Universität Stuttgart

Choreographiebasierte Konsolidierung von interagierenden BPEL-Prozessmodellen

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Sebastian Wagner

aus Staßfurt

Hauptberichter: Prof. Dr. Dr. h.c. Frank Leymann

Mitberichter: Prof. Dr. Mathias Weske

Tag der mündlichen Prüfung: 10. März 2022

Institut für Architektur von Anwendungssystemen
der Universität Stuttgart

2022

Inhaltsverzeichnis

Zusammenfassung

An der Herstellung eines Guts oder bei der Erbringung einer Dienstleistung sind heutzutage verschiedene Unternehmen beteiligt, wobei jedes mit seinen eigenen Prozessen zur Gesamtwertschöpfungskette beiträgt. Um den Arbeitsablauf zu koordinieren, interagieren die Prozesse über Nachrichtenaustausch miteinander. Die beteiligten Unternehmen und die Reihenfolge der Interaktionen werden dabei mittels Choreographien festgelegt.

Die Corona-Pandemie hat vielen Unternehmen die Anfälligkeit ihrer global verteilten Just-in-time-Wertschöpfungsketten vor Augen geführt. Daher geht in jüngster Zeit der Trend zum Insourcing, bei dem die Unternehmen ausgelagerten Prozesse wieder in die eigenen Prozesse eingliedern.

Die Komplexität von Choreographien und Prozessen macht manuelles Insourcing aufwändig, besonders wenn viele Organisationen beteiligt sind. Daher wird in dieser Dissertation eine Methode vorgestellt, mit der das Insourcing von Prozessen, deren Interaktionsverhalten über eine Choreographie spezifiziert ist, automatisiert werden kann. Die Methode hat zum Ziel, aus einer Choreographie einen einzelnen konsolidierten Prozess zu erstellen, der das Verhalten der an der Choreographie beteiligten Prozesse und die Interaktionen zwischen ihnen emuliert.

In dieser Arbeit wird das Verhalten von Choreographien bzw. Prozessen über die erlaubten Zustandstransitionen zwischen deren Aktivitäten definiert. Um das Verhalten einer Choreographie zu beschreiben und mit dem konsolidierten Prozessmodell zu vergleichen, wird ein auf der operationalen Semantik der Choreographiesprache BPEL4Chor und der Prozessmodellierungssprache BPEL basierendes Zustandstransitionsmodell definiert.

Darauf aufbauend wird diskutiert, inwieweit die Konsolidierungsmethode Prozesse wiederherstellen kann, die zuvor durch Outsourcing fragmentiert wurden. Dazu wird untersucht, ob die konsolidierten Prozesse dasselbe Verhalten haben wie die originalen fragmentierten Prozesse.

Die Interaktionsmöglichkeiten von Organisationen und Prozessen in Choreographien können in eine Menge von Interaktionsmustern kategorisiert werden. Es wird gezeigt, dass die Methode, bis auf zwei Ausnahmen, alle Muster konsolidieren kann.

Die praktische Umsetzung der Methode erfolgt mittels der Anwendung Curveé, die Choreographien, die mit BPEL4Chor erstellt wurden, einliest und daraus Prozesse generiert, die in BPEL modelliert sind.

Abstract

Nowadays, various companies are involved in the production of a good or the provision of a service, each contributing to the overall value chain with its own processes. To coordinate the workflow, the processes interact with each other by exchanging messages. The companies involved and the sequence of interactions are defined by means of choreographies.

The Corona pandemic has made many companies aware of the vulnerability of their globally distributed just-in-time value chains. As a result, there has been a recent trend toward insourcing, in which companies reintegrate outsourced processes back into their own processes.

The complexity of choreographies and processes makes manual insourcing costly, especially when many organizations are involved. Therefore, this dissertation presents a method to automate the insourcing of processes whose interaction behavior is specified via a choreography. The method aims to create a single consolidated process from a choreography emulating the behavior of the processes involved in the choreography and the interactions between them.

In this work, the behavior of choreographies or processes is defined by the allowed state transitions between their activities. In order to describe the behavior of a choreography and to compare it with the consolidated process model, a state transition model is defined based on the operational seman-

tics of the choreography language BPEL4Chor and the process modeling language BPEL.

Building on this, the extent to which the consolidation method can restore processes that were previously fragmented by outsourcing is discussed.

The interaction possibilities of organizations and processes in choreographies can be categorized into a set of interaction patterns. It is shown that the method can consolidate all but two of the patterns.

The method is implemented by the tool Curveé, which generates processes modeled in BPEL from choreographies created with BPEL4Chor.

Danksagung

An dem Entstehen dieser Dissertation haben viele Personen direkt oder indirekt mitgewirkt, denen ich herzlich danken möchte. Als erstes möchte ich meinem Doktorvater Prof. Dr. Dr. h. c. Frank Leymann danken, zum einen, weil er mir diese Promotion ermöglicht hat und zum anderen für die spannenden Themengebiete, die ich bei ihm und seinem Institut für die Architektur von Anwendungssystemen (IAAS) der Universität Stuttgart bearbeiten durfte. Prof. Dr. Mathias Weske vom Hasso-Plattner-Institut an der Universität Potsdam danke ich sehr herzlich dafür, dass er sich bereit erklärt hat, als Mitberichter für diese Arbeit zu fungieren.

Mein Dank gehört meinen Kollegen am IAAS, die mich beim Erstellen dieser Dissertation unterstützt und durch ihre Forschungsthemen fachlich bereichert haben. Ich werde unsere Institutsklausuren zwischen den idyllischen Hängen des Schwarzwalds sehr positiv in Erinnerung behalten. Im Besonderen danke ich meinem ehemaligen Kollegen Oliver Kopp, der mir in zahlreichen Diskussionen das Thema Choreographien nahegebracht und mir den Weg aus so mancher fachlichen Sackgasse gezeigt hat.

Mirko Sonntag und Michael Hahn möchte ich dafür danken, dass sie sich bereit erklärt haben, diese umfangreiche Dissertation gegenzulesen und zu

kommentieren. Bernd Förster danke ich dafür, dass er sich als fachfremde Person die Zeit genommen hat, zum sprachlichen Feinschliff beizutragen.

Ein großes Dankeschön gilt meinem grandiosen Freundeskreis, insbesondere Francesca, der mich angetrieben hat, die Dissertation zu beenden und gleichzeitig für die nötige Ablenkung gesorgt hat.

Diese Dissertation möchte ich meinen Eltern Angelika und Andreas Wagner widmen, die meine Leidenschaft für Informatik vom Kindesalter bis hin zu dieser Dissertation aktiv unterstützt haben. Außerdem widme ich die Arbeit meiner Großmutter Eva Wagner, die sehr stolz war, dass ihr Enkel promoviert, aber leider den Abschluss der Arbeit nicht mehr erleben durfte.

Kapitel 1

Einleitung & Motivation

Geschäftsprozesse beschreiben die Ausführungsreihenfolge einer Menge von Schritten oder Aktivitäten, die zur Erreichung von Geschäftszielen notwendig sind [Gyn94], wie die Erbringung einer Dienstleistung oder die Herstellung eines Produkts. Die Aktivitäten und die Reihenfolge ihrer Ausführung kann informell, also zum Beispiel als Freitext, oder aber formal mittels einer entsprechende Sprache beschrieben werden. In dieser Arbeit liegt der Fokus auf *Workflows* [LR00], also Geschäftsprozessen die formal beschrieben werden, um sie mittels Informationssystemen, sogenannter *Workflow Management Systeme (WfMS)*, voll- oder teilautomatisiert auszuführen. Zur formalen Beschreibung von Geschäftsprozessen, d.h. der Erstellung von *Prozessmodellen*, können standardisierte Sprachen wie die *Web Services Business Process Execution Language (BPEL)* [OAS07b] oder die *Business Process Model and Notation (BPMN)* [Obj11] verwendet werden.

Sind aufgrund der Komplexität von Geschäftszielen mehrere Unternehmen an deren Erbringung beteiligt, müssen die einzelnen Prozesse der Unternehmen miteinander kommunizieren, um diese zu erreichen [Wer07]. Internet

Reiseportale bieten beispielsweise die Möglichkeit, Flüge oder Hotels zu buchen. An der Buchung sind neben dem Reiseportal verschiedene weitere Teilnehmer bzw. Organisationen, wie Fluggesellschaften, Kreditkartenfirmen etc. mit ihren individuellen Geschäftsprozessen beteiligt. Das Interaktionsverhalten zwischen ihnen, d.h. in welcher Reihenfolge die Prozesse Nachrichten austauschen müssen, wird über eine *Choreographie* spezifiziert [Pel03]. Abbildung 1.1 zeigt den Auszug einer solchen Choreographie zur Flugbuchung. Einmal täglich empfängt das Reiseportal „Discountflug.org“ ein Kontingent Billigflüge von verschiedenen Fluggesellschaften. Abhängig vom Flugangebot, das ein Kunde aus diesem Kontingent gewählt hat, führt das Reiseportal eine Buchung bei der entsprechenden Fluggesellschaft aus. Diese reserviert einen Sitzplatz und schließt die Buchung ab, nachdem die Kreditkartenfirma „Kreditguru“ die Zahlung bestätigt hat.

Das Verhalten der Teilnehmer wird durch deren Prozessmodelle spezifiziert, zum Beispiel implementieren alle Fluggesellschaften das Prozessmodell $p_{Airline}$. In den Prozessmodellen kann zwischen Aktivitäten unterschieden werden, die die Kommunikation zwischen den Teilnehmern realisieren und denen, die die eigentlichen Geschäftsfunktionen implementieren. Letztere werden in dieser Ausarbeitung als *Geschäftsaktivitäten* bezeichnet. Es kann sich bei ihnen um konkrete ausführbare Aktivitäten handeln, mit denen zum Beispiel ein externes Informationssystem abgefragt wird. Da Prozesse aber kritisch für den Unternehmenserfolg und damit ein Geschäftsgeheimnis sind [LA94], kapseln Geschäftsaktivitäten häufig das konkrete nichtöffentliche Verhalten eines Teilnehmers. Die Geschäftsaktivitäten abstrahieren in diesem Fall also Teil- oder Unterprozesse [KELU10] des Teilnehmers. Folglich unterscheiden sich die konkreten Prozessmodelle der Teilnehmer „Lufthansa“ und „Delta“ voneinander, nur ihr öffentlich sichtbares Verhalten ist dasselbe. Prozessmodelle, die nur öffentliches Verhalten modellieren, werden auch als *abstrakte* Prozessmodelle bezeichnet, im Gegensatz zu *ausführbaren* Prozessmodellen, die das konkrete Verhalten eines Teilnehmers modellieren.

Aufgrund von organisatorischen oder technischen Gründen kann es not-

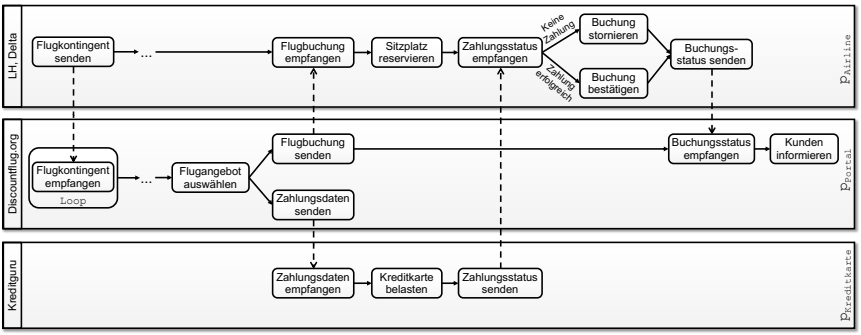

Abbildung 1.1.: Vereinfachte Darstellung einer Choreographie zur Buchung von Flügen über ein Reiseportal (Nachrichtenkanten sind gestrichelt und Kontrollflusskanten durchgängig dargestellt)

wendig sein, die Aktivitäten der Choreographie anderen Teilnehmern zuzuordnen [WFKS12]. Dies erfordert zum einen die Fragmentierung und zum anderen die Konsolidierung von Prozessmodellen.

Ein Ansatz zur automatischen Fragmentierung wurde bereits von Khalaf und Leymann beschrieben [KL06]. Die Fragmentierung enthält als Eingabe ein Prozessmodell, eine Menge von Teilnehmern und Zuordnungen, die jede Aktivität des Prozessmodells genau einem Teilnehmer zuweisen. Als Ergebnis erzeugt die Fragmentierung eine Choreographie bestehend aus Teilnehmerprozessen, die das Verhalten der Teilnehmer mittels der ihn zugeordneten Aktivitäten implementieren. Befinden sich zwei Aktivitäten, die eine Kontroll- oder Datenflussabhängigkeit zueinander hatten, durch die Fragmentierung in unterschiedlichen Teilnehmerprozessen, werden der Kontrollflusszustand bzw. die Daten über Nachrichten zwischen den Aktivitäten ausgetauscht. Die Fragmentierung könnte zum Beispiel dazu genutzt werden, die Aktivität „Sitzplatz reservieren“ an die Tochterfirmen der Fluggesellschaften auszulagern, die dann in den resultierenden Prozessmodellen unabhängig das Geschäftsmodell für die Reservierung eines Sitzplatzes weiterentwickeln können.

Komplementär zur Fragmentierung wird in dieser Dissertation eine Methode vorgestellt, mit der die Geschäftsaktivitäten interagierender Prozessmodelle verschiedener Teilnehmer automatisch konsolidiert, d.h. in ein einzelnes Prozessmodell überführt werden können. Die Prozesskonsolidierung erhält also als Eingabe eine Choreographie und erzeugt als Ausgabe ein einzelnes Prozessmodell. Dieses konsolidierte Prozessmodell enthält alle Geschäftsaktivitäten der Choreographie, die Kommunikationsaktivitäten werden eliminiert. Der Kontrollfluss zwischen den Geschäftsaktivitäten, die aus dem selben Teilnehmerprozess stammen, bleibt im konsolidierten Prozessmodell erhalten. Der Nachrichtenfluss und die draus resultierende Ausführungsreihenfolge zwischen den Geschäftsaktivitäten aus unterschiedlichen Teilnehmerprozessen, werden im konsolidierten Prozessmodell emuliert. Mit der Konsolidierung könnten zum Beispiel die Prozessmodelle der Teilnehmer „Discountflug.org“ und „Kreditguru“ in ein einzelnes Prozessmodell zusammengefasst werden, wenn der Teilnehmer „Discountflug.org“ eine Banklizenz erworben hat und damit die Zahlungen selber verarbeiten darf.

In dieser Arbeit wird die Konsolidierung aller Teilnehmer einer Choreographie formal beschrieben. Soll nur eine Teilmenge konsolidiert werden, zum Beispiel nur die Teilnehmer „Discountflug.org“, „LH“ und „Delta“, müssen die nicht zu konsolidierenden Teilnehmer, hier „Kreditguru“, vorher aus der Choreographie entfernt werden. Die Aktivitäten, die die Kommunikation mit diesen Teilnehmern implementieren, müssen als Geschäftsaktivitäten gekennzeichnet werden. Dies wären in dem Beispiel die Aktivitäten „Zahlungsdaten senden“ bzw. „Zahlungsstatus empfangen“. Die entfernten Teilnehmer werden also als externes Informationssystem abstrahiert und damit nicht bei der Konsolidierung berücksichtigt. Die Konsolidierungsmethode muss also nicht angepasst werden, um nur eine Teilmenge der Teilnehmer in ein einzelnes Prozessmodell zu überführen. Nur der in dieser Arbeit vorgestellte Prototyp, der die Methode implementiert, muss so erweitert werden, dass er als Eingabe neben der Choreographie zusätzlich die Information erhält, welche von deren Teilnehmern konsolidiert werden sollen. Die anderen Teilnehmer kann er automatisch aus der Choreographie entfernen.

Die Konsolidierung wird anhand eines Metamodells erläutert, das sich an BPEL und der Choreographiesprache *BPEL4Chor* [DKLW09] orientiert. BPEL4Chor erlaubt die Modellierung von Choreographien, deren Teilnehmerverhalten über BPEL-Prozesse modelliert ist. Die Prozesskonsolidierung kann neben abstrakten auch ausführbare Prozessmodelle generieren, sofern die Choreographie keine abstrakten BPEL-Aktivitäten enthält.

1.1. Motivation

Beim Outsourcing von Geschäftsbereichen eines Unternehmens werden neben Teilen der Aufbau- auch Teile der Ablauforganisation ausgelagert. Allerdings entscheiden sich Unternehmen aus unterschiedlichen Gründen, wie zum Beispiel wegen Qualitätsproblemen in den ausgelagerten Geschäftsbereichen oder aufgrund fehlender Kontrolle, die Geschäftsbereiche wieder zurück in das Unternehmen zu integrieren [HOH17] (Stichwort Insourcing). Dies erfordert die Reintegration der Ablauforganisation, d.h. der Geschäftsprozesse des Geschäftsbereichs in das Unternehmen. Die Prozesskonsolidierung kann dabei unterstützen, die ausgelagerten Prozesse in ein einzelnes Prozessmodell zusammenzuführen.

Die Prozessmodelle einer Choreographie werden üblicherweise verteilt auf den verschiedenen WfMS der Teilnehmer ausgeführt. Dabei wird für jedes daran teilnehmende Prozessmodell von dem jeweiligen WfMS eine Prozessinstanz erstellt und ausgeführt. Sofern es die rechtlichen und sicherheitstechnischen Rahmenbedingungen erlauben, können die Prozessmodelle der Teilnehmer in ein einzelnes Prozessmodell konsolidiert werden, um die Ausführungskosten wie zum Beispiel die CPU-Last oder den Stromverbrauch [NBF+12] signifikant zu senken. Messungen mit dem WfMS SWoM[1] in [WRK+13] ergaben, abhängig von der Anzahl der Prozessmodelle in der Choreographie und wie häufig diese miteinander kommunizieren, zum Beispiel eine Reduzierung der CPU-Last von bis zu 80 Prozent verglichen

[1] https://www.iaas.uni-stuttgart.de/forschung/projekte/swom/

mit der separaten Ausführung jedes Prozessmodells der Choreographie. Dies liegt an der Reduzierung der Prozessinstanzen, die das WfMS ausführen muss und dass zwischen den Instanzen keine Nachrichten mehr ausgetauscht werden müssen. Es entfällt also die Serialisierung, Übertragung und Deserialisierung der Nachrichten. Das erhöht zusätzlich die Sicherheit, da sensible Informationen im WfMS bleiben und nicht mehr als Nachrichten zwischen den Instanzen ausgetauscht werden müssen. Die Optimierung der Prozessausführung durch Konsolidierung lohnt sich besonders für Instanz-intensive vollautomatisierte Integrationsprozesse, die zum Beispiel kurzlaufende Microservices orchestrieren.

Für BPEL4Chor-Choreographien existieren keine Ausführungsumgebungen, da BPEL4Chor als Beschreibungs- nicht aber als Ausführungssprache für Choreographien konzipiert ist. Daher sind in BPEL4Chor-Choreographien die zur Ausführung der Prozessmodelle notwendigen Informationen oft nicht spezifiziert. Zum einen ist, wie eingangs erwähnt, in den Prozessmodellen der detaillierte Ablauf von internen Aktivitäten, die nicht zur Kommunikation mit den anderen Prozessmodellen dienen, gar nicht oder nur abstrakt dargestellt. Zum anderen fehlen technische Details, wie zum Beispiel die konkreten Endpunkte unter denen die Prozesse oder die von ihnen aufgerufenen Dienste erreichbar sind. Mit der Prozesskonsolidierung können diese Prozessmodelle auf einem WfMS ausführbar gemacht werden. Ein Anwendungsfall dafür ist das von Wagner et al. in [WBK+17] beschriebene Vorgehen, das es ermöglicht, mittels Choreographien und der Prozesskonsolidierung auf dem Cloud Standard TOSCA [OAS13] basierende Managementpläne wiederzuverwenden. Ein TOSCA Managementplan ist ein ausführbares Prozessmodell, dessen Aktivitäten konkrete Schritte implementieren, um eine Applikationskomponente (Datenbank, Web Server, Softwarearchiv etc.) zu administrieren oder in einer bestimmte Umgebungen zu provisionieren. Abhängig von den für den Betrieb der Anwendung benötigten Komponenten und deren Abhängigkeiten untereinander (der Anwendungstopologie), können die Pläne über eine Choreographie kombiniert und deren Informationsaustausch (IP Adressen, Zugangsdaten etc.) festgelegt werden. Die erstellte Choreogra-

phie wird durch die Prozesskonsolidierung in ein einzelnes Prozessmodell transformiert und auf einem WfMS bereitgestellt. Über die Instanzen des Prozessmodells kann die Anwendung dann provisioniert und administriert werden.

1.2. Forschungsbeiträge

Abbildung 1.2 gibt eine Übersicht, über die Forschungsbeiträge dieser Dissertation, die im folgenden vorgestellt werden.

1.2.1. Prozesskonsolidierungsmethode

Der Kern der Ausarbeitung liegt in der Beschreibung der Schritte, mit denen sich die interagierenden Prozessmodelle der Teilnehmer einer Choreographie in ein einzelnes Prozessmodell konsolidieren lassen. Die formale Beschreibung erfolgt dabei anhand des in dieser Arbeit ebenfalls vorgestellten Metamodells. Im ersten Schritt wird ein neues Prozessmodell erstellt, das das konsolidierte Prozessmodell repräsentiert. Das Prozessmodell enthält anfangs weder Kontroll- noch Datenfluss. Um das Verhalten jedes Teilnehmers emulieren zu können, werden in dieses Prozessmodell die von den Teilnehmern ausgeführten Aktivitäten unter Einhaltung ihrer Ausführungsreihenfolge eingefügt.

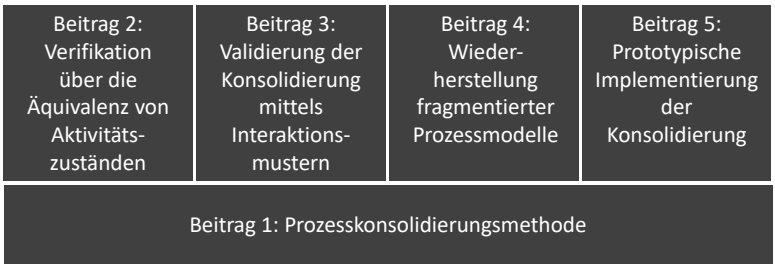

Abbildung 1.2.: Forschungsbeiträge

Die Geschäftsaktivitäten der verschiedenen Teilnehmer haben durch das in der Choreographie spezifizierte Interaktionsverhalten Daten- und Kontrollflussabhängigkeiten zueinander. So muss zum Beispiel die Aktivität „Flugangebot auswählen“ immer vor der Aktivität „Sitzplatz reservieren“ ausgeführt werden, da für die Reservierung Flug- und Passagierdaten bekannt sein müssen. Diese Abhängigkeiten werden im zweiten Schritt – der *Kontrollflussmaterialisierung* – im konsolidierten Prozessmodell mittels entsprechender Kontrollflusskonstrukte für *Basisinteraktionen* abgebildet bzw. emuliert, ohne dass zwischen den Aktivitäten Nachrichten ausgetauscht werden müssen. Eine Basisinteraktion modelliert den Versand und Empfang einer einzelnen Nachricht.

Basisinteraktion können aber auch Teil von komplexeren Interaktionen sein, an denen zum Beispiel mehrere Teilnehmer partizipieren oder bei denen zwei Teilnehmer eine beliebige zur Modellierungszeit unbekannte Anzahl von Nachrichten austauschen (Beitrag 3). Diese Interaktionen werden mittels Schleifen modelliert. Der Empfang der Flugkontingente von mehreren Fluggesellschaften ist ein Beispiel für solch eine komplexe Interaktion. Hier wird über eine Schleife durch die iterative Ausführung der Aktivität „Flugkontingent empfangen“ der Empfang von Nachrichten von verschiedenen Teilnehmern realisiert. Die Abbildung von komplexeren Kommunikationsmustern, die mittels Schleifen modelliert sind, wird im dritten Schritt durch weitere Anpassungen am Kontroll- und Datenfluss des konsolidierten Prozessmodells vorgenommen.

1.2.2. Verifikation über die Äquivalenz von Aktivitätszuständen

Ziel der Konsolidierung ist es, dass das Verhalten des konsolidierten Prozessmodells äquivalent zu dem in der Choreographie modellierten Verhalten der Teilnehmer bzw. derer Prozessmodelle ist. Dazu wird in diesem Beitrag ein Ansatz vorgestellt, das Verhalten von Choreographien und Prozessmodellen über die Reihenfolge von Zustandstransitionen zu beschreiben, die deren Aktivitäten zur Laufzeit durchlaufen dürfen. Die erlaubte Reihenfolge zwischen

den Zustandstransitionen einer Choreographie oder einem Prozessmodell hängt dabei vom modellierten Kontrollfluss zwischen den Aktivitäten ab. Um die Reihenfolge zu ermitteln, werden als Grundlage für diesen Forschungsbeitrag die Zustandsabhängigkeiten definiert, die durch die Kontrollflusskonstrukte des Metamodells zwischen den Aktivitäten impliziert werden. Zustandsbeziehungen werden zum Beispiel durch eine Kontrollflusskante zwischen zwei Aktivitäten impliziert. So darf die Aktivität „Sitzplatz reservieren“ erst in den Ausführungszustand gelangen, nachdem „Flugbuchung empfangen“ erfolgreich ausgeführt wurde, d.h. einen Endzustand erreicht hat. Löst wiederum „Flugbuchung empfangen“ einen Fehler aus und wird damit nicht erfolgreich beendet, kann „Sitzplatz reservieren“ nicht in den Ausführungszustand gelangen.

Basierend auf den Zustandsbeziehungen wird ein Äquivalenzkriterium definiert, das, vereinfacht ausgedrückt, die Choreographie und das konsolidierte Prozessmodell als äquivalent betrachtet, wenn die in der Choreographie zwischen den Geschäftsaktivitäten geltenden Zustandsbeziehungen auch im konsolidierten Prozessmodell gelten. Für jeden der Konsolidierungsschritte wird argumentiert, inwieweit er zur Erfüllung dieses Äquivalenzkriteriums beiträgt.

1.2.3. Validierung der Konsolidierung mittels Interaktionsmuster

Barros, Dumas und ter Hofstede abstrahieren in [BDtH05] eine Menge von Interaktionsmustern, die zwischen den Teilnehmern einer Choreographie modelliert werden können. Diese Muster unterscheiden zwischen der Anzahl der in einer Interaktion involvierten Teilnehmer, der Menge an Nachrichten, die diese austauschen, ob die Teilnehmer direkt oder indirekt miteinander kommunizieren etc. Die Prozesskonsolidierungsmethode wird anhand dieser Muster validiert. Dazu werden auf Basis des Metamodells exemplarisch Choreographien erstellt, die die Muster implementieren. Es wird dann untersucht, ob sich diese Choreographien und damit die Muster konsolidieren lassen.

1.2.4. Wiederherstellung fragmentierter Prozessmodelle

Es war erwartet worden, dass die Prozesskonsolidierung die Umkehroperation zur Fragmentierung von Prozessmodellen ist, die von Khalaf und Leymann in [KL06] beschrieben wurde. Dieser Beitrag diskutiert, inwieweit dies zutrifft und ob sich fragmentierte Prozessmodelle durch die Konsolidierung in das Ursprungsprozessmodell zusammenfügen lassen. Dazu wird untersucht, ob das aus den Fragmenten erstellte Prozessmodell verhaltensäquivalent zum Ursprungsprozessmodell ist und welche „Narben", d.h. strukturelle Unterschiede im Kontrollfluss des wiederhergestellten Ursprungsprozessmodell bestehen bleiben.

1.2.5. Prototypische Implementierung der Konsolidierung

Die Konsolidierungsmethode wird praktisch mit der Java Applikation *Cuvée* umgesetzt, die im Rahmen dieser Dissertation implementiert wurde. Cuvée liest eine BPEL4Chor-Choreographie in Dateiform ein und generiert daraus das konsolidierte BPEL-Prozessmodell. Cuvée erstellt neben der Datei, die die BPEL Beschreibung des konsolidierten Prozessmodells enthält, alle Artefakte, die notwendig sind, um das Prozessmodell auf einem WfMS auszuführen.

1.3. Aufbau der Arbeit

Nachdem in diesem Kapitel eine Übersicht über die Prozesskonsolidierung gegeben wurde, werden in Kapitel 2 wissenschaftliche Arbeiten vorgestellt, die thematisch mit den Forschungsbeiträgen dieser Dissertation verwandt sind.

In Kapitel 3 wird das Metamodell eingeführt, anhand dessen die Konsolidierungsmethode formal beschrieben wird.

Kapitel 4 legt die Grundlage für die Verifikation der Konsolidierungsschritte

über die Äquivalenz von Aktivitätszuständen. Dazu werden Zustandsbeziehungen zwischen Aktivitäten definiert, die gelten, wenn diese eine direkte Kontrollflussbeziehung zueinander haben. Die möglichen Zustandsbeziehungen werden dabei durch die operationale Semantik des jeweiligen Kontrollflusskonstrukts impliziert, das die Aktivitäten zueinander in Beziehung setzt. So haben zum Beispiel die Aktivitäten „Flugbuchung empfangen“ und „Sitzplatz reservieren“ in Abbildung 1.1 über die Kontrollflusskante eine direkte Beziehung. Aus den direkten Beziehungen können dann transitiv auch Zustandsbeziehungen zwischen Aktivitäten abgeleitet werden, die nur eine indirekte Kontrollflussbeziehung haben, wie zum Beispiel „Flugbuchung empfangen“ und „Zahlungsstatus empfangen“. Die Gesamtheit der Beziehungen zwischen den Aktivitäten einer Choreographie beschreibt ihr Verhalten. Basierend auf deren Verhalten definiert das Kapitel ein Äquivalenzkriterium für Choreographien.

Die oben erwähnten Prozesskonsolidierungsschritte werden dann in den folgenden zwei Kapiteln erläutert. In Kapitel 5 wird der erste Schritt, die Erstellung des konsolidierten Prozessmodells und der zweite Schritt, die Materialisierung von Basisinteraktionen, beschrieben. Auf den dritten Schritt, der Materialisierung von komplexen Schleifen-basierten Interaktionen, wird in Kapitel 6 eingegangen.

In Kapitel 7 wird die Konsolidierung anhand der Interaktionsmuster von Barros, Dumas und ter Hofstede [BDtH05] validiert.

Kapitel 8 diskutiert, ob sich die mit dem Ansatz von Khalaf und Leymann [KL06] fragmentierten Prozessmodelle konsolidieren lassen und ob das fragmentierte Ursprungsmodell verhaltensäquivalent zum konsolidierten Prozessmodell ist.

Die prototypische Implementierung der Prozesskonsolidierung mit dem Tool Cuvée wird in Kapitel 9 vorgestellt.

Die Ausarbeitung schließt in Kapitel 10 mit einer Zusammenfassung und der kritischen Auseinandersetzung mit den Ergebnissen der Forschungsbeiträge.

In diesem Zuge wird eine Übersicht über weitere mögliche Arbeiten im Kontext der Beiträge gegeben.

1.4. Veröffentlichungen

Die Publikationen, die während der Promotion in Journalen, Konferenzen und Workshops als Erstautor veröffentlicht wurden, sind in der unten stehenden Liste zusammengefasst. Vor deren Veröffentlichung wurden alle Publikationen von mehreren externen Gutachtern auf fachliche Korrektheit geprüft.

1. S. Wagner et al. „Fostering the Reuse of TOSCA-based Applications by Merging BPEL Management Plans“. In: *Cloud Computing and Services Science: 6th International Conference (CLOSER 2016) - Revised Selected Papers*. Bd. 740. Communications in Computer and Information Science. Springer International Publishing, Juli 2017, S. 232–254

2. S. Wagner, U. Breitenbücher und F. Leymann. „A Method For Reusing TOSCA-based Applications and Management Plans“. In: *Proceedings of the 6th International Conference on Cloud Computing and Service Science (CLOSER 2016)*. Rome: SciTePress, Apr. 2016, S. 181–191

3. S. Wagner, O. Kopp und F. Leymann. „Choreography-based Consolidation of Interacting Processes Having Activity-based Loops“. In: *Proceedings of the 5th International Conference on Cloud Computing and Service Science (CLOSER 2015)*. Stuttgart: SciTePress, Mai 2015, S. 284–296

4. S. Wagner, O. Kopp und F. Leymann. „Choreography-based Consolidation of Multi-Instance BPEL Processes“. In: *Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER 2014)*. Hrsg. von SciTePress. Barcelona: SciTePress, Apr. 2014, S. 287–298

5. S. Wagner et al. „Performance Optimizations for Interacting Business Processes“. In: *Proceedings of the first IEEE International Conference on Cloud Engineering (IC2E 2013)*. San Francisco: IEEE Computer Society, März 2013, S. 210–216

6. S. Wagner, O. Kopp und F. Leymann. „Consolidation of Interacting BPEL Process Models with Fault Handlers“. In: *Proceedings of the 5th Central-European Workshop on Services and their Composition (ZEUS 2013)*. Rostock: CEUR Workshop Proceedings, Feb. 2013, S. 9–16

7. S. Wagner et al. „State Propagation-based Monitoring of Business Transactions“. In: *Proceedings of the 2012 IEEE International Conference on Service-Oriented Computing and Applications (SOCA 2012)*. Taipeh: IEEE Xplore, Dezember 2012

8. S. Wagner, O. Kopp und F. Leymann. „Towards Verification of Process Merge Patterns with Allen's Interval Algebra“. In: *Proceedings of the 4th Central-European Workshop on Services and their Composition (ZEUS 2012)*. Bamberg: CEUR Workshop Proceedings, März 2012, S. 81–88

9. S. Wagner, O. Kopp und F. Leymann. „Towards Choreography-based Process Distribution In The Cloud“. In: *Proceedings of the 2011 IEEE International Conference on Cloud Computing and Intelligence Systems*. Bejing: IEEE Xplore, Sep. 2011, S. 490–494

Kapitel 2

Stand der Forschung

Dieses Kapitel gibt einen Überblick über die wissenschaftlichen Arbeiten, die zu den in dieser Ausarbeitung adressierten Themenbereichen verwandt sind. Verschiedene Ansätze zur Modellierung von Choreographien werden in Abschnitt 2.1 vorgestellt und verglichen. Die Konsolidierungsschritte und die Verifikation der Verhaltensäquivalenz von konsolidierten Prozessmodellen zu den Choreographien, aus denen sie erstellt wurden, werden anhand von formalen Modellen beschrieben. Existierende Formalisierungsansätze werden dazu in Abschnitt 2.2 untersucht. Abschnitt 2.3 gibt eine Übersicht von existierenden Arbeiten, die sich ebenfalls mit der Konsolidierung von Prozessmodellen befassen und grenzt diese von dem hier beschriebenen Konsolidierungsansatz ab.

2.1. Choreographien

Orchestrierungen sind Workflows, also ausführbare Geschäftsprozesse, die die organisationsinternen Arbeitsabläufe eines Dienstleisters modellieren,

indem sie die Ausführungsreihenfolge von Aktivitäten festlegen, die notwendig sind, um eine bestimmte Dienstleistung zu erbringen [DD04]. Die Aktivitäten werden durch Webservices implementiert. Technisch ausgedrückt, aggregieren Orchestrierungen also Webservices [KL03; LON+13], indem sie ihre Ausführungsreihenfolge sowie den Datenfluss zwischen ihnen mittels Prozessmodellen spezifizieren. Die Ausführung eines Prozessmodells wird wiederum von einer *Wokflow-Engine* koordiniert. Die prominentesten Vertreter von ausführbaren Orchestrierungs- bzw. Prozessmodellierungssprachen sind die Business Process Execution Language (BPEL) [OAS07b] und die Business Process Model and Notation (BPMN) [Obj11].

Choreographien hingegen beschreiben die Reihenfolge des öffentlich sichtbaren Nachrichtenaustausches zwischen zwei oder mehreren Dienstleistern bzw. Teilnehmern, die kollaborieren, um ein Geschäftsziel zu erreichen [Pel03]. Im Gegensatz zu Prozessen, werden Choreographien nicht über einen zentralen Koordinator gesteuert.

In der Literatur wird zwischen zwei Modellierungskonzepten für Choreographien unterschieden – *Verbindungsmodelle* und *Interaktionsmodelle* [DKB08].

In einem Verbindungsmodell wird das lokale Verhalten jedes Teilnehmers über ein Prozessmodell beschrieben. Die Prozessmodelle stellen dabei meist nur das öffentlich sichtbare Verhalten der Teilnehmer dar. Der Nachrichtenaustausch wird mittels Nachrichtenkanten modelliert, die die Kommunikationsaktivitäten der Prozessmodelle miteinander verbinden. Ein Beispiel für ein Verbindungsmodell wurde bereits in Abbildung 1.1 dargestellt. Verbindungsmodelle können zum Beispiel mit BPMN-Kollaborationsdiagrammen [Obj11], *BPEL4Chor* [DKLW09] oder WSFL [Ley01] erstellt werden.

Bei Interaktionsmodellen wird der Nachrichtenaustausch zwischen den Teilnehmern einer Choreographie über eine Menge von atomaren Interaktionen modelliert. Die Reihenfolge zwischen den Interaktionen kann über Kontrollflusskonstrukte festgelegt werden. Im Gegensatz zu Verbindungsmodellen abstrahieren Interaktionsmodelle vom lokalen Verhalten der Teilnehmer. Eine Interaktion spezifiziert also nur, dass zwei Teilnehmer Nachrichten aus-

tauschen, aber nicht welche Kommunikationsaktivitäten in dem Austausch involviert sind. Interaktionsmodelle können zum Beispiel mit BPMN Choreographiediagrammen [Obj11], *Let's Dance* [ZBDtH06] oder für BPEL-Prozesse mit *BPELgold* [KEvL+10] erstellt werden.

Die Eigenschaften der beiden Modellierungsansätze wurden in verschiedenen Arbeiten verglichen [DW11; DKLW09; KLW11]. Bei der Prozesskonsolidierung sollen die interagierenden Prozessmodelle der Teilnehmer in ein einzelnes Prozessmodell zusammengefasst werden. Folglich kann die Konsolidierung nur auf Choreographien angewandt werden, die als Verbindungsmodelle modelliert sind und die somit ihr Teilnehmerverhalten spezifizieren. Decker et al. [DKLW09] sowie Kopp, Leymann und Wagner [KLW11] haben BPEL4Chor evaluiert und mit anderen Choreographiesprachen verglichen. BPEL4Chor ist die mächtigste Sprache, um komplexes Interaktionsverhalten über Verbindungsmodelle zu modellieren. Mit BPEL4Chor können, wie Kopp [Kop16a] gezeigt hat, bis auf eine Ausnahme alle Interaktionsmuster zwischen Teilnehmern modelliert werden, die von Barros, Dumas und ter Hofstede [BDtH05] formuliert wurden. Darüber hinaus nutzt BPEL4Chor BPEL als Modellierungssprache zur Beschreibung des Teilnehmerverhaltens. BPEL hat im Gegensatz zu BPMN eine klar spezifizierte operationale Semantik [Ley10]. Aus diesen Gründen wird BPEL4Chor Grundlage für das in dieser Ausarbeitung genutzte Metamodell für die Beschreibung der Konsolidierungsoperation genutzt.

2.2. Formalisierung von BPEL-Prozessmodellen

Für BPEL-Prozesse und deren Kommunikationsverhalten existieren eine Vielzahl von Formalisierungen, mit denen neben anderen Eigenschaften auch die Verhaltensäquivalenz von Prozessen zueinander verifiziert werden können [HDvdA+05]. Eine Literaturrecherche zu existierenden BPEL-Formalisierungen wurde von Breugel und Koshkina erstellt [vBK06]. Dort kategorisieren die Autoren in Formalisierungen mit Petri-Netzen, Prozessalge-

bren und Abstract State Machines. An dieser Kategorisierung orientiert sich auch dieser Abschnitt. Eine aktuellere Übersicht über BPEL Formalisierungen findet sich in der Arbeit von Boumlik, Mejri und Boucheneb [BMB20].

Die Formalisierungen werden darauf hin untersucht, ob sie die Semantik der Kontroll-, Daten- und Nachrichtenflusskonstrukte abbilden können, die notwendig sind, die Interaktionsmuster aus [BDtH05] mit BPEL4Chor zu modellieren und in einen BPEL-Prozess zu konsolidieren. Dies sind neben den Kommunikationsaktivitäten Invoke und Receive auch Schleifen und Pick-Aktivitäten, mit denen multilaterale Interaktionen umgesetzt werden. Das Kommunikationsverhalten von einigen Interaktionsmustern wird über kontrolliert ausgelöste Fehler modelliert. Daneben können während einer Interaktion auch unerwartete Fehler auftreten. Das formale Modell muss daher die in BPEL spezifizierte Fehlersemantik (mit Ausnahme von Kompensation) und die zugehörigen Aktivitätszustände [KHK+11] formalisieren können, die eine nicht erfolgreiche Ausführung von Aktivitäten kennzeichnen. Die durch die nachrichtenbasierte Kommunikation implizierten Kontrollflussabhängigkeiten zwischen den Aktivitäten unterschiedlicher Teilnehmer werden im konsolidierten Prozessmodell mit Kontrollflusskanten und Eintrittsbedingungen an den jeweiligen Aktivitäten emuliert (siehe Abschnitt 5.4). Daher muss die Kontrollflusskantensemantik inklusive Dead-Path-Elimination ebenfalls im Modell abgebildet werden können. Ein Formalisierung der Assign-Aktivitäten ist notwendig, um den Datenfluss im konsolidierten Prozessmodell zu emulieren.

Zusammenfassend ist festzustellen, dass nur ein kleiner Teil der Formalisierungen die geforderten Konstrukte abdecken. Aus diesem Grund wird eine eigene Formalisierung der Syntax von BPEL und BPEL4Chor in Kapitel 3 sowie deren Semantik in Kapitel 4 eingeführt.

2.2.1. Petri-Netze

Es existieren unterschiedliche Arbeiten, die die Transformation von BPEL-Prozessmodellen in Petri-Netze [Pet62] zum Gegenstand haben [Sta04; HSS05; YTYL05; OVvdA+05; OVvdA+07; Loh07a; YZ14; KI14]. Die meisten dieser Arbeiten betrachten die Transformation von Teilmengen von Kontrollflusskonstrukten aus der BPEL-Spezifikation. Eine vollständige Transformation von allen BPEL 2.0 Kontrollflusskonstrukten in *Open Workflow Nets (oWFNs)* [MRS05] wird von Lohmann beschrieben [Loh07a; Loh07b]. oWFNs sind eine Klasse von Petri-Netzen, die eine Menge von Schnittstellen, d.h. Eingabe- und Ausgabestellen besitzen sowie eine definierte Menge von Anfangs- und von Endmarkierungen. Die Formalisierung wird von Lohmann et al. [LKLR07] zur Transformation von BPEL4Chor-Choreographien in oWFNs erweitert. Das aus der Choreographie resultierende oWFN kann auf verschiedene Eigenschaften, wie zum Beispiel auf das Vorhandensein von Deadlocks oder Livelocks, geprüft werden.

Die aus der Transformation resultierenden oWFNs sind aufgrund der komplexen Kontrollflusssemantik von BPEL, wie zum Beispiel Dead-Path-Elimination und Fehlerbehandlung, die zu einer kombinatorischen Explosion von Stellen und Transitionen führen kann [Loh07a; LKLR07; OVvdA+07], sehr umfangreich und sind damit nur noch mit Toolunterstützung analysierbar. Die Komplexität der Petri-Netze erschwert daher eine anschauliche Diskussion der Verhaltensäquivalenz der Konsolidierungsschritte, die in dieser Arbeit erläutert werden.

Des Weiteren abstrahieren die oben genannten Petri-Netz-Transformationen vom Datenfluss der Prozessmodelle. Hat zum Beispiel eine Aktivität zwei ausgehende Kontrollflusskanten, von denen abhängig vom Wert einer Variable im BPEL-Prozessmodell genau eine aktiviert wird (dies wird über deren Transitionsbedingungen modelliert), kann dieses Verhalten im Petri-Netz nicht abgebildet werden. Die Entscheidung, welche Kante aktiviert wird, wird im Petri-Netz nicht-deterministisch getroffen [MKL+09; OVvdA+07]. Bei der Konsolidierung muss aber sichergestellt werden, dass die durch

den Kontroll- und Nachrichtenfluss modellierten Beziehungen zwischen den Aktivitäten, wie zum Beispiel die Parallelität oder Exklusivität zwischen Aktivitätsausführungen, erhalten bleiben.

Grundsätzlich hätten die Petri-Netz-Formalisierung von Lohmann et al. genutzt werden können, um die Konsolidierungsschritte zu verifizieren. Allerdings hätten diese Transformationen angepasst werden müssen, um die Erhaltung der Beziehungen wie Exklusivität oder Parallelität zwischen den Aktivitäten bei der Konsolidierung zu verifizieren. Die resultierenden Petri-Netze wären, wie oben erwähnt, sehr umfangreich geworden und hätten die Diskussion der Verifikation erschwert. Aus diesen Gründen wurde darauf verzichtet, diese Formalisierung zur Verifikation des Verhaltens zu verwenden.

2.2.2. Prozessalgebren

Foster et al. [FUMK03] beschreiben eine Transformation von BPEL-Prozessmodellen in *Finite State Process (FSP)* [MK06]. In einer weiteren Arbeit [FUMK05] erweitern sie den Ansatz auf Choreographien. Bei FSPs handelt es sich um eine Prozessalgebra, die eine kompaktere Formalisierung von endlichen Transitionensystemen ermöglicht. Durch die Umwandlung der BPEL-Prozesse in FSPs bzw. in Transitionensysteme können diese unter anderem auf Äquivalenz überprüft werden [HDvdA+05]. Wie die Petri-Netz-Formalisierungen abstrahieren auch die FSPs vom Datenfluss. Die resultierenden FSPs formalisieren keine Aktivitätszustände. Es kann also nicht unterschieden werden, ob eine Aktivität erfolgreich oder in einem Fehlerzustand beendet wurde. Damit kann mit FSPs nicht verifiziert werden, ob zum Beispiel zwei Prozessmodelle das gleiche Fehlerverhalten abbilden.

Eine π-Kalkül Formalisierung von BPEL 1.1 wurde von Fadlisyah [Fad04] beschrieben. Die Formalisierung bildet Aktivitätszustände ab. Schleifen, Kontrollflusskanten und Event-Handler wurden nicht formalisiert. Diese sind aber für die Modellierung von multilateralen Interaktionen unabdingbar.

Ein Formalisierungsansatz von BPEL 2.0 mit dem π-Kalkül wird von Lucchi und Mazzara [LM07] beschrieben. Die Semantik von Kontrollflusskanten, wie zum Beispiel Eintrittsbedingungen von Aktivitäten (siehe Abschnitt 3.2.3.1) und Dead-Path-Elimination, wurde ausgeklammert. Die Formalisierung unterstützt zwar Event-Handler allerdings keine Schleifen. Daher ist es auch mit dieser Formalisierung nicht möglich, multilaterale Interaktionen zu modellieren.

Abouzaid und Mullins [AM08] definieren basierend auf der Arbeit von Lucchi und Mazzara ein π-Kalkül namens *BP-Kalkül*, das zur Formalisierung von Workflows im generellen dient. Sie übersetzen die Syntax und Semantik von BPEL 2.0 in dieses Kalkül. Allerdings fehlt auch in dieser Arbeit die Formalisierung von Schleifen.

Die Formalisierung von Weidlich, Decker und Weske [WDW07] ermöglicht eine sehr kompakte π-Kalkül Darstellung von BPEL 2.0 Prozessmodellen. Dafür wird auf die Unterscheidung von Aktivitätszuständen verzichtet. Es wird nur die Standardeintrittsbedingung von BPEL-Aktivitäten unterstützt, d.h. eine eingehende Kontrollflusskante muss aktiviert sein, um eine Aktivität auszuführen. Für die Umsetzung der Konsolidierung werden aber auch komplexere Eintrittsbedienungen benötigt.

Koshkina und Breugel [KvB04] präsentieren eine Prozessalgebra namens *BPE-Kalkül*, die BPEL 1.1 Prozessmodelle in Axiome und Regeln abstrahiert. Die Formalisierung fokussiert sich auf die Semantik von Kontrollflusskanten, Eintrittsbedingungen von Aktivitäten und Dead-Path-Elimination im regulären Kontrollfluss. Mögliche Fehlerszenarien, die bei der Ausführung von Aktivitäten auftreten können, wurden in der Arbeit jedoch ausgeklammert.

Mit der Prozessalgebra *Language Of Temporal Ordering Specification (LOTOS)* [LJ98] können zum einen Daten und Operationen und zum anderen Prozesse formalisiert werden. Die Formalisierung von Prozessen erfolgt dabei ähnlich wie beim *Calculus of Communicating Systems (CCS)* von Milner [Mil80]. Ferrara [Fer04] beschreibt, wie ein BPEL 1.1 Prozessmodell in

LOTOS transformiert werden kann. Die aus dem Prozessmodel erstellte LOTOS Formalisierung beinhaltet neben dem Kontrollfluss auch dessen Kommunikations- (Webservice-Operationen, Partner Links) und Datenflussaspekte, wie Variablen und Ausdrücke, die zum Beispiel die Eintrittsbedingungen von Aktivitäten festlegen. Die Fehlerbhandlungskonzepte von BPEL inklusive Terminierung und Kompensation werden ebenfalls formalisiert. Die LOTOS-Formalisierung unterstützt allerdings keine zeitbasierten Ereignisse, um Aktivitäten, zum Beispiel in einem Event-Handler, nach dem Ablauf einer bestimmten Zeitdauer anzustoßen. Außerdem muss für jede Aktivität zur Entwurfszeit bereits bekannt sein, mit welchem Partner sie kommuniziert. Durch diese beiden Einschränkungen werden die modellierbaren Interaktionsmuster stark limitiert (siehe Kapitel 7). Da neben dem Kontrollfluss auch alle Datenfluss- und Kommunikationsaspekte eines BPEL-Prozesses transformiert werden, wird die resultierende LOTOS-Formalisierung sehr komplex [WDW07; SFC04].

Pu et al. [PZWQ06; PZQ+06] präsentieren eine auf Axiomen und Regeln basierende Prozessalgebra, die sich auf die Formalisierung des Fehlerverhaltens von BPEL-Prozessen fokussiert. Interaktionen von Prozessmodellen werden daher nicht betrachtet.

Die von Butler und Ferreira entwickelte Prozessalgebra *Structured Activity Compensation (StAC)* [BF04] dient zur Formalisierung von langlaufenden Geschäftstransaktionen und deren Kompensationsverhalten. StAC basiert auf dem CCS von Milner und den *Communicating Sequential Processes (CSP)* von Hoare [Hoa78]. Um die Ausdrucksfähigkeit der Algebra zu demonstrieren, beschreiben Butler, Ferreira und Ng [BFN05] eine BPEL 1.1 Formalisierung mit StAC. Die Formalisierung unterstützt weder Transitionsbedingungen bei Kontrollflusskanten noch Eintrittsbedingungen von Aktivitäten. Folglich können komplexere Verzweigungen im Kontrollfluss nicht ausgedrückt werden. Da der Fokus der Arbeit auf der Kompensation liegt, wurden die Interaktionen von Prozessen nicht formalisiert.

Virvoli [Vir04] beschreibt die Formalisierung einer Teilmenge der Syntax und

der operationalen Semantik von BPEL 1.1 Prozessen mittels Transitionensystemen. Aspekte wie Aktivitätszustände, Fehlerbehandlung und zeitbasierte Ereignisse wurden nicht formalisiert. Daher ist es nicht möglich, mit der Formalisierung komplexe Interaktionsmuster zu modellieren.

Boumlik, Mejri und Boucheneb [BMB20] analysieren Spezialfälle im Verhalten von Fault-, Termination-, Compensation- und Event-Handler. Dazu übersetzen sie die XML-Syntax von BPEL in eine von ihnen entwickelte formale Repräsentation namens *Abbreviated Version of BPEL (AV-BPEL)*. Die operationale Semantik der BPEL-Konstrukte wird mittels Axiomen und Transitionsregeln über die AV-BPEL-Elemente definiert. Die Autoren beschreiben mit den Regeln die Semantik eines Großteils der Kontrollflusskonstrukte von BPEL. Kontrollflusskanten und deren Semantik wurden allerdings nicht formalisiert.

Zhu, Huang und Zhou [ZHZ17] erläutern die Generierung von *Communicating Sequential Processes (CSPs)* [Hoa85] aus BPEL 2.0 Prozessen. Die Generierung deckt nur eine Teilmenge der Kontrollflusskonstrukte ab. So können zum Beispiel Handler nicht in CSPs transformiert werden [BMB20].

Duan et al. [DBLL04] präsentieren ein Modell für abstrakte BPEL 1.1 Prozesse, mit dem sich die Semantik von Aktivitäten mittels schwächsten Vor- und stärksten Nachbedingungen beschreiben lässt [Hoa69]. Das Modell dient zum einen zur Verifikation von abstrakten Prozessen und zum anderen können diese aus dem semantischen Modell synthetisiert werden. Das Modell unterstützt nur eine Teilmenge der BPEL-Aktivitäten, so wurden zum Beispiel Scope-Aktivitäten nebst Handler oder Pick-Aktivitäten nicht formalisiert. Es kann nur der positive Kontrollfluss eines Prozesses bzw. seiner Aktivitäten formalisiert werden, aber keine Fehlerzustände.

2.2.3. Abstract State Machines

Abstract State Machines (ASM) werden in unterschiedlichen Bereichen der Informatik zur Formalisierung und Verifikation von Algorithmen verwendet

[Gur18; BS03]. Vereinfacht gesagt, besteht eine ASM aus einer Menge von abstrakten Datenobjekten, Zuständen und Transitionsregeln. Dabei besteht jede Regel aus einer booleschen Funktion, die zu „wahr“ evaluieren muss, um die ebenfalls in der Regel definierten Funktionen, die die Zustandsänderungen beschreiben, auszuführen. Eine erste Formalisierung von BPEL 1.1 Prozessen mittels ASMs wurde von Farahbod, Glässer und Vajihollahi [FGV05] erstellt. Diese Formalisierung wurde von Fahland und Reisig erweitert, um auch die Fehlerbehandlungs- und Kompensationssemantik von BPEL-Prozessen mit ASMs zu formalisieren [FR05; Fah05]. Die aus den Prozessen generierten ASMs bilden den kompletten Kontroll- und Datenfluss von BPEL ab und sind somit sehr umfangreich. Die Formalisierung betrachtet allerdings die booleschen Ausdrücke, die in Transitions- und Eintrittsbedingungen verwendet werden, als abstrakte Zeichenketten, interpretiert also nicht deren Syntax und Semantik [Fah05]. Die erzeugten ASMs bilden auch die in den Prozessen modellierten Elemente zur Kommunikation mit Webservices ab. Da in der BPEL-Spezifikation nur die Schnittstellen von Webservices relevant sind, nicht aber deren Verhalten, formalisieren die ASMs ebenfalls nur die Schnittstellenkommunikation. Um Choreographien mit interagierenden BPEL-Prozessen als ASMs abzubilden, müsste die Formalisierung entsprechend erweitert werden.

2.2.4. SPIN & Promela

SPIN [Hol97] ist ein Tool für die Verifikation von Modellen von verteilten Systemen. Mit dem Tool können somit auch Prozessmodelle auf verschiedene Eigenschaften geprüft werden. Als Eingabe erhält SPIN ein Prozess, der mittels der formalen Sprache *Process Meta Language Promela (Promela)* [Hol93] beschrieben ist. Die zu verifizierenden Eigenschaften sind in *Linearer Temporaler Logik (LTL)* [Pnu77] formuliert. Um die Eigenschaften von BPEL-Prozessen zu prüfen, müssen sie also in Promela transformiert werden.

Fu, Bultan und Su [FBS04a; FBS04b] präsentieren eine Methode, mit der sich die Kommunikation von mit BPEL modellierten Webservices auf un-

terschiedliche Eigenschaften, wie zum Beispiel auf die Kompatibilität zu anderen Services, prüfen lassen. Dazu wird eine Teilmenge von BPEL 1.1 Prozessmodellen in einen *bewachten Zustandsautomaten* (engl. Guarded Automaton) transformiert, der wiederum nach Promela übersetzt wird. Die Autoren beschreiben nur für eine Teilmenge von Kontrollflusselementen, wie diese in Zustandsautomaten übersetzt werden können. Die Übersetzung von Schleifen, Kontrollflusskanten- oder Fehlerbehandlungssemantik wird nicht oder nur teilweise adressiert.

Nakajima [Nak05] beschreibt einen ähnlichen Ansatz, bei dem ein BPEL 1.1 Prozess erst in einen *erweiterten endlichen Zustandsautomaten* und von dort in Promela übersetzt wird. Die Transformation von Nakajima unterstützt dabei mehr Kontrollflusselemente als die von Fu, Bultan und Su. So wird in dem generierten Automaten, und somit auch in Promela, Dead-Path-Elimination abgebildet. Allerdings werden nicht alle BPEL-Konstrukte übersetzt, die nötig sind, um komplexe Interaktionen zu modellieren, wie zum Beispiel Pick-Aktivitäten.

2.2.5. Automaten

Eine Transformation von BPEL 1.1 Prozessen in endliche deterministische Zustandsautomaten wird von Wombacher, Fankhauser und Neuhold beschrieben [WFN04]. Mit den Automaten soll das Kommunikationsverhalten eines mit BPEL implementierten Webservice formalisiert werden, um bei einer Service-Discovery kompatible Webservices zu ermitteln. Die Transformation fokussiert sich daher nur auf BPEL-Kontrollflusselemente, die zur Kommunikation mit externen Webservices nötig sind. Aspekte wie Fehlerbehandlung oder die Semantik von Kontrollflusskanten werden durch die Zustandsautomaten nicht formalisiert.

Arias-Fisteus, Fernández und Kloos [AFK04; AFK05] stellen das *VERification for BUSiness processes (VERBUS)*[1] Framework vor, mit dem Prozessmodelle,

[1] http://www.it.uc3m.es/jaf/verbus/

die in unterschiedlichen Sprachen modelliert sind, in endliche Zustandsautomaten transformiert werden können. Aus diesen Automaten können dann verschiedene Formalisierungen, wie zum Beispiel Promela, erstellt werden. Die Autoren geben an, dass sie, abgesehen von BPELs Kompensationsmechanismus, alle anderen Sprachelemente und deren Semantik mittels Zustandsautomaten abbilden können. Die Arbeiten enthalten aber keine detaillierte Beschreibung darüber, wie die einzelnen Elemente eines BPEL-Prozesses in einen Zustandsautomaten transformiert werden können.

Event-B [Abr10] ist eine mathematische Methode zur Modellierung und Analyse von Software Systemen. Aït-Sadoune und Aït-Ameur [AA12] beschreiben wie BPEL-Prozesse mit Event-B modelliert werden können. Babin, Aït-Ameur und Pantel [BAP17] diskutieren, wie sich das Kompensationsverhalten von BPEL-Prozessen mit Event-B Modellen abbilden lässt. Beide Arbeiten verzichten allerdings auf die Formalisierung der Kontrollflusskanten mit Event-B [BMB20].

Stachtiari und Katsaros [SK18] verfolgen einen Formalisierungsansatz, bei dem im formalen Modell eines BPEL-Prozesses dessen originale Blockstruktur erhalten bleibt. Damit sollen Analyse- bzw. Verifikationsergebnisse vom Modellierer direkt auf die originale BPEL-Prozessbeschreibung übertragen werden können. Als formales Modell nutzen die Autoren das *Behavior-Interaction-Priority (BIP)* Framework [BBB+11], mit dem sich das Verhalten von Komponenten, wie zum Beispiel die Aktivitäten eines Prozesses, als Transitionensysteme mit Schnittstellen modellieren lässt. Verschiedene Koordinatorschichten spezifizieren die Operationen zwischen den Komponenten. Die Autoren bilden einen Großteil der operationalen Semantik von BPEL mit dem BIP Framework ab. Es wird aber auch hier von Daten und Bedingungen, wie zum Beispiel Transitionsbedingungen, abstrahiert. Des Weiteren erschwert dieser blockstrukturerhaltende Ansatz die Verifikation der Konsolidierung, da diese bestimmte strukturierte Aktivitäten entfernt und deren Verhalten durch andere Aktivitäten emuliert.

2.2.6. Aktivitätszustandsbeziehungen

In Kapitel 4 wird das Verhalten von Choreographien und Prozessmodellen über *Aktivitätszustandsbeziehungen* beschrieben. Eine Aktivitätszustandsbeziehung spezifiziert die erlaubte temporale Reihenfolge zwischen zwei Zustandstransitionen von einer oder zwei Aktivitäten, die deren Instanzen zur Laufzeit einhalten müssen (siehe Beispiel in Abschnitt 1.2.2). Die Beziehungen werden dabei durch den expliziten oder transitiven Kontroll- und Nachrichtenfluss impliziert, der zwischen den Aktivitäten gilt. Eine Zustandsbeziehung kann zum Beispiel aufgrund des Kontrollflusses zwischen zwei Aktivitäten a_i und a_j festlegen, dass eine Instanz von Aktivität a_i nie ausgeführt werden kann, wenn eine Instanz von a_j einen Fehlerzustand erreicht hat und dass die Instanz von a_i immer nach der Instanz von a_j einen Fehlerzustand erreichen muss. Die Gesamtheit der Beziehungen zwischen allen Aktivitäten und ihren Zuständen kennzeichnet dabei das Verhalten der Choreographie.

Die Aktivitätszustandsbeziehungen sind von der *verzweigten Punktalgebra* [Rei94; DWM99] inspiriert, mit der qualitative zeitliche Ordnungsrelationen zwischen abstrakten Zeitpunkten definiert werden können.

Weidlich et al. verfolgen einen Ansatz, der die Verhaltenskonsistenz von verschiedenen Petri-Netzen mittels *Verhaltensprofilen* überprüft [WMW11; WPMW11; Wei11]. Ein Verhaltensprofil abstrahiert, ähnlich wie die Aktivitätszustandsbeziehungen, von den möglichen Ausführungsreihenfolgen (engl. Traces) der Transitionen eines Petri-Netzes, indem es eine Ordnungsrelationen zwischen allen Paaren von Transitionen des Netzes festlegt. Die Verhaltensprofile nutzen dabei dieselben Ordnungsrelationen bzw. Beziehungen wie die Aktivitätszustandsprofile. In den oben genannten Arbeiten beschreiben die Autoren, wie sich die Verhaltensprofile von Petri-Netzen algorithmisch bestimmen lassen. Die algorithmische Bestimmung von Aktivitätszustandsprofilen aus Choreographien ist nicht Gegenstand dieser Ausarbeitung, da diese bzw. deren Fragmente, an denen die Konsolidierungsschritte erläutert und verifiziert werden, wenige Aktivitäten enthalten.

Van der Aalst [vdAal11] definiert eine *Footprint Matrix*, die das Verhalten von Prozessen ebenfalls mittels Beziehungen von Aktivitäten spezifiziert. Die Matrix wird im Gegensatz zu den Aktivitätszustandsbeziehungen nicht auf Basis der Kontrollflusses ermittelt, sondern beim Process Mining durch die Analyse von Event-Logs, die die Ausführungshistorie von verschiedenen Instanzen eines Prozessmodells enthalten.

2.3. Konsolidierung von Prozessmodellen

Ein algebraischer Operator zur Konsolidierung von Modellen wird von Brunet et al. [BCE+06] sowie von Pottinger und Bernstein [PB03] unabhängig von einem konkreten Metamodell beschrieben. Die Operation enthält als Eingabe die zu konsolidierenden Modelle, die Beziehungen zwischen ihnen und erzeugt als Ausgabe das konsolidierte Modell. Dies entspricht der in dieser Arbeit vorgestellten Konsolidierungsoperation, die als Eingabe eine Choreographie erhält und als Ausgabe das konsolidierte Prozessmodell generiert. Die Choreographie enthält dabei die zu konsolidierenden Modelle und die Beziehungen zwischen ihnen in Form von Interaktionen bzw. Nachrichtenkanten. Brunet at el. erläutern verschiedene algebraische Eigenschaften, auf die Konsolidierungsoperationen untersucht werden können, wie zum Beispiel Kommutativität, Assoziativität oder Inversität. Die Prüfung der Inversität, d.h. inwieweit sich mit der Konsolidierung die Fragmentierung von Prozessmodellen rückgängig machen lässt, ist ein zentraler Forschungsschwerpunkt dieser Arbeit.

Die Konsolidierung von BPMN Prozessmodellen, deren Interaktionsverhalten mittels BPMN Kollaborationsdiagrammen modelliert ist, werden von Kunchala et al. [KYYH17; KYYL20] beschrieben. Analog zum Ansatz in dieser Arbeit, werden im konsolidierten Prozessmodel die Kontrollflussbeziehungen zwischen den Aktivitäten, die aus unterschiedlichen Prozessmodellen stammen, anhand des im Kollaborationsdiagramm modellierten Interaktionsverhaltens bestimmt. Der Ansatz fokussiert sich auf die Konsolidierung von

Prozessmodellen, an denen zwei Teilnehmer bzw. Prozessmodelle beteiligt sind. Auf die Konsolidierung von komplexen Interaktionen zwischen mehr als zwei Teilnehmern [BDtH05] wird im Unterschied zu dieser Arbeit nicht eingegangen. Kunchala et al. diskutieren allerdings die Konsolidierung von synchronen Interaktionen, die in dieser Arbeit ausgeklammert werden, da sie ebenfalls durch asynchrone Interaktionen emuliert werden können. Eine formale Diskussion der Verhaltensäquivalenz des konsolidierten Prozessmodells verglichen mit den originalen Prozessmodellen findet in der Arbeit nicht statt.

