bei verschiedenen Prozessen, die mit derselben Receive-Aktivität kommunizieren

wir zu z. B. der obigen <reply>-Aktivität mehrere mögliche Empfänger hätten, aber bei einem *Message Link* in der *Participant Topology* nur ein möglicher Empfänger als „receiver“ angegeben werden kann und für die <reply>-Aktivität nur ein *Message Link* existieren darf, in dem diese Aktivität die „sendActivity“ ist. Hier kann Prozess B mittels einer <receive>-Aktivität entweder von Prozess A oder von Prozess C eine Nachricht empfangen und dem entsprechenden Prozess mittels der <reply>-Aktivität antworten. Auch legt BPEL4Chor fest, dass die beim „senders“-Attribut in einem *Message Link* angegebenen Sender vom gleichen Teilnehmertyp sind (siehe Abschnitt 2.2.3). Diese Vorgabe wird in obigem Beispiel ebenfalls verletzt.

Abseits des Abschnitts 3.6 werden Korrelationsmengen in dieser Arbeit nicht berücksichtigt. Dies führt dazu, dass wir in den folgenden Abschnitten Fälle ausschließen müssen, die dort jeweils angegeben werden.

Als Vereinfachung gehen wir davon aus, dass höchstens eine <receive>-Aktivität bzw. ein <onMessage>-Zweig existiert, bei deren Eintreten eine neue Instanz kreiert wird. Eine <pick>-Aktivität mit mehreren instanzerzeugenden <onMessage>-Zweigen schließen wir folglich aus.

3.2. Finden von Aktivitätsverbindungen

Der erste Schritt der Transformation ist das Finden von Paaren miteinander kommunizierender Aktivitäten, den „Aktivitätsverbindungen“.

Eine Aktivitätsverbindung besteht aus zwei kommunizierenden Aktivitäten. Eine dieser Aktivitäten ist dabei die sendende Aktivität, die andere Aktivität ist die empfangende Aktivität. In den weiteren Abschnitten werden die Aktivitätsverbindungen erweitert, um diesen jeweils weitere Informationen hinzuzufügen, die wir zum Generieren der *Message Links* gemäß der von BPEL4Chor vorgegebenen Struktur in den *Participant Groundings* und der *Participant Topology* benötigen.

Die Idee des Vorgehens zum Finden der Aktivitätsverbindungen ist folgende: In jedem gegebenen BPEL-Prozess suchen wir die nächste aktiv werdende kommunizierende Aktivität. Kommt eine <flow>-Aktivität in einem BPEL-Prozess vor, so kann für einen BPEL-Prozess mehr als eine aktiv werdende kommunizierende Aktivität gefunden werden. In der Menge der gefundenen aktiven kommunizierenden Aktivitäten suchen wir nach Aktivitätsverbindungen. Finden wir eine Aktivitätsverbindung, so wird im Anschluss von den beiden beteiligten Aktivitäten aus nach neuen, aktiv gewordenen kommunizierenden Aktivitäten gesucht.

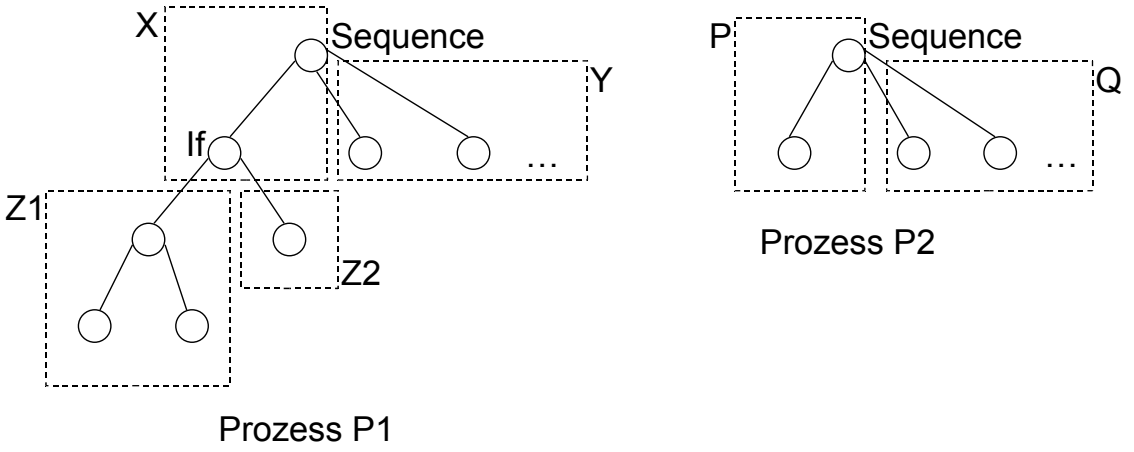

Abbildung 3.3.: Verzweigung in einem Prozess

In Abbildung 3.3 wird das Vorgehen illustriert. Hier sind zwei Prozesse P1 und P2 abgebildet. Der Prozess P1 enthält eine Verzweigung. Es wurden bereits Aktivitätsverbindungen gefunden. Somit sind die Bereiche X und P bereits bearbeitet. If ist die Aktivität, bei der wir uns aktuell bei der Suche nach einer kommunizierenden Aktivität befinden. Z1 und Z2 sind die zu bearbeitenden Zweige. Y ist der restliche Teil des Baums, der noch nicht bearbeitet wurde.

Im Falle einer Verzweigung gehen wir wie folgt vor: Für jeden möglichen Zweig einer Verzweigung bearbeiten wir sowohl diesen Zweig als auch den restlichen Teil des Baums, der noch nicht bearbeitet wurde. In obiger Abbildung 3.3 müssen wir, wenn wir auf die <if>-Aktivität treffen, für den ersten möglichen Zweig in diesem BPEL-Prozess noch Z1 und Y bearbeiten. Für den zweiten möglichen Zweig müssen wir in diesem BPEL-Prozess noch Z2 und Y bearbeiten. Dies führt dazu, dass wir Y mehrmals bearbeiten. In Fällen, in denen für kommunizierende Aktivitäten in Y für beide Zweige dieselben Aktivitätsverbindungen gefunden werden, kann dies überflüssig sein. In solch einem Fall können wir die Suche nach weiteren Aktivitätsverbindungen vorzeitig abbrechen. Auf keinen Fall wird allerdings X nochmals bearbeitet.

An dem Punkt der Verzweigung werden sowohl Y als auch andere BPEL-Prozesse mehrfach bearbeitet. In obigem Beispiel haben wir für den zweiten BPEL-Prozess zu dem Zeitpunkt, an dem wir im ersten BPEL-Prozess auf die <if>-Aktivität treffen, den Teil P des entsprechenden Baumes bereits bearbeitet. Der noch nicht bearbeitete Teil Q wird nun wie Y zweimal bearbeitet, während der bereits bearbeitete Teil P für keinen der beiden Zweige der Verzweigung erneut bearbeitet wird.

Formale Beschreibung

Definition 3.1: *P* ist die Menge der gegebenen BPEL-Prozesse. Jeder BPEL-Prozess ist durch einen QName eindeutig bezeichnet.

Gegeben ist eine Menge von BPEL-Prozessen. Sei k die Anzahl der gegebenen BPEL-Prozesse. Jedem gegebenen BPEL-Prozess wird eine eindeutige Zahl zwischen 1 und k zugeordnet. Da wir später den NCName der BPEL-Prozesse verwenden wollen, sorgen wir mittels Umbenennung dafür, dass keine verschiedenen BPEL-Prozesse mit demselben NCName gegeben sind.

Um später einen eindeutigen Bezeichner für jede kommunizierende Aktivität zu haben, müssen die gegebenen BPEL-Prozesse wie folgt erweitert werden: Jede kommunizierende Aktivität wird um das Attribut „wsu:id“ erweitert. Darüberhinaus wird jedes <scope>- und <forEach>-Konstrukt um dieses Attribut erweitert. Wir werden im Abschnitt 3.7 sehen, weshalb wir diese Erweiterung nicht auf die kommunizierenden Aktivitäten beschränken, sondern diese beiden Konstrukte miteinbeziehen. Der Wert des „wsu:id“-Attributs muss für jede dieser Aktivitäten eindeutig innerhalb des BPEL-Prozesses sein. Dies erreichen wir, indem wir die betreffenden Konstrukte der BPEL-Prozesse durchnummerieren.

Zu bestimmen ist zunächst die Menge der *Partner Links* der gegebenen BPEL-Prozesse.

Definition 3.2: *PL* ist die Menge aller *Partner Links* der gegebenen BPEL-Prozesse.

Die Menge *PL* bilden wir, indem wir über alle BPEL-Prozesse hinweggehen und jeden gefundenen *Partner Link* dieser Menge hinzufügen.

Von Interesse ist, welche anderen *Partner Links* zu einem bestimmten *Partner Link* „passen“.

Definition 3.3: Zwei *Partner Links* passen zueinander, wenn sie vom gleichen Partner-Link-Typ sind, und es gilt:
1. Der Wert des „myRole“-Attributs des einen *Partner Links* stimmt mit dem Wert des „partnerRole“-Attributs des anderen *Partner Links* überein.
2. Der Wert des „partnerRole“-Attributs des einen *Partner Links* stimmt mit dem Wert des „myRole“-Attributs des anderen *Partner Links* überein.
Im Falle einer nicht angegebenen Rolle (siehe Abschnitt 2.1.2 und [JE07]) muss nur eine dieser beiden Bedingungen gelten.

Definition 3.4: $search_{PL} : PL \rightarrow 2^{PL}$ ist die Funktion, die zu einem gegebenen *Partner Link* $pl \in PL$ die Menge der passenden *Partner Links* zurückliefert.

Zu beachten gilt, dass die zurückgelieferte Menge für bestimmte *Partner Links* leer sein kann. Dies kann auftreten, wenn eine Aktivität mit einem BPEL-Prozess kommunizieren will, der nicht gegeben ist.

Definition 3.5: Als *Initiator* bezeichnen wir einen BPEL-Prozess, dessen erste kommunizierende Aktivität eine <invoke>-Aktivität ist. Ein Initiator besitzt folglich keine instanzerzeugende Aktivität.

Bei einem Initiator handelt es sich um einen abstraken Prozess, dessen Zweck es ist, beim Rest der gegebenen BPEL-Prozesse einen oder mehrere Startpunkte festzulegen. Allerdings ist ein Initiator hierfür nicht zwingend notwendig.

Andere abstrakte Prozesse, die nicht als Initiator fungieren, müssen eine instanzerzeugende Aktivität besitzen. Bei abstrakten Prozessen müssen bei kommunizierenden Aktivitäten die „variable"-„inputVariable"- und „outputVariable"-Attribute oder die entsprechenden <fromParts>- und <toParts>-Elemente angegeben werden. Dies benötigen wir, da wir später bei einer <invoke>-Aktivität direkt erkennen wollen, ob diese Aktivität eine synchrone Kommunikation ermöglicht.

Ist in der Menge der gegebenen BPEL-Prozesse kein Initiator vorhanden, so liefert die Funktion $search_{PL}$ für mindestens einen *Partner Link* eine leere Menge zurück. Dies ist der Fall, da bei allen gegebenen BPEL-Prozessen die erste kommunizierende Aktivität eine empfangende, instanzerzeugende Aktivität ist. Ein Initiator besitzt für einen Teil der instanzerzeugenden Aktivitäten dieser BPEL-Prozesse passende *Partner Links*. Da der Initiator fehlt, liefert die Funktion $search_{PL}$ für die entsprechenden *Partner Links* jeweils eine leere Menge zurück.

Zur Verwirklichung der anfangs skizzierten Idee benötigen wir vier Mengen:

Definition 3.6: S ist eine Menge von Konstrukten, von denen aus wir die nächsten kommunizierenden Aktivitäten suchen müssen.

Definition 3.7: O ist die Menge der gefundenen kommunizierenden Aktivitäten, mit denen in dem aktuellen Lauf noch keine Aktivitätsverbindung gebildet wurde.

Im Fall einer Verzweigung kann bereits bei der Betrachtung eines anderen Zweiges mit einer der in O enthaltenen Aktivitäten eine Aktivitätsverbindung gebildet worden sein. Dies ist etwa für Aktivitäten aus den Teilen Y und Q der Bäume aus Abbildung 3.3 der Fall, die für jeden Zweig der Verzweigung bearbeitet werden.

Definition 3.8: B ist die Menge der Konstrukte, die wir bereits bearbeitet haben.

B benötigen wir, um sicherzustellen, dass ein zu einem Zeitpunkt bereits bearbeiteter Teil eines Baumes nicht ein weiteres Mal bearbeitet wird.

Definition 3.9: AV ist die Menge der bereits gefundenen Aktivitätsverbindungen bei der Betrachtung eines einzelnen Zweiges.

Bei diesen Mengen handelt es sich um mathematische Mengen, d. h. es dürfen in einer Menge keine zwei identischen Elemente vorkommen.

Auf den folgenden Seiten behandeln wir zunächst die <receive>-, <invoke>-, <exit>-, <sequence>-, <if>-, <forEach>-, <repeatUntil>-, <while>- und <scope>-Aktivitäten. Dieser Ansatz kann und wird anschließend um die fehlenden <pick>-, <reply>- und <flow>-Aktivitäten erweitert werden. Andere Aktivitäten spielen erst bei der Miteinbeziehung der <flow>-Aktivität eine Rolle, da wir hier die Links betrachten müssen. <elseif>- und <else>-Konstrukte betrachten wir im Folgenden als strukturierte Aktivitäten.

Über die Mengen S, O und B kennen wir den aktuellen Zustand, in dem sich ein BPEL-Prozess befindet. Im Beispiel in Abbildung 3.4 suchen wir die nächste kommunizierende Aktivität der

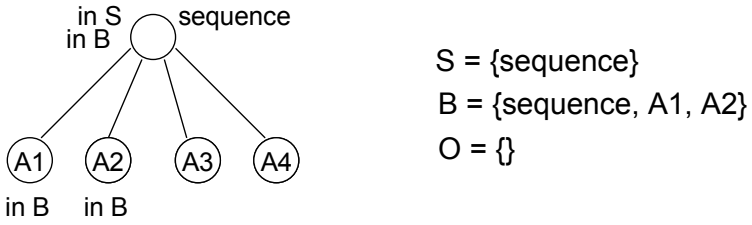

Abbildung 3.4.: Die Mengen *S*, *O* und *B*

<sequence>-Aktivität. Mit A1 und A2 wurden zuvor bereits Aktivitätsverbindungen gebildet. Wir überprüfen nun die Kinder in ihrer Reihenfolge solange, bis wir eines finden, das noch nicht in *B* enthalten ist. In diesem Fall ist dies A3. A3 wird dann in *O* und *B* eingefügt und die <sequence>-Aktivität aus *S* entfernt. Mit A3 kann im Folgenden eine Aktivitätsverbindung generiert werden. Im Anschluss suchen wir die nächste kommunizierende Aktivität der <sequence>-Aktivität.

Die Funktion *FIND-ACTIVITIES* dient der Findung der Aktivitätsverbindungen. *FIND-ACTIVITIES* sucht von einem aus *S* ausgewählten Element nach der darauffolgenden Aktivität. Ist in *S* kein Element vorhanden, so wird die Funktion *GENERATE-ACTIVITY-LINKS* aufgerufen. In dieser werden die Aktivitätsverbindungen generiert. Die Funktion *FIND-ACTIVITIES* bekommt als Parameter obige vier Mengen übergeben. Die Parameterübergabe geschieht dabei „by-value“, nicht „by-reference“. Dies trifft ebenso bei allen anderen Funktionen in dieser Arbeit zu. Wir gehen dabei wie folgt vor:

S enthält die Elemente, von denen aus wir die nächste <receive>- oder <invoke>-Aktivität suchen. Das aktuell betrachtete Element ist $e \in S$, von dem aus wir nach der darauffolgenden Aktivität suchen. Das Element *e* wird aus *S* entfernt.

Ist *e* eine Basisaktivität, so wissen wir, dass unterhalb dieses Elements im Baum keine kommunizierende Aktivität vorkommt. Folglich müssen wir vom Vaterknoten aus weitersuchen. Dazu fügen wir den Vaterknoten von *e* in *S* ein und rufen die Funktion *FIND-ACTIVITIES* rekursiv auf. Nach der Beendigung dieses Aufrufs beenden wir die Funktion.

Ist *e* eine strukturierte Aktivität (aber nicht die verzweigende <if>-Aktivität), so sind die Kinder dieser Aktivität zu betrachten. Die Kinder liegen dabei, außer bei der noch nicht betrachteten <flow>-Aktivität, in der Reihenfolge vor, in der sie abgearbeitet werden. Wir suchen nach dem ersten Kind, das noch nicht bearbeitet wurde. Sei *k* dieses Kind. Nun gibt es drei Fälle:

1. *k* ist eine beliebige strukturierte Aktivität:
   Wir fügen *k* in *S* ein, da wir von *k* aus weitersuchen müssen. Wir fügen *k* in *B* ein, da wir dieses Konstrukt nun bereits bearbeitet haben.

2. *k* ist eine <invoke>- oder <receive>-Aktivität:
   Wir fügen *k* in *O* ein, da wir eine aktiv gewordene kommunizierende Aktivität gefunden haben. Wir fügen *k* in *B* ein, da diese Aktivität nun bereits als bearbeitet gilt.

3. *k* ist eine <exit>-Aktivität:
   In diesem Fall wird die entsprechende Instanz des BPEL-Prozesses beendet. In *O* oder *S* fügen wir nichts ein. Somit befindet sich weder in *O* noch in *S* eine Aktivität dieses BPEL-Prozesses. *k* wird nicht in *S* eingefügt, da wir für diesen Prozess keine weiteren kommunizierenden Aktivitäten finden wollen.

In allen drei Fällen wird die Funktion *FIND-ACTIVITIES* rekursiv aufgerufen. Nach der Beendigung dieses Aufrufs beenden wir die Funktion.

Ist e eine <if>-Aktivität, so sind alle Kinder zu betrachten, die noch nicht bearbeitet wurden, d. h. die nicht in B enthalten sind. Da die <flow>-Aktivität hier noch nicht berücksichtigt wird, sind entweder alle Kinder bereits bearbeitet oder alle Kinder noch nicht bearbeitet. Wir unterscheiden zwischen zwei Fällen:

1. Das aktuell betrachtete Kind ist eine <receive>- oder <invoke>-Aktivität:
   Wir bilden eine Kopie von B und eine Kopie von O. Diese nennen wir B' und O'. Der Menge B' werden alle Kinder von e hinzugefügt. Damit ist sichergestellt, dass später, wenn wir diesen Zweig bearbeitet haben, nicht noch ein anderer Zweig bearbeitet wird. Somit wird hier nur ein Zweig betrachtet. In O' wird das aktuell betrachtete Kind eingefügt, da wir mit diesem eine aktiv gewordene kommunizierende Aktivität gefunden haben. Dies fügen wir nicht in O selbst ein, da diese Aktivität für die anderen möglichen Zweige niemals aktiv wird. Nun wird die Funktion *FIND-ACTIVITIES* aufgerufen, wobei dieser die Mengen AV, S, O' und B' übergeben werden.

2. Das aktuell betrachtete Kind ist eine beliebige andere Aktivität:
   Wir bilden eine Kopie von B und eine Kopie von S. Diese nennen wir B' und S'. Wie oben werden alle Kinder von e in B' eingefügt. Das aktuell betrachtete Kind wird der Menge S' hinzugefügt. Dies fügen wir nicht in S ein, da das Kind nur für einen möglichen Zweig betrachtet werden darf. Nun rufen wir die Funktion *FIND-ACTIVITIES* auf, wobei wir dieser die Mengen AV, S', O und B' übergeben.

   Zu beachten gilt, dass das aktuell betrachtete Kind in diesem Fall auch eine bisher nicht betrachtete Basisaktivität sein kann, da im ersten Zweig eine beliebige Aktivität als Kind vorkommen kann. In allen anderen Zweigen ist das betreffende Kind ein <elseif>- oder ein <else>-Konstrukt.

Hat eine <if>-Aktivität keinen <else>-Zweig, so muss der Fall berücksichtigt werden, dass bei einer Ausführung des BPEL-Prozesses keiner der möglichen Zweige aktiv wird. Wir bilden eine Kopie von S. Wir fügen den Vaterknoten der <if>-Aktivität in S' ein. Wir rufen nun die Funktion *FIND-ACTIVITIES* auf, wobei wir dieser die Mengen AV, S', O und B übergeben. Die <if>-Aktivität ist bereits in B enthalten. Damit ist sichergestellt, dass dieser Teilbaum bei der Verfolgung dieses Zweiges nicht mehr bearbeitet wird.

Wurde für jeden möglichen Zweig *FIND-ACTIVITES* aufgerufen, und alle diese Aufrufe beendet, so beenden wir die Funktion.

Ist e eine <exit>-Aktivität, dann fügen wir nichts in S oder O ein und rufen die Funktion *FIND-ACTIVITIES* auf. In S und O sind dabei keine Aktivitäten dieses Prozesses mehr vorhanden. Diesen Fall müssen wir behandeln, da die <exit>-Aktivität ein Kind eines <if>-Konstrukts sein kann. Nach der Beendigung dieses rekursiven Aufrufs beenden wir die Funktion.

Ist e ein <process>-Konstrukt und alle Kinder sind bereits in B enthalten, so rufen wir *FIND-ACTIVITIES* rekursiv auf. e wurde zuvor bereits aus S entfernt. Dieser Fall bedeutet, dass wir den Prozess bereits vollständig bearbeitet haben und nach keiner kommunizierenden Aktivität in diesem Prozess mehr suchen müssen. Nach der Beendigung dieses rekursiven Aufrufs beenden wir die Funktion.

Ist e eine beliebige strukturierte Aktivität, deren Kinder alle bereits bearbeitet wurden, so fügen wir den Vaterknoten von e in S ein. Wir rufen nun *FIND-ACTIVITIES* rekursiv auf.

Definition 3.10: *ALinks* ist die Menge aller Aktivitätsverbindungen, die sich mit den gegebenen BPEL-Prozessen erzeugen lassen.

Wir unterscheiden zwischen den Mengen *ALinks* und AV, da AV nur die bereits gefundenen Aktivitätsverbindungen zu einem bestimmten Zeitpunkt bei der Betrachtung eines bestimmten Zweiges einer Verzweigung enthält.

Beim Start sind O, B und AV leer, während sich in S die <process>-Konstrukte der gegebenen BPEL-Prozesse befinden. Ist dieser Aufruf beendet, so befinden sich alle Aktivitätsverbindungen, die generiert werden können, in der Menge *ALinks*.

Der nachfolgende Pseudocode dient zur Veranschaulichung des beschriebenen Ansatzes.

Funktion FIND-ACTIVITIES

```
S = Konstrukte, von denen aus wir suchen (siehe Def. 3.6)
O = Menge der gefundenen kommunizierenden Aktivitäten, die noch in keiner Aktivitätsverbindung Verwendung fanden (siehe Def. 3.7)
B = Menge der bereits bearbeiteten Konstrukte (siehe Def. 3.8)
AV = Menge der bereits gefundenen Aktivitätsverbindungen (siehe Def. 3.9)
function FIND-ACTIVITIES(S, O, B, AV)
    if S = ∅ then
        GENERATE-ACTIVITY-LINKS(S, O, B, AV)
    else
        boolean verzweigung ← false
        getFirstElement e ∈ S                    ▷ wir betrachten das erste Element in S
        S.remove(e)
        if e.typ ∈ (process, sequence, scope, elseif, else, forEach, while, repeatUntil) then
            K ← e.children                       ▷ K ist Liste der Kinder in ihrer korrekten Reihenfolge
            for all k ∈ K do
                if k ∉ B then
                    if k.typ ∈ (sequence, scope, if, forEach, while, repeatUntil) then
                        B.add(k)
                        S.add(k)
                        FIND-ACTIVITIES(S, O, B, AV)
                        return
                    else if k.typ ∈ (invoke, receive) then
                        O.add(k)
                        B.add(k)
                        FIND-ACTIVITIES(S, O, B, AV)
                        return
                    else if k.typ = exit then
                        FIND-ACTIVITIES(S, O, B, AV)
                        return
                    end if
                end if
            end for
        else if e.typ = exit then
            FIND-ACTIVITIES(S, O, B, AV)
            return
```

```
        else if e.typ = if then
            K ← e1.children                                   ▷ K ist Liste der Kinder
            for all k ∈ K do
                if k ∉ B then
                    if k.typ ∈ (invoke, receive) then
                        verzweigung ← true
                        O' ← O
                        B' ← B
                        O'.add(k)
                        B'.add(K)                             ▷ fügt alle Kinder B' hinzu
                        FIND-ACTIVITIES(S, O', B', AV)
                    else
                        verzweigung ← true
                        S' ← S
                        B' ← B
                        S'.add(k)
                        B'.add(K)                             ▷ fügt alle Kinder B' hinzu
                        FIND-ACTIVITIES(S', O, B', AV)
                    end if
                end if
            end for
            if ¬∃k ∈ K : k.typ = else then
                verzweigung ← true
                S' ← S
                S'.add(e1.parent)                 ▷ fügt das Vaterelement der Menge S' hinzu
                FIND-ACTIVITIES(S', O, B, AV)
            end if
        end if
        if verzweigung then
            return
        end if
        if e.typ = process then
            FIND-ACTIVITIES(S, O, B, AV)
            return
        else                          ▷ e ist eine Basisaktivität oder eine strukturierte Aktivität,
                                                  ▷ bei der alle Kinder bereits bearbeitet waren.
            S.add(e.parent)       ▷ fügt das Vaterelement unseres aktuellen Elementes S hinzu
            FIND-ACTIVITIES(S, O, B, AV)
            return
        end if
    end if
end function
```

Ist S leer, so können keine weiteren kommunizierenden Aktivitäten gefunden werden. Alle zu diesem Zeitpunkt aktiven kommunizierenden Aktivitäten sind in O enthalten. Die Funktion *GENERATE-ACTIVITY-LINKS* wird aufgerufen, wenn die Menge S leer ist.

In *GENERATE-ACTIVITY-LINKS* wird überprüft, ob sich in O Aktivitäten befinden, die miteinander kommunizieren können. Seien x und y zwei aktuell betrachtete Aktivitäten aus der Menge O. Sei plx der *Partner Link* von x und ply der *Partner Link* von y. Aus diesen beiden Aktivitäten

können wir genau dann eine Aktivitätsverbindung bilden, wenn folgende Bedingungen erfüllt sind:

1. x ist eine sendende und y eine empfangende Aktivität.
2. ply $\in search_{PL}(plx)$.
3. Die Operation von x und die Operation von y sind gleich.

Diese Überprüfung, ob eine Aktivitätsverbindung gebildet werden kann, übernimmt im folgenden Pseudocode die Funktion matches(), die diese Überprüfung für zwei gegebene Aktivitäten vornimmt und einen boolschen Wert zurückliefert.

Kann eine neue Aktivitätsverbindung erzeugt werden, so wird diese in die Menge *AV* eingefügt. Die beiden Aktivitäten x und y werden aus *O* entfernt und in *S* eingefügt, da wir von dort aus nach weiteren kommunizierenden Aktivitäten suchen müssen.

Können keine weiteren Aktivitätsverbindungen generiert werden, so wird die Funktion *FIND-ACTIVITIES* rekursiv aufgerufen.

Konnte nicht mindestens eine Aktivitätsverbindung gebildet werden, so gibt es drei Möglichkeiten:

1. Die Menge der gefundenen kommunizierenden Aktivitäten *O* ist leer. Dann fügen wir alle Elemente der Menge *AV* in *ALinks* ein.
2. Die Menge *O* ist nicht leer. Weiterhin ist jede dort enthaltene Aktivität instanzerzeugend. Dann fügen wir alle Elemente der Menge *AV* in *ALinks* ein.
3. Ansonsten beenden wir diese Funktion, ohne etwas in *ALinks* einzufügen.

Die zweite Möglichkeit kann auftreten, wenn in einem Durchgang, etwa aufgrund einer Verzweigung, ein BPEL-Prozess erst gar nicht eingebunden wird, folglich keine Instanz dieses BPEL-Prozesses generiert wird. Die Überprüfung, ob dies für jedes Element in *O* gilt, übernimmt die Funktion noInstance(), die einen boolschen Wert zurückliefert.

Die dritte Möglichkeit kann auftreten, wenn wir in zwei BPEL-Prozessen auf je eine Verzweigung treffen. In einem solchen Fall wird im Endeffekt jede mögliche Kombination überprüft. Bei einer unsinnigen Kombination tritt irgendwann der Fall ein, dass wir noch Elemente in *O* haben, aber keine weitere Aktivitätsverbindung gebildet werden kann. Somit verwerfen wir die bis dahin gebildeten Aktivitätsverbindungen.

Funktion GENERATE-ACTIVITY-LINKS

S = Konstrukte, von denen aus wir suchen (siehe Def. 3.6)
O = Menge der gefundenen kommunizierenden Aktivitäten, die noch in keiner Aktivitätsverbindung Verwendung fanden (siehe Def. 3.7)
B = Menge der bereits bearbeiteten Konstrukte (siehe Def. 3.8)
AV = Menge der bereits gefundenen Aktivitätsverbindungen (siehe Def. 3.9)
function GENERATE-ACTIVITY-LINKS(S, O, B, AV)
    $boolean\ neue\_AVs \leftarrow false$
    for all $x \in O$ **do**
        for all $y \in O$ **do**
            if $matches(x, y)$ **then**
                $tmp \leftarrow generateNewActivityLink(x, y)$

```
                AV.add(tmp)
                O.remove(x)
                O.remove(y)
                S.add(x)
                S.add(y)
                neue_AVs ← true
            end if
        end for
    end for
    if neue_AVs then
        FIND-ACTIVITIES(S, O, B, AV)
    else
        if O = empty then
            ALinks.addAll(AV)
        else
            if noInstance(O) = true then
                ALinks.addAll(AV)
            end if
        end if
    end if
end function
```

Ergänzung

Obiger Algorithmus berücksichtigt bisher nicht den Fall, dass keiner der gegebenen BPEL-Prozesse ein Initiator ist. Hier tritt der Fall auf, dass wir in O nur instanzerzeugende Aktivitäten haben. Somit kann keine Aktivitätsverbindung gebildet werden.

Kann keine Aktivitätsverbindung gebildet werden, so müssen wir für alle Aktivitäten in O überprüfen, ob die Menge, die von $search_{PL}$ zurückgegeben wird, leer ist. Ist dies der Fall, so entfernen wir die betreffende Aktivität aus O und fügen sie in S ein, damit von dort aus die nächste kommunizierende Aktivität gesucht werden kann. Anschließend wird die Funktion *FIND-ACTIVITIES* rekursiv aufgerufen. Nach der Beendigung dieses Aufrufs wird die Funktion beendet. Folglich können kommunizierende Aktivitäten existieren, die in keiner Aktivitätsverbindung vorkommen. Wie wir mit diesen Aktivitäten später verfahren, sehen wir im Abschnitt 3.9.

Pick

Die <pick>-Aktivität wird entsprechend der <if>-Aktivität bearbeitet. Ein Kind kann entweder ein <onMessage>- oder ein <onAlarm>-Konstrukt sein. Ein <onMessage>-Konstrukt wird als kommunizierende Aktivität angesehen, das selbst wiederum Kinder besitzen kann. Die <onAlarm>-Konstrukte werden als strukturierte Aktivitäten angesehen. Analog zur <if>-Aktivität wird für jeden Zweig einer <pick>-Aktivität die Funktion *FIND-ACTIVITIES* rekursiv aufgerufen. Nach der Beendigung aller Aufrufe wird die Funktion beendet.

Es gilt zu beachten, dass Fälle, in denen mehrere instanzerzeugende <onMessage>-Konstrukte existieren, bereits im Abschnitt 3.1 ausgeschlossen wurden. In diesem Fall können gleichzeitig mehrere Instanzen eines BPEL-Prozesses existieren, was in einem solchen Fall zu berücksichtigen ist.

Flow

Ein Link verknüpft immer zwei Aktivitäten miteinander (vgl. [JE07]). Im Folgenden gehen wir davon aus, dass stets „suppressJoinFailure = yes“ gilt. Dies müssen wir annehmen, da wir die Fehlerbehandlung nicht betrachten.

Definition 3.11: Wir bezeichnen einen Link als „gesetzt“, wenn die Aktivität, bei der dieser Link ein ausgehender Link ist, beendet (oder übersprungen) wurde und dieser Link „true“ oder „false“ wird. Zu jedem gesetzten Link merken wir uns, welchen Status dieser besitzt.

Definition 3.12: Ein Link gilt als „benutzt“, wenn bei der Aktivität, bei der dieser Link ein eingehender Link ist, die ausgehenden Links gesetzt werden können.

Bei der <flow>-Aktivität müssen wir alle Kinder betrachten. Alle kommunizierenden Aktivitäten, deren explizite oder implizite *joinCondition* wahr wird, werden der Menge O hinzugefügt. Strukturierte Aktivitäten, deren explizite oder implizite *joinCondition* wahr wird, werden der Menge S hinzugefügt. All diese Aktivitäten werden ebenfalls der Menge B hinzugefügt. Selbiges geschieht entsprechend für Aktivitäten, die keine eingehenden Links besitzen. Aktivitäten, deren explizite oder implizite *joinCondition* zu „false“ evaluiert, werden nicht aktiv. Allerdings werden die ausgehenden Links dieser Aktivitäten mit dem Wert „false“ belegt. Diese Aktivitäten werden ebenfalls der Menge B hinzugefügt. Die <flow>-Aktivität selbst wird aus S entfernt.

Zunächst benötigen wir neben den vier Mengen B, O, S und AV weitere Mengen:

Definition 3.13: L_P ist die Menge der bereits gesetzten Links für jeden gegebenen BPEL-Prozess P.

Definition 3.14: S_P ist eine Menge von Konstrukten. S_P enthält dabei die Konstrukte der BPEL-Prozesse, von denen aus wir unter bestimmten Umständen die nächsten kommunizierenden Aktivitäten suchen müssen.

Die Menge S_P benötigen wir, da Fälle auftreten können, in denen z. B. innerhalb einer <flow>-Aktivität eine <sequence>-Aktivität vorkommt, und bei der Suche nach einer kommunizierenden Aktivität innerhalb dieser <sequence>-Aktivität wir zwar eine solche finden, diese allerdings einen Link benötigt, der zu diesem Zeitpunkt noch nicht gesetzt wurde. Daher müssen wir zu einem späteren Zeitpunkt hier weitersuchen. Nämlich genau dann, wenn einer der dort eingehenden Links gesetzt wurde.

Die beiden Mengen L_P und S_P werden bei den Aufrufen der Funktionen *FIND-ACTIVITIES* und *GENERATE-ACTIVITY-LINKS* diesen als Parameter übergeben. Die Übergabe geschieht dabei „by-value“.

Seien wir auf der Suche nach einer kommunizierenden Aktivität. Sei e unser aktuell betrachtetes Element. Dann müssen wir für die betrachteten Kinder berücksichtigen, ob die eingehenden Links bereits alle gesetzt wurden.

Ist dies der Fall, so können wir, wie in *FIND-ACTIVITIES* beschrieben, weiterarbeiten, wenn die explizite oder implizite *joinCondition* wahr wird oder keine eingehenden Links existieren. Wird die explizite oder implizite *joinCondition* nicht wahr, so wird die Aktivität nur in B eingefügt. Alle ausgehenden Links werden auf „false“ gesetzt. Der Vaterknoten wird in S eingefügt, da wir von dort aus nach der nächsten kommunizierenden Aktivität suchen müssen.

Sind noch nicht alle eingehenden Links gesetzt, so wird e in die Menge S_P eingefügt. Das entsprechende Kind wird in diesem Fall nicht in die Menge B eingefügt, da es noch zu bearbeiten ist.

Das Element e wird in die Menge S_P eingefügt, wenn Folgendes gilt:

1. e ist vom Typ sequence, scope, else, elseif, onMessage oder onAlarm:
   Das aktuell betrachtete Kind ist eine beliebige Aktivität (auch eine bisher nicht betrachtete Aktivität wie etwa <assign>), die eingehende Links besitzt, die noch nicht gesetzt wurden.

2. e ist vom Typ flow, if oder pick:
   Es existiert ein Kind, das nicht bearbeitet werden konnte, da es eingehende Links besitzt, die noch nicht gesetzt wurden.

Im Falle einer <pick>- oder <if>-Aktivität gehen wir wie folgt vor: Für jeden Zweig, den wir bereits bearbeiten können, rufen wir die Funktion *FIND-ACTIVITIES* entsprechend dem bisherigen Vorgehen auf. Existiert nun mindestens ein Zweig, der noch nicht bearbeitet werden kann, so rufen wir dafür die Funktion *FIND-ACTIVITIES* auf. Dabei werden nur die Kinder, die zu diesem Zeitpunkt bearbeitet werden können der dort übergebenen Kopie B' hinzugefügt. Ebenfalls nur hier wird die <pick>- oder <if>-Aktivität in die entsprechende Menge S_P eingefügt, da für die Funktionsaufrufe von *FIND-ACTIVITIES* für die bereits bearbeitbaren Zweige die <pick>- oder <if>-Aktivität als bearbeitet gilt. In diesen Fällen müssen wir von dort aus folglich keine kommunizierende Aktivität mehr suchen. Ein Zweig einer Verzweigung, der eingehende Links hat, dessen *joinCondition* allerdings zu „false“ evaluiert wird, wird nicht verfolgt, gilt aber als bearbeitbarer Zweig.

Wie bisher wird nach der Beendigung all dieser Aufrufe die Funktion beendet.

Links werden gesetzt, d. h. der Menge L_P hinzugefügt, wenn eine der folgenden drei Bedingungen erfüllt ist:

1. Wenn alle Elemente des Teilbaums mit der Wurzel e bereits in B enthalten sind oder e eine Basisaktivität ist.

2. Ist ein betrachtetes Kind eine der Aktivitäten, die wir bisher nicht betrachtet haben (etwa die <assign>-Aktivität). So werden, wenn alle von dieser Aktivität geforderten Links gesetzt wurden, die Links gesetzt, die von dieser Aktivität ausgehen.

3. Die explizite oder implizite *joinCondition* einer Aktivität evaluiert zu „false“. Dann werden alle von dieser Aktivität ausgehenden Links mit dem Wert „false“ gesetzt.

Das Element e wird aus der Menge S_P entfernt, wenn Folgendes gilt:

1. e ist vom Typ sequence, scope, else, elseif, onMessage oder onAlarm:
   Es wird bei der Suche eine Aktivität gefunden, die bearbeitet werden kann (und nicht eine Aktivität, die Links voraussetzt, die noch nicht gesetzt wurden) oder alle Kinder sind bereits bearbeitet.

2. e ist vom Typ flow, if oder pick:
   Alle Kinder sind bearbeitet.

Wird bei einem BPEL-Prozess ein Link gesetzt, so wird das Element aus der Menge S_P in die Menge S eingefügt, dessen Kind das Ziel dieses Links ist.

Ist im Folgenden das Belegen von ausgehenden Links mit dem Wert „true“ bei einer Aktivität mittels *transitionCondition* an Bedingungen geknüpft.

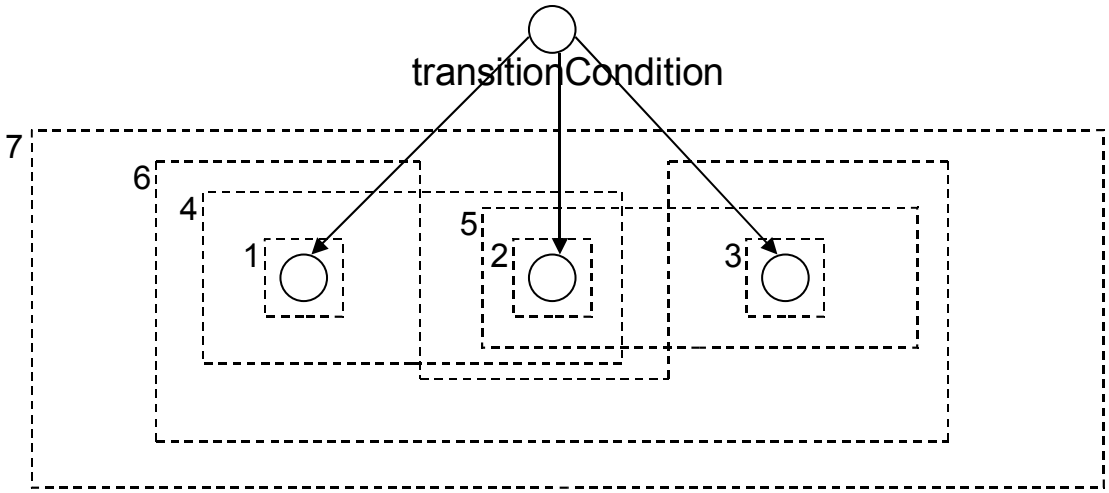

Abbildung 3.5.: Verzweigung in einem Flow mittels *transitionCondition*

Ähnlich dem Vorgehen bei der <if>- und der <pick>-Aktivität rufen wir die Funktion *FIND-ACTIVITIES* mehrmals auf, jeweils mit einer anderen Kombination der Belegung der an Bedingungen geknüpften Links. Sind alle Aufrufe beendet, so beenden wir wie bisher die Funktion.

Allerdings muss das Setzen dieser Links mit dem Wert „true“nicht disjunkt sein. Die Funktion *FIND-ACTIVITIES* wird für jede mögliche Kombination dieser Links aufgerufen. Abbildung 3.5 zeigt einen Fall, in dem eine Funktion drei mittels *transitionCondition* an Bedingungen geknüpfte, ausgehende Links hat. Eingezeichnet sind die sieben möglichen Kombinationen, in denen diese Links den Wert „true“ annehmen können. Ein weiterer, achter Fall kann auftreten, wenn keiner dieser Links auf „true“ gesetzt wird. Jede dieser Kombination betrachten wir als einen eigenen Zweig, den wir verfolgen müssen. Somit hat diese Verzweigung acht mögliche Zweige.

Dieses Vorgehen ist eine Approximation an das tatsächliche Linkverhalten. Zukünftige Arbeiten werden untersuchen, ob sich zwei XPath-Ausdrücke wechselseitig ausschließen. Die Funktion $mutuallyExclusive(Expr1, Expr2) \mapsto true/false$ kann dann entsprechend eingebunden werden. Eine laufende Arbeit, die sich mit diesem Thema befasst, ist [Sch07].

Treffen wir auf eine <exit>-Aktivität, so entfernen wir alle Aktivitäten des betreffenden BPEL-Prozesses aus S und O, bevor wir die Funktion *FIND-ACTIVITIES* erneut aufrufen.

Beispiel 3.1:
Dieses Beispiel bezieht sich auf Abbildung 3.6.

Zunächst finden wir die Aktivität „receive loan request“. Wird diese ausgeführt, so überprüfen wir alle Kombinationen der Belegungen der ausgehenden Links. Dass sich einige XPath-Ausdrücke hier wechselseitig ausschließen, erkennen wir nicht. Die nachfolgenden drei Aktivitäten „ask second clerk“, „ask clerk“ und „automatic decision“ werden entweder aktiv oder

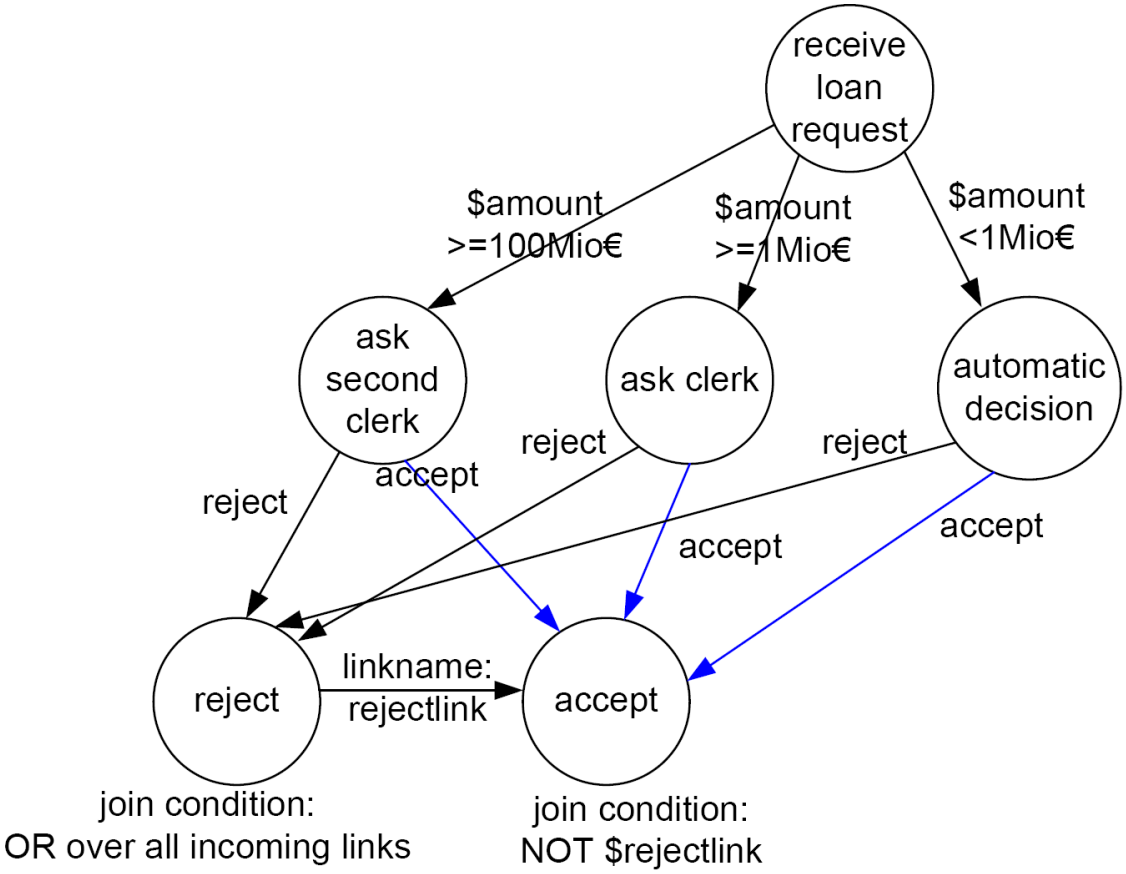

Abbildung 3.6.: Beispiel für die Betrachtung der Link-Semantik

übersprungen. Werden sie übersprungen, so werden die ausgehenden Links auf „false“ gesetzt. Wird eine Aktivität ausgeführt, so werden alle möglichen Kombinationen der Belegungen der ausgehenden Links betrachtet. Dass sich „reject“ und „accept“ wechselseitig ausschließen, erkennen wir nicht. Gilt für einen der bei „reject“ eingehenden Links „true“, so wird diese Aktivität aktiv. Nach der Ausführung dieser Aktivität wird der ausgehende Link auf „true“ gesetzt. Sind alle eingehenden Links hingegen „false“, so evaluiert die *joinCondition* ebenfalls zu „false“. Hier wird diese Aktivität übersprungen und der ausgehende Link auf „false“ gesetzt. Die Aktivität „accept“ wird ausgeführt, wenn dieser Link „false“ ist, d. h. somit keine der Aktivitäten „ask second clerk“, „ask clerk“ und „automatic decision“ den „reject“-Link auf „true“ gesetzt hat.

Behandlung von Sonderfällen

Kommt es in einem Prozess zu einer Verzweigung mittels <if>- oder <pick>-Aktivität und angenommen wir verfolgen einen Zweig, dann müssen alle von Aktivitäten ausgehenden Links, die in anderen, nicht betrachteten Zweigen liegen, auf „false“ gesetzt werden. Zunächst werden die Links auf „false“ gesetzt, deren Aktivitäten, von denen diese Links ausgehen, keine eingehenden Links besitzen oder alle eingehenden Links bereits gesetzt sind. Jedesmal, wenn hier (oder auch später) ein Link gesetzt wird, so prüfen wir, ob das Ziel dieses Links in einem solchen, nicht betrachteten Zweig liegt und ob alle eingehenden Links dieser Aktivität bereits gesetzt wurden. Ist dies der Fall, so werden alle ausgehenden Links auf „false“ gesetzt.

Treffen wir in einer beliebigen, nicht verzweigenden, strukturierten Aktivität, nicht aber der <flow>-Aktivität, auf ein Kind, dessen *joinCondition* zu „false“ evaluiert, dann „überspringen“ wir diese Aktivität, setzen aber alle ausgehenden Links dieser Aktivität auf „false“. Das Überspringen erreichen wir, indem wir diese Aktivität in B einfügen und den Vaterknoten in S. Ist das betrachtete Kind eine strukturierte Aktivität, so müssen, wie bei nicht betrachteten Zweigen oben, ebenfalls alle ausgehenden Links von dort eingebetteten Aktivitäten auf „false“ gesetzt werden, sobald der Status aller eingehenden Links bekannt ist.

Weitere Aspekte beim Miteinbeziehen der <flow>-Aktivität

Wir müssen verhindern, dass wir von einer Aktivität aus mehrmals nach neuen, aktiv gewordenen kommunizierenden Aktivitäten suchen und dabei für diesen Zeitpunkt zu viele kommunizierende Aktivitäten finden.

Wir wollen nun in der Funktion *FIND-ACTIVITIES* vom Vaterknoten aus weitersuchen. Dann betrachten wir zunächst den Teilbaum, der den Vaterknoten als Wurzel hat. Existiert in S, O oder der Menge S_p ein Element, das ein Element dieses Teilbaums ist, so dürfen wir den Vaterknoten nicht in S einfügen.

Der folgende Beispielfall aus Abbildung 3.7 verdeutlicht diese Problematik.

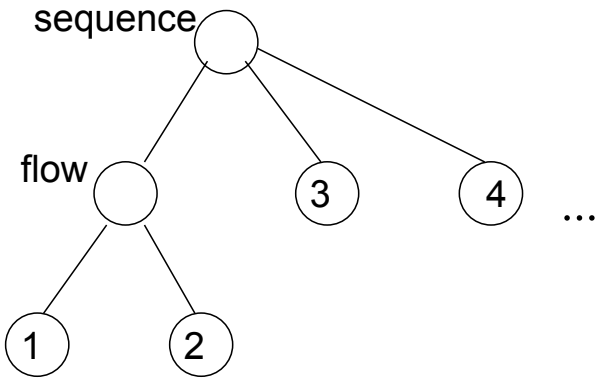

Abbildung 3.7.: Einmalige Suche von derselben Aktivität aus

Konnten mit den Aktivitäten „1“ und „2“ zum selben Zeitpunkt Aktivitätsverbindungen generiert werden, dann befinden sich beide Aktivitäten in S. Suchen wir von „1“ aus nach der nächsten kommunizierenden Aktivität, so suchen wir vom Vaterknoten aus weiter. Der Vaterknoten ist das <flow>-Konstrukt. Allerdings befindet sich ein Element des Teilbaums in S. Folglich suchen wir nicht weiter.

Nun befindet sich noch Aktivität „2“ in S. Hier suchen wir auf die gleiche Weise nach einer neuen kommunizierenden Aktivität. Hier sind keine Elemente des Teilbaums in S, O oder S_p enthalten. Folglich darf die <flow>-Aktivität in S eingefügt werden. Von dort aus müssen wir vom Vaterknoten aus weitersuchen, da alle Kinder bereits bearbeitet sind. Da keine Elemente des Teilbaums in S, O oder S_p enthalten sind, darf die <sequence>-Aktivität in S eingefügt werden. Von dort aus finden wir die nächste kommunizierende Aktivität „3“.

Betrachten wir das nicht, so finden wir an dieser Stelle neben der Aktivität „3“ die Aktivität „4“ , obwohl diese noch gar nicht aktiv werden darf. Diese darf erst aktiv werden, wenn die Aktivität „3“ beendet wird.

Reply

Für die Bearbeitung der <reply>-Aktivität benötigen wir eine weitere Menge:

Definition 3.15: *I* ist die Menge der <invoke>-Aktivitäten, mit denen synchrone *Request-Response-Operationen* verwirklicht werden, und die bisher nur als sendende Aktivität in einer Aktivitätsverbindung vorkommen. Für diese <invoke>-Aktivitäten muss eine zugehörige <reply>-Aktivität gefunden werden.

Die Menge *I* wird ebenfalls den Funktionen *FIND-ACTIVITIES* und *GENERATE-ACTIVITY-LINKS* als Parameter übergeben.

Ob eine <invoke>-Aktivität eine synchrone *Request-Response-Operation* verwirklichen soll, erkennen wir daran, ob bei dieser <invoke>-Aktivität ein „outputVariable"-Attribut vorkommt oder <fromPart>-Elemente vorhanden sind.

Der Zeitpunkt des Einfügens einer solchen <invoke>-Aktivität in *I* ist nach der Generierung einer neuen Aktivitätsverbindung, die eine solche <invoke>-Aktivität als sendende Aktivität besitzt. Nach dem Einfügen der neuen Aktivitätsverbindung in die Menge *AV* wird diese <invoke>-Aktivität in *I* eingefügt. Sie wird noch nicht in *S* eingefügt, da die <invoke>-Aktivität erst dann beendet ist, wenn eine neue Aktivitätsverbindung generiert wird, die eine <reply>-Aktivität als sendende Aktivität und diese <invoke>-Aktivität als empfangende Aktivität beinhaltet.

Bei der Suche nach einer kommunizierenden Aktivität verfahren wir bei einer <reply>-Aktivität genauso, wie es bei den <invoke>- und <receive>-Aktivitäten der Fall ist. Das bedeutet, dass wir gefundene <reply>-Aktivitäten in *O* einfügen. Bei der Generierung neuer Aktivitätsverbindungen verfahren wir in dem Fall, in dem wir in *O* auf eine <reply>-Aktivität treffen, wie folgt: Wir durchsuchen die Menge *I* nach einer passenden <invoke>-Aktivität. Haben wir eine solche gefunden, so generieren wir mit den beiden Aktivitäten eine neue Aktivitätsverbindung, die wir in *AV* einfügen. Die <reply>-Aktivität wird aus *O* entfernt. Die <invoke>-Aktivität wird aus *I* entfernt. Beide Aktivitäten werden in *S* eingefügt.

Ob eine <reply>-Aktivität und eine <invoke>-Aktivität potentiell zueinander passen überprüft die in der Funktion *GENERATE-ACTIVITY-LINKS* bereits benutzte Funktion *match()*. Ist ein „messageExchange"-Attribut angegeben, so müssen wir zusätzlich überprüfen, ob das bei der <reply>-Aktivität angegebene „messageExchange"-Attribut denselben Wert hat, wie das „messageExchange"-Attribut bei der empfangenden Aktivität in der Aktivitätsverbindung, die die betrachtete <invoke>-Aktivität als sendende Aktivität beinhaltet.

Konnte allgemein keine neue Aktivitätsverbindung gebildet werden, wobei die Menge *I* nicht leer ist, so wird die Funktion abgebrochen ohne etwas in *ALinks* einzufügen.

Treffen wir auf eine <exit>-Aktivität, so werden zusätzlich alle Aktivitäten des betreffenden Prozesses aus *I* entfernt, bevor die Funktion *FIND-ACTIVITIES* erneut aufgerufen wird.

Bedingung für einen erfolgreichen Abbruch

Findet in einem BPEL-Prozess eine Verzweigung statt, so rufen wir für jeden möglichen Zweig die Funktion *FIND-ACTIVITIES* rekursiv auf. Nun kann der Fall auftreten, dass wir bei der Abarbeitung eines Zweigs ab einem gewissen Punkt nur noch Aktivitätsverbindungen finden, die bereits in *ALinks* vorhanden sind. Im Fall in Bild 3.8 etwa liefert bereits der rekursive

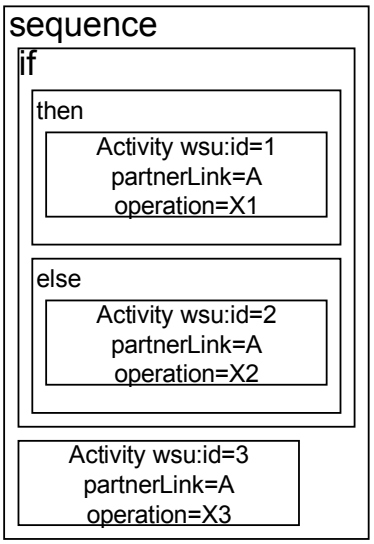

Abbildung 3.8.: Beispielfall für die Abbruchbedingung

Aufruf für den ersten Zweig eine Aktivitätsverbindung für die Aktivität mit der „wsu:id“= 3 (und für alle möglicherweise nachfolgenden Aktivitäten). Gehen wir davon aus, dass beim Partnerprozess keine Verzweigung stattgefunden hat. Sei die Aktivitätsverbindung, die wir für die Aktivität mit der „wsu:id“= 3 bei der Verfolgung des zweiten Zweiges bilden, AL. Dann haben wir bei der Verfolgung des ersten Zweiges für diese Aktivität bereits eine zu AL identische Aktivitätsverbindung gebildet.

Um herauszufinden, ob wir bei der Suche nach neuen Aktivitätsverbindungen abbrechen können, müssen wir allerdings zunächst eine Aktivitätsverbindung bilden, die identisch ist zu einer, die wir bereits kennen. Bilden wir die Aktivitätsverbindung für die Aktivität mit der „wsu:id“= 3, wobei eine identische Aktivitätsverbindung bereits in *ALinks* enthalten ist, so können wir, wenn bestimmte Bedingungen erfüllt sind, diese Suche abbrechen, und die Elemente der aktuellen Menge *AV* in *ALinks* einfügen. Somit wird verhindert, dass wir weitere, bereits bekannte Aktivitätsverbindungen bilden, ohne eine neue, bisher unbekannte Aktivitätsverbindung zu finden. In obigem Beispiel ist das der Fall, wenn nach der Aktivität mit der „wsu:id“= 3 weitere Aktivitäten vorkommen.

Zunächst benötigen wir eine weitere Menge:

Definition 3.16: *UL* beinhaltet die Menge der Links aller Prozesse P, die beim Finden der in diesem Lauf bisher bestimmten Aktivitätsverbindungen bereits „benutzt“ wurde.

Zur Erinnerung: Einen Link betrachten wir als benutzt, wenn bei der Aktivität, bei der dieser Link ein eingehender Link ist, die ausgehenden Links gesetzt werden können (siehe Definition 3.12). Zu jedem benutzten Link wird auch dessen Status vermerkt.

UL enthält folglich die aktuell gesetzten Links, die bereits benutzt wurden. Somit ist die aktuelle Menge *UL* eine Teilmenge der aktuellen Menge L_P.

Können wir nur Aktivitätsverbindungen bilden, die wir bereits kennen, dann wollen wir überprüfen, ob wir noch Aktivitäten erreichen können, die bisher noch in keiner Aktivitätsverbindung vorkommen.

Die Menge *UL* wird den Funktionen *FIND-ACTIVITIES* und *GENERATE-ACTIVITY-LINKS* bei deren Aufruf mit übergeben. Wird ein Lauf erfolgreich abgeschlossen, d. h. alle Aktivitäten wurden bearbeitet, oder es konnte, wie in diesem Kapitel untersucht, erfolgreich abgebrochen werden, so wird die Menge *UL* der Menge *UL_Sets* als Element hinzugefügt.

Definition 3.17: *UL_Sets* beinhaltet die Mengen der benutzten Links als Elemente, die benutzt waren, als ein Lauf beendet wurde.

Nachfolgende Überprüfung geschieht nach der Generierung von Aktivitätsverbindungen. Wir können die Funktion *GENERATE-ACTIVITY-LINKS* erfolgreich abbrechen und die Elemente der aktuellen Menge *AV* in die Menge *ALinks* einfügen, wenn die folgenden Bedingungen erfüllt sind:

1. Alle neu gebildeten Aktivitätsverbindungen waren bereits in der Menge *ALinks* vorhanden.

2. Jede Aktivität in *O* ist bereits Bestandteil einer Aktivitätsverbindung in *ALinks* oder ist instanzerzeugend.

3. Jede Aktivität in *I* ist als empfangende Aktivität Bestandteil einer Aktivitätsverbindung in *ALinks*.

4. Die Menge der gesetzten Links, die noch nicht benutzt wurden, kommen alle in demselben Element aus *UL_Sets* vor, wobei der Status dieser Links mit dem in diesem Element vermerkten Status übereinstimmen muss.

Der vierte Punkt bedeutet: Sind nur Links gesetzt, die bereits in einem Element aus *UL_Sets* enthalten sind, so wissen wir, dass wir keine Aktivitäten mehr finden können, die nicht schon Bestandteil einer Aktivitätsverbindung sind oder bereits bei der erfolgreichen Bestimmung von Aktivitätsverbindungen in einem anderen Zweig betrachtet wurden. Dies können wir schließen, da ein Link nur genau zwei Aktivitäten miteinander verknüpft (siehe [JE07]). Mit den gesetztem Links in L_P können somit keine kommunizierenden Aktivitäten mehr gefunden werden, die nicht sowieso bereits in Aktivitätsverbindungen vorkommen. Durch diese vier Punkte ist gewährleistet, dass wir von diesem Punkt aus keine Aktivitätsverbindung finden, die nicht bereits in identischer Form in *ALinks* enthalten ist. Somit können wir erfolgreich abbrechen und die Elemente der Menge *AV* in die Menge *ALinks* einfügen.

Annahmen und ausgeschlossene Fälle

In dem vorgestellten Algorithmus wurden neben den in Abschnitt 3.1 vorgestellten ausgeschlossenen Fälle und getroffenen Annahmen folgende weiteren Annahmen getroffen und Fälle ausgeschlossen.

Ausgeschlossen wird der Fall, dass ein Prozess in sequentieller Folge mehrere Instanzen eines anderen BPEL-Prozesses einbindet, oder selbst in sequentieller Folge von mehreren Instanzen eines Partnerprozesses eingebunden wird (siehe Bild 3.9). Dies müssen wir ausschließen, da wir die Korrelationsmengen bei unserem Ansatz bisher nicht beachten. Dasselbe gilt für den Fall, dass diese Aktivitäten in einem Flow vorkommen und dort parallel ausgeführt werden dürfen.

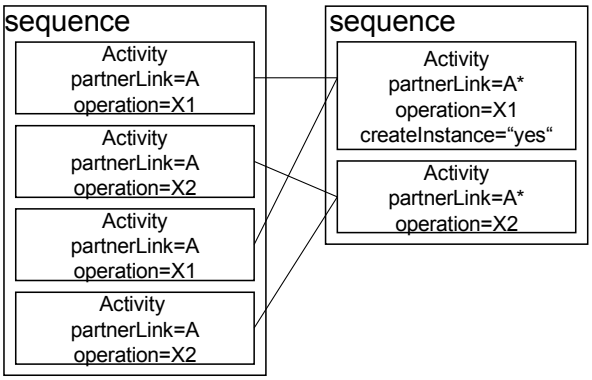

Abbildung 3.9.: Sequentielles Einbinden mehrere Instanzen eines Partnerprozesses

Analog dazu schließen wir aus, dass ein Prozess mit sequentiell aufeinanderfolgenden Aktivitäten mehrere Instanzen eines Partnerprozesses aus einer Menge von Instanzen auswählt. Dasselbe gilt für den Fall, dass diese Aktivitäten in einem Flow vorkommen und dort parallel ausgeführt werden dürfen.