Es existieren verschiedene Arbeiten, die, im Gegensatz zu dem in dieser Arbeit beschriebenen Ansatz zur Konsolidierung von kommunizierenden bzw. kollaborierenden Prozessmodellen, die Konsolidierung von semantisch ähnlichen Prozessmodellen beschreiben. Dazu müssen vorher mit verschiedenen Techniken die semantisch äquivalenten Elemente [KW17; BBGK17] oder Teilgraphen identifiziert werden [HAS21]. Eine umfassende Literaturrecherche in diesem Bereich wurde von Belchior et al. durchgeführt [BGVC20].

Küster et al. [KGFE08a] stellen ein Plugin für den IBM WebSphere Business Modeler[1] vor, das einen Prozessmodellierer dabei unterstützt, verschiedene Varianten desselben Prozessmodells manuell wieder in ein einzelnes Prozessmodel zu überführen. Das Plugin nutzt dabei Techniken, die ebenfalls von Küster et al. [KGFE08b] beschrieben wurden, um Unterschiede zwischen Prozessmodellen zu bestimmen. Weitere Ansätze zur Generierung eines Prozessmodells aus verschiedenen Prozessmodellvarianten werden zum Beispiel von Li et al. [LRW09] erläutert.

Sun et al. [SKY06] beschreiben einen Ansatz, komplementäre Workflow-Netze [vdAal98] in ein einzelnes Workflow Netz zu konsolidieren. Dazu müssen zuerst ein oder mehrere *Verknüpfungspunkte* identifiziert werden, die angeben, an welcher Stelle die Kontrollflussgraphen der Workflow-Netze (oder Teile davon) zusammengefügt werden sollen. Sie definieren dazu eine

[1] https://www.ibm.com/support/pages/download-websphere-business-modeler-advanced-v70

Menge von *Verknüpfungsmustern*. Jedes Muster beschreibt, wie die Kontrollflussgraphen der beiden Workflow-Netze an den Verknüpfungspunkten kombiniert werden können, zum Beispiel parallel, sequentiell oder ersetzend. Die Autoren fokussieren sich in der Arbeit darauf, Regeln zu formulieren, die beschreiben, wie Workflow-Netze korrekt konsolidiert werden können, ohne dass ein syntaktisch falsches Workflow-Netz entsteht oder Informationen aus den originalen Worklfow-Netzen verloren gehen. Im Gegensatz zu der vorliegenden Arbeit, wo die Verknüpfungspunkte automatisch durch den Nachrichtenfluss der Choreographie ermittelt werden, werden die Verknüpfungspunkte und die Verknüpfungsmuster bei dem Ansatz von Sun et al. manuell durch einen Modellierer ausgewählt.

Gottschalk et al. [GvdAJ08] präsentieren einen Algorithmus und ein Tool, mit dem zwei *Event-Driven Process Chains (EPCs)* [STA05] in eine einzelne EPC konsolidiert werden können, die verhaltensäquivalent zu diesen beiden EPCs ist. Bevor die EPCs konsolidiert werden können, muss der Modellierer angeben, welche Funktionen der beiden EPCs semantisch identisch sind. Die identischen Funktionen dienen als Verknüpfungspunkte und werden in eine einzelne Funktion zusammengefasst. Über sie wird der Kontrollfluss zwischen den Ereignissen und Funktionen aus den beiden EPCs im konsolidierten EPC so kombiniert, dass die anderen Funktionen die gleiche Ausführungsreihenfolge haben wie in den originalen EPCs. Dieser Ansatz entspricht der Prozesskonsolidierung dahingehend, dass dort im konsolidierten EPC analog zum konsolidierten Prozessmodell ebenfalls das originale Verhalten beibehalten werden soll. Die Auswahl der Verknüpfungspunkte erfolgt im Gegensatz zur Prozesskonsolidierung manuell.

Mendling und Simon [MS06] diskutieren einen zu dem von Gottschalk et al. ähnlichen Ansatz. Dort werden EPCs, die unterschiedliche Sichten von verschiedenen Akteuren auf dasselbe Prozessmodell repräsentieren, integriert. Es werden auch bei diesem Ansatz semantisch äquivalente Funktionen manuell identifiziert.

Rosa et al. [RDUD10] stellen einen Algorithmus vor, der als Eingabe zwei als

Prozessmodellgraphen[1] spezifizierte Prozessmodelle erhält und aus diesen ein *konfigurierbares Prozessmodell* [RvdA07] erstellt. Dieses Prozessmodell subsumiert deren Verhalten und enthält Informationen darüber, aus welchen Eingabemodellen die Kontrollflusselemente stammen und welche Fragmente semantisch äquivalent sind [RDUD13]. Dies dient zur Unterstützung von Modellierern oder Analysten, die das konfigurierbare Prozessmodell dann *individualisieren*, d.h. konkrete Prozessmodelle aus ihm erstellen können. Der Fokus liegt also auch dort nicht auf der automatischen Konsolidierung von Prozessmodellen. Ähnlich wie in dieser Arbeit wird das Verhalten der Prozessmodelle über die möglichen Ausführungsreihenfolgen von deren Aktivitäten beschrieben.

Eine weiteres Forschungsfeld im Bereich der Konsolidierung ist das Zusammenführen von wiederverwendbaren *Prozessmodellfragmenten* [SLM+10] zu einem Prozessmodell. Fragmente sind unterspezifizierte und daher nicht ausführbare Prozessmodelle, bei denen zum Beispiel nicht alle Kontrollflusskanten mit Quell- und Zielaktivitäten verknüpft sind. Sie modellieren üblicherweise Geschäftslogik, die in verschiedenen Kontexten wiederverwendet werden kann.

Zemni, Mammar und Hadj-Alouane [ZMH16] beschreiben einen Ansatz, der Modellierer bei der Erstellung von Prozessmodellen aus existierenden Fragmenten unterstützt, die semantisch äquivalente Aktivitäten enthalten. Der Modellierer muss die äquivalenten Aktivitäten identifizieren, um sie im konsolidierten Prozessmodell in eine einzelne Aktivität zusammenzufassen. Das konsolidierte Prozessmodell stellt dabei sicher, dass jede in den Fragmenten modellierte Ausführungsreihenfolge zwischen den Aktivitäten beibehalten wird. Die Ausführungsreihenfolge von Aktivitäten, die aus unterschiedlichen Fragmenten stammen, muss vom Modellierer manuell mittels *Behavioral Constraint Annotations* spezifiziert werden. Neben der Fokussierung auf die Konsolidierung von Fragmenten unterscheidet sich der Ansatz daher von dem in dieser Arbeit zusätzlich dadurch, dass die semantischen Äquivalenzen

[1] Rose et al. nutzen Prozessgraphen als Abstraktion von konkreten Prozessmodellierungssprachen. Sie bestehen nur aus abstrakten Knoten, Kanten und Gateways.

und die Behavioral Constraint Annotations in den Fragmenten manuell vom Modellierer spezifiziert werden müssen.

Sousa und Colaço [CS17] argumentieren, dass in Unternehmen, zum Beispiel abhängig von der jeweiligen Abteilung, unterschiedliche Sichten auf dasselbe Prozessmodell existieren. Diese Sichten sind ebenfalls als Prozessmodelle oder Fragmente modelliert. Sie beschreiben eine Methode, mit der die Sichten in einzelne Prozessmodelle zusammengefügt werden können [CS17; SCC19; CS20]. Dazu wird eine Taxonomie erstellt, die die Aktivitäten in den Sichten nach den Dimensionen von Zachmann [Zac87] kategorisiert. Diese Dimensionen beschreiben zum Beispiel, wo eine Aktivität ausgeführt wird, wer sie ausführt oder wann sie ausgeführt wird. Mittels der Taxonomieelemente werden die semantischen Beziehungen zwischen den Aktivitäten identifiziert. Basierend auf diesen Beziehungen werden wiederum die Kontrollflussbeziehungen abgeleitet, die die Aktivitäten im konsolidierten Prozessmodell zueinander haben.

Eberle et al. [ELS+10] definieren Operationen, um nicht überlappende, d.h. semantisch disjunkte Fragmente zur Laufzeit zu Prozessmodellen hinzuzufügen. Dazu werden die Fragmente während der Ausführung des Prozessmodells an unterspezifizierte Kontrollflusskanten angefügt und das Prozessmodell damit sukzessive vervollständigt. Sonntag und Karastoyanova [SK13] beschreiben, wie sich wissenschaftliche Prozessmodelle zur Laufzeit explorativ modellieren lassen, indem deren Instanzen zur Laufzeit pausiert und neue Aktivitäten vom Modellierer hinzugefügt bzw. entfernt werden. Weiß et al. erweitern diesen Ansatz auf die explorative Modellierung von Choreographien [WAHK17]. Die Ansätze haben also das Ziel, Prozessmodelle manuell zur Laufzeit zu erstellen bzw. zu modifizieren. Um die Laufzeitmodifikationen zu ermöglichen, muss, wie zum Beispiel von Eberle et al.[ELU10] beschrieben, die Workflow-Engine entsprechend angepasst werden. Die Prozesskonsolidierung findet automatisiert zur Entwurfszeit statt und konsolidierte Prozessmodelle können auf jeder BPEL-kompatiblen Workflow-Engine ausgeführt werden.

Ein weiteres Forschungsfeld befasst sich mit der Integration von Prozessmodellen, die mittels unterschiedlicher Metamodellen erstellt wurden. Dort steht also weniger die Konsolidierung von Prozessmodellen im Vordergrund, sondern deren Unifizierung in ein einheitliches Metamodell, so dass diese auf derselben Workflow-Engine ausgeführt werden und somit miteinander interagieren können. Eine Methode für die Transformation von Prozessmodellen wird von Hornung, Koschmider und Mendling [HKM06] beschrieben. Görlach, Leymann und Claus [GLC13] stellen einen Ansatz vor, mit dem Prozessmodelle, die auf verschiedenen Metamodellen basieren, unifiziert werden können, indem sie in eine einheitliche Grammatik transformiert werden.

Die Konsolidierung von Modellen ist auch Forschungsgegenstand im Datenbankbereich, um verschiedene Datenbankschemata in ein einzelnes Schema zu integrieren [RB01; BLN86]. Bei den zusammenzuführenden Schemata kann es sich um unterschiedliche Varianten desselben Schemas oder komplett unabhängig entwickelter Schemata handeln. Ein wesentlicher Schritt bei der Schemaintegration ist die Identifikation von semantisch äquivalenten Elementen in den unterschiedlichen Schemata, zum Beispiel semantisch äquivalente Entitäten in Entity-Relationship-Modellen [Che77]. Da in den meisten Fällen nicht alle semantisch äquivalenten Elemente automatisch bestimmt werden können [MWJ99], kann dieser Schritt maximal teilautomatisiert werden. Datenbankschemata modellieren, wie Informationen abgespeichert werden und welche Beziehungen zwischen diesen bestehen, sie bilden aber kein Verhalten ab. Daher können die Techniken zur Integration von Datenbankschemata nicht direkt auf die Prozesskonsolidierung übertragen werden.

Software Engineering beschäftigt sich primär mit der Zusammenführung von unterschiedlichen Varianten von Programmcode, die ihren Ursprung in der gleichen Codebasis haben [Men02]. Daher liegt dort der Fokus auf der Zusammenführung von Textdateien und der Auflösung von Konflikten zwischen diesen, aber nicht auf der Konsolidierung von Prozessmodellen bzw. von Modellen im Allgemeinen [RDUD13].

Ein Ziel der Prozesskonsolidierung ist es, Choreographien auf Workflow-Engines ausführbar zu machen, die die Choreographieausführung nicht unterstützen. Reimann et al. [RKDL08] verfolgen dasselbe Ziel. Dazu beschreiben sie die Schritte, mit denen eine BPEL4Chor-Choreographie in ausführbare BPEL-Prozessmodelle transformiert werden kann. Es werden dort aus der Choreographie allerdings mehrere kommunizierende Prozessmodelle erstellt und nicht ein einzelnes, wie in dem hier vorgestellten Prozesskonsolidierungsansatz.

Kapitel 3

Metamodell für Choreographien

Die Konsolidierungsoperation wird auf Choreographien angewandt, die als Verbindungsmodelle [Kop16b; DKB08] modelliert sind. Bei Verbindungsmodellchoreographien (im weiteren Verlauf der Arbeit kurz Choreographien genannt) ist das Verhalten der Teilnehmer über Prozessmodelle und die Kommunikation zwischen den Teilnehmern über die Kommunikationsaktivitäten der Prozessmodelle modelliert.

Weiß et al. [WAHK17] haben für diese Art von Choreographien ein Metamodell beschrieben. Für Prozessmodelle wurde von Leymann und Roller [LR00] ein Metamodell definiert. Beide Arbeiten bilden die Grundlage für das hier vorgestellte Metamodell. Da der Konsolidierungsansatz speziell für BPEL-Prozessmodelle entwickelt wurde, berücksichtigt das Metamodell zusätzlich Eigenschaften der von Decker et al. entwickelten auf BPEL-basierende Choreographiesprache BPEL4Chor [DKLW07] und Modellelemente aus der

BPEL-Formalisierung von Kopp, Mietzner und Leymann [KML08].

Dieses Kapitel ist wie folgt aufgebaut. Das primäre Element des Metamodells ist die Choreographie, deren Bestandteile in Abschnitt 3.1 beschrieben werden. Die Elemente des Metamodells, die zur Modellierung der Prozessmodelle notwendig sind, werden in Abschnitt 3.2 erläutert. Abschnitt 3.3 geht auf Choreographie- und Prozessfragmente ein. In Abschnitt 3.4 werden die Elemente von Konversationen, also Instanzen von Choreographien definiert.

3.1. Choreographien

Eine Choreographie c beschreibt das Verhalten der Teilnehmer und deren Interaktionen. Sie wird durch das folgende Tupel beschrieben:

$$c = (\mathfrak{P}, \mathfrak{P}^{set}, ML, P)$$

Die Teilnehmer einer Choreographie werden durch die Menge $\mathfrak{P}$ repräsentiert. Sie müssen der Konsolidierungsoperation bekannt sein (siehe Abschnitt 5.3). Daher darf die Menge $\mathfrak{P}$ zur Laufzeit nicht verändert werden, d.h. es dürfen während der Ausführung keine Teilnehmer dynamisch zu $\mathfrak{P}$ hinzugefügt werden. Dies ist verglichen mit BPEL4Chor eine Einschränkung, wo Teilnehmer auch erst zur Laufzeit einer Choreographie hinzugefügt werden können. Es müssen allerdings nicht alle Teilnehmer aus $\mathfrak{P}$ zur Laufzeit an den Interaktionen teilnehmen. $\mathfrak{P}$ wird im folgenden deshalb als die Menge der *potentiellen Teilnehmer* der Choreographie bezeichnet.

Die Menge $\mathfrak{P}^{set} \subseteq \wp(\mathfrak{P})$ dient zu deren Gruppierung. In ML sind die Nachrichtenkanten der Choreographie enthalten. Die Prozessmodelle, die das Verhalten der Teilnehmer spezifizieren, werden durch die Menge P repräsentiert.

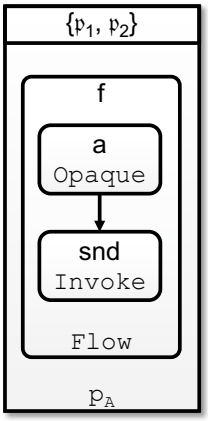

Abbildung 3.1.: Beispiel einer Teilnehmermenge mit Teilnehmertyp

3.1.1. Teilnehmer und Teilnehmermengen

Einem Teilnehmer $\mathfrak{p} \in \mathfrak{P}$ einer Choreographie muss eine Verhaltensbeschreibung in Form eines Prozessmodells zugewiesen werden. Die Zuweisung erfolgt dabei über die Abbildung $\text{type}_{\mathfrak{p}} : \mathfrak{P} \rightarrow P$. Dieses Prozessmodell wird auch als *Teilnehmertyp* bezeichnet. Die Teilnehmer $\mathfrak{p}_1$ und $\mathfrak{p}_2$ in Abbildung 3.1 haben zum Beispiel den Teilnehmertyp p_A.

Eine Teilnehmermenge $\mathfrak{p}^{set} \in \mathfrak{P}^{set}$ gruppiert eine Menge von Teilnehmern, über die zum Beispiel in Schleifen, wie in Abschnitt 3.2.4.10 erläutert, iteriert werden kann. Es gilt $\mathfrak{P}^{set} \subseteq \wp(\mathfrak{p})$. Jeder Teilnehmer einer Teilnehmermenge $\mathfrak{p}^{set}$ muss die gleiche Verhaltensbeschreibung besitzen:

$$\forall \mathfrak{p}_i, \mathfrak{p}_j \in \mathfrak{p}^{set} : \text{type}_{\mathfrak{p}}(\mathfrak{p}_i) = \text{type}_{\mathfrak{p}}(\mathfrak{p}_j)$$

3.1.2. Nachrichtenkanten

Die Interaktionen, d.h. der Nachrichtenaustausch zwischen den Teilnehmern einer Choreographie, werden mittels Nachrichtenkanten zwischen Sendeaktivitäten $A_{\texttt{invoke}}$ und Empfangskonstrukten CO_{rcv} modelliert, die in Abschnitt 3.2.4 definiert werden. Die Nachrichtenkanten werden dabei durch die Menge $ML \subseteq \mathfrak{P}^{set} \times A_{\texttt{invoke}} \times \mathfrak{P}^{set} \times CO_{rcv}$ repräsentiert. In einer Nachrichtenkante $ml = (\mathfrak{p}^{set}_{snd}, a_{snd}, \mathfrak{p}^{set}_{rcv}, co_{rcv}) \in ML$ enthält die Teilnehmermenge $\mathfrak{p}^{set}_{snd}$ die potentiellen Sender der Nachricht. Diese senden die Nachricht über die Aktivität a_{snd} an die potentiellen Empfänger $\mathfrak{p}^{set}_{rcv}$, die diese wiederum über ein Empfangskonstrukt co_{rcv} empfangen. Das Empfangskonstrukt kann, wie im Abschnitt 3.2.4.2 erläutert, entweder als ein Receive oder ein Nachrichtenereignis innerhalb eines Picks modelliert werden.

Jede Nachrichtenkante ml muss dabei die folgenden Eigenschaften erfüllen [DKLW07]:

(I) Es kann der Versand einer Nachricht entweder von mehreren Sendern an einen Empfänger oder von einem Sender an mehrere Empfänger modelliert werden:

$$(|\mathfrak{p}^{set}_{snd}| \geq 1 \wedge |\mathfrak{p}^{set}_{rcv}| = 1) \vee (|\mathfrak{p}^{set}_{snd}| = 1 \wedge |\mathfrak{p}^{set}_{rcv}| \geq 1)$$

(II) Die tatsächlichen Sender bzw. Empfänger werden zur Laufzeit der Choreographie bestimmt und müssen einer Teilmenge von $\mathfrak{p}^{set}_{snd}$ bzw. $\mathfrak{p}^{set}_{rcv}$ entsprechen. Existieren mehrere potentielle Sender oder Empfänger, muss sich die Sendeaktivität bzw. das Empfangskonstrukt in einer Schleife $a_{loop} \in A_{\texttt{loop}}$ (siehe Abschnitt 3.2.4) befinden:

$$|\mathfrak{p}^{set}_{snd}| \geq 1 \Leftrightarrow \textsc{Ancestors}(a_{snd}) \in A_{\texttt{loop}}$$

bzw.

$$|\mathfrak{p}^{set}_{rcv}| \geq 1 \Leftrightarrow \textsc{Ancestors}(co_{rcv}) \in A_{\texttt{loop}}$$

Um zu prüfen, ob sich a_{snd} in einer Schleife befindet, ermittelt die Funktion $\textsc{Ancestors} : (A \cup E^{msg}) \rightarrow \wp(A)$, die in Abschnitt 3.2.3 definiert ist, deren direkte und indirekte Elternaktivitäten.

(III) Die Sender einer Nachrichtenkante können nicht gleichzeitig deren Empfänger sein, d.h. $\pi_1(ml) \cap \pi_3(ml) = \emptyset$. Das bedeutet insbesondere, dass ein Teilnehmer keine Nachricht an sich selber senden darf.

(IV) Die Sendeaktivität muss sich in der Verhaltensbeschreibung also im Prozessmodell der sendenden Teilnehmer befinden:

$$\forall \mathfrak{p}_{snd} \in \mathfrak{p}_{snd}^{set} : a_{snd} \in \pi_1(p_{snd}) \text{ mit } p_{snd} = \mathsf{type}_{\mathfrak{p}}(\mathfrak{p}_{snd})$$

Prozessmodelle werden im folgenden Abschnitt definiert. Mittels der Projektion $\pi_1(p_{snd})$ können deren Aktivitäten ermittelt werden.

(V) Das Empfangskonstrukt co_{rcv} muss sich in der Verhaltensbeschreibung der potentiellen Empfänger befinden, entweder als Receive-Aktivität oder als Pick-Aktivität, die das Nachrichtenereignis enthält :

$$\forall \mathfrak{p}_{rcv} \in \mathfrak{p}_{rcv}^{set} : a_{rcv} \in \pi_1(p_{rcv}) \text{ mit } p_{rcv} = \mathsf{type}_{\mathfrak{p}}(\mathfrak{p}_{rcv})$$
$$\text{und } a_{rcv} = \begin{cases} co_{rcv}, co_{rcv} \in A_{\texttt{receive}} \\ \textsc{Parent}(co_{rcv}), co_{rcv} \in E^{msg} \end{cases}$$

Die Menge der Receive-Aktivitäten $A_{\texttt{receive}}$ wird in Abschnitt 3.2.4 und die Menge der Nachrichtenereignisse E^{msg} in Abschnitt 3.2.2 definiert. Die Funktion $\textsc{Parent} : A \cup E^{msg} \rightarrow A \cup \{\perp\}$ gibt die Elternaktivität des Empfangskonstrukt zurück (siehe Abschnitt 3.2.3).

(VI) Ein Invoke kann nur als Sendeaktivität genau einer Nachrichtenkante fungieren:

$$\forall ml_1, ml_2 \in ML : \pi_2(ml_1) = \pi_2(ml_2) \Rightarrow ml_1 = ml_2$$

(VII) Ein Empfangskonstrukt kann nur genau einer Nachrichtenkante zugewiesen werden:

$$\forall ml_1, ml_2 \in ML : \pi_4(ml_1) = \pi_4(ml_2) \Rightarrow ml_1 = ml_2$$

3.2. Prozessmodelle

Sei P die Menge alle Prozessmodelle, dann wird ein Prozessmodell $p \in P$ durch das folgende Tupel beschrieben:

$$p = (A, HR, L, V, \mathfrak{C}, Ex_{\mathcal{B}}, \mathcal{L}, E)$$

Die Menge A repräsentiert die Aktivitäten, die Menge HR die Hierarchiebeziehungen, die Menge L die Kontrollflusskanten, die Menge V die Variablen, die Menge $\mathfrak{C}$ die Bedingungen, die Menge $Ex_{\mathcal{B}}$ die booleschen Ausdrücke, die Menge $\mathcal{L}$ die Bezeichner und die Menge E die Ereignisse des Prozessmodells.

3.2.1. Variablen, Ausdrücke, Datentypen & Bedingungen

Während der Prozessausführung werden die Daten in den Variablen V gespeichert. Es werden in dieser Arbeit nur konkret die Variablenwerte benötigt, die entweder Teilnehmer $\mathfrak{P}$ oder boolesche Werte $\mathcal{B} = \{\texttt{true}, \texttt{false}\}$ repräsentieren. Alle andere Werte (Zeichenketten, Zahlen etc.) werden abstrakt durch die Menge O (Objekt) repräsentiert. Einer Variable wird ein Wert über die Abbildung $\text{value}_V : V \rightarrow \mathfrak{P} \cup \mathcal{B} \cup O$ zugewiesen.

Boolesche Ausdrücke $Ex_{\mathcal{B}}$ sind Prädikate über Variablen des Prozessmodells,

die zu `true` oder `false` evaluieren. Eine Bedingung $c \in \mathfrak{C}$ kann über einen booleschen Ausdruck mittels der Abbildung $\text{expr}_{\mathcal{B}} : \mathfrak{C} \to Ex_{\mathcal{B}}$ definiert werden, zum Beispiel $\text{expr}_{\mathcal{B}}(c) = v < 5$ (mit $v \in V$).

3.2.2. Ereignisse

Zur Laufzeit eines BPEL-Prozesses können verschiedene Ereignisse E auftreten, die bestimmte *Event-Handler* aktivieren, die wiederum Aktivitäten zur Verarbeitung des jeweiligen Ereignisses ausführen. Die Scope-Aktivität (siehe Abschnitt 3.2.4.8) stellt beispielsweise Handler bereit, um auf Fehlerereignisse zu reagieren. Die verschiedenen Ereignistypen werden durch die Menge T_E repräsentiert:

$$T_E := \{t_E^{msg}, t_E^{timer}, t_E^{fault}\}$$

Über die Abbildung $\text{type}_E : E \to T_E$ kann einem Ereignis sein Typ zugewiesen werden. Mengen von Ereignissen eines bestimmten Typs werden durch einen Index gekennzeichnet, d.h. $E^{t_E} = \{e \in E \mid \text{type}_E(e) = t_E\}$. Es kann zwischen Standard- oder Fehlerereignissen unterschieden werden:

$$E := E^{std} \cup E^{fault}$$

Die Menge $E^{std} := E^{timer} \cup E^{msg}$ repräsentiert Timer- und Nachrichtenereignisse, also Ereignisse des regulären Kontrollflusses. Timer-Ereignisse E^{timer} werden zur Laufzeit nach einer bestimmten Zeitdauer ausgelöst, nachdem die Aktivität, der das Ereignis zugeordnet ist, gestartet wurde. Ereignisse können, wie in Abschnitt 3.2.4.4 bzw. 3.2.4.8 beschrieben, Pick-Aktivitäten und Scopes zugeordnet werden. Die Zuweisung der Zeitdauer zu einem Ereignis erfolgt über einen Ausdruck mittels der Abbildung $\text{for} : E^{timer} \to \mathbb{T}$. Die Zeitdauer wird dabei abstrakt durch natürliche Zahlen repräsentiert, es gilt damit $\mathbb{T} \subseteq \mathbb{N}$.

Nachrichtenereignisse werden beim Empfang bestimmter Nachrichten aktiviert. Die Abbildung $\text{message} : E^{msg} \to ML$ weist dem Ereignis eine Nachrich-

tenkante zu, über die die das Ereignis aktivierende Nachricht gesendet wird. Eine Nachrichtenkante darf dabei genau nur einem Nachrichtenereignis zugewiesen werden:

$$\forall e_1^{msg}, e_2^{msg} \in E^{msg} : e_1^{msg} \neq e_2^{msg} \Rightarrow \mathsf{message}(e_1^{msg}) \neq \mathsf{message}(e_2^{msg})$$

Die Menge E^{fault} repräsentiert Fehlereignisse, also Abweichungen vom regulären Kontrollfluss. In BPEL können aus Fehlerereignissen Terminierungs- und Kompensationsereignisse resultieren. Diese werden in dieser Arbeit aber nur am Rande betrachtet.

Zur Laufzeit wird, wie in Abschnitt 3.4.2 erläutert, zwischen regulären und irregulären Kontrollfluss anhand von Aktivitätszuständen unterschieden.

3.2.3. Kontrollfluss

Dieses Metamodell erlaubt wie BPEL die blockbasierte und graphbasierte Kontrollflussmodellierung [KMWL09]. Die blockbasierte Modellierung erfolgt mittels strukturierter Aktivitäten, die abhängig von ihrem Typ zur Laufzeit eine bestimmte Ausführungsreihenfolge für ihre Kindaktivitäten festlegen. Bei der graphbasierten Modellierung wird die Ausführungsreihenfolge von Aktivitäten durch Kontrollflusskanten zwischen diesen definiert.

3.2.3.1. Blockbasierter Kontrollfluss

Die blockbasierte Modellierung wird in dieser Arbeit durch Hierarchiebeziehungen zwischen Prozessmodellen bzw. strukturierten Aktivitäten $A_{\texttt{struc}}$ und ihren direkten Kindaktivitäten abgebildet. Hierarchiebeziehungen *HR* sind folgendermaßen definiert:

$$HR \subseteq (P \cup A_{\texttt{struc}}) \times T_{HR} \times A$$

Jede Hierarchiebeziehung $hr = (a_{parent}, \mathfrak{t}_{HR}, a_{child}) \in HR$ charakterisiert die Schachtelung zwischen einem Prozessmodell bzw. einer Elternaktivität a_{parent} und der Kindaktivität a_{child} über ihren Beziehungstypen $\mathfrak{t}_{HR} \in T_{HR}$, wobei die Menge der Typen wie folgt definiert ist:

$$T_{HR} := \{\mathfrak{t}_{HR}^{ch}\} \cup E$$

Der Beziehungstyp $\mathfrak{t}_{HR}^{ch}$ wird für direkte Kindaktivitäten verwendet, deren Ausführung bedingungslos möglich ist, sobald die Elternaktivität ausgeführt wird. Für jede strukturierte Aktivität muss genau eine Hierarchiebeziehung definiert sein, in der diese Aktivität Elternaktivität ist. Falls ein bestimmtes Ereignis $e \in E$ auftreten muss, um die Kindaktivitäten auszuführen, entspricht der Beziehungstyp diesem Ereignis.

Zwei Hierarchiebeziehungen repräsentieren dasselbe Element aus der Menge *HR*, wenn sie die folgenden Eigenschaften erfüllen:

$$\forall hr_i, hr_j \in HR : \pi_1(hr_i) = \pi_1(hr_j) \wedge \pi_3(hr_i) = \pi_3(hr_j) \Rightarrow hr_i = hr_j$$

Folglich dürfen zwei Aktivitäten nur über genau eine Hierarchiebeziehung miteinander in Relation gesetzt werden. Außerdem dürfen keine zyklischen Hierarchiebeziehungen entstehen:

$$\forall hr_i, hr_j \in HR : \pi_1(hr_i) = \pi_3(hr_i) \wedge \pi_1(hr_j) = \pi_3(hr_i) \Rightarrow hr_i = hr_j$$

Mit den folgenden Funktionen können, basierend auf den in einer Choreographie definierten Hierarchiebeziehungen, die Kindaktivitäten, Elternaktivitäten, Vorfahren etc. einer Aktivität a ermittelt werden [KML08]:

- PARENT : $A \cup E^{msg} \rightarrow A \cup \{\perp\}$ gibt die Elternaktivität der Aktivität zurück oder $\perp$, falls die Aktivität keine Elternaktivität besitzt. Wird ein Nachrichtenereignis übergeben, wird die Pick-Aktivität zurück

gegeben, die dem Nachrichtenereignis zugeordnet ist.

- CHILDREN : $A_{\texttt{struc}} \rightarrow \wp(A)$ ermittelt die direkten Kindaktivitäten einer Aktivität[1].
- DESCENDANTS : $A_{\texttt{struc}} \rightarrow \wp(A)$ bestimmt alle Nachfahren, d.h. direkte und indirekte Kindaktivitäten einer Aktivität.
- ANCESTORS : $(A \cup E^{msg}) \rightarrow \wp(A)$ bestimmt alle Vorfahren einer Aktivität, d.h. direkte und indirekte Elternaktivitäten. Wird ein Nachrichtenereignis übergeben, werden die Vorfahren der zugehörigen Pick-Aktivität bestimmt.
- PARENTSCOPE : $(A \cup E^{msg}) \rightarrow S)$ gibt den direkten Eltern-Scope einer Aktivität oder eines Nachrichtenereignisses zurück.

3.2.3.2. Graphbasierter Kontrollfluss

Kontrollflusskanten $L \subseteq A \times A \times \mathfrak{C}$ repräsentieren die gerichteten Kanten, die zwischen zwei Aktivitäten eine direkte Kontrollflussabhängigkeit definieren. Jede Kontrollflusskante $(a_{src}, a_{trg}, \mathfrak{c}) \in L$ besteht aus einer Quellaktivität a_{src}, einer Zielaktivität a_{trg} und einer Transitionsbedingung $\mathfrak{c} \in \mathfrak{C}$.

Zwei Kontrollflusskanten repräsentieren dasselbe Element aus der Menge L, wenn sie die folgenden Eigenschaften erfüllen:

$$\forall l_i, l_j \in L : \pi_1(l_i) = \pi_1(l_j) \wedge \pi_2(l_i) = \pi_2(l_j) \Rightarrow l_i = l_j$$

Mit den folgenden Funktionen können basierend auf den in der Choreographie definierten Kontrollflusskanten eingehende und ausgehende Kanten, sowie die Vorgänger- und Nachfolgeaktivitäten einer Aktivität $a \in A$ bestimmt werden [KML08]:

[1] Strukturierte Aktivitäten $A_{\texttt{struc}}$ und Scope-Aktivitäten (kurz Scopes) S werden in Abschnitt 3.2.4 definiert.

- $\textsc{Incoming}_L : A \to \wp(L)$ ermittelt alle eingehenden Kanten einer Aktivität.
- $\textsc{Outgoing}_L : A \to \wp(L)$ ermittelt alle ausgehenden Kanten einer Aktivität.
- $\textsc{Predecessors} : A \to \wp(A)$ ermittelt die direkten Vorgängeraktivitäten einer Aktivität.
- $\textsc{Predecessors}_{all} : A \to \wp(A)$ ermittelt die direkten und indirekten Vorgängeraktivitäten einer Aktivität.
- $\textsc{Successors} : A \to \wp(A)$ ermittelt die direkten Nachfolgeaktivitäten einer Aktivität.
- $\textsc{Successors}_{all} : A \to \wp(A)$ ermittelt die direkten und indirekten Nachfolgeaktivitäten einer Aktivität.

Gilt $a_i \in \textsc{Predecessors}_{all}(a_j)$, so hat die Aktivität a_j eine *Synchronisationsabhängigkeit* zu Aktivität a_i, d.h. eine Instanz von a_i muss immer einen Endzustand erreicht haben, bevor eine Instanz von a_j ausgeführt werden kann[1].

Jeder Aktivität mit eingehenden Kontrollflusskanten kann mit der Abbildung $\mathsf{joinCond} : A \to Ex_{\mathcal{B}}$ eine *Eintrittsbedingung* (engl. join condition) zugewiesen werden. Diese legt fest, welche eingehenden Kanten aktiviert werden müssen, um die Ausführung der Aktivität zu starten. Der boolesche Ausdruck für die Eintrittsbedingung referenziert dabei die Kontrollflusskanten, die aktiviert sein müssen. Zum Beispiel definiert $\mathsf{joinCond}(a) = \mathtt{l}_1 \vee \mathtt{l}_2$, dass die Kontrollflusskante l_1 oder l_2 aktiviert sein muss, um die Aktivität a zu starten. Ist die Eintrittsbedingung für eine Aktivität nicht explizit spezifiziert, wird analog zur BPEL-Semantik mit der Ausführung der Aktivität dann begonnen, wenn eine ihrer eingehenden Kontrollflusskanten aktiviert wurde.

[1] Auf Synchronisationsabhängigkeiten zwischen Aktivitäten und wie diese die Zustände von Aktivitätsinstanzen beeinflussen, wird im Detail in Kapitel 4 eingegangen.

Evaluiert die Eintrittsbedingung zur Laufzeit zu `false`, kann eine Aktivität einen *Join-Fehler* $e^{joinFault} \in E^{joinFault} \subset E^{fault}$ auslösen. Dazu muss der Aktivität über die Abbildung $\mathsf{joinFault} : A \rightarrow Ex_{\mathcal{B}}$ der Wert `true` zugewiesen werden. Standardmäßig wird in dieser Arbeit aber davon ausgegangen, dass die Aktivitäten keine Fehler auslösen, wenn die Eintrittsbedingung zu `false` evaluiert.

Die Eintrittsbedingung einer Aktivität wird erst dann ausgewertet, wenn die Transitionsbedingungen aller eingehenden Kanten evaluiert wurden.

3.2.3.3. Restriktionen der Kontrollflussmodellierung

Für Kontrollflusskanten gelten die folgenden vom BPEL-Metamodell abgeleiteten Restriktionen, die ebenfalls von Kopp, Mietzner und Leymann in [KML08] formalisiert wurden und hier deshalb nur informell beschrieben werden:

- Kontrollflusskanten dürfen keinen Zyklus im Kontrollfluss erzeugen.
- Kontrollflusskanten dürfen nicht in den Fault-Handler eines Scopes hinein zeigen, d.h. die Quellaktivität darf sich nicht außerhalb und die Zielaktivität nicht innerhalb eines Fault-Handlers befinden. Ausgehende Kontrollflusskanten, deren Quellaktivität sich innerhalb und deren Zielaktivität sich außerhalb des Fault-Handlers befindet, sind hingegen erlaubt.
- Kontrollflusskanten dürfen aus Schleifen und Event-Handlern von Scopes weder hinaus noch hinein zeigen, d.h. die Quell- und Zielaktivität einer Kontrollflusskante muss sich in derselben Schleife bzw. im selben Event-Handler befinden.
- Die Quell- und Zielaktivität müssen dieselbe Flow-Aktivität (siehe Abschnitt 3.2.4 Seite 69) als indirekten Vorfahren besitzen.

Zur Reduzierung der Fallunterscheidungen bei der Verifikation der Konso-

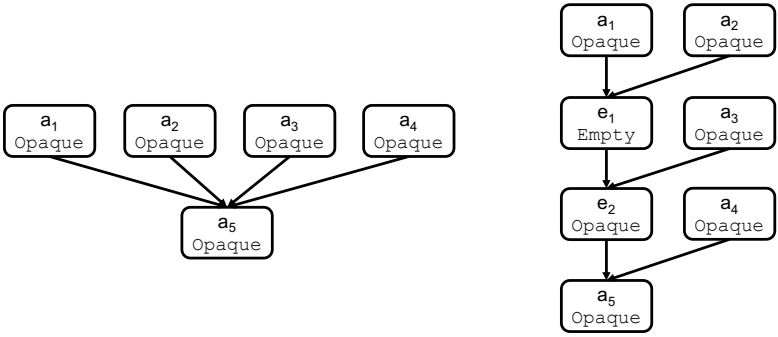

Abbildung 3.2.: Reduzierung der eingehenden Kontrollflusskanten einer Aktivität durch das Einfügen von Empty-Aktivitäten

lidierungsoperation gilt die zusätzliche Einschränkung, dass eine Aktivität maximal zwei eingehende und ausgehende Kontrollflusskanten hat:

$$\forall a \in A : |\textsc{Incoming}_L(a)| \leq 2 \wedge |\textsc{Outgoing}_L(a)| \leq 2$$

Diese Einschränkung dient ausschließlich zur Vereinfachung der Verifikation und schränkt nicht die Interaktionsmuster ein, die konsolidiert werden können (siehe Kapitel 7). Die in der Arbeit vorgestellten Konsolidierungsalgorithmen unterstützen auch Aktivitäten mit mehr als zwei eingehenden bzw. ausgehenden Kontrollflusskanten. Außerdem ist es möglich, vor der Konsolidierung den Kontrollfluss so zu transformieren, dass durch das Einfügen von Kaskaden von Empty-Aktivitäten $A_{\texttt{empty}}$ (siehe Abschnitt 3.2.4) jede Aktivität nur zwei Vorgänger hat (wie in Abschnitt 4.2.3 erläutert, führen Empty-Aktivitäten keine Funktionen aus). Dies ist exemplarisch in Abbildung 3.2 dargestellt, wo die Aktivität a_5 vier und durch das Einfügen von Empty e nur zwei eingehende Kontrollflusskanten hat und somit der oben erwähnten Kontrollflusseinschränkung entspricht. Die eingefügten Empty-Aktivitäten können nach der Konsolidierung wieder entfernt werden.

3.2.4. Aktivitäten

Aktivitäten repräsentieren die Arbeitsschritte eines Prozessmodells. Die operationale Semantik einer Aktivität wird durch ihren Typ bestimmt. Das Metamodell orientiert sich an den BPEL-Aktivitätstypen und umfasst nur die Typen, die notwendig sind, um die Interaktionsmuster aus [BDtH05] zu modellieren und zu konsolidieren (siehe Kapitel 7). So werden zum Beispiel Kompensations- oder Sequence-Aktivitäten ausgeklammert[1]. Zusammenfassend unterscheidet das Metamodell daher die folgenden Typen, die in diesem Abschnitt erläutert werden:

$$T_A := \{\texttt{invoke}, \texttt{receive}, \texttt{assign}, \texttt{empty}, \texttt{throw}, \texttt{opaque}$$
$$\texttt{wait}, \texttt{scope}, \texttt{flow}, \texttt{pick}, \texttt{while}, \texttt{forEach}\}$$

Der Typ kann einer Aktivität über die Abbildung $\text{type}_A : A \to T_A$ zugewiesen werden. Zur Kennzeichnung einer Menge von Aktivitäten eines bestimmten Typs wird ebenfalls ein Index verwendet:

$$A_{t_A} := \{a \in A \mid \text{type}_A(a) = t_A\} \text{ für } t_A \in T_A$$

Es wird zwischen *Basisaktivitäten* A_{Basic} und *strukturierten Aktivitäten* $A_{\texttt{struc}}$ unterschieden. Letztere enthalten eine oder mehrere Kindaktivitäten und ermöglichen somit die Modellierung von blockbasiertem Kontrollfluss (siehe Abschnitt 3.2.3):

$$A_{\texttt{struc}} := A_{\texttt{scope}} \cup A_{\texttt{flow}} \cup A_{\texttt{pick}} \cup A_{\texttt{while}} \cup A_{\texttt{forEach}}$$

Die Menge der Basisaktivitäten ist folglich definiert als $A_{Basic} := A \setminus A_{\texttt{struc}}$. Als Standardaktivitäten werden alle Aktivitäten bezeichnet, die nicht vom Typ `scope` sind, d.h. $A_{standard} := \{a \in A \mid \text{type}_A(a) \neq \texttt{scope}\}$.

[1] Die sequentielle Ausführung von Aktivitäten lässt sich alternativ über den graphbasierten Kontrollfluss modellieren.

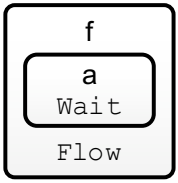

Abbildung 3.3.: Beispiel einer strukturierten Flow-Aktivität mit Kindaktivität

Bei der grafischen Darstellung von Aktivitäten wird immer der Aktivitätstyp mit angegeben. Abbildung 3.3 zeigt zum Beispiel eine Flow-Aktivität f mit einer Wait-Aktivität a als Kind.

3.2.4.1. Geschäftsaktivitäten

Geschäftsaktivitäten A_{opaque} dienen als Platzhalter für eine oder mehrere Aktivitäten, die eine bestimmte Geschäftslogik implementieren. Diese schließen Aktivitäten mit ein, die die Kommunikation mit externen Webservices realisieren. Alle Aktivitäten die die Kommunikation der Teilnehmer innerhalb der Choreographie modellieren, werden nicht durch Geschäftsaktivitäten abstrahiert. Dies sind zum Beispiel Kommunikationsaktivitäten und die Schleifen, in denen sie sich befinden. Mit diesen Schleifen wird die Kommunikation mit mehreren Teilnehmern realisiert. Sie müssen somit bei der Konsolidierung, wie in Kapitel 6 erläutert, berücksichtigt werden.

Über die Relation $\equiv_A \subseteq A_{\text{opaque}} \times A_{\text{opaque}}$ wird definiert, dass eine Geschäftsaktivität identisch zu einer anderen Geschäftsaktivität ist. Als Schreibweise für die Identität wird $a_i \equiv_A a_j$ verwendet.

Mittels der Identitätsrelation wird in dieser Arbeit festgelegt, welche Aktivität in der originalen Choreographie durch welche Aktivität im konsolidierten Prozessmodell repräsentiert wird. Die identische Aktivität im konsolidierten Prozessmodell wird in Kapitel 5 auch als *Teilnehmerduplikat* der entsprechenden Aktivität aus der Choreographie bezeichnet.

3.2.4.2. Kommunikationsaktivitäten

Kommunikationsaktivitäten A_{com} dienen ausschließlich dazu, den Nachrichtenaustausch zwischen den Teilnehmern der Choreographien zu modellieren. Dies sind also Invoke-, Receive und Pick-Aktivitäten[1]:

$$A_{com} := A_{\texttt{invoke}} \cup A_{\texttt{receive}} \cup A_{\texttt{pick}}$$

Invoke-Aktivitäten dienen zum Versand von Nachrichten an einen anderen Teilnehmer. Dieser kann eine Nachricht dann über ein Nachrichtenempfangendes Konstrukt, d.h. über ein Receive oder ein Nachrichtenereignis innerhalb eines Picks empfangen. Die Empfangskonstrukte werden durch die Menge $CO_{rcv} = A_{\texttt{receive}} \cup E^{msg}$ repräsentiert. Der sendende Teilnehmer wird dem Empfangskonstrukt über die Abbildung $\mathsf{sender} : CO_{rcv} \times \mathbb{N} \rightarrow \mathfrak{P}$ zugewiesen.

Wird dasselbe Receive bzw. ein Nachrichtenereignis während einer Konversation (Abschnitt 3.4) mehrmals ausgeführt, zum Beispiel wenn es sich in einer Schleife befindet, kann ihm während jeder Ausführung $n \in \mathbb{N}$ ein anderer Sender zugewiesen werden. Dabei muss zur Laufzeit sichergestellt werden, dass sich der sendende Teilnehmer in der Teilnehmermenge der Sender befindet, in der das Receive bzw. das Nachrichtenereignis als Empfangskonstrukt fungiert:

$$\forall co_{rcv} \in CO_{rcv} \forall ml \in ML \forall n \in \mathbb{N} :$$
$$\mathsf{sender}(co_{rcv}, n) \in \pi_1(ml) \Rightarrow co_{rcv} = \pi_4(ml)$$

Mittels der Abbildung $\mathsf{outputVar} : CO_{rcv} \rightarrow V$ wird jedem Empfangskonstrukt eine Ausgabevariable zugewiesen, in die der Inhalt der zu empfangenen Nachricht kopiert wird. Bei der iterative Ausführung eines Empfangskonstrukts nutzt es immer dieselbe Ausgabevariable.

[1] Da in dieser Arbeit nur die Konsolidierung von asynchronen Kommunikationsszenarien diskutiert wird, sind Reply-Aktivitäten kein Bestandteil des Metamodells.

3.2.4.3. Invoke

Invoke-Aktivitäten $A_{\texttt{invoke}}$ kann der Nachrichtenempfänger mit der Abbildung $\text{receiver} : A_{\texttt{invoke}} \times \mathbb{N} \rightarrow \mathfrak{P}$ zugewiesen werden. Analog zu Empfangskonstrukten kann ein Invoke während jeder Ausführung eine Nachricht an einen anderen Empfänger senden. Der Empfänger muss sich dabei in der Teilnehmermenge befinden, die in der Nachrichtenkante definiert ist, in der das Invoke die sendende Aktivität ist:

$$\forall a_{snd} \in A_{\texttt{invoke}} \forall ml \in ML \forall n \in \mathbb{N} :$$
$$\text{receiver}(snd, n) \in \pi_3(ml) \Rightarrow a_{snd} = \pi_2(ml)$$

Für jedes Invoke muss festgelegt werden, aus welcher Variable des Prozessmodells der Inhalt der zu sendenden Nachricht kopiert wird. Dazu kann einer Invoke-Aktivität mittels der Abbildung $\text{inputVar} : A_{\texttt{invoke}} \rightarrow V$ eine Eingabevariable zugewiesen werden.

Sind ein Invoke und ein Empfangskonstrukt über eine Nachrichtenkante verbunden, muss der Wertebereich der Variablen, der dem Invoke mit inputVar zugewiesen wurde, dem Wertebereich entsprechen, der den Empfangskonstrukt über outputVar zugewiesen wurde.

Das Empfangskonstrukt darf zur Laufzeit die Nachricht erst in seine Ausgabevariable schreiben, nachdem das Invoke die Eingabevariable gelesen und damit deren Inhalt in die Nachricht geschrieben hat. Dies wird, wie in Abschnitt 4.2.14 erläutert, dadurch sichergestellt, dass die Instanz des Empfangskonstrukts erst ausgeführt wird, wenn die Ausführung der Instanz des Invokes erfolgreich beendet wurde.

3.2.4.4. Pick

Pick-Aktivitäten werden durch die Menge $A_{\texttt{pick}}$ repräsentiert. Dabei ist jede Kindaktivität dem Pick über ein Nachrichten- oder ein Timer-Ereignis zugewiesen:

$$A_{\texttt{pick}} \times T_{HR} \times A = A_{\texttt{pick}} \times E^{std} \times A$$

Eine Pick-Aktivität hat eine oder mehrere direkte Kindaktivitäten, die diesen Ereignissen zugewiesen sind:

$$\forall a_{\texttt{pick}} \in A_{\texttt{pick}} : \textsc{Children}(a_{\texttt{pick}}) \neq 0$$

Zur Laufzeit wird nur eine Kindaktivität ausgeführt und zwar die, deren Ereignis zuerst eintritt.

3.2.4.5. Assign

Assign-Aktivitäten $A_{\texttt{assign}}$ kopieren einen Wert von einem Quell- in ein Zielkonstrukt. Das Quellkonstrukt, eine Variable oder ein Teilnehmer, wird über die Abbildung $\text{assign}_{src} : A_{\texttt{assign}} \rightarrow V \cup \mathfrak{P}$ und das Zielkonstrukt über die Abbildung $\text{assign}_{trg} : A_{\texttt{assign}} \rightarrow V \cup \mathfrak{P}$ festgelegt. Handelt es sich bei beiden Konstrukten um eine Variable, müssen sie den gleichen Datentyp besitzen. Handelt es sich bei einem der Konstrukte um einen Teilnehmer, muss das andere Konstrukt eine Variable sein, die ebenfalls einen Teilnehmer enthält. In BPEL kann ein Assign mehrere Kopieroperationen ausführen. In diesem Metamodell werden zur Vereinfachung der Konsolidierungsalgorithmen nur Assigns mit einer Kopieroperation betrachtet.

3.2.4.6. Wait

Wait-Aktivitäten $A_{\texttt{wait}}$ werden über die Abbildung $\text{timer} : A_{\texttt{wait}} \rightarrow E^{timer}$ mit Timer-Ereignissen assoziiert. Tritt das Timer-Ereignis ein, wird die Ausführung der Wait-Aktivität beendet.

3.2.4.7. Throw

Throw-Aktivitäten $A_{\texttt{throw}}$ lösen bestimmte Fehlerereignisse aus. Das auszulösende Fehlerereignis kann einem Throw mittels $\text{throws} : A_{\texttt{throw}} \rightarrow E^{fault}$ zugewiesen werden.

3.2.4.8. Scope

In diesem Metamodell kann mittels Scopes S die Fehler- und Ereignisbehandlung für ihre direkten und indirekten Kindaktivitäten modelliert werden. Die Kompensation oder Terminierung von Kindaktivitäten wird hier nicht betrachtet. Es gilt $S = A_{\texttt{scope}} \cup P$. Es werden zwei Arten von Scopes unterschieden, Scope-Aktivitäten $A_{\texttt{scope}}$ und Prozessmodelle P. Prozessmodelle haben die gleichen Eigenschaften wie Scopes, außer dass sie keine Elternaktivität besitzen.

Zur Vereinfachung wird hier davon ausgegangen, dass Scope-Aktivitäten keine weiteren Scope-Aktivitäten als direkte oder indirekte Kindaktivitäten enthalten:

$$\forall s \in A_{\texttt{scope}} : \textsc{Descendants}(s) \cap A_{\texttt{scope}} = \emptyset$$

Ein Scope hat genau eine Hierarchiebeziehung, die den Scope mit seiner primären Kindaktivität assoziiert:

$$\forall s \in S : |\{hr \in HR \mid \pi_1(hr) = s \wedge \pi_2(hr) = \mathfrak{t}_{HR}^{ch}\}| = 1$$

Diese Kindaktivität eines Scopes kann über die Funktion $\textsc{child}_{pr} : S \rightarrow A$ bestimmt werden.

In BPEL können Scopes parallel zur Ausführung ihrer Kindaktivität Nachrichten- und Timer-Ereignisse verarbeiten, wobei das Eintreten eines Ereignisses zur Ausführung einer *Ereignisbehandlungsaktivität* führt. Diese Arbeit beschränkt

sich darauf, dass für einen Scope keine oder eine Ereignisbehandlungsaktivität für ein Timer-Ereignis definiert werden kann:

$$\forall s \in S : |\{hr \in HR \mid \pi_1(hr) = s \wedge \pi_2(hr) \in E^{timer}\}| \leq 1$$

Ein Scope besitzt außerdem einen oder mehrere Fault-Handler, die auf Fehlerereignisse E^{fault} reagieren, die von seinen Kindaktivitäten geworfen werden können:

$$\forall s \in S : |\{hr \in HR \mid \pi_1(hr) = s \wedge \pi_2(hr) \in E^{fault}\}| > 0$$

Dabei reagiert jeder Fault-Handler auf ein bestimmtes Fehlerereignis. Diesen können Fehlernamen über die Abbildung $\mathsf{faultName} : E^{fault} \rightarrow \mathcal{L} \cup \{\bot\}$ zugewiesen werden.

Für jeden Scope muss ein Standard-Fault-Handler $e^{faultStd} \in E^{faultStd}$ definiert werden, der auf alle Fehlerereignisse reagiert, die von den anderen Fault-Handlern nicht verarbeitet werden können, d.h. zu deren Fehlernamen kein passender Fault-Handler existiert:

$$E^{faultStd} := \{e^{fault} \in E^{fault} \mid \mathsf{faultName}(e^{fault}) = \bot\}$$

Mit der Abbildung $\mathsf{rethrow} : S \times E^{fault} \rightarrow \mathcal{B}$ kann verhindert werden, dass der Fault-Handler eines Scopes, einen Fehler zum Eltern-Scope weiter propagiert, falls eine seiner Fehlerbehandlungsaktivitäten einen Fehler auslöst.

Scopes werden in dieser Arbeit wie im Beispiel in Abbildung 3.4 dargestellt. Die Fault-Handler sind mit `Catch` bzw. `CatchAll` und Event-Handler mit `EH` gekennzeichnet. Der Scope s hat zum Beispiel eine primäre Kindaktivität a_{pch}, die Fehlerbehandlungsaktivität a_{fh} und eine Ereignisbehandlungsaktivität a_{eh}.

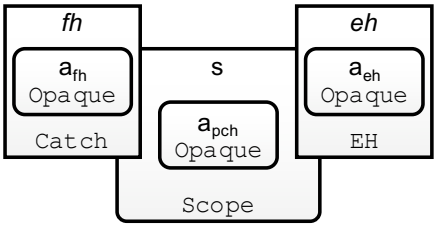

Abbildung 3.4.: Beispiel einer Scope-Aktivität mit Event- und Fault-Handler

3.2.4.9. Flow

Flow-Aktivitäten $A_{\texttt{flow}}$ dienen zur parallelen und synchronisierten Ausführung von Aktivitäten. Sie können daher zwei oder mehrere Kindaktivitäten enthalten. Die Anforderung, dass die Flow-Aktivität mindestens zwei Aktivitäten enthalten muss, ist nicht Bestandteil der BPEL-Spezifikation. Sie dient aber hier zur Reduzierung der Fallunterscheidungen und damit zur Verbesserung der Lesbarkeit der Algorithmen in den folgenden Kapiteln.

3.2.4.10. Kommunizierende Schleifen

Schleifen dienen in dieser Arbeit nur dazu, die Kommunikation eines Teilnehmers mit einer Menge von anderen Teilnehmern zu realisieren. Sie müssen deshalb immer eine Kommunikationsaktivität als direkte oder indirekte Kindaktivität besitzen. Kommunizierende Schleifen werden durch die Menge $A_{\texttt{loop}}$ repräsentiert:

$$A_{\texttt{loop}} := A_{\texttt{forEach}} \cup A_{\texttt{while}}$$

Es können zwei Arten von Schleifen mittels entsprechender Aktivitäten modelliert werden und zwar sequentielle ForEach- und While-Schleifen. BPEL bietet zusätzlich die Möglichkeit, parallele ForEach-Schleifen zu modellieren, diese werden hier aus Platzgründen nicht betrachtet.

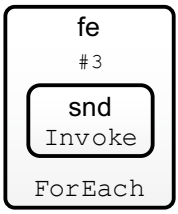

Abbildung 3.5.: Beispiel einer ForEach-Schleife

Sequentielle ForEach-Schleifen $A_{\texttt{forEach}}$ müssen während jeder Iteration mit einem anderen Teilnehmer kommunizieren. Die Teilnehmermenge, über die das ForEach iterieren muss, wird ihm über die Abbildung $\mathsf{partSet} : A_{\texttt{forEach}} \rightarrow \mathfrak{P}^{set}$ zugewiesen. Dabei ist keine Reihenfolge vorgegeben, in der ein ForEach über die Teilnehmer iteriert. Allerdings kann die maximale Anzahl der Iterationen eines ForEachs durch die Teilnehmermenge bestimmt werden. Abbildung 3.5 zeigt die grafische Notation einer ForEach-Schleife. Die Teilnehmermenge oder die Anzahl der Teilnehmer in der Teilnehmermenge, über die das ForEach iteriert, wird hinter dem Rautenzeichen angegeben.

Befindet sich ein Invoke *snd* oder ein Empfangskonstrukt co_{rcv} innerhalb einer ForEach-Schleife $fe \in A_{\texttt{forEach}}$, muss *snd* bzw. co_{rcv} während jeder Iteration von *fe* mit einem anderen Teilnehmer kommunizieren. Das impliziert auch, dass die Teilnehmermenge, über die das ForEach iteriert, der entsprechen muss, die dem Invoke bzw. dem Empfangskonstrukt in ihrer jeweiligen Nachrichtenkante zugeordnet ist (siehe Abschnitt 3.1.2).

While-Schleifen $A_{\texttt{while}}$ iterieren sequentiell über den Prozessmodellgraph im Schleifenkörper, so lange die Schleifenbedingung wahr ist. Die Schleifenbedingung kann mit der Abbildung $\mathsf{cond}_{\texttt{while}} : A_{\texttt{while}} \rightarrow Ex_{\mathcal{B}}$ zugewiesen werden. Beim While muss die Teilnehmermenge nicht vor seiner Ausführung bekannt sein. Im Gegensatz zu ForEach-Schleifen kann ein While daher auch in unterschiedlichen Iterationen mit ein und demselben Teilnehmer kommunizieren. Für bestimmte While-Schleifen kann, abhängig von deren

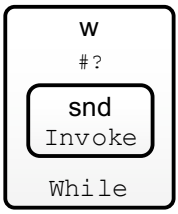

Abbildung 3.6.: Beispiel einer While-Schleife

Schleifenbedingung und -körper, deren maximale Anzahl an Iterationen zur Entwurfszeit mittels der in Abschnitt 6.2 beschriebenen Analysetechniken bestimmt werden. Das ist jedoch nicht für alle Whiles möglich, wie zum Beispiel für das While in Abbildung 3.6. Dies ist durch das Fragezeichen gekennzeichnet, andernfalls würde dort die maximale Iterationsanzahl dargestellt sein.

Zur Vereinfachung der Konsolidierungsalgorithmen und der Verifikation unterliegen kommunizierende ForEach- und While-Schleifen dabei den folgenden Restriktionen:

(I) Eine Verschachtlung der Schleifen ist nicht erlaubt:

$$\forall a_{loop} \in A_{\texttt{loop}} : A_{descLp} \cap A_{\texttt{loop}} = \emptyset \text{ mit } A_{descLp} = \textsc{Descendants}(a_{loop})$$

(II) Der Schleifenkörper darf nur genau eine Kindaktivität enthalten, d.h. es gilt: $\forall a_{loop} \in A_{\texttt{loop}} : |\textsc{Children}(a_{loop})| = 1$

(III) Die Kommunikationsaktivität muss im regulären Kontrollfluss in jeder Iteration ausgeführt werden, sie darf sich also nicht auf einem parallelen Pfad befinden:

$$\forall a_{com} \in (A_{descLp} \cap A_{com}) :$$
$$\textsc{Predecessors}_{all}(a_{com}) \cup \textsc{Successors}_{all}(a_{com}) \cup \{a_{com}\} = A_{descLp}$$

(IV) Der Schleifenkörper darf nur genau eine Sende- oder eine Empfangsaktivität enthalten, d.h. $\forall a_{loop} \in A_{\texttt{loop}} : |A_{descLp} \cap A_{com}| = 1$

3.3. Fragmente

Die Interaktionen einer Choreographie und die Schritte zu deren Konsolidierung werden in Kapitel 5 abstrakt anhand von Choreographiefragmenten erläutert. Entsprechend der Definition von Weiß et al. in [WAHK15] ist ein Choreographiefragment $c^F \in C^F$ eine Choreographie, die unterspezifizierte Nachrichtenkanten oder Prozessmodellfragmente enthält.

$$C^F := \{c_i \in C \mid \exists p \in \pi_4(c_i) : p \in P^F \vee \exists ml \in \pi_3(c_i) : ml \in ML^F\}$$

Bei Prozessmodellfragmenten $P^F \subset P$ handelt es sich um Prozessmodelle, deren Hierarchiebeziehungen oder Kontrollflusskanten unterspezifiziert sind [ELS+10; SKK+11]:

$$P^F := \{p \in P \mid ((\pi_2(p) \cup \pi_3(p)) \cap (HR^F \cup L^F)) \neq \emptyset\}$$

Bei unterspezifizierten Hierarchiebeziehungen $HR^F \subset HR$ ist entweder die Eltern- oder die Kindaktivität nicht angegeben:

$$HR^F := \{hr \in HR \mid \pi_1(hr) = \bot \dot{\vee} \pi_3(hr) = \bot\}^1$$

Entsprechend dazu ist bei unterspezifizierten Kontrollflusskanten $L^F \subset L$ entweder die Quell- oder Zielaktivität nicht angegeben:

$$L^F := \{l \in L \mid \pi_1(l) = \bot \dot{\vee} \pi_2(l) = \bot\}$$

[1] Das Symbol $\dot{\vee}$ ist die logische Kontravalenz.

3.4. Konversationen

Konversationen $\mathcal{K}$ sind die Laufzeitinstanzen von Choreographien [WAHK17]. Die von einer Choreographie erstellten Konversationen werden ihr über die Abbildung $\mathsf{conversations} : C \to \wp(\mathcal{K})$ zugewiesen.

Eine Konversation besteht aus miteinander kommunizierenden Teilnehmern $\mathfrak{P}$, den Instanzen der die Teilnehmer implementierenden Prozessmodelle und deren Elemente. Formal wird eine Konversation daher durch das folgende Tupel repräsentiert:

$$K := (\mathfrak{P}, P^I, A^I, L^I)$$

Die Menge P^I repräsentiert die Prozess-, A^I die Aktivitäts- und L^I die Kontrollflusskanteninstanzen. Jede dieser Instanzen befindet sich zu jedem Zeitpunkt in einem bestimmten Zustand. Auf die Formalisierung der Instanzen der anderen Prozessmodellelemente, wie beispielsweise Variablen, wird verzichtet, da diese für die weitere Betrachtungen in dieser Arbeit nicht relevant sind.

3.4.1. Prozessinstanz

Die Menge P^I repräsentiert die Instanzen von Prozessmodellen bzw. Verhaltensbeschreibungen. Die Ausführung eines Teilnehmers entspricht deshalb der Ausführung einer Prozessinstanz. Da zur Verifikation der Konsolidierung ausschließlich über die Zustände von Aktivitätsinstanzen argumentiert wird, wird im Rahmen dieser Arbeit nicht weiter auf Prozessinstanzen eingegangen.

3.4.2. Aktivitätsinstanz

Einer Aktivität werden über die Abbildung $\mathsf{inst}_A : A \to \wp(A^I)$ ihre Aktivitätsinstanzen zugewiesen. In dieser Arbeit werden die Instanzen von Aktivitäten

mit einem hochgestellten I gekennzeichnet. Die Instanz der Aktivität a_i wird so zum Beispiel durch a_i^I repräsentiert, d.h es gilt $a_i^I \in \mathsf{inst}_A(a_i)$.

Zur Laufzeit können Aktivitätsinstanzen die Zustände Υ in einer bestimmten Reihenfolge durchlaufen. Auf die Reihenfolge und die Semantik der Zustände wird in Abschnitt 4.1 eingegangen. Für Aktivitäten gilt $\Upsilon = \Upsilon_{reg} \cup \Upsilon_{error}$, wobei die Menge Υ_{reg} die *regulären Zustände* und die Menge Υ_{error} die *Fehlerzustände* repräsentiert:

$$\Upsilon_{reg} := \{initial, executing, completed, dead\}$$

$$\Upsilon_{error} := \{aborted, terminated, faulted\}$$

Der Startzustand jeder Aktivitätsinstanz ist *initial*. Die Menge $\Upsilon_{final} \subset \Upsilon$ umfasst die Endzustände einer Aktivitätsinstanz:

$$\Upsilon_{final} := \Upsilon \setminus \{initial, executing\}$$

3.4.3. Kontrollflusskanteninstanz

Die Zuweisung von Kontrollflusskanteninstanzen zu ihren Kontrollflusskanten erfolgt über die Abbildung $\mathsf{inst}_L : L \to \wp(L^I)$. Eine Kontrollflusskanteninstanz befindet sich zu jedem Zeitpunkt abhängig von der Instanz der Quellaktivität und der spezifizierten Transitionsbedingung in einem der folgenden Zustände:

$$\Lambda := \{undef., activated, deactivated\}$$

Der Status *undef.* impliziert, dass die Instanz der Quellaktivität noch keinen Endzustand erreicht hat und somit die in der Kontrollflusskante spezifizierte

Transitionsbedingung noch nicht evaluiert werden konnte. Wenn die Instanz der Quellaktivität erfolgreich ausgeführt wurde, kann die Transitionsbedingung ihrer ausgehenden Kanten ausgewertet werden. Evaluiert diese zu `true`, wird die Kontrollflusskanteninstanz aktiviert, d.h. sie geht in den Status *activated*. Erreicht die Quellaktivität den Zustand *dead* oder evaluiert die Transitionsbedingung zu `false`, wird die Kontrollflusskanteninstanz deaktiviert und geht somit in den Status *deactivated*. Der Endzustand einer Kontrollflusskanteninstanz kann zur Laufzeit eines Prozesses nur einmal gesetzt werden, d.h. es ist zum Beispiel nicht möglich, dass sie von *activated* in *deactivated* wechselt.

3.5. Zusammenfassung

Dieses Kapitel bildet die formale Grundlage für die Diskussion der Konsolidierungsoperation. Dazu wurde ein Metamodell für Choreographien definiert, das sich an der Choreographiesprache BPEL4Chor orientiert. Um das Interaktionsverhalten der Teilnehmer von Choreographien beschreiben zu können, wurde als Teil des Metamodells ein auf BPEL basierendes Prozessmetamodell definiert, das sich auf Kommunikationsaspekte fokussiert. Neben den Konstrukten zur Modellierung von Choreographien und Prozessmodellen wurde auch ein formales Modell zur Beschreibung ihrer Instanzen definiert, deren Verhalten im folgenden Kapitel erläutert wird.

Kapitel 4

Choreographie-verhaltensbeschreibung mit Aktivitätszustands-beziehungen

Aktivitätsinstanzen durchlaufen während der Ausführung einer Choreographie verschiedene Zustände [LR00; Wes12]. Die Reihenfolge, in der die Instanzen die Zustände durchlaufen können, hängt dabei von den Kontrollflussbeziehungen zwischen den Aktivitäten ab. In den Choreographiefragmenten c_1^F und c_2^F in Abbildung 4.1 können aufgrund der Kontrollflusskanten in jeder Konversation die Instanzen des Scopes *s* erst ausgeführt werden, also in den Zustand *executing* gehen, nachdem die Instanz von a_1 erfolgreich ausgeführt wurde. Die Instanzen der Fehlerbehandlungsaktivität a_{fh}, die sich im Fault Handler von *s* befindet, können wiederum nur ausgeführt wer-

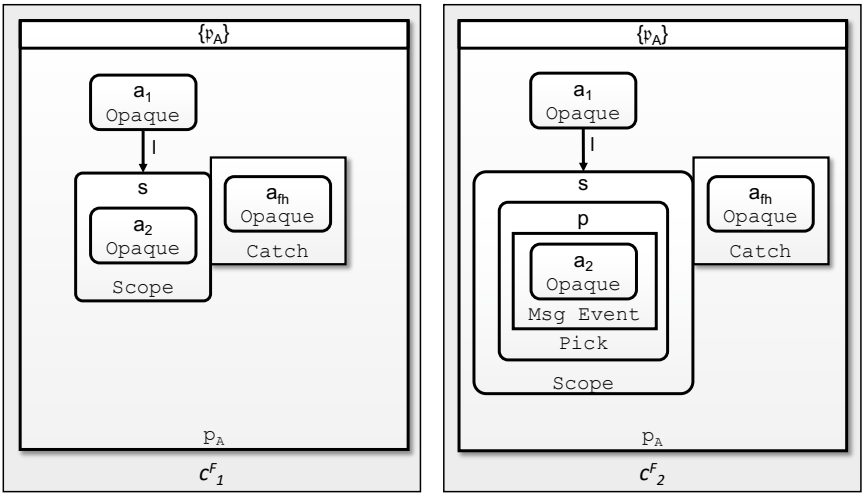

Abbildung 4.1.: Zwei Choreographiefragmente c_1^F und c_2^F

den, nachdem die Instanzen von a_2 bzw. von p einen Fehlerzustand erreicht haben.

Diese durch den Kontrollfluss implizierte Ordnung zwischen den Zuständen von zwei Aktivitäten wird in dieser Arbeit als *Aktivitätszustandsbeziehung* bezeichnet. Sie gilt also in jeder Konversation zwischen den beiden Zuständen der Instanzen der Aktivitäten. Mittels Aktivitätszustandsbeziehungen kann, wie in diesem Kapitel gezeigt wird, das Verhalten von Choreographien beschrieben und verglichen werden. Dazu wird geprüft, inwieweit deren identische Aktivitäten dieselben Aktivitätszustandsbeziehungen haben, auch wenn die Choreographien unterschiedlich modelliert sind. Im Choreographiefragment c_2^F gilt zum Beispiel ebenfalls, dass a_2 nur ausgeführt werden kann, nachdem a_1 erfolgreich beendet wurde, auch wenn sich a_2 jetzt zusätzlich in einer Pick-Aktivität befindet. Die Aktivitäten a_1 und a_2 haben also bezüglich ihres Ausführungszustands *executing* in c_1^F und c_2^F die gleiche Aktivitätszustandsbeziehung.

Der Vergleich von Aktivitätszustandsbeziehungen wird im weiteren Verlauf der Arbeit dazu genutzt, um die Korrektheit der Konsolidierungsschritte zu verifizieren und um damit eine Aussage treffen zu können, ob das Verhalten des konsolidierten Prozessmodells dem der Choreographie entspricht, aus der es erstellt wurde. Die für den Vergleich notwendigen Aktivitätszustandsbeziehungen müssen über den Kontroll- und Nachrichtenfluss ermittelt werden. Daher wird in Abschnitt 4.2 die operationale Semantik der in Kapitel 3 eingeführten Kontroll- bzw. Nachrichtenflusskonstrukte diskutiert. Basierend auf der Semantik des jeweiligen Konstrukts werden Axiome aufgestellt, die die Aktivitätszustandsbeziehungen zwischen den durch die Konstrukte direkt miteinander verbundenen Aktivitäten definieren. So können zum Beispiel die Beziehungen zwischen Aktivität a_1 und s mit den in Abschnitt 4.2.9 definierten Axiomen für die sequentielle Ausführung von Aktivitäten bestimmt werden. Die Beziehungen zwischen Scope s und seinen Kindaktivitäten a_2 bzw. p können ebenfalls mit entsprechenden Axiomen ermittelt werden. Die transitive Anwendung der Axiome ermöglicht es, Beziehungen zwischen Aktivitäten zu ermitteln, die indirekte Kontrollflussbeziehungen haben, wie zum Beispiel die Aktivitäten a_1 und a_2.

Bevor in Abschnitt 4.2 Axiome für die verschiedenen Konstrukte des Metamodells definiert werden, werden im folgenden Abschnitt die formalen Grundlagen zur Beschreibung von Aktivitätszustandsbeziehungen diskutiert. Auf den Vergleich des Verhaltens von Choreographien bzw. deren Fragmenten wird in Abschnitt 4.3 eingegangen.

4.1. Aktivitätszustands-Traces, -Historien, -Beziehungen & -Profile

In Abschnitt 3.4.2 wurde die Menge Υ der Zustände definiert, die die Instanzen von Aktivitäten durchlaufen können. Wechselt eine Instanz zu einem Zeitpunkt t in einen bestimmten Zustand, wird dies als *Zustandstransitionsereignis* bezeichnet.

Definition 4.1 (Zustandstransitionsereignis)
Zustandstransitionsereignisse werden durch die Menge $E^{\Upsilon} \subseteq A^I \times \Upsilon \times \mathbb{T} \times \mathcal{K}$ repräsentiert. Dabei identifiziert jedes Element $(a^I, \upsilon, t, K) \in E^{\Upsilon}$ die Transition einer zur Konversation K gehörenden Aktivitätsinstanz a^I in einen Zustand υ zu einem bestimmten Zeitpunkt t. Zu jedem Zeitpunkt $t \in \mathbb{T}$ kann während der Ausführung einer Konversation genau eine Zustandstransition durchgeführt werden:

$$\forall e_i^{\Upsilon}, e_j^{\Upsilon} \in E^{\Upsilon} : \pi_3(e_i^{\Upsilon}) = \pi_3(e_j^{\Upsilon}) \Rightarrow e_i^{\Upsilon} = e_j^{\Upsilon}$$

Der *Aktivitätszustands-Trace* (kurz Trace) einer Konversation ist eine gültige Sequenz von Zustandstransitionsereignissen [vdAtHW03]. Die gültigen Sequenzen von Zustandstransitionsereignissen der Aktivitätsinstanzen einer Konversation werden, wie eingangs erwähnt, durch die operationale Semantik der Aktivitätstypen, durch ihren Kontrollflusskontext und den Nachrichtenfluss innerhalb der Choreographie bestimmt. Da zu jedem Zeitpunkt nur ein Zustandstransitionsereignis stattfinden darf, besteht eine Ordnung bzw. Reihenfolge der Ereignisse über den Zeitpunkt ihres Auftretens. Ein *vollständiger Trace* ist wiederum der Trace einer beendeten Konversation, in der keine Zustandstransitionsereignisse mehr auftreten können, da jede Instanz ihren Endzustand erreicht hat. Alle möglichen vollständigen Traces die von den Konversationen einer Choreographie erzeugt werden können, werden dabei als *Historie* der Choreographie bezeichnet.

Definition 4.2 (Trace, vollständiger Trace & Historie)
Die Traces $T \subseteq \wp(E^{\Upsilon})$ einer Konversation umfassen die Zustandstransitionen, die während der Ausführung von Konversationen entstehen können.

Die Menge der vollständigen Traces wird durch $\hat{T} \subset T$ repräsentiert. Die Menge der vollständigen Traces $\hat{T}_K$ einer Konversation $K \in \mathcal{K}$ ist wie folgt definiert:

$$\hat{T}_K := \{\hat{\tau} \in \hat{T} \mid \forall e^{\Upsilon} \in \hat{\tau} : \pi_4(e^{\Upsilon}) = K\}$$

Jeder vollständige Trace $\hat{\tau}_K \in \hat{T}_K$ einer Konversation muss dabei die folgenden Eigenschaften erfüllen:

1. *Der Trace umfasst die Zustandstransitionen aller Aktivitätsinstanzen der Konversation:*
$$(\bigcup_{e^{\Upsilon} \in \hat{\tau}_K} \pi_1(e^{\Upsilon})) = \pi_3(K)$$

2. *Alle Aktivitätsinstanzen müssen sich in einem Endzustand befinden:*
$$\forall a^I \in \pi_3(K) \exists! e^{\Upsilon} \in \hat{\tau}_K : \pi_1(e^{\Upsilon}) = a^I \wedge \pi_2(e^{\Upsilon}) \in \Upsilon_{final}$$

3. *Es existiert genau ein Zustandstransitionsereignis, das zum Startzeitpunkt $t = 0$ der Konversation ausgeführt wird:*
$$\exists! e^{\Upsilon} \in \hat{\tau}_K : \pi_3(e^{\Upsilon}) = 0$$

Die Abbildung $\mathsf{history_c} : C \rightarrow \wp(\hat{T})$ weist einer Choreographie ihre Historie, d.h. die Menge alle möglichen vollständigen Traces zu, die die Konversationen der Choreographie erzeugen können:

$$\mathsf{history_c}(c) = \{\hat{\tau} \in \hat{T} \mid \forall e^{\Upsilon} \in \hat{\tau} : \pi_4(e^{\Upsilon}) \in \mathsf{conversations}(c)\}$$

Die Abbildung $\mathsf{history_p} : P \rightarrow \wp(\hat{T})$ weist einem Prozessmodell alle vollständigen Traces zu, die dessen Aktivitätsinstanzen erzeugen können:

$$\mathsf{history_p}(p) = \{\hat{\tau} \in \hat{T} \mid \forall e^{\Upsilon} \in \hat{\tau} : \pi_1(e^{\Upsilon}) \in \bigcup_{a_p \in \pi_1(p)} \mathsf{inst}_A(a_p)\}$$

Die Abbildung $\mathsf{history_a} : A \rightarrow \wp(\hat{T})$ weist einer Aktivität a alle vollständigen Traces zu, die deren Instanzen und die Instanzen der Nachfahren von a erzeugen können. Die Aktivität a und ihre Nachfahren werden durch die Menge $A_{clan} = \textsc{Descendants}(a) \cup \{a\}$ repräsentiert.

$$\mathsf{history}_{\mathsf{a}}(a) = \{\hat{\tau} \in \hat{T} \mid \forall e^{\Upsilon} \in \hat{\tau} : \pi_1(e^{\Upsilon}) \in \bigcup_{a_{clan} \in A_{clan}} \mathsf{inst}_A(a_{clan})\}$$

Es dürfen über die Abbildung $\mathsf{history}_{\mathsf{a}}$ *der Aktivität a nur die vollständigen Traces zugewiesen werden, die aufgrund der Kontrollflussbeziehungen zwischen den Aktivitäten in* A_{clan} *gelten. Das heißt Traces, die aufgrund von Kontrollflussbeziehungen zu anderen Aktivitäten entstehen, die nicht Teil von* A_{clan} *sind, werden über die Abbildung der Aktivität a nicht zugewiesen.*

Die Funktion $\textsc{events} : A \times \Upsilon \times T \rightarrow \wp(E^{\Upsilon})$ gibt die Zustandstransitionsereignisse einer Aktivität a aus einem Trace τ zurück, in denen ihre Instanzen den Zustand υ erreicht haben:

$$\textsc{events}(a, \upsilon, \tau) = \{e^{\Upsilon} \in E^{\Upsilon} \mid e^{\Upsilon} \in \tau \wedge \pi_1(e^{\Upsilon}) \in \mathsf{inst}_A(a) \wedge \upsilon = \pi_2(e^{\Upsilon})\}$$

Die zulässigen Traces bzw. die zulässige Historie einer Choreographie kann deklarativ über eine Menge von Aktivitätszustandsbeziehungen beschrieben werden. Informell definiert eine Aktivitätszustandsbeziehung eine Ordnung zwischen zwei Aktivitätszuständen, die in jeder Historie bestehen muss. So wird zum Beispiel in Abschnitt 4.2.9 die Beziehung definiert, dass für zwei über eine Kontrollflusskante verbundene Aktivitäten, die Zielaktivität a_{trg} der Kante erst dann ausgeführt werden darf, d.h. in den Zustand *executing* wechseln kann, nachdem die Quellaktivität a_{src} erfolgreich beendet wurde, also in den Zustand *completed* gewechselt ist. Daraus folgt, dass in jedem Trace die Zustandstransition einer Instanz der Zielaktivität in den Zustand *executing* zeitlich erst nach der Transition einer Instanz der Quellaktivität in den Zustand *completed* auftreten darf. Die Beziehung impliziert allerdings nicht, dass die Zielaktivität in den Zustand *completed* gehen muss. Sie könnte alternativ deaktiviert werden, also in den Zustand *dead* gehen. Dass in einem Trace der Zustand *dead* der Zielaktivität nach dem Zustand *completed* der Quellaktivität auftreten darf und dass beide von ihr im selben Trace nicht erreicht werden können, wird durch weitere Beziehungen beschrieben.

Formal kann eine Aktivitätszustandsbeziehung mittels der Abbildung ω_Υ festgelegt werden:

$$\omega_\Upsilon : A \times \Upsilon \times A \times \Upsilon \to \Gamma$$

Die Abbildung $\omega_\Upsilon(a_l, v_l, a_r, v_r)$ kann zwischen dem Zustand v_l der Aktivität a_l und dem Zustand v_r der Aktivität a_r eine Aktivitätszustandsbeziehung $\Gamma := \{\rightarrow, \leftarrow, \|, \oplus\}$ zuweisen. Die Beziehungen sind wie folgt definiert:

$\rightarrow$ **(strenge Ordnung)**: Erreicht in einer Konversation eine Instanz der Aktivität a_l den Zustand v_l und eine Instanz der Aktivität a_r den Zustand v_r, muss die Instanz von a_l in den Zustand v_l gehen, bevor a_r in den Zustand v_r geht:

$$\forall \hat{\tau} \in \mathsf{history_c}(c) \forall e_l^\Upsilon \in \textsc{events}(a_l, v_l, \hat{\tau}) \forall e_r^\Upsilon \in \textsc{events}(a_r, v_r, \hat{\tau}) : \pi_3(e_l^\Upsilon) < \pi_3(e_r^\Upsilon)$$

Diese Zustandsbeziehung definiert nur eine temporale Ordnung zwischen den Zuständen aber keine Kausalität. Sie impliziert also nicht, dass, wenn die eine Instanz den Zustand v_l erreicht, die andere Instanz den Zustand v_r erreichen muss.

$\oplus$ **(Exklusivität)**: Erreichen die Instanzen von Aktivität a_l den Zustand v_l, können die Instanzen von Aktivität a_r den Zustand v_r in der gleichen Konversation nicht mehr erreichen:

$$\begin{aligned} &\forall \tau \in \mathsf{history_c}(c) : \\ &\quad \exists e_l^\Upsilon \in \textsc{events}(a_l, v_l, \tau) \Rightarrow \nexists e_r^\Upsilon \in \textsc{events}(a_r, v_r, \tau) \\ &\quad \wedge \exists e_r^\Upsilon \in \textsc{events}(a_r, v_r, \tau) \Rightarrow \nexists e_l^\Upsilon \in \textsc{events}(a_l, v_l, \tau) \end{aligned}$$

$\|$ **(keine Ordnung)**: Die Instanzen von Aktivität a_l erreichen den Zustand v_l vor oder nachdem die Instanzen von Aktivität a_r den Zustand v_r erreicht haben:

$$\begin{aligned} &\exists \hat{\tau}_i \in \mathsf{history_c}(c) \exists \hat{\tau}_j \in (\mathsf{history_c}(c) \setminus \{\hat{\tau}_i\}) : \\ &\quad (\exists e_{li}^\Upsilon \in \textsc{events}(a_l, v_l, \hat{\tau}_i)\ \exists e_{ri}^\Upsilon \in \textsc{events}(a_r, v_r, \hat{\tau}_j) : \pi_3(e_{li}^\Upsilon) < \pi_3(e_{ri}^\Upsilon)) \\ &\quad \wedge (\exists e_{lj}^\Upsilon \in \textsc{events}(a_l, v_l, \hat{\tau}_i)\ \exists e_{rj}^\Upsilon \in \textsc{events}(a_r, v_r, \hat{\tau}_j) : \pi_3(e_{lj}^\Upsilon) > \pi_3(e_{rj}^\Upsilon)) \end{aligned}$$

Die Beziehung $\leftarrow$ ist die inverse Beziehung der „strengen Ordnung“ :

$$\omega_\Upsilon(a_r, \upsilon_r, a_l, \upsilon_l) = \rightarrow \iff \omega_\Upsilon(a_l, \upsilon_l, a_r, \upsilon_r) = \leftarrow$$

Die Aktivitätszustandsbeziehungen zwischen allen Zuständen von zwei Aktivitäten bzw. deren Instanzen können mittels eines *Aktivitätszustandsprofils* (kurz Profil) definiert werden.

Definition 4.3 (Aktivitätszustandsprofil)
Ein Aktivitätszustandsprofil (kurz Profil) $\mathsf{profile} : A \times A \rightarrow \bigcup \omega_\Upsilon$ *weist zwei Aktivitäten* $a_l, a_r \in A$ *die Aktivitätszustandsbeziehungen zu, die paarweise zwischen allen ihren Zuständen gelten:*

$$\mathsf{profile}(a_l, a_r) = \bigcup_{\upsilon_l, \upsilon_r \in \Upsilon} \omega_\Upsilon(a_l, \upsilon_l, a_r, \upsilon_r)$$

4.2. Aktivitätszustandsaxiome

Ziel dieses Abschnitts ist es, Zustandstransitionsprofile für alle Aktivitäten des in dieser Arbeit verwendeten Metamodells zu erstellen, die eine direkte Kontroll- oder Nachrichtenflussbeziehung zueinander besitzen. Wie in Abschnitt 3.2 erläutert, haben zwei Aktivitäten eine direkte Kontrollflussbeziehung, wenn sie über eine Kontrollflusskante oder eine Hierarchiebeziehung miteinander verbunden sind. Nachrichtenkanten werden hier zusätzlich berücksichtigt, da sich der Empfang einer Nachricht von einer Sendeaktivität ebenfalls direkt auf den Zustand der Empfangsaktivität auswirkt.

Um die Profile zu ermitteln, werden für die Aktivitäten mit einer direkten Kontroll- bzw. Nachrichtenflussbeziehung *Aktivitätszustandssaxiome* (kurz Axiome) definiert. Die Axiome sind Aktivitätszustandsbeziehungen, die auf Basis der operationalen Semantik der BPEL-Konstrukte und des von Kopp et al. in [KHK+11] präsentierten Zustandstransitionsmodell für BPEL-Aktivitäten abgeleitet werden.

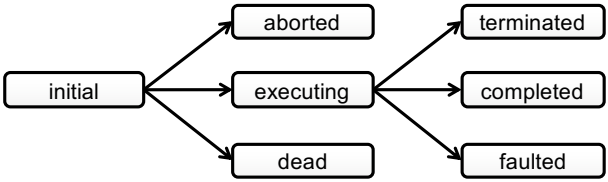

Abbildung 4.2.: Zustandstransitionsmodell Aktivität

Initial sind zwischen den Zuständen von Aktivitäten alle Beziehungen möglich, d.h. es gelten zwischen den Zuständen die Beziehungen $\{\|, \oplus\}$. Durch Anwendung der Axiome aufgrund der expliziten Kontrollflussbeziehungen werden diese Beziehungen eingeschränkt. Die Axiome in Abschnitt 4.2.5 legen zum Beispiel fest, welche Zustandsbeziehungen die Instanzen der Kindaktivitäten haben, die sich in derselben strukturierten Aktivität befinden. Darauf aufbauend schränken die Axiome die Zustandsbeziehungen weiter für Aktivitäten ein, die sich in derselben strukturierten Aktivität befinden und sequentiell ausgeführt werden.

In Abschnitt 4.2.1 und Abschnitt 4.2.2 sind die grundlegenden Axiome beschrieben. Sie definieren die temporalen Beziehungen zwischen den Zuständen einer Aktivität und die, die zwischen allen Aktivitäten einer Choreographie gelten. Darauf aufbauend werden die Axiome für die Aktivitäten des Metamodells erstellt. van der Aalst et al. [vdAtHKB03] beschreiben verschiedene Workflow-Muster, die mittels graph- oder blockbasiertem Kontrollfluss modelliert werden können. Diese Muster implizieren ebenfalls Zustandsbeziehungen zwischen den beteiligten Aktivitäten. Für die Muster, die für die Konsolidierung in Kapitel 5 benötigt werden, werden in diesem Abschnitt ebenfalls entsprechende Axiome definiert.

4.2.1. Aktivität

Das auf der Basis der BPEL-Spezifikation von Kopp et al. erstellte Zustandstransitionsmodell für Aktivitäten ist in Abbildung 4.2 dargestellt. Es definiert,

in welcher Reihenfolge die Zustände durchlaufen werden können.

Formal sind dort folgende Zustandstransitionen $\Delta \subset \Upsilon \times \Upsilon$ erlaubt[1]:

$$\Delta := \{(initial, aborted), (initial, dead), (initial, executing), (executing, faulted), (executing, terminated), (executing, completed)\}$$

In einer Transition $\delta \in \Delta$ wird der Zustand $\pi_2(\delta)$ als direkter Nachfolgezustand von $\pi_1(\delta)$ bezeichnet. Analog ist $\pi_1(\delta)$ der direkte Vorgängerzustand von $\pi_2(\delta)$.

Aus den Transitionen können die folgenden Zustandstransitionspfade $\Psi \subset \Upsilon_1 \times \ldots \times \Upsilon_n$ abgeleitet werden:

$$\begin{aligned}\Psi := &\{(initial, aborted)\} \\ &\cup \{(initial, dead)\} \\ &\cup \{(initial, executing, completed)\} \\ &\cup \{(initial, executing, terminated)\} \\ &\cup \{(initial, executing, faulted)\}\end{aligned}$$

Der Startzustand jeder Aktivitätsinstanz ist *initial*. Die möglichen Endzustände einer Aktivitätsinstanz sind *dead*, *aborted*, *terminated*, *faulted* und *completed*. Der Zustandstransitionspfad $(initial, aborted)$ impliziert, dass eine Instanz aufgrund eines Fehlers in einer anderen Aktivitätsinstanz noch vor ihrer Ausführung terminiert wurde. Evaluiert die Eintrittsbedingung einer Instanz zu `false` oder wird die Instanz durch Dead-Path-Elimination [LA94; OAS07b] deaktiviert, durchläuft sie den Pfad $(initial, dead)$. Eine Instanz durchläuft den Pfad $(initial, executing, completed)$, wenn ihre Eintrittsbedingung zu `true` evaluiert, ihre Implementierung im Zustand *executing* erfolgreich ausgeführt wurde und sie somit den Zustand *completed* erreicht hat.

[1] Für Throw- und Empty-Aktivitäten werden die Zustandstransitionen in Abschnitt 4.2.3 bzw. 4.2.4 eingeschränkt. Für Scopes gilt, wie in Abschnitt 4.2.6 erläutert, ebenfalls dieses Transitionsmodell.

Der Pfad (*initial*, *executing*, *faulted*) impliziert, dass ihre Implementierung während der Ausführung einen Fehler ausgelöst hat und sie somit in den Zustand *faulted* gewechselt ist. Wird die Aktivitätsinstanz während der Ausführung von ihrer Eltern-Aktivität terminiert, wird (*initial*, *executing*, *terminated*) durchlaufen. Aus der Reihenfolge der Zustandstransitionen in den Pfaden können die Axiome *Act1* und *Act2* abgeleitet werden.

Axiom *Act1* definiert für jede Instanz einer Aktivität $a \in A$ die Ordnung zwischen Vorgänger- und direkten Nachfolgezustand. In jedem gültigen Aktivitätszustands-Trace muss eine Instanz den Nachfolgezustand nach dem Vorgängerzustand erreichen:

$$\forall \delta \in \Delta : \omega_{\Upsilon}(a, \pi_1(\delta), a, \pi_2(\delta)) \coloneqq \rightarrow \tag{Act1}$$

Das Axiom *Act2* definiert für alle Aktivitätsinstanzen die Exklusivität von sich nicht auf demselben Zustandstransitionspfad befindlichen Aktivitätszuständen υ_j und υ_k. Es existiert also kein Aktivitätszustands-Trace, indem dieselbe Instanz diese Zustände erreichen kann.

$$\begin{gathered} \forall \psi \in \Psi \forall \upsilon_j, \upsilon_k \in \Upsilon : \\ |(\bigcup_{i=1}^{|\psi|} \pi_i(\psi)) \cap \{\upsilon_j, \upsilon_k\}| < 2 \Leftrightarrow \omega_{\Upsilon}(a, \upsilon_j, a, \upsilon_k) \coloneqq \oplus \end{gathered} \tag{Act2}$$

Aus den Axiomen kann das in Abbildung 4.3 dargestellte Profil abgeleitet werden, das die Beziehungen zwischen allen Zuständen einer Aktivität darstellt. Alle anderen Beziehungen unterhalb der Hauptdiagonalen lassen sich durch die inverse Operation ableiten. Da die Zustände derselben Aktivität nicht zueinander in Beziehung stehen können, bleibt die Hauptdiagonale unbesetzt.

		a						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a	INITIAL		→ Act1	→ Act1	→ Act1	→ Act1	→ Act1	→ Act1
	DEAD	← Act1		⊕ Act2	⊕ Act2	⊕ Act2	⊕ Act2	⊕ Act2
	ABORTED	← Act1	⊕ Act2		⊕ Act2	⊕ Act2	⊕ Act2	⊕ Act2
	EXECUTING	← Act1	⊕ Act2	⊕ Act2		→ Act1	→ Act1	→ Act1
	TERMINATED	← Act1	⊕ Act2	⊕ Act2	← Act1		⊕ Act2	⊕ Act2
	FAULTED	← Act1	⊕ Act2	⊕ Act2	← Act1	⊕ Act2		⊕ Act2
	COMPLETED	← Act1	⊕ Act2	⊕ Act2	← Act1	⊕ Act2	⊕ Act2	

Abbildung 4.3.: Zustandstransitionsprofil Aktivität

4.2.2. Choreographie

In dieser Arbeit wird davon ausgegangen, dass bei der Erzeugung einer Konversation alle in der Choreographie modellierten Aktivitäten instanziiert und die Instanzen in den Zustand *initial* gesetzt werden. Dabei werden separate Instanzen für jeden Teilnehmer erstellt. Bei der Erzeugung einer Konversation aus der Choreographie in Abbildung 4.4[1] werden zum Beispiel zwei Instanzen der Aktivität a_i aus Prozessmodell p_A erstellt und zwar eine für Teilnehmer $\wp_1$ und eine für Teilnehmer $\wp_2$. Von Aktivität a_j aus Prozessmodell p_B wird nur eine Instanz für Teilnehmer $\wp_3$ erstellt.

BPEL4Chor ist eine Beschreibungssprache für Choreographien, die keine operationale Semantik für die Aktivitäten der Choreographie bzw. die Aktivitätsinstanzen einer Konversation spezifiziert. Deshalb werden hier die folgenden Zustandsbeziehungen zwischen den Instanzen einer Konversation per Konvention festgelegt.

Zur Laufzeit müssen sich alle Aktivitätsinstanzen einer Konversation bzw. einer Prozessinstanz in einem bestimmten Zustand befinden. Sie werden

[1] Nachrichtenkanten sind für die Betrachtungen in diesem Abschnitt nicht relevant und wurden deshalb nicht dargestellt.

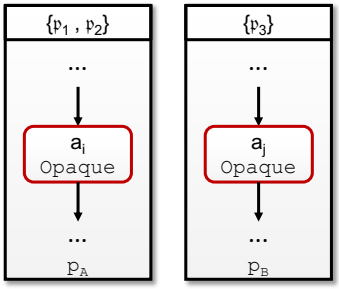

Abbildung 4.4.: Choreographiefragment mit drei Teilnehmern

daher bei der Erstellung einer Konversation oder der Instanziierung eines Prozessmodells zuerst alle in den Zustand *initial* gesetzt. Dabei ist keine Reihenfolge festgelegt, in der die Instanzen *initial* erreichen müssen:

$$\forall a_i, a_j \in \bigcup_{p \in \pi_4(c)} \pi_1(p) : \omega_\Upsilon(a_i, initial, a_j, initial) := \| \qquad (Chor1)$$

Die eigentliche Ausführung der Konversation kann erst beginnen, wenn sich alle Aktivitätsinstanzen im Zustand *initial* befinden, d.h. dass die Aktivitätsinstanzen erst in einen Nachfolgezustand von *initial* gehen können, nachdem alle Instanzen *initial* erreicht haben:

$$\begin{aligned} \forall a_i, a_j \in \bigcup_{p \in \pi_4(c)} \pi_1(p), \forall \upsilon \in (\Upsilon \setminus \{initial\}) : \\ \omega_\Upsilon(a_i, initial, a_j, \upsilon) := \rightarrow \end{aligned} \qquad (Chor2)$$

Das aus den Axiomen *Chor1* und *Chor2* resultierende Profil der Zustandsbeziehungen zwischen den Aktivitäten einer Choreographie ist in Abbildung 4.5 dargestellt. Da in diesem Abschnitt nur die Aktivitäten einer Choreographie betrachtet werden, ohne die Berücksichtigung ihrer Kontrollflussabhängigkeiten, gilt in dem Profil zwischen den Nachfolgezuständen von *initial* weiterhin die Relation ||. Es existieren also keine Einschränkungen bezüglich der Zu-

		a_j						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_i	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖	‖	‖	‖	‖	‖
	ABORTED	← Chor2	‖	‖	‖	‖	‖	‖
	EXECUTING	← Chor2	‖	‖	‖	‖	‖	‖
	TERMINATED	← Chor2	‖	‖	‖	‖	‖	‖
	FAULTED	← Chor2	‖	‖	‖	‖	‖	‖
	COMPLETED	← Chor2	‖	‖	‖	‖	‖	‖

Abbildung 4.5.: Zustandsprofil zwischen den Aktivitäten a_i und a_j einer Choreographie ohne Berücksichtigung ihrer Kontrollflussabhängigkeiten

stände, die die Aktivitätsinstanzen der Choreographie erreichen können und in welcher Reihenfolge sie diese erreichen.

4.2.3. Empty

Eine Empty-Aktivität $e \in A_{\texttt{empty}}$ führt zur Laufzeit keine Funktionalität aus. Sie dient hier nur als Synchronisationspunkt für Kontrollflusskanten. Da eine Empty-Instanz keine Funktionalität ausführt, kann sie keinen Fehler auslösen. Daher wird das Zustandstransitionsmodell für Empty-Instanzen so eingeschränkt, dass diese nie den Zustand *faulted* erreichen können:

$$\forall \upsilon \in \Upsilon : \omega_{\Upsilon}(e, \upsilon, e, faulted) := \oplus \qquad (Act_{Empty})$$

Das Axiom schränkt die Axiome *Act1* und *Act2* ein. Das aus den Axiomen resultierende Zustandsprofil der Empty-Aktivität ist in Abbildung 4.6 dargestellt. Folglich unterscheidet sich dieses Profil von dem der anderen Aktivitätstypen in Abbildung 4.3 bezüglich der vorletzten Zeile bzw. Spalte.

		e						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
e	INITIAL		→ Act1	→ Act1	→ Act1	→ Act1	⊕ Act_{Empty}	→ Act1
	DEAD	← Act1		⊕ Act2	⊕ Act2	⊕ Act2	⊕ Act_{Empty}	⊕ Act2
	ABORTED	← Act1	⊕ Act2		⊕ Act2	⊕ Act2	⊕ Act_{Empty}	⊕ Act2
	EXECUTING	← Act1	⊕ Act2	⊕ Act2		→ Act1	⊕ Act_{Empty}	→ Act1
	TERMINATED	← Act1	⊕ Act2	⊕ Act2	← Act1		⊕ Act_{Empty}	⊕ Act2
	FAULTED	⊕ Act_{Empty}	⊕ Act_{Empty}	⊕ Act_{Empty}	⊕ Act_{Empty}	⊕ Act_{Empty}		⊕ Act_{Empty}
	COMPLETED	← Act1	⊕ Act2	⊕ Act2	← Act1	⊕ Act2	⊕ Act_{Empty}	

Abbildung 4.6.: Zustandstransitionsprofil Empty-Aktivität *e*

4.2.4. Throw

Eine Throw-Aktivität $t \in A_{\texttt{throw}}$ löst bei ihrer Ausführung einen definierten Fehler aus (Abschnitt 3.2.4.7). Daher wird das Zustandstransitionsmodell einer Throw-Instanz so eingeschränkt, dass sie vom Zustand *executing* nicht den Zustand *completed* erreichen kann:

$$\forall \upsilon \in \Upsilon : \omega_{\Upsilon}(t, \upsilon, t, completed) := \oplus \qquad (Act_{Throw})$$

Intuitiver wäre es, wenn die Instanzen des Throws in den Zustand *completed* gesetzt werden würden, nachdem sie den Fehler ausgelöst haben. Dies hätte allerdings zur Folge, dass die Axiome der strukturierten Aktivität in Abschnitt 4.2.5 und des Scopes in Abschnitt 4.2.6 um Sonderfälle ergänzt werden müssten. Diese müssten berücksichtigen, dass ein Fehler in der Instanz der strukturierten Aktivität auch dann auftreten kann, wenn eine Throw-Instanz, die sich innerhalb dieser Aktivität befindet, den Zustand *completed* erreicht. Daher dient die Einschränkung nur zur Vereinfachung der Axiome. Das Zustandsprofil der Throw-Aktivität, das aus den Axiomen *Act1* und *Act2* sowie dem Axiom Act_{Throw} resultiert, ist in Abbildung 4.7 dargestellt.

		t						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
t	INITIAL		→ Act1	→ Act1	→ Act1	→ Act1	→ Act1	⊕ Act_{Throw}
	DEAD	← Act1		⊕ Act2	⊕ Act2	⊕ Act2	⊕ Act2	⊕ Act_{Throw}
	ABORTED	← Act1	⊕ Act2		⊕ Act2	⊕ Act2	⊕ Act2	⊕ Act_{Throw}
	EXECUTING	← Act1	⊕ Act2	⊕ Act2		→ Act1	→ Act1	⊕ Act_{Throw}
	TERMINATED	← Act1	⊕ Act2	⊕ Act2	← Act1		⊕ Act2	⊕ Act_{Throw}
	FAULTED	← Act1	⊕ Act2	⊕ Act2	← Act1	⊕ Act2		⊕ Act_{Throw}
	COMPLETED	⊕ Act_{Throw}	⊕ Act_{Throw}	⊕ Act_{Throw}	⊕ Act_{Throw}	⊕ Act_{Throw}	⊕ Act_{Throw}	

Abbildung 4.7.: Zustandsprofil Throw-Aktivität t

4.2.5. Strukturierte Aktivität

Eine strukturierte Aktivität $a_{\texttt{struc}} \in A_{\texttt{struc}}$, beispielsweise der Flow in Abbildung 4.8, enthält eine oder mehrere Kindaktivitäten $A_{ch} = \textsc{Children}(a_{\texttt{struc}})$. Die Beziehung zwischen der strukturierten Aktivität und ihren Kindaktivitäten wird über Hierarchiebeziehungen hergestellt. Durch diese direkten Kontrollflussbeziehungen beeinflusst der Zustand der Instanz von $a_{\texttt{struc}}$ die Zustände der Instanzen ihrer Kindaktivitäten. Die Kindaktivitätsinstanzen beeinflussen wiederum den Zustand der Instanz von $a_{\texttt{struc}}$. Die Instanzen können zum Beispiel erst ausgeführt werden, wenn die Instanz der strukturierten Aktivität ausgeführt wird, und ein Fehler in einer der Instanzen führt zum Abbruch der Ausführung der Instanz von $a_{\texttt{struc}}$. Über den Zustand der Instanz von $a_{\texttt{struc}}$ beeinflussen sich damit auch die Instanzen der Kindaktivitäten untereinander, da beispielsweise der durch den Fehler verursachte Abbruch der Ausführung der Instanz von $a_{\texttt{struc}}$ zum Abbruch der Ausführung der anderen noch laufenden Kindaktivitätsinstanzen führt.

In diesem Abschnitt werden auf Basis der operationalen Semantik von BPEL die Zustandsbeziehungen zwischen einer strukturierten Aktivität und ihren Kindaktivitäten sowie die Zustandsbeziehungen zwischen den Kindaktivitä-

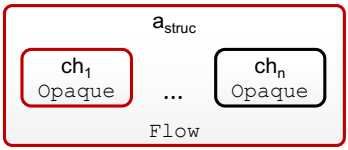

Abbildung 4.8.: Beispiel einer strukturierten Flow-Aktivität a_{struc} mit ihren Kindaktivitäten $ch_1, \ldots, ch_n \in A_{ch}$

ten im Detail erläutert und die entsprechenden Axiome definiert. Basierend auf diesen und den in den vorherigen Abschnitten definierten Axiomen, werden die Aktivitätszustandsprofile zwischen den Aktivitäten ermittelt. Da die Axiome in diesem Abschnitt auf alle strukturierten Aktivitätstypen anwendbar sein sollen, werden die durch Kontrollflusskanten oder den konkreten Aktivitätstyp implizierten Synchronisationsabhängigkeiten zwischen den Aktivitätsinstanzen, die ebenfalls Einfluss auf die Zustandsbeziehungen haben, erst in den folgenden Abschnitten betrachtet.

Die vorzeitige Terminierung der Instanz $a^I_{\texttt{struc}}$ einer strukturierten Aktivität $a_{\texttt{struc}}$ durch ihre Elternaktivität hat zur Folge, dass zuvor die Instanzen ihrer Kindaktivitäten ebenfalls vorzeitig terminiert werden müssen. Sie erreichen somit nur den Zustand *aborted*:

$$\begin{aligned} &\forall ch \in A_{ch} \forall \upsilon_{ch} \in (\Upsilon \setminus \{initial, aborted\}) : \\ &\quad \omega_{\Upsilon}(a_{\texttt{struc}}, aborted, ch, aborted) := \leftarrow \qquad (St_{CH}1) \\ &\quad \wedge\, \omega_{\Upsilon}(a_{\texttt{struc}}, aborted, ch, \upsilon_{ch}) := \oplus \end{aligned}$$

Zwischen den Instanzen der Kindaktivitäten besteht keine Terminierungsreihenfolge. Dies wird durch das weiter unten stehende Axiom $St_{CH2CH}3$ definiert.

Bei der Deaktivierung von $a^I_{\texttt{struc}}$ durch Dead-Path-Elimination (siehe Abschnitt 4.2.9) müssen die Instanzen aller Kindaktivitäten ebenfalls den Zustand *dead* erreichen. Dabei ist durch BPEL nicht vorgegeben, ob zuerst

$a^I_{\texttt{struc}}$ oder ihre Kindaktivitätsinstanzen in *dead* gehen müssen:

$$\begin{aligned} \forall ch \in A_{ch} \forall v_{ch} \in (\Upsilon \setminus \{initial, dead\}) : \\ \omega_\Upsilon(a_{\texttt{struc}}, dead, ch, dead) \coloneqq \| \\ \wedge\, \omega_\Upsilon(a_{\texttt{struc}}, dead, ch, v_{ch}) \coloneqq \oplus \end{aligned} \qquad (St_{CH}2)$$

Die Reihenfolge in der die Instanzen der Kindaktivitäten den Zustand *dead* erreichen, wird durch die Kontrollflusskanten und durch die von den Eintrittsbedingungen implizierten Synchronisationsabhängigkeiten zwischen den Aktivitäten vorgegeben. Da hier die Synchronisationsabhängigkeiten innerhalb der strukturierten Aktivität nicht berücksichtigt werden, ist auch keine Reihenfolge vorgegeben, in der die Kindaktivitätsinstanzen *dead* erreichen müssen. Dies wird weiter unten im Axiom $St_{CH2CH}1$ definiert. Befindet sich die Zielaktivität einer Kontrollflusskante in einer strukturierten Aktivität, deren Instanz deaktiviert wurde, wird die Instanz der Zielaktivität ebenfalls deaktiviert, ohne dass auf die Evaluierung des Zustands ihrer eingehenden Kontrollflusskante gewartet wird.

Die Ausführung der Instanzen der Kindaktivitäten kann erst beginnen, nachdem $a^I_{\texttt{struc}}$ den Zustand *executing* erreicht hat. Folglich können die Instanzen alle direkten und indirekten Folgezustände von *executing* erst erreichen, nachdem $a^I_{\texttt{struc}}$ den Zustand *executing* erreicht hat. Eine vorzeitige Terminierung oder Deaktivierung der Instanzen ist weiterhin möglich, nachdem $a^I_{\texttt{struc}}$ den Zustand *executing* erreicht hat.

$$\forall ch \in A_{ch} \forall v \in \Upsilon \setminus \{initial\} : \omega_\Upsilon(a_{\texttt{struc}}, executing, ch, v) \coloneqq \rightarrow \qquad (St_{CH}3)$$

Solange während der Ausführung von $a^I_{\texttt{struc}}$ keine der Instanzen der Kindaktivitäten einen Fehler auslöst, beeinflussen sich, aufgrund der Nichtberücksichtigung der Synchronisationsabhängigkeiten zwischen ihnen, ihre regulären Zustände nicht. Das impliziert, dass keine Reihenfolge vorgegeben ist, in der die Instanzen in den Zustand *executing* oder *dead* gehen. Da

completed ein Nachfolgezustand von *executing* ist, folgt daraus auch, dass die Instanzen unabhängig voneinander in *completed* wechseln können:

$$\forall ch_i \neq ch_j \in A_{ch} \forall \upsilon_i, \upsilon_j \in \Upsilon_{reg} : \omega_\Upsilon(ch_i, \upsilon_i, ch_j, \upsilon_j) := \| \qquad (St_{CH2CH}1)$$

Löst eine Instanz der Kindaktivitäten einen Fehler aus und wechselt somit in den Zustand *faulted*, werden die Instanzen, die noch keinen Endzustand erreicht haben, terminiert. Danach erreicht $a^I_{\texttt{struc}}$ ebenfalls den Zustand *faulted*. Daher kann der Zustand *faulted* von $a^I_{\texttt{struc}}$ jedem anderen Zustand der Instanzen der Kindaktivitäten folgen:

$$\forall ch \in A_{ch} \forall \upsilon_{ch} \in \Upsilon : \omega_\Upsilon(a_{\texttt{struc}}, faulted, ch, \upsilon_{ch}) := \leftarrow \qquad (St_{CH}4)$$

Es kann nur eine Instanz der Kindaktivitäten den Zustand *faulted* erreichen. Alle Instanzen, die noch keinen Endzustand erreicht haben, müssen (vorzeitig) terminiert werden, nachdem der Fehler aufgetreten ist:

$$\begin{aligned} \forall ch_i \neq ch_j \in A_{ch} \forall \upsilon_{term} \in \{aborted, terminated\} : & \\ \omega_\Upsilon(ch_i, faulted, ch_j, faulted) := \oplus & \qquad (St_{CH2CH}2) \\ \wedge\, \omega_\Upsilon(ch_i, faulted, ch_j, \upsilon_{term}) := \rightarrow & \end{aligned}$$

Wird die Instanz $a^I_{\texttt{struc}}$ während der Ausführung von ihrem Eltern-Scope terminiert, werden vorher alle Instanzen der Kindaktivitäten terminiert, die sich nicht in einem Endzustand befinden. Die Instanz $a^I_{\texttt{struc}}$ kann nicht terminiert werden, wenn eine Instanz der Kindaktivitäten den Zustand *faulted* erreicht hat, da dies implizieren würde, dass sich $a^I_{\texttt{struc}}$ bereits im Zustand *faulted* befände. Daher kann eine Instanz von $a_{\texttt{struc}}$ den Zustand *terminated* nach jedem Kindaktivitätsinstanzzustand außer *faulted* erreichen:

$$\forall ch \in A_{ch} \forall \upsilon \in \Upsilon \setminus \{faulted\} :$$
$$\omega_\Upsilon(a_{\texttt{struc}}, terminated, ch, faulted) := \oplus \qquad (St_{CH}5)$$
$$\wedge\, \omega_\Upsilon(a_{\texttt{struc}}, terminated, ch, \upsilon) := \leftarrow$$

Bei der Terminierung von $a^I_{\texttt{struc}}$ bzw. wenn eine der Instanzen der Kindaktivitäten von $a_{\texttt{struc}}$ einen Fehler auslöst, ist ebenfalls keine Reihenfolge vorgegeben, in der die Instanzen der Kindaktivitäten terminiert werden:

$$\forall ch_i \neq ch_j \in A_{ch} \forall \upsilon_i, \upsilon_j \in \{aborted, terminated\} :$$
$$\omega_\Upsilon(ch_i, \upsilon_i, ch_j, \upsilon_j) := \| \qquad (St_{CH2CH}3)$$

Eine Instanz $a^I_{\texttt{struc}}$ kann den Zustand *completed* nur erreichen, wenn keine der Kindaktivitätsinstanzen einen Fehler ausgelöst hat, d.h. alle Instanzen einen regulären Endzustand erreicht haben:

$$\forall ch \in A_{ch} \forall \upsilon_{error} \in \Upsilon_{error} \forall \upsilon_{endReg} \in (\Upsilon_{reg} \cap \Upsilon_{final}) :$$
$$\omega_\Upsilon(a_{\texttt{struc}}, completed, ch, \upsilon_{endReg}) := \leftarrow \qquad (St_{CH}6)$$
$$\wedge\, \omega_\Upsilon(a_{\texttt{struc}}, completed, ch, \upsilon_{error}) := \oplus$$

Ein Fehlerzustand in einer der Instanzen der Kindaktivitäten führt zum Abbruch der Ausführung von $a^I_{\texttt{struc}}$. Während der Ausführung von $a^I_{\texttt{struc}}$ können die Instanzen der Kindaktivitäten deshalb nur so lange einen regulären Zustand erreichen, bis eine andere oder mehrere Instanzen einen Fehlerzustand erreicht haben:

$$\forall ch_i \neq ch_j \in A_{ch} \forall \upsilon_{reg} \in \Upsilon_{reg} \forall \upsilon_{error} \in \Upsilon_{error} :$$
$$\omega_\Upsilon(ch_i, \upsilon_{reg}, ch_j, \upsilon_{error}) := \rightarrow \qquad (St_{CH2CH}4)$$

Auf Basis der hier definierten Axiome und den Axiomen *Act1*, *Act2*, *Chor1*, *Chor2* kann das in Abbildung 4.9 dargestellte Profil zwischen einer struk-

		A_{ch}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{struc}	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ $St_{CH}2$	⊕ $St_{CH}2$	⊕ $St_{CH}2$	⊕ $St_{CH}2$	⊕ $St_{CH}2$	⊕ $St_{CH}2$
	ABORTED	← Chor2	⊕ $St_{CH}1$	‖ $St_{CH}1$	⊕ $St_{CH}1$	⊕ $St_{CH}1$	⊕ $St_{CH}1$	⊕ $St_{CH}1$
	EXECUTING	← Chor2	→ $St_{CH}3$	→ $St_{CH}3$	→ $St_{CH}3$	→ $St_{CH}3$	→ $St_{CH}3$	→ $St_{CH}3$
	TERMINATED	← Chor2	← $St_{CH}5$	‖ $St_{CH}5$	← $St_{CH}5$	‖ $St_{CH}5$	⊕ $St_{CH}5$	← $St_{CH}5$
	FAULTED	← Chor2	← $St_{CH}4$	← $St_{CH}4$	← $St_{CH}4$	← $St_{CH}4$	← $St_{CH}4$	← $St_{CH}4$
	COMPLETED	← Chor2	← $St_{CH}6$	← $St_{CH}6$	← $St_{CH}6$	⊕ $St_{CH}6$	⊕ $St_{CH}6$	← $St_{CH}6$

Abbildung 4.9.: Zustandstransitionsprofil zwischen einer strukturierten Aktivität a_{struc} und ihrer Kindaktivitäten A_{ch}

turierten Aktivität $a_{\texttt{struc}}$ und ihren Kindaktivitäten A_{ch} abgeleitet werden. Das Profil, das basierend auf den vorher erwähnten Axiomen sowie den Axiomen $St_{CH2CH}1$ bis $St_{CH2CH}4$ zwischen allen Paaren von Kindaktivitäten ch_i und ch_j von $a_{\texttt{struc}}$ gilt, ist in Abbildung 4.10 dargestellt.

4.2.6. Scope & Prozessmodell

Ein Scope bzw. ein Prozessmodell[1] $s \in S$ (Abschnitt 3.2.4.8) ist eine strukturierte Aktivität mit einer Menge von Kindaktivitäten A_{ch}:

$$A_{ch} := A_{rch} \cup A_{fh}$$

Die Menge $A_{rch} := \{a_{pch}\} \cup \{a_{eh}\}$ repräsentiert dabei die regulären Kindaktivitäten des Scopes, wobei a_{pch} die primäre Kindaktivität und Aktivität a_{eh} die Ereignisbehandlungsaktivität des Scopes ist. Die Menge A_{fh} umfasst dessen Fehlerbehandlungsaktivitäten. Jede der Fehlerbehandlungsaktivitäten ist

[1] Die Beschreibungen für Scopes in diesem Abschnitt schließen Prozessmodelle mit ein, d.h. die Axiome lassen sich auch auf Prozessmodelle anwenden.

		ch_j						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
ch_i	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$
	ABORTED	← Chor2	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	← $St_{CH2CH}2$	← $St_{CH2CH}4$
	EXECUTING	← Chor2	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$
	TERMINATED	← Chor2	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	← $St_{CH2CH}2$	← $St_{CH2CH}4$
	FAULTED	← Chor2	← $St_{CH2CH}4$	→ $St_{CH2CH}2$	← $St_{CH2CH}4$	→ $St_{CH2CH}2$	⊕ $St_{CH2CH}2$	← $St_{CH2CH}4$
	COMPLETED	← Chor2	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$

Abbildung 4.10.: Zustandstransitionsprofil zwischen verschiedenen Kindaktivitäten ch_i und ch_j einer strukturierten Aktivität

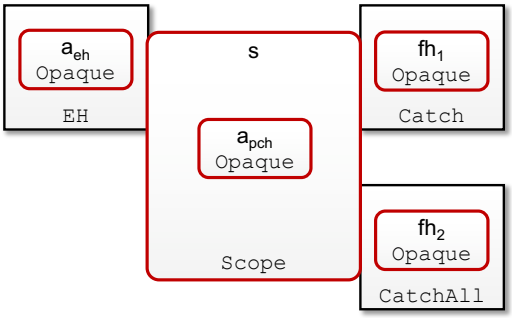

Abbildung 4.11.: Kindaktivitäten von Scopes

genau einem Fault-Handler und die Ereignisbehandlungsaktivität einem Event-Handler zugeordnet.

Der in Abbildung 4.11 dargestellte Scope hat zum Beispiel einen Event-Handler sowie zwei Fault-Handler mit den Fehlerbehandlungsaktivitäten a_{fh1} und a_{fh2}. Die Instanz der Ereignisbehandlungsaktivität a_{eh} kann im regulären Kontrollfluss durch Timer-Ereignisse parallel zu der Instanz der primären Kindaktivität ausgeführt werden. Löst eine Instanz der regulären Kindaktivi-

täten einen Fehler aus, kann, abhängig von der Art des Fehlers, mittels der Fehlerbehandlungsaktivität a_{fh1} oder a_{fh2} darauf reagiert werden.

Neben den Aktivitätszuständen Υ definieren Kopp et al. in ihrem Zustandsmodell für Scopes in [KHK+11] noch weitere Zustände, die den Status der Fehlerbehandlung einer Scope-Instanz näher kennzeichnen. In diesem Modell befindet sich eine Scope-Instanz beispielsweise im Zustand *faulthandling*, so lange eine Instanz der Fehlerbehandlungsaktivitäten ausgeführt wird. Außerdem existiert eine Erweiterung des Zustands *completed*, um zu kennzeichnen, dass eine Scope-Instanz erfolgreich einen Fehler behandelt hat und der Fehler nicht weiter zur Instanz der Elternaktivität des Scopes propagiert wurde. Um Fallunterscheidungen zwischen den Zuständen von Scope-Instanzen und Instanzen anderer Aktivitätstypen zu vermeiden, werden diese Zustände hier nicht berücksichtigt. Behandelt eine Scope-Instanz einen Fehler, befindet sie sich weiter im Zustand *executing*, bis die Fehlerbehandlung abgeschlossen ist. Hat eine Scope-Instanz erfolgreich einen Fehler behandelt, wird dieser nicht weiter propagiert und die Instanz erreicht den Zustand *completed*. In dieser Arbeit wird also angenommen, dass Scope-Instanzen erfolgreich behandelte Fehler immer unterdrücken. Daher erreichen Scope-Instanzen nur den Zustand *faulted*, wenn die Instanz einer der Fehlerbehandlungsaktivitäten einen Fehler ausgelöst hat.

Ein Scope ist eine strukturierte Aktivität, daher gelten zwischen dem Scope und seinen Kindaktivitäten A_{ch} die in den Axiomen $St_{CH}1$ bis $St_{CH}6$ definierten Zustandsbeziehungen und das daraus abgeleitete Profil in Abbildung 4.9. Es bestehen allerdings Unterschiede zwischen den Zustandsbeziehungen der Kindaktivitäten, da sich diese, abgesehen von der primären Kindaktivität, alle in separaten Handlern befinden. Die Beziehungen zwischen der primären Kindaktivität und der Ereignisbehandlungsaktivität wird in Abschnitt 4.2.6.1 diskutiert. Abschnitt 4.2.6.2 beschreibt die Beziehungen, die zwischen den regulären Kindaktivitäten und den Fehlerbehandlungsaktivitäten sowie zwischen den Fehlerbehandlungsaktivitäten in verschiedenen Fault-Handlern gelten.

4.2.6.1. Ereignisbehandlungsaktivitäten

In dieser Arbeit wird zur Vereinfachung der Axiome und der Verifikation davon ausgegangen, dass die Ereignisbehandlungsaktivität a_{eh} im Event-Handler nur durch ein einmaliges Timer-Ereignis aktiviert werden darf. Es kann also zur Laufzeit nur eine Instanz von a_{eh} erzeugt werden. Dies ist eine Einschränkung zu BPEL, wo derselbe Event-Handler aufgrund von sich wiederholenden Timer- oder Nachrichtenereignissen zur Laufzeit mehrmals aktiviert werden kann und damit Ereignisbehandlungsaktivitäten mehrmals instanziiert werden können.

Diese Vereinfachung impliziert keine Einschränkung der möglichen Interaktionsmuster einer Choreographie, die modelliert und konsolidiert werden können. Der mehrmalige Empfang von Nachrichten kann, wie in Abschnitt 7.2 erläutert, mit dem Metamodell über Schleifen realisiert werden, die Empfangsaktivitäten enthalten. Pick-Aktivitäten können zur Behandlung von alternativen Nachrichtenereignissen genutzt werden. Wie ebenfalls in Abschnitt 7.2 diskutiert, können diesen Interaktionen konsolidiert werden. Die einzige Einschränkung durch die fehlende Unterstützung für Nachrichtenereignisse im Event-Handler besteht darin, dass die Nachrichten nicht parallel zum Kontrollfluss eines Scopes verarbeitet werden können.

Die Ereignisbehandlungsaktivität ist eine reguläre Kindaktivität des Scopes. Es können somit die in Abschnitt 4.2.5 definierten Zustandsbeziehungen angewandt werden. Dadurch gelten, mit den hier diskutierten Einschränkungen, die Axiome zwischen Kindaktivitäten von strukturierten Aktivitäten. Der Event-Handler eines Scopes wird installiert, wenn die Instanz des Scopes in den Zustand *executing* geht. Ab diesem Zeitpunkt kann auch die Instanz der primären Kindaktivität a^I_{pch} und die Instanz a^I_{eh} der Ereignisbehandlungsaktivität ausgeführt werden, sobald das Timer-Ereignis eintritt. Löst a^I_{pch} oder a^I_{eh} einen Fehler aus, wird die jeweils andere Instanz terminiert. Wird eine Scope-Instanz vorzeitig terminiert oder in den Zustand *dead* gesetzt, werden auch die Instanzen a^I_{pch} und a^I_{eh} in den Zustand *aborted* bzw. *dead* gesetzt. Im regulären Kontrollfluss wird der Event-Handler deaktiviert, sobald a^I_{pch} und

somit auch die Scope-Instanz den Zustand *completed* erreicht hat. Bei der Deaktivierung des Event-Handlers wird die Instanz von a^I_{eh} in *dead* gesetzt, sofern das Timer-Ereignis nicht ausgelöst wurde.

Ein Scope hat nur eine primäre Kindaktivität. Sie kann daher nur den Zustand *dead* erreichen, wenn die Instanz des Scopes in den Zustand *dead* gesetzt wurde. Dies impliziert, dass a^I_{eh} ebenfalls in den Zustand *dead* gehen muss, wenn a^I_{pch} den Zustand *dead* erreicht hat und somit keinen anderen Zustand mehr erreichen kann:

$$\forall \upsilon_{eh} \in (\Upsilon \setminus \{initial, dead\}) : \omega_\Upsilon(a_{pch}, dead, a_{eh}, \upsilon_{eh}) \coloneqq \oplus \qquad (Sc_{PCH2EH}1)$$

Timer-Ereignisse können parallel zur Ausführung der Instanz der primären Kindaktivität eintreten. Daher kann die Instanz der Ereignisaktivität unabhängig von der Instanz der primären Kindaktivität die Zustände des regulären Kontrollflusses erreichen. Das Axiom $St_{CH2CH}1$, das keine Reihenfolge zwischen dem Erreichen des Zustands *executing* der Kindaktivitätsinstanzen vorgibt, gilt daher weiter. Folglich gilt auch weiterhin das Axiom $St_{CH2CH}4$. Denn da die Instanz a^I_{eh} den Zustand *executing* vor a^I_{pch} erreichen kann, kann ein Fehler in a^I_{eh} zur vorzeitigen Terminierung von a^I_{pch} führen.

Der Event-Handler eines Scopes wird nach seiner Ausführung deinstalliert und die Instanz der Ereignisbehandlungsaktivität wird, falls sie noch nicht ausgeführt wurde, in den Zustand *dead* gesetzt. Da a_{pch} die einzige Kindaktivität des Scopes ist, wird die Ausführung der Scope-Instanz beendet, nachdem a^I_{pch} den Zustand *completed* erreicht hat. Die Instanz a^I_{eh} kann daher nur den Zustand *executing* erreichen, so lange a^I_{pch} nicht im Zustand *completed* ist. Nachdem a^I_{pch} in den Zustand *completed* gewechselt ist, kann a^I_{eh} vom Zustand *initial* nur den Zustand *dead* erreichen. Den Zustand *aborted* kann a^I_{eh} ebenfalls nicht mehr erreichen, da a^I_{pch} keinen Fehler mehr auslösen und es somit nicht zur vorzeitigen Terminierung von a^I_{eh} kommen kann.

$$\begin{aligned} &\omega_{\Upsilon}(a_{pch}, completed, a_{eh}, executing) := \leftarrow \\ \wedge\, &\omega_{\Upsilon}(a_{pch}, completed, a_{eh}, dead) := \rightarrow \\ \wedge\, &\omega_{\Upsilon}(a_{pch}, completed, a_{eh}, aborted) := \oplus \end{aligned} \qquad (Sc_{PCH2EH}4)$$

Dies ist eine Einschränkung des Axioms $St_{CH2CH}1$, das keine Reihenfolge zwischen den regulären Aktivitätszuständen der Kindaktivitäten vorgibt. Zusätzlich wird das Axiom $St_{CH2CH}4$ hier eingeschränkt, da Instanzen von a_{eh} den Zustand *aborted* nicht erreichen können, nachdem die zugehörigen Instanzen von a_{pch} in *completed* gegangen sind. Befindet sich eine Instanz von a_{eh} bereits im Zustand *executing*, wenn eine Instanz von a_{pch} in den Zustand *completed* geht, darf sie zu Ende ausgeführt werden, d.h. dahingehend wird das Axiom $St_{CH2CH}1$ nicht eingeschränkt.

Die Terminierung oder ein Fehler während der Ausführung von a^I_{pch} führt zur (vorzeitigen) Terminierung von a^I_{eh}. Daraus folgt, dass eine Instanz von a_{eh} nie den Zustand *dead* erreichen kann, wenn die Instanz von a_{pch} den Zustand *terminated* oder *faulted* erreicht hat:

$$\begin{aligned} &\omega_{\Upsilon}(a_{pch}, terminated, a_{eh}, dead) := \oplus \\ \wedge\, &\omega_{\Upsilon}(a_{pch}, faulted, a_{eh}, dead) := \oplus \end{aligned} \qquad (Sc_{PCH2EH}5)$$

Das ist eine Einschränkung des Axioms $St_{CH2CH}2$, das festlegt, dass die Instanzen der Kindaktivitäten jeden regulären Zustand erreichen können, bis eine dieser Instanzen einen Fehlerzustand erreicht hat.