Ausgeschlossen wird der Fall, bei dem in einem BPEL-Prozess eine Schleife durchlaufen wird, im Partnerprozess dies jedoch sequentiell bearbeitet wird (siehe Bild 3.10). Ebenso wird der Fall ausgeschlossen, dass in zwei BPEL-Prozessen, die miteinander kommunizieren, die zwei zu einer Aktivitätsverbindung gehörenden Aktivitäten sich je in einer Schleife befinden, diese aber unterschiedlich lang laufen, d. h. in einem der beiden Prozesse müsste dies sequentiell abgefangen werden. Der Fall, dass eine Schleife gar nicht durchlaufen wird, muss bei einer zukünftigen Erweiterung dieser Arbeit noch berücksichtigt werden. Hier gehen wir davon aus, dass eine Schleife mindestens einmal durchlaufen wird.

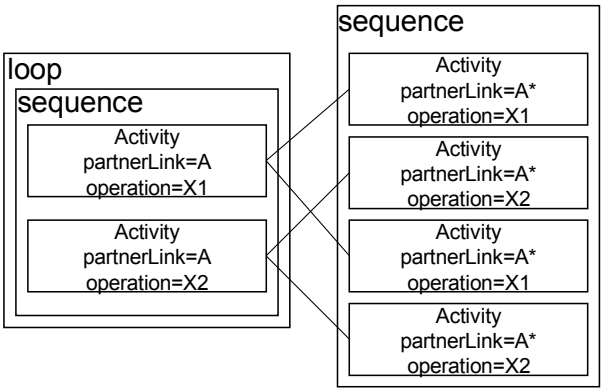

Abbildung 3.10.: Sequentielles Abarbeiten einer Schleife beim Partnerprozess

Eine Annahme ist, dass zu einem Zeitpunkt keine zwei Aktivitäten aktiv sind, die in Konflikt zueinander stehen. Ein Beispiel wären zwei <receive>-Aktivitäten, die denselben *Partner Link* und die selbe Operation besitzen. Diese Annahme müssen wir treffen, da wir Korrelationsmengen in unserem Ansatz bisher nicht beachten.

Ausgeschlossen werden Fälle, in denen wir bei zwei miteinander kommunizierenden BPEL-Prozessen auf je eine asynchrone <invoke>-Aktivität treffen, die nach dem Senden jeweils

beendet wird, und im Anschluss daran die jeweils passende <receive>-Aktivität zur schon beendeten <invoke>-Aktivität des Partnerprozesses aktiv wird (siehe Bild 3.11). In einem solchen Fall würden wir keine Aktivitätsverbindung finden und abbrechen.

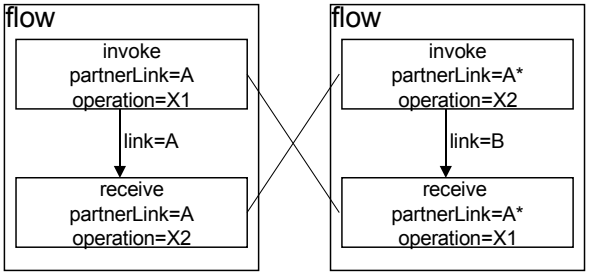

Abbildung 3.11.: Asynchrone <invoke>-Aktivitäten

3.3. Beziehungen zwischen BPEL-Prozessen

Von einem BPEL-Prozess können zu einem Zeitpunkt mehrere Instanzen existieren. Wir unterscheiden dabei zwischen *Instanzklassen*. Werden von einer Instanz eines BPEL-Prozesses P1, wobei hier nur diese Instanz existiert, mehrere Instanzen eines anderen BPEL-Prozesses P2 eingebunden, so sagen wir, dass sich P2 in der Instanzklasse n^1 befindet. Bindet nun jede Instanz von P2 mehrere Instanzen eines anderen BPEL-Prozesses P3 ein, so sagen wir, dass sich P3 in der Instanzklasse n^2 befindet.

Wir wollen zu jedem Zeitpunkt wissen, in welcher Instanzklasse sich ein BPEL-Prozess in diesem Moment befindet. Damit können wir bestimmen, ob bei einer Aktivitätsverbindung ein Prozess mit potentiell mehreren Instanzen eines anderen BPEL-Prozesses kommuniziert. Bevor wir den Ansatz näher betrachten, treffen wir einige Annahmen und schließen einige Fälle ganz aus.

Unter der „Schleifentiefe“ einer Aktivität verstehen wir die Anzahl der Schleifen, innerhalb derer sich eine Aktivität befindet. Diese Anzahl ergibt sich, indem wir dem Pfad von dieser Aktivität zur Wurzel folgen und dabei die Anzahl der so gefundenen Schleifen zählen.

Annahmen und ausgeschlossene Fälle

Zusätzlich zu den in den Abschnitten 3.1 und 3.2 getroffenen Annahmen und ausgeschlossenen Fällen treffen wir hier weitere Annahmen und schließen weitere Fälle aus.

Ausgeschlossen wird n:m-Kommunikation, wie sie etwa bei Peer-to-Peer-Anwendungen Verwendung findet.

Bindet ein Prozess einen neuen BPEL-Prozess ein, von dem dann eine Instanz kreiert wird, so treffen wir die Annahme, dass es nur zwei Möglichkeiten für eine Beziehung zwischen den beiden BPEL-Prozessen zu diesem Zeitpunkt gibt. Entweder sie stehen in einer 1:1-Beziehung zueinander, oder in einer 1:n-Beziehung, mit Instanzkreierung auf der n-Seite. Eine n:1-Beziehung, mit Instanzkreierung auf der 1er-Seite schließen wir aus. Abbildung 3.12 illustriert eine n:1-Beziehung, bei dem mehrere Instanzen eines Prozesses genau eine Instanz eines anderen Prozesses instanziieren. Dieser konkrete Fall wurde bereits im vorherigen Abschnitt 3.2 ausgeschlossen.

Zu einer <invoke>-Aktivität auf der Seite des Partnerprozesses dürfen nicht zwei passende <receive>-Aktivitäten sequentiell aufeinanderfolgen.

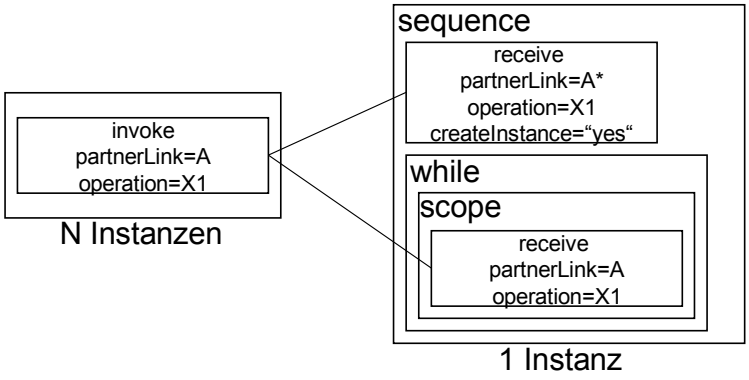

Abbildung 3.12.: n:1-Beziehung beim Einbinden eines neuen Prozesses

Zu beachten gilt, dass sich dies nur auf den Zeitpunkt bezieht, an dem von einem der teilnehmenden BPEL-Prozesse eine neue Instanz oder neue Instanzen generiert werden. Im sonstigen Verlauf kann es zu n:1-Beziehungen zwischen zwei BPEL-Prozessen kommen.

Da Fälle, in denen von einem BPEL-Prozess mehr als n Instanzen existieren, in der Praxis recht selten sind, wollen wir uns hier auf Fälle beschränken, in denen von einem BPEL-Prozess eine Instanz oder n Instanzen existieren. Den Fall, dass sich ein BPEL-Prozess in der Instanzklasse n^2 oder einer größeren Instanzklasse befindet berücksichtigen wir nicht. Dieser Fall kann beispielsweise dann auftreten, wenn jede Instanz eines BPEL-Prozesses, von dem n Instanzen existieren, wiederum n Instanzen eines anderen BPEL-Prozesses einbinden, von dem nun n^2 Instanzen existieren (siehe Abbildung 3.13).

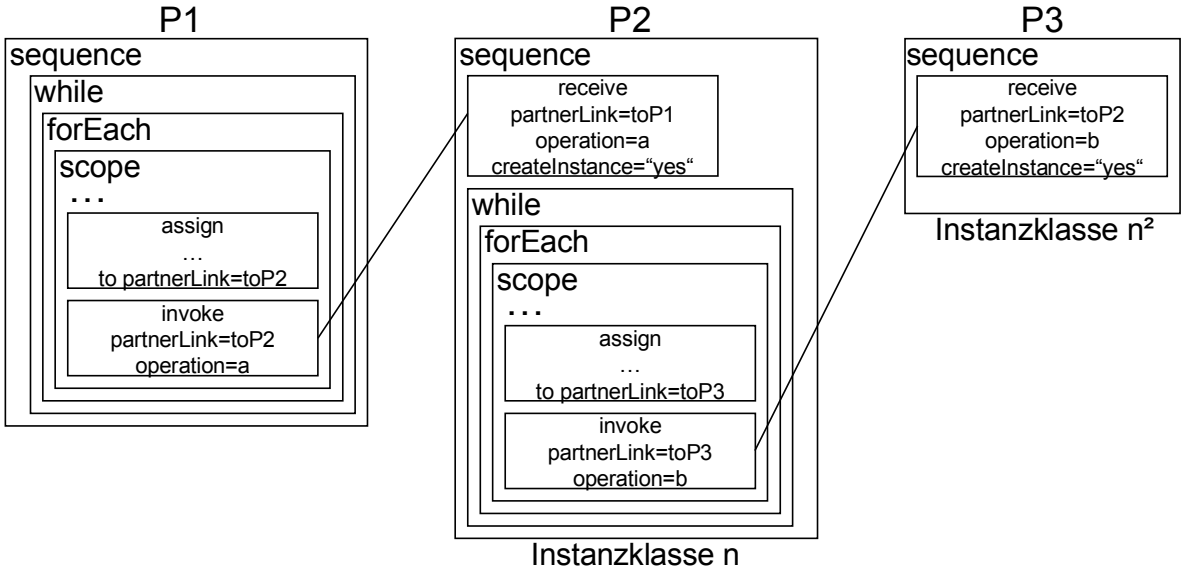

Abbildung 3.13.: Instanzklasse n^2

Folglich können von einem BPEL-Prozess nur eine Instanz oder n Instanzen existieren. Oder keine, wenn der BPEL-Prozess überhaupt nicht eingebunden wird.

Würden wir Fälle miteinbeziehen, in denen mehr als n Instanzen eines BPEL-Prozesses existieren können, so muss der in diesem Abschnitt vorgestellte Ansatz erweitert und teilweise abgeändert werden. Das Problem ist hierbei, zu erkennen, ob eine Schleife über n Instanzen eines Partnerprozesses iteriert oder über n^2. Um dies erkennen zu können, müssen wir das Weiterreichen von Endpunktreferenzen betrachten. Da wir dies bereits ausgeschlossen haben, besteht keine Möglichkeit, obiges Problem zu lösen. Aus diesem Grund beschränken wir uns auf die maximale Instanzklasse n^1.

Die Unterscheidung über Eigenschaften von Nachrichten (*Message Properties*) nehmen wir nicht vor, da wir Korrelationsmengen in diesem Zusammenhang nicht betrachten (siehe Abschnitt 3.1). Bindet ein Prozess n Instanzen eines BPEL-Prozesses ein, so kann dies ohne Korrelationsmenge nur wie folgt verwirklicht werden: Auf der Seite, auf der ein Prozess mit den n Instanzen eines anderen BPEL-Prozesses kommuniziert, wird jeweils die Endpunktreferenz mittels der <assign>-Aktivität in die „partnerRole“des entsprechenden *Partner Links* kopiert. Ein Beispiel findet sich in Abbildung 3.13.

Ausgeschlossen wird der Fall, dass bei einer Verzweigung in verschiedenen Zweigen in Bezug auf kommunizierender Aktivitäten dasselbe geschieht, nur einmal innerhalb einer Schleife, das andere mal nicht innerhalb einer Schleife (siehe Abbildung 3.14). Hier kann das Problem auftreten, dass wir unterschiedliche Beziehungen zwischen Prozessen erkennen, die in der tatsächlichen Ausführung nie vorkommen. In nachfolgenden Aktivitäten dieses Prozesses würde bei den zugehörigen Aktivitätsverbindungen bei der Bestimmung der Beziehung zwischen den beiden BPEL-Prozessen ein Folgefehler auftreten.

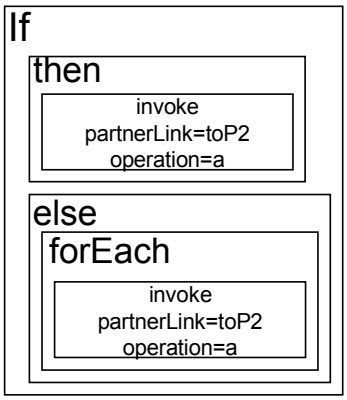

Abbildung 3.14.: Dieselben Aktivitäten in verschiedenen Zweigen in unterschiedlicher Schleifentiefe

Wir fordern, dass Schleifen bei verschiedenen Prozessen gleich oft durchlaufen werden. Nehmen wir an, dass zwei Aktivitäten miteinander kommunizieren. Die Aktivität beim einen BPEL-Prozess befindet sich in der Schleifentiefe eins, die Aktivität beim anderen BPEL-Prozess in der Schleifentiefe zwei. Im Beispiel in Abbildung 3.15 wird die eine Schleife bei Prozess P2 genauso oft durchlaufen wie die innere Schleife bei Prozess P1. Dasselbe gilt für Schleifen bei BPEL-Prozessen, die nicht direkt miteinander kommunizieren, wie in Abbildung 3.16 die Prozesse P1 und P3.

Falls zu einer instanzerzeugenden Aktivität im BPEL-Prozess P1 mehrere passende <invoke>-Aktivitäten beim Partnerprozess P2 existieren, dann fordern wir, dass sich P1 in allen möglichen

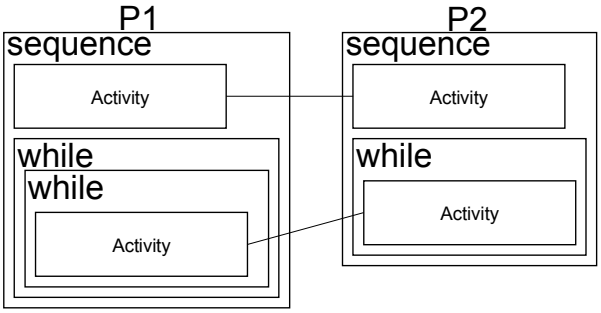

Abbildung 3.15.: Unterschiedliche Schleifen in verschiedenen Prozessen (I)

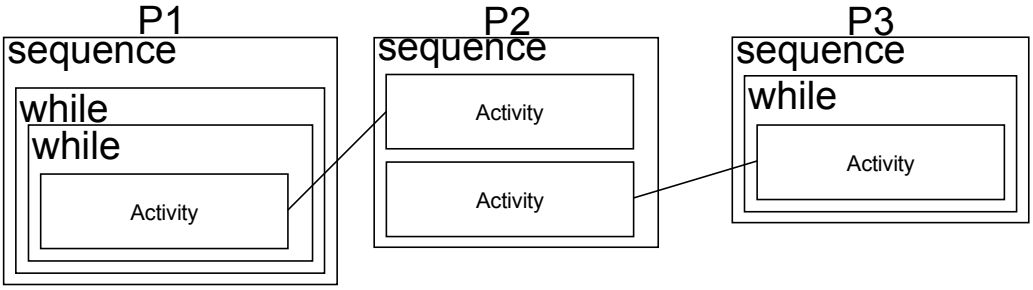

Abbildung 3.16.: Unterschiedliche Schleifen in verschiedenen Prozessen (II)

Fällen in derselben Instanzklasse befindet. Wir schließen Fälle aus, in denen diese Forderung nicht erfüllt ist.

Wir fordern, dass die äquivalenten, zueinander passenden Schleifen bei verschiedenen Prozessen in derselben Reihenfolge vorliegen.

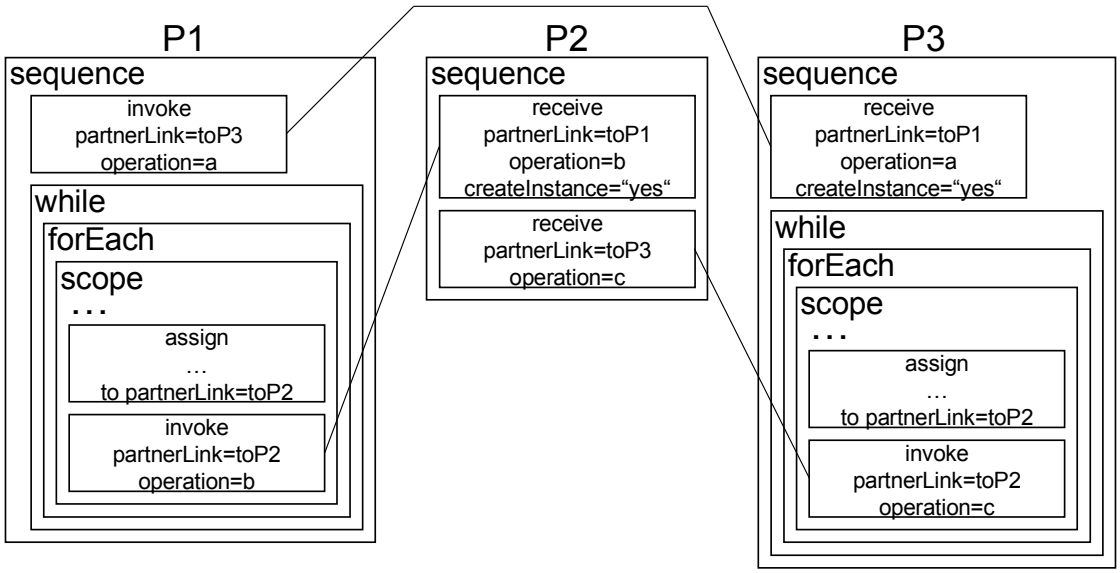

Abbildung 3.17.: Gleiche Reihenfolge der Schleifen

Im Beispiel in Abbildung 3.17 iterieren sowohl P1 als auch P3 über die n Instanzen des BPEL-Prozesses P2. Wegen der <while>-Schleife geschieht dies mehrmals. P1 bindet P2 ein. Wir fordern, dass die Reihenfolge der Schleifen bei P1 der Reihenfolge der Schleifen bei P3 entspricht. Fälle, in denen dies nicht der Fall ist, schließen wir aus, da wir ansonsten nicht einfach bestimmen können, welche der Schleifen die Kommunikation mit den n Instanzen eines Partnerprozesses ermöglicht.

Bindet ein BPEL-Prozess, von dem n Instanzen existieren, einen anderen BPEL-Prozess ein, und es herrscht zu diesem Zeitpunkt eine 1:1-Beziehung zwischen diesen BPEL-Prozessen, so gehen wir davon aus, dass sich der eingebundene BPEL-Prozess ebenfalls in der Instanzklasse n^1 befindet.

Betrachten wir den Fall, dass n Instanzen eines BPEL-Prozesses eingebunden werden, d. h. die Instanzen dieses Prozesses werden zu diesem Zeitpunkt erzeugt. Die zur instanzerzeugenden Aktivität passende <invoke>-Aktivität befindet sich in einer Schleifentiefe größer eins. Dann wissen wir, dass die erste Schleife, die wir auf dem Pfad von der <invoke>-Aktivität zur Wurzel finden, über die n Instanzen des neu eingebundenen BPEL-Prozesses iteriert (siehe Abbildung 3.17). Aufgrund der anderen Schleifen geschieht dies mehrmals. Wären die Schleifen vertauscht, so würde versucht werden, mehrere identische Instanzen zu generieren, was nicht möglich ist.

Benötigte Konstrukte

Zunächst benötigen wir die folgenden Variablen:

1. active[] ist ein Array von Booleans. Zu Beginn sind alle Werte „false“.
2. instances[] ist ein Array von Integern. Zu Beginn sind alle Werte mit dem Default-Wert $\perp$ belegt. $\perp$ bedeutet „undefiniert“.

In active[] wird für jeden BPEL-Prozess festgehalten, ob er zum aktuellen Zeitpunkt bereits aktiv ist. Es gibt drei Möglichkeiten, damit ein BPEL-Prozess aktiv wird:

1. Ein BPEL-Prozess wird von einem anderen Prozess eingebunden.
2. Ein BPEL-Prozess ist Initiator und wir bilden eine Aktivitätsverbindung, in der eine Aktivität des Initiators vorkommt.
3. Wir entfernen die instanzerzeugende Aktivität eines BPEL-Prozesses aus O, da der BPEL-Prozess, in dem die passende <invoke>-Aktivität stehen müsste, nicht gegeben ist.

In instances[] wird für jeden BPEL-Prozess festgehalten, in welcher Instanzklasse sich dieser zum aktuellen Zeitpunkt befindet. Eine „0“ steht dabei für die Instanzklasse n^0, eine „1“ für die Instanzklasse n^1. Diese beiden Arrays werden den Funktionen *FIND-ACTIVITIES* und *GENERATE-ACTIVITY-LINKS* aus dem vorherigen Abschnitt 3.2 als Parameter übergeben. Die Übergabe geschieht ebenfalls „by-value“. Der aktuelle Wert von instances[] und von active[] wird zur Laufzeit bestimmt.

Wir benötigen die folgende Funktion:

Definition 3.18: Die Funktion $s : \mathcal{A} \rightarrow \mathbb{N}$ gibt zu einer gegebenen Aktivität dessen Schleifentiefe zurück. $\mathcal{A}$ ist die Menge der Aktivitäten, wie sie in [MKL07] definiert wird.

Weiter benötigen wir zwei Matrizen B und D. Im Folgenden sei k die Anzahl der gegebenen BPEL-Prozesse.

Definition 3.19: D ist eine k×k-Matrix. In einem Feld in D wird vermerkt, bei welcher Differenz der Schleifentiefen der beiden an einer Aktivitätsverbindung beteiligten Aktivitäten die im entsprechenden Feld in Matrix B vermerkte Beziehung zwischen den beiden BPEL-Prozessen besteht. ⊥ wird verwendet, um „unbekannt“ auszudrücken.

Definition 3.20: B ist eine k×k-Matrix. In einem Feld in B wird vermerkt, welche Beziehung zwischen zwei BPEL-Prozessen herrschen kann. Eine „0“ steht für eine 1:1-Beziehung und eine „1“ für eine 1:n-Beziehung. Eine „-1“ steht für eine n:1-Beziehung. Diese Beziehung zwischen zwei BPEL-Prozessen besteht genau dann, wenn die Differenz der Schleifentiefen der beiden an einer Aktivitätsverbindung beteiligten Prozesse dem im entsprechenden Feld in Matrix D vermerkten Wert entspricht. Weicht die tatsächliche Differenz ab, so besteht zu diesem Zeitpunkt eine andere Beziehung zwischen den beiden BPEL-Prozessen. ⊥ wird verwendet, um „unbekannt“ auszudrücken.

In den beiden letzten Fällen bei der Bestimmung, ob ein BPEL-Prozess aktiv ist oder nicht, legen wir fest, dass von diesem BPEL-Prozess genau eine Instanz existiert. Das entsprechende Feld in instances[] wird mit „0“ belegt.

Seien X und Y zwei BPEL-Prozesse, x eine Aktivität von X und y die instanzerzeugende Aktivität von Y und kann mit x und y eine Aktivitätsverbindung gebildet werden. Dann wird im Feld D[X,Y] vermerkt, in welcher Schleifentiefe sich x befindet. Die Schleifentiefe von x kann direkt übernommen werden, da sich y nicht innerhalb einer Schleife befinden darf.

Finden wir innerhalb der ersten Schleife, die wir auf dem Pfad von x zur Wurzel finden, eine <assign>-Aktivität, bei dem in den von x verwendeten *Partner Link* eine Endpunktreferenz kopiert wird, dann tragen wir im Feld B[X,Y] eine „1“ ein, da vom BPEL-Prozess Y n Instanzen eingebunden werden, folglich eine 1:n-Beziehung besteht. Finden wir keine solche <assign>-Aktivität, so tragen wir eine „0“ ein, da zwischen BPEL-Prozess X und BPEL-Prozess Y eine 1:1-Beziehung besteht. Ein einmal in B oder D eingetragener Wert ändert sich nicht mehr. Dies ergibt sich aufgrund der getroffenen Einschränkungen.

Für B[Y,X] und D[Y,X] gilt jeweils B[Y,X] = -B[X,Y] und D[Y,X] = - D[X,Y].

Beispiel 3.2:
P1, P2 und P3 sind drei BPEL-Prozesse (siehe Abbildung 3.18).

P1 bindet P2 ein, P2 bindet P3 ein und die Einträge für beide Matrizen sind folgende:

	P1	P2	P3
P1	0	1	⊥
P2	-1	0	0
P3	⊥	0	0

Folglich bindet P1 n Instanzen von P2 ein, und jede Instanz von P2 bindet eine Instanz von P3 ein, so dass von P3 ebenfalls n Instanzen existieren. P1 ist dabei Initiator. Wir gehen davon aus, dass von einem Initiator immer genau eine Instanz existiert.

Wir können nun berechnen, in welchem Verhältnis P1 zu P3 stehen kann. Dies können wir tun, indem wir D[P1,P2] + D[P2,P3] berechnen, was nun D[P1,P3] entspricht (dieses Vorgehen

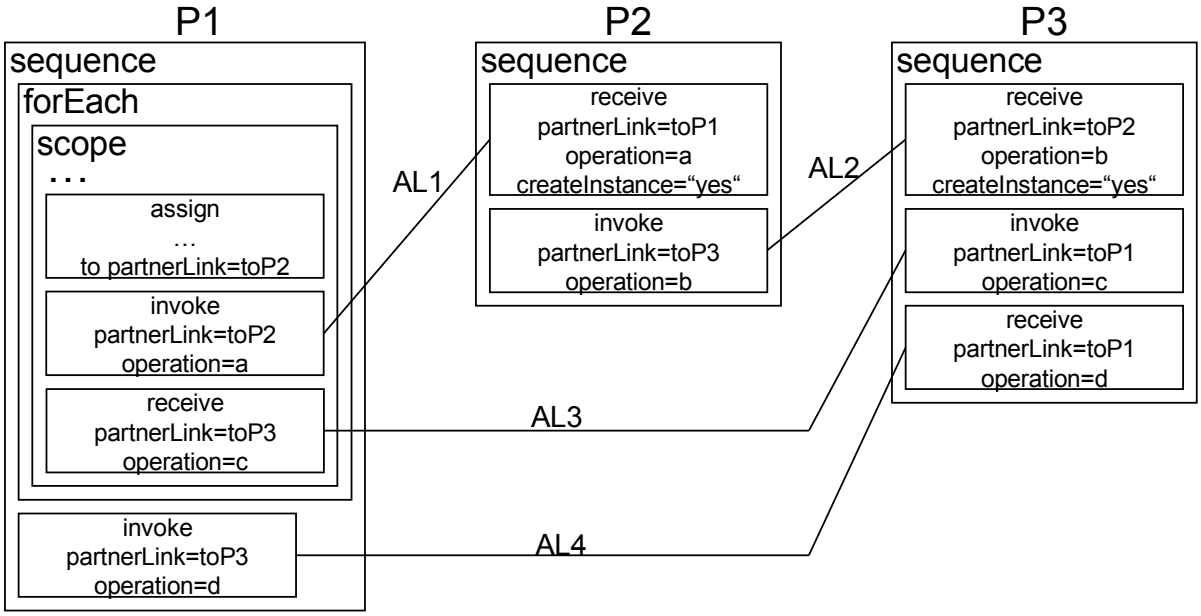

Abbildung 3.18.: Beispielprozesse, für die wir die aktuelle Instanzklasse zum Zeitpunkt einer Aktivitätsverbindung bestimmen (I)

bezeichnen wir als „Regel 1“). Für Matrix B gilt dasselbe. In diesem Beispiel ergeben sich die gleichen Werte. Die Einträge sind nun folgende:

	P1	P2	P3
P1	0	1	1
P2	-1	0	0
P3	-1	0	0

Das bedeutet P1 kann mit P3 in einer 1:n-Beziehung stehen, wegen B[P1,P3]. Wegen D[P1,P3] = 1 wissen wir, dass P1 genau dann in einer 1:n-Beziehung mit P3 steht, wenn für die zwei Aktivitäten a in P1 und c in P3, wobei diese beiden eine Aktivitätsverbindung bilden, gilt: s(a)-s(c) = 1, d. h. die Schleifentiefe der beiden Aktivitäten unterscheidet sich um 1. Gilt hingegen s(a)-s(c) = 0, dann können wir schließen, dass P1 nur noch mit einer Instanz von P3 kommuniziert.

Wir erkennen bei der Aktivitätsverbindung AL1, dass zwischen P1 und P2 eine 1:n-Beziehung besteht. Bei AL2 erkennen wir, dass zwischen P2 und P3 hier eine 1:1-Beziehung besteht. Anhand der in den Matrizen D und B errechneten Werte können wir bei AL3 erkennen, dass hier eine 1:n-Beziehung zwischen P1 und P3 besteht. In D[P1,P3] steht eine „1“, in B[P1,P3] steht ebenfalls eine „1“. Bei AL4 erkennen wir, dass die Schleifendifferenz „0“ ist und somit von dem Wert in D[P1,P3] abweicht. Wir können daraus schließen, dass zwischen P1 und P3 zu diesem Zeitpunkt keine 1:n-Beziehung mehr besteht, sondern eine 1:1-Beziehung. Dies können wir schließen, da bei P1 eine Schleife weniger vorhanden ist, als in dem Fall, in dem eine 1:n-Beziehung bestehen würde. Aufgrund der in diesem Abschnitt und in Abschnitt 3.2 getroffenen Annahmen („Gleiche Anzahl der Durchläufe der Schleifen bei verschiedenen Prozessen“ und „Verbot von sequentiellem Abarbeiten einer Schleife beim Partnerprozess“) können wir schließen, dass genau die Schleife fehlt, die die Kommunikation mit der Menge der Instanzen von P3 verwirklicht.

Daraus können wir ebenfalls schließen, dass sich P3 nicht mehr in der Instanzklasse n^1, sondern in der Instanzklasse n^0 befindet.

Eine weitere Möglichkeit, die Matrizen zu vervollständigen, benötigen wir für den Fall, dass ein BPEL-Prozess mehrere andere BPEL-Prozesse einbindet. Seien P1, P2 und P3 drei BPEL-Prozesse. Dabei bindet P1 sowohl P2 als auch P3 ein. P1 ist folglich Initiator. Wir wollen berechnen, in welchem Verhältnis P2 zu P3 stehen kann. Es gilt dabei: B[P1,P2] - B[P1,P3] = B[P3,P2] (dieses Vorgehen bezeichnen wir als „Regel 2“). Für Matrix D gilt entsprechend dasselbe.
Bindet P1 etwa n Instanzen von P2 ein (B[P1,P2]=1) und P1 bindet eine Instanz von P3 ein B[P1,P3]=0), dann ergibt sich für B[P3,P2] der Wert „1“, da zwischen P3 und P2 eine 1:n-Beziehung bestehen kann.

Definition 3.21: Es gelten die beiden Regeln B[P1,P2] + B[P2,P3] = B[P1,P3] und B[P1,P2] - B[P1,P3] = B[P3,P2]. Für Matrix D gilt dasselbe.

B und D dienen uns im Folgenden als Basis zur Bestimmung der Beziehung zwischen zwei BPEL-Prozessen zum Zeitpunkt eines Nachrichtenaustausches über eine Aktivitätsverbindung. Dies geschieht zur Laufzeit bei der Generierung neuer Aktivitätsverbindungen. Damit können wir bestimmen, ob von einem BPEL-Prozess weniger Instanzen existieren als zuvor, d. h. ob die Instanzklasse gewechselt wird, was im entsprechenden Feld in instances[] vermerkt wird.

Ergänzende Überprüfung bei der Generierung von Aktivitätsverbindungen

Wir müssen nun den Fall betrachten, dass die instanzerzeugende Aktivität eines BPEL-Prozesses in einer <flow>-Aktivität eingebettet ist. Hier kann der Fall auftreten, dass in dieser <flow>-Aktivität weitere Aktivitäten eingebettet sein können, die, aufgrund der vorhandenen Link-Semantik, gleichzeitig zu der instanzerzeugenden Aktivität aktiv werden können. Die Funktion *FIND-ACTIVITIES* findet in diesem Fall all diese Aktivitäten gleichzeitig mit der instanzerzeugenden Aktivität. Allerdings muss erst die instanzerzeugende Aktivität in einer Aktivitätsverbindung vorhanden sein, bevor mit diesen anderen Aktivitäten Aktivitätsverbindungen generiert werden dürfen.

Seien x und y die beiden Aktivitäten, mit denen eine neue Aktivitätsverbindung gebildet werden kann. x ist dabei die sendende und y die empfangende Aktivität. X und Y seien dabei die BPEL-Prozesse, zu denen x bzw. y gehören. Dann müssen wir zusätzlich überprüfen, ob wir die Aktivitätsverbindung zu diesem Zeitpunkt überhaupt bereits bilden dürfen. Dazu müssen wir prüfen, ob beide Prozesse bereits aktiv sind, d. h. ob „active[X]=false“ oder „active[Y]=false“gilt.

Ist X inaktiv (active[X]=false), dann überprüfen wir, ob eine zu X gehörende, instanzerzeugende Aktivität in O vorkommt. Ist dies nicht der Fall, dann schließen wir, dass X ein Initiator ist und bilden eine Aktivitätsverbindung. Ansonsten wird die Aktivitätsverbindung noch nicht gebildet.

Ist Y inaktiv, dann überprüfen wir, ob y eine instanzerzeugende Aktivität ist. Ist dies der Fall, so bilden wir eine Aktivitätsverbindung. Ansonsten darf die Aktivitätsverbindung noch nicht gebildet werden, da entweder eine instanzerzeugende Aktivität existiert oder Y ein Initiator ist, bei dem die erste Aktivität allerdings ein <invoke> sein muss. In diesen Fällen muss zuerst die Aktivitätsverbindung gebildet werden, die eine dieser anderen Aktivitäten enthält.

Eintragen der Werte in die neuen Konstrukte

Kann eine Aktivitätsverbindung gebildet werden, so unterscheiden wir zwischen drei Fällen:

1. X ist inaktiv, Y ist inaktiv:
   Wir setzen active[X] und active[Y] auf „true“. Wir setzen instances[X] = 0, da X ein Initiator ist und wir davon ausgehen, dass von einem Initiator genau eine Instanz existiert. Wir überprüfen nun, ob sich x innerhalb einer Schleife befindet, und ob wir innerhalb dieser Schleife eine <assign>-Aktivität finden, bei der eine Endpunktreferenz in den betreffenden *Partner Link* kopiert wird. Ist dies der Fall, so nehmen wir an, dass von Y n Instanzen existieren und setzen instances[Y] auf „1“, ansonsten auf „0“. Finden wir eine solche <assign>-Aktivität innerhalb dieser Schleife, aber von Y soll nur eine Instanz eingebunden werden, wobei dies mehrmals geschieht, so erkennen wir dies hier nicht. In diesem Fall darf sich die betreffende <assign>-Aktivität nicht innerhalb der Schleife befinden, in der sich auch x befindet.

   D[X,Y] setzen wir auf die Schleifentiefe von x. Dies können wir tun, da sich y nicht innerhalb einer Schleife befinden darf. B[X,Y] setzen wir auf den Wert von instances[Y]. Dies können wir tun, da von X genau eine Instanz existiert. Existieren nun n Instanzen von Y, so ist der entsprechende Wert „1“.

2. X ist aktiv, Y ist inaktiv:
   Wir setzen active[Y] auf „true“. In instances[X] steht, wieviele Instanzen des BPEL-Prozesses X aktuell existieren. Zwischen X und Y herrscht nun entweder eine 1:1-Beziehung oder eine 1:n-Beziehung. Entsprechend setzen wir instances[Y] entweder auf instances[X] oder auf instances[X]+1. D[X,Y] setzen wir auf die Schleifentiefe von x. B[X,Y] setzen wir auf „instances[Y] - instances[X]“.

3. X ist aktiv, Y ist aktiv:
   In diesem Fall müssen wir hier keine Anpassungen von instances[], B und D vornehmen.

Der Fall, dass X inaktiv, aber Y aktiv ist, kommt nicht vor, da X in diesem Fall ein Initiator sein müsste. Initiatoren haben allerdings nur einen Sinn, wenn sie andere Prozesse einbinden, die noch inaktiv sind.

Jedesmal, wenn wir in B, und somit entsprechend in D, einen Wert eintragen, müssen wir die Matrizen vervollständigen, entsprechend den beiden in der Definition 3.21 vorgestellten Regeln. Die Funktion *UPDATE-MATRIX* führt dies durch. Hat in einem Durchgang eine Änderung stattgefunden, so findet ein weiterer Durchgang statt. Dies geschieht solange, bis keine Änderung mehr stattgefunden hat. Übergeben wird der Funktion nichts, da B und D global definiert sind.

Erweiterung der Aktivitätsverbindungen

Sobald wir eine neue Aktivitätsverbindung generiert haben, wollen wir bestimmen, in welchen Instanzklassen sich die beiden beteiligten BPEL-Prozesse zu diesem Zeitpunkt befinden, d. h. instances[] anpassen. Dies benötigen wir, um bestimmen zu können, welche Beziehung zwischen den zwei beteiligten BPEL-Prozessen zu diesem Zeitpunkt besteht. Dieses Wissen benötigen wir später, bei der Generierung der Teilnehmer. Dort müssen wir wissen, ob eine Instanz eines BPEL-Prozesses mit einer Instanz eines anderen BPEL-Prozesses kommuniziert, oder mit einer

Menge von Instanzen. Hierzu benötigen wir die Matrizen B und D, sowie die bisher in den beiden Feldern in instances[] vermerkten Werte.

Seien x und y die Aktivitäten, die die Aktivitätsverbindung bilden. x ist die sendende Aktivität, y die empfangene Aktivität. X und Y sind entsprechend die zugehörigen BPEL-Prozesse. Wir berechnen nun die Abweichung in der Differenz der Schleifentiefen zu dem Fall, in dem die in B[X,Y] vermerkte Beziehung zwischen X und Y besteht. Sei Δ = D[X,Y]-(s(x)-s(y)). Über diese Abweichung können wir die aktuelle Beziehung *bez* zwischen X und Y bestimmen. Für diese gilt *bez* = B[X,Y]-Δ.

Wir wissen folgendes: Existiert bei der Ausführung einer Aktivität nur eine Instanz eines Prozessmodells, so können bei nachfolgenden Aktivitäten dieses BPEL-Prozesses nicht mehr als eine Instanz existieren.

Wir können daher drei Fälle unterscheiden:

1. instances[X] + *bez* < instances[Y]:
   Da es von X nicht plötzlich mehr Instanzen geben kann, muss instances[Y] auf „instances[X]+*bez*“ gesetzt werden.

2. instances[X] + *bez* > instances[Y]:
   Da es von Y nicht plötzlich mehr Instanzen geben kann, muss instances[X] auf „instances[Y]-*bez*“ gesetzt werden.

3. instances[X] + *bez* = instances[Y]:
   In diesem Fall müssen wir nichts ändern.

Jede Aktivitätsverbindung erweitern wir um die Informationen instX = instances[X], instY = instances[Y], s(x) und s(y). Diese benötigen wir im folgenden Abschnitt 3.4, in dem wir Teilnehmer generieren.

Wir haben somit für jede Aktivitätsverbindung festgehalten, in welchen Instanzklassen sich die beiden zugehörigen BPEL-Prozesse zu diesem Zeitpunkt befinden.

Das folgende Beispiel dient der Illustration des eben beschriebenen Vorgehens.

Beispiel 3.3:
Gegeben sind die drei BPEL-Prozesse P1, P2 und P3 aus Abbildung 3.19.

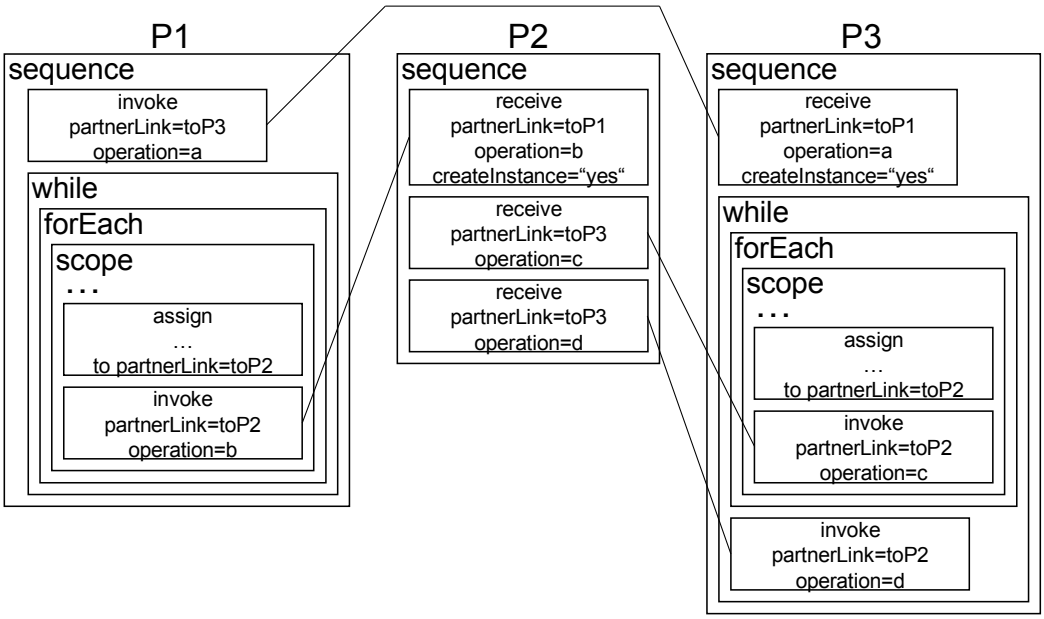

Abbildung 3.19.: Beispielprozesse, für die wir die aktuelle Instanzklasse zum Zeitpunkt einer Aktivitätsverbindung bestimmen (II)

P1 ist der Initiator. Von P1 existiert daher eine Instanz. P1 bindet P3 mit der Operation „a“ ein. Dabei herrscht zwischen P1 und P3 eine 1:1-Beziehung. Von P3 existiert daher hier eine Instanz. P1 bindet P2 mit der Operation „b“ ein. Wegen der <assign>-Aktivität, die eine Endpunktreferenz in die *partnerRole* des „toP2“-*Parter Links* kopiert, besteht zwischen P1 und P2 eine 1:n-Beziehung. P2 befindet sich somit in der Instanzklasse n^1. Der Unterschied zum vorherigen Beispiel 3.2 besteht darin, dass sich diese forEach-Schleife in einer weiteren Schleife befindet. Somit unterscheiden sich die in B und D eingetragenen Werte voneinander.

Nach dem Erzeugen der ersten beiden Aktivitätsverbindungen sieht Matrix D wie folgt aus:

	P1	P2	P3
P1	0	2	0
P2	-2	0	$\perp$
P3	0	$\perp$	0

Matrix B sieht entsprechend wie folgt aus:

	P1	P2	P3
P1	0	1	0
P2	-1	0	$\perp$
P3	0	$\perp$	0

Nach der Vervollständigung der beiden Matrizen mit Hilfe der Funktion *UPDATE-MATRIX* ergibt sich für D:

	P1	P2	P3
P1	0	2	0
P2	-2	0	-2
P3	0	2	0

und für B:

	P1	P2	P3
P1	0	1	0
P2	-1	0	-1
P3	0	1	0

Es gilt zu diesem Zeitpunkt: instances[P1] = 0, instances[P2] = 1 und instances[P3] = 0. Alle drei BPEL-Prozesse sind aktiv, d. h. alle Einträge in active[] sind „true“.

Die nächsten beiden Aktivitätsverbindungen betrachten wir genauer. Zunächst wird die Aktivitätsverbindung mit der Operation „c“ zwischen P2 und P3 gebildet. Damit P3 dies überhaupt tun kann, muss P3 von P1 die entsprechende Menge der Endpunktreferenzen erhalten haben. Dies lassen wir hier außer acht. Die Schleifentiefe der betreffenden Aktivität bei P3 ist „2“, bei P2 entsprechend „0“.

Es gilt daher:

$$\Delta = D[P3,P2]-(2-0) = 0.$$

Daher gilt:

$$bez = B[P3,P2]-\Delta = 1.$$

Und somit:

$$instances[P3]+bez = 1 = instances[P2].$$

Die Instanzklasse ändert sich folglich nicht. Die Aktivitätsverbindung wird nun um die weiter oben genannten Informationen ergänzt.

Die nächste Aktivitätsverbindung ist die, die mit den beiden Aktivitäten bei P2 und P3 gebildet werden kann, deren Operation „d“ ist. Die Schleifentiefe der Aktivität bei P3 ist „1“, die bei P2 entsprechend „0“.

Es gilt daher:

$$\Delta = D[P3,P2]-(1-0) = 1.$$

Daher gilt:

$$bez = B[P3,P2]-\Delta = 0.$$

Und somit:

$$\text{instances[P3]+bez} = 0 < \text{instances[P2]} = 1.$$

Somit folgt nun:

$$\text{instances[P2]} = \text{instances[P3]+bez} = 0.$$

Ab diesem Zeitpunkt existiert von P2 genau eine Instanz. Die Aktivitätsverbindung wird ebenfalls um die entsprechenden Informationen erweitert.

3.4. Generierung von Teilnehmermengen und Teilnehmerreferenzen

Im Folgenden wollen wir die benötigten Teilnehmermengen und Teilnehmerreferenzen generieren. In jeder Aktivitätsverbindung wird vermerkt, welche Teilnehmermengen oder Teilnehmerreferenzen senden, und welche Teilnehmerreferenzen empfangen.

Das Verfahren, Teilnehmermengen und Teilnehmerreferenzen zu generieren, könnten wir mit dem Finden von Aktivitätsverbindungen kombinieren. Aus Gründen der Übersichtlichkeit generieren wir die Teilnehmermengen und Teilnehmerreferenzen separat.

Bis jetzt wurde die Menge aller Aktivitätsverbindungen *ALinks* bestimmt. Jede Aktivitätsverbindung besteht aus einer sendenden und einer empfangenden Aktivität. Zusätzlich wurde in jeder Aktivitätsverbindung vermerkt, in welcher Instanzklasse sich die zu den beiden Aktivitäten gehörenden BPEL-Prozesse zu diesem Zeitpunkt befinden. Ebenso kennen wir die Schleifentiefen, innerhalb derer sich die beiden Aktivitäten befinden.

Zu jeder Teilnehmermenge wird eine zugehörige Teilnehmerreferenz gebildet. Diese Teilnehmerreferenz dient als Platzhalter für einen Teilnehmer der Teilnehmermenge. Im Folgenden sagen wir, dass eine solche Teilnehmerreferenz zu einer Teilnehmermenge „gehört“. Sonstige Teilnehmerreferenzen „gehören“ folglich zu keiner Teilnehmermenge.

Wie in Abschnitt 2.2.3 beschrieben, können Teilnehmermengen und Teilnehmerreferenzen auf bestimmte Gültigkeitsbereiche des Partnerprozesses beschränkt sein.

Im Folgenden lassen wir die <flow>-Aktivität zunächst außer acht. Im Anschluss betrachten wir gesondert, was bei der Miteinbeziehung der <flow>-Aktivität zu berücksichtigen ist.

Die Idee des Ansatzes ist Folgende: In einem gegebenen BPEL-Prozess suchen wir, wie beim Vorgehen bei der Funktion *FIND-ACTIVITIES* in Abschnitt 3.2, nach der nächsten aktiv werdenden kommunizierenden Aktivität. Allerdings bearbeiten wir hier jeden BPEL-Prozess separat und nicht alle gleichzeitig. Wir merken uns die zuletzt verwendete Teilnehmerreferenz. Finden wir eine kommunizierende Aktivität, so ermitteln wir die zugehörige Aktivitätsverbindung. Es können drei Fälle auftreten:

1. Wir müssen eine neue Teilnehmermenge oder Teilnehmerreferenz bilden.
2. Wir können eine bereits gebildete Teilnehmermenge oder Teilnehmerreferenz wiederverwenden.
3. Wir müssen die zuletzt verwendete Teilnehmerreferenz weiterverwenden.

Die Funktion, die diese Teilnehmermengen und Teilnehmerreferenzen generiert, nennen wir *GENERATE-PARTS*. *GENERATE-PARTS* wird für jeden BPEL-Prozess gesondert aufgerufen. Treffen wir auf eine <exit>-Aktivität, so wird die Funktion *GENERATE-PARTS* vorzeitig verlassen. Bei einer Verzweigung wird wie im Verfahren im Abschnitt 3.2 verfahren. Folglich können Teile eines Baumes mehrmals bearbeitet werden.

Wir merken uns stets die zuletzt verwendete Teilnehmerreferenz, da wir diese unter Umständen weiterverwenden müssen. Diese Teilnehmerreferenz bezeichnen wir im Folgenden als „aktuelle Teilnehmerreferenz“. Diese aktuelle Teilnehmerreferenz übergeben wir der Funktion *GENERATE-PARTS* als Parameter. Die Übergabe geschieht „by-value“.

In der Funktion *FIND-ACTIVITIES* vermerken wir für jeden Zweig einer Verzweigung, für den wir keine leere Menge an Aktivitätsverbindungen erhalten haben, welche kommunizierenden Aktivitäten dieses Prozesses zuvor bearbeitet wurden. Treffen wir in der Funktion *GENERATE-PARTS* auf eine Verzweigung, so wird ein Zweig nur bearbeitet, wenn wir dieselbe Menge an kommunizierenden Aktivitäten bearbeitet haben, wie zuvor für diesen Zweig vermerkt. Ist dies nicht der Fall, so wissen wir, dass dieser Zweig hier nicht bearbeitet werden darf.

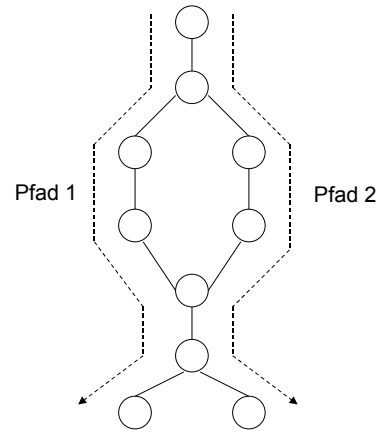

Abbildung 3.20.: Unterschiedliche mögliche Pfade eines BPEL-Prozesses

Abbildung 3.20 zeigt die Reihenfolge der Ausführung der kommunizierenden Aktivitäten eines BPEL-Prozesses. Möglich sind zwei Pfade. Alle anderen vorstellbaren Pfade sind nicht möglich. Durch die bereits bearbeiteten Aktivitäten wissen wir, auf welchem Pfad wir uns befinden. Somit wissen wir in der Funktion *GENERATE-PARTS* bei der zweiten Verzweigung, welchem Zweig wir für diesen Pfad folgen müssen.

Bestimmen wir die Teilnehmer zusammen mit den Aktivitätsverbindungen, so entfällt diese Überprüfung. Allerdings müssten dann Eintragungen in Aktivitätsverbindungen rückgängig gemacht werden, die in Zweigen auftreten, die nicht erfolgreich abgebrochen werden. Ebenso müssten in solchen Zweigen generierte Teilnehmermengen und Teilnehmerreferenzen verworfen werden.

Eine Aktivität kann in mehreren Aktivitätsverbindungen vorkommen, etwa aufgrund einer Verzweigung. Im Folgenden existiere zu einer Aktivität genau eine Aktivitätsverbindung. Nur die <invoke>-Aktivität darf in mehreren Aktivitätsverbindungen vorkommen. Einmal als sendende und beliebig oft als empfangende Aktivität. Im Anschluss betrachten wir den Fall, dass eine Aktivität in mehreren Aktivitätsverbindungen vorkommt. Ein Beispiel für eine Aktivität, die sich in mehreren Aktivitätsverbindungen befindet, findet sich in Abbildung 3.25.

Beispiel 3.4:
Betrachten wir P2 aus dem Beispiel in Abbildung 3.21.

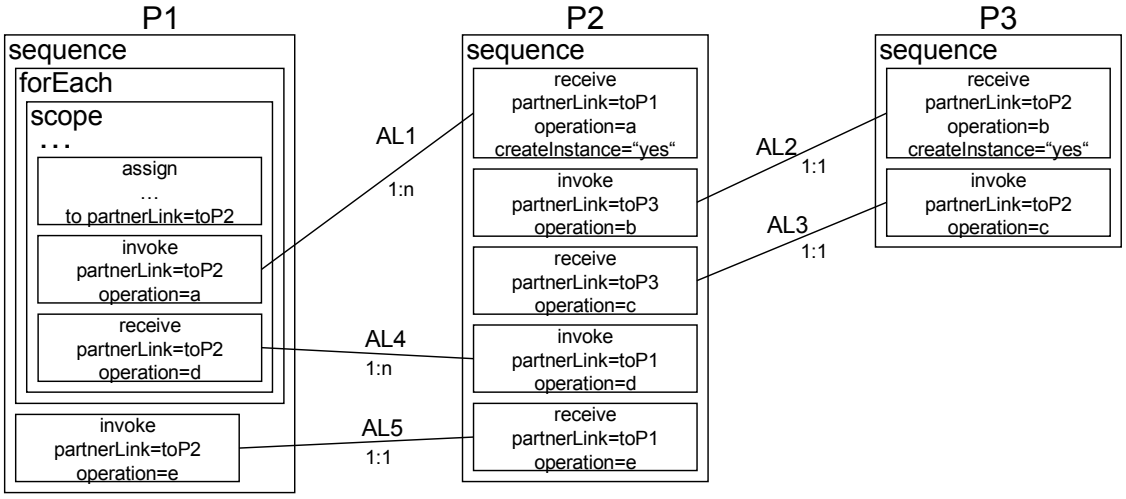

Abbildung 3.21.: Beispielprozesse, für die wir die Generierung von Teilnehmermengen und Teilnehmerreferenzen betrachten (I)

Für die erste kommunizierende Aktivität finden wir die Aktivitätsverbindung AL1. Hier sehen wir, dass sich P2 zu diesem Zeitpunkt in der Instanzklasse n^1 befinden muss. Folglich müssen wir eine Teilnehmermenge mit zugehöriger Teilnehmerreferenz bilden. Sei die Teilnehmermenge „P2_set_1“ und die zugehörige Teilnehmerreferenz „current_P2_1“. Als Empfänger in AL1 vermerken wir die Teilnehmerreferenz „current_P2_1“. Die aktuelle Teilnehmerreferenz ist „current_P2_1“.

Zur nächsten kommunizierenden Aktivität finden wir die Aktivitätsverbindung AL2. P2 befindet sich hier in der Instanzklasse n^1. Zwischen P2 und P3 besteht hier eine 1:1-Beziehung. Somit müssen wir die aktuelle Teilnehmerreferenz wiederverwenden. Als Sender in AL2 vermerken wir „current_P2_1“. Für AL3 gilt dasselbe, nur wird hier als Empfänger „current_P2_1“vermerkt.

Bei AL4 befindet sich P2 in der Instanzklasse n^1. Wir sehen, dass wir die alte Teilnehmermenge „P2_set_1“ wiederverwenden können, da bereits eine Teilnehmermenge existiert, die auf diesen Gültigkeitsbereich beim Partnerprozess beschränkt ist. Da wir die zugehörige Teilnehmerreferenz bereits verwendet haben, können wir in AL4 „current_P2_1“ als Sender vermerken.

Bei AL5 befindet sich P2 in der Instanzklasse n^0. Wir müssen folglich eine neue Teilnehmerreferenz bilden. Diese Teilnehmerreferenz sei „selected_P2_1“. Diese Teilnehmerreferenz vermerken wir in AL5 als Empfänger.

Interne Struktur für Teilnehmermengen und Teilnehmerreferenzen

Für Teilnehmermengen nutzen wir *partSet*- und für Teilnehmerreferenzen *part*-Objekte. Beide erben von der Klasse *genericParts*.

Ein *genericParts*-Objekt besitzt folgende Attribute:

- name
  Der Name eines *genericParts*-Objekts ist eindeutig.
- type
  type gibt an, von welchem Typ dieses *genericParts*-Objekt ist. Für jeden gegebenen BPEL-Prozess existiert ein eigener Typ. Somit kann ein *genericParts*-Objekt eindeutig einem BPEL-Prozess zugeordnet werden.
- Menge von Aktivitäten *L*
  Die Elemente dieser Menge sind <forEach>-, <repeatUntil>- oder <while>-Aktivitäten. In *L* wird festgehalten, auf welche Schleifen sich eine Teilnehmermenge oder eine Teilnehmerreferenz beziehen. Damit halten wir fest, welche Schleifen des Partnerprozesses die Kommunikation mit der Menge von Instanzen des betrachteten BPEL-Prozesses ermöglichen. Damit kennen wir ebenso die Gültigkeitsbereiche des Partnerprozesses, auf die eine Teilnehmermenge oder Teilnehmerreferenz beschränkt ist.
- Menge von NCNames *select_parts*
  Gibt an, welcher Teilnehmer welchen anderen Teilnehmer auswählt. Die Generierung dieser Menge wird in Abschnitt 3.7 beschrieben.

part-Objekte besitzen das folgende, weitere Attribut:

- set
  Verweist auf die Teilnehmermenge, zu der diese Teilnehmerreferenz gehört. Gehört eine Teilnehmerreferenz nicht zu einer Teilnehmermenge, so wird auf keine Teilnehmermenge verwiesen.

partSet-Objekte besitzen die folgenden, zusätzlichen Attribute:

- part_ref
  Verweist auf die zur Teilnehmermenge gehörende Teilnehmerreferenz. Wie anfangs beschrieben, generieren wir zu jeder Teilnehmermenge eine Teilnehmerreferenz, die als Repräsentant dieser Menge dient. Diese Teilnehmerreferenz vermerken wir hier.
- Boolean bound
  Gibt an, ob die Teilnehmermenge bereits an die in part_ref angegebene Teilnehmerreferenz gebunden wurde.
- Menge von *partSet*-Objekten *part_sets*
  Hier wird auf die Teilnehmermengen verwiesen, die Teilmengen dieser Teilnehmermenge sind.
- Menge von Aktivitäten *initial_acts*
  Hier wird festgehalten welche Aktivität beim ersten Auftreten der aktuellen Teilnehmermenge aktuell betrachtet wurde. Im Falle einer Verzweigung können dies mehrere Aktivitäten sein. Dies benötigen wir in Abschnitt 3.7.

Für das „bound“-Attribut gilt folgendes zu beachten: Verzweigt sich der betrachtete BPEL-Prozess, so rufen wir die Funktion *GENERATE-PARTS* für jeden Zweig rekursiv auf. Eine in einem Zweig neu generierte Teilnehmermenge kann in einem anderen Zweig ebenfalls Verwendung finden. Bei den neu erhaltenen, zum Zeitpunkt der Verzweigung noch nicht verwendeten *partSet*-Objekten müssen wir die Werte der „bound“-Attribute auf den Wert „false“ setzen. Es kann vorkommen, dass eine Teilnehmermenge in verschiedenen Zweigen der Verzweigung bei verschiedenen Aktivitätsverbindungen auf die zugehörige Teilnehmerreferenz gebunden werden muss.

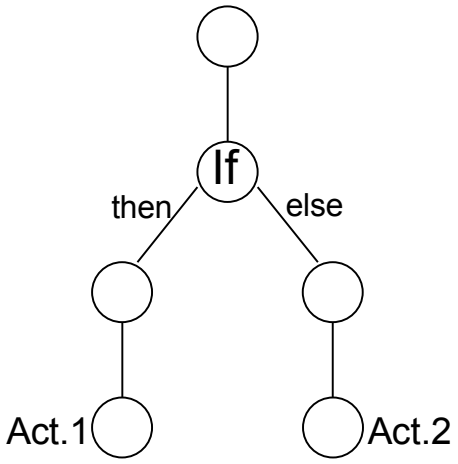

Abbildung 3.22.: Binden einer Teilnehmermenge auf eine Teilnehmerreferenz in verschiedenen Ästen

Beim Beispiel in Abbildung 3.22 wird beim Verfolgen des ersten Zweiges die Teilnehmermenge „example_set“ erstmals bei der zur Aktivität „Act.1“ gehörenden Aktivitätsverbindung verwendet. Bei der Verfolgung des zweiten Zweiges kann dieselbe Teilnehmermenge wiederverwendet werden. Hier tritt sie erstmalig in der zu „Act.2“ gehörenden Aktivitätsverbindung auf. Da „Act.1“ und „Act.2“ in verschiedenen Zweigen liegen, muss in beiden Fällen die Teilnehmermenge an die zugehörige Teilnehmerreferenz gebunden werden. In „initial_acts“ des entsprechenden „partSet“-Objekts wird sowohl „Act.1“ als auch „Act.2“ eingefügt.

Das „type“-Attribut erhält als Wert den NCName des betreffenden Prozesses.

Bei der Generierung eines *part*- oder *partSet*-Objekts wird das „name“-Attribut mit einem eindeutigen Wert belegt. Für Teilnehmermengen ergibt sich dieser aus „Name des betreffenden Prozesses + _set + _ + Zählerwert“. Für die zugehörige Teilnehmerreferenzen ergibt sich dieser Wert aus „current_ + Name des betreffenden Prozesses + _ + Zählerwert“. Existiert von einem BPEL-Prozess zu einem Zeitpunkt nur eine Instanz, so wird eine einfache Teilnehmerreferenz gebildet, die zu keiner Teilnehmermenge gehört. In diesem Fall setzt sich der Wert des „name“-Attributs aus „selected_ + Name des betreffenden Prozesses + _ + Zählerwert“ zusammen, da aus einer Teilnehmermenge ein Teilnehmer ausgewählt wurde. Allerdings vermerken wir hier, welche Teilnehmermenge direkt zuvor verwendet wurde. Dies benötigen wir, da wir bei der *Participant Topology* diese Teilnehmerreferenz in diese Teilnehmermenge einbetten wollen. Damit wird deutlich, von welcher Teilnehmermenge ein Teilnehmer ausgewählt wurde. Existiert von einem BPEL-Prozess zu keinem Zeitpunkt mehr als eine Instanz, so erhält das „name“-Attribut dieser Teilnehmerreferenz den Namen des Prozesses als Wert. Für jeden BPEL-Prozess läuft ein Zähler mit, der festhält, wieviele Teilnehmermengen für diesen BPEL-Prozess, und

damit für welchen Teilnehmertyp, bereits generiert wurden. Denselben Zähler können wir für die zugehörigen Teilnehmerreferenzen verwenden. Für Teilnehmerreferenzen, die zu keiner Teilnehmermenge gehören, läuft ein separater Zähler mit.