Das aus den Axiomen resultierende Profil ist Abbildung 4.12 dargestellt. Bis auf eine werden alle Beziehungen direkt durch die Axiome $St_{CH2CH}1$ bis $St_{CH2CH}4$ sowie die Axiome $Sc_{PCH2EH}1$ bis $Sc_{PCH2EH}4$ definiert. Die Beziehung $\omega_{\Upsilon}(a_{pch}, executing, a_{eh}, dead) := \rightarrow$ folgt daraus, dass eine Instanz von a_{eh} den Zustand *dead* erreichen kann, nachdem die zugehörige Instanz von a_{pch} *completed* erreicht hat (Axiom $Sc_{PCH2EH}4$) und dass der Zustand *completed* auf den Zustand *executing* (Axiom *Act1*) folgt.

		a_{eh}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{pch}	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ $St_{CH2CH}1$	⊕ $Sc_{PCH2EH}1$	⊕ $Sc_{PCH2E}1$	⊕ $Sc_{PCH2E}1$	⊕ $Sc_{PCH2E}1$	⊕ $Sc_{PCH2E}1$
	ABORTED	← Chor2	⊕ $Sc_{PCH2E}1$	‖ $St_{CH2CH}3$	← $St_{CH2CH}4$	⊕ $Sc_{PCH2E}3$	⊕ $Sc_{PCH2E}3$	⊕ $Sc_{PCH2E}3$
	EXECUTING	← Chor2	→ $Sc_{PCH2EH}4$, Act1	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$
	TERMINATED	← Chor2	⊕ $Sc_{PCH2EH}5$	‖ $St_{CH2CH}3$	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	← $St_{CH2CH}2$	← $St_{CH2CH}4$
	FAULTED	← Chor2	⊕ $Sc_{PCH2EH}5$	→ $St_{CH2CH}4$	← $St_{CH2CH}4$	→ $St_{CH2CH}2$	⊕ $St_{CH2CH}2$	← $St_{CH2CH}4$
	COMPLETED	← Chor2	→ $Sc_{PCH2EH}4$	⊕ $Sc_{PCH2EH}4$	← $Sc_{PCH2EH}4$	→ $St_{CH2CH}4$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$

Abbildung 4.12.: Zustandstransitionsprofil zwischen der primären Kindaktivität a_{pch} und der Ereignisbehandlungsaktivität a_{eh} eines Scopes

4.2.6.2. Fehlerbehandlungsaktivitäten

Im Folgenden werden die Zustandsbeziehungen erläutert, die zwischen der primären Kindaktivität a_{pch} und den Fehlerbehandlungsaktivitäten A_{fh} sowie die, die zwischen den Fehlerbehandlungsaktivitäten aus verschiedenen Fault-Handlern eines Scopes s gelten. Das Metamodell lässt keine Event-Handler zu, die Nachrichten empfangen und auch die Konsolidierung erzeugt keine neuen Kontrollflusskonstrukte zwischen der Ereignisbehandlungsaktivität und den Fehlerbehandlungsaktivitäten von Scopes. Daher werden die Zustandsbeziehungen zwischen der Ereignisbehandlungsaktivität und den Fehlerbehandlungsaktivitäten hier nicht erläutert.

Die Menge A_{fh} aller Fehlerbehandlungsaktivitäten eines Scopes s ist dabei folgendermaßen definiert:

$$A_{fh} := \{a \in A \mid \exists hr \in HR : \pi_1(hr) = s \wedge \pi_2(hr) \in E^{fault} \wedge a = \pi_3(hr)\}$$

In BPEL wird ein Fault-Handler aktiviert und damit die Instanz seiner Fehlerbehandlungsaktivität in den Zustand *executing* gesetzt, nachdem die Instanz der regulären Kindaktivitäten einen Fehler ausgelöst hat. Dabei wird immer nur die Instanz einer Fehlerbehandlungsaktivität ausgeführt und zwar die des Fault-Handlers, der den Fehler bzw. das Fehlerereignis verarbeiten kann. Löst zum Beispiel eine Instanz der Aktivität a_{pch} in Abbildung 4.11 einen Fehler aus, der nur vom Standard-Fault-Handler verarbeitet werden kann (mit `catchAll` gekennzeichnet), würde die Instanz der Aktivität a_{fh1} in den Zustand *dead* und die Instanz der Aktivität a_{fh2} innerhalb des Standard-Fault-Handlers in den Zustand *executing* gehen. Alle anderen Fault-Handler werden deinstalliert und die Instanzen ihrer Fehlerbehandlungsaktivitäten werden somit in den Zustand *dead* gesetzt. Dabei ist durch die BPEL-Spezifikation nicht vorgegeben, in welcher Reihenfolge die Fault-Handler aktiviert und deinstalliert werden.

Alle Instanzen der Kindaktivitäten des Scopes werden in den Zustand *dead* gesetzt, wenn der Scope deaktiviert wurde. In diesem Fall kann die Instanz der primären Kindaktivität den Zustand *executing* nicht erreichen und damit keinen Fehler auslösen. Folglich können die Instanzen der Fehlerbehandlungsaktivitäten nur den Zustand *dead* erreichen:

$$\forall a_{fh} \in A_{fh} \forall \upsilon \in \Upsilon \setminus \{initial, dead\} : \omega_{\Upsilon}(a_{pch}, dead, a_{fh}, \upsilon) := \oplus \quad (Sc_{PCH2FH}1)$$

Die Instanz der primären Kindaktivität kann ebenfalls keinen Fehler auslösen, wenn sie terminiert wurde. Die Instanzen der Fehlerbehandlungsaktivitäten können nur den Zustand *aborted* erreichen:

$$\begin{aligned} &\forall a_{fh} \in A_{fh}\ \forall \upsilon_{term} \in \{aborted, terminated\} \\ &\quad \forall \upsilon \in \Upsilon \setminus \{initial, aborted\} : \qquad (Sc_{PCH2FH}2) \\ &\quad\quad \omega_{\Upsilon}(a_{pch}, \upsilon_{term}, a_{fh}, \upsilon) := \oplus \end{aligned}$$

Nachdem die Instanz der primären Kindaktivität den Zustand *faulted* erreicht

hat, kann die Instanz der aktivierten Fehlerbehandlungsaktivität den Zustand *executing* (mit dessen Nachfolgezuständen) und die Instanzen der deaktivierten Fehlerbehandlungsaktivitäten den Zustand *dead* erreichen.

$$\begin{aligned} &\forall a_{fh} \in A_{fh}\ \forall \upsilon_{exec} \in \Upsilon \setminus \{initial, aborted\} : \\ &\quad \omega_{\Upsilon}(a_{pch}, faulted, a_{fh}, \upsilon_{exec}) := \rightarrow \\ &\wedge\, \omega_{\Upsilon}(a_{pch}, faulted, a_{fh}, aborted) := \oplus \end{aligned} \qquad (Sc_{PCH2FH}3)$$

Die Instanzen aller Fehlerbehandlungsaktivitäten gehen in *dead*, wenn kein Fehler ausgelöst wird. Tritt ein Fehler auf, kann nur eine Instanz der Fehlerbehandlungsaktivitäten den Zustand *executing* erreichen, während die Instanzen der anderen in den Zustand *dead* gesetzt werden:

$$\begin{aligned} &\forall a_{fhi} \neq a_{fhj} \in A_{fh} \forall \upsilon_i, \upsilon_j \in \Upsilon \setminus \{initial, dead, aborted\} : \\ &\quad \omega_{\Upsilon}(a_{fhi}, \upsilon_i,\ a_{fhj}, \upsilon_j) := \oplus \end{aligned} \qquad (Sc_{FH2FH})$$

Zwischen der Deinstallation und der Aktivierung von Fault-Handlern ist in BPEL keine Reihenfolge spezifiziert. Daher ist auch keine Reihenfolge vorgegeben, in der die Instanzen der Fehlerbehandlungsaktivitäten den Zustand *dead* und *executing* erreichen müssen. Dies wird bereits durch das Axiom $St_{CH2CH}1$ abgebildet.

Haben die Instanzen der regulären Kindaktivitäten keinen Fehler ausgelöst und den Zustand *completed* erreicht, werden danach alle Fault-Handler deinstalliert. Damit gehen die Instanzen der Fehlerbehandlungsaktivitäten direkt danach in den Zustand *dead* und können somit auch keinen anderen Zustand mehr erreichen:

$$\begin{aligned} &\forall a_{fh} \in A_{fh} \forall \upsilon \in \Upsilon \setminus \{dead\} : \\ &\quad \omega_{\Upsilon}(a_{pch}, completed, a_{fh}, dead) := \rightarrow \\ &\wedge\, \omega_{\Upsilon}(a_{pch}, completed, a_{fh}, \upsilon) := \oplus \end{aligned} \qquad (Sc_{PCH2FH}4)$$

		a_{fh}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{pch}	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ $St_{CH2CH}1$	⊕ $Sc_{PCH2FH}1$	⊕ $Sc_{PCH2FH}1$	⊕ $Sc_{PCH2FH}1$	⊕ $Sc_{PCH2FH}1$	⊕ $Sc_{PCH2FH}1$
	ABORTED	← Chor2	⊕ $Sc_{PCH2FH}2$	‖ $St_{CH2CH}3$	⊕ $Sc_{PCH2FH}2$	⊕ $Sc_{PCH2FH}2$	⊕ $Sc_{PCH2FH}2$	⊕ $Sc_{PCH2FH}2$
	EXECUTING	← Chor2	‖ $Sc_{PCH2FH}4$,Act1	→ $St_{CH2CH}4$	→ $Sc_{PCH2FH}3$	→ $Sc_{PCH2FH}3$	→ $Sc_{PCH2FH}3$	→ $Sc_{PCH2FH}3$
	TERMINATED	← Chor2	⊕ $Sc_{PCH2FH}2$	‖ $St_{CH2CH}3$	⊕ $Sc_{PCH2FH}2$	⊕ $Sc_{PCH2FH}2$	⊕ $Sc_{PCH2FH}2$	⊕ $Sc_{PCH2FH}2$
	FAULTED	← Chor2	→ $Sc_{PCH2FH}3$	⊕ $St_{CH2CH}3$	→ $Sc_{PCH2FH}3$	→ $Sc_{PCH2FH}3$	→ $Sc_{PCH2FH}3$	→ $Sc_{PCH2FH}3$
	COMPLETED	← Chor2	→ $Sc_{PCH2FH}4$	⊕ $Sc_{PCH2FH}4$	⊕ $Sc_{PCH2FH}4$	⊕ $Sc_{PCH2FH}4$	⊕ $Sc_{PCH2FH}4$	⊕ $Sc_{PCH2FH}4$

Abbildung 4.13.: Zustandstransitionsprofil einer primären Kindaktivität a_{pch} und den Fehlerbehandlungsaktivitäten $a_{fh} \in A_{fh}$ eines Scopes

Das aus diesen Axiomen resultierende Profil zwischen der primären Kindaktivität und den Fehlerbehandlungsaktivitäten ist in Abbildung 4.13 dargestellt.

Abbildung 4.14 zeigt das Profil, das zwischen allen Paaren von Fehlerbehandlungsaktivitäten eines Scopes gilt. Wie oben erwähnt, ist in BPEL nur spezifiziert, dass die Fault-Handler, die einen Fehler nicht verarbeiten können deinstalliert und die Fehlerbehandlungsaktivitäten in den Zustand *dead* gesetzt werden müssen. Es ist allerdings nicht spezifiziert, wann dies geschehen muss. Daher besteht theoretisch die Möglichkeit, dass eine Fehlerbehandlungsaktivität einen Fehler verarbeitet und *completed* erreicht hat, bevor die Fehlerbehandlungsaktivitäten der deinstallierten Fault-Handler in den Zustand *dead* gesetzt wurden. Deshalb gilt in dem Profil zum Beispiel weiterhin die Beziehung $\omega_{\gamma}(a_{fhi}, completed, a_{fhj}, dead) = \|$. Dies führt auch dazu, dass die Fehlerbehandlungsaktivität des deinstallierten Fault-Handlers anstatt in *dead* in *aborted* gesetzt werden kann, falls der Scope während der Fehlerbehandlung terminiert wird. Folglich gilt zwischen den regulären Zuständen und dem Zustand *aborted* weiterhin das Axiom $St_{CH2CH}4$.

		a_{fhj}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{fhi}	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$
	ABORTED	← Chor2	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	← $St_{CH2CH}2$	← $St_{CH2CH}4$
	EXECUTING	← Chor2	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	⊕ Sc_{FH2FH}	⊕ Sc_{FH2FH}	⊕ Sc_{FH2FH}	⊕ Sc_{FH2FH}
	TERMINATED	← Chor2	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	⊕ Sc_{FH2FH}	⊕ Sc_{FH2FH}	⊕ Sc_{FH2FH}	⊕ Sc_{FH2FH}
	FAULTED	← Chor2	← $St_{CH2CH}4$	→ $St_{CH2CH}2$	⊕ Sc_{FH2FH}	⊕ Sc_{FH2FH}	⊕ Sc_{FH2FH}	⊕ Sc_{FH2FH}
	COMPLETED	← Chor2	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	⊕ Sc_{FH2FH}	⊕ Sc_{FH2FH}	⊕ Sc_{FH2FH}	⊕ Sc_{FH2FH}

Abbildung 4.14.: Zustandstransitionsprofil zwischen den Fehlerbehandlungsaktivitäten $a_{fhi}, a_{fhj} \in A_{fh}$ $(a_{fhi} \neq a_{fhj})$ aus verschiedenen Fault-Handlern eines Scopes

4.2.7. Isolierte Aktivitäten

Im weiteren Verlauf wird zwischen den Zustandsbeziehungen von Aktivitäten aufgrund ihrer Isolation voneinander unterschieden, wobei die folgenden drei Isolationsgrade betrachtet werden. Eine Aktivität a_j ist *vollständig isoliert* (kurz isoliert) von a_i, wenn Zustandsänderungen der Instanzen von a_i weder direkt noch indirekt zu Zustandsänderungen in den Instanzen von a_j führen. Führt ein Fehler in den Instanzen von a_i nicht zur Terminierung der Instanzen von a_j, ist a_j *fehlerisoliert* von a_i. Bei fehlerisolierten Aktivitäten ist es möglich, dass eine reguläre Zustandsänderung von a_i direkt oder indirekt zu einer Zustandsänderung von a_j führt. Aktivität a_j ist *nicht isoliert* von a_i, wenn a_j nicht fehlerisoliert und damit auch nicht vollständig isoliert von a_i ist.

Auf Choreographieebene ist eine Aktivität a_j isoliert von a_i, wenn beide Aktivitäten über die Verhaltensbeschreibungen verschiedenen Teilnehmern $\mathfrak{p}_i$ und $\mathfrak{p}_j$ zugeordnet sind. Nachrichtenkanten können Abhängigkeiten zwischen den Zuständen von Aktivitäten in verschiedenen Teilnehmern implizieren (siehe Abschnitt 4.2.14). Es darf daher keine Nachrichtenkante zwischen $\mathfrak{p}_i$ und $\mathfrak{p}_j$ existieren, bei der a_i Sendeaktivität oder Vorgänger der Sendeaktivität

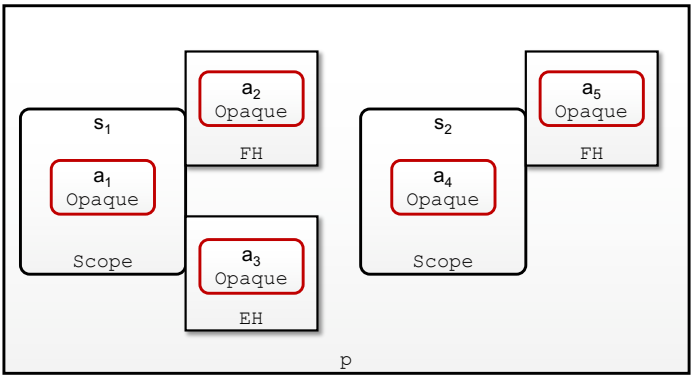

Abbildung 4.15.: Isolation von Aktivitäten durch verschiedene Scopes

und a_j Empfangsaktivität bzw. Nachfolger der Empfangsaktivität ist. Wäre dies der Fall, könnte zum Beispiel ein Fehler in der Instanz von a_i dazu führen, dass die Nachricht nicht gesendet wird und die Instanz von a_j nicht ausgeführt werden kann.

Auf Prozessmodellebene ist a_j von a_i isoliert, wenn sie sich in unterschiedlichen nicht verschachtelten Scopes s_i und s_j befinden. Der Fault-Handler des Scopes s_i darf keine Fehler weiter propagieren, da dies zur Terminierung der Instanzen von a_j führen würde. Außerdem darf a_j keine Synchronisationsabhängigkeit zu a_i haben, da im regulären Kontrollfluss a_j erst in einen Nachfolgezustand von *initial* gehen könnte, nachdem a_i einen Endzustand erreicht hat. Befinden sich a_i und a_j in nicht verschachtelten Scopes und a_j hat eine Synchronisationsabhängigkeit zu a_i, ist a_j fehlerisoliert von a_i. In Abbildung 4.15 werden durch die Scopes s_1 und s_2 zum Beispiel die Aktivitäten $a_1 - a_3$ von den Aktivitäten a_4 und a_5 isoliert.

In diesem Abschnitt werden die Zustandsbeziehungen von vollständig isolierten Aktivitäten betrachtet. Auf fehlerisolierte Aktivitäten wird in den nachfolgenden Abschnitten eingegangen.

Da sich die Zustände von Instanzen vollständig isolierter Aktivitäten a_i

		a_j						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_i	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso
	ABORTED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso
	EXECUTING	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso
	TERMINATED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso
	FAULTED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso
	COMPLETED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso

Abbildung 4.16.: Zustandstransitionsprofil von zwei vollständig isolierten Aktivitäten a_i und a_j

und a_j nicht beeinflussen, existiert auch keine Reihenfolge zwischen ihren Zustandstransitionen :

$$\begin{gathered} \forall \upsilon_i, \upsilon_j \in \Upsilon : \\ \omega_\Upsilon(a_i, \upsilon_i, a_j, \upsilon_j) := \| \end{gathered} \tag{Iso}$$

Die Beziehung $\oplus$ gilt zwischen a_i und a_j nicht, da dies bedeuten würde, dass die Instanzen von a_j einen bestimmten Zustand υ_j nicht einnehmen könnten, wenn sich die Instanzen von a_i in einem Zustand υ_i befänden, was der Isolation widerspräche.

Wie in Axiom *Chor1* definiert, werden alle Aktivitätsinstanzen einer Konversation während ihrer Instanziierung in den Zustand *initial* gesetzt. Aus diesem Grund besteht nur die Beziehung zwischen den Aktivitäten, dass diese auf *initial* gesetzt werden, bevor sie einen Nachfolgezustand von *initial* erreichen. Abbildung 4.16 zeigt das entsprechende Profil für isolierte Aktivitäten.

4.2.8. Flow

Die Flow-Aktivität ist eine strukturierte Aktivität mit mehreren Kindaktivitäten (Abschnitt 3.2.4.9). Die durch Kontrollflusskanten und Gateways implizierten Synchronisationsabhängigkeiten zwischen den Kindaktivitäten werden in den folgenden Abschnitten diskutiert. Daher gelten zwischen den Zuständen des Flows und seinen Kindaktivitäten sowie zwischen den Kindaktivitäten untereinander ohne Einschränkungen die in Abschnitt 4.2.5 definierten Axiome und die daraus resultierenden Profile aus Abbildung 4.9 bzw. Abbildung 4.10.

4.2.9. Sequenz

Die sequentielle Hintereinanderausführung von Aktivitäten bzw. deren Instanzen (Workflow-Muster „Sequence“ aus [vdAtHKB03]) kann mittels des Metamodells über eine Kontrollflusskante l realisiert werden, die eine Vorgängeraktivität a_{pred} mit einer Nachfolgeaktivität a_{succ} verbindet ($a_{pred}, a_{succ} \in A$). Die Instanz von a_{succ} muss im regulären Kontrollfluss immer genau dann ausgeführt werden, wenn die Instanz von a_{pred} erfolgreich beendet wurde. Die Transitionsbedingung dieser Kante muss zur Laufzeit immer zu `true` evaluieren, damit die Kante aktiviert wird. Zusätzlich muss die Eintrittbedingung von a_{succ} so spezifiziert sein, dass die Instanz von a_{succ} ausgeführt wird, wenn die eingehende Kante aktiviert wurde. Befinden sich a_{pred} und a_{succ} in unterschiedlichen Elternaktivitäten[1] $a_{parPred}$ und $a_{parSucc}$, muss im Kontrollfluss sichergestellt werden, dass deren Instanzen $a^I_{parPred}$ bzw. $a^I_{parSucc}$ zur Laufzeit immer in den Zustand *executing* gehen. Würde zum Beispiel $a^I_{parPred}$ den Zustand *executing* erreichen und $a^I_{parSucc}$ den Zustand *dead*, führte dies dazu, dass a^I_{succ} ebenfalls in *dead* geht. Folglich wäre die sequentielle Hintereinanderausführung der Instanzen von a_{pred} und a_{succ} nicht sichergestellt.

Abhängig davon, ob die Vorgängeraktivität und ihre Nachfolgeaktivität nicht

[1] Es gilt: $a_{parPred} = \textsc{Parent}(a_{pred})$ und $a_{parSucc} = \textsc{Parent}(a_{succ})$

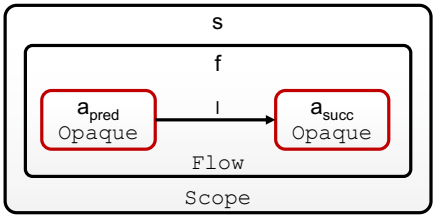

Abbildung 4.17.: Sequentielle Ausführung von nicht isolierter Vorgänger- und Nachfolgeaktivität

isoliert oder fehlerisoliert voneinander sind, gelten unterschiedliche Zustandsbeziehungen zwischen ihnen, die in den folgenden Abschnitten erläutert werden.

4.2.9.1. Nicht isolierte Sequenz

Sind die Aktivitäten a_{pred} und a_{succ} wie im Beispiel in Abbildung 4.17 nicht fehlerisoliert, befinden sich die beiden Aktivitäten entweder im selben Scope oder in verschachtelten Scopes:

$$s_{pred} = s_{succ} \vee s_{pred} \in \textsc{Ancestors}(s_{succ}) \vee s_{succ} \in \textsc{Ancestors}(s_{pred})$$
$$\text{mit } s_{pred} = \textsc{ParentScope}(a_{pred}) \text{ und } s_{succ} = \textsc{ParentScope}(a_{succ})$$

Da Fehler somit zwischen den Instanzen von a_{pred} und a_{succ} propagiert werden würden, dienen die in Abschnitt 4.2.5 diskutierten Zustandsbeziehungen (Axiome $St_{CH2CH}1 - St_{CH2CH}4$) zwischen den Kindaktivitäten einer strukturierten Aktivität als Grundlage und werden hier weiter eingeschränkt. Wie eingangs beschrieben, muss die Instanz a^I_{succ} im regulären Kontrollfluss *executing* und dessen Nachfolgezustände erreichen, nachdem die Instanz a^I_{pred} den Zustand *executing* bzw. *completed* erreicht hat. Den Zustand *dead* kann a^I_{succ} also nicht mehr erreichen, nachdem a^I_{pred} in *executing* gesetzt wurde:

$$\forall \upsilon_i \in \{executing, completed\} \forall \upsilon_j \in \Upsilon \setminus \{initial, dead\} :$$
$$\omega_\Upsilon(a_{pred}, \upsilon_i, a_{succ}, \upsilon_j) := \rightarrow \qquad (Sq1)$$
$$\wedge\, \omega_\Upsilon(a_{pred}, \upsilon_i, a_{succ}, dead) := \oplus$$

Erreicht a^I_{pred} den Zustand *dead*, geht a^I_{succ} aufgrund von Dead-Path-Elimination danach ebenfalls in den Zustand *dead*:

$$\forall \upsilon \in \Upsilon \setminus \{initial, dead\} :$$
$$\omega_\Upsilon(a_{pred}, dead, a_{succ}, dead) := \rightarrow \qquad (Sq2)$$
$$\wedge\, \omega_\Upsilon(a_{pred}, dead, a_{succ}, \upsilon) := \oplus$$

Während der Evaluierung der eingehenden Kontrollflusskante von a_{succ} kann eine Aktivitätsinstanz auf einem parallelen Pfad von a_{succ} (die parallele Aktivität ist nicht fehlerisoliert von a_{succ}) einen Fehler auslösen. Deshalb ist es ebenfalls möglich, dass eine Instanz von a_{succ} vorzeitig terminiert wird und somit in den Zustand *aborted* geht, auch nachdem die Instanz von a_{pred} den Zustand *completed* oder *dead* erreicht hat. Dies ist bereits durch das Axiom $St_{CH2CH}4$ abgebildet.

Das Axiom $St_{CH2CH}4$ definiert auch, dass jede Aktivitätsinstanz des Flows einen Endzustand erreichen kann, solange keine Instanz seiner Kindaktivitäten einen Fehlerzustand erreicht hat. Durch die Kontrollflusskante kann aber eine Instanz von a_{succ} nie den Zustand *executing* oder *dead* erreichen, nachdem eine Instanz von a_{pred} einen Fehlerzustand erreicht hat. Daher gilt zwischen sequentiell ausgeführten Aktivitäten a_{pred} und a_{succ} anstatt dem Axiom $St_{CH2CH}4$ das Axiom *Sq3*:

$$\forall \upsilon_{error} \in \Upsilon_{error} \forall \upsilon \in \Upsilon \setminus \{initial, aborted\} :$$
$$\omega_\Upsilon(a_{pred}, \upsilon_{error}, a_{succ}, \upsilon) := \oplus \qquad (Sq3)$$

Wie in Abschnitt 4.2.5 erläutert, werden, nachdem eine Aktivitätsinstanz

		a_{succ}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{pred}	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	→ Sq2	→ $St_{Ch2Ch}4$	⊕ Sq2	⊕ Sq2	⊕ Sq2	⊕ Sq2
	ABORTED	← Chor2	⊕ Sq3	‖ $St_{Ch2Ch}3$	⊕ Sq3	⊕ Sq3	⊕ Sq3	⊕ Sq3
	EXECUTING	← Chor2	⊕ Sq1	→ $St_{Ch2Ch}4$	→ Sq1	→ Sq1	→ Sq1	→ Sq1
	TERMINATED	← Chor2	⊕ Sq3	‖ $St_{Ch2Ch}3$	⊕ Sq3	⊕ Sq3	⊕ Sq3	⊕ Sq3
	FAULTED	← Chor2	⊕ Sq3	→ $St_{Ch2Ch}2$	⊕ Sq3	⊕ Sq3	⊕ Sq3	⊕ Sq3
	COMPLETED	← Chor2	⊕ Sq1	→ $St_{Ch2Ch}4$	→ Sq1	→ Sq1	→ Sq1	→ Sq1

Abbildung 4.18.: Zustandstransitionsprofil von nicht isolierten sequentiellen Aktivitäten a_{pred} und a_{succ}

im Scope, indem sich der Flow befindet, einen Fehler ausgelöst hat, alle anderen Aktivitätsinstanzen im Flow terminiert. Folglich wird auch die Instanz von a_{succ} terminiert, nachdem die Instanz von a_{pred} in den Zustand *faulted* gegangen ist (Axiom $St_{CH2CH}2$).

In BPEL ist nicht spezifiziert, dass eine Kontrollflusskante eine Terminierungsreihenfolge zwischen Aktivitätsinstanzen impliziert. Aus diesem Grund gilt weiterhin das Axiom $St_{CH2CH}3$, d.h. eine Instanz von a_{succ} kann vor oder nach einer Instanz von a_{pred} terminiert werden. Das aus den Axiomen *Sq1* - *Sq3* sowie den Axiomen $St_{CH2CH}2$ und $St_{CH2CH}4$ resultierende Profil ist in Abbildung 4.18 dargestellt.

4.2.9.2. Fehlerisolierte Sequenz

Im Gegensatz zur nicht isolierten Sequenz befinden sich bei einer fehlerisolierten Sequenz die Vorgängeraktivität a_{pred} und Nachfolgeaktivität a_{succ}, wie im Beispiel in Abbildung 4.19, in parallelen Scopes[1]:

[1] Während a_{succ} fehlerisoliert von a_{pred} ist, ist a_{pred} sogar vollständig von a_{succ} isoliert.

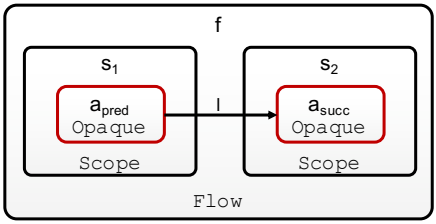

Abbildung 4.19.: Sequentielle Ausführung von fehlerisolierter Vorgänger- und Nachfolgeaktivität

$$s_{succ} \notin \textsc{Ancestors}(s_{pred}) \wedge s_{pred} \notin \textsc{Ancestors}(s_{succ})$$

Im regulären Kontrollfluss muss eine Instanz von a_{succ} weiterhin den Zustand *executing* erreichen, nachdem die Instanz von a_{pred} den Zustand *executing* bzw. *completed* erreicht hat. Das Axiom *Sq1* behält also seine Gültigkeit. Das Axiom *Sq2* bleibt ebenfalls gültig, weil trotz der Isolation von a_{pred} und a_{succ} die Instanz a^I_{succ} weiterhin nicht den Zustand *executing* erreichen kann, wenn a^I_{pred} den Zustand *dead* erreicht hat.

Erreicht a^I_{pred} einen Fehlerzustand, wird dieser Fehler dadurch, dass sich a_{pred} und a_{succ} in unterschiedlichen Scopes befinden, nicht an a^I_{succ} weiter propagiert. Um zu vermeiden, dass die Instanz von a_{succ} keinen Endzustand erreicht und somit „hängt“, ist in BPEL spezifiziert, dass Kontrollflusskanten, deren Vorgängeraktivität mit einer Nachfolgeaktivität in einem anderen Scope verbunden sind, deaktiviert werden, nachdem die Instanz der Vorgängeraktivität einen Fehlerzustand erreicht hat. Daraus folgt, da a_{succ} nur eine eingehende Kontrollflusskante hat, dass die Instanzen von a_{succ} immer in den Zustand *dead* gehen, nachdem eine Instanz von a_{pred} einen Fehlerzustand erreicht hat. Dies ist in Axiom *SqI3* definiert, das anstelle von Axiom *Sq3* verwendet wird, wenn sich Vorgänger- und Nachfolgeaktivität in unterschiedlichen Scopes befinden. Das Axiom gilt auch, wenn sich a_{pred} im Fault-Handler eines Scopes und a_{succ} außerhalb des Fault-Handlers befinden.

		a_{succ}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{pred}	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	→ Sq2	‖ Iso	⊕ Sq2	⊕ Sq2	⊕ Sq2	⊕ Sq2
	ABORTED	← Chor2	→ SqI3	‖ Iso	⊕ SqI3	⊕ SqI3	⊕ SqI3	⊕ SqI3
	EXECUTING	← Chor2	⊕ Sq1	‖ Iso	→ Sq1	→ Sq1	→ Sq1	→ Sq1
	TERMINATED	← Chor2	→ SqI3	‖ Iso	⊕ SqI3	⊕ SqI3	⊕ SqI3	⊕ SqI3
	FAULTED	← Chor2	→ SqI3	‖ Iso	⊕ SqI3	⊕ SqI3	⊕ SqI3	⊕ SqI3
	COMPLETED	← Chor2	⊕ Sq1	‖ Iso	→ Sq1	→ Sq1	→ Sq1	→ Sq1

Abbildung 4.20.: Zustandstransitionsprofil von fehlerisolierten sequentiellen Aktivitäten a_{pred} und a_{succ}

$$\begin{aligned} &\forall \upsilon_{error} \in \Upsilon_{error} \forall \upsilon \in \Upsilon \setminus \{initial, dead, aborted\} : \\ &\quad \omega_{\Upsilon}(pred, \upsilon_{error}, a_{succ}, \upsilon) := \oplus \\ &\wedge\, \omega_{\Upsilon}(pred, \upsilon_{error}, a_{succ}, dead) := \rightarrow \end{aligned} \tag{SqI3}$$

Das Profil in Abbildung 4.20 resultiert aus den Axiomen *Sq1*, *Sq2*, *SqI3* und dem Axiom *Iso*. Letzteres wird hier angewandt, da a_{pred} vollständig von a_{succ} isoliert ist und dadurch die vorzeitige Terminierung der Instanz von a_{succ} keinen Einfluss auf den Zustand der Instanz a^I_{pred} hat. Eine Instanz von a_{succ} kann also *aborted* auch vor allen Zuständen (abgesehen von *initial*) einer Instanz von a_{pred} erreichen.

4.2.10. Parallele Verzweigung

Bei parallelen Verzweigungen (Workflow-Muster „Parallel Split“) wird der Kontrollfluss in parallele Pfade aufgeteilt, die zur Laufzeit simultan ausgeführt werden. Sie kann mit dem Metamodell block- und graphbasiert modelliert werden.

Bei der blockbasierten Modellierung werden alle direkten Kindaktivitäten eines Flows parallel ausgeführt, die keine eingehenden Kanten besitzen.

Die graphbasierte Modellierung erfolgt über eine *parallele Verzweigungsaktivität* a_{split} ($a_{split} \in A$) , die zwei oder mehrere ausgehenden Kontrollflusskanten mit entsprechenden Zielaktivitäten besitzt. Analog zur Sequenz müssen die Transitionsbedingungen der ausgehenden Kanten und die Eintrittsbedingungen der Zielaktivitäten so definiert sein, dass sie alle zur Laufzeit ausgeführt werden, nachdem die Instanz von a_{split} erfolgreich ausgeführt wurde. Befinden sich die parallelen Aktivitäten in unterschiedlichen Elternaktivitäten, müssen diese im regulären Kontrollfluss ebenfalls den Zustand *executing* erreichen können, wenn a_{split} ausgeführt wurde.

Zur Reduzierung der Fallunterscheidungen bei der Verhaltensbeschreibung von parallelen Aktivitäten dürfen die Nachfolgeaktivitäten von a_{split} keine weiteren Vorgängeraktivitäten haben. Die Einschränkung dient nur zur Vereinfachung der Verifizierung und schränkt nicht die Interaktionsmuster ein, die konsolidiert werden können (siehe Kapitel 7). Die Prozesskonsolidierungsalgorithmen in Kapitel 5 sind in der Lage, parallele Aktivitäten mit mehreren Vorgängern zu konsolidieren.

Die parallelen Aktivitäten, die unter Berücksichtigung dieser Vereinfachung durch die block- oder graphbasierte Verzweigung entstehen, werden durch die Menge A_{par} repräsentiert:

$$A_{par} := \{a \in A \mid a_{split} \in \textsc{Predecessors}(a) \wedge |\textsc{Predecessors}(a)| = 1\}$$

In Abschnitt 4.2.10.1 werden die Beziehungen zwischen den Zustandstransitionen von parallelen Verzweigungen diskutiert, bei denen die parallelen Zielaktivitäten nicht isoliert voneinander sind, d.h. $\forall a_i, a_j \in A_{par}$: $\textsc{ParentScope}(a_i) = \textsc{ParentScope}(a_j)$. Abschnitt 4.2.10.2 diskutiert darauf aufbauend die Beziehungen zwischen fehlerisolierten parallelen Aktivitäten, die sich jeweils in einem anderen Scope befinden, d.h. für sie gilt $\forall a_i, a_j \in A_{par} : \textsc{Ancestors}(a_i) \cap \textsc{Ancestors}(a_j) \cap A_{\texttt{scope}} = \emptyset$.

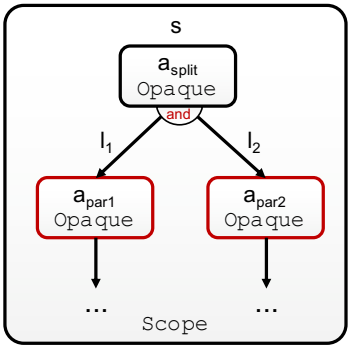

Abbildung 4.21.: Graphbasierte parallele Verzweigung mit nicht isolierten parallelen Aktivitäten

4.2.10.1. Nicht isolierte parallele Verzweigung

Eine graphbasierte parallele Verzweigung, bei der sich die parallelen Ziel- bzw. Nachfolgeaktivitäten von a_{split} im selben Scope befinden, ist in Abbildung 4.21[1] dargestellt. Die parallele Verzweigungsaktivität ist in der Abbildung durch die Bezeichnung *and* gekennzeichnet. Die Beziehungen zwischen den Zuständen der Instanzen der Verzweigungsaktivität a_{split} und ihrer jeweiligen Nachfolgeaktivitäten wurde bereits in Abschnitt 4.2.9 diskutiert. Deshalb werden hier nur die Beziehungen zwischen den Zuständen der Instanzen der parallelen Aktivitäten A_{par} betrachtet. Die Aktivitäten befinden sich in derselben strukturierten Aktivität. Daher gelten zwischen ihnen die Zustandsbeziehungen zwischen den Kindaktivitäten einer strukturierten Aktivität aus Abschnitt 4.2.5.

Die einzige zusätzliche Einschränkung besteht darin, dass im regulären Kontrollfluss entweder alle Instanzen der parallelen Aktivitäten in den Zustand *executing* oder alle Instanzen in den Zustand *dead* gehen (Axiom And_{split}):

[1] Der Flow, das die Aktivitäten beinhaltet, ist der Übersichtlichkeit wegen hier und in den folgenden Abbildungen nicht mehr dargestellt.

$$\forall a_{pari} \neq a_{parj} \in A_{par}\ \forall \upsilon_{exec} \in \Upsilon \setminus \{initial, dead, aborted\} : \omega_{\Upsilon}(a_{pari}, dead, a_{parj}, \upsilon_{exec}) := \oplus \quad (And_{split})$$

Dabei ist die Reihenfolge, in der die Instanzen den Zustand *executing* bzw. *dead* erreichen müssen, nicht vorgegeben[1]. Dies wird bereits durch Axiom $St_{CH2CH}1$ abgebildet.

Im Fehlerfall besteht daher die Möglichkeit, dass nicht alle Instanzen der parallelen Aktivitäten den Zustand *executing* bzw. *dead* erreichen, sondern vorzeitig terminiert werden. So kann zum Beispiel in Abbildung 4.21 eine Instanz der Aktivität a_{par1}, wenn die Kante l_1 als erstes evaluiert wurde, direkt den Zustand *executing* erreichen. Geht die Instanz dann in den Zustand *faulted* bevor eine Instanz von a_{par2} *executing* erreicht, wird danach die Instanz von a_{par2} von *initial* in den Zustand *aborted* gesetzt. Daher behält das Axiom $St_{CH2CH}4$ seine Gültigkeit. Das aus den Axiomen resultierende Profil ist in Abbildung 4.22 dargestellt.

4.2.10.2. Fehlerisolierte parallele Verzweigung

Befinden sich die parallelen Aktivitäten in unterschiedlichen nicht verschachtelten Scopes, wie im Beispiel in Abbildung 4.23, sind sie im Fehlerfall voneinander isoliert und es besteht nur die durch die parallele Verzweigung implizierte Kontrollflussbeziehung zwischen ihnen. Löst also eine Instanz der parallelen Aktivitäten einen Fehler aus oder wird sie von ihrem Eltern-Scope terminiert, werden die anderen Instanzen nicht terminiert, sondern weiter regulär ausgeführt.

Durch die Isolation der parallelen Aktivitäten in unterschiedlichen Scopes, basieren die Beziehungen zwischen den Zustandstransitionen auf dem Axi-

[1] In BPEL hängt die tatsächliche Reihenfolge in der Kontrollflusskanten evaluiert werden, von ihrem Auftreten im `<source>` Element der Aktivität ab. Da das Metamodell Mengen verwendet, um die ausgehenden Kanten zu repräsentieren, ist die Reihenfolge nicht-deterministisch.

		a_{parj}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{pari}	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	⊕ And_{split}	⊕ And_{split}	⊕ And_{split}	⊕ And_{split}
	ABORTED	← Chor2	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	← $St_{CH2CH}2$	← $St_{CH2CH}4$
	EXECUTING	← Chor2	⊕ And_{split}	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$
	TERMINATED	← Chor2	⊕ And_{split}	‖ $St_{CH2CH}3$	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	← $St_{CH2CH}2$	← $St_{CH2CH}4$
	FAULTED	← Chor2	⊕ And_{split}	→ $St_{CH2CH}2$	← $St_{CH2CH}4$	→ $St_{CH2CH}2$	⊕ $St_{CH2CH}2$	← $St_{CH2CH}4$
	COMPLETED	← Chor2	⊕ And_{split}	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$

Abbildung 4.22.: Zustandstransitionsprofil von nicht isolierten parallelen Aktivitäten a_{pari} und a_{parj}

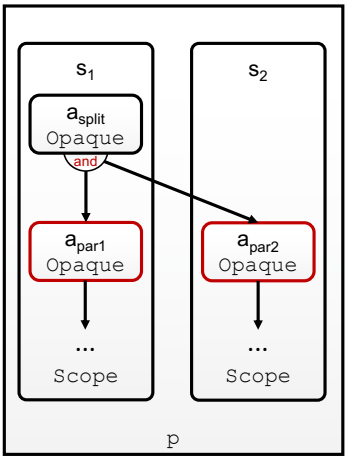

Abbildung 4.23.: Fehlerisolierte parallele Verzweigung

		a_{parj}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{pari}	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ Iso	‖ Iso	⊕ And_{split}	⊕ And_{split}	⊕ And_{split}	⊕ And_{split}
	ABORTED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso
	EXECUTING	← Chor2	⊕ And_{split}	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso
	TERMINATED	← Chor2	⊕ And_{split}	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso
	FAULTED	← Chor2	⊕ And_{split}	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso
	COMPLETED	← Chor2	⊕ And_{split}	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso

Abbildung 4.24.: Zustandstransitionsprofil von fehlerisolierten parallelen Aktivitäten a_{pari} und a_{parj}

om *Iso* (Abschnitt 4.2.7). Eingeschränkt wird das Axiom *Iso* nur durch das oben definierte Axiom And_{split}, das weiterhin seine Gültigkeit behält. Daraus resultiert das Profil in Abbildung 4.24.

4.2.11. Exklusive Verzweigung

Die exklusive Verzweigung (Workflow-Muster „Exclusive Choice“) dient dazu, den Kontrollfluss in alternative Pfade aufzuteilen, von denen abhängig von einer Verzweigungsbedingung genau einer zur Laufzeit aktiviert wird. Mit dem hier verwendeten Metamodell kann eine exklusive Verzweigung implizit mittels einer *exklusiven Verzweigungsaktivität* a_{split} realisiert werden, die zwei oder mehrere ausgehende Kontrollflusskanten mit disjunkten Transitionsbedingungen besitzt. Die Zielaktivitäten dieser Kanten werden im Folgenden als alternative Aktivitäten bezeichnet und durch die Menge A_{alt} repräsentiert.

Analog zu den parallelen Aktivitäten wird hier ebenfalls davon ausgegangen, dass die alternativen Aktivitäten nur die Vorgängeraktivität a_{split} haben, d.h. sie besitzen keine weiteren eingehenden Kontrollflusskanten. Wie bei

den parallelen Aktivitäten dient diese Einschränkung zur Vereinfachung der Verifizierung. Die konsolidierbaren Interaktionsmuster werden dadurch nicht eingeschränkt und die Algorithmen in Kapitel 5 sind in der Lage, parallele Aktivitäten mit mehreren Vorgängern zu konsolidieren.

$$A_{alt} := \{a \in A \mid a_{split} \in \textsc{Predecessors}(a) \wedge |\textsc{Predecessors}(a)| = 1\}$$

Zwischen der Aktivität a_{split} und ihren alternativen Nachfolgeaktivitäten A_{alt} gelten die in Abschnitt 4.2.9 definierten Beziehungen. Im Folgenden werden deshalb die Beziehungen zwischen den alternativen Aktivitäten diskutiert. Wie bei der parallelen Verzweigung wird hier zwischen Zustandsbeziehungen unterschieden, bei denen die alternativen Aktivitäten nicht isoliert und fehlerisoliert voneinander sind.

4.2.11.1. Nicht isolierte alternative Aktivitäten

Ein Beispiel für eine exklusive Verzweigung mit zwei alternativen Aktivitäten ist in Abbildung 4.25 dargestellt (die Verzweigungsaktivität ist dort mit *XOR* gekennzeichnet). Da bei der exklusiven Verzweigung nur eine ausgehende Kante der Verzweigungsaktivität aktiviert werden darf, kann im regulären Kontrollfluss auch nur eine der alternativen Aktivitäten bzw. deren Instanzen den Zustand *executing* erreichen, während die andere in *dead* gesetzt werden muss. Dies ist im Axiom Xor_{split} definiert:

$$\begin{aligned} \forall a_{alti} \neq a_{altj} \in A_{alt}\ \forall \upsilon_i, \upsilon_j \in \Upsilon \setminus \{initial, dead, aborted\} : \\ \omega_{\Upsilon}(a_{alti}, \upsilon_i, a_{altj}, \upsilon_j) := \oplus \end{aligned} \qquad (Xor_{split})$$

Durch den Nichtdeterminismus bei der Evaluation der Transitionsbedingungen der Kanten ist keine Reihenfolge vorgegeben, in der die Instanzen *dead* bzw. *executing* erreichen. Dies ist bereits durch Axiom $St_{CH2CH}1$ abgebildet. Das resultierende Profil ist in Abbildung 4.26 dargestellt. Wie bei der nicht isolierten parallelen Verzweigung, besteht auch hier die Möglichkeit, dass

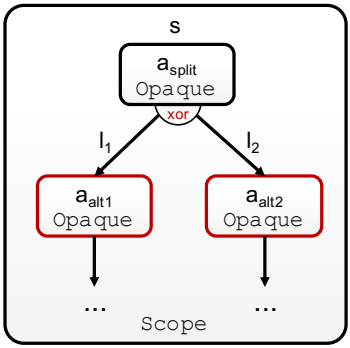

Abbildung 4.25.: Exklusive Verzweigung mit nicht isolierten alternativen Aktivitäten

		a_{altj}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{alti}	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$
	ABORTED	← Chor2	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	← $St_{CH2CH}2$	← $St_{CH2CH}4$
	EXECUTING	← Chor2	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}
	TERMINATED	← Chor2	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}
	FAULTED	← Chor2	← $St_{CH2CH}4$	→ $St_{CH2CH}2$	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}
	COMPLETED	← Chor2	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}

Abbildung 4.26.: Zustandstransitionsprofil zwischen nicht isolierten alternativen Aktivitäten a_{alti} und a_{altj}

nur eine Instanz der alternativen Aktivitäten den Zustand *executing* bzw. *dead* erreicht, bevor ein Fehler auftritt und somit die Instanz der anderen alternativen Aktivität, die diesen Zustand nicht erreicht hat, vorzeitig terminiert wird.

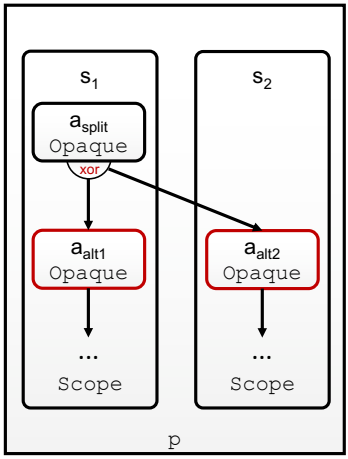

Abbildung 4.27.: Exklusive Verzweigung mit fehlerisolierten alternativen Aktivitäten

4.2.11.2. Fehlerisolierte alternative Aktivitäten

Eine exklusive Verzweigung bei der sich die alternativen Aktivitäten in jeweils separaten Scopes befinden, ist in Abbildung 4.27 dargestellt. Wie bei parallelen Aktivitäten in unterschiedlichen Scopes, basieren die Beziehungen zwischen den Zustandstransitionen der alternativen Aktivitäten ebenfalls auf dem Axiom *Iso*. Zusammen mit dem Axiom Xor_{split} führt dies zum Profil in Abbildung 4.28.

4.2.12. Parallele Vereinigung

Die parallele Vereinigung (Workflow-Muster „Synchronization“) dient dazu, die parallelen Pfade im Kontrollfluss, die aus einer parallelen Verzweigung resultieren, wieder zu einem Pfad zusammenzuführen. Die Vereinigung kann mittels einer *parallelen Vereinigungsaktivität* a_{join} ($a_{join} \in A$) mit zwei eingehenden Kontrollflusskanten modelliert werden. Die Eintrittsbedingung von a_{join} muss dabei so definiert sein, dass alle eingehenden Kontrollflusskan-

		a_{altj}						
		INITIAL	**DEAD**	**ABORTED**	**EXECUTING**	**TERMINATED**	**FAULTED**	**COMPLETED**
a_{alti}	**INITIAL**	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso
	ABORTED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso
	EXECUTING	← Chor2	‖ Iso	‖ Iso	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}
	TERMINATED	← Chor2	‖ Iso	‖ Iso	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}
	FAULTED	← Chor2	‖ Iso	‖ Iso	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}
	COMPLETED	← Chor2	‖ Iso	‖ Iso	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}	⊕ Xor_{split}

Abbildung 4.28.: Zustandstransitionsprofil zwischen fehlerisolierten alternativen Aktivitäten a_{alti} und a_{altj}

ten aktiviert sind, um a_{join} ausführen zu können. Daher muss für sie gelten $\text{joinCond}(a_{join}) = \bigwedge_{i=1}^{n} 1_i \ (n = |\textsc{Predecessors}(a_{join})|)$.

Die Quellaktivitäten der eingehenden Kontrollflusskanten, d.h. die parallelen Vorgängeraktivitäten der Vereinigungsaktivität a_{join} werden durch die Menge A_{pred} repräsentiert:

$$A_{pred} := \{a \in A \mid a \in \textsc{Predecessors}(a_{join})\}$$

Zwischen den Instanzen der parallelen Vorgängeraktivitäten A_{pred} gelten die in Abschnitt 4.2.10 definierten Zustandsbeziehungen. Hier werden daher die Zustandsbeziehungen zwischen der Vereinigungsaktivität und ihren parallelen Vorgängeraktivitäten diskutiert. Dabei werden zuerst die Beziehungen betrachtet, die gelten, wenn sich a_{join} und A_{pred} im selben Scope befinden und somit nicht isoliert voneinander sind. Dann wird der fehlerisolierte Fall diskutiert, bei dem sich a_{split} und A_{pred} in unterschiedlichen Scopes befinden.

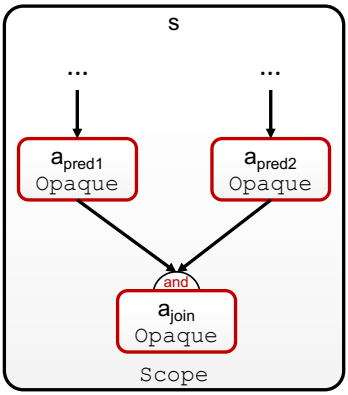

Abbildung 4.29.: Parallele Vereinigung bei der die Vorgängeraktivitäten und die Vereinigungsaktivität nicht isoliert sind

4.2.12.1. Nicht isolierte parallele Vereinigung

Das Beispiel in Abbildung 4.29 zeigt eine Vereinigung von zwei parallelen Pfaden (die parallele Vereinigungsaktivität ist mit einem *and* gekennzeichnet). Dadurch, dass die Aktivitäten aus A_{pred} direkte Vorgängeraktivitäten von a_{join} sind, ähneln die Zustandsbeziehungen denen, die für die Sequenz in Abschnitt 4.2.9 diskutiert wurden. Der Unterschied besteht durch die Eintrittsbedingung darin, dass hier im regulären Kontrollfluss eine Instanz von a_{join} nur den Zustand *executing* erreichen kann, wenn alle ihre eingehenden Kanten aktiviert wurden. Daher müssen alle Instanzen der parallelen Vorgängeraktivitäten in den Zustand *executing* bzw. *completed* gehen, bevor eine Instanz von a_{join} den Zustand *executing* und dessen Nachfolgezustände erreichen kann (Axiom $And_{Join}1$). Gehen eine oder mehrere Instanzen der Vorgängeraktivitäten in *dead*, erreicht auch die Instanz von a_{join} den Zustand *dead* (Axiom $And_{Join}2$), da die ausgehenden Kanten der sich in *dead* befindlichen Instanzen deaktiviert werden und damit auch die Eintrittsbedingung zu `false` evaluiert. Deshalb kann, im Gegensatz zu dem Axiom *Sq1*, auf den Zustand *executing* und *completed* einer Instanz der Vorgängeraktivität weiterhin der Zustand *dead* der entsprechenden Instanz von a_{join} folgen.

$$\forall a_{pred} \in A_{pred}\ \forall v_i \in \{executing, completed\}$$
$$\forall v_j \in \Upsilon \setminus \{initial, aborted\} : \qquad (And_{Join}1)$$
$$\omega_\Upsilon(a_{pred}, v_i, a_{join}, v_j) := \rightarrow$$

$$\forall a_{pred} \in A_{pred}\ \forall v_{exec} \in \Upsilon \setminus \{initial, dead, aborted\} :$$
$$\omega_\Upsilon(a_{pred}, dead, a_{join}, dead) := \rightarrow \qquad (And_{Join}2)$$
$$\wedge\, \omega_\Upsilon(a_{pred}, dead, a_{join}, v_{exec}) := \oplus$$

In BPEL muss der Zustand aller eingehenden Kontrollflusskanten einer Aktivität bekannt sein, bevor ihre Eintrittsbedingung evaluiert werden kann. Die Kanten können wiederum nur evaluiert werden, wenn alle Instanzen der Vorgängeraktivitäten den Zustand *dead* oder *completed* erreicht haben. Daraus resultiert zum Beispiel, dass, wenn die Instanz einer Vorgängeraktivität direkt in *dead* geht, während die anderen Instanzen noch im Zustand *executing* sind, die Instanz von a_{join} nicht ebenfalls direkt in den Zustand *dead* wechseln kann. Folglich gilt die Beziehung $\omega_\Upsilon(a_{pred}, completed, a_{join}, dead) := \rightarrow$ anstatt der Beziehung $\omega_\Upsilon(a_{pred}, completed, a_{join}, dead) := \|$.

Wie bei der sequentiellen Ausführung (Axiom *Sq1*), wird die Instanz von a_{join} vorzeitig terminiert, nachdem eine der Instanzen ihrer Vorgängeraktivitäten einen Fehlerzustand erreicht hat und kann somit keinen anderen Zustand als *aborted* erreichen:

$$\forall a_{pred} \in A_{pred}\ \forall v_{error} \in \Upsilon_{error}\ \forall v \in \Upsilon \setminus \{initial, aborted\} : \qquad (And_{Join}3)$$
$$\omega_\Upsilon(a_{pred}, v_{error}, a_{join}, v) := \oplus$$

Das entsprechende Profil, das die Beziehungen zwischen den Zuständen der Vereinigungsaktivität a_{join} und aller ihrer Vorgängeraktivitäten beschreibt, ist in Abbildung 4.30 dargestellt.

		a_{join}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{pred}	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	→ $And_{join}2$	→ $St_{Ch2Ch}4$	⊕ $And_{join}2$	⊕ $And_{join}2$	⊕ $And_{join}2$	⊕ $And_{join}2$
	ABORTED	← Chor2	⊕ $And_{join}3$	‖ $St_{Ch2Ch}2$	⊕ $And_{join}3$	⊕ $And_{join}3$	⊕ $And_{join}3$	⊕ $And_{join}3$
	EXECUTING	← Chor2	→ $And_{join}1$	→ $St_{Ch2Ch}4$	→ $And_{join}1$	→ $And_{join}1$	→ $And_{join}1$	→ $And_{join}1$
	TERMINATED	← Chor2	⊕ $And_{join}3$	‖ $St_{Ch2Ch}2$	⊕ $And_{join}3$	⊕ $And_{join}3$	⊕ $And_{join}3$	⊕ $And_{join}3$
	FAULTED	← Chor2	⊕ $And_{join}3$	→ $St_{Ch2Ch}2$	⊕ $And_{join}3$	⊕ $And_{join}3$	⊕ $And_{join}3$	⊕ $And_{join}3$
	COMPLETED	← Chor2	→ $And_{join}1$	→ $St_{Ch2Ch}4$	→ $And_{join}1$	→ $And_{join}1$	→ $And_{join}1$	→ $And_{join}1$

Abbildung 4.30.: Zustandstransitionsprofil nicht isolierter paralleler Vereinigung zwischen Vorgängeraktivitäten $a_{pred} \in A_{pred}$ und Vereinigungsaktivität a_{join}

4.2.12.2. Fehlerisolierte parallele Vereinigung

Bei einer fehlerisolierten parallelen Vereinigung befinden sich die Vorgängeraktivitäten A_{pred} in einem anderen Scope als die parallele Vereinigungsaktivität a_{join}. Dabei können sich die Vorgängeraktivitäten in verschiedenen oder wie im Beispiel in Abbildung 4.31 im selben Scope befinden.

Im regulären Kontrollfluss kommen weiterhin die Axiome $And_{Join}1$ und $And_{Join}2$ zur Anwendung. Wie schon bei der fehlerisolierten Sequenz diskutiert, wird ein Fehler bei der Ausführung einer der Instanzen der parallelen Vorgängeraktivitäten nicht weiter propagiert. Stattdessen wird die ausgehende Kontrollflusskante der fehlerverursachenden Aktivitätsinstanz deaktiviert und damit die Instanz von a_{join} in den Zustand *dead* gesetzt. Dies wird durch das Axiom $AndI_{Join}3$ abgebildet, das das Axiom $And_{Join}3$ ersetzt.

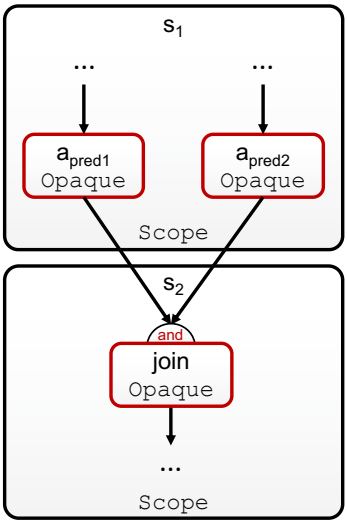

Abbildung 4.31.: Parallele Vereinigung bei der die Vorgängeraktivitäten und die Vereinigungsaktivität fehlerisoliert sind

$$\begin{aligned} &\forall a_{pred} \in A_{pred}\ \forall \upsilon_{error} \in \{aborted, terminated, faulted\} \\ &\quad \forall \upsilon \in \Upsilon \setminus \{initial, aborted, dead\} : \\ &\quad\quad \omega_{\Upsilon}(a_{pred}, \upsilon_{error}, a_{join}, \upsilon) := \oplus \\ &\quad\quad \wedge\, \omega_{\Upsilon}(a_{pred}, \upsilon_{error}, a_{join}, dead) := \rightarrow \end{aligned} \qquad (AndI_{Join}3)$$

Das resultierende Profil ist in Abbildung 4.32 dargestellt. Das Axiom *Iso* gilt nur in Zustandsbeziehungen, in denen der Zustand *aborted* involviert ist, da die Aktivitäten aus A_{pred} und a_{join} aufgrund der Fehlerisolation unabhängig voneinander terminiert werden können.

		a_{join}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{pred}	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	→ $And_{join}2$	‖ Iso	⊕ $And_{join}2$	⊕ $And_{join}2$	⊕ $And_{join}2$	⊕ $And_{join}2$
	ABORTED	← Chor2	→ $AndI_{join}3$	‖ Iso	⊕ $AndI_{join}3$	⊕ $AndI_{join}3$	⊕ $AndI_{join}3$	⊕ $AndI_{join}3$
	EXECUTING	← Chor2	→ $And_{join}1$	‖ Iso	→ $And_{join}1$	→ $And_{join}1$	→ $And_{join}1$	→ $And_{join}1$
	TERMINATED	← Chor2	→ $AndI_{join}3$	‖ Iso	⊕ $AndI_{join}3$	⊕ $AndI_{join}3$	⊕ $AndI_{join}3$	⊕ $AndI_{join}3$
	FAULTED	← Chor2	→ $AndI_{join}3$	‖ Iso	⊕ $AndI_{join}3$	⊕ $AndI_{join}3$	⊕ $AndI_{join}3$	⊕ $AndI_{join}3$
	COMPLETED	← Chor2	→ $And_{join}1$	‖ Iso	→ $And_{join}1$	→ $And_{join}1$	→ $And_{join}1$	→ $And_{join}1$

Abbildung 4.32.: Zustandstransitionsprofil fehlerisolierte parallele Vereinigung zwischen Vorgängeraktivitäten $pred \in A_{pred}$ und Vereinigungsaktivität a_{join}

4.2.13. Exklusive Vereinigung

Mittels der exklusiven Vereinigung (Workflow-Muster „Simple Merge“) können alternative Kontrollflusspfade, die durch die exklusive Verzweigung entstanden sind, wieder zu einem Pfad zusammengeführt werden. Die exklusive Vereinigung wird durch eine *exklusive Vereinigungsaktivität* a_{join} ($a_{join} \in A$) realisiert, die zwei eingehende Kontrollflusskanten hat. Die Transitionsbedingungen dieser Kanten müssen disjunkt sein, um zu garantieren, dass nur genau eine Kante zur Laufzeit aktiviert wird. Die Menge A_{pred} repräsentiert dabei die alternativen Vorgängeraktivitäten der exklusiven Vereinigungsaktivität:

$$A_{pred} = \{a \in A \mid a \in \textsc{Successors}(a)\}$$

Die Eintrittsbedingung von a_{join} muss so definiert sein, dass deren Instanz ausgeführt wird, wenn eine der eingehenden Kontrollflusskanten aktiviert wird, d.h. es gilt $\text{joinCond}(a_{join}) = \bigvee_{i=1}^{n} 1_i$ ($n = |\textsc{Predecessors}(a_{join})|$).

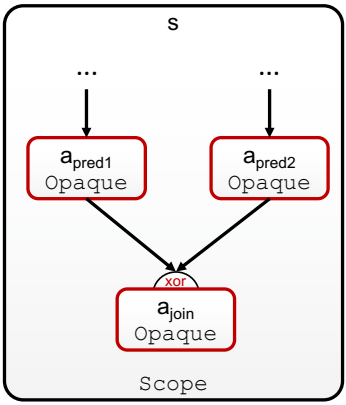

Abbildung 4.33.: Exklusive Vereinigung bei der die Vorgängeraktivitäten und die Vereinigungsaktivität nicht isoliert sind

Die Zustandsbeziehungen zwischen den alternativen Vorgängeraktivitäten wurden bereits in Abschnitt 4.2.11 definiert. In diesem Abschnitt werden deshalb die Zustandsbeziehungen zwischen der exklusiven Vereinigungsaktivität und den alternativen Vorgängeraktivitäten diskutiert. Dabei wird wieder zwischen Beziehungen unterschieden, die gelten, wenn die Vereinigungsaktivität und ihre Vorgängeraktivitäten nicht isoliert und fehlerisoliert sind.

4.2.13.1. Nicht isolierte exklusive Vereinigung

Im Beispiel in Abbildung 4.33 ist eine Vereinigungsaktivität a_{join} (exklusive Vereinigungsaktivitäten sind mit einem *xor* gekennzeichnet) mit zwei Vorgängeraktivitäten dargestellt. Erreicht im regulären Kontrollfluss eine Instanz der Vorgängeraktivitäten den Zustand *executing* und damit *completed*, geht die Instanz von a_{join}, nachdem die anderen Vorgängeraktivitätsinstanzen den Endzustand *dead* erreicht haben, in den Zustand *executing*. In diesem Fall kann die Instanz von a_{join} den Zustand *dead* nicht mehr erreichen:

$$\begin{aligned}
&\forall a_{pred} \in A_{pred}\ \forall \upsilon_{exec} \in \{executing, completed\} \\
&\quad \forall \upsilon_j \in \Upsilon \setminus \{initial, dead, aborted\} : \\
&\qquad \omega_\Upsilon(a_{pred}, \upsilon_{exec}, a_{join}, \upsilon_j) := \rightarrow \\
&\qquad \wedge\, \omega_\Upsilon(a_{pred}, \upsilon_{exec}, a_{join}, dead) := \oplus
\end{aligned} \tag{$Xor_{Join}1$}$$

Erreichen hingegen alle Instanzen der Vorgängeraktivitäten den Zustand *dead*, wird die Instanz von a_{join} danach ebenfalls in *dead* gesetzt. Wie oben erwähnt, kann die Instanz von a_{join} weiterhin *executing* erreichen, nachdem eine Instanz der Vorgängeraktivitäten *dead* erreicht hat, da nur genau eine dieser Instanzen in *completed* gehen muss.

$$\begin{aligned}
&\forall a_{pred} \in A_{pred}\ \forall \upsilon_j \in \Upsilon \setminus \{initial, aborted\} : \\
&\qquad \omega_\Upsilon(a_{pred}, dead, a_{join}, \upsilon_j) := \rightarrow
\end{aligned} \tag{$Xor_{Join}2$}$$

Eine Instanz von a_{join} geht, wie durch Axiom $St_{CH2CH}2$ bereits definiert, in den Zustand *aborted*, wenn die Instanz einer ihrer Vorgängeraktivitäten in einen Fehlerzustand gegangen ist. Folglich kann sie keinen Zustand des regulären Kontrollflusses mehr erreichen:

$$\begin{aligned}
&\forall a_{pred} \in A_{pred}\ \forall \upsilon_{error} \in \Upsilon_{error}\ \forall \upsilon_i \in \Upsilon \setminus \{initial, aborted\} : \\
&\qquad \omega_\Upsilon(a_{pred}, \upsilon_{error}, a_{join}, \upsilon_i) := \oplus
\end{aligned} \tag{$Xor_{Join}3$}$$

Im Profil in Abbildung 4.34 sind die aus den Axiomen resultierenden Beziehungen zwischen den Vorgängeraktivitäten A_{pred} und der exklusiven Vereinigungsaktivität a_{join} dargestellt.

4.2.13.2. Fehlerisolierte exklusive Vereinigung

Abbildung 4.35 zeigt das Beispiel einer exklusiven Vereinigung, bei der sich die Vereinigungsaktivität a_{join} und ihre Vorgängeraktivitäten A_{pred} in

		a_{join}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{pred}	INITIAL	$\parallel$ Chor1	$\rightarrow$ Chor2	$\rightarrow$ Chor2	$\rightarrow$ Chor2	$\rightarrow$ Chor2	$\rightarrow$ Chor2	$\rightarrow$ Chor2
	DEAD	$\leftarrow$ Chor2	$\rightarrow$ $Xor_{join}2$	$\rightarrow$ $St_{Ch2Ch}4$	$\rightarrow$ $Xor_{join}2$	$\rightarrow$ $Xor_{join}2$	$\rightarrow$ $Xor_{join}2$	$\rightarrow$ $Xor_{join}2$
	ABORTED	$\leftarrow$ Chor2	$\oplus$ $Xor_{join}3$	$\parallel$ $St_{Ch2Ch}2$	$\oplus$ $Xor_{join}3$	$\oplus$ $Xor_{join}3$	$\oplus$ $Xor_{join}3$	$\oplus$ $Xor_{join}3$
	EXECUTING	$\leftarrow$ Chor2	$\oplus$ $Xor_{join}1$	$\rightarrow$ $St_{Ch2Ch}4$	$\rightarrow$ $Xor_{join}1$	$\rightarrow$ $Xor_{join}1$	$\rightarrow$ $Xor_{join}1$	$\rightarrow$ $Xor_{join}1$
	TERMINATED	$\leftarrow$ Chor2	$\oplus$ $Xor_{join}3$	$\parallel$ $St_{Ch2Ch}2$	$\oplus$ $Xor_{join}3$	$\oplus$ $Xor_{join}3$	$\oplus$ $Xor_{join}3$	$\oplus$ $Xor_{join}3$
	FAULTED	$\leftarrow$ Chor2	$\oplus$ $Xor_{join}3$	$\rightarrow$ $St_{Ch2Ch}2$	$\oplus$ $Xor_{join}3$	$\oplus$ $Xor_{join}3$	$\oplus$ $Xor_{join}3$	$\oplus$ $Xor_{join}3$
	COMPLETED	$\leftarrow$ Chor2	$\oplus$ $Xor_{join}1$	$\rightarrow$ $St_{Ch2Ch}4$	$\rightarrow$ $Xor_{join}1$	$\rightarrow$ $Xor_{join}1$	$\rightarrow$ $Xor_{join}1$	$\rightarrow$ $Xor_{join}1$

Abbildung 4.34.: Zustandstransitionsprofil nicht isolierter exklusiver Vereinigung zwischen Vorgängeraktivitäten $a_{pred} \in A_{pred}$ und Vereinigungsaktivität a_{join}

jeweils einem separaten Scope befinden. Die im Folgenden diskutierten Beziehungen zwischen den Zuständen der Vereinigungsaktivität und den Vorgängeraktivitäten gelten auch, wenn sich alle Vorgängeraktivitäten im selben Scope befinden.

Im regulären Kontrollfluss gelten weiter die Axiome $Xor_{Join}1$ und $Xor_{Join}2$. Allerdings kann die Instanz der Vereinigungsaktivität selbst dann den Zustand *executing* oder *dead* erreichen, wenn eine oder mehrere Instanzen der Vorgängeraktivitäten einen Fehlerzustand erreicht haben. Das ist möglich, da die Eintrittsbedingung der Vereinigungsaktivität nur verlangt, dass eine der eingehenden Kanten zur Laufzeit aktiviert wird. Geht nun eine Instanz der Vorgängeraktivitäten in einen Fehlerzustand, zum Beispiel die Instanz der Aktivität a_{pred1}, wird ihre ausgehende Kontrollflusskante deaktiviert (siehe Abschnitt 4.2.9.2). Geht die Instanz einer anderen Vorgängeraktivität allerdings in den Zustand *completed*, wie beispielsweise die Instanz von a_{pred2}, wird ihre ausgehende Kontrollflusskante aktiviert, d.h. die Vereinigungsaktivität wird trotz des Fehlers ausgeführt. Erreichen alle Instanzen der

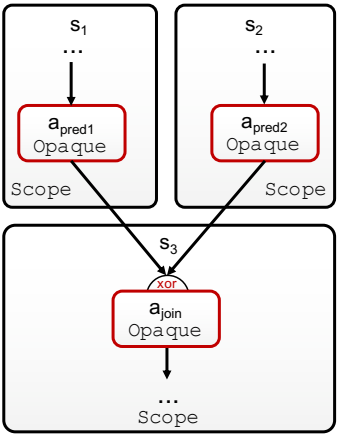

Abbildung 4.35.: Exklusive Vereinigung bei der die Vorgängeraktivitäten und die Vereinigungsaktivität fehlerisoliert sind

Vorgängeraktivitäten den Zustand *dead* oder einen Fehlerzustand, geht die Vereinigungsaktivität ebenfalls in *dead*. Die Beziehung, dass auf einen Fehlerzustand bei den Vorgängeraktivitäten die Vereinigungsaktivität in den Zustand *executing* oder *dead* gehen kann, ist in Axiom $XorI_{Join}3$ definiert, das anstatt des Axioms $Xor_{Join}3$ verwendet wird, wenn sich die Vereinigungsaktivität nicht im selben Scope befindet wie ihre Vorgängeraktivitäten.

$$\forall a_{pred} \in A_{pred}\ \forall \upsilon_{error} \in \Upsilon_{error}\ \forall \upsilon_i \in \Upsilon \setminus \{initial, aborted\} : \\ \omega_{\Upsilon}(a_{pred}, \upsilon_{error}, a_{join}, \upsilon_i) := \rightarrow \qquad (XorI_{Join}3)$$

Aus den Axiomen $Xor_{Join}1$, $Xor_{Join}2$, $XorI_{Join}3$ und *Iso* resultiert das Profil in Abbildung 4.36.

4.2.14. Asynchrone Interaktion zwischen Invoke- und Receive-Aktivität

Eine asynchrone Interaktion zwischen einem Invoke *snd* und einem Receive *rcv* wird, wie im Beispiel in Abbildung 4.37, über eine Nachrichtenkante *ml*

		a_{join}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{pred}	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	→ $Xor_{join}2$	‖ Iso	→ $Xor_{join}2$	→ $Xor_{join}2$	→ $Xor_{join}2$	→ $Xor_{join}2$
	ABORTED	← Chor2	→ $Xorl_{join}3$	‖ Iso	→ $Xorl_{join}3$	→ $Xorl_{join}3$	→ $Xorl_{join}3$	→ $Xorl_{join}3$
	EXECUTING	← Chor2	⊕ $Xor_{join}1$	‖ Iso	→ $Xor_{join}1$	→ $Xor_{join}1$	→ $Xor_{join}1$	→ $Xor_{join}1$
	TERMINATED	← Chor2	→ $Xorl_{join}3$	‖ Iso	→ $Xorl_{join}3$	→ $Xorl_{join}3$	→ $Xorl_{join}3$	→ $Xorl_{join}3$
	FAULTED	← Chor2	→ $Xorl_{join}3$	‖ Iso	→ $Xorl_{join}3$	→ $Xorl_{join}3$	→ $Xorl_{join}3$	→ $Xorl_{join}3$
	COMPLETED	← Chor2	→ $Xor_{join}2$	‖ Iso	⊕ $Xor_{join}2$	⊕ $Xor_{join}2$	⊕ $Xor_{join}2$	⊕ $Xor_{join}2$

Abbildung 4.36.: Zustandstransitionsprofil fehlerisolierter exklusiver Vereinigung zwischen Vorgängeraktivitäten $a_{pred} \in A_{pred}$ und Vereinigungsaktivität a_{join}

modelliert, bei der *snd* die Quell- und *rcv* die Zielaktivität ist. Die Aktivitäten sind verschiedenen Teilnehmern zugeordnet (im Beispiel $\mathfrak{p}_{snd}$ und $\mathfrak{p}_{rcv}$) und haben nur durch die Nachrichtenübertragung Zustandsbeziehungen.

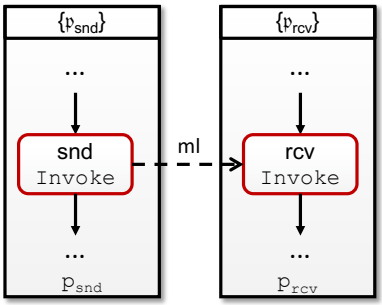

Abbildung 4.37.: Nachrichtenkante zwischen einem Invoke und Receive

Eine Invoke-Instanz sendet eine Nachricht, wenn sie sich im Zustand *executing* befindet. Ist der Versand der Nachricht erfolgreich abgeschlossen, wechselt die Instanz in den Zustand *completed*. Auf der Empfängerseite muss sich die Receive-Instanz im Zustand *executing* befinden, um die Nachricht zu emp-

fangen. Nach dem erfolgreichen Empfang geht sie ebenfalls in den Zustand *completed*. Tritt beim Empfang der Nachricht ein Fehler auf, wechselt die Receive-Instanz in den Zustand *faulted*. Es wird hier davon ausgegangen, dass die Instanz nur in *faulted* gehen kann, nachdem sie eine Nachricht empfangen hat, d.h. sie kann *faulted* nicht erreichen, während sie im Zustand *executing* auf die Nachricht wartet. Folglich besteht eine Kontrollflussabhängigkeit und damit die Zustandsabhängigkeiten, dass die Receive-Instanz nur in *completed* oder *faulted* wechseln kann, nachdem die Instanz des Invokes den Zustand *completed* und damit davor auch *executing* erreicht hat. Geht die Invoke-Instanz also in den Zustand *completed*, kann die Instanz des Receives danach den Zustand *completed* oder *faulted* erreichen:

$$\forall v_i \in \{executing, completed\} \forall v_j \in \{completed, faulted\} : \\ \omega_\Upsilon(snd, v_i, rcv, v_j) \coloneqq \rightarrow \qquad (Ml_{InvRcv}1)$$

Sendet die Invoke-Instanz die Nachricht, bevor die Instanz des Receives den Zustand *executing* erreicht hat, wird die Nachricht in einem unbegrenzt großen Speicher zwischengespeichert [Loh10] und die Aktivität empfängt diese, sobald sie in den Zustand *executing* wechselt.

Erreicht eine Invoke-Instanz einen anderen Endzustand als *completed*, beispielsweise *faulted* oder *dead*, wird keine Nachricht versandt und die Instanz des Receives kann den Zustand *completed* nicht erreichen. Dies würde allerdings bedeuten, dass die Instanz des Receives im Zustand *executing* „hängen" würde. Da eine Voraussetzung für die Prozesskonsolidierung eine Deadlockfreie Choreographie ist (siehe Abschnitt 5.2), muss diese so modelliert sein, dass die Receive-Instanz in diesem Fall entweder gar nicht erst ausgeführt oder terminiert wird. Es kann also angenommen werden, dass das Receive immer in die Zustände *dead*, *aborted* oder *terminated* geht, wenn das Invoke einen anderen Endzustand als *completed* erreicht:

		rcv						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
snd	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	⊕ $Ml_{InvRcv}2$	⊕ $Ml_{InvRcv}2$
	ABORTED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	⊕ $Ml_{InvRcv}2$	⊕ $Ml_{InvRcv}2$
	EXECUTING	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	→ $Ml_{InvRcv}1$	→ $Ml_{InvRcv}1$
	TERMINATED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	⊕ $Ml_{InvRcv}2$	⊕ $Ml_{InvRcv}2$
	FAULTED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	⊕ $Ml_{InvRcv}2$	⊕ $Ml_{InvRcv}2$
	COMPLETED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	→ $Ml_{InvRcv}1$	→ $Ml_{InvRcv}1$

Abbildung 4.38.: Zustandstransitionsprofil zwischen einer über eine Nachrichtenkante verbundene Invoke-Aktivität *snd* und Receive-Aktivität *rcv*

$$\begin{aligned} &\forall v_i \in \{dead, aborted, terminated, faulted\} \\ &\quad \forall v_j \in \{completed, faulted\} : \qquad (Ml_{InvRcv}2) \\ &\quad\quad \omega_{\Upsilon}(snd, v_i, rcv, v_j) := \oplus \end{aligned}$$

Da hier nur asynchrone „fire and forget“ Interaktionen [BDtH05] zwischen Invoke- und Receive-Aktivitäten betrachtet werden, ist der Zustand der Receive-Instanz für die Instanz des Invokes transparent, d.h. der Zustand der Receive-Instanz hat keinen Einfluss auf den Zustand der Invoke-Instanz. Erreicht das Receive also den Zustand *dead* oder wird es terminiert, geschieht dies komplett unabhängig vom jeweiligen Zustand des Invokes (Axiom *Iso* aus Abschnitt 4.2.7). Das Invoke ist daher vollständig isoliert vom Receive und das Receive ist fehlerisoliert vom Invoke. Aus den Axiomen *Iso*, $Ml_{InvRcv}1$ und $Ml_{InvRcv}2$ resultiert das Profil in Abbildung 4.38.

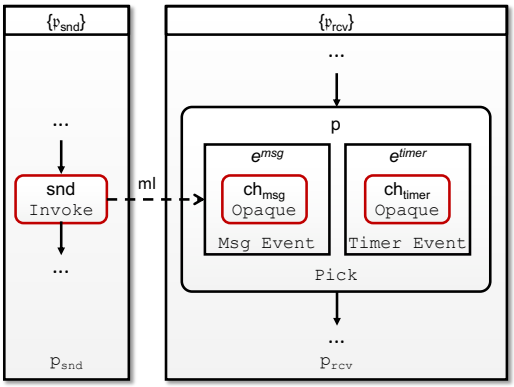

Abbildung 4.39.: Pick-Aktivität die über ein Nachrichtenereignis e^{msg} mit der Invoke-Aktivität *snd* verbunden ist.

4.2.15. Ereignisbasierte Verzweigung

Bei der ereignisbasierten Verzweigung (Workflow-Muster „Deferred Choice“) wird einer von mehreren alternativen Pfaden abhängig von einem bestimmten Ereignis aktiviert. Es wird der Pfad ausgeführt, bei dem das ihm zugeordnete Ereignis zuerst eintritt. Mittels des Metamodells kann die ereignisbasierte Verzweigung über eine Pick-Aktivität p ($p \in A_{\texttt{pick}}$) modelliert werden, die mehrere Kindaktivitäten $A_{event} = \textsc{Children}(p)$ besitzt. Dabei ist jede Kindaktivität einem Timer- oder Nachrichtenereignis zugeordnet. Während der Ausführung der Pick-Aktivität kann genau eines der Ereignisse ausgelöst und die Instanz der zugehörigen Kindaktivität ausgeführt werden. Eine Pick-Aktivität mit zwei Ereignisaktivitäten ch_{msg} und ch_{timer}, die jeweils dem Nachrichtenereignis e^{msg} und Timer-Ereignis e^{timer} zugeordnet sind, ist in Abbildung 4.39 dargestellt. Das Nachrichtenereignis e^{msg} wird ausgelöst, wenn die Instanz eines Invokes *snd* über die Nachrichtenkante *ml* eine Nachricht sendet und das Timer-Ereignis nicht zuvor ausgelöst wurde. Über das Invoke und das empfangende Nachrichtenereignis wird also ebenfalls eine asynchrone Interaktion realisiert.

In Abschnitt 4.2.15.1 werden die Zustandsbeziehungen zwischen den Kind-

aktivitäten erläutert. Die Beziehungen zwischen dem Invoke *snd* und der Kindaktivität ch_{msg}, die über ein Nachrichtenereignis und einer Nachrichtenkante mit dem Invoke verbunden ist, werden in Abschnitt 4.2.15.2 diskutiert.

4.2.15.1. Kindaktivitäten einer Pick-Aktivität

Da es sich beim Pick um eine strukturierte Aktivität handelt, gelten zwischen seinen Kindaktivitäten die Axiome $St_{CH2CH}1 - St_{CH2CH}4$. Eingeschränkt werden diese Axiome dadurch, dass während der Ausführung der Pick-Instanz nur genau ein Ereignis ausgelöst werden kann. Damit kann auch nur genau eine Instanz der Kindaktivitäten in den Zustand *executing* und dessen Nachfolgezustände wechseln:

$$\forall ch_i \neq ch_j \in A_{event}\ \forall \upsilon_i\ \upsilon_j \in \Upsilon \setminus \{initial, dead, aborted\} : \\ \omega_\Upsilon(ch_i, \upsilon_i, ch_j, \upsilon_j) := \oplus \qquad (Pick_{Event})$$

Die Instanzen der Ereignisaktivitäten, die nicht ausgelöst wurden, werden in den Zustand *dead* gesetzt. Dabei ist nicht vorgegeben, in welcher Reihenfolge die Aktivitäten den Zustand *dead* bzw. *executing* erreichen müssen. Dies ist bereits durch das Axiom $St_{CH2CH}1$ definiert.

Das daraus resultierende Zustandsprofil zwischen den Kindaktivitäten ch_i und ch_j einer Pick-Aktivität ist in Abbildung 4.40 dargestellt. Da das Axiom $Pick_{Event}$ dem Axiom Xor_{split} der exklusiven Verzweigung und dem Axiom Sc_{FH2FH} der alternativen Fault-Handler entspricht, das ebenfalls die Axiome $St_{CH2CH}1 - St_{CH2CH}4$ einschränkt, gleicht das Profil dem der exklusiven Verzweigung in Abbildung 4.26 und dem der Fehlerbehandlungsaktivitäten in Abbildung 4.14. Die Axiome definieren nur die Ordnung zwischen den Zuständen von ch_i und ch_j, nicht aber den genauen Zeitpunkt, zu dem die Zustände erreicht werden müssen, da dies abhängig von der Implementierung der Workflow-Engine ist, auf der ein Prozess ausgeführt wird. Es ist deshalb zum Beispiel auch hier möglich, dass eine Instanz von ch_i in *executing* gesetzt

ch_i		ch_j						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$
	ABORTED	← Chor2	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	← $St_{CH2CH}2$	← $St_{CH2CH}4$
	EXECUTING	← Chor2	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	⊕ $Pick_{Event}$	⊕ $Pick_{Event}$	⊕ $Pick_{Event}$	⊕ $Pick_{Event}$
	TERMINATED	← Chor2	← $St_{CH2CH}4$	‖ $St_{CH2CH}3$	⊕ $Pick_{Event}$	⊕ $Pick_{Event}$	⊕ $Pick_{Event}$	⊕ $Pick_{Event}$
	FAULTED	← Chor2	← $St_{CH2CH}4$	→ $St_{CH2CH}2$	⊕ $Pick_{Event}$	⊕ $Pick_{Event}$	⊕ $Pick_{Event}$	⊕ $Pick_{Event}$
	COMPLETED	← Chor2	‖ $St_{CH2CH}1$	→ $St_{CH2CH}4$	⊕ $Pick_{Event}$	⊕ $Pick_{Event}$	⊕ $Pick_{Event}$	⊕ $Pick_{Event}$

Abbildung 4.40.: Zustandstransitionsprofil zwischen den Kindaktivitäten ch_i und ch_j einer Pick-Aktivität

wird und die Instanz des Eltern-Picks direkt danach terminiert wird, ohne dass die Instanz von ch_j vorher den Zustand *dead* erreichen konnte.

4.2.15.2. Asynchrone Interaktion zwischen Invoke-Aktivität und einer einem Nachrichtenereignis zugeordneten Kindaktivität

Hier werden die Zustandsbeziehungen zwischen dem Invoke *snd* und der Kindaktivität ch_{msg} beschrieben, die dem Nachrichtenereignis e^{msg} zugeordnet ist, mit dem das Invoke über eine Nachrichtenkante verbunden ist. Wie in Abschnitt 4.2.14 erläutert, hat die Instanz des Invokes eine Nachricht erfolgreich versandt, wenn es den Zustand *completed* erreicht. Ist die Instanz des Picks im Zustand *executing*, kann das Nachrichtenereignis ausgelöst werden und die Instanz von ch_{msg} kann in den Zustand *executing* und dessen Nachfolgezustände gehen. Die Instanz von ch_{msg} erreicht den Zustand *executing* also nachdem die Instanz von *snd* den Zustand *completed* und somit davor den Zustand *executing* erreicht hat:

$$\forall v_{snd} \in \{executing, completed\}$$
$$\forall v_{exec} \in \Upsilon \setminus \{initial, dead, aborted\} : \qquad (Ml_{InvPck}1)$$
$$\omega_{\Upsilon}(snd, v_{snd}, ch_{msg}, v_{exec}) := \rightarrow$$

Sendet die Invoke-Instanz die Nachricht, bevor die Instanz des Picks den Zustand *executing* erreicht hat, wird die Nachricht zwischengespeichert und das Nachrichtenereignis wird ausgelöst, sobald die Pick-Instanz in den Zustand *executing* wechselt.

Geht die Instanz des Invokes in einen anderen Endzustand als *completed*, kann das Nachrichtenereignis nicht ausgelöst werden und die Instanz von ch_{msg} kann den Zustand *executing* nicht erreichen. Um auch hier die Voraussetzung der Deadlock-freien Choreographie zu erfüllen, muss ein anderes Nachrichten- oder Timer-Ereignis ausgelöst werden. Alternativ kann die Instanz der Pick-Aktivität entweder in den Zustand *dead* gehen oder sie muss terminiert werden.

$$\forall v_{noSnd} \in \{dead, aborted, terminated, faulted\}$$
$$\forall v_{exec} \in \Upsilon \setminus \{initial, dead, aborted\} : \qquad (Ml_{InvPck}2)$$
$$\omega_{\Upsilon}(snd, v_{noSnd}, ch_{msg}, v_{exec}) := \oplus$$

Analog zur Isolation von Invoke- und Receive-Aktivitäten in verschiedene Teilnehmer sind auch hier das Invoke *snd* und das Ereignis ch_{msg} voneinander durch die Teilnehmerzugehörigkeit isoliert. Der Zustand der Instanz des Picks und seiner Kindaktivität ch_{msg} hat keinen Einfluss auf den Zustand der Invoke-Instanz, d.h. das Invoke ist vollständig isoliert von beiden Aktivitäten. Daher resultiert das Profil zwischen dem Invoke *snd* und ch_{msg}, das in Abbildung 4.41 abgebildet ist, aus den Axiomen *Iso*, $Ml_{InvPck}1$ und $Ml_{InvPck}2$.

		ch_{msg}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
snd	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ Iso	‖ Iso	⊕ $MI_{InvPck}2$	⊕ $MI_{InvPck}2$	⊕ $MI_{InvPck}2$	⊕ $MI_{InvPck}2$
	ABORTED	← Chor2	‖ Iso	‖ Iso	⊕ $MI_{InvPck}2$	⊕ $MI_{InvPck}2$	⊕ $MI_{InvPck}2$	⊕ $MI_{InvPck}2$
	EXECUTING	← Chor2	‖ Iso	‖ Iso	→ $MI_{InvPck}1$	→ $MI_{InvPck}1$	→ $MI_{InvPck}1$	→ $MI_{InvPck}1$
	TERMINATED	← Chor2	‖ Iso	‖ Iso	⊕ $MI_{InvPck}2$	⊕ $MI_{InvPck}2$	⊕ $MI_{InvPck}2$	⊕ $MI_{InvPck}2$
	FAULTED	← Chor2	‖ Iso	‖ Iso	⊕ $MI_{InvPck}2$	⊕ $MI_{InvPck}2$	⊕ $MI_{InvPck}2$	⊕ $MI_{InvPck}2$
	COMPLETED	← Chor2	‖ Iso	‖ Iso	→ $MI_{InvPck}1$	→ $MI_{InvPck}1$	→ $MI_{InvPck}1$	→ $MI_{InvPck}1$

Abbildung 4.41.: Zustandstransitionsprofil zwischen einem Invoke *snd* und der Kindaktivität ch_{msg}

4.2.16. Schleifen

Die iterative Ausführung einer Menge von direkten und indirekten Kindaktivitäten einer Schleife $a_{loop} \in A_{\texttt{loop}}$ (Workflow-Muster „Structured Loop“) kann mittels einer While- oder wie im Beispiel in Abbildung 4.42 einer ForEach-Aktivität modelliert werden. Von den Kindaktivitäten, im Folgenden auch als *iterierbare Aktivitäten* $A_{it} = \textsc{Descendants}(a_{loop})$ bezeichnet, werden in jeder Konversation keine (falls die Schleifenbedingung nie wahr und somit keine Iteration ausgeführt wird), eine oder mehrere Instanzen erstellt. Die genaue Anzahl von Iterationen einer Schleife kann zur Entwurfszeit weder für While- noch für ForEach-Schleifen bestimmt werden (siehe Abschnitt 3.2.4.10). Folglich ist es auch nicht möglich, die Anzahl der Instanzen der iterierbaren Aktivitäten zu bestimmen. In Abschnitt 4.2.2 wurde allerdings festgelegt, dass sich alle Aktivitätsinstanzen zur Laufzeit in einem bestimmten Zustand befinden müssen. Um dieser Anforderung weiter zu entsprechen und damit eine Ordnung über die Aktivitätszustände abbilden zu können, wird von der Anzahl der Iterationen und somit auch von der Anzahl der Instanzen abstrahiert. Es wird also weiterhin davon ausgegangen, dass alle Instanzen der iterierbaren Aktivitäten einer Konversation bekannt sind und in den Zustand

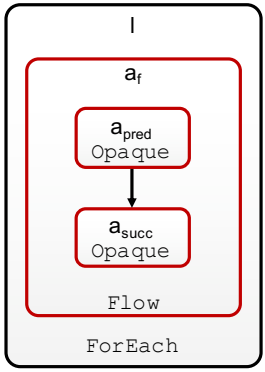

Abbildung 4.42.: ForEach-Aktivität mit Kindaktivitäten

initial gesetzt werden, bevor eine Aktivitätsinstanz der Konversation in einen anderen Zustand wechseln kann (Axiom *Chor2*). Dies ist ein Unterschied zur technischen Umsetzung von Schleifen in BPEL Workflow-Engines, bei denen die Instanzen der Aktivitäten erst erzeugt werden, wenn die jeweilige Iteration ausgeführt wird. Die Folgezustände von *initial* der Instanzen der iterierbaren Aktivitäten werden entsprechend der BPEL-Spezifikation gesetzt, d.h. abhängig vom Zustand der Schleifeninstanz und der Iteration zu der die Instanz zugeordnet ist. Die Zuordnung einer Instanz zu einer Iteration erfolgt dabei über eine hochgestellte Nummer. Die Instanz $a_{pred}^{I,1}$ ist zum Beispiel der ersten und $a_{pred}^{I,2}$ der zweiten Iteration einer Schleife zugeordnet.

Betrachtet man allerdings die Zustandsbeziehungen zwischen den Kindaktivitäten über alle Iterationen hinweg, gelten diese Beziehungen zwischen den regulären Aktivitätszuständen nicht mehr, da die Instanzen im regulären Kontrollfluss während jeder Iteration wieder von *initial* jeden möglichen Zustand erreichen können. Für die sequentielle Ausführung der Quellaktivität a_{pred} und der Zielaktivität a_{succ} ist in Abschnitt 4.2.9 zum Beispiel definiert, dass a_{succ} nicht mehr den Zustand *executing* erreichen kann, nachdem a_{pred} den Zustand *dead* erreicht hat. Befinden sich a_{pred} und a_{succ} in einer Schleife, gilt

also in der jeweiligen Iteration $\omega_{\Upsilon}(a_{pred}, dead, a_{succ}, executing) \coloneqq \oplus$. Betrachtet man die Zustandsbeziehung zwischen a_{pred} und a_{succ} über alle Iterationen der Schleife, gilt aber $\omega_{\Upsilon}(a_{pred}, dead, a_{succ}, executing) \coloneqq \|$. Eine Reihenfolge zwischen den Zustandstransitionen der verschiedenen Instanzen existiert zwar dadurch, dass diese unterschiedlichen Iterationen zugewiesen sind und somit nacheinander ausgeführt werden. Im Beispiel in Abbildung 4.42 erreicht die Instanz $a_{succ}^{I,2}$ den Zustand *completed* erst, nachdem die Instanz von $a_{succ}^{I,1}$ *dead* erreicht hat. Da die Aktivitätszustandsbeziehungen nicht zwischen Instanzen unterscheiden, kann diese durch die Iterationen implizierte Zustandstransitionsreihenfolge nicht abgebildet werden. Daher existiert auf Ebene der Zustandsbeziehungen keine Reihenfolge, in der die iterierbaren Aktivitäten den Zustand des regulären Kontrollflusses erreichen müssen. Dies wird durch das Axiom $Lp_{CH2CH}1$ abgebildet, das die Beziehungen zwischen den regulären Zuständen der Kindaktivitäten durch die Beziehung $\|$ ersetzt:

$$\begin{gathered} \forall ch_i \neq ch_j \in A_{it}\ \forall \upsilon_i, \upsilon_j \in \Upsilon_{reg} : \\ \omega_{\Upsilon}(ch_i, \upsilon_i, ch_j, \upsilon_j) \coloneqq \| \end{gathered} \qquad (Lp_{CH2CH}1)$$

Löst eine Instanz der iterierbaren Aktivitäten einen Fehler aus, wird dieser danach zu den Instanzen aller ihrer Vorfahren weiter propagiert. Diese Instanzen erreichen dann ebenfalls den Zustand *faulted*. Alle anderen Instanzen, inklusive denen, die nachfolgenden Iterationen zugeordnet sind, werden terminiert. Löst zum Beispiel die Instanz der Aktivität a_{pred} aus Abbildung 4.42 während der ersten Iteration einen Fehler aus, wird die Instanz von $a_f^{I,1}$ danach in den Zustand *faulted* gesetzt. Die Instanzen von a_f, a_{pred} und a_{succ} der nachfolgenden Iterationen werden in den Zustand *aborted* gesetzt. Dies ist in Axiom $Lp_{CH2CH}2$ definiert:

$$\begin{aligned} &\forall ch_i, ch_j \in A_{it} \forall v_j \in \{aborted, terminated\} : \\ &\quad \omega_\Upsilon(ch_i, faulted, ch_j, faulted) := \| \\ &\wedge\, \omega_\Upsilon(ch_i, faulted, ch_j, v_j) := \rightarrow \end{aligned} \qquad (Lp_{CH2CH}2)$$

Die Beziehung $\omega_\Upsilon(ch_i, faulted, ch_j, faulted) := \|$ resultiert daraus, dass die Menge A_{it} keine Hierarchiebeziehungen berücksichtigt. Daher gilt, wenn ch_j ein Vorfahre von ch_i ist, dass eine Instanz von ch_j in *faulted* geht, nachdem die Instanz von ch_i *faulted* erreicht hat. Ist die Aktivität ch_i ein Nachfahre von ch_j, gehen ihre Instanzen in den Zustand *faulted*, bevor die Instanzen von ch_j den Zustand *faulted* erreicht haben.

Eine (vorzeitige) Terminierung einer Instanz der iterierbaren Aktivitäten resultiert entweder aus der Terminierung der Schleife oder, wie oben diskutiert, aus einem Fehler, der von einer Instanz der iterierbarer Aktivitäten ausgelöst wurde. Im ersten Fall werden alle Instanzen der iterierbaren Kindaktivitäten terminiert, wobei auch hier wie innerhalb der anderen strukturierten Aktivitäten keine Terminierungsreihenfolge zwischen den Instanzen vorgegeben ist, unabhängig davon, welcher Iteration diese zugeordnet sind. Im zweiten Fall werden alle Instanzen terminiert, nachdem die fehlerhafte Instanz in den Zustand *faulted* gegangen ist. Die Terminierungsreihenfolgen der Instanzen werden durch das folgende Axiom abgebildet:

$$\begin{aligned} &\forall ch_i, ch_j \in A_{it}\ \forall v_i, v_j \in \{aborted, terminated\} : \\ &\quad \omega_\Upsilon(ch_i, v_i, ch_j, v_j) := \| \end{aligned} \qquad (Lp_{CH2CH}3)$$

Analog zu den anderen strukturierten Aktivitäten können in jeder Iteration die Instanzen der Kindaktivitäten der Schleife nur so lange einen regulären Zustand erreichen, bis eine der Instanzen einen Fehlerzustand erreicht hat:

		ch_j						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
ch_i	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ $Lp_{CH2CH}1$	→ $Lp_{CH2CH}4$	‖ $Lp_{CH2CH}1$	→ $Lp_{CH2CH}4$	→ $Lp_{CH2CH}4$	‖ $Lp_{CH2CH}1$
	ABORTED	← Chor2	← $Lp_{CH2CH}4$	‖ $Lp_{CH2CH}3$	← $Lp_{CH2CH}4$	‖ $Lp_{CH2CH}3$	← $Lp_{CH2CH}3$	← $Lp_{CH2CH}4$
	EXECUTING	← Chor2	‖ $Lp_{CH2CH}1$	→ $Lp_{CH2CH}4$	‖ $Lp_{CH2CH}2$	→ $Lp_{CH2CH}4$	→ $Lp_{CH2CH}4$	‖ $Lp_{CH2CH}1$
	TERMINATED	← Chor2	← $Lp_{CH2CH}4$	‖ $Lp_{CH2CH}3$	← $Lp_{CH2CH}4$	‖ $Lp_{CH2CH}3$	← $Lp_{CH2CH}3$	← $St_{CH2CH}4$
	FAULTED	← Chor2	← $Lp_{CH2CH}4$	→ $Lp_{CH2CH}2$	← $Lp_{CH2CH}4$	→ $Lp_{CH2CH}2$	‖ $Lp_{CH2CH}2$	← $St_{CH2CH}4$
	COMPLETED	← Chor2	‖ $Lp_{CH2CH}1$	→ $Lp_{CH2CH}4$	‖ $Lp_{CH2CH}1$	→ $Lp_{CH2CH}2$	→ $St_{CH2CH}4$	‖ $St_{CH2CH}1$

Abbildung 4.43.: Zustandstransitionsprofil zwischen den iterierbaren Kindaktivitäten ch_i und ch_j einer Schleife

$$\forall ch_i, ch_j \in A_{it}\ \forall \upsilon_i \in \Upsilon_{reg}\ \forall \upsilon_{error} \in \Upsilon_{error} : \\ \omega_{\Upsilon}(ch_i, \upsilon_i, ch_j, \upsilon_{error}) := \rightarrow \qquad (Lp_{CH2CH}4)$$

Das Profil, das aus den Axiomen folgt, ist in Abbildung 4.43 dargestellt.

4.2.16.1. Sendende Schleifen

Eine sendende Schleife, wie zum Beispiel fe_{snd} in Abbildung 4.44, implementiert den Nachrichtenversand an mehrere Empfänger. Die Sendeaktivität *snd* interagiert mit einer Receive-Aktivität *rcv*, die den Empfängern zugeordnet ist und somit zur Laufzeit ebenfalls mehrfach instanziiert wird. Während jeder Iteration sendet eine Instanz von *snd* eine Nachricht an einen anderen Empfänger. Aufgrund der Mehrfachinstanziierung gelten zwischen *snd* und *rcv* andere Zustandsbeziehungen als bei Invoke-Receive-Interaktionen.

Jede Instanz von *snd*, die im Zustand *completed* beendet wurde, sendet eine Nachricht an die Receive-Instanz eines bestimmten Teilnehmers. Die Receive-Instanz kann dann entweder den Zustand *completed* oder *faulted*

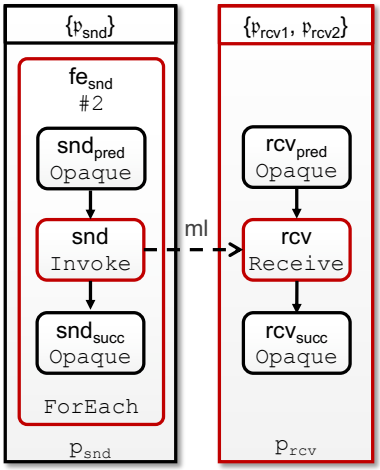

Abbildung 4.44.: Sendende ForEach-Schleife

erreichen (Axiom $Ml_{InvRcv}1$). Endet die Ausführung einer Instanz von *snd* in *dead*, können in den nächsten Iterationen die anderen Instanzen von *snd* trotzdem weiter Nachrichten senden. Wie eingangs diskutiert, geht in den Aktivitätszustandsbeziehungen die Unterscheidung zwischen Instanzen verloren. Sendet zum Beispiel die erste Instanz von *snd* eine Nachricht an $\mathfrak{p}_{rcv1}$, kann die empfangende *rcv*-Instanz von $\mathfrak{p}_{rcv1}$ den Zustand *completed* erreichen, bevor die *snd*-Instanz der zweiten Iteration in *executing* oder *completed* geht. Daher existiert, wie auch bei den Kindaktivitäten einer Schleife (Axiom $Lp_{CH2CH}1$), keine Reihenfolge zwischen den Zuständen *dead*, *executing* bzw. *completed* von *snd* und *completed* bzw. *faulted* von *rcv*. Das Axiom $Lp_{SND}1$ wird also anstelle von Axiom $Ml_{InvRcv}1$ verwendet, wenn sich *snd* in einer Schleife befindet:

$$\forall v_i \in \{dead, executing, completed\} \forall v_j \in \{completed, faulted\} : \\ \omega_\gamma(snd, v_i, rcv, v_j) := \| \qquad (Lp_{SND}1)$$

Wird die Iteration der Schleife im Fehlerfall gestoppt, kann keine Nachricht

mehr gesendet werden und die Instanz von *rcv* kann weder in den Zustand *completed* noch *faulted* gehen. Daher können die Instanzen von *rcv* diese Zustände nur erreichen, so lange eine Instanz von *snd* keinen Fehlerzustand erreicht hat. Das Axiom $Lp_{SND}2$ ersetzt also Axiom $Ml_{InvRcv}2$.

$$\forall \upsilon_i \in \{aborted, terminated, faulted\} \forall \upsilon_j \in \{completed, faulted\} : \\ \omega_\gamma(snd, \upsilon_i, rcv, \upsilon_j) := \leftarrow \qquad (Lp_{SND}2)$$

Das aus Axiom $Lp_{SND}1$ und $Lp_{SND}2$ resultierende Profil zwischen *snd* und *rcv* ist in Abbildung 4.45 dargestellt. Durch die Isolation der von *snd* und *rcv* gilt zwischen den anderen Zuständen das Axiom *Iso*.

		rcv						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
snd	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso
	ABORTED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	← $Lp_{SND}2$	← $Lp_{SND}2$
	EXECUTING	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ Iso
	TERMINATED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	← $Lp_{SND}2$	← $Lp_{SND}2$
	FAULTED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	← $Lp_{SND}2$	← $Lp_{SND}2$
	COMPLETED	← Chor2	‖ $Lp_{SND}1$	‖ Iso	‖ Iso	‖ Iso	‖ $Lp_{SND}1$	‖ $Lp_{SND}1$

Abbildung 4.45.: Zustandstransitionsprofil zwischen der Aktivität *snd* einer sendenden Schleife und der Receive-Aktivität *rcv* eines empfangenden Teilnehmers

4.2.16.2. Empfangende Schleifen

Eine empfangende Schleife enthält eine Receive-Aktivität *rcv*, die Nachrichten von verschiedenen Sendern empfängt, die diese zur Laufzeit über unterschiedliche Instanzen der gleichen Invoke-Aktivität *snd* übertragen. Ein Beispiel für eine empfangende Schleife ist in Abbildung 4.46 dargestellt.

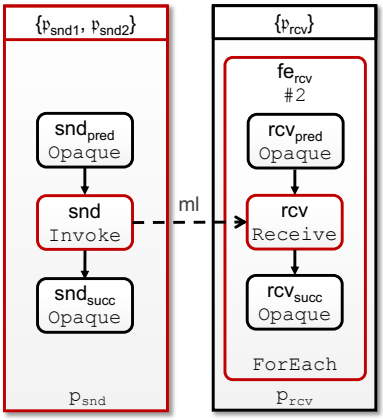

Abbildung 4.46.: Empfangende ForEach-Schleife

Durch die iterative Ausführung von *rcv* empfangen deren Instanzen mehrere Nachrichten und können damit in *completed* oder *faulted* gehen, bevor eine *snd*-Instanz aus einer späteren Iteration *executing* erreicht hat. Die Instanz $a^{I,2}_{snd}$ kann zum Beispiel abhängig von der Ausführungsdauer von $a^{I,1}_{snd_{succ}}$ bzw. $a^{I,2}_{snd_{pred}}$ *executing* erst erreichen, nachdem die Instanz $a^{I,1}_{rcv}$ in *completed* gegangen ist. Daher gilt für *snd* und *rcv* bei empfangenden Schleifen das Axiom $Lp_{Rcv}1$ anstatt Axiom $Ml_{InvRcv}1$.

$$\begin{aligned} &\forall \upsilon \in \{executing, completed\} : \\ &\quad \omega_\Upsilon(snd, \upsilon, rcv, completed) := \| \qquad (Lp_{Rcv}1) \\ &\wedge\, \omega_\Upsilon(snd, \upsilon, rcv, faulted) := \| \end{aligned}$$

Kann keine Nachricht gesendet werden, weil eine Instanz von *snd* in den Zustand *dead* oder einen Fehlerzustand geht, muss die zugehörige Instanz von *rcv* ebenfalls in einen Endzustand gesetzt werden, so dass kein Deadlock auftritt (Axiom $Ml_{InvRcv}2$). Dazu kann die Receive-Instanz entweder terminiert oder in den Zustand *dead* gesetzt werden. Im zweiten Fall wird die Schleife nicht beendet und es ist weiterhin möglich, dass die nächste Instanz von

rcv eine Nachricht empfängt. Dies ist im Axiom $Lp_{Rcv}2$ definiert, das das Axiom $Ml_{InvRcv}2$ ersetzt, wenn sich *rcv* in einer Schleife befindet.

$$\begin{aligned} &\forall \upsilon_{noSnd} \in \{dead, aborted, terminated, faulted\} : \\ &\quad \omega_{\Upsilon}(snd, \upsilon_{noSnd}, rcv, completed) := \| \\ &\quad \wedge\, \omega_{\Upsilon}(snd, \upsilon_{noSnd}, rcv, faulted) := \| \end{aligned} \qquad (Lp_{Rcv}2)$$

Abbildung 4.47 zeigt das aus den Axiomen $Lp_{Rcv}1$ und $Lp_{Rcv}2$ resultierende Zustandsprofil.

		rcv						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
snd	INITIAL	‖ Chor1	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2	→ Chor2
	DEAD	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ $Lp_{RCV}2$	‖ $Lp_{RCV}2$
	ABORTED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ $Lp_{RCV}2$	‖ $Lp_{RCV}2$
	EXECUTING	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ $Lp_{RCV}1$	‖ $Lp_{RCV}1$
	TERMINATED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ $Lp_{RCV}2$	‖ $Lp_{RCV}2$
	FAULTED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ $Lp_{RCV}2$	‖ $Lp_{RCV}2$
	COMPLETED	← Chor2	‖ Iso	‖ Iso	‖ Iso	‖ Iso	‖ $Lp_{RCV}1$	‖ $Lp_{RCV}1$

Abbildung 4.47.: Zustandstransitionsprofil zwischen der Invoke-Aktivität *snd* der sendenden Teilnehmer und der Receive-Aktivität *rcv* in der empfangenden Schleife

Das Metamodell erlaubt ebenfalls die Modellierung von empfangenden Schleifen, die anstatt eines Receives ein Pick mit entsprechenden Nachrichtenereignissen enthalten. Dieses Modellierungsszenario wird hier nicht weiter betrachtet, da sich alle Interaktionsmuster mit empfangenden Schleifen umsetzen lassen, die Receive-Aktivitäten enthalten (z.B. das Muster „One-from-many Receive“ in Abschnitt 7.2.3).

Da eine sendende Schleife Nachrichten an eine Menge von Teilnehmern sendet und eine empfangende Schleife Nachrichten von einer Menge von Teilnehmern empfängt, können sendende und empfangende Schleifen nicht über

eine Nachrichtenkante verbunden werden. Anders ausgedrückt, kann sich immer nur entweder die Sende- oder Empfangsaktivität einer Nachrichtenkante in einer ForEach-Schleife befinden. Folglich existieren keine Zustandsbeziehungen zwischen sendenden und empfangenden Schleifen.

4.3. Verhaltensäquivalenz von Choreographien

Intuitiv werden in dieser Arbeit zwei Choreographien aus Kontrollflusssicht als verhaltensäquivalent angesehen, wenn sie dieselben Geschäftsfunktionen (zum Beispiel die Kommunikation mit externen Services, Berechnungen etc.) in der gleichen Reihenfolge ausführen können. Diese Funktionen werden dabei durch Geschäftsaktivitäten $A_{\texttt{opaque}}$ implementiert. Implementieren zwei Geschäftsaktivitäten a_i und a_j dieselbe Funktionalität, kann dies über die in Abschnitt 3.2.4.1 eingeführte Relation $a_i \equiv_A a_j$ ausgedrückt werden.

Formal ist die Verhaltensäquivalenz von Choreographien (die Definition schließt Choreographiefragmente mit ein) wie folgt definiert:

Definition 4.4 (Verhaltensäquivalenz von Choreographien)
Es gilt $c_i \approx_{\texttt{task}} c_j$, d.h. zwei Choreographien c_i und c_j sind verhaltensäquivalent, wenn sie die folgenden Eigenschaften erfüllen:

1. *Sie besitzen identische Geschäftsaktivitäten A_i und A_j:*
   $\forall a_i \in A_i \exists! a_j \in A_j : a_i \equiv_A a_j \wedge \forall a_j \in A_j \exists! a_i \in A_i : a_j \equiv_A a_i$
   $$\textit{mit } A_i = \bigcup_{p \in \pi_4(c_i)} \pi_1(p) \cap A_{\texttt{opaque}} \textit{ und } A_j = \bigcup_{p \in \pi_4(c_j)} \pi_1(p) \cap A_{\texttt{opaque}}$$
2. *Die Choreographien generieren die gleiche Geschäftsaktivitätshistorie, d.h. die Aktivitäten aus c_i erzeugen dieselben Aktivitätszustands-Traces wie die Aktivitäten aus c_j:*
   $\hat{T}_{A_{\texttt{opaque}}} \cap \mathsf{history_c}(c_i) = \mathsf{history_c}(c_j) \cap \hat{T}_{A_{\texttt{opaque}}}$
   $$\textit{mit } \hat{T}_{A_{\texttt{opaque}}} = \{\hat{\tau} \in \hat{T} \mid \pi_1(\hat{\tau}) \in \bigcup_{a \in A_{\texttt{opaque}}} \mathsf{inst}_A(a)\}$$
   - *Die Körper von identischen Schleifen a_{lp_i} und a_{lp_j} müssen zusätzlich*

ohne Berücksichtigung ihrer iterativen Ausführung dieselbe Historie generieren:

$$\forall a_{lp_i} \equiv_A a_{lp_j} \in (A_i \cup A_j) \cap A_{\texttt{loop}} :$$
$$\hat{T}_{A_{\text{opaque}}} \cap \text{history}_{\mathsf{a}}(a_{ch_i}) = \text{history}_{\mathsf{a}}(a_{ch_j}) \cap \hat{T}_{A_{\text{opaque}}}$$
$$\textit{mit } a_{ch_i} \in \textsc{Children}(a_{lp_i}) \wedge a_{ch_j} \in \textsc{Children}(a_{lp_j})$$

Die Traces der Aktivitäten, die die Kommunikation zwischen den Teilnehmern umsetzen, d.h. Kommunikationsaktivitäten und auch deren Vorfahren (zum Beispiel kommunizierende Schleifen), werden bei der Äquivalenz also nicht berücksichtigt. Formal ist dies durch den Schnitt der Historien mit den Traces der Geschäftsaktivitäten $\hat{T}_{A_{\text{opaque}}}$ realisiert. Dies ist darin begründet, dass verhaltensäquivalente Choreographien die Orchestrierung der identischen Geschäftsaktivitäten mittels unterschiedlichen Kommunikationsszenarien oder auch Teilnehmern realisieren können.

Wie in Abschnitt 4.2.16 erläutert, besteht aufgrund der repetitiven Ausführung der Aktivitäten im Körper einer Schleife zwischen deren regulären Zuständen immer die Beziehung ||, unabhängig davon, wie der Kontrollfluss in der jeweiligen Schleife modelliert wurde. Die Körper von Schleifen generieren daher immer dieselbe Historie. Folglich können die Unterschiede im Verhalten von identischen Schleifen a_{lp_i} und a_{lp_j} nicht identifiziert werden. Um zu prüfen, ob die Schleifenkörper wirklich dasselbe Verhalten implementieren, müssen daher die Historien der Kindaktivitäten von a_{lp_i} und a_{lp_j} verglichen werden, ohne deren iterative Ausführung zu berücksichtigen. Dies wird dadurch erreicht, dass nur die Historien der direkten Kindaktivitäten (und damit auch von ihren Nachfahren) von a_{lp_i} bzw. a_{lp_j} isoliert bestimmt und verglichen werden. Bei der Ermittlung der Historie werden also die durch a_{lp_i} bzw. a_{lp_j} implizierten Zustandsbeziehungen ignoriert.

Ob eine Choreographie c_i mit den Geschäftsaktivitäten A_i dieselbe Geschäftsaktivitätshistorie hat, wie eine Choreographie c_j mit den Geschäftsaktivitäten A_j kann über Aktivitätszustandsprofile geprüft werden:

$$\forall a_l, a_r \in A_i \forall id_l, id_r \in A_j :$$
$$a_l \equiv_A id_l \wedge a_r \equiv_A id_r \Rightarrow \mathsf{profile}(a_l, a_r) = \mathsf{profile}(id_l, id_r)$$

Dazu wird das Aktivitätszustandsprofil zwischen jedem Paar von Geschäftsaktivitäten aus A_i mit dem Profil verglichen, das zwischen dem zu diesem Paar identischen Aktivitäten aus A_j gilt. Die Profile zwischen allen Geschäftsaktivitäten einer Choreographie repräsentieren dabei die komplette Historie durch die Ordnung zwischen deren Zuständen.

Analog dazu kann mittels der Profile überprüft werden, ob die Historien der Schleifenkörper von a_{lp_i} und a_{lp_j} äquivalent sind. In diesem Fall gilt $A_i = \textsc{Descendants}(a_{lp_i})$ und $A_j = \textsc{Descendants}(a_{lp_j})$. Wie oben erwähnt, dürfen bei der Erstellung der Profile zwischen den Aktivitäten in A_i bzw. A_j die von a_{lp_i} bzw. a_{lp_j} implizierten Zustandsbeziehungen nicht berücksichtigt werden, um die iterative Ausführung der Aktivitäten auszuklammern.

van Glabbeek definiert in [vGla90] 11 verschiedene Klassen, um die Verhaltensäquivalenz von Transitionensystemen oder Prozessen, bestehend aus Zuständen und Aktionen zu beschreiben. Die restriktivste Klasse ist die *Bisimulation* von Milner [Mil83] und die allgemeinste Klasse ist die von Hoare definierte *Trace-Äquivalenz* [Hoa85]. Ein Prozess q simuliert einen Prozess p, wenn es zu jeden Zustand in p einen Zustand in q gibt, der dieselben Aktionen ausführen kann. Bisimulation liegt vor, wenn q p und p q simuliert. Zwei Prozesse sind trace-äquivalent, wenn sie die gleiche Folge von Aktionen ausführen können. Die Trace-Äquivalenz vergleicht im Gegensatz zur Bisimulation also nur das beobachtbare Verhalten von zwei Prozessen, nicht aber deren internes Verhalten, d.h. von welchen Zuständen die Aktionen ausgelöst werden können. Die *vollständige Trace-Äquivalenz* (engl. completed trace equivalent) [vGla90] ist restriktiver als die Trace-Äquivalenz. Dort wird geprüft, ob alle vollständigen Traces die Prozesse erzeugen können, d.h. die Traces inklusive Anfangs- und Endzustände, äquivalent sind.

Sind c_i und c_j vollständig Trace-äquivalent zueinander, gilt $c_i \approx_{\texttt{task}} c_j$, da

dann beide Choreographien dieselben Profile haben und das Profil einer Choreographie die Repräsentation aller ihrer erlaubten Aktivitätszustands-Traces (inklusive denen der Geschäftsaktivitäten) ist. Umgekehrt impliziert $c_i \approx_{\texttt{task}} c_j$ aber nicht, dass die beiden Choreographien vollständig Trace-Äquivalent sind, weil die Verhaltensäquivalenz von c_i und c_j nur das Verhalten der Geschäftsaktivitäten berücksichtigt.

Die Profile abstrahieren vom Datenfluss, der wiederum den Kontrollfluss beeinflusst. Daher kann damit nicht ausgedrückt werden, ob sich zwei Choreographien gleich verhalten, wenn ihre Konversationen den gleichen Datenkontext besitzen. Die Axiome der exklusiven Verzweigung (Abschnitt 4.2.11) definieren zum Beispiel nur, dass eine der Instanzen der alternativen Aktivitäten in den Zustand *executing* gehen muss. Die Information, welche Instanz dies abhängig von den Transitionsbedingungen und den Variablenwerten ist, wird nicht abgebildet. Existieren also in c_i und c_j identische Verzweigungsaktivitäten, die ebenfalls mit identischen alternativen Aktivitäten verbunden sind, ist es möglich, dass die Verzweigungsbedingungen in beiden Choreographien unterschiedlich formuliert sind. Die Profile wären in diesem Fall äquivalent, aber es würden, auch wenn beide Fragmente den gleichen Datenkontext haben, unterschiedliche Instanzen der Verzweigungsaktivitäten ausgeführt werden. Die Bedingung die zu einem bestimmten Aktivitätszustand führt, ist aber, wie oben erwähnt, die Voraussetzung dafür, zu entscheiden, ob zwei Transitionensysteme bisimilar sind. Folglich kann aufgrund des Fehlens der Bedingungen in den Profilen nicht festgestellt werden, ob zwischen zwei Choreographien die restriktivste Äquivalenzklasse, also die Bisimulation, gilt.

4.4. Zusammenfassung

In diesem Kapitel wurde erläutert, wie sich das Verhalten von Choreographien über ihre Historie, d.h. der Menge der von ihr zur Laufzeit generierbaren Aktivitätszustands-Traces, beschreiben lässt. Ein Trace bildet die

Reihenfolge der Zustandstransitionen der Aktivitätsinstanzen ab, die an der Ausführung einer von der Choreographie erstellten Konversation beteiligt waren.

Da die Historie das Verhalten einer Choreographie beschreibt, kann durch den Vergleich von ihren Historien überprüft werden, ob zwei Choreographien äquivalent sind, d.h., ob sie das gleiche Verhalten modellieren. Dazu wurde als Äquivalenzkriterium definiert, dass zwei Choreographien dasselbe Verhalten haben, wenn deren Geschäftsaktivitäten die gleiche Historie erzeugen. Dieses Äquivalenzkriterium wird in den folgenden Kapiteln genutzt, um zu prüfen, inwieweit die Konsolidierungsoperation Prozessmodelle erzeugt[1], deren Geschäftsaktivitäten die gleiche Historie haben wie die jeweilige Choreographie, aus der sie erstellt wurden.

Die Historie einer Choreographie kann deklarativ mittels einer Menge von Aktivitätszustandsbeziehungen beschrieben werden. Diese Zustandsbeziehungen definieren die erlaubte temporale Ordnung zwischen den Zustandstransitionen der Aktivitätsinstanzen in den Traces. Die Beziehungen können für Aktivitäten mit direkter Kontroll- oder Nachrichtenflussabhängigkeit anhand der in diesem Kapitel vorgestellten Axiome bestimmt werden. Jedes Axiom definiert die Zustandsbeziehungen für die Kontroll- bzw. Nachrichtenflusskonstrukte des Metamodells basierend auf ihrer operationalen Semantik. Die Zustandsbeziehungen von Aktivitäten, die eine indirekte Abhängigkeit im Kontrollfluss haben, können transitiv über die Anwendung der Axiome bestimmt werden. Ein formaler Algorithmus wurde dafür nicht vorgestellt, da in den folgenden Kapiteln die Konsolidierungsschritte an Choreographie- und Prozessmodellfragmenten erläutert werden, bei denen die Zustandsbeziehungen zwischen Aktivitäten bestimmt werden müssen, die indirekt über maximal vier Aktivitäten Kontrollflussbeziehungen zueinander haben. Aufgrund der geringen Anzahl der Aktivitäten kann die transitive Anwendung der Axiome textuell beschrieben werden.

[1] Das konsolidierte Prozessmodell kann dabei als Choreographie mit nur einer Verhaltensbeschreibung aufgefasst werden.

Kapitel 5

Konsolidierung von interagierenden BPEL-Prozessmodellen

In diesem Kapitel wird die Konsolidierungsoperation diskutiert, mit der die Verhaltensbeschreibungen der Teilnehmer einer Choreographie c in ein einzelnes *konsolidiertes Prozessmodell* p_μ vereinigt werden. Ziel der Konsolidierungsoperation ist es, dass das konsolidierte Prozessmodell das Teilnehmerverhalten der Choreographie ohne Nachrichtenübertragung emuliert. Dazu muss das Prozessmodell die folgenden Eigenschaften erfüllen:

i. Die Choreographie c und p_μ müssen verhaltensäquivalent sein. Die Verhaltensäquivalenz mit c kann über eine Choreographie c_μ mit einem Teilnehmer $\mathfrak{p}_\mu$ ermittelt werden, der von p_μ implementiert wird:

$$c \approx_{\texttt{task}} \overbrace{(\{\mathfrak{p}_\mu\}, \emptyset, \emptyset, p_\mu)}^{c_\mu} \text{ mit } \mathsf{type}_{\mathfrak{p}}(\mathfrak{p}_\mu) = p_\mu$$

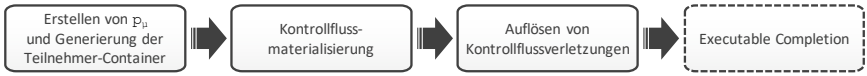

Abbildung 5.1.: Schritte der Konsolidierungsoperation

ii. Es darf sich nicht selbst Nachrichten zuschicken, d.h. es darf kein Nachrichtenfluss implementiert werden, bei dem sich die sendende und die empfangende Aktivität im konsolidierten Prozessmodell befindet.

iii. Der originale Datenfluss zwischen den Geschäftsaktivitäten muss erhalten bleiben. Dies schließt auch den in der Choreographie durch Nachrichtenübertragung modellierten Datenfluss mit ein.

Die Konsolidierungsoperation erhält als Eingabe eine Choreographie. Um daraus das konsolidierte Prozessmodell p_μ zu erstellen, das die oben genannten Eigenschaften erfüllt, implementiert die Operation die in Abbildung 5.1 dargestellten Schritte.

Im ersten Schritt wird das Prozessmodell p_μ erstellt. In p_μ wird für jeden Teilnehmer ein *Teilnehmer-Container* hinzugefügt. Dies ist ein Scope, der den Kontrollfluss der Verhaltensbeschreibung des jeweiligen Teilnehmers enthält. Der Scope dient dazu, die Teilnehmer wie in der Choreographie voneinander zu isolieren, so dass zur Laufzeit Fehler in einem Teilnehmer nicht an einen anderen propagiert werden.

Die Teilnehmer beeinflussen ihr Verhalten in der Choreographie nur durch die zwischen ihnen ausgetauschten Nachrichten. Im zweiten Schritt, der *Kontrollflussmaterialisierung*, wird der Kontroll- und Datenfluss in und zwischen den Teilnehmer-Containern so angepasst, dass in p_μ der Nachrichtenfluss ohne Kommunikationsaktivitäten emuliert wird. Mit welchen Kontrollflusskonstrukten die Emulation des Nachrichtenaustausches realisiert wird, hängt dabei von den in der Choreographie modellierten Interaktionsmustern ab. Dabei impliziert jedes Muster andere Kontroll- und Datenflussabhängigkeiten zwischen den Aktivitäten aus unterschiedlichen Teilnehmern.

Die Kontrollflussmaterialisierung kann bezüglich der BPEL-Syntax bzw. des Metamodells ungültige Kontrollflusskanten erzeugen, zum Beispiel Kanten, deren Quellaktivität innerhalb und deren Zielaktivität außerhalb der einer Schleife liegt. Diese Kontrollflussverletzungen werden im dritten Schritt, *Auflösen von Kontrollflussverletzungen*, behoben, wobei die besondere Herausforderung darin besteht, dass die durch die Kontrollflusskanten implizierten Kontrollflussabhängigkeiten erhalten bleiben.

Um das konsolidierte Prozessmodell auf einer Workflow-Engine auszuführen, müssen unter Umständen weitere Aktivitäten eingefügt werden. Dies ist zum Beispiel der Fall, wenn die Geschäftsaktivitäten, um die interne Struktur eines Prozessmodells nicht öffentlich sichtbar zu machen, als abstrakte Platzhalter fungieren, die vor der Ausführung durch konkrete Aktivitäten ersetzt werden. Zusätzlich müssen die zur Ausführung benötigten technischen Informationen, wie zum Beispiel WSDLs, Service-Bindings [CLS+05] und Workflow-Engine-spezifische Deployment Deskriptoren hinzugefügt werden. Die Aktivitäten und die technischen Informationen werden im Schritt *Executable Completion* vervollständigt. Dieser Schritt wurde bereits von Kopp in [Kop16a] diskutiert und ist für die eigentliche Konsolidierung nicht relevant. Er wird daher in dieser Ausarbeitung nicht weiter diskutiert.

Dieses Kapitel gliedert sich wie folgt. In Abschnitt 5.1 wird eine Beispielchoreographie vorgestellt und in Abschnitt 5.2 werden die Annahmen, Ziele und Einschränkungen der Konsolidierungsoperation konkreter formuliert. Die Erstellung von p_μ und der Teilnehmer-Container wird in Abschnitt 5.3 beschrieben. Die Materialisierung wird in Abschnitt 5.4 und Abschnitt 5.5 diskutiert. Das Auflösen von Kontrollflussverletzungen wird im separaten Kapitel 6 detailliert erläutert.

5.1. Illustrationsszenario

Als Illustrationsszenario für die Konsolidierungsschritte wird die in Abbildung 5.2 dargestellte Choreographie $c_{Flugbuchung}$ benutzt. Die Choreographie

deckt eine Reihe von Interaktionsmustern ab, auf die im Verlauf von diesem und dem nächsten Kapitel genauer eingegangen wird.

Der Kunde, der durch den Teilnehmer $\mathfrak{p}_{Kunde}$ repräsentiert wird, plant zuerst seine Reise und möchte den günstigsten Flug zu seinem Reiseziel suchen. Der Teilnehmer sendet eine Nachricht mit dem gewünschten Reisedatum und dem Zielort an den Teilnehmer $\mathfrak{p}_{Reisebuero}$. Nach dem Empfang dieser Nachricht wählt das Reisebüro zuerst die Fluggesellschaften aus, die Flüge an den Zielort zu dem angegebenen Datum anbieten. An jede infrage kommende Fluggesellschaft wird dann eine Nachricht mit einer Preisanfrage für die Flugdaten gesendet. In der Choreographie stehen dazu zur Vereinfachung nur die Gesellschaften $\mathfrak{p}_{LH}$ und $\mathfrak{p}_{Swiss}$ zur Verfügung, die dieselbe Verhaltensbeschreibung besitzen. Jede Gesellschaft bestimmt den Preis und sendet diesen in einer Antwortnachricht zurück an das Reisebüro. Für den Versand der Antwortnachricht haben die Fluggesellschaften eine Minute Zeit, danach wird die Ausführung der Schleife über einen Fehler beendet. Dieser Fehler wird von der Throw-Aktivität im Event-Handler des Scopes ausgelöst, in dem sich die Schleife befindet. Der Fault-Handler fängt diesen Fehler, so dass der Teilnehmer $\mathfrak{p}_{Reisebuero}$ weiter ausgeführt werden kann. Nach dem Abbruch der Schleife wird die Fluggesellschaft mit dem günstigsten Preis ermittelt und mit der Ticketerstellung beauftragt. Die Pick-Aktivität stellt sicher, dass die Fluggesellschaft die Beauftragung empfängt und die anderen Fluggesellschaften nach einer bestimmten Zeitspanne ihr Angebot verwerfen. Das Ticket wird dann von der beauftragten Fluggesellschaft an den Kunden gesendet. Die Rechnung empfängt dieser vom Reisebüro.