Definition 3.22: *PS* ist die Menge der bereits generierten *partSet*-Objekte für einen BPEL-Prozess.

Wird eine neue Teilnehmermenge - und damit ein neues *partSet*-Objekt - generiert, so wird das neu generierte *partSet*-Objekt in die Menge *PS* eingefügt.

Kommt es zu einer Verzweigung, so generieren wir eine neue Menge *PStemp*. In *PStemp* fügen wir jedes Element aus *PS* ein, dessen Attribut „bound“ den Wert „true“ besitzt. Somit kennen wir die Teilnehmermengen, die bereits benutzt wurden, bevor es zur Verzweigung kommt. Für jeden Zweig der Verzweigung rufen wir die Funktion *GENERATE-PARTS* rekursiv auf. Nach dem Aufruf für einen Zweig setzen wir in allen Elementen in *PS*, die nicht in *PStemp* vorhanden sind, das „bound“-Attribut auf den Wert „false“. Damit stellen wir sicher, dass Teilnehmermengen, die nach dem Zeitpunkt der Verzweigung erstmals verwendet werden, bei der Verfolgung eines anderen Zweiges erneut auf ihre Teilnehmerreferenz gebunden werden müssen (siehe Abbildung 3.22).

Beim ersten Aufruf der *GENERATE-PARTS*-Funktion für einen BPEL-Prozess ist die Menge *PS* leer. Ist diese Funktion fertig, so enthält *PS* die Menge aller benötigten Teilnehmermengen des betrachteten Prozesses. *PS* entspricht dann einem Element der Menge *PSets*.

Definition 3.23: *PSets* ist die Menge der Mengen aller generierten *partSet*-Objekte. Ein Element aus *PSets* ist eindeutig einem BPEL-Prozess zugeordnet.

Für Teilnehmerreferenzen, die zu keiner Teilnehmermenge gehören, wird für jeden BPEL-Prozess eine eigene Menge angelegt.

Definition 3.24: *PRefs* ist die Menge der generierten *part*-Objekte, die Teilnehmerreferenzen repräsentieren, die nicht Repräsentanten einer Teilnehmermenge sind. Ein Element aus *PRefs* ist eindeutig einem BPEL-Prozess zugeordnet.

Zu jeder dieser Teilnehmerreferenzen wird vermerkt, welche Teilnehmermenge direkt zuvor benutzt wurde. Dies benötigen wir, da in der *Participant Topology* diese Teilnehmerreferenz in dieser Teilnehmermenge enthalten sein soll. Somit wird festgehalten, aus welcher Teilnehmermenge ein bestimmter Teilnehmer ausgewählt wird. Wurde zuvor keine Teilnehmermenge verwendet, so wird dies vermerkt.

Finden einer Schleife, die die Kommunikation mit mehreren Instanzen eines Partnerprozesses ermöglicht

Um herauszufinden, ob eine bereits generierte Teilnehmermenge wiederverwendet werden kann, wollen wir ermitteln, auf welche Schleife beim Partnerprozess sich eine Teilnehmermenge bezieht. Damit kennen wir ebenso den Gültigkeitsbereich des Partnerprozesses, auf den diese Teilnehmermenge und die zugehörige Teilnehmerreferenz beschränkt sind.

In einer Aktivitätsverbindung können wir ablesen, in welchen Instanzklassen sich die beteiligten BPEL-Prozesse zu diesem Zeitpunkt befinden. Befindet sich der betrachtete BPEL-Prozess in einer höheren Instanzklasse als der Partnerprozess, so wissen wir, dass beim Partnerprozess eine Schleife existieren muss, die einer Instanz dieses BPEL-Prozesses ermöglicht, mit einer Menge von Instanzen des betrachteten BPEL-Prozesses zu kommunizieren. Dies können wir schließen,

da wir das sequentielle Kommunizieren mit verschiedenen Instanzen eines Partnerprozesses ausgeschlossen haben, da in einem solchen Fall die Korrelationsmengen betrachtet werden müssten, was in Abschnitt 3.1 ausgeschlossen wurde.

Wir wissen aus unseren Annahmen in Abschnitt 3.2, dass dies in diesem Fall eine n:1-Beziehung sein muss, da eine höhere Instanzklasse als n^1 nicht vorgesehen ist.

Sei im Folgenden x die aktuell betrachtete Aktivität im aktuell betrachteten BPEL-Prozess. y ist die andere Aktivität in der Aktivitätsverbindung, in der x vorkommt. X und Y sind die zugehörigen BPEL-Prozesse.

Aufgrund der im Abschnitt 3.3 getroffenen Annahmen („Gleichheit der Durchläufe von Schleifen bei verschiedenen Prozessen“ und „Gleiche Reihenfolge der Schleifen bei verschiedenen Prozessen“), können wir zwischen zwei Fällen unterscheiden:

1. s(x) < s(y):
   Wir suchen auf dem Pfad von y zur Wurzel nach der „s(x)+1“-ten Schleife. (siehe Beispiel 3.5)

2. s(x) $\geq$ s(y):
   Wir suchen nach der letzten Schleife auf dem Pfad von y zur Wurzel. (siehe Beispiel 3.6)

Die folgenden beiden Beispiele dienen zur Illustration des oben beschriebenen Vorgehens.

Beispiel 3.5:
Unser aktuell betrachteter BPEL-Prozess ist der BPEL-Prozess X aus Abbildung 3.23. Der Aktivitätsverbindung AL1 können wir entnehmen, dass zwischen X und Y eine n:1-Beziehung besteht. Wir wissen daher, dass wir eine Teilnehmermenge verwenden müssen.

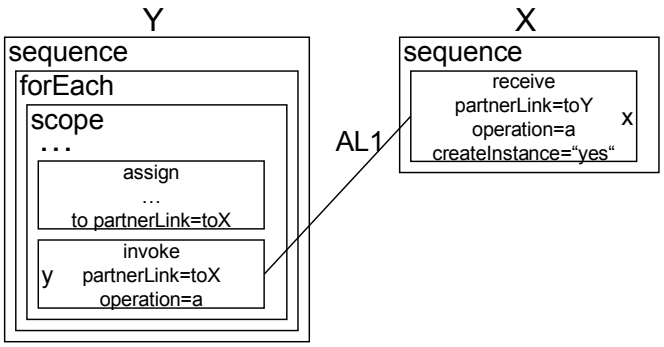

Abbildung 3.23.: Beispielprozesse zur Illustration der Bestimmung der Schleifen, die die Kommunikation mit einer Teilnehmermenge ermöglichen, betrachten (I)

Muss eine Teilnehmermenge verwendet werden, so überprüfen wir, welche Schleife des Partnerprozesses die Kommunikation mit den n Instanzen des betrachteten BPEL-Prozesses ermöglicht.

Nach der obigen Fallunterscheidung suchen wir von y aus Richtung Wurzel nach der „s(x)+1“-ten Schleife. Da s(x) „0“ist, suchen wir folglich nach der ersten Schleife, die wir finden. Gefunden wird die <forEach>-Aktivität bei Y.

Beispiel 3.6:
Unser aktuell betrachteter BPEL-Prozess ist der BPEL-Prozess X aus Abbildung 3.24.

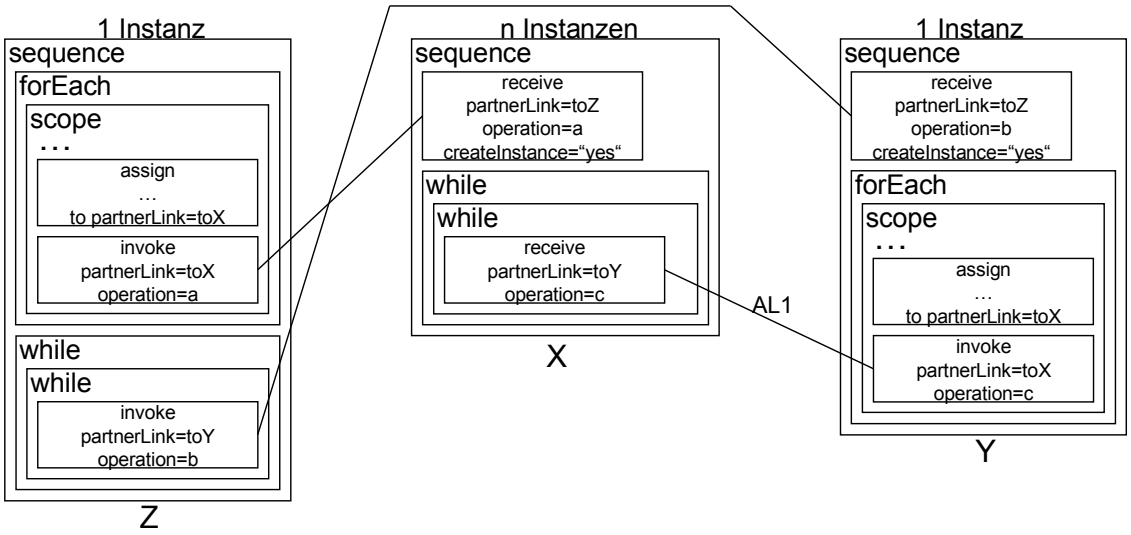

Abbildung 3.24.: Beispielprozesse zur Illustration der Bestimmung der Schleifen, die die Kommunikation mit einer Teilnehmermenge ermöglichen, betrachten (II)

Der Aktivitätsverbindung AL1 können wir entnehmen, dass zwischen X und Y eine n:1-Beziehung herrschen muss. Wir wissen daher, dass wir eine Teilnehmermenge verwenden müssen.

Der obigen Fallunterscheidung folgend suchen wir von der betreffenden Aktivität bei Y aus Richtung Wurzel nach der letzten Schleife auf diesem Pfad. Gefunden wird die <forEach>-Aktivität bei Y.

Erweiterung der Aktivitätsverbindungen

Jede Aktivitätsverbindung wird um folgende Informationen erweitert:

- *SEND_PARTs*
  Liste von *part*- und *partSet*-Objekten. Diese repräsentieren die Teilnehmermengen und Teilnehmerreferenzen, die in dieser Aktivitätsverbindung als Sender fungieren.

- *REC_PARTs*
  Liste von *part*-Objekten. Diese repräsentieren die Teilnehmerreferenzen, die in dieser Aktivitätsverbindung als Empfänger fungieren. Diese werden in einem späteren Schritt zu einem Empfänger zusammengefasst.

- bindSenderTo
  Wird eine Teilnehmermenge auf die zugehörige Teilnehmerreferenz gebunden, so wird hier auf diese Teilnehmerreferenz verwiesen.

Wenn wir eine Teilnehmerreferenz als Sender oder Empfänger vermerken, so bedeutet dies, dass wir das zu dieser Teilnehmerreferenz gehörende *part*-Objekt in *SEND_PARTs* bzw. *REC_PARTs* einfügen. Für Teilnehmermengen gilt entsprechend dasselbe.

Für die Sender und Empfänger benötigen wir eine Liste, da der Fall auftreten kann, dass wir aufgrund von Verzweigungen mehrere verschiedene mögliche Teilnehmerreferenzen als Sender haben. Da wir hier noch keine Maßnahmen dagegen treffen, können im Anschluss in einer Aktivitätsverbindung mehrere Teilnehmerreferenzen als Empfänger fungieren.

Vorbetrachtung eines BPEL-Prozesses

Wir können zwischen drei Fällen unterscheiden:

1. Der betrachtete BPEL-Prozess ist ein Initiator:
   Von diesem BPEL-Prozess existiert genau eine Instanz. Wir benötigen daher keine Teilnehmermenge. Es wird eine einzelne Teilnehmerreferenz gebildet, die in allen Aktivitätsverbindungen angegeben wird, in denen eine Aktivität des betrachteten BPEL-Prozesses vorkommt. Ist die zu diesem BPEL-Prozess gehörende Aktivität die sendende Aktivität, so wird die Teilnehmerreferenz als Sender in dieser Aktivitätsverbindung vermerkt. Ist sie die empfangende Aktivität, so wird die Teilnehmerreferenz als Empfänger vermerkt.

2. Zur ersten kommunizierenden Aktivität finden wir keine zugehörige Aktivitätsverbindung:
   Diesen Fall behandeln wir analog zum ersten Fall, da von diesem BPEL-Prozess ebenfalls genau eine Instanz existiert.

3. Zur ersten kommunizierenden Aktivität finden wir eine zugehörige Aktivitätsverbindung:
   Aus der Aktivitätsverbindung können wir ablesen, in welcher Instanzklasse sich der betrachtete BPEL-Prozess zu diesem Zeitpunkt befindet. Befindet dieser sich in der Instanzklasse n^0, so verfahren wir analog zum ersten Fall. Ansonsten muss eine Teilnehmermenge gebildet werden.

Generieren von Teilnehmermengen und Teilnehmerreferenzen

Eine neu gebildete Teilnehmermenge wird in jedem Fall in die Menge *PS* eingefügt.

Sei im Folgenden die sendende Aktivität einer Aktivitätsverbindung x und die empfangende Aktivität y. X und Y sind die dazugehörigen Prozesse.

Es können zwei Fälle auftreten:

1. Die aktuell betrachtete kommunizierende Aktivität ist eine empfangende Aktivität.

2. Die aktuell betrachtete kommunizierende Aktivität ist eine sendende Aktivität.

Betrachten wir zunächst den ersten Fall. Folglich betrachten wir die Aktivität y einer Aktivitätsverbindung.

Aus einer zugehörigen Aktivitätsverbindung können wir mittels den Attributen „instX“ und „instY“ ablesen, in welchen Instanzklassen sich die beiden BPEL-Prozesse X und Y zu diesem Zeitpunkt befinden. Es gibt folgende Möglichkeiten:

- instX > instY:
  In diesem Fall existiert von Y genau eine Instanz. Wir können daher eine Teilnehmerreferenz, die zu keiner Teilnehmermenge gehört, verwenden. Haben direkt zuvor mehrere Instanzen von Y existiert, so suchen wir in der entsprechenden Menge *PRefs* nach einer Teilnehmerreferenz, zu der wir die zuvor verwendete Teilnehmermenge vermerkt haben. Finden wir eine solche, so können wir diese hier verwenden. Existiert keine solche, so wird sie generiert und in *PRefs* eingefügt.

  Wurde direkt zuvor bereits eine Teilnehmerreferenz verwendet, die zu keiner Teilnehmermenge gehört, so können wir diese hier wiederverwenden.

- instX = instY:
  Hier sind zwei Fälle möglich:

  1. instY = 0:
     In diesem Fall existiert von Y genau eine Instanz. Wie zuvor suchen wir entweder in der entsprechenden Menge *PRefs* nach einer passenden Teilnehmerreferenz, generieren eine neue oder können die zuvor verwendetet Teilnehmerreferenz weiterverwenden.

  2. instY = 1:
     Hier vermerken wir die aktuelle Teilnehmerreferenz als Empfänger in der Aktivitätsverbindung. Existiert noch keine aktuelle Teilnehmerreferenz, so wird eine Teilnehmermenge generiert. Wie bereits erwähnt, erzeugen wir mit einer neuen Teilnehmermenge direkt eine zugehörige Teilnehmerreferenz. In der Aktivitätsverbindung vermerken wir diese Teilnehmerreferenz als Empfänger. Die aktuelle Teilnehmerreferenz ist nun diese neu generierte Teilnehmerreferenz.

- instX < instY:
  In diesem Fall wissen wir, dass von Y eine Menge von Instanzen existieren. Ebenso wissen wir, dass bei X eine Schleife existiert, die die Kommunikation mit der Menge von Instanzen von Y ermöglicht. Nach dem vorgestellten Verfahren können wir diese Schleife bestimmen. Sei diese Schleife *loop*. Im Folgenden betrachten wir *loop* als einelementige Menge, die die gefundene Schleife beinhaltet.

  Wir können zwischen drei Fällen unterscheiden:

  1. In der zur aktuellen Teilnehmerreferenz gehörenden Teilnehmermenge suchen wir in der Menge *part_sets* dieses Objekts nach einem *partSet*-Objekt, für dessen Menge L „$L = loop$“ gilt. Finden wir ein solches *partSet*-Objekt, so können wir dieses weiterverwenden, da somit bereits eine Teilnehmermenge existiert, die auf den entsprechenden Gültigkeitsbereich beim Partnerprozess beschränkt ist. Die aktuelle Teilnehmerreferenz ist nun die zu dieser Teilnehmermenge gehörende Teilnehmerreferenz. Hat das „bound“-Attribut dieses *partSet*-Objekts den Wert „false“, so setzen wir dieses Attribut auf den Wert „true“und fügen die aktuell betrachtete kommunizierende Aktivität in die Menge *initial_acts* dieses *partSet*-Objekts ein.

Somit haben wir in der zuletzt verwendeten Teilnehmermenge eine bereits existierende Teilmenge gefunden, die wiederverwendet werden kann. Finden wir keine, so greift der nachfolgende Fall.

2. Finden wir in der Menge *PS* ein *partSet*-Objekt, für dessen Menge L „$L = loop$“ gilt und dessen „bound“-Attribut den Wert „true“ besitzt, so können wir diese Teilnehmermenge weiterverwenden. Aktuelle Teilnehmerreferenz ist nun die zu dieser Teilnehmermenge gehörende Teilnehmerreferenz.
   Angenommen, der Wert des „bound“-Attributs ist „false“. Dann können wir diese Teilnehmermenge nicht wiederverwenden, da sie erstmals in einem anderen Zweig einer Verzweigung verwendet wurde und nicht Teilmenge unserer aktuellen Teilnehmermenge ist. Da eine Teilnehmermenge nur Teilmenge einer einzigen Teilnehmermenge sein darf, müssen wir in diesem Fall eine neue Teilnehmermenge bilden. Ansonsten wäre dieselbe Teilnehmermenge Teilmenge von verschiedenen Teilnehmermengen. Somit würden in der *Participant Topology* von BPEL4Chor mehrere identische <participantSet>-Konstrukte vorkommen, was wir hiermit vermeiden. Somit können insgesamt mehrere Teilnehmermengen existieren, die auf denselben Gültigkeitsbereich beim Partnerprozess beschränkt sind.

3. Finden wir kein wiederverwendbares *partSet*-Objekt, oder existiert noch keine Teilnehmermenge für den betrachteten BPEL-Prozess, so müssen wir eine neue Teilnehmermenge generieren. Diese neue Teilnehmermenge fügen wir in die Menge *part_sets* der zur aktuellen Teilnehmerreferenz gehörenden Teilnehmermenge ein. Ebenso fügen wir diese Teilnehmermenge in *PS* ein. In die Menge L des zugehörigen *partSet*-Objekts fügen wir *loop* ein. Ebenso in die Menge L des zugehörigen *part*-Objekts. In *initial_acts* fügen wir die aktuell betrachtete kommunizierende Aktivität ein. Das „bound“-Attribut erhält den Wert „true“. Die aktuelle Teilnehmerreferenz ist nun die zur neu generierten Teilnehmermenge gehörende Teilnehmerreferenz.

In allen drei Fällen vermerken wir die nun aktuelle Teilnehmerreferenz als Empfänger in der Aktivitätsverbindung.

Betrachten wir nun den Fall, dass die aktuell betrachtete kommunizierende Aktivität eine sendende Aktivität ist. Folglich betrachten wir die Aktivität x einer Aktivitätsverbindung.

Wir unterscheiden zwischen den folgenden Fällen:

- instX $\leq$ instY
  In diesem Fall vermerken wir die aktuelle Teilnehmerreferenz als Sender in der Aktivitätsverbindung. Es kann hier kein Fall auftreten, in dem wir eine neue Teilnehmerreferenz bilden, da aus einer Teilnehmermenge kein Teilnehmer ausgewählt wird, indem mehrere Instanzen des betrachteten BPEL-Prozesses senden, aber nur für eine Instanz dieser Menge ein anderer Prozess diese Nachricht empfängt.

- instX > instY
  In diesem Fall wissen wir, dass von X eine Menge von Instanzen existiert. Wie oben suchen wir mittels der gefundenen Schleife *loop* in der Menge *part_sets* der zur aktuellen Teilnehmerreferenz gehörenden Teilnehmermenge nach einem passenden *partSet*-Objekt. Wir unterscheiden zwischen vier Fällen:

1. Finden wir ein solches Objekt, und der Wert des „bound“-Attributs dieses Objekts ist „true“, so verwenden wir die zu dieser Teilnehmermenge gehörende Teilnehmerreferenz als aktuelle Teilnehmerreferenz. Als Sender in der Aktivitätsverbindung geben wir diese Teilnehmerreferenz an.

2. Hat das „bound“-Attribut des gefundenen *partSet*-Objekts den Wert „false“, dann bedeutet dies, dass wir diese Teilnehmermenge erneut an ihre zugehörige Teilnehmerreferenz binden müssen. Als Sender in der Aktivitätsverbindung geben wir die Teilnehmermenge an. „bindSenderTo“ der Aktivitätsverbindung verweist auf die zur Teilnehmermenge gehörende Teilnehmerreferenz. Das „bound“-Attribut des *partSet*-Objekts erhält nun den Wert „true“. Die aktuell betrachtete Aktivität fügen wir in die Menge *initial_acts* des *partSet*-Objekts ein. Aktuelle Teilnehmerreferenz ist nun die zur Teilnehmermenge gehörende Teilnehmerreferenz.

3. Finden wir kein solches *partSet*-Objekt, so suchen wir nach einem passenden *partSet*-Objekt in der Menge *PS*, dessen „bound“-Attribut den Wert „true“ besitzt. Aktuelle Teilnehmerreferenz ist die zu dieser Teilnehmermenge gehörende Teilnehmerreferenz, die wir als Sender in der Aktivitätsverbindung vermerken.

4. Haben wir kein passendes *partSet*-Objekt gefunden, so müssen wir eine neue Teilnehmermenge bilden. Das Binden an die zugehörige Teilnehmerreferenz geschieht wie im zweiten Fall. Ebenso fügen wir hier die aktuell betrachtete Aktivität in die Menge *initial_acts* des neu generierten *partSet*-Objekts ein. Die gefundene Schleife fügen wir in die Mengen *L* des neu generierten *partSet*-Objekts und des zugehörigen *part*-Objekts ein. Das neu generierte *partSet*-Objekt fügen wir in die Menge *part_sets* der zur aktuellen Teilnehmerreferenz gehörenden Teilnehmermenge ein. Als Sender in der Aktivitätsverbindung vermerken wir die Teilnehmermenge. In „bindSenderTo“ vermerken wir die zugehörige Teilnehmerreferenz. Aktuelle Teilnehmerreferenz ist im Anschluss die zur neu generierten Teilnehmermenge gehörende Teilnehmerreferenz.

Wird mittels einer <invoke>-Aktivität eine *Request-Response-Operation* verwirklicht, so müssen wir nach Aktivitätsverbindungen suchen, bei denen diese <invoke>-Aktivität die empfangende Aktivität ist. Als Empfänger wird in diesen Aktivitätsverbindungen die aktuelle Teilnehmerreferenz vermerkt.

Betrachten wir nun noch den Fall, dass eine Aktivität in mehreren Aktivitätsverbindungen vorkommt. Abgesehen von der <invoke>-Aktivität, die einmal als sendende und mehrmals als empfangende Aktivität in verschiedenen Aktivitätsverbindungen vorkommen kann.

Seien AL1 und AL2 zwei Aktivitätsverbindungen, bei denen die empfangende Aktivität y des BPEL-Prozesses Y dieselbe ist. In beiden Fällen muss zwischen den beiden beteiligten BPEL-Prozessen dieselbe Beziehung bestehen. Dies muss der Fall sein, da ansonsten die sendenden Aktivitäten innerhalb von verschieden vielen Schleifen liegen würden. Dies haben wir in Abschnitt 3.3 allerdings ausgeschlossen.

Betrachten wir nun AL1 und AL2. Müssen wir nun eine neue Teilnehmermenge bilden oder eine alte weiterverwenden, so können die folgenden Fälle auftreten:

1. Die beiden sendenden Aktivitäten liegen innerhalb derselben Schleife:
   Wie oben finden wir entweder ein *partSet*-Objekt oder müssen ein neues generieren.

2. Die beiden sendenden Aktivitäten liegen in unterschiedlichen Schleifen:
   Existiert bereits eine Teilnehmermenge, die sich auf diese beiden Schleifen bezieht und die wir wiederverwenden können, so verwenden wir diese. Ansonsten erzeugen wir eine neue Teilnehmermenge, in dessem zugehörigem *partSet*-Objekt wir die beiden Schleifen in die Menge L einfügen.

Seien AL1 und AL2 nun zwei Aktivitätsverbindungen, bei denen die sendende Aktivität x übereinstimmt. Wird eine neue Teilnehmermenge gebildet, die in AL1 auf die zugehörige Teilnehmerreferenz gebunden werden muss, so wird diese in AL2 ebenfalls auf dieselbe Teilnehmerreferenz gebunden.

Das nachfolgende Beispiel dient der Illustration des vorgestellten Ansatzes.

Beispiel 3.7:
Betrachten wir im Folgenden P2 aus dem Beispiel in Abbildung 3.25.

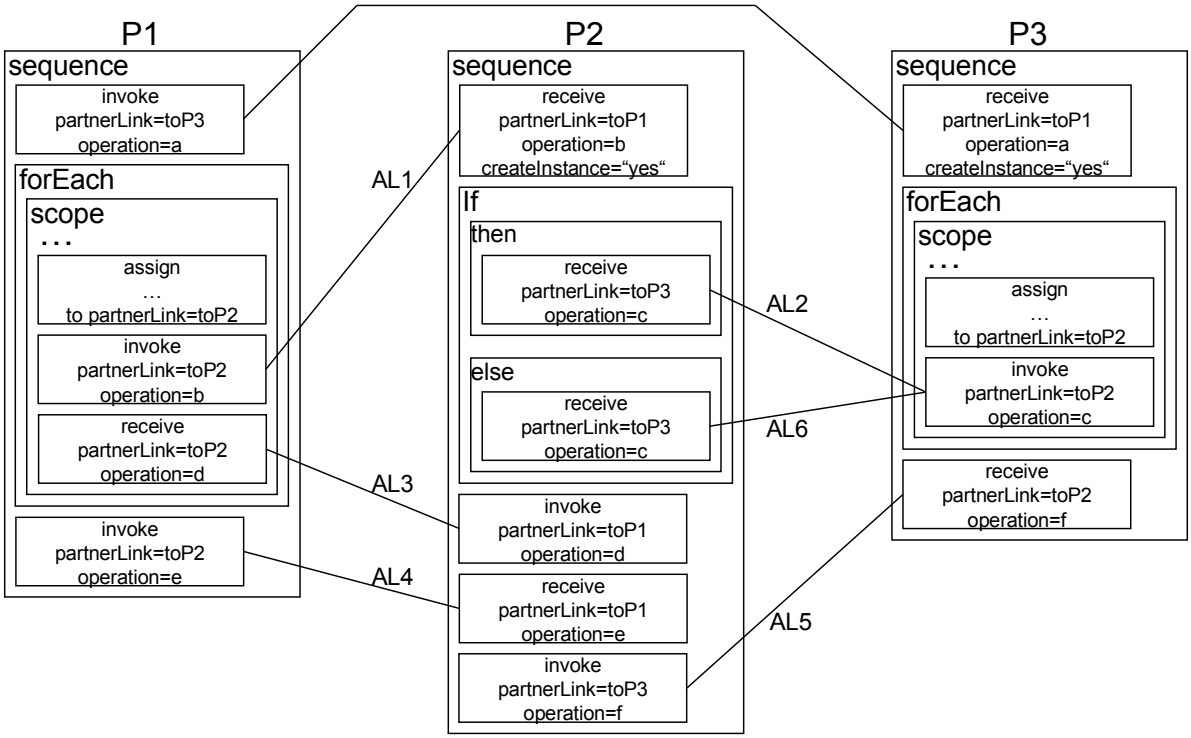

Abbildung 3.25.: Beispielprozesse, für die wir die Generierung von Teilnehmermengen und Teilnehmerreferenzen betrachten (II)

Zur ersten kommunizierenden Aktivität finden wir die zugehörige Aktivitätsverbindung AL1. Daraus können wir ablesen, dass sich P2 zu diesem Zeitpunkt in der Instanzklasse n^1 befindet. Folglich müssen wir eine Teilnehmermenge generieren. Wir finden die Schleife bei P1, die über die Menge der Instanzen von P2 iteriert. Diese Schleife vermerken wir im aktuell zu generierenden „partSet"-Objekt. Ebenso die aktuell betrachtete Aktivität bei P2. Sei die Teilnehmermenge „P2_set_1" und die zugehörige Teilnehmerreferenz „current_P2_1". Als Empfänger in AL1 vermerken wir „current_P2_1". Das „bound"-Attribut des „partSet"-Objekts erhält den Wert „true".

Nun verzweigt sich P2. Für jeden Zweig rufen wir *GENERATE-PARTS* rekursiv auf. Folgen wir zunächst dem ersten Zweig.

Die nächste betrachtete Aktivitätsverbindung ist AL2. Wir sehen, dass wir eine Teilnehmermenge verwenden müssen. Es existiert noch keine Teilnehmermenge, die sich auf die <forEach>-Aktivität bei P3 bezieht. Folglich müssen wir eine solche generieren. Sei die Teilnehmermenge „P2_set_2“ und die zugehörige Teilnehmerreferenz „current_P2_2“. Wir wissen, dass „P2_set_2“ eine Teilmenge von „P2_set_1“ ist. Folglich fügen wir „P2_set_2“ in *part_sets* des zu „P2_set_1“ gehörenden „partSet“-Objekts ein. Die aktuell betrachtete Aktivität vermerken wir in der Teilnehmermenge „P2_set_2“. Als Empfänger in AL2 geben wir „current_P2_2“ an. Das „bound“-Attribut beim zu „P2_set_2“ gehörenden „partSet“-Objekt setzen wir auf den Wert „true“.

Die nächste betrachtete Aktivitätsverbindung ist AL3. Wir sehen, dass wir eine Teilnehmermenge verwenden müssen. In der Menge *part_sets* der zuletzt verwendeten Teilnehmermenge befindet sich kein *part_set*-Objekt. Folglich durchsuchen wir als nächsten Schritt die Menge *PS*. In der Menge *PS* finden wir eine Teilnehmermenge, die wir wiederverwenden können und die bereits an ihre Teilnehmerreferenz gebunden wurde. Somit können wir als Sender „current_P2_1“ vermerken.

Bei AL4 sehen wir, dass sich P2 nun in der Instanzklasse n^0 befindet. Folglich bilden wir die Teilnehmerreferenz „selected_P2_1“, die wir in AL4 als Empfänger vermerken und in AL5 als Sender. Zu „selected_P2_1“ vermerken wir die Teilnehmermenge „P2_set_1“.

Nun sind wir für diesen Zweig fertig und geben alle uns zu diesem Zeitpunkt bekannten Teilnehmermengen zurück.

Dem Aufruf von *GENERATE-PARTS* für den zweiten Zweig übergeben wir diese Menge. Dabei müssen wir allerdings in allen Teilnehmermengen, die wir zu dem Zeitpunkt, an dem es zur Verzweigung kommt, noch nicht kennen, das „bound“-Attribut auf den Wert „false“ setzen. Hier ist dies bei „P2_set_2“ der Fall. Folgen wir nun dem zweiten Zweig.