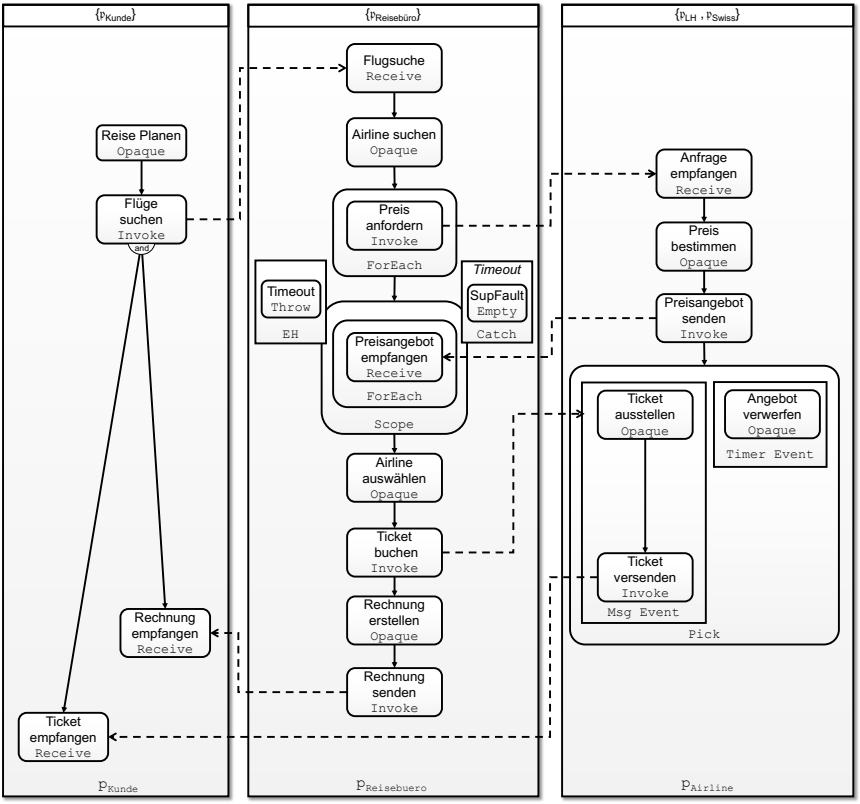

Abbildung 5.2.: Aus [Kop16a] adaptierte Beispielchoreographie $c_{Flugbuchung}$

5.2. Konsolidierungsoperation

Formal erhält die Konsolidierungsoperation μ als Eingabe eine Choreographie c_{input} und erstellt aus dieser das Prozessmodell p_μ mit den am Anfang dieses Kapitels definierten Eigenschaften:

$$\mu : C \rightarrow P$$

Die Konsolidierungsoperation wird durch Algorithmus 5.1 implementiert. Algorithmus 5.1 ruft die Funktionen bzw. Prozeduren auf, die die einzelnen Konsolidierungsschritte implementieren. Diese werden im weiteren Verlauf dieses und des nächsten Kapitels beschrieben.

Algorithmus 5.1 Konsolidierungsoperation

```
1: p_μ                                              ▷ das konsolidierte Prozessmodell
2: procedure μ(c)
3:     p_μ ← CreateParticipantContainer(c)
4:     ML_c ← π_3(c)
5:     Materialization(ML_c)
6:     L_{p_μ} ← π_3(p_μ)
7:     ResolveControlFlowViolations(L_{p_μ})        ▷ Siehe Kapitel 6
8: end procedure
```

Um Fallunterscheidungen in den Konsolidierungsschritten zu minimieren, muss die zu konsolidierende Choreographie c und jede von einer Choreographie erstellte Konversation $K \in \text{conversations}(c)$ die folgenden Voraussetzungen erfüllen:

- Die Fault-Handler der Prozess-Scopes fangen alle Fehler der Kind-Scopes. Es werden also keine Fehler weiter propagiert.
- Um die Verifikation der Prozessmaterialisierung zu vereinfachen, dürfen Kommunikationsaktivitäten maximal eine Vorgänger- oder Nachfolgeaktivität besitzen.
- Um Deadlocks zu vermeiden, muss jede Instanz einer Empfangsaktivität einen Endzustand erreichen können, auch wenn zur Laufzeit einer Choreographie an sie keine Nachricht gesendet wird (siehe Abschnitt 5.4.1.2).
- Für den Datenfluss gilt das Bernstein Kriterium [Bae73], d.h. es dürfen keine Aktivitäten parallel lesend und schreibend bzw. schreibend und schreibend auf ein Element des Datenkontextes zugreifen.

Bei der Verifikation der Kontrollflussmaterialisierung in Abschnitt 5.4 werden

unter anderem die Zustandsbeziehungen zwischen den direkten Vorgänger- und Nachfolgeaktivitäten der Kommunikationsaktivitäten untersucht. Die Einschränkung, dass diese nur einen direkten Vorgänger bzw. Nachfolger haben dürfen, reduziert bei der Verifikation die Fallunterscheidungen. Ansonsten müssten zum Beispiel die Zustandsbeziehungen betrachtet werden, die gelten, wenn es sich bei den Vorgängern um parallele und exklusive Aktivitäten handelt. Dieselben Fallunterscheidungen müssten auch für die Nachfolgeaktivitäten gemacht werden als auch zwischen Vorgängern und Nachfolgern.

5.3. Generierung der Teilnehmer-Container

In diesem Konsolidierungsschritt wird das konsolidierte Prozessmodell p_μ erstellt. Dabei muss p_μ nach diesem Schritt die folgenden Eigenschaften der Choreographie widerspiegeln:

i. Das Verhalten jedes potentiellen Teilnehmers der Choreographie ist in p_μ modelliert.

ii. Die Isolation zwischen den Teilnehmern bleibt in p_μ erhalten.

Der Erhalt der ersten Eigenschaft stellt sicher, dass die Instanzen von p_μ die Geschäftsaktivitätszustandstraces jedes Teilnehmers generieren können. Bezogen auf das Beispielszenario in Abbildung 5.2 sind das die Traces der Geschäftsaktivitäten der vier Teilnehmer $\mathfrak{p}_{Kunde}$, $\mathfrak{p}_{Reisebuero}$, $\mathfrak{p}_{LH}$ und $\mathfrak{p}_{Swiss}$.

Mit der zweiten Eigenschaft soll erreicht werden, dass die Zustandstransitionen der Aktivitätsinstanzen eines Teilnehmers nicht direkt[1] die Zustände der Aktivitätsinstanzen eines anderen Teilnehmers beeinflussen. So darf zum Beispiel ein Fehler während der Emulation des Teilnehmerverhaltens von $\mathfrak{p}_{Swiss}$ nicht dazu führen, dass die Aktivitätsinstanzen, die die anderen Teilnehmer emulieren, terminiert werden.

[1] Die Zustände der Teilnehmer beeinflussen sich indirekt über deren Interaktionen miteinander, die aber erst in Abschnitt 5.4 berücksichtigt werden.

Algorithmus 5.2 Erstellen von $p_\mu$und Hinzufügen des Kontrollflusses der zu konsolidierenden Teilnehmerprozesse zu p_μ

1: **function** CREATEPARTICIPANTCONTAINER(c)
2: $p_\mu = new$ `process`
3: $a_{flow} \leftarrow$ CREATEACTIVITY(`flow`)
4: $\mathfrak{p}_c^{set} = \pi_1(c)$
5: **for all** $\mathfrak{p} \in \mathfrak{p}_c^{set}$ **do**
6: $p_\mathfrak{p} \leftarrow$ DUPLICATEPROCESSMODEL($\text{type}_\mathfrak{p}(\mathfrak{p}), \mathfrak{p}$)
7: $\pi_1(p_\mu) \leftarrow \pi_1(p_\mu) \cup \pi_1(p_\mathfrak{p})$
8: $\pi_2(p_\mu) \leftarrow \pi_2(p_\mu) \cup \pi_2(p_\mathfrak{p})$
9: ...
10: $\pi_8(p_\mu) \leftarrow \pi_8(p_\mu) \cup \pi_8(p_\mathfrak{p})$
11: $s_\mathfrak{p} \leftarrow$ CREATEACTIVITY(`scope`)
12: participantScope($\mathfrak{p}$) $\leftarrow s_\mathfrak{p}$
13: ADDCHILDACTIVITIES($a_{flow}, \{s_\mathfrak{p}\}$)
14: $HR_p := \{hr \in HR \mid \pi_1(hr) = p_\mathfrak{p}\}$
15: **for all** $hr_p \in HR_p$ **do**
16: CREATEHR($s_\mathfrak{p}, \pi_2(hr_p), \pi_3(hr_p)$)
17: **end for**
18: **if** $\nexists hr_{catchAll} \in HR_{p^-} : \pi_1(hr_{catchAll}) = s_\mathfrak{p}$ $\wedge \pi_2(hr) \in E^{faultStd}$ **then**
19: $e^{fault} = new$ `Event`
20: $\text{type}_E(e^{fault}) \leftarrow t_E^{faultStd}$
21: $a_{noOp} =$ CREATEACTIVITY(`empty`)
22: CREATEHR($s_\mathfrak{p}, e^{fault}, a_{noOp}$)
23: **end if**
24: **end for**
25: **return** p_μ
26: **end function**

Dieser Konsolidierungsschritt ist durch Algorithmus 5.2 implementiert. Das Prozessmodell p_μ wird in Zeile 2 erstellt, dabei ist jedem Element des Tupels von p_μ die leere Menge zugeordnet. Für jeden potentiellen Teilnehmer wird ein *Teilnehmer-Scope* $s_\mathfrak{p}$ in der Aktivität a_{flow} in p_μ erstellt, dem der Kontrollfluss des jeweiligen Teilnehmerprozessmodells zugewiesen wird. Mit der Abbildung $\mathsf{participantScope} : \mathfrak{P} \rightarrow A_{\mathtt{scope}}$ wird einem Teilnehmer sein Teilnehmer-Scope in p_μ zugewiesen. Diese Zuordnung wird in den weiteren Konsolidierungsschritten benötigt.

Da ein Prozessmodell das Verhalten von mehreren Teilnehmern beschreiben kann, wie zum Beispiel das Prozessmodell $p_{Airline}$, wird dazu zunächst in Zeile 6 für jeden Teilnehmer ein Duplikat $p_\mathfrak{p}$ des Prozessmodells mittels der Funktion DuplicateProcessmodel erstellt (siehe Anhang A.3). Alle Elemente des Duplikats (Aktivitäten, Hierarchiebeziehungen, Variablen, Kontrollflusskanten etc.) werden zu p_μ hinzugefügt (Zeile 7–10).

Die primäre Kindaktivität des duplizierten Prozessmodells $p_\mathfrak{p}$ und seine Event- und Fault-Handler werden zum Teilnehmer-Scope $s_\mathfrak{p}$ transferiert. Dazu werden in Zeile 16 neue Hierarchiebeziehungen erstellt, in denen der Teilnehmer-Scope die Eltern-Aktivität der direkten Kind-Aktivitäten von $p_\mathfrak{p}$ wird. Folglich werden auch die indirekten Kind-Aktivitäten von $p_\mathfrak{p}$ indirekte Kindaktivitäten des Teilnehmer-Scopes. Die oben erwähnte Isolation der Aktivitäten der verschiedenen Teilnehmer wird dadurch erreicht, dass dem Teilnehmer-Scope ein Fault-Handler zugewiesen ist, der alle Fehler fängt, die nicht von den anderen Fault-Handlern des Scopes verarbeitet werden können. Dieser Fault-Handler wird, falls er nicht schon vom Teilnehmerprozessmodell transferiert wurde[1], in den Zeilen 19 - 22 erstellt. Die Aktivität a_{noOp} dient dazu, den Fehler zu unterdrücken. Die Erstellung dieser Aktivität sowie das Hinzufügen zum Fault-Handler und zu p_μ wird durch die Funktionen CreateActivity bzw. CreateHR realisiert (Anhang A.1).

Der Kontrollfluss des Prozessmodells p_μ, das bei Anwendung des Algorithmus

[1] Wie in den Annahmen in Abschnitt 5.2 beschrieben, dürfen die Fault-Handler eines Prozess-Scopes keine Rethrow-Aktivität enthalten.

auf die Flugbuchungschoreographie $c_{Flugbuchung}$ erstellt wurde, ist in Abbildung 5.3 dargestellt. In p_μ wurde für jeden Teilnehmer aus $c_{Flugbuchung}$ ein Teilnehmer-Scope erstellt, wobei die Teilnehmer-Scopes s_{LH} und s_{Swiss} auf Basis der Verhaltensbeschreibung $p_{Airline}$ generiert wurden.

5.3.1. Verhaltensäquivalenz von c und p_μ

Der Algorithmus stellt sicher, dass das Verhalten jedes Teilnehmers (Eigenschaft (i)) abgebildet ist, indem für jeden Teilnehmer der Choreographie der Kontrollfluss, der in seiner Verhaltensbeschreibung spezifiziert ist, als Duplikat in p_μ einfügt wird. Das heißt, die Aktivitäten und die Kontrollflusskanten sowie die Hierarchiebeziehungen zwischen den Aktivitäten bleiben unverändert erhalten. Der Datenfluss bleibt ebenfalls erhalten, da die Funktion DuplicateProcessmodel auch ein Duplikat der Variablen der Verhaltensbeschreibung erstellt und die Variablenduplikate den entsprechenden Aktivitätsduplikaten in den Teilnehmer-Scopes zuweist.

In diesem Konsolidierungsschritt werden die Nachrichtenkanten der Choreographie nicht berücksichtigt. Daher sind die Instanzen der Aktivitäten der Teilnehmer der Choreographie, wie in Abschnitt 4.2.7 beschrieben, vollständig isoliert voneinander. Es gilt also zwischen allen Aktivitäten aus c, die unterschiedlichen Teilnehmern zugeordnet sind, das in Abbildung 4.16 dargestellte Profil.

In p_μ enthalten die Teilnehmer-Scopes die direkten und indirekten Kindaktivitäten jedes emulierten Teilnehmers und fangen alle potentiellen Fehler, die zur Laufzeit von deren Instanzen ausgelöst werden können. Zusätzlich sorgt die Parallelität der Teilnehmer-Scopes in a_{flow} dafür, dass deren Kindaktivitäten keine Synchronisationsabhängigkeiten zueinander haben. Folglich sind auch in p_μ die Aktivitäten aus verschiedenen Teilnehmern vollständig isoliert voneinander und es gilt zwischen ihnen ebenfalls das Profil aus Abbildung 4.16. Eigenschaft (ii) bleibt also erhalten.

Zur Laufzeit wird durch die durch das Flow a_{flow} modellierte parallele Verzwei-

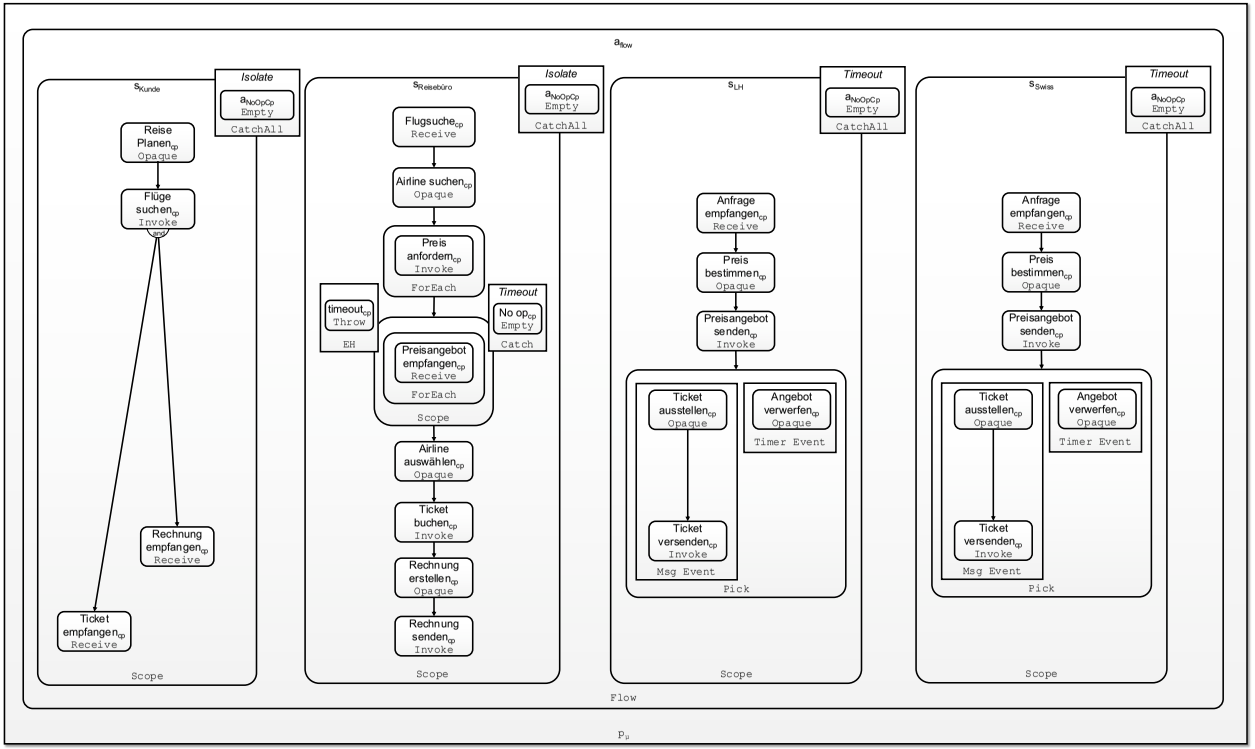

Abbildung 5.3.: Das aus $c_{Flugbuchung}$ durch Algorithmus 5.2 generierte Prozessmodell p_μ mit seinen Teilnehmer-Containern

gung in p_μ der Teilnehmer-Scope jedes potentiellen Teilnehmers ausgeführt, selbst wenn die Instanz der Teilnehmeraktivitäten zur Laufzeit nicht ausgeführt werden. Stellt zum Beispiel die Aktivität *Airline suchen* des Teilnehmers $\mathfrak{p}_{Reisebuero}$ fest, dass eine bestimmte Flugroute nur von $\mathfrak{p}_{LH}$ angeboten wird, kommen der Choreographie die Aktivitätsinstanzen des Teilnehmers $\mathfrak{p}_{Swiss}$ nicht zur Ausführung, da dieser keine Nachricht von Teilnehmer $\mathfrak{p}_{Reisebuero}$ empfangen würde. In einer Instanz von p_μ würde der Teilnehmer-Scope, der den Teilnehmer $\mathfrak{p}_{Swiss}$ repräsentiert, trotzdem ausgeführt werden, nicht aber seine Geschäftsaktivitäten. Dies wird durch die Kontrollflussmaterialisierung sichergestellt, die in Abschnitt 5.4 beschrieben wird.

5.3.2. Einschränkungen bezüglich der Teilnehmerermittlung zur Laufzeit

Die Erstellung von p_μ und die Generierung der Teilnehmer-Scopes findet zur Entwurfszeit statt. Aus diesem Grund müssen, wie in Abschnitt 3.1 definiert, alle potentiellen Teilnehmer der Choreographie zur Entwurfszeit bekannt sein. BPEL und BPEL4Chor erlauben allerdings auch die Modellierung von Choreographien, deren Teilnehmer erst zur Laufzeit bestimmt werden. Diese Choreographien lassen sich folglich nicht mit dem oben vorgestellten Algorithmus konsolidieren. In [WKL14] stellen Wagner, Kopp und Leymann einen Ansatz vor, der es ermöglicht, auch Choreographien zu konsolidieren, deren Teilnehmer erst zur Laufzeit bestimmt werden. Wie ebenfalls in [WKL14] diskutiert, schränkt der Ansatz die Isolation zwischen den Aktivitäten in p_μ ein, die aus verschiedenen Teilnehmern stammen und wird deshalb hier nicht weiter diskutiert.

5.4. Kontrollflussmaterialisierung

Im vorherigen Schritt wurden die Teilnehmer-Scopes inklusive Kontrollflussgraphen und Variablen der Teilnehmer zu p_μ hinzugefügt. Die Teilnehmer-Scopes und deren Kindaktivitäten sind vollständig voneinander isoliert und enthalten weiterhin Kommunikationsaktivitäten. Ein Ansatz die Isolation zwi-

schen den Aktivitäten in den Teilnehmer-Scopes aufzuheben, wäre es, dass diese wieder miteinander kommunizieren, indem eine Aktivität Nachrichten an eine andere in p_μ sendet. Dies widerspräche allerdings dem Ziel der Konsolidierung, dass innerhalb von p_μ keine Kommunikation stattfinden soll und ist laut BPEL-Spezifikation unzulässig. Daher müssen die Kommunikationsaktivitäten entfernt und dabei sichergestellt werden, dass die Daten, die in der Choreographie zwischen den Aktivitäten der Teilnehmer über Nachrichten ausgetauscht wurden, ebenfalls in p_μ zwischen den Aktivitäten der Teilnehmer-Scopes ausgetauscht werden. Die Kontrollflussmaterialisierung verfolgt daher die folgenden drei Ziele:

i. Eliminierung der Kommunikationsaktivitäten

ii. Emulation des Nachrichtenflusses

iii. Verknüpfung der Kontrollflussgraphen der verschiedenen Teilnehmer-Scopes

Wie der Nachrichtenfluss emuliert und die Kontrollflussgraphen in p_μ verknüpft werden, hängt dabei von den in der Choreographie modellierten Interaktionen ab, da jede Interaktion eigene Daten- und Kontrollflussabhängigkeiten zwischen den Teilnehmern impliziert. Die komplexen Interaktionen, die in Kapitel 7 beschrieben sind, können durch eine Menge von Nachrichtenkanten sowie weiteren Kontrollflusskonstrukten, wie beispielsweise Schleifen, modelliert werden.

Mittels einer Nachrichtenkante kann entweder die Interaktion zwischen einer Invoke- und einer Receive-Aktivität oder die Interaktion zwischen einer Invoke-Aktivität und einem Nachrichtenereignis modelliert werden (siehe Abschnitt 3.1.2). Deren Materialisierung wird in Abschnitt 5.4.1 bzw. Abschnitt 5.5 im Detail erläutert. Im Folgenden werden diese Interaktionen als *Basisinteraktionen* bezeichnet, da sie als Grundlage für alle anderen Interaktionen dienen. Die Materialisierung der Basisinteraktionen ist daher, wie in Kapitel 7 erläutert, die Voraussetzung für die Materialisierung von komplexen Interaktionen.

Die Materialisierung der Basisinteraktionen wird von der Prozedur MATERIALIZATION in Algorithmus 5.3 angestoßen. Die Prozedur iteriert über jede Nachrichtenkante der Choreographie und überprüft, ob es sich um eine Invoke-Receive- oder Invoke-Pick-Interaktion handelt und ruft die entsprechenden Prozeduren MATERIALIZEINVOKERECEIVE bzw. MATERIALIZEINVOKEPICK auf, um die Interaktionen zu materialisieren. Eine Nachrichtenkante kann mehrere sendende oder mehrere empfangende Teilnehmer miteinander verbinden. Daher iterieren die zwei inneren Schleifen über alle sendenden bzw. empfangenden Teilnehmer und ermitteln in p_μ die Sende- bzw. Empfangsaktivität im jeweiligen Teilnehmer-Scope. Dies geschieht in den Zeilen 5 und 6, wo anhand der originalen Kommunikationsaktivitäten die jeweiligen Duplikate in den Teilnehmer-Scopes bestimmt werden. Die Abbildung actCopy ist in Anhang A.3 definiert.

Algorithmus 5.3 Kontrollflussmaterialisierung

```
 1: procedure MATERIALIZATION(ML_c)
 2:     for all ml ∈ ML_c do
 3:         for all 𝔭_snd ∈ π_1(ml) do
 4:             for all 𝔭_rcv ∈ π_3(ml) do
 5:                 snd ← actCopy(𝔭_snd, ⊥, π_2(ml))
 6:                 co_rcv ← actCopy(𝔭_rcv, ⊥, π_4(ml))
 7:                 if co_rcv ∈ A_receive then
 8:                     MATERIALIZEINVOKERECEIVE(𝔭_snd, snd, 𝔭_rcv, co_rcv)
 9:                 else
10:                     MATERIALIZEINVOKEMSGEVENT(𝔭_snd, snd, 𝔭_rcv, co_rcv)
11:                 end if
12:             end for
13:         end for
14:     end for
15: end procedure
```

5.4.1. Materialisierung von Invoke-Receive-Basisinteraktionen

In diesem Abschnitt wird zuerst erläutert, wie Invoke- Receive-Interaktionen materialisiert werden können und darauf aufbauend wird der zugehörige

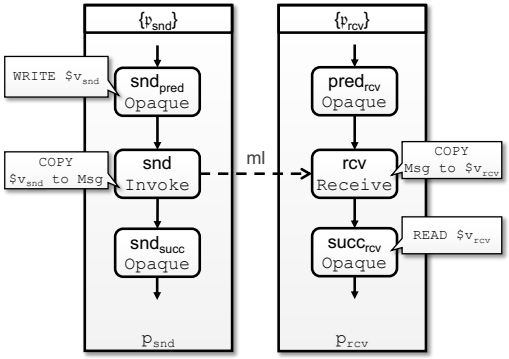

Abbildung 5.4.: Invoke-Receive-Basisinteraktion

Algorithmus für die Materialisierung beschrieben. Außerdem wird diskutiert, inwieweit p_μ durch den Materialisierungsansatz verhaltensäquivalent zur originalen Choreographie bleibt.

Eine Invoke-Receive-Interaktion wird, wie im Beispiel in Abbildung 5.4 dargestellt[1], mittels eines Invoke *snd* und eines Receive *rcv* modelliert, die über eine Nachrichtenkante *ml* miteinander verbunden sind. Das Invoke befindet sich im sendenden Teilnehmer $\mathfrak{p}_{snd}$ und das Receive im empfangenden Teilnehmer $\mathfrak{p}_{rcv}$. Formal wird die Interaktion also durch ein Choreographiefragment $c^F = (\mathfrak{P}, \mathfrak{P}^{set}, ML, P)$ implementiert, das die folgenden Eigenschaften erfüllt:

1. $\mathfrak{P} := \{\mathfrak{p}_{snd}, \mathfrak{p}_{rcv}\}$
2. $\mathfrak{P}^{set} := \{\{\mathfrak{p}_{snd}\}, \{\mathfrak{p}_{rcv}\}\}$
3. $P := \{p_{snd}, p_{rcv}\}$
   a) $\text{type}_{\mathfrak{p}}(\mathfrak{p}_{snd}) = p_{snd}$
   b) $\text{type}_{\mathfrak{p}}(\mathfrak{p}_{rcv}) = p_{rcv}$

[1] Die Flow-Aktivitäten, in denen die Kontrollflussgraphen modelliert sind, werden hier und in den folgenden Abbildungen aus Gründen der Übersichtlichkeit nur noch dargestellt, wenn dies für das Verständnis erforderlich ist.

4. $ML := \{ml\}$ mit $ml = (\{\mathfrak{p}_{snd}\}, snd, \{\mathfrak{p}_{rcv}\}, rcv)$
5. $A := \{snd, rcv\}$ mit $snd \in \pi_1(p_{snd}) \wedge rcv \in \pi_1(p_{rcv})$
   a) $snd \in A_{\texttt{invoke}} \wedge \mathsf{receiver}(snd) = (\mathfrak{p}_{rcv})$
   b) $rcv \in A_{\texttt{receive}} \wedge \mathsf{sender}(rcv) = (\mathfrak{p}_{snd})$
6. $V := \{v_{snd}, v_{rcv}\}$ mit $v_{snd} \in \pi_4(p_{snd}) \wedge v_{rcv} \in \pi_4(p_{rcv})$
   a) $\mathsf{inputVar}(snd) = v_{snd}$
   b) $\mathsf{outputVar}(rcv) = v_{rcv}$

Zur Laufzeit kopiert die Instanz des Invokes *snd* den Inhalt der Eingabevariable in die Nachricht und sendet diese zu einer Instanz von *rcv*, die diese in die Ausgabevariable kopiert. Auf die Variable kann dann von den nachfolgenden Aktivitätsinstanzen zugegriffen werden.

5.4.1.1. Ziel der Materialisierung

Um die oben erwähnten Ziele (i) – (iii) umzusetzen, wird jede in der Choreographie modellierte Invoke-Receive-Interaktion in p_μ wie folgt materialisiert. In dem Teilnehmer-Scope der in p_μ den sendenden Teilnehmer $\mathfrak{p}_{snd}$ repräsentiert, wird das Teilnehmerduplikat snd_{cp} des Invokes *snd* durch ein Assign syn_{snd} ersetzt:

1. $\textsc{Parent}(syn_{snd}) = \textsc{Parent}(snd_{cp})$ mit $snd_{cp} = \mathsf{actCopy}(\mathfrak{p}_{snd}, \bot, snd)$
   (Duplikat von *snd*)
2. $\mathsf{assign}_{src}(syn_{snd}) = v_{sndCp}$ mit $v_{sndCp} = \mathsf{varCopy}(\mathfrak{p}_{snd}, \bot, \mathsf{inputVar}(snd))$
   (Duplikat der Eingabevariable von *snd*)
3. $\mathsf{assign}_{trg}(syn_{snd}) = v_{rcvCp}$ mit $v_{rcvCp} = \mathsf{varCopy}(\mathfrak{p}_{rcv}, \bot, \mathsf{outputVar}(rcv))$
   (Duplikat der Ausgabevariable von *rcv*)
4. $\textsc{Incoming}_L(syn_{snd}) = \textsc{Incoming}_L(snd_{cp})$
5. $\textsc{Outgoing}_L(syn_{snd}) = \textsc{Outgoing}_L(snd_{cp}) \cup \{l_{syn}\}$
   mit $l_{syn} = (syn_{snd}, syn_{rcv}, \mathsf{receiver}(syn_{snd}) = \mathfrak{p}_{rcv})$
6. $\mathsf{joinCond}(syn_{snd}) = \mathsf{joinCond}(snd_{cp})$

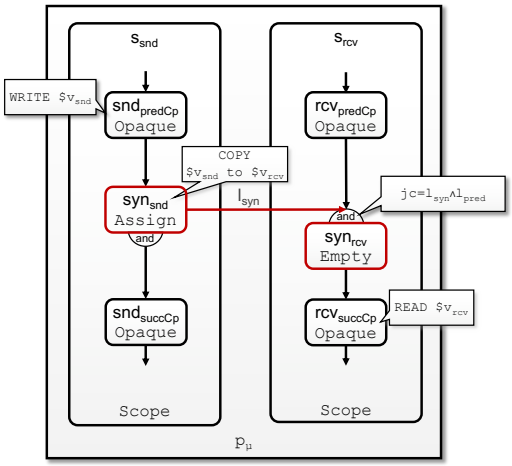

Abbildung 5.5.: Materialisierte Invoke- und Receive-Interaktion

Im Teilnehmer-Scope, der den empfangenden Teilnehmer $\mathfrak{p}_{rcv}$ repräsentiert, wird analog dazu das Duplikat von *rcv* durch eine neue Empty-Aktivität syn_{rcv} ersetzt, die die folgenden Eigenschaften besitzt:

1. $\textsc{Parent}(syn_{rcv}) = \textsc{Parent}(rcv_{cp})$ mit $rcv_{cp} = \mathsf{actCopy}(\mathfrak{p}_{rcv}, \bot, rcv)$ (Duplikat von *rcv*)
2. $\textsc{Incoming}_L(syn_{rcv}) = \textsc{Incoming}_L(rcv_{cp}) \cup \{l_{syn}\}$
3. $\textsc{Outgoing}_L(syn_{rcv}) = \textsc{Outgoing}_L(rcv_{cp})$
4. $\mathsf{joinCond}(syn_{rcv}) = \mathsf{joinCond}(rcv_{cp}) \wedge 1_{syn}$

Anschaulich ist das Ergebnis der Materialisierung der Interaktion in Abbildung 5.5 dargestellt. Die Assign-Aktivität syn_{snd} dient zum einen dazu, den Nachrichtenfluss über den Datenfluss in p_μ zu emulieren und zum anderen verknüpft sie die Kontrollflussgraphen der Teilnehmer-Scopes des sendenden und des empfangenden Teilnehmers. Um den Nachrichtenfluss über den Datenfluss zu emulieren, kopiert syn_{snd} den Inhalt des Teilnehmerduplikats der Eingabevariable des Invokes direkt in das Teilnehmerduplikat der Aus-

gabevariable des Receives (beide Variablen haben den gleichen Datentyp – siehe Abschnitt 3.1.2). Damit können die Nachfolgeaktivitäten von syn_{rcv} weiterhin auf die Daten zugreifen, die ursprünglich in der Nachricht übertragen wurden, ohne angepasst werden zu müssen. So kann die Aktivität rcv_{succCp} in Abbildung 5.5 zum Beispiel weiterhin auf das Teilnehmerduplikat der Variable v_{rcv} zugreifen.

Die Kontrollflussgraphen der Teilnehmer-Scopes werden über eine neue Kontrollflusskante l_{syn} verknüpft, womit sichergestellt wird, dass die Aktivität syn_{rcv} und ihre Nachfolgeaktivitäten erst ausgeführt werden, nachdem der Nachrichtenfluss von syn_{snd} emuliert wurde. Mittels der Transitionsbedingung $\mathsf{receiver}(syn_{snd}) = \mathfrak{p}_{rcv}$ wird gewährleistet, dass die Kante nur aktiviert wird, wenn zur Laufzeit die Kommunikation zwischen $\mathfrak{p}_{snd}$ und $\mathfrak{p}_{rcv}$ emuliert wird. Diese Prüfung ist für die Materialisierung von komplexeren Interaktionen notwendig, bei denen der Empfänger an den ein Invoke eine Nachricht sendet, zur Laufzeit dynamisch bestimmt werden kann. Dies ist zum Beispiel in der Choreographie in Abbildung 5.2 der Fall, wo das Invoke *Ticket buchen* nur eine Nachricht an die Airline sendet, die das günstigste Preisangebot gemacht hat. Bei der Materialisierung von diesen Interaktionen würde das das Invoke emulierende Assign syn_{snd} mit Empty-Aktivitäten syn_{rcv} in unterschiedlichen Teilnehmer-Scopes über jeweils eine Kante l_{syn} verbunden werden. Zur Laufzeit darf nur eine dieser Kanten aktiviert werden und zwar die, die mit dem Teilnehmer-Scope verbunden ist, mit dem die Kommunikation emuliert werden soll.

In der Choreographie müssen sowohl die Vorgängeraktivität von *rcv* als auch *snd* erfolgreich ausgeführt werden, bevor die Ausführung von *rcv* beendet werden kann (siehe Abschnitt 4.2.14). Daher die wird Eintrittsbedingung von syn_{rcv} so angepasst, dass die eingehenden Kontrollflusskanten aktiviert sein müssen, um syn_{rcv} auszuführen (parallele Vereinigung).

Alternativ könnte der Nachrichtenfluss auch ohne die Aktivität syn_{rcv} emuliert werden, indem die Aktivität syn_{snd} direkt mit dem Duplikat der Nachfolgeaktivität von *rcv* über die Kontrollflusskante l_{syn} verbunden wird. Hat

rcv allerdings mehrere Vorgänger bzw. Nachfolger, müssten alle Paare der Duplikate der Vorgänger- und Nachfolgeaktivitäten von *rcv* direkt über Kontrollflusskanten miteinander verbunden werden. Zusätzlich müsste für jede Nachfolgeaktivität jeweils eine Kante l_{syn} erstellt werden, die diese mit syn_{snd} verbindet. Daher wird hier, um die Anzahl der neu erstellten Kanten in p_μ und damit dessen Kontrollflusskomplexität [CMNR06] zu reduzieren, der Ansatz mit der Nachrichtenflussemulation über syn_{rcv} verfolgt.

Da die neu erstellte Assign- und Empty-Aktivität den Kontrollfluss zwischen den vormals interagierenden Kontrollflussgraphen synchronisieren, werden sie zusammen mit der Throw-Aktivität (siehe Invoke- und Pick-Interaktionen in Abschnitt 5.5) im Kontext der Materialisierung auch als *Synchronisationsaktivitäten* $SYN = A_{\texttt{assign}} \cup A_{\texttt{empty}} \cup A_{\texttt{throw}}$ bezeichnet.

5.4.1.2. Materialisierungsalgorithmus

Formal sind die Schritte zur Materialisierung von Invoke- und Receive-Interaktionen in der Prozedur MaterializeInvokeReceive (Algorithmus 5.4) beschrieben. Diese wird von der Prozedur Materialization in Algorithmus 5.3 für jedes Teilnehmerpaar aufgerufen, bei dem ein Invoke *snd* mit einem Receive-Aktivität *rcv* über eine Nachrichtenkante verbunden ist.

Im ersten Schritt überprüft der Algorithmus in Zeile 5 , ob das Teilnehmerduplikat snd_{cp} des Invokes bereits vorher durch eine Synchronisationsaktivität ersetzt wurde. Dazu wird Abbildung com2Syn : $A_{com} \cup CO_{rcv} \rightarrow SYN \cup \{\bot\}$ verwendet, die einer Kommunikationsaktivität a_{com} bzw. einem Nachrichtenereignis co_{rcv} die Synchronisationsaktivität zuweist, durch die sie in p_μ ersetzt wurde bzw. $\bot$ falls noch keine Ersetzung stattfand. Ein Invoke oder eine Kommunikationsaktivität im generellen kann vorher schon durch eine Synchronisationsaktivität ersetzt worden sein, falls die Invoke-Receive-Interaktion Teil einer komplexeren Interaktion ist, wie zum Beispiel einer „One-to-many Send“ Interaktion (siehe Abschnitt 7.2.2). Bei diesen Interaktionen, in der ein Teilnehmer Nachrichten an eine Menge von Teilnehmern sendet, würde

Algorithmus 5.4 Materialisierung von Invoke- und Receive-Interaktionen

```
 1: procedure MaterializeInvokeReceive(𝔭_snd, snd, 𝔭_rcv, rcv)
 2:     snd_cp ← actCopy(𝔭_snd, ⊥, snd)
 3:     rcv_cp ← actCopy(𝔭_rcv, ⊥, rcv)
 4:     syn_snd ← ⊥
 5:     if com2Syn(snd_cp) = ⊥ then
 6:         syn_snd ← CreateActivity(assign)
 7:         assign_src(syn_snd) ← inputVar(snd_cp)
 8:         assign_trg(syn_snd) ← outputVar(rcv_cp)
 9:         ReplaceActivity(snd_cp, syn_snd)
10:         com2Syn(snd_cp) ← syn_snd
11:     else
12:         syn_snd ← com2Syn(snd_cp)
13:     end if
14:     syn_rcv ← ⊥
15:     if com2Syn(rcv_cp) = ⊥ then
16:         syn_rcv ← CreateActivity(empty)
17:         ReplaceActivity(rcv_cp, syn_rcv)
18:         com2Syn(rcv_cp) ← syn_rcv
19:         l_syn ← CreateControlLink(syn_snd, syn_rcv, receiver(syn_snd) = 𝔭_rcv)
20:         joinCond(syn_rcv) ← joinCond(syn_rcv) ∧ 1_syn
21:     else
22:         syn_rcv ← com2Syn(a_rcv)
23:         l_syn ← CreateControlLink(syn_snd, syn_rcv, receiver(syn_snd) = 𝔭_rcv)
24:         joinCond(syn_rcv) ← joinCond(syn_rcv) ∧ 1_syn
25:     end if
26: end procedure
```

die Materialisierung zum Beispiel immer für die gleiche Aktivität snd_{cp} aber für die verschiedenen Teilnehmerduplikate des Receives der empfangenden Teilnehmer von Algorithmus 5.3 aufgerufen werden.

Wurde snd_{cp} noch nicht materialisiert, gilt $\mathsf{com2Syn}(snd_{cp}) = \bot$ und es wird in Zeile 6 die Assign-Aktivität syn_{snd} erstellt, die den Nachrichtenfluss emuliert. Das Ersetzen von snd_{cp} durch syn_{snd} im Kontrollfluss wird durch die Funktion ReplaceActivity umgesetzt, die in Anhang A.2 beschrieben ist. Die Aktivität snd_{cp} wird dann in Zeile 10 als materialisiert gekennzeichnet.

Mehrere eingehenden Kontrollflusskanten $l_{syn_1}, \ldots, l_{syn_n}$ werden erzeugt, falls die Receive-Aktivität Teil einer komplexen multilateralen Interaktion ist, wie zum Beispiel eine „One-from-many Receive“ Interaktion (siehe Abschnitt 7.2.3), in der eine Nachrichtenkante mehrere Sender mit einem Empfänger verbindet. Da auch, wie oben erwähnt, mehrere Teilnehmer über dasselbe Invoke mit einem Receive interagieren können, wird für das Teilnehmerduplikat rcv_{cp} des Receives in Zeile 15 ebenfalls überprüft, ob es zuvor schon materialisiert wurde. Ist dies nicht der Fall, wird eine neue Empty-Aktivität syn_{rcv} erstellt, die rcv_{cp} im Kontrollfluss des Teilnehmer-Scopes ersetzt. In Zeile 19 bzw. 23 wird die Kontrollflusskante l_{syn} zwischen den Synchronisationsaktivitäten mittels der Funktion CreateControlLink (siehe Anhang A.1) erstellt. Da zur Laufzeit eine Instanz von *rcv* nur eine Nachricht empfangen kann, muss die Eintrittsbedingung von syn_{rcv} so spezifiziert sein, dass neben der Eintrittbedingung von *rcv* gilt, dass eine der eingehenden Kontrollflusskanten $l_{syn_1}, \ldots, l_{syn_n}$ aktiviert ist.

5.4.1.3. Verhaltensäquivalenz

In diesem Abschnitt wird untersucht, inwieweit die Duplikate der Geschäftsaktivitäten in p_μ nach der Materialisierung verhaltensäquivalent zu ihren Originalen in $\mathfrak{p}_{snd}$ und $\mathfrak{p}_{rcv}$ sind. Dazu werden die Zustandsbeziehungen zwischen allen Paaren von originalen Geschäftsaktivitäten ermittelt, die direkt von der Materialisierung betroffen sind. Diese Zustandsbeziehungen werden dann mit denen verglichen, die nach der Materialisierung in p_μ zwischen den Duplikaten der originalen Geschäftsaktivitäten gelten.

Der Materialisierungsalgorithmus ersetzt die Aktivitäten *snd* und *rcv*. Folglich sind die Geschäftsaktivitäten von der Materialisierung betroffen, deren Zustände direkt von *snd* oder *rcv* beeinflusst werden. Dies sind alle, mit denen sich *snd* bzw. *rcv* in derselben Elternaktivität befinden[1]:

[1] Die Instanzen von *snd* bzw. *rcv* beeinflussen auch den Zustand der Instanzen ihrer jeweiligen Elternaktivität. Allerdings handelt es sich in diesem Kontext bei ihr nicht um eine Geschäftsaktivität.

- $A_{snd} := \{a \in A_{\text{opaque}} \mid a \in \textsc{Children}(\textsc{Parent}(snd))\}$
- $A_{rcv} := \{a \in A_{\text{opaque}} \mid a \in \textsc{Children}(\textsc{Parent}(rcv))\}$

Zur Vereinfachung der Verifikation werden hier nur Sende- oder Empfangsaktivitäten betrachtet, die genau eine parallele und eine Vorgänger- bzw. Nachfolgeaktivität besitzen. Andernfalls müssten bei der Diskussion des Verhaltens Fallunterscheidungen bezüglich der Anzahl der Vorgänger bzw. Nachfolger sowie deren Beziehungen untereinander gemacht werden (Parallelität, Exklusivität usw.). Dies ist keine Einschränkung der Allgemeinheit, da es sich bei den Aktivitäten um Geschäftsaktivitäten handelt, die wiederum eine Menge von parallelen Aktivitäten sowie Vorgänger- bzw. Nachfolgeaktivitäten abstrahieren können.

Für den paarweisen Vergleich können in A_{snd} die folgenden Aktivitäten unterschieden werden:

- snd_{pred} repräsentiert die direkte Vorgängeraktivität von snd:
  $snd_{pred} \in A_{snd} \cap \textsc{Predecessors}(snd) \wedge |\textsc{Successors}(snd_{pred})| = 1$
- snd_{succ} repräsentiert die direkte Nachfolgeaktivität von snd:
  $snd_{succ} \in A_{snd} \cap \textsc{Successors}(snd) \wedge |\textsc{Predecessors}(snd_{succ})| = 1$
- snd_{par} repräsentiert die zu snd parallele Aktivität:
  $snd_{par} \in A_{snd} \setminus (\textsc{Successors}_{all}(snd) \cup \textsc{Predecessors}_{all}(snd))$

Die Aktivitäten werden hier auch als *Umgebung* von snd bezeichnet. Sie sind in Abbildung 5.6 zusammen mit der Umgebung von rcv anschaulich dargestellt.

Formal ist die Umgebung von rcv, d.h. die Aktivitäten in A_{rcv}, wie folgt definiert:

- rcv_{pred} repräsentiert die direkte Vorgängeraktivität von rcv:
  $rcv_{pred} \in A_{rcv} \cap \textsc{Predecessors}(rcv) \wedge |\textsc{Successors}(rcv_{pred})| = 1$
- rcv_{succ} repräsentiert die direkte Nachfolgeaktivität von rcv:

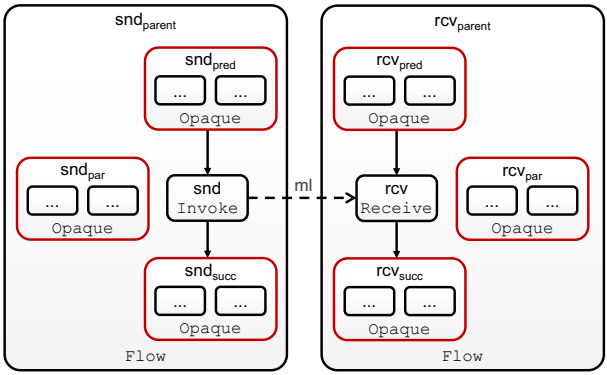

Abbildung 5.6.: Von der Materialisierung betroffene Aktivitäten in der Umgebung des Invokes *snd* und des Receives *rcv*

$$rcv_{succ} \in A_{rcv} \cap \textsc{Successors}(rcv) \wedge |\textsc{Predecessors}(rcv_{succ})| = 1$$

- rcv_{par} repräsentiert die parallele Aktivität von rcv:
  $rcv_{par} \in A_{rcv} \setminus (\textsc{Successors}(rcv) \cup \textsc{Predecessors}(rcv))$

Die Duplikate der Geschäftsaktivitäten in p_μ werden hier ebenfalls durch das Subskript Cp gekennzeichnet. Die Mengen A_{sndCp} oder A_{rcvCp} repräsentieren also die Duplikate von A_{snd} und A_{rcv}:

- $A_{sndCp} := \{a \in A_{\text{opaque}} \mid \exists a_{snd} \in A_{snd} : \text{actCopy}(\mathfrak{p}_{snd}, \bot, a_{snd}) = a\}$
- $A_{rcvCp} := \{a \in A_{\text{opaque}} \mid \exists a_{rcv} \in A_{rcv} : \text{actCopy}(\mathfrak{p}_{rcv}, \bot, a_{rcv}) = a\}$

Das Ergebnis der Untersuchung ist in Abbildung 5.7 zusammengefasst. Es gelten nach der Materialisierung zwischen allen Duplikaten der Geschäftsaktivitäten in p_μ dieselben Zustandsbeziehungen, wie zwischen den originalen Aktivitäten in der Invoke-Receive-Interaktion. Eine Ausnahme bilden die Duplikate snd_{succCp} und rcv_{succCp}, deren Zustandsbeziehungen nicht denen von snd_{succ} und rcv_{succ} entsprechen. Detailliert werden die Zustandsbeziehungen zwischen den Geschäftsaktivitäten im Verlauf dieses Abschnitts diskutiert.

		A_{snd}			A_{rcv}		
		snd_{par}	snd_{pred}	snd_{succ}	rcv_{par}	rcv_{pred}	rcv_{succ}
A_{snd}	snd_{par}		✓(1)	✓(1)	✓(6)	✓(3)	✓(5)
	snd_{pred}	✓(1)		✓(2)	✓(6)	✓(3)	✗(4)
	snd_{succ}	✓(1)	✓(2)		✓(6)	✓(3)	✓ (7)
A_{rcv}	rcv_{par}	✓(6)	✓(6)	✓(6)		✓(1)	✓(1)
	rcv_{pred}	✓(3)	✓(3)	✓(3)	✓(1)		✗(2)
	rcv_{succ}	✓(5)	✗(4)	✓(7)	✓(1)	✗(2)	

Abbildung 5.7.: Verhaltensäquivalenz zwischen den Geschäftsaktivitäten in p_μ nach der Materialisierung von Invoke-Receive-Interaktionen

(1) Beziehungen von parallelen Aktivitäten snd_{par} bzw. rcv_{par}

Hier werden die Beziehungen der Geschäftsaktivitäten snd_{par} zu den Geschäftsaktivitäten snd_{pred} sowie snd_{succ} in der Choreographie und nach der Materialisierung in p_μ untersucht. Analog dazu werden die Beziehungen diskutiert, die die Aktivitäten in rcv_{par} zu den Aktivitäten rcv_{pred} und rcv_{succ} haben.

Da sich snd_{pred} bzw. snd_{succ} in der gleichen Elternaktivität wie snd_{par} befinden und zwischen ihnen und der Aktivität snd_{par} keine Synchronisationsabhängigkeiten bestehen, gilt zwischen snd_{par} und snd_{pred} bzw. snd_{succ} das Profil für die Kindaktivitäten von strukturierten Aktivitäten aus Abbildung 4.10. Dieses Profil gilt aus den gleichen Gründen zwischen der Aktivität rcv_{par} und rcv_{pred} bzw. rcv_{succ}.

Bei der Generierung der Teilnehmer-Scopes von $\mathfrak{p}_{snd}$ und $\mathfrak{p}_{rcv}$ wurde sichergestellt, dass Duplikate der Elternaktivitäten von snd_{par} bzw. rcv_{par} sowie die Duplikate von snd_{par} bzw. rcv_{par} im jeweiligen Teilnehmer-Scope erstellt wurden. Dabei wurden ebenfalls die originalen Kontrollflussbeziehungen zwischen den Duplikaten in den Teilnehmer-Scopes abgebildet. Diese Kontrollflussbeziehungen werden durch den Materialisierungsalgorithmus nicht

verändert, da zwischen ihnen und auch den Vorgänger- und Nachfolgeaktivitäten von *snd* bzw. *rcv* keine neuen Kontrollflusskonstrukte erzeugt werden. Folglich gelten nach der Materialisierung in p_μ zwischen den Duplikaten snd_{parCp} und snd_{predCp} bzw. snd_{succCp} dieselben Zustandsbeziehungen wie zwischen den Originalen in der Choreographie. Die Beziehungen zwischen rcv_{parCp} und rcv_{predCp} bzw. rcv_{succCp} werden daher ebenfalls beibehalten.

(2) Beziehungen zwischen snd_{pred} und snd_{succ} bzw. rcv_{pred} und rcv_{succ}

Die Aktivitäten snd_{pred} und snd_{succ} befinden sich in derselben strukturierten Aktivität und werden über die Aktivität *snd* sequentiell ausgeführt[1]. Daher können die zwischen snd_{pred} und snd_{succ} geltenden Zustandsbeziehungen transitiv über die Zustände zwischen snd_{pred} (Sequenz) und *snd* sowie *snd* und snd_{succ} (Sequenz) abgeleitet werden. Die Instanz von snd_{succ} kann den Zustand *executing* und dessen Nachfolgezustände nur erreichen, nachdem erst die Instanz von snd_{pred} und danach die Instanz von *snd* den Zustand *completed* erreicht haben (Axiom *Sq1*). Wird hingegen snd_{pred} in den Zustand *dead* gesetzt, erreicht danach erst die Instanz *snd* und dann snd_{succ} den Zustand *dead* (Axiom *Sq2*). Die Terminierung der Instanz von snd_{pred} führt ebenfalls zur Terminierung von *snd* und snd_{succ}, wobei keine Terminierungsreihenfolge festgelegt ist (Axiom $St_{CH2CH}3$). Löst eine Instanz von snd_{pred} einen Fehler aus, werden *snd* und snd_{succ} danach terminiert (Axiom $St_{CH2CH}2$).

Die Aktivitäten rcv_{pred} und rcv_{succ} befinden sich ebenfalls in derselben strukturierten Aktivität und werden über die Aktivität *rcv* sequentiell ausgeführt. Folglich gelten zwischen rcv_{pred} und rcv_{succ} die Zustandsbeziehungen, die auch zwischen snd_{pred} und snd_{succ} gelten. Das resultierende Zustandsprofil zwischen snd_{pred} und snd_{succ} bzw. rcv_{pred} und rcv_{succ} ist Abbildung 5.8 dargestellt.

Der Materialisierungsalgorithmus ersetzt im Teilnehmer-Scope von $\mathfrak{p}_{snd}$ die

[1] Der Fall, dass sich die Vorgänger- oder Nachfolgeaktivität von *snd* oder *rcv* in anderen Elternaktivitäten als *snd* bzw. *rcv* befinden, wird hier nicht betrachtet.

snd_{pred} / rcv_{pred}		snd_{succ} / rcv_{succ}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	→	→	⊕	⊕	⊕	⊕
	ABORTED	←	⊕	‖	⊕	⊕	⊕	⊕
	EXECUTING	←	⊕	→	→	→	→	→
	TERMINATED	←	⊕	‖	⊕	⊕	⊕	⊕
	FAULTED	←	⊕	→	⊕	⊕	⊕	⊕
	COMPLETED	←	⊕	→	→	→	→	→

Abbildung 5.8.: Zustandsprofil zwischen snd_{pred} und snd_{succ} bzw. rcv_{pred} und rcv_{succ}

Aktivität *snd* durch die Aktivität syn_{snd}, die alle eingehenden und ausgehenden Kontrollflusskanten von *snd* zugewiesen bekommt. Folglich sind die Teilnehmerduplikate snd_{predCp} und snd_{succCp} weiterhin sequentiell über die Aktivität syn_{snd} miteinander verbunden. Daher gelten zwischen ihnen die gleichen Zustandsbeziehungen wie zwischen snd_{pred} und snd_{succ}[1]. Die parallele Verzweigung an syn_{snd} mit der ausgehenden Kontrollflusskante l_{syn} hat keinen Einfluss auf die Zustandsbeziehungen, da die Kante keine neuen Kontrollflussabhängigkeiten zwischen snd_{predCp} und snd_{succCp} impliziert.

Die Aktivität *rcv* wurde durch syn_{rcv} ersetzt, die ebenfalls alle eingehenden und ausgehenden Kontrollflusskanten von *rcv* zugewiesen bekommt. Die Aktivität syn_{rcv} fungiert als parallele Vereinigungsaktivität für die Kontrollflusskante l_{syn} und die ausgehende Kante von rcv_{predCp}. Durch die parallele Vereinigung gilt zwischen dem Zustand *executing* von rcv_{predCp} und dem Zustand *dead* von rcv_{succCp} die Beziehung → anstatt der Beziehung ⊗, die zwischen den originalen Aktivitäten rcv_{pred} und rcv_{succ} gilt (im Profil in Abbildung 5.9 rot markiert).

In der Choreographie hat im regulären Kontrollfluss aufgrund der sequentiellen Ausführung jede Instanz von rcv_{succ} den Zustand *executing* oder einen Feh-

[1] Da die Zustandsbeziehungen erhalten bleiben, ist das Profil zwischen snd_{predCp} und snd_{succCp} hier nicht dargestellt.

		rcv_{succCp}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
rcv_{predCp}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	→	→	⊕	⊕	⊕	⊕
	ABORTED	←	⊕	‖	⊕	⊕	⊕	⊕
	EXECUTING	←	→	→	→	→	→	→
	TERMINATED	←	⊕	‖	⊕	⊕	⊕	⊕
	FAULTED	←	⊕	→	⊕	⊕	⊕	⊕
	COMPLETED	←	⊕	→	→	→	→	→

Abbildung 5.9.: Zustandsprofil zwischen rcv_{predCp} und rcv_{succCp}

lerzustand erreichen müssen, nachdem eine Instanz von rcv_{pred} den Zustand *executing* erreicht hat. Das heißt, dass eine Instanz von rcv_{succ} im regulären Kontrollfluss den Zustand *dead* nicht erreichen konnte, wenn vorher die Instanz von rcv_{pred} den Zustand *executing* erreicht hat. Durch die parallele Vereinigung an Aktivität syn_{rcv} kann, falls die Kontrollflusskante l_{syn} deaktiviert wurde, eine Instanz von rcv_{succCp} auch den Zustand *dead* erreichen, wenn die Instanz von rcv_{predCp} den Zustand *dead* erreicht hat. Die Kante l_{syn} wird allerdings nur deaktiviert, wenn syn_{snd} nicht erfolgreich ausgeführt wurde. Übertragen auf die originale Invoke-Receive-Interaktion bedeutet die nicht erfolgreiche Ausführung von syn_{snd}, dass keine Nachricht gesendet wurde und dass die Instanz von *rcv*, wie in Abschnitt 4.2.14 beschrieben, hätte terminiert werden müssen, um nicht im Zustand *executing* zu „hängen“. Das heißt auch, dass in p_μ die Instanz syn_{rcv} und somit auch die Instanz von rcv_{succCp} terminiert[1] werden würde, also nicht den Zustand *dead* erreichen kann. Daher hat es für die Untersuchung der Aktivitätszustandsbeziehungen keine Relevanz, dass in p_μ die Instanzen von rcv_{succCp} potentiell den Zustand *dead* erreichen können, nachdem eine Instanz von rcv_{predCp} den Zustand *executing* erreicht hat.

[1] Wie die Terminierung in der Choreographie bzw. p_μ modelliert ist, wird hier nicht berücksichtigt.

		rcv_{succ}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
snd_{pred}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	‖	‖	⊕	⊕	⊕	⊕
	ABORTED	←	‖	‖	⊕	⊕	⊕	⊕
	EXECUTING	←	‖	‖	→	→	→	→
	TERMINATED	←	‖	‖	⊕	⊕	⊕	⊕
	FAULTED	←	‖	‖	⊕	⊕	⊕	⊕
	COMPLETED	←	‖	‖	→	→	→	→

Abbildung 5.10.: Zustandsprofil zwischen snd_{pred} und rcv_{succ}

(3) Beziehungen zwischen den Aktivitäten in A_{snd} und rcv_{pred}

Die Aktivitäten A_{snd} und rcv_{pred} befinden sich in unterschiedlichen Teilnehmern. Da rcv_{pred} eine Vorgängeraktivität von rcv ist, impliziert die Nachrichtenkante keine Zustandsabhängigkeit zwischen ihnen. Die Aktivitäten aus A_{snd} sind daher vollständig isoliert von der Aktivität rcv_{pred} und es gilt das Profil aus Abbildung 4.16.

Die Duplikate A_{sndCp} und rcv_{predCp} befinden sich auch in p_{μ} in unterschiedlichen Teilnehmer-Scopes. Der Materialisierungsalgorithmus erstellt keine Kontrollflusskonstrukte, mit denen eine Zustandsabhängigkeit zwischen den Aktivitäten aus A_{sndCp} und rcv_{predCp} entsteht, da die generierte Kontrollflusskante l_{syn} erst die Nachfolgeaktivität von rcv_{predCp} mit den Aktivitäten aus A_{sndCp} verbindet. Es gilt daher zwischen den Aktivitäten aus A_{sndCp} und rcv_{predCp} ebenfalls das Profil für vollständige isolierte Aktivitäten.

(4) Beziehungen zwischen den Aktivitäten snd_{pred} und rcv_{succ}

In Abbildung 5.10 sind die transitiven Beziehungen zwischen snd_{pred} und rcv_{succ} dargestellt. Die Aktivitäten snd_{pred} und rcv_{succ} sind über zwei Kontrollfluss- und eine Nachrichtenkante miteinander verbunden. Daher können ihre Zustandsbeziehungen transitiv über die Beziehungen abgeleitet werden, die zwischen snd_{pred} und snd (Sequenz), snd und rcv (Nachrichtenkanten –

siehe Abschnitt 4.2.14) sowie *rcv* und rcv_{succ} (Sequenz) gelten.

Eine Instanz von *snd* kann nur den Zustand *executing* erreichen und eine Nachricht senden, wenn snd_{pred} den Zustand *completed* erreicht hat (Axiom *Sq1*). Daraus folgt, dass die empfangende Instanz von *rcv* nur in den Zustand *faulted* oder *completed* gehen kann, wenn die Instanz von *snd* (Axiom $Ml_{InvRcv}1$) und somit auch snd_{pred} den Zustand *completed* erreicht hat. Aufgrund der sequentiellen Ausführung von *rcv* und rcv_{succ} impliziert dies wiederum, dass snd_{pred} den Zustand *completed* erreichen muss, bevor rcv_{succ} den Zustand *executing* und somit *completed*, *faulted* oder *terminated* erreichen kann. Erreicht die Instanz von snd_{pred} den Zustand *dead* oder einen Fehlerzustand, kann die Instanz von *snd* keine Nachricht senden (Axiom *Sq3*) und die Instanz von *rcv* muss ebenfalls in den Zustand *dead* oder einen Fehlerzustand gesetzt werden (Axiom $Ml_{InvRcv}2$). Dies führt wiederum dazu, dass die Instanz von rcv_{succ} den Zustand *dead* oder einen Fehlerzustand erreicht (Axiom *Sq2* bzw. *Sq3*). Durch die Isolation von snd_{pred} und rcv_{succ} ist keine Reihenfolge vorgegeben, wann rcv_{succ} in einen dieser Zustände wechselt (Axiom *Iso*).

Die Aktivitäten snd_{predCp} und rcv_{succCp} sind im Kontrollfluss transitiv über die durch die Materialisierung erstellten Aktivitäten syn_{snd} und syn_{rcv} miteinander verbunden, wobei die Aktivitäten syn_{snd} und syn_{rcv} über die neu erstellte Kontrollflusskante l_{syn} sequentiell ausgeführt werden. Das Zustandstransitionsprofil zwischen snd_{pred} und rcv_{succ}, das aus den transitiven Kontrollflussabhängigkeiten resultiert, ist in Abbildung 5.11 dargestellt.

Die Instanz von rcv_{succCp} erreicht, wie bei der Invoke-Receive-Interaktion, nur den Zustand *executing* und dessen Nachfolgezustände, nachdem die Instanz von snd_{predCp} den Zustand *completed* erreicht hat. Durch die sequentielle Ausführung von snd_{predCp}, syn_{snd}, syn_{rcv} und rcv_{succCp} sowie der parallelen Vereinigung an Aktivität syn_{rcv} (Abschnitt 4.2.12) muss jede Instanz dieser Aktivitäten den Zustand *completed* erreichen, bevor die Instanz von rcv_{succCp} in *executing* gehen kann.

Erreicht die Instanz von snd_{predCp} einen Fehlerzustand, wird die Instanz von

		rcv_{succCp}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
snd_{predCp}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	→	‖	⊕	⊕	⊕	⊕
	ABORTED	←	→	‖	⊕	⊕	⊕	⊕
	EXECUTING	←	→	‖	→	→	→	→
	TERMINATED	←	→	‖	⊕	⊕	⊕	⊕
	FAULTED	←	→	‖	⊕	⊕	⊕	⊕
	COMPLETED	←	→	‖	→	→	→	→

Abbildung 5.11.: Zustandsprofil zwischen snd_{predCp} und rcv_{succCp}

syn_{snd} terminiert und die Instanz von syn_{rcv} geht in den Zustand *dead*, da sich die Aktivitäten in unterschiedlichen Scopes befinden (Axiom *SqI3*) und es sich bei syn_{rcv} um eine parallele Vereinigungsaktivität handelt. Folglich kann, wie bei der Invoke-Receive-Interaktion, die Aktivität rcv_{succCp} die Zustände *executing*, *terminated* und *completed* nicht erreichen, wenn sich snd_{predCp} in einem Fehlerzustand befindet.

Da sich snd_{predCp} und rcv_{succCp} in verschiedenen Teilnehmer-Scopes befinden, können die Instanzen beider Aktivitäten weiterhin unabhängig voneinander in den Zustand *aborted* gehen.

Der Unterschied zwischen den Profilen in Abbildung 5.10 und Abbildung 5.11 (rot dargestellt) besteht darin, dass die Instanz von rcv_{succCp} den Zustand *dead* erst erreichen kann, nachdem snd_{predCp} einen Endzustand erreicht hat. Im Gegensatz zur Invoke-Receive-Interaktion, wo die Instanzen von snd_{pred} und rcv_{succ} den Zustand *dead* unabhängig voneinander erreichen können. Diese Zustandsabhängigkeit resultiert daher, dass durch die Kontrollflusskante l_{syn} die Aktivität snd_{predCp} eine indirekte Vorgängeraktivität von rcv_{succCp} geworden ist. Folglich muss der Endzustand der Instanzen von rcv_{predCp}, syn_{rcv}, syn_{snd} und somit auch von snd_{predCp} bekannt sein, bevor rcv_{succCp} in einen regulären Nachfolgezustand von *initial* gehen kann.