Die nächste betrachtete Aktivitätsverbindung ist AL6. Aktuelle Teilnehmerreferenz ist „current_P2_1“. Als dazugehörige Teilnehmermenge finden wir „P2_set_1“. In der Menge *part_sets* finden wir die Teilnehmermenge „P2_set_2“, die wir bei AL6 wiederverwenden können. Als Empfänger in AL6 geben wir „current_P2_2“ an. Das „bound“-Attribut von „P2_set_2“ setzen wir auf „true“. Die aktuell betrachtete Aktivität vermerken wir in der Teilnehmermenge „P2_set_2“.

Die Aktivitätsverbindungen AL3 bis AL5 werden bei der Abarbeitung des zweiten Zweiges analog zu vorher bearbeitet, wobei dort „selected_P2_1“ wiederverwendet werden kann.

Ergänzungen

Beziehen wir im Folgenden die <flow>-Aktivität mit ein. Anstatt einer einzelnen kommunizierenden Aktivität finden wir hier unter Umständen eine Menge dieser Aktivitäten, die gleichzeitig aktiv werden können. Da wir hier einen BPEL-Prozess separat und unabhängig von den anderen BPEL-Prozessen bearbeiten, wissen wir nicht, in welcher Reihenfolge die gefundenen Aktivitäten bei der tatsächlichen Ausführung verwendet werden, da dies durch die Link-Semantik nicht unbedingt gegeben ist, sondern durch die Reihenfolge der dazu passenden Aktivitäten beim Partnerprozess festgelegt werden kann.

Um die Reihenfolge feststellen zu können, müssen wir den in Abschnitt 3.2 vorgestellten Ansatz erweitern. Beim Bilden der Aktivitätsverbindungen lassen wir einen Zähler mitlaufen. Der Zähler wird den Funktionen *FIND-ACTIVITIES* und *GENERATE-ACTIVITY-LINKS* als Parameter übergeben. Der Zähler gibt an, wieviele Aktivitätsverbindungen bis zu der aktuellen Aktivitätsverbindung bereits generiert wurden. Aus dem Wert kann man schließen, welche Aktivitäten vor der aktuellen Aktivität ausgeführt wurden. Der aktuelle Wert des Zählers wird in der neuen Aktivitätsverbindung vermerkt.

Bei einer Verzweigung rufen wir für jeden möglichen Zweig die Funktion *FIND-ACTIVITIES* auf. Jedem Aufruf wird der zu diesem Zeitpunkt aktuelle Zählerwert übergeben. Folglich können Aktivitätsverbindungen mit demselben Zählerwert in verschiedenen Zweigen existieren.

Bilden wir eine Aktivitätsverbindung, die bereits zuvor gebildet wurde (durch den Aufruf der Funktion *FIND-ACTIVITIES* bei einem anderen Zweig der Verzweigung). Ist der Zählerwert zu diesem Zeitpunkt größer als der Zählerwert in der bereits vorhandenen Aktivitätsverbindung, so aktualisieren wir den dort gespeicherten Zählerwert mit dem aktuellen Zählerwert.

Zusätzlich müssen wir die Abbruchbedingung für den Fall abändern, dass wir keine neuen Aktivitätsverbindungen bilden können, da identische Aktivitätsverbindungen schon vorhanden sind. Tritt nun der Fall ein, dass unser Zählerwert größer ist als der in den entsprechenden, bereits vorhandenen Aktivitätsverbindungen angegebene Wert, so aktualisieren wir diese Werte und dürfen nicht abbrechen.

Änderungen, die in einem Zweig stattfinden, der abgebrochen wird, da keine Aktivitätsverbindungen gebildet werden können, müssen rückgängig gemacht werden.

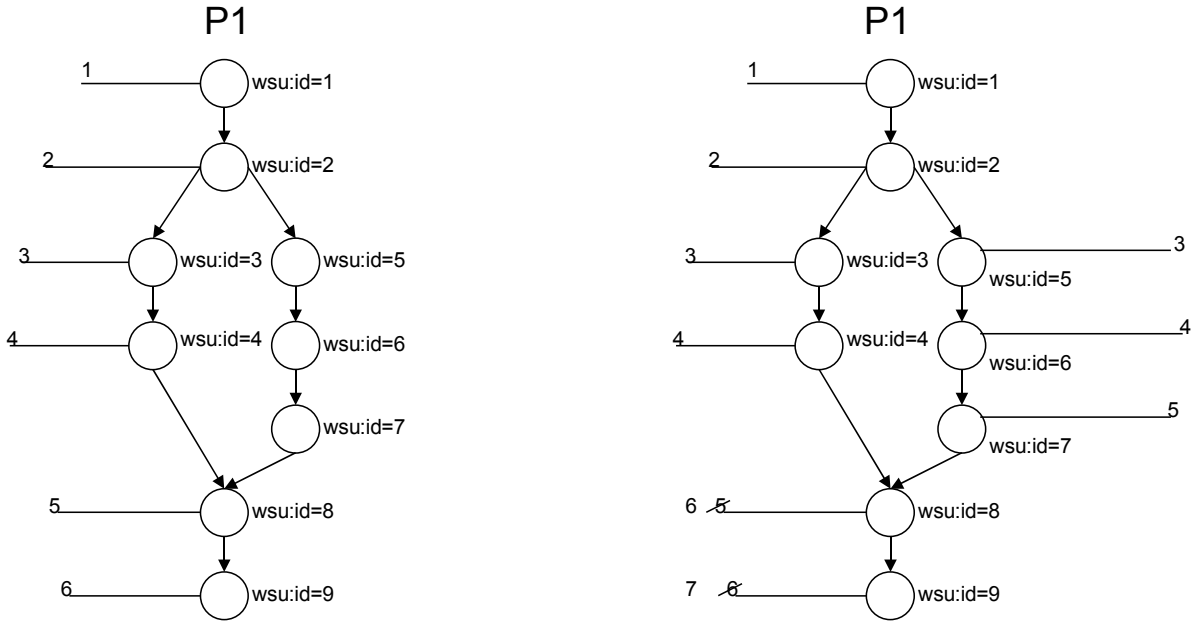

Abbildung 3.26.: Eintragen der Zählerwerte in Aktivitätsverbindungen

Abbildung 3.26 zeigt die Reihenfolge der Ausführung der kommunizierenden Aktivitäten bei BPEL-Prozess P1. Bei der Aktivität mit „wsu:id=8“ treffen diese Pfade wieder aufeinander. Nehmen wir an, es ist ein weiterer BPEL-Prozess P2 vorhanden, mit dem P1 kommuniziert. Dieser besitzt eine entsprechende Verzweigung, so dass bei P2 für die Aktivität mit der „wsu:id=8“

genau eine passende Aktivität vorkommt. Links sehen wir die angedeuteten Aktivitätsverbindungen mit dem eingetragenen Zählerwert, nachdem der erste Zweig der Verzweigung vollständig bearbeitet wurde. Rechts sehen wir die angedeuteten Aktivitätsverbindungen, nachdem beide Zweige der Verzweigung bearbeitet wurde. Wir sehen, dass wir in der zur Aktivität mit „wsu:id=8“ gehörenden Aktivitätsverbindung den vermerkten Zählerwert aktualisieren müssen, da der aktuelle Zählerwert größer ist als der bisher in der Aktivitätsverbindung vermerkte.

Somit ist gegeben, dass eine Aktivitätsverbindung, die zeitlich nach einer anderen Aktivitätsverbindung aktiv wird, einen größeren Wert als diese gespeichert hat.

Zusätzlich merken wir uns zu jeder Aktivitätsverbindung für den aktuell betrachteten Prozess, welche Aktivitätsverbindung direkt zuvor generiert wurde, an der eine Aktivität dieses Prozesses beteiligt war. Zu jeder Aktivitätsverbindung kann, dank Verzweigungen, eine Menge von Aktivitätsverbindungen vermerkt sein. Die Bedingung für einen erfolgreichen Abbruch in Abschnitt 3.2 modifizieren wir, indem wir die Bedingung hinzufügen, dass keine Aktivitätsverbindung als Vorläufer einer Aktivitätsverbindung vermerkt werden darf, die nicht bereits dort vermerkt war. Wird für eine Aktivitätsverbindung ein neuer Vorgänger vermerkt, so dürfen wir folglich nicht abbrechen.

Haben wir nun, wie in Abschnitt 3.2 bei der Suche nach kommunizierenden Aktivitäten, eine Menge von kommunizierenden Aktivitäten gefunden, so suchen wir alle zu diesen Aktivitäten gehörenden Aktivitätsverbindungen. Die aktuell betrachtete Aktivitätsverbindung ist die mit dem kleinsten vermerkten Zählerwert. Sei die zugehörige Aktivität A. Wir untersuchen, ob eine weitere kommunizierende Aktivität gefunden wurde, die in einer Aktivitätsverbindung vorkommt, die direkter Vorläufer einer zu A gehörenden Aktivitätsverbindung ist. Ist dies der Fall, so betrachten wir im folgenden diese Aktivität auf die gleiche Weise. Dies geschieht so lange, bis wir eine Aktivität gefunden haben, bei der dies nicht der Fall ist. Tritt dabei ein Zyklus auf, so wählen wir eine beliebige daran beteiligte Aktivität aus. Ein Beispiel für einen solchen Fall findet sich in Abbildung 3.27.

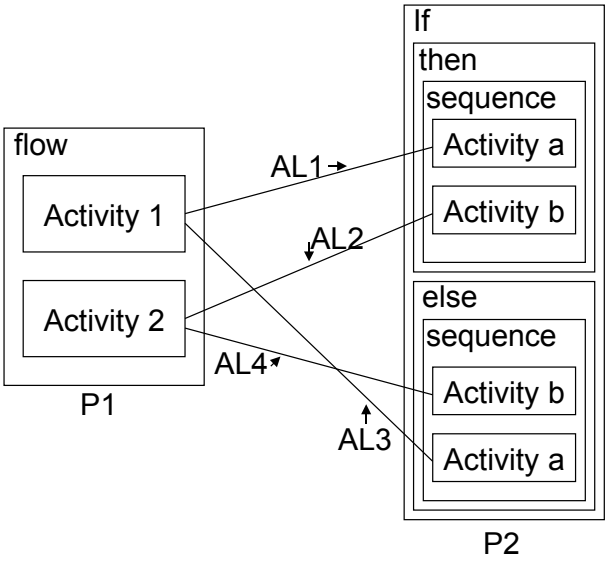

Abbildung 3.27.: Auftreten eines Zyklus bei der Bestimmung der Reihenfolge der Ausführung der kommunizierenden Aktivitäten

Beispiel 3.8:
Im Beispiel in Abbildung 3.27 finden wir bei der Betrachtung des BPEL-Prozesses P1 die beiden kommunizierenden Aktivitäten „Activity 1“ und „Activity 2“ im selben Durchgang. Betrachten wir zunächst „Activity 1“. „Activity 1“ ist in den Aktivitätsverbindungen AL1 und AL3 enthalten. Als direkter Vorgänger wurde bei AL3 AL4 vermerkt. In AL4 ist „Activity 2“ enthalten, die wir ebenfalls als ausführbare Aktivität gefunden haben. Folglich betrachten wir als nächstes „Activity 2“. „Activity 2“ ist in AL2 und AL4 enthalten. Als direkter Vorgänger wurde für AL2 AL1 vermerkt. Somit haben wir einen Zyklus entdeckt. Folglich können wir sowohl „Activity 1“ als auch „Activity 2“ auswählen.

In der Regel wird die Aktivität, die in der Aktivitätsverbindung mit dem kleinsten Wert vorkommt, die auszuwählende Aktivität sein.

Sei im Folgenden die ausgewählte Aktivität A. Nach der Betrachtung dieser Aktivität werden nicht die anderen, bereits gefundenen Aktivitäten betrachtet. Stattdessen suchen wir von A aus nach weiteren kommunizierenden Aktivitäten. Dies geschieht, da wir von A aus eine Aktivität, und somit eine zugehörige Aktivitätsverbindung finden können, bei der der vermerkte Zählerwert kleiner ist als bei allen anderen, ebenfalls noch zu betrachtenden Aktivitätsverbindungen oder die Vorgänger einer der zu den anderen bereits gefundenen Aktivitäten gehörenden Aktivitätsverbindung ist.

Es kann noch Fälle geben, bei denen auch hier noch eine Ungenauigkeit herrscht. Einen solchen Fall haben wir in Beispiel 3.8 gesehen. Dies hat nur Auswirkungen auf die Teilnehmerreferenzen, die wir in Aktivitätsverbindungen angeben, bei denen eine 1:1- oder eine 1:n-Beziehung besteht, da wir hier die zuletzt verwendete Teilnehmerreferenz weiterverwenden. Welche Teilnehmerreferenz die zuletzt verwendete ist, hängt von der Reihenfolge ab, in der die Aktivitäten betrachtet und somit die Teilnehmer gebildet und in den Aktivitätsverbindungen vermerkt werden.

Um absolut sicher zu gehen, müssten wir das Generieren der Teilnehmer mit dem Finden der Aktivitätsverbindungen kombinieren, da dort die korrekte Reihenfolge der Ausführung der Aktivitäten stets gewahrt wird. Das grundsätzliche Verfahren der Generierung der Teilnehmer bleibt dann gleich. Der Unterschied ist nur, dass wir dann nur die aktuell generierte Aktivitätsverbindung betrachten. Somit tritt dort der Fall nicht auf, dass eine Aktivität in mehreren Aktivitätsverbindungen vorkommt.

Wie für andere Verzweigungen merken wir uns in der Funktion *FIND-ACTIVITIES* für Aktivitäten, bei denen sich der BPEL-Prozess mittels *transitionConditions* verzweigt (siehe Kapitel 3.2), welche Aktivitäten dieses Prozesses zu diesem Zeitpunkt bearbeitet sind. Zusätzlich vermerken wir für diese Zweige die Menge der gesetzten Links dieses Prozesses mitsamt dem Status jedes gesetzten Links. Treffen wir in der Funktion *GENERATE-PARTS* auf eine solche Aktivität, so müssen wir nur die Zweige betrachten, die in der tatsächlichen Ausführung vorkommen können, d. h. für die wir in der Funktion *FIND-ACTIVITIES* keine leere Menge von Aktivitätsverbindungen erhalten haben. Konnte kein Zweig betrachtet werden, so vermerken wir diese Aktivität, um diese erneut zu überprüfen, wenn weitere Aktivitätsverbindungen bearbeitet wurden. Dies muss beachtet werden, da wir bei *GENERATE-ACTIVITY-LINKS* mehrere Aktivitätsverbindungen nacheinander generieren, bei *GENERATE-PARTS* allerdings nur jeweils eine Aktivitätsverbindung betrachtet wird. Somit können für einen Zweig einer Verzweigung zunächst mehr Aktivitäten als bearbeitet vermerkt sein, als zu diesem Zeitpunkt bei *GENERATE-PARTS* als bearbeitet gelten.

Wenn wir die Teilnehmer direkt bei der Bestimmung der Aktivitätsverbindungen generieren würden, dann könnten wir uns dies sparen, ebenso wie das Durchzählen der Aktivitätsverbindungen. Allerdings müssten wir Eintragungen in Aktivitätsverbindungen verwerfen, die in Zweigen auftreten würden, die nicht erfolgreich abgebrochen werden. Ebenso müssten Teilnehmermengen und Teilnehmerreferenzen verworfen werden, die in solchen Zweigen generiert werden würden.

3.5. Bereinigung der Aktivitätsverbindungen

In der Menge der Aktivitätsverbindungen *ALinks* können verschiedene Aktivitätsverbindungen vorkommen, die dieselbe empfangende Aktivität besitzen oder die dieselbe sendende Aktivität besitzen. Bei BPEL4Chor wird für diese Aktivitätsverbindungen ein einzelner *Message Link* in der *Participant Topology* und in den *Participant Groundings* angegeben (vgl. [DKLW07a]).

Aus diesem Grund vereinigen wir solche Aktivitätsverbindungen. In einer Aktivitätsverbindung können somit mehrere sendende und mehrere empfangende Aktivitäten vorkommen. Wir erweitern jede Aktivitätsverbindung daher um folgende Attribute:

- Die Menge der sendenden Aktivitäten *SEND_ACTs*
- Die Menge der empfangenden Aktivitäten *REC_ACTs*

Bevor wir überprüfen, welche Aktivitätsverbindungen wir vereinigen können, fügen wir die beiden an einer Aktivitätsverbindung beteiligten Aktivitäten in die jeweilige Menge ein, damit wir bei der Überprüfung nur *SEND_ACTs* und *REC_ACTs* überprüfen müssen

Wir durchsuchen die Menge *ALinks* nach Aktivitätsverbindungen, die in *SEND_ACTs* eine übereinstimmende Aktivität besitzen. Finden wir für eine Aktivitätsverbindung *al* eine Menge *AL* von anderen Aktivitätsverbindungen, für die dies zutrifft, dann entfernen wir die in *AL* enthaltenen Aktivitätsverbindungen aus *ALinks*. Die in diesen Aktivitätsverbindungen enthaltenen Informationen fügen wir in *al* ein. Dazu vereinigen wir die Listen *SEND_ACTs* und *REC_ACTs* der betreffenden Aktivitätsverbindungen. Diese Listen werden den entsprechenden Listen bei *al* hinzugefügt. Ebenso verfahren wir mit den Listen *SEND_PARTs* und *REC_PARTs*.

Im Anschluss verfahren wir für Aktivitätsverbindungen mit übereinstimmenden Aktivitäten in *REC_ACTs* analog. Allerdings müssen wir hier die Menge der Aktivitätsverbindungen so oft durchgehen, bis keine Aktivitätsverbindungen mehr zusammengefasst werden können. Somit haben wir hier Fälle abgedeckt, in denen zwei Aktivitätsverbindungen nur in Verbindung mit einer dritten Aktivitätsverbindung zusammengefasst werden können. Abbildung 3.28 zeigt einen solchen Fall. Ri sind dabei die empfangenden Aktivitäten, Si sind entsprechend die sendenden Aktivitäten.

In Abbildung 3.28 haben wir bereits die Aktivitätsverbindungen zusammengefasst, die die gleichen sendenden Aktivitäten haben. Aus den ursprünglich sechs Aktivitätsverbindungen wurden die drei Aktivitätsverbindungen AL1 bis AL3. Wir fassen nun Aktivitätsverbindungen zusammen, die gleiche empfangende Aktivitäten haben. Im ersten Durchgang vereinigen wir AL1 und AL2. Erst im zweiten Durchgang können wir die neue Aktivitätsverbindung mit AL3 zusammenfassen.

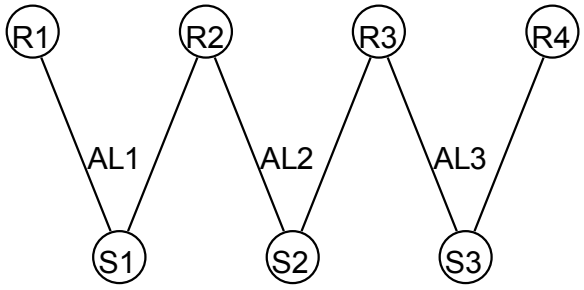

Abbildung 3.28.: Vereinigung von Aktivitätsverbindungen mit gleichen empfangenden Aktivitäten

Nun sind keine weiteren Aktivitätsverbindungen mehr vorhanden, die übereinstimmende sendende oder empfangende Aktivitäten besitzen. Ist in *SEND_ACTs* mehr als ein Element vorhanden, so ändern wir bei allen darin enthaltenen Aktivitäten das „wsu:id“-Attribut auf denselben Wert. Dieser Wert muss dabei mit dem Wert des „wsu:id“-Attributs einer in *SEND_ACTs* enthaltenen Aktivität sein, um sicherzustellen, dass keine weitere Aktivität dieses BPEL-Prozesses in einem anderen *Message Link* existiert, bei der das „wsu:id“-Attribut denselben Wert besitzt. Für die Aktivitäten in *REC_ACTs* verfahren wir analog.

Nun können bei Aktivitätsverbindungen folgende Fälle auftreten:

1. In *SEND_PARTs* sind unterschiedliche *part*-Objekte enthalten.
2. In *REC_PARTs* sind unterschiedliche *part*-Objekte enthalten.

Der erste Fall ist in Ordnung, da BPEL4Chor diesen Fall erlaubt. Durch unsere Annahme haben wir bereits ausgeschlossen, dass verschiedene Prozessmodelle als Sender auftreten können.

Tritt der zweite Fall ein, so müssen wir reagieren, da als Empfänger in einem *Message Link* bei BPEL4Chor, und somit bei den Aktivitätsverbindungen, nur eine Teilnehmerreferenz angegeben sein darf.

Verhindern einer Menge unterschiedlicher Teilnehmerreferenzen in der Liste *REC_PARTs* einer Aktivitätsverbindung

Zunächst vermerken wir zu jeder Aktivitätsverbindung, welche Teilnehmermengen oder Teilnehmerreferenzen bisher in dieser Aktivitätsverbindung als Sender bzw. als Empfänger fungierten. Dies benötigen wir, da wir die entsprechenden Mengen, in denen diese Informationen bisher enthalten sind, in der Folge verändern und somit den Ausgangszustand nicht mehr kennen würden.

Wie beim Generieren der Teilnehmermengen und Teilnehmerreferenzen in Abschnitt 3.4 suchen wir in der Funktion *NEW-PARTICIPANT-REFERENCES* in einem Prozess nach den nächsten aktiv werdenden kommunizierenden Aktivitäten. Jeder gegebene BPEL-Prozess wird separat bearbeitet. Zu einer kommunizierenden Aktivität finden wir höchstens genau eine Aktivitätsverbindung. Wir wissen, ob diese Aktivität eine sendende oder empfangende Aktivität ist und somit, ob wir die sendenden oder empfangenen Teilnehmermengen bzw. Teilnehmerreferenzen betrachten müssen.

Definition 3.25: *PR* enthält die Menge der neu generierten Teilnehmerreferenzen. Ein Element dieser Menge ist das Paar einer (neu generierten) Teilnehmerreferenz und einer Liste von Teilnehmerreferenzen.

Ein Element gibt an, welche Teilnehmerreferenzen durch eine neue Teilnehmerreferenz ersetzt werden können. Die ersetzbaren Teilnehmerreferenzen befinden sich in dem Paar in der Liste der Teilnehmerreferenzen. Über diese Liste können wir später die Gültigkeitsbereiche bestimmen, auf die die neue Teilnehmerreferenz beschränkt sein muss.

Definition 3.26: *R* ist eine Menge von neu generierten Teilnehmerreferenzen, wobei zu jeder dieser Teilnehmerreferenzen eine Liste von Teilnehmerreferenzen vermerkt ist. Ein Element dieser Menge ist somit das Paar einer Teilnehmerreferenz und einer Liste von Teilnehmerreferenzen.

PR ist unsere globale Menge, in der wir vermerken, welche anderen Teilnehmerreferenzen die neu generierte Teilnehmerreferenz beim ersten Auftreten ersetzt. Im Gegensatz zu Elementen in der Menge *R* dürfen Elemente in der Menge *PR* nicht mehr verändert werden.

R wird der Funktion *NEW-PARTICIPANT-REFERENCES* als Parameter übergeben. Die Übergabe geschieht „by-value“. Kommt es im betrachteten BPEL-Prozess zu einer Verzweigung, so wird für jeden Zweig, den wir bearbeiten müssen, die Funktion *NEW-PARTICIPANT-REFERENCES* rekursiv aufgerufen. Dabei wird jedem Aufruf die zu diesem Zeitpunkt aktuelle Menge *R* übergeben.

Angenommen, wir finden in einer Aktivitätsverbindung mehrere unterschiedliche sendende Teilnehmermengen oder Teilnehmerreferenzen in der Liste *SEND_PARTs*. Dieser Fall ist zwar erlaubt, allerdings binden wir diese Teilnehmermengen oder Teilnehmerreferenzen an eine neue Teilnehmerreferenz. Somit können diese sendenden Referenzen in der Folge in anderen Aktivitätsverbindungen durch diese neue Teilnehmerreferenz ersetzt werden.

Wir generieren dazu eine neue Teilnehmerreferenz, die vom gleichen Typ ist wie die angegebenen Sender. Diese Teilnehmerreferenz muss auf die gleichen Gültigkeitsbereiche beschränkt sein wie die angegebenen Sender. Folglich ist die Menge *L* des zugehörigen *part*-Objekts gleich der Vereinigung der Mengen *L* der angegebenen Sender. Mittels dem „bindSenderTo“-Attribut der Aktivitätsverbindung binden wir die Sender an diese neue Teilnehmerreferenz. Wir bilden das Paar der Teilnehmerreferenz und der Liste der Teilnehmerreferenzen, die als Sender fungieren. Ist eine Teilnehmermenge ein Sender, so wird die zugehörige Teilnehmerreferenz in diese Menge eingefügt. Die Liste gibt uns an, welche Teilnehmerreferenzen wir in darauffolgenen Aktivitätsverbindungen durch diese neue Teilnehmerreferenz ersetzen können. Dieses Paar wird in die Menge *R* und in die Menge *PR* eingefügt.

Angenommen, wir finden eine Aktivitätsverbindung mit mehreren unterschiedlichen Teilnehmerreferenzen in der Liste *REC_PARTs*, dann erzeugen wir wie zuvor eine neue Teilnehmerreferenz. Wir entfernen alle Elemente aus *REC_PARTs* und fügen die neue Teilnehmerreferenz ein. Wie oben wird das gebildete Paar in *PR* und *R* eingefügt.

Finden wir in einer nachfolgenden Aktivitätsverbindung in *SEND_PARTs* mehrere unterschiedliche Teilnehmerreferenzen. Finden wir in *R* eine Teilnehmerreferenz, in deren zugehöriger Liste eine bei *SEND_PARTs* angegebene Teilnehmerreferenz enthalten ist. Dann entfernen wir alle Elemente der Liste aus *SEND_PARTs* und fügen die gefundene Teilnehmerreferenz in *SEND_PARTs* ein. Ist ein Element in *SEND_PARTs* n mal vorhanden und in der Liste der gefundenen Teilnehmerreferenz m mal und es gilt n > m, dann werden von diesen n Elementen nur m entfernt, da die neue Teilnehmerreferenz nur n dieser Teilnehmerreferenzen ersetzen darf. Dies müssen wir

beachten, da der Rest dieser Referenzen aufgrund eines anderen Zweiges einer Verzweigung dort angegeben sein kann, den wir aktuell aber nicht verfolgen.

Es ist möglich, dass wir mehrere Teilnehmerreferenzen in *R* finden, in deren Listen unterschiedliche, bei *SEND_PARTs* angegebene Teilnehmerreferenzen enthalten sind. In solch einem Fall sind im Anschluss mehrere in *R* enthaltenen Teilnehmerreferenzen in der Menge *SEND_PARTs* enthalten.

Sind in *SEND_PARTs* nun noch unterschiedliche Teilnehmerreferenzen oder noch Teilnehmermengen vorhanden, so müssen wir eine neue Teilnehmerreferenz bilden. Da wir zu Beginn vermerkt haben, welche Teilnehmerreferenzen oder Teilnehmermengen ursprünglich in *SEND_PARTs* vorhanden waren, wissen wir, welche Teilnehmerreferenzen wir in der Folge mit dieser neuen Teilnehmerreferenz ersetzen können. Wir gehen nun die Menge *R* durch und entfernen aus den Listen die Teilnehmerreferenzen, die in der zur neuen Teilnehmerreferenz gehörenden Liste vorhanden sind, da diese Teilnehmerreferenzen ab diesem Zeitpunkt an die neue Teilnehmerreferenz gebunden werden. Anschließend fügen wir das neu generierte Paar in *R* und in *PR* ein.

Sind in einer nachfolgenden Aktivitätsverbindung in *REC_PARTs* mehrere unterschiedliche Teilnehmerreferenzen gegeben, so verfahren wir analog.

Beispiel 3.9:
Angenommen, wir haben in *R* eine Teilnehmerreferenz neu_Ref1 mit zugehöriger Liste (Ref1,Ref2,Ref3) gegeben. Refi sind dabei Teilnehmerreferenzen. Wir finden in einer Aktivitätsverbindung die Liste (Ref1,Ref2,Ref3,Ref1). Wir wissen nun, dass wir (Ref1,Ref2,Ref3) durch neu_Ref1 ersetzen können. Wir erhalten nun (neu_Ref1,Ref1). Wir müssen daher eine neue Teilnehmerreferenz neu_Ref2 bilden, da noch mehrere Elemente in der Liste vorhanden sind. Die zu neu_Ref2 gehörende Liste ist dann (Ref1,Ref2,Ref3,Ref1). neu_Ref1 wird vollständig durch neu_Ref2 ersetzt. Somit kann neu_Ref1 aus *R* entfernt werden.

Beispiel 3.10:
Angenommen, wir haben in *R* eine Teilnehmerreferenz neu_Ref1 mit zugehöriger Liste (Ref1,Ref2,Ref3) gegeben. Wir finden in einer Aktivitätsverbindung die Liste (Ref3,Ref4). Wir erhalten nun (neu_Ref1,Ref4). Wir müssen daher eine neue Teilnehmerreferenz neu_Ref2 bilden, da in der Liste noch unterschiedliche Teilnehmerreferenzen vorhanden sind. Die zu neu_Ref2 gehörende Liste ist dann (Ref3,Ref4). Aus der zu neu_ref1 gehörenden Liste entfernen wir Ref3, da Ref3 ab diesem Zeitpunkt auf neu_Ref2 gebunden wird. In *R* befinden sich nun die beiden Paare (neu_Ref1, (Ref1,Ref2)) und (neu_Ref2, (Ref3,Ref4)).

Folgen wir nun einem anderen Zweig einer Verzweigung. Etwa bei der ersten Verzweigung dem zweiten Pfad beim Beispiel in Abbildung 3.29. Dann kann der Fall auftreten, dass wir auf eine Aktivitätsverbindung treffen, die in *REC_PARTs* eine neue Teilnehmerreferenz enthält, die in *PR* enthalten ist. Wir wissen, welche Teilnehmerreferenzen ursprünglich in *REC_PARTs* vorhanden waren. Somit wissen wir, welche Teilnehmerreferenzen diese nun enthaltene Teilnehmerreferenz ersetzt. Wir können damit ein entsprechendes Paar bilden und in *R* einfügen, wenn diese Teilnehmerreferenz nicht schon enthalten war. War sie noch nicht in *R* enthalten, so entfernen wir aus der Liste eines jeden Paares die Teilnehmerreferenzen, die ursprünglich in *REC_PARTs* enthalten waren.

Treffen wir auf eine Aktivitätsverbindung, bei deren „bindSenderTo“-Attribut auf eine Teilnehmerreferenz verwiesen wird, die in *PR* enthalten ist, so verfahren wir analog. Der Unterschied

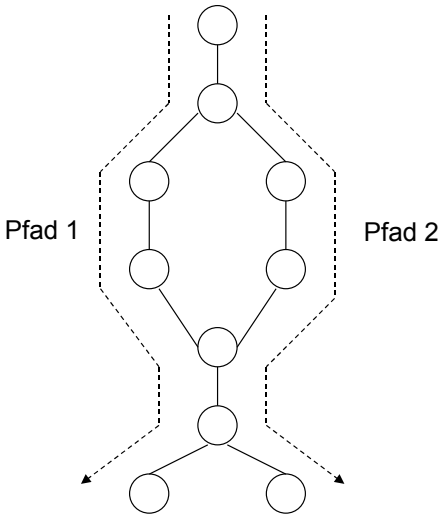

Abbildung 3.29.: Unterschiedliche mögliche Pfade eines BPEL-Prozesses

ist hier der, dass wir zuvor unter Umständen noch Teilnehmerreferenzen aus *SEND_PARTs* mit Teilnehmerreferenzen aus *R* ersetzen.

Beispiel 3.11:
Treffen wir auf eine Aktivitätsverbindung, wobei bei *SEND_PARTs* (Ref1,Ref2,Ref3) angegeben ist und bei „bindSenderTo“ auf neu_Ref1 verwiesen wird. In *R* ist (neu_Ref2,(Ref2,Ref3,Ref4)) enthalten. Wir ersetzen in *SEND_PARTs* Ref2 und Ref3 mit neu_Ref2 und erhalten (Ref1,neu_Ref2). Wir bilden nun das Paar (neu_Ref1,(Ref1,Ref2,Ref3)). Wir entfernen Ref2 und Ref3 aus der zu neu_Ref2 gehörenden Liste, da diese Teilnehmerreferenzen im Folgenden an neu_Ref1 gebunden werden. Anschließend fügen wir das zu neu_Ref1 gehörende Paar in *R* ein. *R* enthält nun (neu_Ref1,(Ref1,Ref2,Ref3)) und (neu_Ref2,(Ref4)).

Im Anschluss enthält *PR* alle neu generierten Teilnehmerreferenzen aller Prozesse, wobei zu jeder dieser Teilnehmerreferenzen vermerkt ist, welche anderen Teilnehmerreferenzen darauf gebunden werden können.

3.6. Korrelationsmengen

Wie in Abschnitt 2.2.2 beschrieben, dürfen in den PBDs, wie in BPEL selbst, Korrelationsmengen vorkommen. Allerdings werden hier die QNames der Eigenschaften als NCNames verwendet und nicht als Referenzen auf in WSDL-Dokumenten definierte Eigenschaftswerte. Diese Typisierung findet erst in den *Participant Groundings* statt.

Im Folgenden betrachten wir einen QName als Konkatenation zweier Strings, die durch einen Doppelpunkt getrennt werden. Der String vor dem Doppelpunkt fungiert als Namensraumpräfix, der String nach dem Doppelpunkt als NCName.

Ein Beispiel für eine Korrelationsmenge, das [JE07] entnommen wurde, ist:
<correlationSet name=“PurchaseOrder“ properties=“cor:customerID cor:orderNumber“/>

Für die Umwandlung eines BPEL-Prozesses in eine PBD werden die Namensraumpräfixe, in diesem Fall „cor“, entfernt. Bezugnehmend auf die NCNames der Eigenschaften, in diesem

Fall „customerID“ und „orderNumber“, wird in den *Participant Groundings* die Typisierung vorgenommen.

Nun kann der Fall eintreten, dass in verschiedenen Korrelationsmengen Eigenschaften mit denselben NCNames verwendet werden. Dies muss verhindert werden.

Definition 3.27: *NCn* ist eine Menge von NCNames, die bereits für Eigenschaften einer Korrelationsmenge verwendet wurden.

Für jeden beim „properties“-Attribut einer Korrelationsmenge angegebenen QName wird ein *corr_prop*-Objekt generiert. Ein *corr_prop*-Objekt setzt sich zusammen aus:

- NCname
  Der NCName der Eigenschaft. Möglicherweise umbenannt.
- property_name
  Der ursprüngliche NCName der Eigenschaft.
- namespace_URI
  Die Namensraum-URI, die mittels des Namensraumpräfixes ermittelt wird.

Definition 3.28: *CORR_PROPs* ist eine Menge von *corr_prop*-Objekten.

Definition 3.29: *URIs* ist eine Menge von Namensraum-URIs.

Vorgegangen wird wie folgt: Wir durchsuchen jeden Prozess nach <correlationSets>-Elementen. Für jedes in einem dieser Elemente eingefassten <correlationSet>-Element erhalten wir eine Liste von QNames der Eigenschaften. Für jeden QName wird ein *corr_prop*-Objekt generiert. Mittels des Namensraumpräfixes, der bei jedem QName angegeben ist, bestimmen wir die Namensraum-URI, die wir im entsprechenden Attribut des *corr_prop*-Objektes vermerken. Die Namensraum-URI wird ebenfalls der Menge *URIs* hinzugefügt. Anschließend entfernen wir den Namensraumpräfix aus dem QName und erhalten einen NCName. Diesen NCName vermerken wir im „property_name“-Attribut. Nun durchsuchen wir die Menge *NCn*, ob dieser NCName dort bereits vorhanden ist. Ist dies nicht der Fall, so fügen wir den NCName in *NCn* ein und vermerken diesen NCName im „NCname“-Attribut. Ist der NCName allerdings bereits in *NCn* vorhanden, so muss dieser umbenannt werden, so dass der neue NCName nicht bereits in *NCn* vorhanden ist. Der neue NCName wird im „NCname“-Attribut vermerkt. Der QName in der QName-Liste beim „properties“-Attribut des <correlationSet>-Elements wird durch diesen NCName ersetzt.

Betrachten wir aktuell einen QName. Existiert bereits ein *corr_prop*-Objekt mit derselben Namensraum-URI und demselben NCName im „property_name“-Attribut, so wird für diesen QName kein neues *corr_prop*-Objekt erzeugt. Der QName im „properties“-Attribut des entsprechenden <correlationSet>-Elements wird durch den NCName ersetzt, der im „NCname“-Attribut des entsprechenden *corr_prop*-Objekts angegeben ist.

Ein Namensraumpräfix wird nicht vermerkt. Dies geschieht erst im Abschnitt 3.8, aus dem Grund, dass verschiedene Namensraumpräfixe bei verschiedenen Korrelationsmengen auf dieselbe Namensraum-URI verweisen können. In diesem Fall benötigen wir für die entsprechende Namensraum-URI nur eine einzelne Namensraumdeklaration in den *Participant Groundings*, wobei der zugehörige Namensraumpräfix bei den verschiedenen Eigenschaften Verwendung findet.

Das aktuell generierte *corr_prop*-Objekt wird der Menge *CORR_PROPs* hinzugefügt.

3.7. Generierung der Participant Topology

Als ersten der drei Bestandteile von BPEL4Chor erzeugen wir die *Participant Topology*.

Zunächst müssen wir die Mengen *select_parts* der *part*-Objekte generieren. Mittels der Elemente in *select_parts* wird angegeben, welche Teilnehmerreferenz welchen anderen Teilnehmer oder welche andere Teilnehmermenge auswählt. Allerdings nur dann, wenn von dem eingebundenen Teilnehmer dort neue Instanzen des entsprechenden BPEL-Prozesses generiert werden.

Wir suchen in der Menge der Aktivitätsverbindungen *ALinks* nach Aktivitätsverbindungen, die eine instanzerzeugende Aktivität beinhalten. Über eine Aktivitätsverbindung, in der dies der Fall ist, erhalten wir einen Verweis auf das *part*-Objekt, das die empfangende Teilnehmerreferenz darstellt. Anhand dieser Teilnehmerreferenz suchen wir nach einer zugehörigen Teilnehmermenge. Finden wir eine Teilnehmermenge T, so merken wir uns diese. Ansonsten merken wir uns die Teilnehmerreferenz t.

Über die Aktivitätsverbindung erhalten wir ebenso den Verweis auf die sendende Teilnehmerreferenz. In *select_parts* des zu dieser Teilnehmerreferenz gehörenden *part*-Objekts fügen wir den Namen von T oder den Namen von t ein. Existieren mehrere sendende Teilnehmerreferenzen, so fügen wir den Namen von T oder den Namen von t entsprechend in jedem zugehörigen *part*-Objekt hinzu.

Im nachfolgenden Listing sehen wir die Struktur einer *Participant Topology*:

```
<topology name="NCName"
        targetNamespace="URI"
        xmlns... *>

        <participantTypes>
                <participantType name="NCName"
                        participantBehaviorDescription="QName"
                />*
        </participantTypes>

        <participants>
                <participant      name="NCName" type="NCName"
                        selects="NCNames" ?
                        forEach="QNames" ?
                        scope="QNames" ? />+
                <participantSet  name="NCName" type="NCName"
                        forEach="QNames">*
                        (<participant  .../ > | <participantSet  .../ >)+
                </participantSet>
        </participants>

        <messageLinks>
                <messageLink
                        name="NCName"
                        (sender="NCName" | senders="NCNames")
                        sendActivity="NCName"
                        receiver="NCName"
                        receiveActivity="NCName"
                        bindSenderTo="NCName"?
                        messageName="NCName"
                        (participantRefs="NCNames"
```

```
                    copyParticipantRefsTo="NCNames"?)?
                />*
        </messageLinks>

</topology>
```

Listing 3.1: Struktur der *Participant Topology* [DKLW07b]

Das Wurzelelement der *Participant Topology* ist <topology>. Als Attribute besitzt es „name“ und „targetNamespace“. Ebenso werden hier die erforderlichen Namensraumdeklarationen angegeben. Der Wert des „name“-Attributs ist beliebig. Als Wert des „targetNamespace“-Attributs geben wir „urn:chor“ an. Das <topology>-Element besitzt drei Kinder: <participantTypes>, <participants> und <messageLinks>.

Das <participantTypes>-Element besitzt <participantType>-Elemente als Kinder. Für jeden teilnehmenden BPEL-Prozess wird ein <participantType>-Element erzeugt. Ein <participantType>-Element enthält die beiden Attribute „name“ und „participantBehaviorDescription“. Das „name“-Attribut wird mit dem Wert „Name des Prozesses + _type“ belegt. Da wir bereits in Abschnitt 3.2 dafür gesorgt haben, dass jeder teilnehmende BPEL-Prozess einen eindeutigen Namen besitzt, hat ebenfalls jeder Teilnehmertyp einen eindeutigen Namen. Der Wert des „participantBehaviorDescription“-Attributs ergibt sich aus „Name des Prozesses:Name des Prozesses“. Nun können wir beim <topology>-Element die entsprechende Namensraumdeklaration vornehmen. Diese ist „xmlns:Name des Prozesses = Wert des „targetNamespace“-Attributs des Prozesses“.

Als nächstes generieren wir die <participant>- und <participantSet>-Elemente, die Kinder des <participants>-Elements bzw. Kinder eines <participantSet>-Elements sind. Die Hierarchie dieser Elemente ist dadurch festgelegt, dass Teilnehmermengen in der Menge *part_sets* eines *partSet*-Objekts enthalten sein können. Ebenso ist bei einem solchen Objekt bei „part_ref“ die zugehörige Teilnehmerreferenz angegeben. Die Menge aller Teilnehmermengen ist mit *PSets* gegeben (siehe Kapitel 3.4).

Als äußerstes Element können wir, für einen Prozess, maximal ein *partSet*-Objekt erhalten. Ob ein *partSet*-Objekt für ein äußerstes Element steht, erkennen wir daran, ob dieses *partSet*-Objekt in keiner Menge *part_sets* eines anderen *partSet*-Objekts enthalten ist. Alle anderen Teilnehmermengen sind Teilmenge dieser Teilnehmermenge.

Zu jeder in *PRefs* enthaltenen Teilnehmerreferenz erzeugen wir ein <participant>-Element. Durch die zu einer solchen Teilnehmerreferenz vermerkten Teilnehmermenge wissen wir, von welchem <participantSet>-Element dieses <participant>-Element das Kind ist. Ist keine Teilnehmermenge vermerkt, so ist das <participants>-Element der Vaterknoten.

Jedes <participant>- und jedes <participantSet>-Element kann die Attribute „name“ und „type“ enthalten. Das „type“-Attribut wird nur für die äußeren Elemente gesetzt und erhält den Wert „Name des Prozesses + _type“. Jedes Element erhält als Wert des „name“-Attributs den Wert des „name“-Attributs des zugehörigen *partSet*-Objekts oder *part*-Objekts.

Ein <participantSet>-Element kann, genau wie das zur zugehörenden Teilnehmerreferenz gehörende <participant>-Element, ein „forEach“-Attribut besitzen. Der Wert des Attributs enthält eine Liste von QNames. Dieses Attribut muss nur angegeben werden, wenn die im zu dieser Teilnehmermenge gehörenden *partSet*-Objekt vermerkten Schleifen <forEach>-Aktivitäten sind und diese forEach-Schleifen über diese Teilnehmermenge iterieren. Wir haben in einem *partSet*-Objekt vermerkt, welche Aktivitäten bei dessen erstem Auftreten benutzt wurden.

Für jede dort angegebene empfangende Aktivität müssen wir folgendes überprüfen: Über die zu dieser Aktivität gehörende Aktivitätsverbindung finden wir die sendenden Aktivitäten beim Partnerprozess, die sich innerhalb einer der beim *partSet*-Objekt vermerkten Schleifen befinden müssen. Aus diesen sendenden Aktivitäten lesen wir die dort angegebenen *Partner Links* aus. Wir suchen nun innerhalb all der beim *partSet*-Objekt angegebenen <forEach>-Schleifen nach einer <assign>-Aktivität, bei der eine Endpunktreferenz in einen dieser *Partner Links* kopiert wird. Finden wir eine solche <assign>-Aktivität, so überprüfen wir weiter, ob im <from>-Element des <copy>-Elements dieser <assign>-Aktivität ein XPath Ausdruck angegeben ist, in dem die Counter-Variable der gerade überprüften Schleife enthalten ist. Finden wir ein solche <assign>-Aktivität, so können wir das „forEach“-Attribut angeben. Das „forEach“-Attribut wird in dem Fall sowohl bei dem zur Teilnehmermenge gehörenden <participantSet>-Element, als auch in dem <participant>-Element angegeben, dass zur Teilnehmerreferenz gehört, die zu dieser Teilnehmermenge gehört.

Ein QName ergibt sich aus der Konkatenation des Namens des Prozesses, zu dem eine <forEach>-Aktivität gehört, und der „wsu:id“ der <forEach>-Aktivität. Diese beiden Werte werden durch einen Doppelpunkt getrennt. Somit ist der Name des Prozesses ein Namensraumpräfix, der in der entsprechenden Namensraumdeklaration im <topology>-Element bereits angegeben ist. Im „forEach“-Attribut wird der entsprechende QName eingefügt.

Jede auf diese Weise gefundene <forEach>-Schleife wird der Menge *FE* hinzugefügt.

Definition 3.30: *FE* ist die Menge aller <forEach>-Schleifen, die über eine Menge von Instanzen eines Partnerprozesses iterieren.

Bei der Generierung der PBDs benötigen wir diese Menge.

Neben diesen Attributen kann ein <participant>-Element ein „scope“-Attribut besitzen, welches als Wert eine Liste von QNames enthält. In diesem Attribut wird angegeben, auf welchen Gültigkeitsbereich des Partnerprozesses eine Teilnehmerreferenz beschränkt ist. Wir nehmen an, dass die <scope>-Aktivitäten, die die Gültigkeitsbereiche festlegen, direkt als Kinder der beim entsprechenden *part*-Objekt vermerkten Schleifen enthalten sind. Der QName setzt sich zusammen als Konkatenation des Namens des Prozesses, zu dem die betreffende <scope>-Aktivität gehört, und der „wsu:id“ der <scope>-Aktivität, wobei diese beiden Werte durch einen Doppelpunkt getrennt werden. Für jede in *L* vermerkte Schleife, die nicht im „forEach“-Attribut angegeben wird, suchen wir nach einer solchen <scope>-Aktivität, dessen QName wir bei „scope“ einfügen.

Sind bei einem *part*-Objekt keine Schleifen vermerkt, so geben wir das „scope“-Attribut nicht an. Sind alle Schleifen bereits im „forEach“-Attribut angegeben, so geben wir das „scope“-Attribut ebenfalls nicht an.

Ist die Menge *select_parts* des *part*-Objekts nicht leer, so besitzt das zugehörige <participant>-Element ein „selects“-Attribut. Der Wert dieses Attributs ist die Liste der in *select_parts* vermerkten Namen.

Anschließend erzeugen wir analog zu jeder in *PR* enthaltenen neuen Teilnehmerreferenz ein <participant>-Element. Jedes dieser <participant>-Elemente ist Kind des <participants>-Elementes. Ob dort eine in der Menge *L* des entsprechenden *part*-Objekts enthaltene Schleife beim „forEach“-Attribut des <participant>-Elements angegeben wird oder die eingebettete <scope>-Aktivität beim „scope“-Attribut angegeben wird, erkennen wir daran, ob die Schleife bei den Teilnehmerreferenzen, die an die neue Teilnehmerreferenz gebunden werden können, im zugehörigen <participant>-Element im „forEach“-Attribut angegeben ist. Nur Schleifen,

die dort angegeben sind, werden beim „forEach“-Attribut des zur neuen Teilnehmerreferenz gehörenden <participant>-Elements angegeben.

Das <messageLinks>-Element hat <messageLink>-Elemente als Kinder. Unsere Aktivitätsverbindungen haben noch keinen NCName. Wir ergänzen unsere Aktivitätsverbindungen um ein „name“-Attribut. Wir gehen über die Menge *ALinks* hinweg und belegen das „name“-Attribut der aktuell betrachteten Aktivitätsverbindung mit einem Zählerwert. Damit ist garantiert, dass jede Aktivitätsverbindung einen eindeutigen Namen hat.

Ein <messageLink>-Element hat folgende Attribute: „name“, „sender“ oder „senders“, „sendActivity“, „receiver“, „receiveActivity“, „bindSenderTo“ und „messageName“. Die beiden weiteren Attribute „participantRefs“ und „copyParticipantRefsTo“ geben wir nicht an, da wir das Weiterreichen von Endpunktreferenzen nicht betrachten.

Für jede in *ALinks* enthaltene Aktivitätsverbindung erzeugen wir ein <messageLink>-Element. Das „name“-Attribut des <messageLink>-Elements erhält einen beliebigen, aber eindeutigen Wert. Den selben Wert erhält das „messageName“-Attribut. Als Wert können wir den Wert des „name“-Attributs der entsprechenden Aktivitätsverbindung übernehmen.

Sind in der Liste *SEND_PARTs* der aktuell betrachteten Aktivitätsverbindung nur identische Objekte enthalten, so fügen wir im <messageLink>-Element das Attribut „sender“ ein, ansonsten „senders“. Hier geben wir die Werte des „name“-Attributs der verschiedenen in *SEND_PARTs* enthaltenen *part*- oder *partSet*-Objekte an.

Als Wert des „sendActivity“-Attributs geben wir die „wsu:id“ einer der in der Menge *SEND_ACTs* der Aktivitätsverbindung enthaltenen Aktivität an. In Abschnitt 3.5 haben wir bereits dafür gesorgt, dass alle in *SEND_ACTs* enthaltenen Aktivitäten denselben Wert in ihrem „wsu:id“-Attribut besitzen.

Als Wert des „receiver“-Attributs geben wir den Wert des „name“-Attributs des in *REC_PARTs* enthaltenen *part*-Objekts an. Den Fall, dass in *REC_PARTs* mehrere verschiedene *part*-Objekte vorhanden sind, haben wir in Abschnitt 3.5 verhindert.

Als Wert des „receiveActivity“-Attributs geben wir die „wsu:id“ einer in der Menge *REC_ACTs* der Aktivitätsverbindung enthaltenen Aktivität an.

Findet sich im „bindSenderTo“-Attribut der Aktivitätsverbindung ein Verweis auf eine Teilnehmerreferenz, so wird das „bindSenderTo“-Attribut in das <messageLink>-Element eingefügt. Der Wert dieses Attributs ist der Wert des „name“-Attributs des im „bindSenderTo“-Attribut der Aktivitätsverbindung angegebenen *part*-Objekts.

3.8. Generierung der Participant Groundings

Die Struktur der *Participant Groundings* sehen wir im folgenden Listing:

```
<grounding
        topology="QName"
        xmlns:top="URI"
        xmlns... *>

        <messageLinks>
                <messageLink
                        name="NCName"
                        portType="QName"
                        operation="NCName"
                />*
        </messageLinks>

        <properties>
                <property
                        name="NCName"
                        WSDLproperty="QName"
                />*
        </properties>

        <participantRefs>
                <participantRef
                        name="NCName"
                        WSDLproperty="QName"
                />*
        </participantRefs>

</grounding>
```

Listing 3.2: Struktur der *Participant Groundings* [DKLW07b]

Das Wurzelelement der *Participant Groundings* ist das <grounding>-Element. Es enthält das „topology“-Attribut sowie die Namensraumdeklarationen. Der Wert des „topology“-Attributs wird durch die Konkatenation des Strings „top“ und des Namens der *Participant Topology* festgelegt, wobei die beiden Strings durch einen Doppelpunkt getrennt werden. „top“ ist dabei ein Namensraumpräfix. In der entsprechenden Namensraumdeklaration wird als Namensraum die beim „targetNamespace“-Attribut der Participant Toplogy angegebene URI verwendet. Als Kinder enthält das <grounding>-Element die drei Elemente <messageLinks>, <participantRefs> und <properties>, wobei wir kein <participantRefs>-Element angeben, da wir das Weiterreichen von Endpunktreferenzen nicht betrachten.

Zunächst erweitern wir jede Aktivitätsverbindung um die Attribute „uri“und „ptype“. Für jede Aktivitätsverbindung betrachten wir eine der dort angegebenen sendenden oder empfangenden Aktivitäten.

Über den angegebenen *Partner Link* können wir auf den Partner-Link-Typ schließen. Über den Partner-Link-Typ erhalten wir einen Namensraumpräfix. Mit Hilfe dieses Präfixes bestimmen wir die zugehörige URI.

Über die angegebene Operation suchen wir in den entsprechenden WSDL-Definitionen nach dem zugehörigen *Port Type*. Das „ptype“-Attribut belegen wir mit dem NCName dieses *Port Types*. Die URI des Namensraums, in dem der *Port Type* deklariert ist, wird im Attribut „uri“ der Aktivitätsverbindung vermerkt. Ebenso wird diese URI der Menge *URIs* hinzugefügt.

Für jede in *URIs* enthaltene URI führen wir eine Namensraumdeklaration durch.

Definition 3.31: $prefix_{URI} : URI \rightarrow Namensraumpraefix$ ist die Funktion, die uns zu gegebener URI den passenden Namensraumpräfix in den *Participant Groundings* liefert.

Das <messageLinks>-Element hat als Kinder <messageLink>-Elemente. Das <messageLink>-Element hat die Attribute „name“, „portType“ und „operation“. Für jede Aktivitätsverbindung in *ALinks* erzeugen wir ein neues <messageLink>-Element. Als Wert erhält das „name“-Attribut dieses Elements den Wert des „name“-Attributs der aktuell betrachteten Aktivitätsverbindung. Das „operation“-Attribut erhält als Wert den beim „operation“-Attribut einer der in der Aktivitätserbindung angegebenen sendenden oder empfangenden Aktivitäten angegebenen Namen einer Operation. Für alle in einer Aktivitätsverbindung angegebenen sendenden und empfangenden Aktivitäten ist die angegebene Operation identisch.

Für jede Aktivitätsverbindung kennen wir den NCName des *Port Types*, sowie mittels der Funktion $prefix_{URI}$ den Namensraumpräfix der im „uri“-Attribut vermerkten URI. Der beim „portType“-Attribut des <messageLink>-Elements anzugebende Wert setzt sich aus dem mittels $prefix_{URI}$ bestimmten Namensraumpräfix und dem NCName des *Port Types*, der im „ptype“-Attribut der Aktivitätsverbindung vermerkt wurde, zusammen.

Das <properties>-Element hat als Kinder <property>-Elemente. Ein <property>-Element besitzt die beiden Attribute „name“ und „WSDLproperty“. Für jedes *corr_prop*-Objekt, das in *CORR_PROPs* enthalten ist, erzeugen wir ein neues <property>-Element. Das „name“-Attribut erhält als Wert den im „NCname“-Attribut eines *corr_prop*-Objektes festgehaltenen NCName. Das „WSDLproperty“-Attribut erhält als Wert die Konkatenation des Namensraumpräfixes, den wir mit der Funktion $prefix_{URI}$ erhalten, wenn wir ihr die im „namespace_URI“-Attribut des aktuell betrachteten *corr_prop*-Objektes festgehaltenen URI übergeben, und dem im „property_name“-Attribut des *corr_prop*-Objektes festgehaltenen ursprünglichen NCName.

3.9. Generierung der Participant Behavior Descriptions

Zuletzt erzeugen wir zu jedem gegebenen BPEL-Prozess eine *Participant Behavior Description*. Eine PBD erhalten wir, indem wir in einem gegebenen BPEL-Prozess Änderungen vornehmen. Die in Abschnitt 3.2 eingefügten „wsu:id“-Attribute bleiben erhalten. Ebenso die in Abschnitt 3.6 geänderten „properties“-Attribute der <correlationSet>-Elemente.

Ist der gegebene BPEL-Prozess ein ausführbarer Prozess, so entfernen wir den folgenden Eintrag:
xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable".
Stattdessen fügen wir den folgenden Eintrag ein:
xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract".

Ebenfalls in das <process>-Element fügen wir das Attribut „abstractProcessProfile“ hinzu, womit auf ein Profil verwiesen wird. Wir verwenden eine Variante des in BPEL4Chor

verwendeten *Abstract Process Profile for Participant Behavior Descriptions*. Diese Variante besitzt dieselben Eigenschaften wie das *Abstract Process Profile for Participant Behavior Descriptions*, nur dass zusätzlich noch die <exit>-Aktivität erlaubt ist. Die URI dieses Profils ist „urn:HPI_IAAS:choreography:profile:2006/12-X“.

Wir durchsuchen nun per Tiefensuche den Prozess. Finden wir ein <partnerLinks>-Element, so löschen wir dieses Element mitsamt seinen gesamten Kindern. Finden wir eine <forEach>-Aktivität, so müssen wir überprüfen, ob diese Aktivität in der Menge *FE* enthalten ist. Ist die Aktivität dort enthalten, so wissen wir, dass diese Schleife über eine Menge von Instanzen eines Partnerprozesses iteriert. Ist dies der Fall, so entfernen wir die <startCounterValue>- und <finalCounterValue>-Elemente, die Kinder des <forEach>-Elements sind. Ebenso entfernen wir bei der <forEach>-Aktivität das „counterName“-Attribut.

Finden wir eine <assign>-Aktivität, bei der etwas von einem *Partner Link* oder in einen *Partner Link* kopiert wird, so werden die entsprechenden <copy>-Elemente entfernt. Falls in der betreffenden <assign>-Aktivität keine anderen <copy>-Elemente vorkommen, so wird die gesamte <assign>-Aktivität entfernt. Befindet sich die <assign>-Aktivität innerhalb einer <flow>-Aktivität und es befinden sich innerhalb dieser <assign>-Aktivität <source>- oder <target>-Elemente, so wird diese zu löschende <assign>-Aktivität durch eine <opaqueActivity>-Aktivität mit den gleichen <source>- und <target>-Elementen ersetzt.

Im selben Durchgang entfernen wir aus allen kommunizierenden Aktivitäten die „portType“-, „operation“- und „partnerLink“-Attribute. Finden wir eine kommunizierende Aktivität, die in keiner Aktivitätsverbindung vorkommt, so entfernen wir diese Aktivität. Befindet sich diese Aktivität innerhalb einer <flow>-Aktivität, so wird diese Aktivität wie zuvor die entsprechenden <assign>-Aktivitäten durch eine <opaqueActivity>-Aktivität ersetzt.

Die *Participant Groundings*, die *Participant Topology* und jede Participant Behaviour Description wird als eigene XML-Datei ausgegeben.

3.10. Implementierung eines Prototyps

Der zu implementierende Prototyp verwirklicht den vorgestellten Ansatz, mit der Ausnahme, dass die <flow>-Aktivität und alles was damit zusammenhängt, hier nicht berücksichtigt wird. Ebenso wird hier der Fall außer Acht gelassen, dass in einer Aktivitätsverbindung unterschiedliche sendende Teilnehmermengen und Teilnehmerreferenzen oder unterschiedliche empfangende Teilnehmerreferenzen auftreten können. Folglich existiert keine Funktion *NEW-PARTICIPANT-REFERENCES*. Tritt der Fall auf, dass mehrere unterschiedliche Teilnehmerreferenzen als Empfänger angegeben sind, so geben wir als „receiver“ beim betreffenden *Message Link* in der *Participant Topology* eine Menge von NCNames an. Somit wird in diesem Fall keine gültige Choreographie erzeugt.

Die BPEL-Spezifikation verlangt, dass der Name eines *Partner Links* innerhalb seines Gültigkeitsbereiches einmalig ist (vgl. [JE07]). Darüberhinaus verlangen wir, dass der Name eines *Partner Links* innerhalb seines BPEL-Prozesses einmalig ist, da wir einen *Partner Link* über seinen Namen und den Prozess, in dem er deklariert wird, identifizieren. Dies können wir mittels Umbenennung erzwingen. Finden wir bei der Suche nach *Partner Links* einen *Partner Link*, dessen Name in diesem BPEL-Prozess bereits verwendet wurde, so benennen wir diesen, sowohl bei der Deklaration, als auch bei allen kommunizierenden Aktivitäten innerhalb des entsprechenden Gültigkeitsbereiches, bei denen dieser *Partner Link* verwendet wird, um.

Die verwendete Programmiersprache ist Java. Zur Arbeit mit den XML-Dokumenten wird JDOM verwendet. JDOM speichert ein XML-Dokument als Baum ab, der aus Java-Objekten besteht (siehe [Ull07]). Jedes Element des Baumes stellt ein eigenes Objekt dar.

Die Funktionen und Klassen unterscheiden sich nur geringfügig von den in diesem Kapitel vorgestellten Klassen und Funktionen. Neben diesen existieren einige weitere Klassen und Funktionen. Erläuterungen zum Vorgehen, zum Zweck und zur Funktionalität dieser Klassen und Funktionen finden sich im Quellcode.

Im Anhang findet sich ein Beispiel für die Umwandlung gegebener BPEL-Prozesse in eine BPEL4Chor-Beschreibung mit diesem Prototyp.

KAPITEL 4

ZUSAMMENFASSUNG UND AUSBLICK

In dieser Arbeit wurde untersucht, wie aus gegebenen BPEL-Prozessen und WSDL-Definitionen eine BPEL4Chor-Beschreibung generiert werden kann.

Die Arbeit besteht im Wesentlichen aus drei zentralen Punkten:

- Das Finden von Aktivitätsverbindungen (siehe Abschnitt 3.2), mit denen wir später *Message Links* erzeugen.
- Die Bestimmung der Beziehungen, die zwischen zwei BPEL-Prozessen zu bestimmten Zeitpunkten bestehen (siehe Abschnitt 3.3).
- Das Erzeugen von Teilnehmermengen und Teilnehmerreferenzen (siehe Abschnitt 3.4 und Abschnitt 3.5).

Mit diesem Wissen werden anschließend die drei Bestandteile einer BPEL4Chor-Beschreibung generiert (siehe Abschnitte 3.7, 3.8 und 3.9).

Im Laufe der Bestimmung dieser Punkte mussten einige Fälle ausgeschlossen und einige Annahmen getroffen werden (siehe Anhang A).

Einige Punkte sind noch offen und können die Grundlage für weitere Arbeiten sein. So kann versucht werden, die bisher nicht abgedeckten Fälle und Aktivitäten, sowie Handler und die Unterscheidung nach Korrelationsmengen miteinzubeziehen. Im Zusammenhang mit letzterem bietet sich eine Datenflussanalyse an.

Ein noch unbearbeiteter Themenkomplex im Zusammenhang mit BPEL4Chor ist das Weiterreichen von Endpunktreferenzen in Nachrichten. Da in BPEL4Chor keine *Partner Links* vorhanden sind, muss dies umgangen werden, indem bei *Message Links* die Teilnehmerreferenz angegeben wird, deren Endpunktreferenz dort übertragen wird (siehe Abschnitt 2.2.3).