Dass rcv_{succCp} erst in *dead* gehen kann, wenn snd_{predCp} einen Endzustand

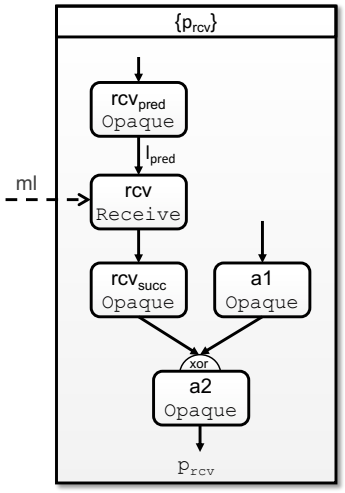

Abbildung 5.12.: Receive mit exklusiver Vereinigung nach rcv_{succ}

erreicht hat, hat zur Folge, dass in p_μ unter Umständen die Ausführung der Instanzen der Nachfolgeaktivitäten von rcv_{predCp} im Vergleich zur originalen Choreographie verzögert wird. In dem Choreographiefragment in Abbildung 5.12 wird zum Beispiel die Instanz von rcv_{succ} in den Zustand *dead* gesetzt, nachdem rcv_{pred} den Zustand *dead* erreicht hat. Dies geschieht unabhängig davon, ob eine Nachricht an *rcv* gesendet wurde.

Das Choreographiefragment nach der Materialisierung ist in Abbildung 5.13 dargestellt. Erreicht dort analog dazu die Instanz von rcv_{predCp} in p_μ den Zustand *dead*, muss die Instanz von rcv_{succCp} warten, bis die das Senden emulierende Kante l_{syn} evaluiert wurde. Erst dann kann auch die Instanz von rcv_{succCp} in *dead* gehen. Dies würde wiederum die Ausführung der Instanz von $a2_{cp}$ (Teilnehmerduplikat von Aktivität *a2*) verzögern, da die Instanzen aller Vorgängeraktivitäten von $a2_{cp}$ den Zustand *dead* oder *completed* erreicht haben müssen, bevor die Instanz von $a2_{cp}$ den Zustand *executing* erreichen kann.

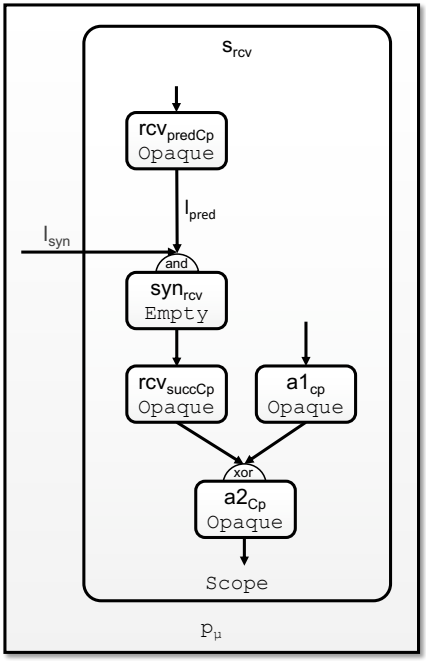

Abbildung 5.13.: p_μ mit exklusiver Vereinigung nach rcv_{succ}

(5) Beziehungen zwischen den Aktivitäten snd_{par} und rcv_{succ}

Die Aktivität snd_{par} ist im Kontrollfluss parallel zu *snd* modelliert. Sie kann daher entweder vor, nach oder während der Ausführung von *snd* einen regulären oder einen Fehlerzustand erreichen.

Da sich *snd* und snd_{par} in der gleichen Elternaktivität befinden, kann eine Instanz von *snd* nur erfolgreich ausgeführt werden, solange die Instanz von snd_{par} nicht in einen Fehlerzustand gesetzt wurde (Axiom $St_{CH2CH}4$). Geht die Instanz von snd_{par} vorher in einen Fehlerzustand, kann die Nachricht nicht gesendet werden und die Instanzen von *rcv* und rcv_{succ} können, wie oben beschrieben, weder den Zustand *completed* noch den Zustand *faulted* erreichen. Erreicht hingegen die Instanz von snd_{par} den Fehlerzustand erst, nachdem

		rcv_{succ}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
snd_{par}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	‖	‖	⊕	⊕	⊕	⊕
	ABORTED	←	‖	‖	‖	‖	‖	‖
	EXECUTING	←	‖	‖	‖	‖	‖	‖
	TERMINATED	←	‖	‖	‖	‖	‖	‖
	FAULTED	←	‖	‖	‖	‖	‖	‖
	COMPLETED	←	‖	‖	‖	‖	‖	‖

Abbildung 5.14.: Zustandsprofil zwischen snd_{par} und rcv_{succ}

die Instanz von *snd* erfolgreich in *completed* gegangen ist, kann die Instanz von rcv_{succ} in *completed* oder *faulted* wechseln. Folglich existiert zwischen den Fehlerzuständen von snd_{par} und rcv_{succ}, wie im Profil in Abbildung 5.14 dargestellt, die Beziehung ‖.

Per Definition wird eine Instanz von snd_{par} immer ausgeführt, wenn die Instanz ihrer Elternaktivität ausgeführt wird. Wird diese deaktiviert, wird die Instanz von snd_{par} und somit auch die Instanz von *snd* in den Zustand *dead* gesetzt (Axiom $St_{CH}2$). In diesem Fall kann die Instanz von *snd* keine Nachricht senden und *rcv* nicht *completed* erreichen. Daher kann die Instanz von rcv_{succ} nicht in *executing* gehen, wenn snd_{par} in *dead* gesetzt wurde.

Da zwischen den Teilnehmerduplikat von snd_{par} und der Aktivität syn_{snd} keine Synchronisationsabhängigkeit besteht, erzeugt die bei der Materialisierung erstellte Kontrollflusskante l_{syn} ebenfalls keine Synchronisationsabhängigkeiten zwischen snd_{parCp} und rcv_{succCp}. Wird die Instanz von snd_{parCp} in *dead* gesetzt, impliziert das wieder, dass ihre Elternaktivität und damit auch die Instanzen von syn_{snd} in *dead* gesetzt wurden. Daher ergibt sich dasselbe Szenario wie in der Choreographie, wenn die Instanzen von snd_{par} und syn_{snd} in *dead* gehen, auch die Instanz von rcv_{succCp} den Zustand *executing* nicht erreicht. Wird die Instanz von snd_{parCp} in *executing* oder *completed* gesetzt, hat dies, aufgrund der fehlenden Synchronisationsabhängigkeit zu rcv_{succCp}, keinen Einfluss auf deren Instanzen.

		rcv_{succCp}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
snd_{parCp}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	‖	‖	⊕	⊕	⊕	⊕
	ABORTED	←	‖	‖	‖	‖	‖	‖
	EXECUTING	←	‖	‖	‖	‖	‖	‖
	TERMINATED	←	‖	‖	‖	‖	‖	‖
	FAULTED	←	‖	‖	‖	‖	‖	‖
	COMPLETED	←	‖	‖	‖	‖	‖	‖

Abbildung 5.15.: Zustandsprofil zwischen snd_{parCp} und rcv_{succCp}

Verursacht die Instanz von snd_{parCp} einen Fehler oder wird sie terminiert, bevor die Instanz von syn_{snd} *completed* erreicht, führt das zu deren Terminierung. Wie für die Beziehungen zwischen snd_{pred} und rcv_{succ} erläutert, wird die Instanz von rcv_{succCp} aufgrund des Axioms *SqI3* in den Zustand *dead* gesetzt. Geht die Instanz von snd_{parCp} in einen Fehlerzustand, nachdem die Instanz von syn_{snd} *completed* erreicht hat, hat dies keine Auswirkung auf die Instanz von rcv_{succCp}. Folglich besteht weiterhin die Beziehung ‖ zwischen den Fehlerzuständen und den Zuständen *completed* und *faulted*. Das Profil ist in Abbildung 5.15 dargestellt.

(6) Beziehungen zwischen den Aktivitäten in rcv_{par} und A_{snd}

Die zu *rcv* parallele Aktivität rcv_{par} hat keine Synchronisationsabhängigkeit zu *rcv* und damit auch keine zu den Aktivitäten des Senders A_{snd}. Die Zustände der Instanzen von A_{snd} und rcv_{par} haben also im regulären Kontrollfluss keinen Einfluss aufeinander. Da sie sich in verschiedenen Teilnehmern befinden, haben die Zustände auch im Fehlerfall keinen Einfluss aufeinander. Es gilt daher zwischen ihnen das Profil für vollständig isolierte Aktivitäten.

Das Duplikat rcv_{parCp} besitzt nach der Materialisierung weiterhin keine Synchronisationsabhängigkeiten zu syn_{rcv} oder einer ihrer Nachfolgeaktivitäten. Daher impliziert auch die Kontrollflusskante l_{syn} zwischen den Duplikaten A_{sndCp} und rcv_{parCp} keine Zustandsabhängigkeiten. Es gilt also zwischen A_{sndCp}

		rcv_{succ}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
snd_{succ}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	‖	‖	⊕	⊕	⊕	⊕
	ABORTED	←	‖	‖	‖	‖	‖	‖
	EXECUTING	←	‖	‖	‖	‖	‖	‖
	TERMINATED	←	‖	‖	‖	‖	‖	‖
	FAULTED	←	‖	‖	‖	‖	‖	‖
	COMPLETED	←	‖	‖	‖	‖	‖	‖

Abbildung 5.16.: Zustandsprofil zwischen snd_{succ} und rcv_{succ}

und rcv_{parCp} ebenfalls das Profil für isolierte Aktivitäten.

(7) Beziehungen zwischen den Aktivitäten snd_{succ} und rcv_{succ}

Die Aktivitäten snd_{succ} und rcv_{succ} befinden sich in unterschiedlichen Teilnehmern und haben keine direkte oder indirekte Kontrollflussabhängigkeit. Aufgrund der Kontrollfluss- und der Nachrichtenkante besitzen aber beide eine direkte bzw. indirekte Zustandsabhängigkeit zu *snd*. Das führt dazu, dass, wenn eine Instanz von snd_{succ} den Zustand *dead* erreicht, die Instanz von rcv_{succ} den Zustand *executing* und seine Nachfolgezustände nicht erreichen kann. Dies liegt darin begründet, dass die Instanz von snd_{succ} nur in den Zustand *dead* gehen kann, wenn die Instanz von *snd* den Zustand *dead* erreicht hat (Axiom *Sq2*). Folglich kann, wie oben erläutert, die Instanz von *snd* keine Nachricht senden und die Instanz von *rcv* damit nicht den Zustand *completed* oder *faulted* erreichen (Axiom $Ml_{InvRcv}2$). Das hat zur Konsequenz, dass die Instanz von rcv_{succ} den Zustand *executing* nicht erreichen kann. Zwischen den anderen Zuständen besteht aufgrund der Isolation von snd_{succ} und rcv_{succ} und dadurch, dass sie keine Kontrollflussabhängigkeit zueinander haben, kein Zusammenhang. Das Profil ist in Abbildung 5.16 dargestellt.

Nach der Materialisierung sind die Duplikate snd_{succCp} und rcv_{succCp} zueinander parallele Aktivitäten, die sich in unterschiedlichen Scopes befinden,

wobei die Aktivität syn_{snd} als Verzweigungsaktivität fungiert. Zwischen den Aktivitäten gelten die in Abschnitt 4.2.10.2 beschriebenen Zustandsbeziehungen für fehlerisolierte parallele Aktivitäten allerdings nur eingeschränkt, da die parallele Vereinigungsaktivität syn_{rcv} ebenfalls ein Vorgänger von rcv_{succCp} ist. Im Gegensatz zu diesen Beziehungen, wo die Instanzen von parallelen Aktivitäten entweder *executing* oder *dead* erreichen müssen (Axiom And_{split}), kann eine Instanz von rcv_{succCp} auch dann in *dead* gehen, wenn die Instanz von snd_{succCp} den Zustand *executing* erreicht hat, und zwar dann, wenn die Instanz syn_{rcv} den Zustand *dead* erreicht hat. Daher entspricht das Profil zwischen snd_{succCp} und rcv_{succCp} dem, das zwischen den originalen Aktivitäten snd_{succ} und rcv_{succ} gilt.

5.5. Materialisierung von Interaktionen zwischen Invoke-Aktivitäten und Nachrichtenereignissen

In diesem Abschnitt wird die Materialisierung der Basisinteraktion zwischen Invoke-Aktivitäten und Nachrichtenereignissen diskutiert. Der Abschnitt ist analog zum vorherigen aufgebaut, d.h. es wird zuerst das Ziel der Materialisierung erläutert und dann der Algorithmus beschrieben, der die Materialisierung umsetzt. Inwieweit die materialisierte Interaktion in p_μ verhaltensäquivalent zu der in der Choreographie ist, wird am Ende untersucht.

In Abbildung 5.17 ist die Interaktion zwischen einem Invoke *snd* und einem Nachrichtenereignis e^{msg} über eine Nachrichtenkante *ml* dargestellt. Wird die Nachricht vom Teilnehmer $\mathfrak{p}_{snd}$ über das Invoke an Teilnehmer $\mathfrak{p}_{rcv}$ gesendet, wird die Instanz der dem jeweiligen Nachrichtenereignis e^{msg} zugeordneten Aktivität ch_{msg} ausgeführt (siehe Abschnitt 4.2.15.2). Nach der Auslösung des Nachrichtenereignis wird der Inhalt der Nachricht vom Pick in eine Variable kopiert.

Einer Pick-Aktivität können weitere Ereignisse E^{rcv} zugewiesen sein, wobei jedes Ereignis einer Aktivität ch_{event} zugeordnet ist, die bei dessen Eintritt

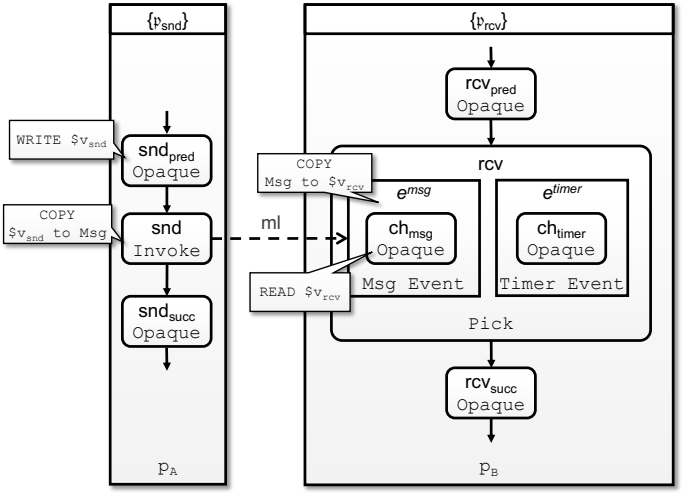

Abbildung 5.17.: Interaktionen zwischen einer Invoke-Aktivität und einem Nachrichtenereignis

ausgeführt wird. So kann ein Pick neben unterschiedlichen Timer- auch mehrere Nachrichtenereignisse enthalten, um zum Beispiel multilaterale Interaktionsmuster wie „Racing Incoming Messages“ (siehe Abschnitt 7.2.1) zu modellieren.

Formal wird eine Interaktion zwischen einer Invoke-Aktivität und einem Nachrichtenereignis durch ein Choreographiefragment $c^F = (\mathfrak{P}, \mathfrak{P}^{set}, P, ML)$ implementiert, das die folgenden Eigenschaften erfüllt:

1. $\mathfrak{P} := \{\mathfrak{p}_{snd}, \mathfrak{p}_{rcv}\}$
2. $\mathfrak{P}^{set} := \{\{\mathfrak{p}_{snd}\}, \{\mathfrak{p}_{rcv}\}\}$
3. $P := \{p_{snd}, p_{rcv}\}$
   a) $\text{type}_{\mathfrak{p}}(\mathfrak{p}_{snd}) = p_{snd}$
   b) $\text{type}_{\mathfrak{p}}(\mathfrak{p}_{rcv}) = p_{rcv}$
4. $ML := \{ml\}$ mit $ml = (\{\mathfrak{p}_{snd}\}, snd, \{\mathfrak{p}_{rcv}\}, e^{msg})$

5. $E := \{e^{msg}\} \cup E^{other}$ mit $E = \pi_8(p_{rcv})$
   a) $e^{msg} \in \{e \in E^{msg} \mid \exists hr \in HR : \pi_1(hr) = rcv \wedge \pi_2(hr) = \pi_4(ml) = e\}$
   b) $E^{other} \in \{e \in (E^{std} \setminus \{e^{msg}\}) \mid \exists hr \in HR : \pi_1(hr) = rcv \wedge \pi_2(hr) = e\}$
6. $HR_{pRcv} := \{hr_{msg}\} \cup HR_{event}$ mit $HR_{pRcv} = \pi_2(p_{rcv})$
   a) $hr_{msg} = (rcv, e^{msg}, ch_{msg})$
   b) $HR_{other} \subset \{hr \in HR_{pRcv} \mid \pi_1(hr) = rcv \wedge \pi_2(rcv) \in E^{other}\}$
7. $A := A_{pSnd} \cup A_{pRcv}$ mit:
   $\pi_1(p_{snd}) = A_{pSnd} = \{snd\} \wedge \pi_1(p_{rcv}) = A_{pRcv} = \{rcv, ch_{msg}, Ch_{event}\}$
   a) $snd \in A_{\texttt{invoke}} \wedge \mathsf{receiver}(snd) = (\mathfrak{p}_{rcv})$
   b) $rcv \in A_{\texttt{pick}}$
   c) $ch_{msg} = \pi_3(hr_{msg}) \wedge \mathsf{sender}(ch_{msg}) = (\mathfrak{p}_{snd})$
   d) $Ch_{event} = \bigcup_{hr \in HR_{other}} \pi_3(hr)$
8. $V := \{v_{snd}, v_{rcv}\}$ mit $v_{snd} \in \pi_4(p_{snd}) \wedge v_{rcv} \in \pi_4(p_{rcv})$
   a) $\mathsf{inputVar}(snd) = v_{snd}$
   b) $\mathsf{outputVar}(rcv) = v_{rcv}$

5.5.1. Ziel der Materialisierung

Bei der Materialisierung von Interaktion zwischen Invoke-Aktivitäten und Nachrichtenereignissen in p_μ müssen ebenfalls die in Abschnitt 5.4 formulierten Ziele (i) – (iii) realisiert werden. Im Teilnehmer-Scope, der in p_μ den sendenden Teilnehmer $\mathfrak{p}_{snd}$ repräsentiert, wird dazu das Teilnehmerduplikat snd_{cp} des Invokes snd durch ein Assign syn_{snd} ersetzt, das zusammen mit der ausgehenden Kontrollflusskante l_{syn} den Nachrichtenfluss emuliert. Die Aktivität syn_{snd} und die Kante l_{syn} besitzen die gleichen Eigenschaften, die in Abschnitt 5.4.1.1 bei der Materialisierung von Invoke-Receive-Interaktionen beschrieben wurden.

Die Pick-Aktivität rcv_{cp} mit dem Nachrichtenereignis e^{msg} wird aus dem Teilnehmer-Scope des empfangenden Teilnehmers $\mathfrak{p}_{rcv}$ entfernt. Sie wird durch den Scope s_{rcv} ersetzt, der das Verhalten des Nachrichtenereignisses

und des Picks emuliert. Der Scope besitzt die folgenden Kontrollflusseigenschaften:

1. $\textsc{Parent}(s_{rcv}) = \textsc{Parent}(rcv_{cp})$ mit $rcv_{cp} = \mathsf{actCopy}(\mathfrak{p}_{rcv}, \bot, \textsc{Parent}(e^{msg}))$
2. $hr_{fhMsg} = (s_{rcv}, e^{fault}, ch_{msg})$
3. $HR_{fhEvent} := \{hr \in HR \mid \pi_1(hr) = s_{rcv} \wedge \pi_2(hr) \in E^{fault} \wedge \pi_3(hr) \in Ch_{event} \cup \{ch_{msg}\}\}$
4. $\textsc{Incoming}_L(s_{rcv}) = \textsc{Incoming}_L(rcv_{cp})$
5. $\textsc{Outgoing}_L(s_{rcv}) = \textsc{Outgoing}_L(rcv_{cp})$
6. $\mathsf{joinCond}(s_{rcv}) = \mathsf{joinCond}(rcv_{cp})$
7. $\textsc{child}_{pr}(s_{rcv}) = f$ mit $f \in A_{\texttt{flow}}$
8. $\textsc{Children}(f) = \{syn_{rcv}\} \cup A_{throw} \cup A_{timer}$
   a) $syn_{rcv} \in A_{\texttt{throw}}$
   b) $A_{throw} := \{a \in A_{\texttt{throw}} \mid \exists! hr_{fhEvent} \in HR_{fhEvent} : \mathsf{throws}(a) = \pi_2(hr_{fhEvent})\}$
   c) $A_{timer} := \{a \in A_{\texttt{wait}} \mid \exists! e \in E^{rcv} : \mathsf{timer}(a) = e\}$
9. $\textsc{Incoming}_L(syn_{rcv}) = \textsc{Incoming}_L(rcv_{cp}) \cup \{l_{syn}\}$
10. $L_{timer} := \{l \in L \mid \pi_1(l) \in A_{timer} \wedge \pi_2(l) \in A_{throw}\}$

Anschaulich ist das Ergebnis der Materialisierung der Interaktion aus Abbildung 5.17 in Abbildung 5.18 dargestellt. Der Scope s_{rcv} erbt die Kontrollflusseigenschaften des Picks, d.h. er befindet sich in derselben Elternaktivität wie das Pick und hat die gleichen Vorgänger- und Nachfolgeaktivitäten.

Der Scope besitzt eine Menge von Fault-Handlern, von denen fh^{msg} (formal durch hr_{fhMsg} repräsentiert) die Kindaktivität ch_{msg} enthält, die im Pick dem Nachrichtenereignis e^{msg} zugeordnet war. Der Fault-Handler wird durch ein Fehlerereignis aktiviert, das von einer sich in s_{rcv} bzw. dessen Kindaktivität f (nicht in Abbildung 5.18 dargestellt) befindlichen Throw-Aktivität syn_{rcv} geworfen wird, die wiederum über die Kontrollflusskante l_{syn} mit dem Assign syn_{snd} verbunden ist. Das heißt, dass das Throw ausgeführt wird und damit

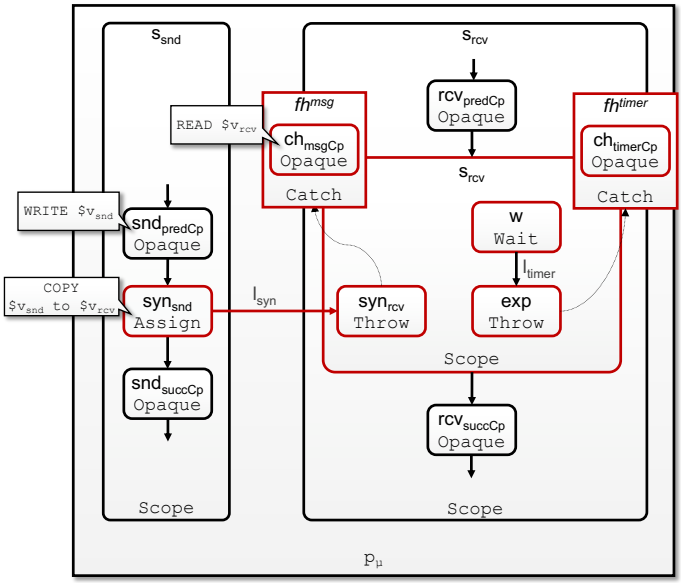

Abbildung 5.18.: Materialisierte Interaktionen zwischen Invoke-Aktivität und Nachrichtenereignis

ebenfalls die Aktivität ch_{msg}, nachdem das Assign syn_{snd} die Nachrichtenübertragung emuliert hat. Wie bereits bei der Materialisierung von Invoke-Receive-Interaktionen erläutert, gewährleistet die Transitionsbedingung der Kante l_{syn}, dass sie zur Laufzeit nur aktiviert wird, wenn die Kommunikation zwischen $\mathfrak{p}_{snd}$ und $\mathfrak{p}_{rcv}$ emuliert wird.

Enthält ein Pick weitere Nachrichten- und Timer-Ereignisse, wird für jedes dieser Ereignisse in s_{rcv} ein separater Fault-Handler erstellt. Diese werden durch die Hierarchiebeziehungen $HR_{fhEvent}$ repräsentiert. Jeder dieser Fault-Handler enthält die Kindaktivität, die im Pick dem jeweiligen Ereignis zugeordnet war. Ausgelöst werden diese Aktivitäten ebenfalls durch eine entsprechende Throw-Aktivität. In Abbildung 5.18 wird daher neben dem Fault-Handler fh^{msg} zusätzlich der Fault-Handler fh^{timer} mit der Aktivität $ch_{timerCp}$ erstellt.

Die Kindaktivitäten, die im Pick durch Timer-Ereignissen aktiviert wurden, dürfen erst nach dem Ablauf der im Ereignis definierten Zeitspanne ausgeführt werden. Dazu wird jeder Throw-Aktivität, die über den Fault-Handler die Ausführung dieser Aktivitäten auslöst (im Beispiel Aktivität *exp*), mittels einer Kontrollflusskante $l_{timer} \in L_{timer}$ eine Wait-Aktivität $w \in A_{timer}$ als Vorgänger hinzugefügt, die erst beendet wird, wenn das Timer-Ereignis eingetreten ist.

Während der Ausführung des Scopes s_{rcv} kann immer nur genau ein Fault-Handler aktiviert und damit auch nur eine Kindaktivität ausgelöst werden. Der Scope s_{rcv} emuliert so das Verhalten des Picks rcv_{cp}, wo ebenfalls nur eine Kindaktivität ausgeführt werden kann. Wird z.B. in Abbildung 5.18 der Fault-Handler fh^{msg} aktiviert, wird der Fault-Handler fh^{timer} deaktiviert und die Kindaktivität $ch_{timerCp}$ kann, wie im Pick in Abbildung 5.17, nicht mehr ausgeführt werden. Konkret wird dies in Abschnitt 5.5.4 erläutert.

5.5.2. Alternative Umsetzung der Materialisierung

Ein Nachteil der im vorherigen Abschnitt skizzierten Materialisierung ist, dass der in der Choreographie reguläre Kontrollfluss, der das Auslösen von Ereignissen umsetzt, in p_μ über Fehlerereignisse emuliert wird. Dies kann die Überwachung der Prozessausführung von p_μ erschweren, da während des Monitorings oder in Audit Logs [LR00] Fehler angezeigt werden, obwohl die jeweilige Instanz von p_μ dem intendierten regulären Prozessablauf gefolgt ist. Dies wird umso problematischer, wenn von der Workflow-Engine automatisch Aktionen ausgelöst werden, wie zum Beispiel der Versand einer Mail an einen Prozessverantwortlichen, wenn Aktivitätsinstanzen den Zustand *faulted* erreichen.

Eine alternative und intuitivere Möglichkeit zur Materialisierung der Interaktion, die nur den regulärer Kontrollfluss nutzt, ist in Abbildung 5.19 dargestellt. Das Pick wird auch dort von einem Scope s_{rcv} ersetzt. Allerdings wird ein Event-Handler genutzt, um die mit den Timer-Ereignissen asso-

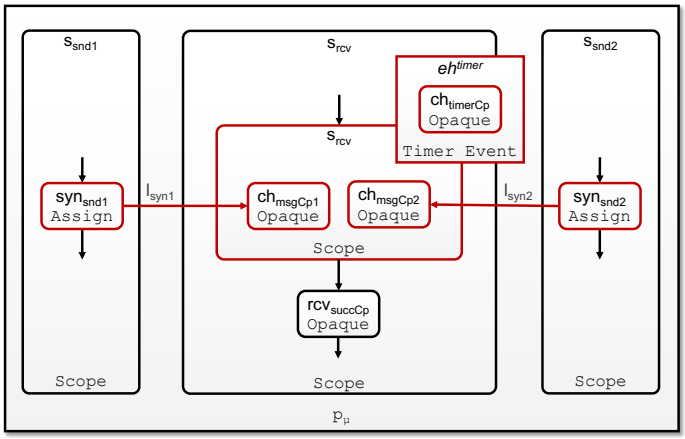

Abbildung 5.19.: Alternativer Ansatz zur Materialisierung von Interaktionen zwischen Invoke-Aktivität und Nachrichtenereignis

ziierten Aktivitäten, wie zum Beispiel $ch_{timerCp}$, nach dem Eintreten eines Ereignisses auszuführen. Die Aktivitäten, die den Nachrichtenereignissen zugeordnet sind, wie zum Beispiel ch_{msgCp1} und ch_{msgCp2}, werden direkt ausgeführt, wenn das Senden der Nachricht mit syn_{snd} emuliert wurde. Der Nachteil an diesem Ansatz ist, dass der Scope s_{rcv} erst beendet wird, wenn alle eingehenden Kontrollflusskanten evaluiert wurden. Ein Ziel des Muster „One-from-many Receive“ ist der Empfang der Nachrichten von einem oder mehreren Teilnehmern. Dies würde, wie in Abbildung 5.19 dargestellt, nach der Materialisierung zu mehreren in s_{rcv} eingehende Kontrollflusskanten l_{syn} führen, wobei jede dieser Kanten den Versand einer Nachricht von einem der Teilnehmer emuliert. Der Scope s_{rcv} würde erst den Zustand *completed* erreichen können, wenn alle diese Kanten evaluiert wären, da nur dann alle seine Kindaktivitäten einen Endzustand erreicht hätten. Dies kann die Ausführung der nachfolgenden Aktivität rcv_{succCp}, abhängig von der Anzahl der sendenden Teilnehmer und wie diese modelliert sind, stark verzögern. Dies ist unabhängig davon, ob der Empfang einer Nachricht oder das Auslösen eines Timer-Ereignisses emuliert wurde. Aus diesem Grund wird hier der

im vorherigen Abschnitt erläuterte Ansatz weiter verfolgt. Da der Scope s_{rcv} nach der Emulation des Empfangs einer Nachricht oder dem Auslösen eines Timer-Ereignisses durch den Fehler sofort beendet wird, kommt es dort nicht zu einer verzögerten Ausführung der Nachfolgeaktivitäten.

5.5.3. Materialisierungsalgorithmus

Die Schritte zur Materialisierung einer Interaktion zwischen einem Invoke *snd* und Nachrichtenereignis e^{msg} sind formal durch Algorithmus 5.5 und Algorithmus 5.6 beschrieben. Einstiegspunkt für die Materialisierung ist dabei die Prozedur MaterializeInvokeMsgEvent in 5.5, die für jede dieser Interaktionen von der Prozedur Materialization (Algorithmus 5.3) aufgerufen wird.

Analog zu Algorithmus 5.4, wird in den Zeilen 5 bis 13 die Invoke-Aktivität *snd* durch die Synchronisationsaktivität syn_{snd} ersetzt, falls sie aufgrund von multilateralen Interaktionen nicht schon vorher ersetzt wurde.

Um die Pick-Aktivität durch einen Scope s_{rcv} zu ersetzen und die dem Nachrichtenereignis zugeordnete Kindaktivität in einen Fault-Handler zu transferieren, wird in Zeile 15 die Funktion ReplacePickWithScope in Algorithmus 5.6 aufgerufen. Enthält das Pick mehrere Nachrichtenereignisse, die mit verschiedenen Nachrichtenkanten verbunden sind, wird die Prozedur MaterializeInvokeMsgEvent und somit auch die Prozedur ReplacePickWithScope für jedes Nachrichtenereignis über die Schleife in der Prozedur Materialization für dasselbe Pick mehrmals aufgerufen. Um zu verhindern, dass für das gleiche Pick mehrere Scopes erstellt werden, wird in Zeile 2 bzw. 3 über die Abbildung $\text{pickScope} : A_{\texttt{pick}} \rightarrow A_{\texttt{scope}}$ geprüft, ob für das jeweilige Pick bereits ein Scope erstellt wurde. Ist dies nicht der Fall, wird ein entsprechender Scope s_{rcv} erstellt, der im Kontrollfluss von p_{μ} das Pick ersetzt (Zeile 5).

Mittels der Schleife wird in s_{rcv} für jedes Ereignis im Pick in Zeile 12 ein Fault-Handler erstellt (in Abschnitt 5.5.1 als hr_{fhMsg} und $HR_{fhEvent}$ bezeichnet), in den die dem jeweiligen Ereignis zugeordnete Aktivität transferiert wird.

Algorithmus 5.5 Materialisierung von Interaktionen zwischen Invoke-Aktivitäten und Nachrichtenereignissen

```
 1: procedure MaterializeInvokeMsgEvent(𝔭_snd, snd_cp, 𝔭_rcv, e^msg)
 2:     snd_cp ← actCopy(𝔭_snd, ⊥, snd)
 3:     e^msg_cp ← eventCopy(𝔭_rcv, ⊥, e^msg)
 4:     syn_snd ← ⊥
 5:     if com2Syn(snd_cp) = ⊥ then
 6:         syn_snd ← CreateActivity(assign)
 7:         assign_src(syn_snd) ← inputVar(snd)
 8:         assign_trg(syn_snd) ← outputVar(e^msg)
 9:         ReplaceActivity(snd_cp, syn_snd)
10:         com2Syn(snd_cp) ← syn_snd
11:     else
12:         syn_snd ← com2Syn(snd)
13:     end if
14:     rcv_cp ← Parent(e^msg)
15:     ReplacePickWithScope(rcv_cp)
16:     syn_rcv ← com2Syn(e^msg_cp)
17:     l_syn ← CreateControlLink(syn_snd, syn_rcv, receiver(syn_snd) = 𝔭_rcv)
18: end procedure
```

In Abbildung 5.18 sind dies, wie oben erwähnt, die zwei Fault-Handler fh^{msg} und fh^{timer}.

Für Aktivitäten, die im Pick einem Timer-Ereignis zugeordnet sind, wird in den Zeilen 14 bis 19 zusätzlich die Wait-Aktivität als Vorgängeraktivität für das Throw erstellt.

Alle Throw-Aktivitäten, die durch ein Nachrichtenereignis aktiviert werden, werden in Zeile 24 dem Nachrichtenereignis, dessen Aktivierung sie emulieren, zugewiesen. Damit kann in der Prozedur MaterializeInvokeMsgEvent in Zeile 17 das Assign syn_{snd} über die Kontrollflusskante l_{syn} mit dem entsprechenden Throw syn_{rcv}, das das Ereignis e^{msg} emuliert, verbunden werden. Existieren mehrere Teilnehmer, die über die gleiche Nachrichtenkante über ein Invoke eine Nachricht an dasselbe Nachrichtenereignis senden, wird von syn_{snd} aus jedem Teilnehmer-Scope eine Kontrollflusskante l_{syn} erstellt,

Algorithmus 5.6 Ersetzen von Nachrichtenereignissen und Pick-Aktivitäten

1: **procedure** REPLACEPICKWITHSCOPE(rcv)
2: $s_{rcv} \leftarrow \mathsf{pickScope}(rcv)$
3: **if** $s_{rcv} = \bot$ **then**
4: $s_{rcv} \leftarrow$ CREATEACTIVITY(`scope`)
5: REPLACEACTIVITY(rcv_{cp}, s_{rcv})
6: $f \leftarrow$ CREATEACTIVITY(`flow`)
7: CREATEHR(s_{rcv}, t^{ch}_{HR}, f)
8: $A_{children} = \emptyset$
9: **for all** $hr_{\texttt{pick}} \in \{hr \in HR \mid \pi_1(hr) = rcv_{cp} \wedge \pi_2(hr) \in E^{std}\}$ **do**
10: $e_{fault} = new\ t^{fault}_{E}$
11: $ch_{event} \leftarrow \pi_3(hr_{\texttt{pick}})$
12: CREATEHR($s_{rcv}, e_{fault}, ch_{event}$)
13: **if** $\pi_2(hr_{\texttt{pick}}) \in E^{timer}$ **then**
14: $exp \leftarrow$ CREATEACTIVITY(`throw`)
15: $\mathsf{throws}(exp) = e_{fault}$
16: $w \leftarrow$ CREATEACTIVITY(`wait`)
17: $\mathsf{timer}(w) \leftarrow \pi_2(hr_{\texttt{pick}})$
18: $A_{children} \leftarrow \{exp, w\}$
19: CREATECONTROLLINK($w, exp, \bot$)
20: **else**
21: $syn_{rcv} \leftarrow$ CREATEACTIVITY(`throw`)
22: $\mathsf{throws}(syn_{rcv}) = e_{fault}$
23: $A_{children} \leftarrow \{syn_{rcv}\}$
24: $\mathsf{com2Syn}(\pi_2(hr_{\texttt{pick}})) \leftarrow syn_{rcv}$
25: **end if**
26: ADDCHILDACTIVITIES($f, A_{children}$)
27: **end for**
28: **end if**
29: **end procedure**

dessen Zielaktivität syn_{rcv} ist. Die Eintrittsbedingung von syn_{rcv} muss nicht angepasst werden, da die implizite Eintrittsbedingung definiert, dass die Aktivität ausgeführt wird, sobald eine dieser Kanten aktiviert wurde (siehe Abschnitt 3.2.3.2).

5.5.4. Zustandsbeziehungen zwischen den Geschäftsaktivitäten

Wie für die Invoke-Receive-Interaktionen wird hier untersucht, ob die Duplikate in p_μ nach der Materialisierung verhaltensäquivalent zu ihren originalen Geschäftsaktivitäten aus $\mathfrak{p}_{snd}$ und $\mathfrak{p}_{rcv}$ sind. Dazu werden wieder die Zustandsbeziehungen zwischen allen Geschäftsaktivitäten verglichen, die direkt von der Materialisierung betroffen sind, weil sie ich in der gleichen Elternaktivität wie das Invoke *snd* und das Pick *rcv* befinden. Dies schließt auch die Geschäftsaktivitäten mit ein, die Kindaktivitäten von *rcv* sind.

- $A_{snd} := \{a \in A_{\texttt{opaque}} \mid a \in \textsc{Children}(\textsc{Parent}(snd))\}$
- $A_{rcv} := \{a \in A_{\texttt{opaque}} \mid a \in (\textsc{Children}(\textsc{Parent}(rcv)) \cup \textsc{Children}(rcv))\}$

Zur Vereinfachung wird wieder davon ausgegangen, dass die Sendeaktivität *snd* jeweils genau eine direkte Vorgängeraktivität snd_{pred}, eine direkte Nachfolgeaktivität snd_{succ} und eine parallele Aktivität snd_{par} besitzt, die sich mit ihr in der gleichen Elternaktivität befinden:

$$A_{snd} := \{snd_{pred}, snd_{succ}, snd_{par}\}$$

Für die Aktivitäten gelten also die folgenden Beziehungen zu *snd*:

- $snd_{pred} \in A_{snd} \cap \textsc{Predecessors}(snd) \wedge |\textsc{Successors}(snd_{pred})| = 1$
- $snd_{succ} \in A_{snd} \cap \textsc{Successors}(snd) \wedge |\textsc{Predecessors}(snd_{succ})| = 1$
- $snd_{par} \in A_{snd} \cap \textsc{Successors}_{all}(snd) \setminus \textsc{Predecessors}_{all}(snd)$

Analog dazu hat *rcv* eine direkt Vorgängeraktivität snd_{pred}, eine direkte

Nachfolgeaktivität snd_{succ} und eine parallele Aktivität snd_{par}. Zusätzlich besitzt rcv noch Kindaktivitäten Ch_{rcv}.

$$A_{rcv} := \{rcv_{pred}, rcv_{succ}, rcv_{par}\} \cup Ch_{rcv}$$

Formal sind die Beziehungen der Aktivitäten zu rcv also wie folgt:

- $rcv_{pred} \in A_{rcv} \cap \textsc{Predecessors}(rcv) \wedge |\textsc{Successors}(rcv_{pred})| = 1$
- $rcv_{succ} \in A_{rcv} \cap \textsc{Successors}(rcv) \wedge |\textsc{Predecessors}(rcv_{succ})| = 1$
- $rcv_{par} \in A_{rcv} \cap \textsc{Successors}_{all}(rcv) \setminus \textsc{Predecessors}_{all}(rcv)$
- $Ch_{rcv} := \{ch_{msg}\} \cup Ch_{event}$ (siehe Definition in Abschnitt 5.5)

Die Aktivität ch_{msg} ist dem Empfangsereignis von rcv und die Kindaktivitäten Ch_{event} sind anderen Ereignissen von rcv zugeordnet.

Das Ergebnis der Untersuchung ist in Abbildung 5.20 zusammengefasst. Die Materialisierung von Interaktionen zwischen Invoke-Aktivitäten und Nachrichtenereignissen erhalten alle Zustandsbeziehungen zwischen den betroffenen Geschäftsaktivitäten in p_μ. Die Beziehungen werden im Detail in den folgenden Abschnitten diskutiert. Die Duplikate der Aktivitäten aus A_{snd} bzw. A_{rcv} in p_μ werden wieder mit dem Subskript cp gekennzeichnet.

(1) Beziehungen von parallelen Aktivitäten snd_{par} bzw. rcv_{par}

Die Aktivität snd_{par} hat zu snd_{pred} bzw. snd_{succ} die gleichen Kontrollfluss- und damit auch die gleichen Zustandsbeziehungen, die in Abschnitt 5.4.1.3 für Invoke-Receive-Interaktionen beschrieben wurden. Analog verhält es sich mit den Beziehungen der Aktivität rcv_{par} zu den Aktivitäten rcv_{pred} bzw. rcv_{succ}. Es gilt also zwischen ihnen das Profil für Kindaktivitäten von strukturierten Aktivitäten aus Abbildung 4.10. Dieses Profil gilt ebenfalls zwischen den Aktivitäten aus Ch_{rcv} und rcv_{par}, da es sich bei ersteren um Kindaktivitäten des zu rcv_{par} parallelen Picks handelt.

			A_{snd}			A_{rcv}				
						Ch_{rcv}				
			snd_{par}	snd_{pred}	snd_{succ}	ch_{msg}	Ch_{event}	rcv_{par}	rcv_{pred}	rcv_{succ}
A_{snd}	snd_{par}			✓(1)	✓(1)	✓(11)	✓(12)	✓(6)	✓(3)	✓(5)
	snd_{pred}		✓(1)		✓(2)	✓(9)	✓(10)	✓(6)	✓(3)	✓(4)
	snd_{succ}		✓(1)	✓(2)		✓(15)	✓(16)	✓(6)	✓(3)	✓(7)
A_{rcv}	Ch_{rcv}	ch_{msg}	✓(11)	✓(9)	✓(15)		✓(8)	(1)	✓(13)	✓(14)
		Ch_{event}	✓(12)	✓(10)	✓(16)	✓(8)	✓(8)	(1)	✓(12)	✓(14)
	rcv_{par}		✓(6)	✓(6)	✓(6)	✓(1)	✓(1)		✓(1)	✓(1)
	rcv_{pred}		✓(3)	✓(3)	✓(3)	✓(13)	✓(13)	✓(1)		✓(2)
	rcv_{succ}		✓(5)	✓(4)	✓(7)	✓(14)	✓(14)	✓(1)	✓(2)	

Abbildung 5.20.: Verhaltensäquivalenz zwischen den Geschäftsaktivitäten in p_μ nach der Materialisierung von Invoke-Aktivitäten und Nachrichtenereignissen

Dasselbe Profil gilt auch zwischen deren Duplikaten snd_{parCp} und snd_{predCp} bzw. snd_{succCp} sowie zwischen den Duplikaten rcv_{parCp} und rcv_{predCp} bzw. rcv_{succCp}. Zwischen ihnen wurden keine neuen Kontrollflusskonstrukte erzeugt. Die Aktivitäten Ch_{rcv} befinden sich nach der Materialisierung zwar in einem Scope, dieser ist aber weiterhin parallel zur Aktivität rcv_{parCp}. Daher bleiben auch zwischen rcv_{parCp} und Ch_{rcv} die Beziehungen erhalten.

(2) Beziehungen zwischen den Aktivitäten snd_{pred} und snd_{succ} bzw. rcv_{pred} und rcv_{succ}

Zwischen snd_{pred} und snd_{succ} gelten die gleichen Kontrollflussbeziehungen und damit auch dieselben in Abbildung 5.8 dargestellten Zustandsbeziehungen, die für diese Aktivitäten bei der Invoke-Receive-Interaktion gelten. Algorithmus 5.5 führt auf der Senderseite die gleichen Materialisierungsschritte aus wie Algorithmus 5.4 für Invoke-Receive-Interaktionen. Daher werden auch für snd_{predCp} und snd_{succCp} die Beziehungen beibehalten.

Zwischen den Aktivitäten rcv_{pred} und rcv_{succ} gelten die in Abbildung 5.8 für die Invoke-Receive-Interaktion dargestellten Zustandsbeziehungen. Die Aktivität *rcv* ist zwar eine strukturierte Aktivität, aber rcv_{pred} und rcv_{succ} fungieren

		rcv_{succCp}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
rcv_{predCp}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	→	→	⊕	⊕	⊕	⊕
	ABORTED	←	⊕	‖	⊕	⊕	⊕	⊕
	EXECUTING	←	⊕	→	→	→	→	→
	TERMINATED	←	⊕	‖	⊕	⊕	⊕	⊕
	FAULTED	←	⊕	→	⊕	⊕	⊕	⊕
	COMPLETED	←	⊕	→	→	→	→	→

Abbildung 5.21.: Zustandsprofil zwischen rcv_{predCp} und rcv_{succCp}

weiterhin als direkter Vorgänger bzw. Nachfolger für diese Aktivität. Nach der Materialisierung sind die Duplikate rcv_{predCp} und rcv_{succCp} weiterhin transitiv über s_{rcv} miteinander verbunden. Die neue Kontrollflusskante l_{syn} führt hier, im Gegensatz zur materialisierten Invoke-Receive-Interaktion, allerdings nicht zu einer Synchronisationsabhängigkeit zwischen syn_{snd} und rcv_{succCp}, da sich ihre Zielaktivität syn_{rcv} im Scope s_{rcv} befindet. Erreicht also eine Instanz von rcv_{predCp} den Zustand *executing*, geht im regulären Kontrollfluss auch die Instanz von s_{rcv} und damit auch die Instanz von rcv_{succCp} in *executing* (Axiom *Sq1*), unabhängig davon, welchen Zustand syn_{snd} erreicht hat. Geht die Instanz von rcv_{predCp} wiederum in *dead*, wird auch die Instanz von s_{rcv} und die Instanz von rcv_{succCp} in *dead* gesetzt (Axiom *Sq2*). Ein Fehler oder die Terminierung der Instanz von rcv_{predCp} führt wiederum zur vorzeitigen Terminierung der Instanz von rcv_{succCp}, da sich beide in derselben Elternaktivität befinden (Axiome $St_{CH2CH}2$ bzw. $St_{CH2CH}3$). Daher gilt zwischen rcv_{predCp} und rcv_{succCp} das Profil in Abbildung 5.21, das dem Profil der originalen Aktivitäten aus Abbildung 5.8 entspricht.

(3) Beziehungen zwischen den Aktivitäten in A_{snd} und rcv_{pred}

Wie für die Invoke-Receive-Interaktion beschrieben, sind die Aktivitäten in A_{snd} vollständig von der Aktivität rcv_{pred} isoliert. Diese Isolation wird auch von den Aktivitäten A_{sndCp} und rcv_{predCp} beibehalten.

		rcv_{succ}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
snd_{pred}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	‖	‖	‖	‖	‖	‖
	ABORTED	←	‖	‖	‖	‖	‖	‖
	EXECUTING	←	‖	‖	‖	‖	‖	‖
	TERMINATED	←	‖	‖	‖	‖	‖	‖
	FAULTED	←	‖	‖	‖	‖	‖	‖
	COMPLETED	←	‖	‖	‖	‖	‖	‖

Abbildung 5.22.: Zustandsprofil zwischen snd_{pred} und rcv_{succ}

(4) Beziehungen zwischen den Aktivitäten snd_{pred} und rcv_{succ}

Zwischen snd_{pred} und rcv_{succ} gilt das in Abbildung 5.22 dargestellte Profil, in dem zwischen allen Zuständen bis auf *initial* die Beziehung ‖ besteht. Dies liegt neben der Teilnehmerisolation daran, dass das Pick und somit auch rcv_{succ} keine Synchronisationsabhängigkeit zu snd_{pred} haben, und dass das Pick auch andere Ereignisse als das Nachrichtenereignis e^{msg} auslösen kann. Wird im Pick zum Beispiel ein anderes Ereignis ausgelöst, bevor die Instanz von snd_{pred} in den Zustand *completed* geht und eine Nachricht gesendet wird, kann auch die Instanz von rcv_{succ} vor snd_{pred} den Zustand *executing* und dessen Nachfolgezustände erreichen. Trotzdem besteht aufgrund der Nachrichtenkante keine vollständige Zustandsisolation zwischen snd_{pred} und rcv_{succ}. Geht snd_{pred} in *completed*, wird über die gesendete Nachricht das Nachrichtenereignis ausgelöst, das die Ausführung von ch_{msg} startet (Axiom $Ml_{InvPck}1$). In diesem Szenario wird also der Zustand der Pick-Instanz von dem seiner Kindaktivität ch_{msg} beeinflusst, deren Zustand wiederum von der Instanz von snd_{pred} abhängt. Folglich wird der Zustand von rcv_{succ} über die Pick-Instanz vom Zustand der Instanz von snd_{pred} beeinflusst. Geht zum Beispiel ch_{msg} aufgrund der Nachricht in den Zustand *faulted*, geht die Pick-Instanz in *faulted* (Axiom $St_{CH}4$) und auch rcv_{succ} wird terminiert (Axiom $St_{CH2CH}4$).

Die Fehlerisolation der Duplikate snd_{predCp} und rcv_{succCp} besteht in p_{μ} durch

die Teilnehmer-Scopes weiterhin. Die Aktivitäten s_{rcv} und rcv_{succCp} haben ebenfalls keine Synchronisationsabhängigkeit zu snd_{predCp}. Daher können die Ereignisse über die Wait- und Throw-Aktivitäten und die Fault-Handler unabhängig vom Zustand von snd_{predCp} bzw. syn_{snd} emuliert werden. Eine Ausnahme bildet das Throw syn_{rcv}, das die Ausführung von ch_{msgCp} nur starten kann, wenn die Instanz von snd_{predCp} in *completed* gesetzt und vorher keine andere Kindaktivität von s_{rcv} ausgeführt wurde. In diesem Fall besteht zwischen snd_{predCp} und rcv_{succCp} wie bei den originalen Aktivitäten eine Zustandsabhängigkeit. Dadurch dass rcv_{succCp} alle Zustände auch unabhängig vom Zustand von snd_{predCp} erreichen kann, gilt zwischen diesen Aktivitäten ebenfalls das Profil aus Abbildung 5.22. Dies ist ein Unterschied zur materialisierten Invoke-Receive-Interaktion, wo das Profil von snd_{predCp} und rcv_{succCp} nicht dem von snd_{pred} und rcv_{succ} entsprach.

(5) Beziehungen zwischen den Aktivitäten in snd_{par} und rcv_{succ}

Im regulären Kontrollfluss hat die Instanz der zur Sendeaktivität *snd* parallelen Aktivität snd_{par} keinen Einfluss auf *snd* und damit auch nicht auf rcv_{succ}. Die Aktivitätsinstanz von snd_{par} kann einen Fehler auslösen, wenn *snd* noch nicht ausgeführt wurde. Bei Invoke-Receive-Interaktionen kann dies dazu führen, dass auch *snd* terminiert wird, falls es den Zustand *completed* noch nicht erreicht hat und damit die Instanz von rcv_{succ} den Zustand *executing* nicht erreichen kann. Hier führt ein Fehler in der Instanz von snd_{par} nicht zwangsläufig dazu, dass die Instanz von rcv_{succ} nicht *executing* erreichen kann, da ein anderes Ereignis des Picks ausgelöst und die Pick-Instanz somit erfolgreich beendet werden kann. Daher kann rcv_{succ} anders als bei Invoke-Receive-Interaktionen auch den Zustand *executing* erreichen, wenn die Instanz von snd_{par} in *faulted* geht, bevor *snd* ausgeführt wurde. Es gilt also deshalb zwischen snd_{par} und rcv_{succ} dasselbe Profil aus Abbildung 5.22, das zwischen den Zuständen von snd_{predCp} und rcv_{succCp} gilt, also zwischen allen Zuständen außer *initial* die Beziehung ||.

Dieses Profil wird auch zwischen snd_{parCp} und rcv_{succCp} beibehalten. Im re-

gulären Kontrollfluss beeinflusst eine Instanz von snd_{parCp} die Instanz von rcv_{succCp} über syn_{snd} nicht. Wird syn_{snd} durch einen Fehler in der Instanz von snd_{parCp} terminiert, resultiert dies, wie im vorherigen Abschnitt erläutert, nicht unbedingt darin, dass die Instanz von rcv_{succCp} einen Fehlerzustand erreicht.

(6) Beziehungen zwischen den Aktivitäten rcv_{par} und A_{snd}

Aus den gleichen Gründen wie bei der Invoke-Receive-Interaktion sind die Aktivitäten rcv_{par} und A_{snd} vollständig voneinander isoliert. Dies gilt auch für deren Duplikate rcv_{parCp} und A_{sndCp}.

(7) Beziehungen zwischen den Aktivitäten snd_{succ} und rcv_{succ}

Wie oben für die Beziehungen zwischen snd_{pred} und rcv_{succ} erläutert, hat rcv_{succ} keine Synchronisationsabhängigkeit zu snd_{pred}, wobei snd_{succ} diese Abhängigkeit zu snd_{pred} besitzt. Es existiert daher anders als bei den Invoke-Receive-Interaktionen auch kein impliziter Zusammenhang zwischen den Zuständen von snd_{succ} und rcv_{succ}. Da auch keine weiteren transitiven Abhängigkeiten über ml bestehen, gilt zwischen ihnen das Profil für vollständig isolierte Aktivitäten.

Das Duplikat snd_{succCp} hat nach der Materialisierung über l_{syn} ebenfalls keine Synchronisationsabhängigkeit zu rcv_{succCp}. Es bestehen weiterhin auch keine impliziten Abhängigkeiten über snd_{predCp} zwischen den Duplikaten, da rcv_{succCp} weiterhin keine Synchronisationsabhängigkeit zu snd_{predCp} hat. Folglich gilt auch zwischen snd_{succCp} und rcv_{succCp} das Profil für vollständig isolierte Aktivitäten.

(8) Beziehungen zwischen den Aktivitäten aus Ch_{rcv}

Zwischen den den Ereignissen des Picks zugeordneten Geschäftsaktivitäten Ch_{rcv} gelten die in Abbildung 4.40 dargestellten Zustandsbeziehungen. Das

Hauptmerkmal der Beziehungen ist, dass die Instanz genau einer Aktivität aus Ch_{rcv} den Zustand *executing* erreichen kann.

Die Teilnehmerduplikate Ch_{rcvCp} der Aktivitäten Ch_{rcv} befinden sich in p_μ in jeweils einem separaten Fault-Handler des Scopes s_{rcv}. Das Zustandsprofil zwischen ch_{cp_i} und ch_{rcvCp_j} ($ch_{rcvCp_i}, ch_{rcvCp_i} \in Ch_{rcvCp}$) entspricht also dem der Fehlerbehandlungsaktivitäten in Abbildung 4.14. Da zur Laufzeit nur ein Fault-Handler von s_{rcv} aktiviert werden kann, kann auch wie beim Pick nur eine Instanz der Aktivitäten in Ch_{rcvCp} den Zustand *executing* erreichen (Axiom Sc_{FH2FH}). Da sich die Profile zwischen den Kindaktivitäten des Picks und den Fehlerbehandlungsaktivitäten der Fault-Handler gleichen, gelten zwischen den Aktivitäten in Ch_{rcv} und Ch_{rcvCp} die gleichen Zustandsbeziehungen.

(9) Beziehungen zwischen den Aktivitäten snd_{pred} und ch_{msg}

Die Aktivitäten snd_{pred} und ch_{msg} sind über die Aktivität *snd* mittels einer Kontrollfluss- und einer Nachrichtenkante miteinander verbunden. Die Zustandsbeziehungen zwischen den beiden Aktivitäten können daher transitiv über die Beziehungen zwischen snd_{pred} und *snd* (siehe Abschnitt 4.2.9) sowie denen zwischen *snd* und ch_{msg} (siehe Abschnitt 4.2.15.2) abgeleitet werden.

Eine Instanz der Aktivität ch_{msg} erreicht den Zustand *executing* und dessen Nachfolgezustände, nachdem eine Instanz der Aktivität *snd* den Zustand *completed* erreicht hat (Axiom $Ml_{InvPck}1$). Eine Instanz von *snd* kann diesen Zustand wiederum nur erreichen, nachdem eine Instanz von snd_{pred} *completed* erreicht hat (Axiom *Sq1*). Daher kann eine Instanz von ch_{msg} nur in den Zustand *executing* gehen, nachdem eine Instanz von snd_{pred} *completed* erreicht hat. Geht eine Instanz von snd_{pred} in einen anderen Endzustand als *completed*, kann auch die Instanz von *snd* den Zustand *completed* nicht mehr erreichen (Axiom *Sq3*). Folglich kann die Instanz von ch_{msg} den Zustand *executing* nicht mehr erreichen (Axiom $Ml_{InvPck}2$). Durch die Isolation von snd_{pred} und ch_{msg}

		ch_{msg}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
snd_{pred}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	‖	‖	⊕	⊕	⊕	⊕
	ABORTED	←	‖	‖	⊕	⊕	⊕	⊕
	EXECUTING	←	‖	‖	→	→	→	→
	TERMINATED	←	‖	‖	⊕	⊕	⊕	⊕
	FAULTED	←	‖	‖	⊕	⊕	⊕	⊕
	COMPLETED	←	‖	‖	→	→	→	→

Abbildung 5.23.: Zustandsprofil zwischen snd_{pred} und ch_{msg} in der Choreographie

besteht keine Reihenfolge zwischen *dead* bzw. *aborted* von ch_{msg} und den Nachfolgezuständen von *initial* der Aktivität snd_{pred} (Axiom *Iso*). Das entsprechende Zustandsprofil zwischen snd_{pred} und ch_{msg} ist in Abbildung 5.23 dargestellt.

Die Teilnehmerduplikate snd_{predCp} und ch_{msgCp} haben in p_{μ} eine indirekte Kontrollflussbeziehung über syn_{snd}, der Throw-Aktivität syn_{rcv} und dem Fault-Handler des Scopes s_{rcv} , in dem sich ch_{msgCp} befindet. Dies resultiert in den folgenden Zustandsbeziehungen zwischen snd_{predCp} und ch_{msgCp}. Erreicht die Instanz von snd_{predCp} den Zustand *dead* oder einen Fehlerzustand, wird die Instanz von syn_{snd} ebenfalls in *dead* oder einen Fehlerzustand gesetzt (Axiome *Sq2* bzw. *Sq3*). Dies führt dazu, dass danach die Instanz von syn_{rcv} ebenfalls den Zustand *dead* erreicht (Axiom *Sq2* bzw. Axiom *SqI3*) und somit keinen Fehler auslöst, der vom Fault-Handler des Scopes s_{rcv}, in dem sich ch_{msgCp} befindet, gefangen werden kann. Die Instanz von ch_{msgCp} kann dann nur in den Zustand *dead* gehen, wenn s_{rcv} diesen Zustand erreicht hat oder ein anderer Fault-Handler aktiviert wurde. Aufgrund der Isolation von ch_{msgCp} und snd_{predCp} kann die Instanz von ch_{msgCp} diese Zustände unabhängig vom Zustand der Instanz von snd_{predCp} erreichen (Axiom *Iso*).

Erreicht die Instanz von snd_{predCp} hingegen den Zustand *completed*, kann die Instanz von syn_{snd} danach in *completed* gehen (Axiom *Sq1*) und die Instanz von syn_{rcv} kann aufgrund der Aktivierung der Kontrollflusskante l_{syn}

		ch_{msgCp}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
snd_{predCp}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	‖	‖	⊕	⊕	⊕	⊕
	ABORTED	←	‖	‖	⊕	⊕	⊕	⊕
	EXECUTING	←	‖	‖	→	→	→	→
	TERMINATED	←	‖	‖	⊕	⊕	⊕	⊕
	FAULTED	←	‖	‖	⊕	⊕	⊕	⊕
	COMPLETED	←	‖	‖	→	→	→	→

Abbildung 5.24.: Zustandsprofil zwischen snd_{predCp} und ch_{msgCp}

den Zustand *executing* erreichen (Axiom $Sq1$). Von diesem Zustand geht sie, da es sich um eine Throw-Aktivität handelt, direkt in den Zustand *faulted* (Axiom 4.2.4) und löst somit den Fehler e_{fault} aus. Damit erreicht dann auch die Instanz des Flows f den Zustand *faulted* (Axiom $St_{CH}4$) und propagiert diesen an die Instanz des Scopes s_{rcv}. Dies resultiert in der Ausführung der Instanz von ch_{msgCp} (Axiom $Sc_{PCH2FH}1$). Jede Instanz von ch_{msgCp} erreicht also, wie ihr Original ch_{msg}, weiterhin den Zustand *executing* und dessen Nachfolgezustände, nachdem eine Instanz von snd_{predCp} den Zustand *completed* erreicht hat. Das Zustandsprofil zwischen snd_{predCp} und ch_{msgCp} ist in Abbildung 5.24 dargestellt und entspricht dem zwischen snd_{pred} und ch_{msg} aus Abbildung 5.23.

(10) Beziehungen zwischen den Aktivitäten snd_{pred} und Ch_{event}

Die Aktivitäten snd_{pred} und Ch_{event} befinden sich in unterschiedlichen Teilnehmern und sind nicht direkt über eine Nachrichtenkante miteinander verbunden. Allerdings existiert eine Zustandsbeziehung über die Aktivität ch_{msg}, da snd_{pred} mit dieser über die Nachrichtenkante verbunden ist und sich ch_{msg} wiederum mit Ch_{event} in derselben Elternaktivität befindet. Erreicht eine Instanz von snd_{pred} den Zustand *completed*, kann dies dazu führen, dass die Instanz von ch_{msg} ausgeführt wird und damit die Instanzen der Ch_{event} nicht mehr in *executing* gehen können (Axiom $Pick_{Event}$). Allerdings kann eine

Instanz der Aktivitäten in Ch_{event} auch unabhängig von der Instanz der Aktivität snd_{pred} jeden beliebigen Zustand, wie z.B. *executing*, erreichen, wenn das ihr zugeordnete Ereignis ausgelöst wird. Deshalb gilt zwischen allen Zuständen von snd_{pred} und Ch_{event} (bis auf *initial*) ebenfalls die Beziehung ||. Das Profil entspricht daher dem, das zwischen snd_{pred} und ch_{msg} gilt (siehe Abbildung 5.22).

In p_{μ} sind die Teilnehmerduplikate snd_{predCp} und die Aktivitäten in $Ch_{eventCp}$ weiterhin fehlerisoliert voneinander. Die bei der Materialisierung erstellte Kontrollflusskante l_{syn} stellt aber ebenfalls eine Zustandsbeziehung zwischen snd_{predCp} und $Ch_{eventCp}$ über ch_{msgCp} her. Wie bei der Diskussion der Zustandsbeziehungen zwischen den Aktivitäten in Ch_{rcvCp} erläutert, gilt zwischen allen ihren Zuständen bis auf *initial* die Beziehung ||. Folglich gelten diese Beziehungen auch zwischen snd_{predCp} und $Ch_{eventCp}$.

(11) Beziehungen zwischen den Aktivitäten snd_{par} und ch_{msg}

Zwischen der Aktivität snd_{par} und ch_{msg} gelten, wie in Abbildung 5.25 dargestellt, dieselben Zustandsbeziehungen wie zwischen snd_{par} und rcv_{succ} in Invoke-Receive-Interaktionen. Aufgrund der Parallelität von snd_{par} und *snd* beeinflussen im regulären Kontrollfluss die Instanzen von snd_{par} die von ch_{msg} nicht. Die Instanz von ch_{msg} kann den Zustand *executing* nicht erreichen, wenn die Instanz von snd_{par} einen Fehler auslöst, bevor *snd completed* erreicht hat. Löst die Instanz von snd_{par} danach einen Fehler aus, kann ch_{msg} in *executing* gesetzt werden, sofern das Pick ausgeführt wird (Axiom $St_{CH}3$).

Es gelten nach der Materialisierung zwischen den Duplikaten snd_{parCp} und ch_{msgCp} dieselben Kontrollflussbeziehungen wie zwischen den Duplikaten snd_{parCp} und rcv_{succCp} in der materialisierten Invoke-Receive-Interaktion. Dies liegt daran, dass snd_{parCp} und ch_{msgCp} durch syn_{snd} und die Kontrollflusskante l_{syn} die gleichen Kontrollflussbeziehungen besitzen wie snd_{parCp} und rcv_{succCp}. Es wird hier deshalb auf eine detaillierte Beschreibung der Zustandsbeziehungen verzichtet.

		ch_{msg}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
snd_{par}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	‖	‖	‖	‖	‖	‖
	ABORTED	←	‖	‖	‖	‖	‖	‖
	EXECUTING	←	‖	‖	→	→	→	→
	TERMINATED	←	‖	‖	‖	‖	‖	‖
	FAULTED	←	‖	‖	‖	‖	‖	‖
	COMPLETED	←	‖	‖	→	→	→	→

Abbildung 5.25.: Zustandsprofil zwischen snd_{par} und ch_{msg}

(12) Beziehungen zwischen den Aktivitäten snd_{par} und Ch_{event}

Die Aktivität snd_{par} und die in Ch_{event} befinden sich in verschiedenen Teilnehmern. Sie haben, da sie alternative Aktivitäten zu ch_{msgCp} sind (Axiom $Pick_{Event}$), durch die Nachrichtenkante ml im regulären Kontrollfluss implizite Zustandsbeziehungen zueinander. Es besteht natürlich auch hier die Möglichkeit, dass eine Ereignisbehandlungsaktivität aus Ch_{event} nur deshalb ausgeführt werden kann, weil aufgrund der nicht gesendeten Nachricht ch_{msgCp} nicht in *executing* gehen kann. Da hier aber von konkreten Ereignissen und den Aktivitäten in Ch_{event} abstrahiert wird, kann keine Aussage darüber getroffen werden, ob eine Aktivität aus Ch_{event} nur ausgeführt wird, wenn eine Instanz aus A_{sndPar} den Zustand *faulted* erreicht. Es gilt daher zwischen A_{sndPar} und Ch_{event} ebenfalls zwischen allen Nachfolgezuständen von *initial* die Beziehung ‖.