Ist kein Initiator gegeben, so kann versucht werden, einen Initiator zu generieren. Dies hätte den Vorteil, dass bei der Generierung einer PBD unter Umständen keine kommunizierenden Aktivitäten entfernt oder durch <opaqueActivity>-Aktivitäten ersetzt werden müssen.

ANHANG A

ANHANG

Hier findet sich eine Zusammenfassung der in Kapitel 3 ausgeschlossenen Fälle, getroffenen Einschränkungen und Annahmen. Zu den angegebenen Fällen wird, wo bekannt, ein möglicher Lösungsansatz skizziert. Bei jedem Fall wird auf den entsprechenden Abschnitt verwiesen, in dem der Fall ursprünglich erwähnt wurde.

A.1. Ausgeschlossene Fälle, Einschränkungen und Annahmen

- **Nicht behandelt: Unterscheidung von Nachrichten nach Korrelationsmengen (Abschnitt 3.1)**
  Korrelationsmengen werden in dieser Arbeit, abseits von Abschnitt 3.6, nicht behandelt. Folglich können keine Fälle behandelt werden, in denen in sequentieller Folge mehrere Instanzen eines Partnerprozesses eingebunden werden (siehe Abbildung A.1). Dasselbe gilt für den Fall, dass diese Aktivitäten in einem Flow vorkommen und dort parallel ausgeführt werden dürfen. (Abschnitt 3.2) Analog dazu schließen wir aus, dass ein Prozess mit

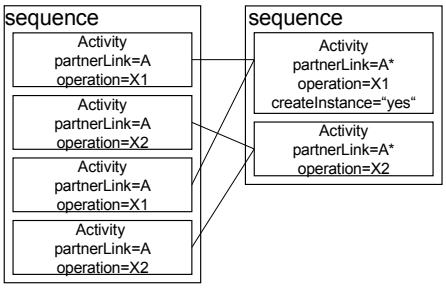

Abbildung A.1.: Sequentielles Einbinden mehrere Instanzen eines Partnerprozesses

sequentiell aufeinanderfolgenden Aktivitäten mehrere Instanzen eines Partnerprozesses aus einer Menge von Instanzen auswählt. Dasselbe gilt für den Fall, dass diese Aktivitäten in einem Flow vorkommen und dort parallel ausgeführt werden dürfen.

Ebenso müssen Fälle ausgeschlossen werden, in denen Aktivitäten zu einem Zeitpunkt aktiv sind, die in Konflikt zueinander stehen. Ein Beispiel wären zwei <receive>-Aktivitäten, die denselben *Partner Link* und dieselbe Operation besitzen. (Abschnitt 3.2)

- **Nicht behandelt: Weiterreichen von Endpunktreferenzen (Abschnitt 3.1)**
  Das Weiterreichen von Endpunktreferenzen wird in dieser Arbeit nicht behandelt.
- **Nicht behandelt: Handler (Abschnitt 3.1)**
  Handler werden in dieser Arbeit nicht behandelt. Die Problematik besteht hierbei, dass für eine Instanz eines Prozesses mehrere Instanzen eines zugehörigen *Event Handlers* gleichzeitig aktiv sein können. Da *Fault Handler* nicht vorkommen dürfen, fordern wir, dass stets „suppressJoinFailure = yes“ gilt.
- **Nicht behandelt: <extensionActivity>-Aktivität (Abschnitt 3.1)**
  Die <extensionActivity>-Aktivität wird in dieser Arbeit nicht behandelt.
- **Vereinfachung: Nur eine instanzerzeugende Aktivität pro Prozess (Abschnitt 3.1)**
  Die Problematik hierbei wäre, dass mehrere Instanzen eines BPEL-Prozesses existieren könnten und wir die Zuordnung der Nachrichten über Korrelationsmengen vornehmen müssten. Wie bereits oben erwähnt, behandeln wir dies in dieser Arbeit nicht.
- **Einschränkung: Einschränkung der abtsrakten Prozesse (Abschnitt 3.2)**
  Ein abstrakter Prozess ist entweder Initiator (siehe Abschnitt 3.2), oder eine instanzerzeugende Aktivität ist vorhanden.
- **Einschränkung: Weitere Einschränkung der abtsrakten Prozesse (Abschnitt 3.2)**
  Bei kommunizierenden Aktivitäten müssen die „variable“-,„inputVariable“- und „outputVariable“-Attribute oder die entsprechenden <fromParts>- und <toParts>-Elemente angegeben werden. Dies benötigen wir, da wir bei einer <invoke>-Aktivität direkt erkennen wollen, ob diese eine synchrone Kommunikation ermöglicht.
  Lösungsvorschlag: Bei einem alternativen Vorgehen, bei dem wir beim Partnerprozess nach einer <reply>-Aktivität suchen, die an die empfangende Aktivität gebunden ist, könnte man auf diese Einschränkung verzichten.
- **Ausgeschlossener Fall: Aktivitäten bei verschiedenen Prozessen, die mit derselben Aktivität eines Prozesses kommunizieren (Abschnitt 3.1)**
  Dieser Fall wird von BPEL4Chor bisher nicht unterstützt. Ein Beispiel für diesen Fall findet sich in Abbildung A.2.

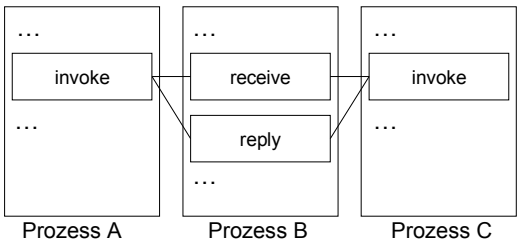

Abbildung A.2.: Invoke-Aktivitäten bei verschiedenen Prozessen, die mit derselben Receive-Aktivität kommunizieren

- **Ausgeschlossener Fall: Sequentielles Abfangen einer Schleife (Abschnitt 3.2)**
  Ausgeschlossen wird der Fall, bei dem in einem BPEL-Prozess eine Schleife durchlaufen wird, im Partnerprozess dies jedoch sequentiell bearbeitet wird (siehe Abbildung A.3). Damit folgt ebenfalls, dass Schleifen bei verschiedenen Prozessen gleich lang laufen müssen, da dies ansonsten bei einem Prozess sequentiell abgefangen werden müsste.

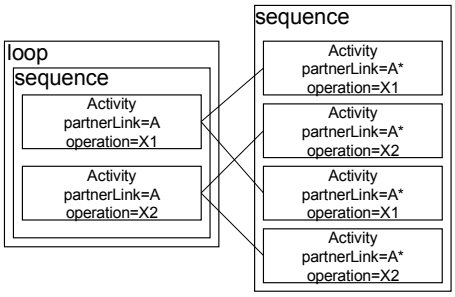

Abbildung A.3.: Sequentielles Abarbeiten einer Schleife beim Partnerprozess

- **Annahme: Eine Schleife wird mindestens einmal durchlaufen (3.2)**
  Wir gehen davon aus, dass eine Schleife mindestens einmal durchlaufen wird.

  Lösungsvorschlag: Wir behandeln diesen Zweig analog zu zu einer Verzweigung. Man muss zwischen den beiden Fällen „Schleife wird mindestens einmal durchlaufen“ und „Schleife wird kein einziges mal durchlaufen“ unterscheiden.

- **Ausgeschlossener Fall: Überkreuzende <invoke>- und <receive>-Aktivitäten (Abschnitt 3.2)**
  Ausgeschlossen werden Fälle, in denen wir bei zwei miteinander kommunizierenden BPEL-Prozessen auf je eine asynchrone <invoke>-Aktivität treffen, die nach dem Senden jeweils beendet wird, und im Anschluss daran die jeweils passende <receive>-Aktivität zur schon beendeten <invoke>-Aktivität des Partnerprozesses aktiv wird (siehe Abbildung A.4).

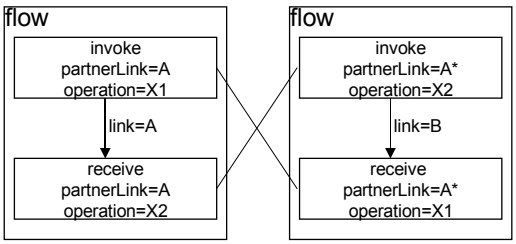

Abbildung A.4.: Asynchrone <invoke>-Aktivitäten

  Lösungsvorschlag: Wird zu einem Zeitpunkt keine Aktivitätsverbindung gefunden, dann suchen wir von diesen <invoke>-Aktivitäten aus nach den nächsten kommunizierenden Aktivitäten. Allerdings müssen wir uns diese <invoke>-Aktivitäten merken, damit von dort aus nicht später erneut nach kommunizierenden Aktivitäten gesucht wird, wenn mit diesen <invoke>-Aktivitäten Aktivitätsverbindungen gebildet werden.

- **Ausgeschlossener Fall: n:m-Kommunikation (Abschnitt 3.3)**
  Ausgeschlossen wird n:m-Kommunikation, wie sie etwa bei Peer-to-Peer-Anwendungen Verwendung findet.

- **Vereinfachung: Ein BPEL-Prozess befindet sich höchstens in der Instanzklasse n^1 (Abschnitt 3.3)**
  Da Fälle, in denen von einem BPEL-Prozess mehr als n Instanzen existieren, in der Praxis recht selten sind, wollen wir uns hier auf Fälle beschränken, in denen von einem BPEL-Prozess eine Instanz oder n Instanzen existieren.

- **Ausgeschlossener Fall: n:1-Beziehung zu Beginn (Abschnitt 3.3)**
  Bindet ein Prozess einen neuen BPEL-Prozess ein, von dem dann eine Instanz kreiert wird, dann schließen wir den Fall aus, dass eine n:1-Beziehung besteht, wobei die Instanzkreierung auf der 1er-Seite stattfindet. In einem solchen Fall würden zwei Aktivitäten sequentiell aufeinanderfolgen, die zu einer Aktivität eines anderen BPEL-Prozesses passen (siehe Abbildung A.5). Diesen Fall haben wir bereits ausgeschlossen.

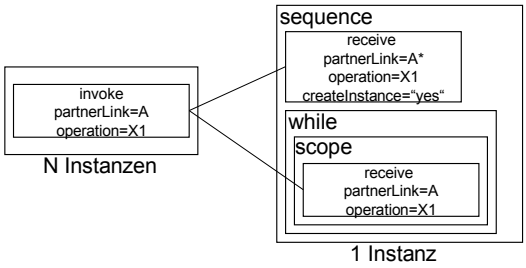

Abbildung A.5.: n:1-Beziehung beim Einbinden eines neuen Prozesses

Lösungsvorschlag: Solch einen Spezialfall gesondert betrachten.

- **Annahme: Einbinden von n Instanzen eines Partnerprozesses nur über Endpunktreferenzen (Abschnitt 3.3)**
  Die Unterscheidung über Eigenschaften von Nachrichten (*Message Properties*) wurde bereits ausgeschlossen.

- **Vereinfachung: Gleiche Anzahl der Instanzen in verschiedenen Zweigen (Abschnitt 3.3)**
  Existieren zu einer instanzerzeugenden Aktivität in einem BPEL-Prozess mehrere passende <invoke>-Aktivitäten beim Partnerprozess, dann fordern wir, dass sich der BPEL-Prozess in allen möglichen Fällen in derselben Instanzklasse befindet. Fälle, in denen dies nicht zutrifft, schließen wir aus.

- **Ausgeschlossener Fall: Dieselben Aktivitäten in verschiedenen Zweigen in unterschiedlicher Schleifentiefe (Abschnitt 3.3)**
  Ausgeschlossen wird der Fall, dass bei einer Verzweigung in verschiedenen Zweigen in Sachen kommunizierender Aktivitäten dasselbe geschieht, nur einmal innerhalb einer Schleife, das andere mal nicht innerhalb einer Schleife (siehe Abbildung A.6).

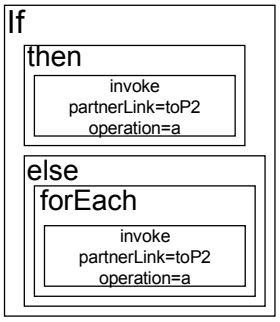

Abbildung A.6.: Dieselben Aktivitäten in verschiedenen Zweigen in unterschiedlicher Schleifentiefe

- **Annahme: Durchlaufen von Schleifen bei verschiedenen Prozessen (Abschnitt 3.3)**
  Fälle, wie in Abbildung A.7 oder A.8 schließen wir aus, da diese Schleifen nicht gleich oft durchlaufen werden.

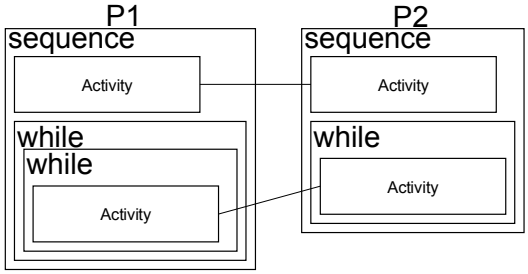

Abbildung A.7.: Unterschiedliche Schleifen in verschiedenen Prozessen (I)

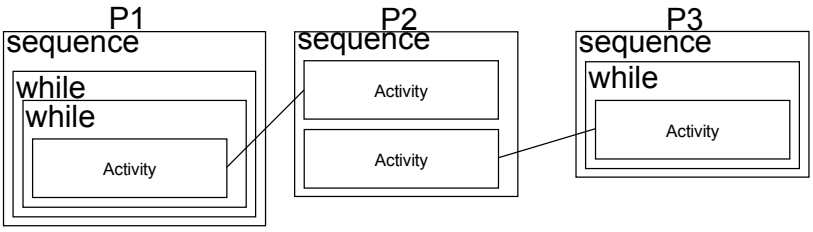

Abbildung A.8.: Unterschiedliche Schleifen in verschiedenen Prozessen (II)

- **Annahme: Gleiche Reihenfolge der zueinander passenden Schleifen bei verschiedenen Prozessen (Abschnitt 3.3)**
  Wir fordern, dass die zueinander passenden Schleifen bei verschiedenen Prozessen in derselben Reihenfolge vorliegen, wie im Beispiel in Abbildung A.9.

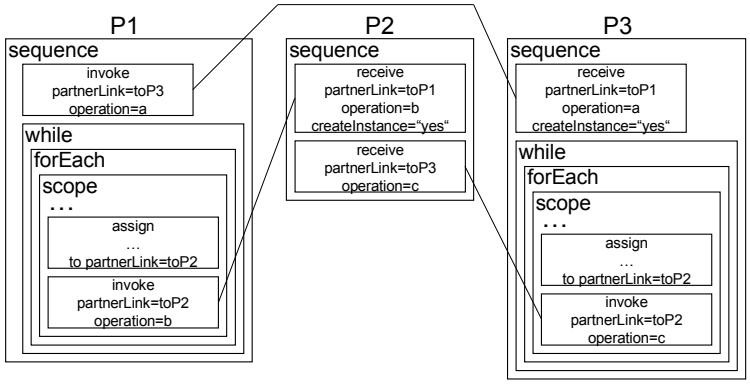

Abbildung A.9.: Gleiche Reihenfolge der Schleifen

- **Annahme: Einbinden von n Instanzen eines Partnerprozesses über eine 1:1-Beziehung (Abschnitt 3.3)**
  Bindet ein BPEL-Prozess, von dem n Instanzen existieren, einen anderen BPEL-Prozess ein, und es herrscht zu diesem Zeitpunkt eine 1:1-Beziehung zwischen diesen BPEL-Prozessen, so gehen wir davon aus, dass sich der eingebundene BPEL-Prozess ebenfalls in der Instanzklasse n^1 befindet.

A.2. Beispiel: Umwandlung gegebener BPEL-Prozesse in eine BPEL4Chor Beschreibung mit dem Prototyp

Zum Beispiel aus Abschnitt 2.2.1 (siehe Abbildung A.10) haben wir folgende abstrakte BPEL-Prozesse gegeben.

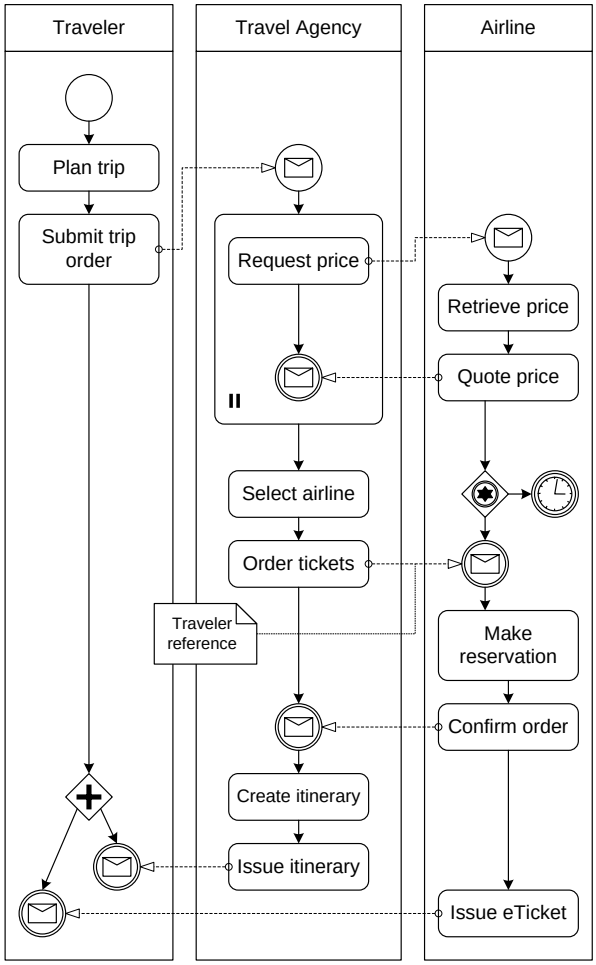

Abbildung A.10.: Beispiel einer Choreografie [DKLW07a]

Zum Prozess „Traveler“ ist gegeben:

```
<?xml version="1.0" encoding="UTF−8"?>
<process name="Traveler"
        targetNamespace="http://www.example.com/1"
        xmlns="http://docs.oasis−open.org/wsbpel/2.0/process/abstract"
        abstractProcessProfile="http://docs.oasis−open.org/wsbpel/2.0/process/
                                abstract/ap11/2006/08"
        xmlns:ns="http://www.example.com/2">


        <partnerLinks>
                <partnerLink name="toAgency" partnerLinkType="ns:TTA"
                        myRole="traveler" partnerRole="agency"/>
```

```
                <partnerLink name="toAirline" partnerLinkType="ns:TA"
                        myRole="traveler" partnerRole="airline"/>
        </partnerLinks>

        <sequence>

        <invoke name="invokeAgency"
                partnerLink="toAgency"
                operation="init"
                inputVariable="opaque"
                >
        </invoke>

        <receive name="getItinerary"
                partnerLink="toAgency"
                operation="itinerary"
                variable="opaque"
                >
        </receive>

        <receive name="geteTicket"
                partnerLink="toAirline"
                operation="eTicket"
                variable="opaque"
                >
        </receive>

        </sequence>

</process>
```

Listing A.1: BPEL-Prozess Traveler

Zum Prozess „Agency“ ist gegeben:

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="Agency"
        targetNamespace="http://www.example.com/1"
        xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
        abstractProcessProfile="http://docs.oasis-open.org/wsbpel/2.0/process/
                                abstract/ap11/2006/08"
        xmlns:ns="http://www.example.com/2">

        <partnerLinks>
                <partnerLink name="toTraveler" partnerLinkType="ns:TTA"
                        myRole="agency" partnerRole="traveler"/>
                <partnerLink name="toSelectedAirline" partnerLinkType="ns:TAA"
                        myRole="agency" partnerRole="airline"/>
        </partnerLinks>


        <sequence>

        <receive name="invokation"
                partnerLink="toTraveler"
                operation="init"
```

```
        variable="opaque"
        createInstance="yes"
        >
</receive>

<forEach counterName="Counter">
        <startCounterValue>0</startCounterValue>
        <finalCounterValue>count($set/)</finalCounterValue>

        <scope>

        <partnerLinks>
                <partnerLink name="toAirline" partnerLinkType="ns:TAA"
                        myRole="agency" partnerRole="airline"/>
        </partnerLinks>

                <assign>
                        <copy>
                                <from >
                                        <query>[$Counter]</query>
                                </from>
                                <to partnerLink="toAirline"/>
                        </copy>
                </assign>


                <invoke name="requestPrice"
                        partnerLink="toAirline"
                        operation="requestPrice"
                        inputVariable="opaque"
                        >
                </invoke>

                <receive name="getPrice"
                        partnerLink="toAirline"
                        operation="getPrice"
                        variable="opaque"
                        >
                </receive>

        </scope>

</forEach>

<invoke name="orderTicket"
        partnerLink="toSelectedAirline"
        operation="TicketOrder"
        inputVariable="opaque"
        >
</invoke>

<invoke name="IssueItinerary"
        partnerLink="toTraveler"
        operation="itinerary"
        inputVariable="opaque"
```

```
            >
        </invoke>

        </sequence>

</process>
```

Listing A.2: BPEL-Prozess Agency

Zum Prozess „Airline“ ist gegeben:

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="Airline"
        targetNamespace="http://www.example.com/1"
        xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
        abstractProcessProfile="http://docs.oasis-open.org/wsbpel/2.0/process/
                                abstract/ap11/2006/08"
        xmlns:ns="http://www.example.com/2">

        <partnerLinks>
                <partnerLink name="toAgency" partnerLinkType="ns:TAA"
                        myRole="airline" partnerRole="agency"/>
                <partnerLink name="toTraveler" partnerLinkType="ns:TA"
                        myRole="airline" partnerRole="traveler"/>
        </partnerLinks>

        <sequence>

                <receive name="invokation"
                        partnerLink="toAgency"
                        operation="requestPrice"
                        variable="opaque"
                        createInstance="yes"
                        >
                </receive>

                <invoke name="sendPrice"
                        partnerLink="toAgency"
                        operation="getPrice"
                        inputVariable="opaque"
                        >
                </invoke>

                <pick>
                        <onMessage name="TicketOrder"
                                partnerLink="toAgency"
                                operation="TicketOrder"
                                variable="opaque"
                        >

                                <invoke name="sendeTicket"
                                        partnerLink="toTraveler"
                                        operation="eTicket"
                                        inputVariable="opaque"
                                >
                                </invoke>
```

```
                    </onMessage>

                    <onAlarm>
                    <for>'P1D'</for>
                            <empty/>
                    </onAlarm>
                </pick>


        </sequence>


</process>
```

Listing A.3: BPEL-Prozess Airline

Als WSDL-Definitionen sind gegeben:

```
<wsdl:definitions
targetNamespace="http://www.example.com/2"
xmlns:pos="http://www.example.com/3"
xmlns:sns="http://www.example.com/4"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 ...

        <plnk:partnerLinkType name="TTA">
                <plnk:role name="traveler"
                        portType="pos:travelerPT" />
                <plnk:role name="agency"
                        portType="pos:agencyPT" />
        </plnk:partnerLinkType>

        <plnk:partnerLinkType name="TAA">
                <plnk:role name="agency"
                        portType="pos:agencyPT" />
                <plnk:role name="airline"
                        portType="pos:airlinePT" />
                </plnk:partnerLinkType>

        <plnk:partnerLinkType name="TA">
                <plnk:role name="traveler"
                        portType="pos:travelerPT" />
                <plnk:role name="airline"
                        portType="pos:airlinePT" />
        </plnk:partnerLinkType>

</wsdl:definitions>
```

Listing A.4: Ausschnitt aus den WSDL-Definitionen

Als Ergebnisse erhalten wir mit dem Prototyp folgende Dokumente:

Die PBD für den Prozess „Traveler“:

```
<process xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
         xmlns:ns="http://www.example.com/2"
         xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
         oasis-200401-wss-wssecurity-utility-1.0.xsd"
         name="Traveler"
         targetNamespace="http://www.example.com/1"
         abstractProcessProfile="urn:HPI_IAAS:choreography:profile:2006/12-X">
  <sequence>
    <invoke name="invokeAgency" inputVariable="opaque" wsu:id="8" />
    <receive name="getItinerary" variable="opaque" wsu:id="9" />
    <receive name="geteTicket" variable="opaque" wsu:id="10" />
  </sequence>
</process>
```

Listing A.5: PBD für Traveler

Die PBD für den Prozess „Agency“:

```
<process xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
         xmlns:ns="http://www.example.com/2"
         xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
         oasis-200401-wss-wssecurity-utility-1.0.xsd"
         name="Agency"
         targetNamespace="http://www.example.com/1"
         abstractProcessProfile="urn:HPI_IAAS:choreography:profile:2006/12-X">
  <sequence>
    <receive name="invokation" variable="opaque" createInstance="yes" wsu:id="1" />
    <forEach wsu:id="2">
      <scope wsu:id="3">
        <invoke name="requestPrice" inputVariable="opaque" wsu:id="4" />
        <receive name="getPrice" variable="opaque" wsu:id="5" />
      </scope>
    </forEach>
    <invoke name="orderTicket" inputVariable="opaque" wsu:id="6" />
    <invoke name="IssueItinerary" inputVariable="opaque" wsu:id="7" />
  </sequence>
</process>
```

Listing A.6: PBD für Agency

Die PBD für den Prozess „Airline“:

```
<process xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
         xmlns:ns="http://www.example.com/2"
         xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
         oasis-200401-wss-wssecurity-utility-1.0.xsd"
         name="Airline"
         targetNamespace="http://www.example.com/1"
         abstractProcessProfile="urn:HPI_IAAS:choreography:profile:2006/12-X">
  <sequence>
    <receive name="invokation" variable="opaque" createInstance="yes" wsu:id="11" />
    <invoke name="sendPrice" inputVariable="opaque" wsu:id="12" />
    <pick>
      <onMessage name="TicketOrder" variable="opaque" wsu:id="13">
        <invoke name="sendeTicket" inputVariable="opaque" wsu:id="14" />
```

```
      </onMessage>
      <onAlarm>
        <for>'P1D'</for>
        <empty />
      </onAlarm>
    </pick>
  </sequence>
</process>
```

Listing A.7: PBD für Airline

Als *Participant Topology* erhalten wir:

```
<?xml version="1.0" encoding="UTF−8"?>
<topology xmlns:Agency="http://www.example.com/1"
    xmlns:Traveler="http://www.example.com/1" xmlns:Airline="http://www.example.com/1"
    name="topology" targetNamespace="urn:chor">
  <participantTypes>
    <participantType name="Agency_type" participantBehaviorDescription="Agency:Agency" />
    <participantType name="Traveler_type" participantBehaviorDescription="Traveler:Traveler" />
    <participantType name="Airline_type" participantBehaviorDescription="Airline:Airline" />
  </participantTypes>
  <participants>
    <participantSet name="Airline_set_1" type="Airline_type" forEach="Agency:2">
      <participant name="current_Airline_1" forEach="Agency:2" />
      <participant name="selected_Airline_1" />
    </participantSet>
    <participant name="Agency" selects="Airline_set_1" type="Agency_type" />
    <participant name="Traveler" selects="Agency" type="Traveler_type" />
  </participants>
  <messageLinks>
    <messageLink name="1" sender="Traveler" sendActivity="8" receiver="Agency"
        receiveActivity="1" messageName="1" />
    <messageLink name="2" sender="Agency" sendActivity="4" receiver="current_Airline_1"
        receiveActivity="11" messageName="2" />
    <messageLink name="3" sender="current_Airline_1" sendActivity="12" receiver="Agency"
        receiveActivity="5" messageName="3" />
    <messageLink name="4" sender="Agency" sendActivity="6" receiver="selected_Airline_1"
        receiveActivity="13" messageName="4" />
    <messageLink name="5" sender="Agency" sendActivity="7" receiver="Traveler"
        receiveActivity="9" messageName="5" />
    <messageLink name="6" sender="selected_Airline_1" sendActivity="14" receiver="Traveler"
        receiveActivity="10" messageName="6" />
  </messageLinks>
</topology>
```

Listing A.8: *Participant Topology*

Als *Participant Groundings* erhalten wir:

```
<?xml version="1.0" encoding="UTF−8"?>
<groundings xmlns:top="urn:chor" xmlns:ns1="http://www.example.com/3"
    topology="top:topology">
  <messageLinks>
    <messageLink name="1" portType="ns1:agencyPT" operation="init" />
    <messageLink name="2" portType="ns1:airlinePT" operation="requestPrice" />
```

```
    <messageLink name="3" portType="ns1:agencyPT" operation="getPrice" />
    <messageLink name="4" portType="ns1:airlinePT" operation="TicketOrder" />
    <messageLink name="5" portType="ns1:travelerPT" operation="itinerary" />
    <messageLink name="6" portType="ns1:travelerPT" operation="eTicket" />
  </messageLinks>
  <properties />
</groundings>
```

Listing A.9: *Participant Groundings*

LITERATURVERZEICHNIS

[Bac06] BACHER, Andreas: *Choreografie von Geschäftsprozessen: Beschreibung und Werkzeugunterstützung*, Institut für Architektur von Anwendungssystemen der Universität Stuttgart, Diplomarbeit, Oktober 2006

[BDO05] BARROS, Alistair ; DUMAS, Marlon ; OAKS, Phillipa: *A Critical Overview of the Web Services Choreography Description Language (WS-CDL)*. Artikel von BPTrends, 2005

[DKLW07a] DECKER, Gero ; KOPP, Oliver ; LEYMANN, Frank ; WESKE, Mathias: BPEL4Chor: Extending BPEL for Modeling Choreographies. In: *ICWS*, IEEE Computer Society, 2007, 296–303

[DKLW07b] DECKER, Gero ; KOPP, Oliver ; LEYMANN, Frank ; WESKE, Mathias: *BPEL4Chor: Extending BPEL for Modeling Choreographies Technical Report*. Unveröffentlichter technischer Bericht des Instituts für Architektur von Anwendungssystemen der Universität Stuttgart und dem Hasso-Plattner-Institut der Universität Potsdam, Version vom 18. 4. 2007

[JE07] JORDAN, Diane ; EVDEMON, John: *Web Services Business Process Execution Language Version 2.0*. Organization for the Advancement of Structured Information Standards (OASIS), 11. April 2007

[MKL07] MIETZNER, Ralph ; KOPP, Oliver ; LEYMANN, Frank: *Abstract Syntax of WS-BPEL 2.0*. Unveröffentlichter technischer Bericht des Instituts für Architektur von Anwendungssystemen der Universität Stuttgart, Version vom 17. 8. 2007

[Sch07] SCHURR, Bastian: *Analyse von XPath Ausdrücken in BPEL Prozessbeschreibungen*, Institut für Architektur von Anwendungssystemen der Universität Stuttgart, Diplomarbeit, laufend, 2007

[Ull07] ULLENBOOM, Christian: *Java ist auch eine Insel*. 6. Auflage. Galileo Computing, 2007

[Wic06] WICKENHÄUSER, Andreas: *Business Process Management und Workflow Technologie*. Arbeitspapier der IBM, Dezember 2006

ABBILDUNGSVERZEICHNIS

LISTINGS

ABKÜRZUNGSVERZEICHNIS

BPEL	Business Process Execution Language
BPEL4Chor	Business Process Execution Language for Choreography
BPEL4WS	Business Process Execution Language for Web Services
JDOM	Java Document Object Model
OASIS	Organization for the Advancement of Structured Information Standards
PBD	Participant Behavior Description
SOA	Service-orientierte Architektur
URI	Uniform Resource Identifier
WS-BPEL	Web Service Business Process Execution Language
WSDL	Web Service Definition Language
XML	Extensible Markup Language
XPath	XML Path Language

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Thomas Steinmetz)