Diese Zustandsbeziehungen gelten auch zwischen den Duplikaten snd_{parCp} und $Ch_{eventCp}$. Auch hier kann aufgrund der Abstraktion der Ereignisse keine Zustandsabhängigkeit zwischen snd_{parCp} und $Ch_{eventCp}$ bestimmt werden, d.h. ob eine Instanz aus $Ch_{eventCp}$ nur ausgeführt wird, wenn sich die Instanz von snd_{parCp} in einem Fehlerzustand befindet, und damit die Kante l_{syn} nicht aktiviert wurde.

		Ch_{rcv}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
rcv_{pred}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	→	→	⊕	⊕	⊕	⊕
	ABORTED	←	⊕	‖	⊕	⊕	⊕	⊕
	EXECUTING	←	→	→	→	→	→	→
	TERMINATED	←	⊕	‖	⊕	⊕	⊕	⊕
	FAULTED	←	⊕	→	⊕	⊕	⊕	⊕
	COMPLETED	←	→	→	→	→	→	→

Abbildung 5.26.: Zustandsprofil zwischen rcv_{pred} und Ch_{rcv}

(13) Beziehungen zwischen den Aktivitäten rcv_{pred} und Ch_{rcv}

In der Choreographie hat die Aktivität rcv_{pred} über das Pick *rcv* eine indirekte Kontrollflussbeziehung zu den Aktivitäten in Ch_{rcv}. Wird die Instanz von rcv_{pred} in den Zustand *dead* gesetzt, erreicht danach auch die Instanz von *rcv* den Zustand *dead* (Axiom *Sq2*) und damit auch die Instanzen der direkten und indirekten Kindaktivitäten von *rcv* (Axiom $St_{CH}2$). Die (vorzeitige) Terminierung einer Instanz von rcv_{pred} führt zur vorzeitigen Terminierung der Instanz von *rcv*, wobei keine Terminierungsreihenfolge zwischen beiden Instanzen vorgegeben ist (Axiom $St_{CH2CH}3$). Folglich existiert auch keine Terminierungsreihenfolge zwischen den Instanzen von Ch_{rcv} und der Instanz von rcv_{pred}, da die Instanzen der Kindaktivitäten terminiert werden, wenn die Instanz von *rcv* terminiert wird (Axiom $St_{CH}1$). Erreicht die Instanz von rcv_{pred} den Zustand *faulted*, wird danach die Instanz von *rcv* in *aborted* gesetzt (Axiom *Sq2*) und somit auch die Instanzen ihrer Kindaktivitäten (Axiom $St_{CH}1$). Den Zustand *executing* und dessen Nachfolgezustände erreichen die Instanzen der Kindaktivitäten von *rcv* nur, nachdem die Instanz von *rcv* (Axiom $St_{CH}3$) und damit die Instanz von rcv_{pred} den Zustand *completed* erreicht hat (Axiom *Sq1*). Die Instanzen von Ch_{rcv} können auch in *dead* gehen, nachdem rcv_{pred} *completed* erreicht hat, da nur die Instanz einer Aktivität aus Ch_{rcv} den Zustand *executing* erreichen kann. Das Zustandsprofil zwischen rcv_{pred} und den Kindaktivitäten Ch_{rcv} des Picks *rcv* ist in Abbildung 5.26 dargestellt.

In p_μ hat das Duplikat rcv_{predCp} eine indirekte Kontrollflussbeziehung mit den Duplikaten in Ch_{rcv} über die Fault-Handler des Scopes s_{rcv}, die von den jeweiligen Throw-Aktivitäten ausgelöst werden. Wie in der Choreographie sind auch die Aktivitäten Ch_{rcv} Kindaktivitäten einer strukturierten Aktivität, die die direkte Nachfolgeaktivität von rcv_{predCp} ist. Daher erreichen die Instanzen von Ch_{rcv} den Zustand *dead*, nachdem die Aktivität rcv_{predCp} *dead* erreicht hat. Zwischen den Instanzen von Ch_{rcv} und der Instanz von rcv_{predCp} ist ebenfalls keine Terminierungsreihenfolge vorgegeben. Ein Fehler in der Instanz von rcv_{predCp} führt dazu, dass danach die Instanzen von Ch_{rcv} vorzeitig terminiert werden. Erreicht die Instanz von rcv_{predCp} den Zustand *completed*, geht danach die Instanz des Scopes s_{rcv} in den Zustand *executing* (Axiom *Sq1*) und damit auch die Instanz des Flows f (Axiom $St_{CH}3$). Dies ermöglicht den Instanzen der Aktivitäten, die sich im Flow f befinden, ebenfalls den Zustand *executing* (Axiom $St_{CH}3$) zu erreichen. Die Instanzen der Wait-Aktivitäten werden ausgeführt, sobald die Instanz von f ausgeführt wird. Die Instanzen der Throw-Aktivitäten können in *executing* und damit in *faulted* gehen (Axiom Act_{Throw}), falls ihre eingehenden Kontrollflusskanten aktiviert werden. Folglich können, wie in der Choreographie die Instanzen von Ch_{rcv} in den Fault-Handlern, die die Fehler der Throw-Aktivitäten verarbeiten, weiterhin den Zustand *executing* erreichen, nachdem die Instanz von rcv_{predCp} den Zustand *completed* erreicht hat. Da nur ein Fault-Handler aktiviert werden kann, können die Instanzen von Ch_{rcv} ebenfalls den Zustand *dead* erreichen, nachdem rcv_{predCp} den Zustand *completed* erreicht hat. Das Zustandsprofil zwischen rcv_{predCp} und Ch_{rcv} ist in Abbildung 5.27 dargestellt und entspricht dem zwischen rcv_{pred} und Ch_{rcv}.

(14) Beziehungen zwischen den Aktivitäten Ch_{rcv} und rcv_{succ}

In der Choreographie hat die Aktivität rcv_{succ} als direkte Nachfolgeaktivität des Picks *rcv* eine indirekte Kontrollflussbeziehung zu den Aktivitäten in Ch_{rcv} und zwar über die Eltern-Kind-Beziehung zwischen *rcv* und Ch_{rcv} sowie der Kontrollflusskante zwischen *rcv* und rcv_{succ}. Wird die Instanz von *rcv* in *dead*

		Ch_{rcvCp}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
rcv_{predCp}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	→	→	⊕	⊕	⊕	⊕
	ABORTED	←	⊕	‖	⊕	⊕	⊕	⊕
	EXECUTING	←	→	→	→	→	→	→
	TERMINATED	←	⊕	‖	⊕	⊕	⊕	⊕
	FAULTED	←	⊕	→	⊕	⊕	⊕	⊕
	COMPLETED	←	→	→	→	→	→	→

Abbildung 5.27.: Zustandsprofil zwischen rcv_{predCp} und Ch_{rcvCp}

gesetzt, werden zuerst die Instanzen von Ch_{rcv} in *dead* gesetzt (Axiom $St_{CH}2$) und dann die Instanz von rcv_{succ} (Axiom *Sq2*). Nur eine Instanz der Aktivitäten in Ch_{rcv} kann den Zustand *executing* erreichen (Axiom $Pick_{Event}$). Folglich impliziert die (vorzeitige) Terminierung einer Instanz der Aktivitäten aus Ch_{rcv}, dass danach auch die Instanz von *rcv* terminiert wird (Axiom $St_{CH}5$), da keine andere Instanz der Aktivitäten aus Ch_{rcv} die Terminierung dadurch ausgelöst haben kann, dass sie in *faulted* gegangen ist. Da zwischen der Instanz von *rcv* und der von rcv_{succ} keine (vorzeitige) Terminierungsreihenfolge vorgegeben ist (Axiom $St_{CH2CH}3$), ist auch keine Terminierungsreihenfolge zwischen rcv_{succ} und den Instanzen von Ch_{rcv} vorgegeben. Löst eine Instanz von Ch_{rcv} einen Fehler aus, geht die Instanz von *rcv* ebenfalls in *faulted* (Axiom $St_{CH}4$) und die Instanz von rcv_{succ} wird danach terminiert (Axiom $St_{CH2CH}2$). Erreicht eine Instanz von Ch_{rcv} den Zustand *completed* und die anderen Instanzen *dead*, kann die Instanz von *rcv* in *completed* gehen (Axiom $St_{CH}6$) und die Instanz von rcv_{succ} kann danach in *executing* (Axiom *Sq1*) oder *aborted* wechseln. Das Zustandsprofil zwischen den Kindaktivitäten Ch_{rcv} und der Aktivität rcv_{succ} ist in Abbildung 5.28 dargestellt.

In p_{μ} ist die Aktivität rcv_{succCp} die direkte Nachfolgeaktivität des Scopes s_{rcv}, der die Duplikate in Ch_{rcvCp} als Fehlerbehandlungsaktivitäten enthält. Die Instanzen von Ch_{rcv} erreichen *dead* bevor s_{rcv} und damit auch bevor rcv_{succCp} *dead* erreicht (Axiom *Sq2*). Eine (vorzeitige) Terminierungsreihenfolge von Ch_{rcvCp} und rcv_{succCp} besteht hier ebenfalls nicht, da auch keine Terminie-

		rcv_{succ}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
Ch_{rcv}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	→	→	→	→	→	→
	ABORTED	←	⊕	‖	⊕	⊕	⊕	⊕
	EXECUTING	←	⊕	→	→	→	→	→
	TERMINATED	←	⊕	‖	⊕	⊕	⊕	⊕
	FAULTED	←	⊕	→	⊕	⊕	⊕	⊕
	COMPLETED	←	⊕	→	→	→	→	→

Abbildung 5.28.: Zustandsprofil zwischen Ch_{rcv} und rcv_{succ}

rungsreihenfolge zwischen der Instanz von s_{rcv} und der Instanz von rcv_{succCp} besteht (Axiom $St_{CH2CH}3$). Erreicht eine Instanz von Ch_{rcvCp} den Zustand *faulted*, geht auch die Instanz von s_{rcv} in den Zustand *faulted* (Axiom $St_{CH}4$) und die Instanz von rcv_{succCp} wird, wie in der Choreographie, danach terminiert (Axiom $St_{CH2CH}2$). Ebenfalls wie in der Choreographie, wird die Instanz von rcv_{succCp} in *executing* gesetzt, nachdem eine Instanz der Aktivitäten aus Ch_{rcvCp} *completed* und die anderen *dead* erreicht haben, da dann die Scope-Instanz *completed* erreicht (Axiom *Sq1* und Axiom $St_{CH}6$). Das resultierende Zustandsprofil zwischen Ch_{rcv} und rcv_{succCp} in Abbildung 5.27 entspricht dem zwischen rcv_{succ} und Ch_{rcv} in Abbildung 5.28.

		rcv_{succCp}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
Ch_{rcvCp}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	→	→	→	→	→	→
	ABORTED	←	⊕	‖	⊕	⊕	⊕	⊕
	EXECUTING	←	⊕	→	→	→	→	→
	TERMINATED	←	⊕	‖	⊕	⊕	⊕	⊕
	FAULTED	←	⊕	→	⊕	⊕	⊕	⊕
	COMPLETED	←	⊕	→	→	→	→	→

Abbildung 5.29.: Zustandsprofil zwischen Ch_{rcvCp} und rcv_{succCp}

(15) Beziehungen zwischen den Aktivitäten snd_{succ} und ch_{msg}

In der Choreographie bestehen zwischen snd_{succ} und ch_{msg} dieselben impliziten Zustandsbeziehungen über *snd*, wie bei Invoke-Receive-Interaktionen zwischen snd_{succ} und rcv_{succ} (siehe in Abbildung 5.16). Der Grund dafür ist e^{msg}, welches hier die Funktion der Receive-Aktivität übernimmt und nach dessen Aktivierung ch_{msg} erst ausgeführt werden kann. Ist also snd_{succ} im Zustand *dead*, muss auch *snd* im Zustand *dead* sein (Axiom *Sq2*). Es kann also keine Nachricht gesendet werden und die Instanz ch_{msg} kann damit nicht den Zustand *executing* erreichen. Zwischen den anderen Zuständen von snd_{succ} und ch_{msg} besteht keine Reihenfolge, da die Aktivitäten voneinander isoliert sind.

Die durch die Materialisierung erstellte Aktivität syn_{snd} und die Kontrollflusskante l_{syn} besitzen snd_{succCp} und ch_{msgCp} weiterhin eine indirekte Zustandsabhängigkeit. Das heißt, auch hier impliziert der Zustand *dead* von snd_{succCp}, dass syn_{snd} im Zustand *dead* ist und dass ch_{msgCp} *executing* somit nicht mehr erreichen kann. Alle sonstigen Zustände können die Aktivitäten snd_{succCp} und ch_{msgCp} durch die Isolation in Teilnehmer-Scopes und dadurch, dass die Aktivitäten keine Synchronisationsabhängigkeit zueinander haben, unabhängig voneinander erreichen.

(16) Beziehungen zwischen den Aktivitäten snd_{succ} und Ch_{event}

Die Aktivitäten snd_{succ} und Ch_{event} sind vollständig isoliert voneinander, da sie sich in unterschiedlichen Teilnehmern befinden und auch keine Kontrollflussbeziehung über die Nachrichtenkante zwischen ihnen besteht. Es gilt daher das Profil der vollständigen Isolation aus Abbildung 4.16 zwischen ihnen.

Diese vollständige Isolation wird auch für die Duplikate snd_{succCp} und $Ch_{eventCp}$ beibehalten, da sie sich in unterschiedlichen Teilnehmer-Scopes befinden und auch durch die bei der Materialisierung erstellte Kante l_{syn} keine Kontrollflussabhängigkeit zwischen ihnen besteht.

5.5.5. Illustrationsszenario nach der Materialisierung

Abbildung 5.30 zeigt das Prozessmodell p_μ, das aus der Flugbuchungschoreographie generiert wurde[1], nach der Kontrollflussmaterialisierung. Das Prozessmodell verdeutlicht eine Limitation des Materialisierungsansatzes, die schon bei der Diskussion der Zustandsbeziehungen in Abschnitt 5.4.1.3 abstrakt erläutert wurde. In der Choreographie sendet nur die günstigste Fluggesellschaft ein Ticket an den Kunden, die andere Gesellschaft verwirft ihr Angebot nach einer bestimmten Zeit. Dies wird durch die Invoke-Receive-Interaktion zwischen *Ticket versenden* und *Ticket empfangen* zwischen dem Teilnehmer $\mathfrak{p}_{Kunde}$ und den Teilnehmern $\mathfrak{p}_{LH}$ sowie $\mathfrak{p}_{Swiss}$ modelliert. Aus dieser Interaktion wurde in p_μ im Teilnehmer-Scope s_{Kunde} die Aktivität syn_{rcv} mit zwei eingehenden Kontrollflusskanten l_{synLH} und $l_{synSwiss}$ erstellt. Die Instanz von syn_{rcv} kann erst in den Zustand *executing* wechseln, wenn alle ihre eingehenden Kontrollflusskanten evaluiert wurden. Wird zur Laufzeit in p_μ der Nachrichtentransfer, dass Teilnehmer $\mathfrak{p}_{LH}$ das Ticket an $\mathfrak{p}_{Kunde}$ sendet, emuliert, wird Kante l_{synLH} aktiviert. Die Instanz von syn_{rcv} muss aber warten, bis die Kante $l_{synSwiss}$ deaktiviert wurde. Dies geschieht erst, wenn die Instanz des Waits w_{Swiss} das Timer-Ereignis auslöst, was dazu führt, dass die Ausführung der Instanz von $s_{rcvSwiss}$ beendet wird. Abhängig davon, welche Zeitspanne im Timer-Ereignis definiert ist, kann das die Ausführung der Instanz von syn_{rcv} und potentiellen Nachfolgeaktivitäten erheblich verzögern. In der Choreographie besteht diese Verzögerung nicht, da die Ausführung der Receive-Aktivität direkt nach dem Empfang einer Nachricht endet.

Das Problem kann umgegangen werden, indem die Aktivität *Ticket empfangen* als Pick- anstatt als Receive-Aktivität modelliert wird. Die Nachrichtenkante würde *Ticket versenden* mit einem entsprechenden Nachrichtenereignis verbinden. Das Pick würde dann, wie in Abschnitt Abschnitt 5.5 erläutert, materialisiert werden. Der das Pick ersetzende Scope s_{rcv} würde sofort beendet werden, nachdem eine der Kanten l_{synLH} oder $l_{synSwiss}$ aktiviert wurde.

[1] In der Realität würde vermutlich nur ein Teil der Choreographie konsolidiert werden. Es ist zum Beispiel vorstellbar, dass die Prozesse der Teilnehmer $\mathfrak{p}_{Reisebüro}$ und $\mathfrak{p}_{Swiss}$ konsolidiert werden würden, wenn ihre Organisationen fusionieren.

Durch die Materialisierung können temporär Kontrollflusskanten entstehen, die das Metamodell verletzen, da sich zum Beispiel ihre Quell- und Zielaktivität nicht in derselben Schleife befinden. Dies ist auch in dem konsolidierten Prozessmodell in Abbildung 5.30 der Fall, wo sich zum Beispiel die Aktivität syn_{snd} (*Preis anfordern*) im Teilnehmer-Scope $s_{Reisebüro}$ in einer ForEach-Schleife befindet und Quellaktivität für zwei durch die Materialisierung erstellte Kontrollflusskanten ist, deren Zielaktivitäten sich im Teilnehmer-Scope s_{LH} und s_{Swiss} außerhalb der Schleife befinden.

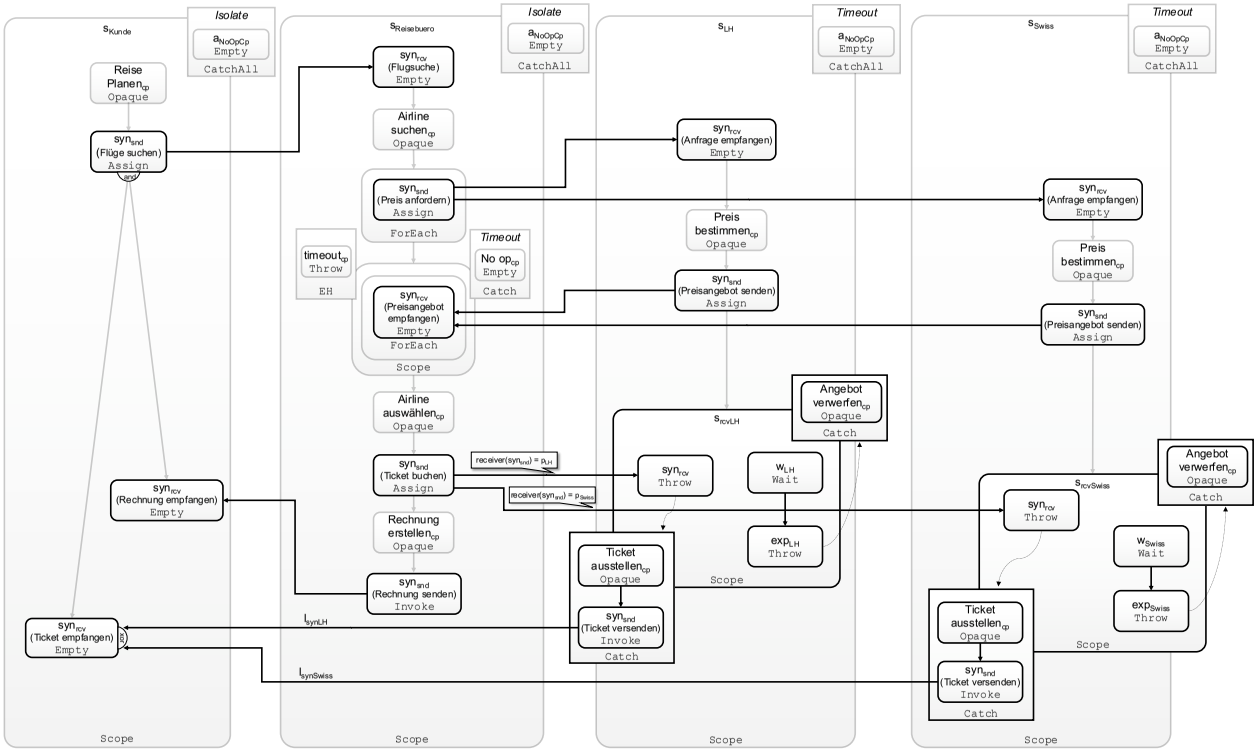

Abbildung 5.30.: Das aus $c_{Flugbuchung}$ generierte Prozessmodell p_μ nach der Materialisierung

5.6. Zusammenfassung

In diesem Kapitel wurden die Konsolidierungsoperation definiert und die Schritte zu deren Umsetzung skizziert. Im Detail wurde der erste Konsolidierungsschritt beschrieben, der das konsolidierte Prozessmodell p_μ erzeugt und dort für jeden potentiellen Teilnehmer der Choreographie einen Teilnehmer-Scope erstellt, der das Verhalten des jeweiligen Teilnehmers emuliert.

Daneben wurde der zweite Konsolidierungsschritt, die Kontrollflussmaterialisierung, detailliert diskutiert, mit dem die Interaktionen der Choreographie ohne Kommunikationsaktivitäten in p_μ emuliert werden. Dazu wurde erläutert, wie die Basisinteraktionen Invoke-Receive und Invoke-Nachrichtenereignis materialisiert werden, auf denen die komplexen Interaktionen, die in Kapitel 7 vorgestellt werden, aufbauen. Für beide Basisinteraktionen wurde untersucht, inwieweit das Verhalten der von der Materialisierung betroffenen Geschäftsaktivitäten in p_μ erhalten bleibt. Dabei wurde festgestellt, dass die Ausführungsreihenfolge der Zustandstransitionen der Geschäftsaktivitäten für beide Basisinteraktionen erhalten wird, es jedoch verglichen mit der Choreographie bei materialisierten Invoke-Receive-Interaktionen zu Verzögerungen in der Ausführung von Aktivitätsinstanzen geben kann.

Wie im vorherigen Abschnitt erläutert, erzeugt die Materialisierung ungültige Kontrollflusskanten, die das Metamodell verletzen, falls sich die Kommunikationsaktivitäten in Schleifen befinden. Im folgenden Kapitel wird daher im Rahmen des dritten Konsolidierungsschritts diskutiert, wie diese Kontrollflussverletzungen aufgelöst werden können, um auch multilaterale Interaktionen zu materialisieren, bei denen ein Teilnehmer über eine Kommunikationsaktivität in einer Schleife mit einer Menge von anderen Teilnehmern kommuniziert.

Kapitel 6

Konsolidierung von Interaktionen mit kommunizierenden Schleifen

Interaktionen, an denen eine zur Entwurfszeit unbekannte Menge an Teilnehmern partizipieren oder bei denen Teilnehmer eine unbekannte Anzahl an Nachrichten austauschen, werden mittels den in Abschnitt 3.2.4.10 definierten kommunizierenden Schleifen modelliert. Die Anzahl der Iterationen der Schleifen wird zur Laufzeit entweder vor oder dynamisch während deren Ausführung bestimmt. Das Reisebüro in der Choreographie aus Abbildung 5.2 ermittelt beispielsweise die Airlines, von denen es ein Preisangebot anfordert, abhängig von der Buchungsanfrage erst zur Laufzeit. An die infrage kommenden Airlines werden über eine Schleife sukzessive Nachrichten

gesendet, mit denen das Preisangebot angefordert wird. Dabei wird während jeder Iteration mit einer anderen Airline kommuniziert. Analog empfängt das Reisebüro über eine weitere Schleife die Preisangebote von den Airlines. Für diese Schleifen sind also die Teilnehmer, mit denen sie kommunizieren, vor deren Ausführung bekannt.

BPEL und somit auch das hier verwendete Metamodell definieren für strukturierte Aktivitäten bestimmte Restriktionen bezüglich der eingehenden und ausgehenden Kontrollflusskanten, die bereits in Abschnitt 3.2.3.3 diskutiert wurden. Für Schleifenrümpfe sind weder eingehende noch ausgehende Kontrollflusskanten erlaubt, da Kontrollflusskanten nur einmal ihren Zustand wechseln können. Fault-Handler hingegen dürfen nur ausgehende aber keine eingehenden Kontrollflusskanten besitzen.

Die Materialisierung der Basisinteraktionen erstellt allerdings Kontrollflusskanten zwischen den Synchronisationsaktivitäten der verschiedenen Teilnehmer von p_μ, ohne diese Restriktionen zu berücksichtigen. Dies kann zu Kontrollflussverletzungen und somit zu einem, bezogen auf das Metamodell, ungültigem Prozessmodell p_μ führen, da sich eine Synchronisationsaktivität innerhalb und die andere außerhalb der jeweiligen kommunizierenden Schleife befindet. In dem Beispielszenario wurden durch die Materialisierung zum Beispiel Kontrollflusskanten erzeugt, deren Zielaktivität im ForEach liegt (Abbildung 5.30). Kanten, die diese Einschränkungen verletzen, werden als *grenzverletzende Kontrollflusskanten* bezeichnet. Um diese Kontrollflussverletzungen zu vermeiden, könnte das Metamodell Kommunikationsaktivitäten in Schleifen verbieten. Dies würde es aber unmöglich machen, einen großen Teil der in Kapitel 7 vorgestellten komplexen Interaktionsmuster zu konsolidieren.

Im Verlauf dieses Kapitels werden daher basierend auf der Arbeit von Wagner, Kopp und Leymann [WKL15] sowie der von Kukillaya [Kuk17] zwei Ansätze beschrieben, diese Kontrollflussverletzungen für kommunizierende Schleifen aufzulösen. Dazu werden in Abschnitt 6.1 verschiedene mit ForEach- bzw. While-Schleifen modellierbare Kommunikationsszenarien diskutiert und die

Kontrollflussverletzungen, die durch deren Materialisierung entstehen können. Kann für diese Kommunikationsszenarien die Anzahl der Iterationen der involvierten Schleifen bestimmt werden, können die Verletzungen durch das in Abschnitt 6.3 formal beschriebene *Ausrollen von Schleifen* aufgelöst werden. Ist dies nicht möglich, kann als zweiter Ansatz die *Fusion von Schleifen* zur Auflösung der Verletzungen genutzt werden. Dieser Ansatz wird in Abschnitt 6.6 nur informell vorgestellt, da damit bestimmte Fehlerszenarien, auf die ebenfalls in dem Abschnitt eingegangen wird, nicht abgebildet werden können.

Das Auflösen der Verletzungen für Fault-, Termination-, Compensation und Event-Handler wird in diesem Kapitel aus Platzgründen und da sie nicht für die Konsolidierung von komplexen Interaktionsmustern benötigt werden, ausgeklammert. Wagner, Kopp und Leymann [WKL13] beschreiben aber, wie grenzverletzende Kontrollflusskanten bei Fault-Handlern aufgelöst werden können, indem die Fehlerbehandlungsaktivität und ihre Nachfahren aus dem Fault-Handler transferiert werden. Darauf aufbauend wird im Rahmen von studentischen Arbeiten ein ähnliches Vorgehen von Berger [Ber13] für Termination-Handler, von Phadnis [Pha15] für Compensation-Handler und von Milutinovic [Mil14] für Event-Handler beschrieben.

6.1. Modellierungsszenarien für kommunizierende Schleifen

Das Metamodell der kommunizierenden Schleifen (Abschnitt 3.2.4.10) und der Nachrichtenkanten (Abschnitt 3.1.2) erlaubt in Choreographien die Modellierung der folgenden Szenarien, in denen eine Invoke-Aktivität *snd* über eine Nachrichtenkante *ml* mit einem Empfangskonstrukt co_{rcv} verbunden ist und sich *snd* und/oder co_{rcv} in einer Schleife befinden:

(I) Die Sendeaktivität befindet sich in einer ForEach-Schleife, während sich das Empfangskonstrukt in keiner Schleife befindet und umgekehrt:

$$(\textsc{Ancestors}(snd) \cap A_{\texttt{forEach}} \neq \emptyset \wedge \textsc{Ancestors}(co_{rcv}) \cap A_{\texttt{loop}} = \emptyset)$$
$$\vee(\textsc{Ancestors}(snd) \cap A_{\texttt{loop}} = \emptyset \wedge \textsc{Ancestors}(co_{rcv}) \cap A_{\texttt{forEach}} \neq \emptyset)$$

(II) Die Sendeaktivität und das Empfangskonstrukt befinden sich in zwei verschiedenen ForEach-Schleifen:

$$(\textsc{Ancestors}(snd) \cap A_{\texttt{forEach}} \neq \emptyset \wedge \textsc{Ancestors}(co_{rcv}) \cap A_{\texttt{forEach}} \neq \emptyset)$$
$$\wedge(\textsc{Ancestors}(snd) \cap \textsc{Ancestors}(co_{rcv}) \cap A_{\texttt{forEach}} = \emptyset)$$

(III) Die Sendeaktivität befindet sich in einer While-Schleife während sich das Empfangskonstrukt nicht in einer Schleife befindet und umgekehrt:

$$(\textsc{Ancestors}(snd) \cap A_{\texttt{while}} \neq \emptyset \wedge \textsc{Ancestors}(co_{rcv}) \cap A_{\texttt{loop}} = \emptyset)$$
$$\vee(\textsc{Ancestors}(snd) \cap A_{\texttt{loop}} = \emptyset \wedge \textsc{Ancestors}(co_{rcv}) \cap A_{\texttt{while}} \neq \emptyset)$$

(IV) Die Sendeaktivität und das Empfangskonstrukt befinden sich in zwei verschiedenen While-Schleifen:

$$(\textsc{Ancestors}(snd) \cap A_{\texttt{while}} \neq \emptyset \wedge \textsc{Ancestors}(co_{rcv}) \cap A_{\texttt{while}} \neq \emptyset)$$
$$\wedge(\textsc{Ancestors}(snd) \cap \textsc{Ancestors}(co_{rcv}) \cap A_{\texttt{while}} = \emptyset)$$

(V) Die Sendeaktivität befindet sich in einer ForEach-Schleife, während sich das Empfangskonstrukt in einer While-Schleife befindet und umgekehrt:

$$(\textsc{Ancestors}(snd) \cap A_{\texttt{forEach}} \neq \emptyset \wedge \textsc{Ancestors}(co_{rcv}) \cap A_{\texttt{while}} \neq \emptyset)$$
$$\vee(\textsc{Ancestors}(snd) \cap A_{\texttt{while}} \neq \emptyset \wedge \textsc{Ancestors}(co_{rcv}) \cap A_{\texttt{forEach}} \neq \emptyset)$$

Die Modellierungsszenarien und die Kontrollflussverletzungen, die aus deren Materialisierung resultieren, werden im Folgenden anhand von Beispielen näher erläutert.

Durch die Modellierung des Szenarios I kann ein Teilnehmer über die ForEach-Schleife mit einer beliebigen Menge von Teilnehmern kommunizieren. Dabei kommuniziert der Teilnehmer während jeder Iteration des ForEachs mit einem anderen Teilnehmer. In Abbildung 6.1 kann Teilnehmer $\mathfrak{p}_{rcv}$ über das ForEach zur Laufzeit zum Beispiel eine Nachricht von Teilnehmer $\mathfrak{p}_{snd1}$ und eine von Teilnehmer $\mathfrak{p}_{snd2}$ empfangen. Alternativ kann $\mathfrak{p}_{rcv}$ auch keine oder nur eine Nachricht, entweder von $\mathfrak{p}_{snd1}$ oder von $\mathfrak{p}_{snd2}$, empfangen. Die Materialisierung erzeugt grenzverletzende Kontrollflusskanten, bei

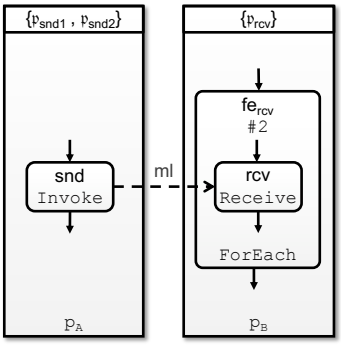

Abbildung 6.1.: Szenario I – Ein ForEach interagiert mit zwei Teilnehmern deren Kommunikationsaktivität sich nicht in einer Schleife befindet.

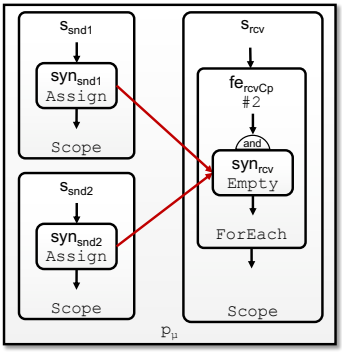

Abbildung 6.2.: Szenario I nach der Materialisierung

denen sich eine Aktivität innerhalb und die andere außerhalb einer Schleife befindet, wie zum Beispiel in Abbildung 6.2 dargestellt, die die Interaktion aus Abbildung 6.1 nach dem Materialisierungsschritt zeigt.

Mittels des Szenarios II kann die Interaktion einer Menge von Teilnehmern mit einer anderen Menge von Teilnehmern modelliert werden, wobei jeder Teilnehmer über ein ForEach mit den anderen Teilnehmern kommuniziert.

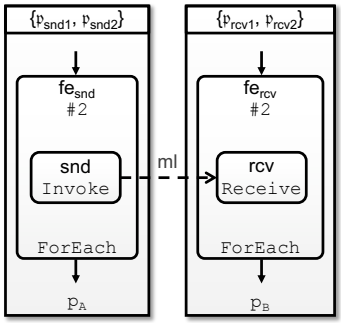

Abbildung 6.3.: Szenario II – ForEach interagiert mit ForEach

Dies ist beispielhaft in Abbildung 6.3 dargestellt, indem die Teilnehmer $\mathfrak{p}_{snd1}$ und $\mathfrak{p}_{snd2}$ eine Nachricht an die Teilnehmer $\mathfrak{p}_{rcv1}$ und $\mathfrak{p}_{rcv2}$ senden können. Auch hier ist es möglich, dass $\mathfrak{p}_{snd1}$ bzw. $\mathfrak{p}_{snd2}$ nur eine Nachricht an $\mathfrak{p}_{rcv1}$ bzw. $\mathfrak{p}_{rcv2}$ sendet. Dabei muss zur Laufzeit sichergestellt werden (wie auch bei den anderen Szenarien), dass die Empfangsaktivitäten nicht auf eine Nachricht warten, weil die Schleife, über die gesendet wird, weniger Iterationen ausführt als die empfangende Schleife. Im Beispiel in Abbildung 6.3 wäre dies der Fall, wenn zur Laufzeit die Schleife fe_{snd} von Teilnehmer $\mathfrak{p}_{rcv1}$ nur einmal und die Schleife fe_{rcv} von $\mathfrak{p}_{rcv1}$ zweimal iteriert. Während der letzten Iteration würde das Receive *rcv* in diesem Fall auf eine Nachricht warten, die nie gesendet wird.

Die Materialisierung dieses Szenarios erzeugt, wie im Beispiel in Abbildung 6.4, grenzverletzende Kontrollflusskanten, deren Quell- und Zielaktivitäten sich jeweils in einer anderen ForEach-Schleife befinden.

Das Szenario III ist eine alternative Möglichkeit eine Interaktionen zu modellieren, bei denen ein Teilnehmer mit einer Menge von Teilnehmern interagiert. Der Unterschied zum Szenario I besteht darin, dass die Teilnehmer, mit denen interagiert wird, während und nicht vor der Ausführung der Schleife bestimmt werden können bzw. die Schleife nach jeder Iteration abgebrochen werden kann. In Abbildung 6.5 geschieht dies über die Aktivität

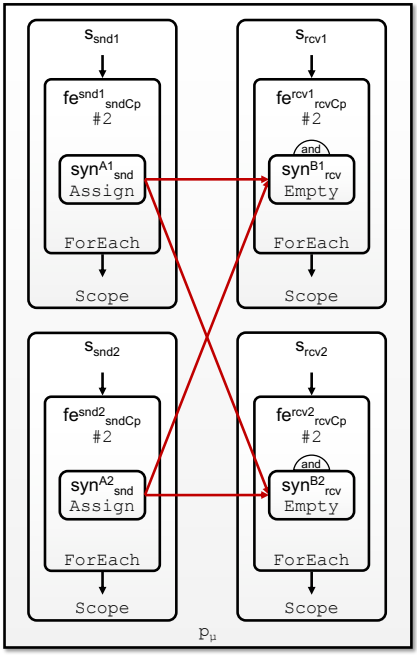

Abbildung 6.4.: Szenario II nach der Materialisierung

rcv_{pred}, die während jeder Iteration den prüft, ob und von welchem Teilnehmer das Receive *rcv* Nachrichten empfangen soll. Anhand welcher Kriterien rcv_{pred} dies entscheidet, wird hier offen gelassen. Analog zum Szenario I entstehen bei der Materialisierung von Szenario III ebenfalls grenzverletzende Kontrollflusskanten, bei denen sich eine Synchronisationsaktivität innerhalb und die andere außerhalb des Whiles befindet.

Das Szenario IV, die Interaktion von zwei While-Schleifen, unterscheidet sich zur Interaktion von zwei ForEach-Schleifen (Szenario II) dadurch, dass die kommunizierenden While-Schleifen nicht zwangsläufig während jeder Iteration mit einem anderen Teilnehmer interagieren müssen und dass auch hier nach jeder Iteration ein Abbruch möglich ist. Im Beispiel in Abbildung 6.6 kann Teilnehmer $\mathfrak{p}_{snd}$ so sukzessive Nachrichten an $\mathfrak{p}_{rcv}$ senden, wenn die

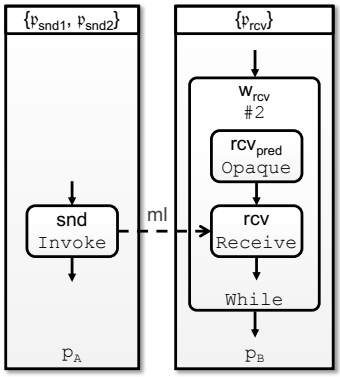

Abbildung 6.5.: Szenario III – Ein While interagiert mit zwei Teilnehmern deren Sendeaktivität sich nicht in einer Schleife befindet.

Aktivität snd_{pred} im Schleifenkörper von w_{snd} während mehrerer Iterationen die Teilnehmerreferenz von $\mathfrak{p}_{rcv}$ im Invoke *snd* setzt. Analog dazu muss die Aktivität rcv_{succ} in w_{rcv} prüfen, ob weiterhin Nachrichten von $\mathfrak{p}_{snd}$ empfangen werden müssen. Alternativ wäre es wie im Szenario II möglich, dass zum Beispiel zwei Sender $\mathfrak{p}_{snd1}$ und $\mathfrak{p}_{snd2}$ mit zwei Empfängern $\mathfrak{p}_{rcv1}$ und $\mathfrak{p}_{rcv2}$ über While-Schleifen interagieren. In beiden Fällen befinden sich die Quell- und Zielaktivitäten der grenzverletzenden Kontrollflusskanten, die durch die Materialisierung erstellt wurden, in unterschiedlichen While-Schleifen[1].

Das Szenario V, das in Abbildung 6.7 beispielhaft dargestellt ist, bietet eine weitere Modellierungsmöglichkeit um Interaktionen zwischen zwei Mengen von Teilnehmern zu realisieren. Im Unterschied zum Szenario II können die Teilnehmer, die die Interaktion über eine While-Schleife modellieren, diese während jeder Iteration abbrechen. Es ist allerdings nicht wie im Szenario IV möglich, dass zur Laufzeit eine Schleife in mehreren Iterationen mit demselben Teilnehmer interagiert, da das ForEach während jeder Iteration mit einem anderen Teilnehmer kommunizieren muss. Die Materialisierung

[1]Die grafische Darstellung der Materialisierung von Szenario IV findet sich in Abbildung 6.19 in Abschnitt 6.6, in dem die Fusion von While-Schleifen erläutert wird.

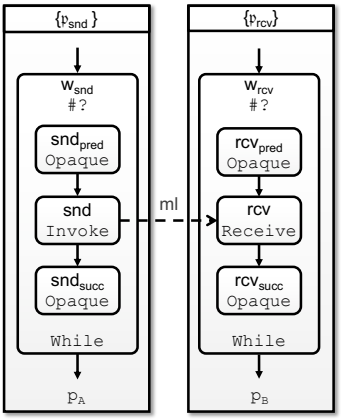

Abbildung 6.6.: Szenario IV – While interagiert mit While

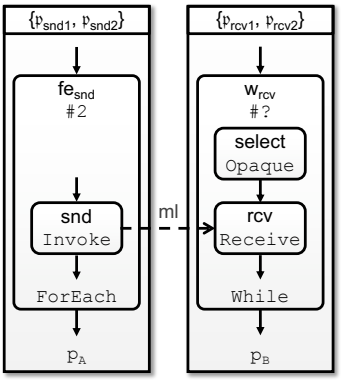

Abbildung 6.7.: Szenario V – ForEach interagiert mit While

dieses Szenarios erzeugt, wie in Abbildung 6.8 dargestellt, grenzverletzende Kontrollflusskanten, bei denen sich eine Aktivität der jeweiligen Kante in einer ForEach- und die andere in einer While-Schleife befindet.

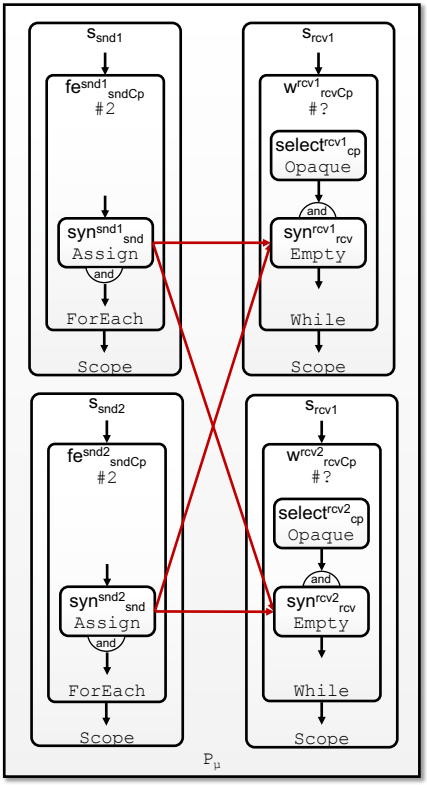

Abbildung 6.8.: Szenario V nach der Materialisierung

6.2. Bestimmen der maximalen Iterationen einer Schleife

Die durch die Materialisierung der Szenarien I, II,III und V erzeugten grenzverletzenden Kontrollflusskanten können mit dem in Abschnitt 6.3 erläuterten Ansatz zum Ausrollen von Schleifen aufgelöst werden. Für das materialisierte Szenario IV können die grenzverletzenden Kontrollflusskanten nur durch die in Abschnitt 6.6 beschriebene Fusion der Schleifen eliminiert werden.

Beim Ausrollen einer Schleife wird für jede mögliche Iteration ein Duplikat

des Schleifenkörpers in p_μ erstellt (siehe Abschnitt 6.3). Um dies zu ermöglichen, muss die zur Laufzeit maximal mögliche Anzahl an Iterationen der Schleife ermittelt werden. In diesem Abschnitt wird daher erläutert, wie die Iterationsanzahl für kommunizierende Schleifen anhand der Informationen in der Choregraphie bestimmt werden kann. Die Grundidee wird im Folgenden skizziert und im weiteren Verlauf formal beschrieben.

Ein ForEach interagiert mit einer Menge von Teilnehmern über eine Kommunikationsaktivität bzw. ein Empfangskonstrukt in seinem Schleifenkörper. Die Teilnehmer, mit denen das ForEach interagieren darf, wird durch die Nachrichtenkante vorgegeben, der die Aktivität bzw. das Konstrukt zugeordnet ist. Sendet die Schleife eine Nachricht über ein Invoke im Schleifenkörper, können über die Nachrichtenkante, in der das Invoke als Sendeaktivität fungiert, die potentiellen Empfänger bestimmt werden (siehe Abschnitt 3.1.2). Analog dazu können mittels der Nachrichtenkante, der ein Empfangskonstrukt zugeordnet ist, die potentiellen Sender bestimmt werden. Die Nachrichtenkante *ml* in Abbildung 6.1 hat zum Beispiel die zwei potentiellen Sender $\mathfrak{p}_{snd1}$ und $\mathfrak{p}_{snd2}$. Folglich kann das ForEach fe_{rcv} maximal zweimal iterieren. Der Ansatz funktioniert, weil das Metamodell verlangt, dass alle potentiellen Teilnehmer einer Choreographie bekannt sein müssen (siehe Abschnitt 3.1).

Die maximale Anzahl von Iterationen von While-Schleifen kann über die Informationen in der Choreographie nur bestimmt werden, wenn diese über eine Nachrichtenkante mit einer Kommunikationsaktivität verbunden ist, die sich innerhalb eines ForEachs (Szenario V) oder wenn sich die Kommunikationsaktivität außerhalb einer Schleife befindet (Szenario III). In diesen Szenarien kann die While-Schleife ebenfalls maximal so häufig iteriert werden, wie es potentielle Teilnehmer gibt, mit denen sie interagieren kann[1].

Im Szenario IV, der Interaktion von zwei While-Schleifen, kann die maximale

[1] Es wird davon ausgegangen, dass eine sendende While-Schleife keine Nachricht ins „Leere“ sendet, d.h., dass potentiell immer ein Empfänger auf die vom While gesendete Nachricht wartet. Die Annahme schließt nicht aus, dass der Empfänger die Nachricht aufgrund eines Fehlers nicht mehr empfangen kann.

Anzahl an Iterationen daher nicht mit dem oben beschriebenen Verfahren bestimmt werden, da in diesem Szenario ein While auch mehrmals mit demselben Teilnehmer interagieren kann. Die Anzahl der Iterationen einer While-Schleife kann auch mit anderen Techniken anhand des Kontroll- und Datenflusses in den Prozessmodellen ermittelt werden. So wurden zum Beispiel von Heinze, Amme und Moser [HAM12] sowie von Monakova et al. [MKL+09] Ansätze beschrieben, um Zusicherungen betreffs der Ausführung von bestimmten Pfaden im Kontrollfluss zu ermitteln. Diese Techniken können die Iterationen auch nicht in allen Fällen bestimmen und sind deshalb nicht Gegenstand dieser Arbeit. Es wird daher davon ausgegangen, dass für Whiles, die mit Whiles interagieren, die maximale Iterationsanzahl nicht bestimmt werden kann.

Formal wird der eben beschriebene Ansatz durch die Funktion GetMaxIter in Algorithmus 6.1 umgesetzt. Die Schleife wird im Parameter a_{loop} übergeben. Die Funktion ermittelt zuerst die Schleifen $A^{peer}_{\texttt{loop}}$, mit denen a_{loop} interagiert, also über Nachrichtenkanten ihrer Kommunikationsaktivitäten im Schleifenkörper verbunden ist. Diese Menge ist leer, falls a_{loop} nur mit Teilnehmern interagiert, deren Kommunikationsaktivitäten sich nicht in einer Schleife befinden (Szenarios I und III). In Zeile 4 wird überprüft, ob es sich bei a_{loop} um eine While-Schleife handelt, die mit einer anderen While-Schleife interagiert. Ist dies der Fall, gibt die Funktion $\perp$ zurück, da sich die maximale Iterationsanzahl nicht bestimmen und sich somit die Schleife nicht ausrollen lässt. Andernfalls prüft die Funktion, mit wie vielen Teilnehmern a_{loop} interagieren kann.

Die Ermittlung der potentiellen Teilnehmer ist durch die Funktion GetInteractingParticipants in Algorithmus 6.2 implementiert. Die Funktion ermittelt alle Nachrichtenkanten ml_{lpSnd} mit einer Sendeaktivität und alle Nachrichtenkanten ml_{lpRcv} mit einem Empfangskonstrukt, das sich im Schleifenkörper von a_{loop} befindet. Aus diesen Nachrichtenkanten werden dann die potentiellen empfangenden Teilnehmer $\mathfrak{p}^{set}_{rcv}$ und sendenden Teilnehmer $\mathfrak{p}^{set}_{snd}$ bestimmt. Diese werden dann von der Funktion zurückgegeben. Da in dem Metamodell die Einschränkung besteht, dass eine Schleife entweder Nachrichten senden

Algorithmus 6.1 Bestimmen der maximalen Iterationsanzahl einer Schleife

1: **function** GETMAXITER(a_{loop})
2: $A_{desc} = \text{DESCENDANTS}(a_{loop})$
$A^{peer}_{\texttt{loop}} = \{a^{peer}_{loop} \in A_{\texttt{loop}} \mid \exists ml \in ML :$
3: $(\pi_2(ml) \in A_{desc} \wedge \pi_4(ml) \in \text{DESCENDANTS}(a^{peer}_{loop}))$
$\vee(\pi_2(ml) \in \text{DESCENDANTS}(a^{peer}_{loop}) \wedge \pi_4(ml) \in A_{desc})\}$
4: **if** $a_{loop} \in A_{\texttt{while}} \wedge (|A^{peer}_{\texttt{loop}} \cap A_{\texttt{while}}| \neq 0)$ **then**
5: **return** $\perp$
6: **end if**
7: $\mathfrak{P}_p = \text{GETINTERACTINGPARTICIPANTS}(a_{loop})$
8: **return** $|\mathfrak{P}_p|$
9: **end function**

oder empfangen kann aber nicht beides, ist eine dieser Mengen für a_{loop} immer leer.

Algorithmus 6.2 Ermitteln der potentiellen Teilnehmer, mit denen eine Schleife interagieren kann.

1: **function** GETINTERACTINGPARTICIPANTS(a_{loop})
2: $ML_{lpSnd} = \{ml \in ML \mid a_{loop} \in \text{ANCESTORS}(\pi_2(ml))\}$
3: $ML_{lpRcv} = \{ml \in ML \mid a_{loop} \in \text{ANCESTORS}(\pi_4(ml))\}$
4: $\mathfrak{p}^{set}_{rcv} = \bigcup_{ML_{lpSnd} \in ml_{lpSnd}} \pi_3(ml_{lpSnd})$
5: $\mathfrak{p}^{set}_{snd} = \bigcup_{ML_{lpRcv} \in ml_{lpRcv}} \pi_1(ml_{lpRcv})$
6: **return** $\mathfrak{p}^{set}_{snd} \cup \mathfrak{p}^{set}_{rcv}$
7: **end function**

6.3. Ausrollen von Schleifen

Das Ausrollen von Schleifen ist ursprünglich eine Technik aus dem Compilerbau, mit der die Ausführungszeit von Programmen optimiert werden kann [KM93; QCS02; LZSS04]. In dieser Arbeit wird diese Technik verwendet, um die grenzverletzenden Kontrollflusskanten in p_μ zu eliminieren, die bei der Materialisierung der Szenarien I, II, III und V entstehen. Dazu wird der Körper der Schleife für jeden potentiellen Teilnehmer, mit dem die

Schleife interagieren kann, in p_μ eingefügt. Die ausgerollten Schleifenkörper in p_μ werden als *Teilnehmeriterationen* bezeichnet und jede Teilnehmeriteration emuliert die Kommunikation mit einem bestimmten Teilnehmer. Beim Ausrollen einer Schleife in Teilnehmeriterationen müssen die folgenden Eigenschaften erfüllt werden:

i. Um die Kommunikation mit jedem Teilnehmer zu emulieren, mit dem auch die Schleife in der Choreographie hätte kommunizieren können, muss jede potentielle Teilnehmeriteration ausgerollt werden.

ii. Die ausgerollte Teilnehmeriteration darf nur ausgeführt werden, wenn sich der Teilnehmer zur Laufzeit tatsächlich in der Menge der Teilnehmer befindet, mit denen die Schleife kommunizieren soll. Diese wird der Schleife über die Abbildung $\mathsf{partSet} : A_{\texttt{forEach}} \rightarrow \mathfrak{P}^{set}$ zugewiesen.

iii. Die Teilnehmeriterationen müssen im Kontrollfluss sequentiell verknüpft werden, um weiterhin die Hintereinanderausführung der Iterationen wie in der originalen Schleife zu gewährleisten[1].

iv. Die Teilnehmeriterationen müssen das Verhalten des Schleifenkörpers der originalen Schleife emulieren. Das schließt mit ein, dass jede Teilnehmeriteration die Kommunikation mit einem anderen Teilnehmer emuliert.

6.3.1. Erstellung von Teilnehmeriterationen in p_μ

Die Erstellung einer ausgerollten Schleife mittels Teilnehmeriterationen wird erst informell erläutert, bevor im weiteren Verlauf dieses Abschnitts ein entsprechender Algorithmus vorgestellt wird.

Das Duplikat einer ForEach- oder While-Schleife a_{loopCp} die mit einer Menge Teilnehmern $\mathfrak{P}_{loop}$ interagieren kann, wird in p_μ durch ein Flow *ul* ersetzt, das die ausgerollte Schleife repräsentiert. Da der Flow die Schleife a_{loopCp}

[1] Das Ausrollen von Schleifen, deren Körper parallel ausgeführt werden können, wird in dieser Arbeit aus Platzgründen nicht diskutiert.

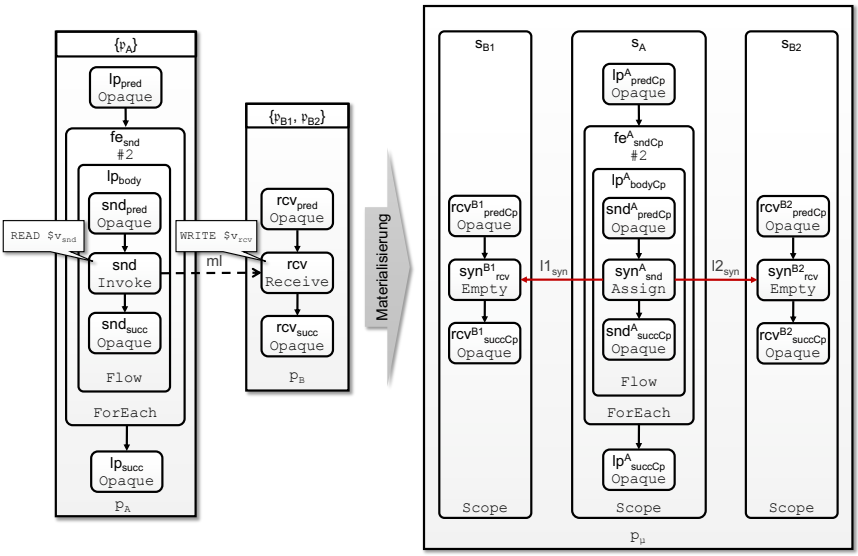

Abbildung 6.9.: Sendende ForEach-Schleife mit durch die Materialisierung erzeugten grenzverletzenden Kontrollflusskanten

im Kontrollfluss ersetzt, übernimmt er deren eingehende und ausgehende Kontrollflusskanten sowie deren Eintrittsbedingung.

In dem Flow wird für jeden potentiellen Teilnehmer aus $\mathfrak{p}_{loop} \in \mathfrak{P}_{loop}$ ein Duplikat des Schleifenkörpers, d.h. der direkten und indirekten Kindaktivitäten von a_{loopCp} erstellt. Jedes Duplikat repräsentiert also eine Teilnehmeriteration des potentiellen Teilnehmers (Eigenschaft (i)).

Abbildung 6.9 illustriert ein weiteres Beispiel für ein sendendes ForEach fe_{snd}, für dessen Duplikat fe^A_{sndCp} in p_μ durch die Materialisierung grenzverletzende Kontrollflusskanten erzeugt wurden. Durch das Ausrollen von fe^A_{sndCp} wird die ausgerollte Schleife ul^A erzeugt, die in Abbildung 6.10 dargestellt ist. Da fe_{snd} mit zwei Teilnehmern kommunizieren kann, enthält ul^A zwei Duplikate des Schleifenkörpers von fe^A_{sndCp}.

While-Schleifen können beliebige Iteratoren bzw. Schleifenbedingungen

besitzen, d.h. sie iterieren nicht zwangsläufig über eine Teilnehmermenge. Aber sie müssen in allen Interaktionsszenarien außer Szenario IV während jeder Iteration mit einem anderen Teilnehmer kommunizieren. Daher können sie auch in Teilnehmeriterationen ausgerollt werden.

Die Kontrollflusskanten zwischen den Duplikaten der Synchronisationsaktivitäten in der ausgerollten Schleife und denen in den Teilnehmer-Scopes werden beim Ausrollen nicht direkt wiederhergestellt. Dies geschieht in einem späteren Schritt, der in Abschnitt 6.3.3 erläutert wird.

In einer Schleife dürfen die Teilnehmeriterationen nur unter bestimmten Bedingungen ausgeführt werden (Eigenschaft (ii)). Bei ForEach-Schleifen darf eine Teilnehmeriteration zur Laufzeit nur ausgeführt werden, wenn sich der Teilnehmer in der Teilnehmermenge $\mathfrak{p}_{loop}^{set}$ befindet, die dem ForEach über die Abbildung partSet zugewiesen wurde ($\mathfrak{p}_{loop}^{set} = \mathsf{partSet}(a_{loopCp})$). Die Teilnehmeriterationen einer While-Schleife dürfen wiederum nur ausgeführt werden, wenn die der Schleife über die Abbildung $\mathsf{cond}_{\texttt{while}}$ zugewiesene Schleifenbedingung wahr ist. Die bedingte Ausführung der Duplikate des Schleifenkörpers wird dadurch erreicht, dass dieser von einer exklusiven Verzweigungsaktivität mit zwei ausgehenden Kontrollflusskanten l_{nxt} und l_{skip} synchronisiert wird. Handelt es sich bei a_{loopCp} um eine ForEach-Schleife, prüft die Transitionsbedingung von l_{nxt}, ob sich der Teilnehmer, dessen ausgerollte Teilnehmeriteration ausgeführt werden soll, tatsächlich in der Menge $\mathfrak{p}_{loop}^{set}$ befindet. Handelt es sich bei a_{loopCp} um eine While-Schleife, ist l_{nxt} als Transitionsbedingung die Schleifenbedingung des Whiles zugewiesen. Evaluiert die Transitionsbedingung von l_{nxt} zu `true`, wird die Kante aktiviert und die ausgerollte Teilnehmeriteration kann ausgeführt werden. Andernfalls wird die Kante l_{skip} aktiviert, die Teilnehmeriteration wird übersprungen und es wird geprüft, ob die nächste Teilnehmeriteration ausgeführt werden darf. Befindet sich zum Beispiel zur Laufzeit nur der Teilnehmer $\mathfrak{p}_{snd2}$ in der Menge $\mathfrak{p}_{loop}^{set}$, würde so die Ausführung der ausgerollten Teilnehmeriteration von $\mathfrak{p}_{snd1}$ übersprungen werden.

Mittels des hier verwendeten Metamodells kann für ForEach- und While-

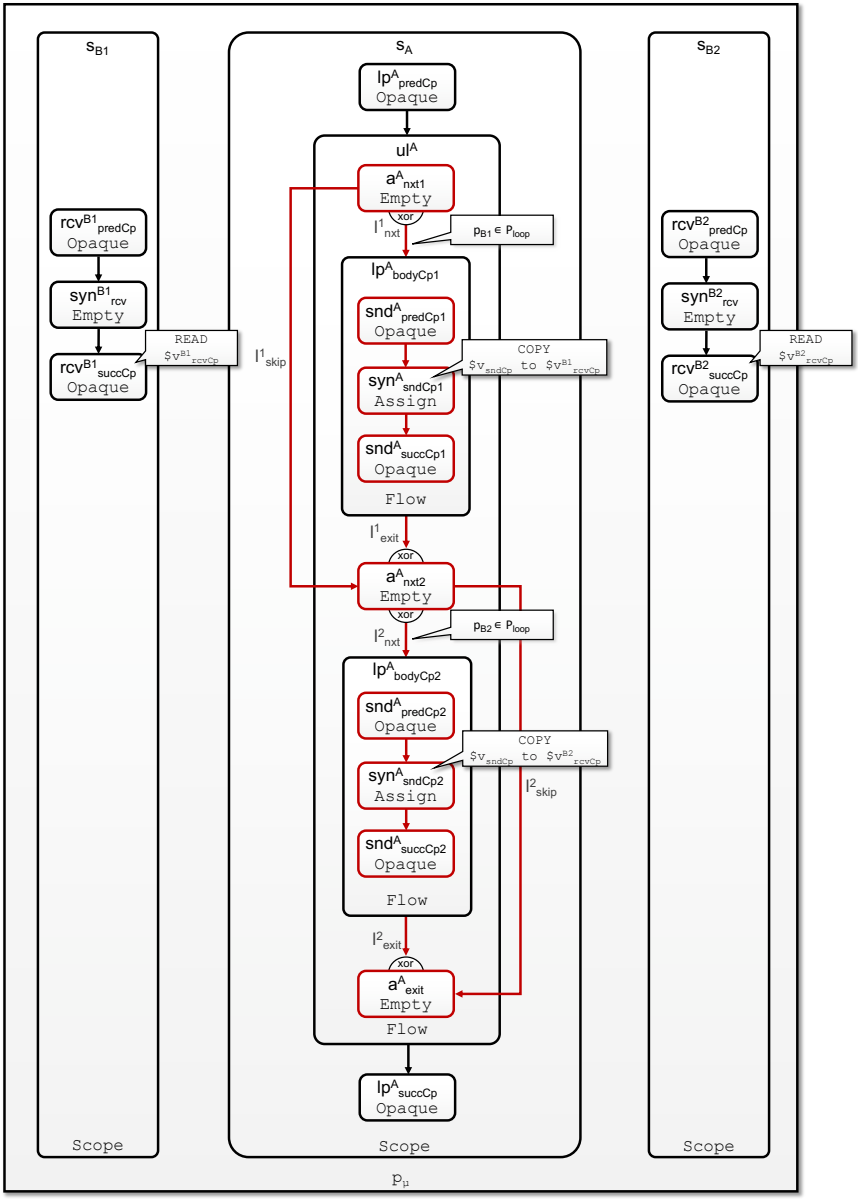

Abbildung 6.10.: Ausgerollte ForEach-Schleife aus Abbildung 6.9

Schleifen die Reihenfolge, in denen mit den Teilnehmern interagiert wird, nicht spezifiziert werden. Erst während der „executable completion“ wird, wie von Reimann et al. [RKDL08] diskutiert, festgelegt, wie diese Teilnehmer aufgerufen werden und somit auch in welcher Reihenfolge. Daher ist auch keine Reihenfolge vorgegeben, in der die Teilnehmeriterationen ausgerollt werden müssen. Soll eine Reihenfolge definiert werden, muss nach der Konsolidierung p_μ entsprechend modifiziert werden, zum Beispiel durch die Modifikation der Transitionsbedingungen von l_{nxt} und l_{skip}.

Im Detail wird das Ausrollen von sequentiellen Schleifen von der Prozedur UnrollSeqLoop in Algorithmus 6.3 beschrieben. Der Prozedur wird die auszurollende Schleife a_{loopCp}, der Teilnehmer $\mathfrak{p}_{loop}$ in dessen Teilnehmer-Scope sich a_{loopCp} befindet und die Menge der potentiellen Teilnehmer $\mathfrak{P}_{partner}$ übergeben, mit denen a_{loopCp} interagieren kann. In Zeile 2 wird der Flow ul erstellt, das die Schleife a_{loopCp} im Kontrollfluss ersetzt (Zeile 35).

In Zeile 3 wird die Verzweigungsaktivität a_{nxt} erstellt, die mit ihren ausgehenden Kontrollflusskanten l_{nxt} und l_{skip} die oben beschriebene bedingte Ausführung der ersten Teilnehmeriteration sicherstellt. Die Transitionsbedingungen werden l_{nxt} und l_{skip} für die erste und die folgenden Teilnehmeriterationen in den Zeilen 12 bis 18 zugewiesen. Die in Zeile 5 erstellte Aktivität a_{exit} dient als Vereinigungsaktivität für die Kanten, die die bedingte Ausführung der letzten ausgerollten Teilnehmeriteration ermöglichen.

Das Ausrollen des Schleifenkörpers wird sukzessive für jeden potentiellen Teilnehmer in den Zeilen 7 bis 34 durchgeführt. Die Funktion DuplicateLoopBody (siehe Algorithmus A.13 im Anhang A.3) dupliziert dabei den Schleifenkörper. Das heißt, sie erstellt ein Duplikat der direkten und indirekten Kindaktivitäten der Schleife sowie den Kontrollflussbeziehungen zwischen diesen Aktivitäten. Die Kontrollflusskanten, deren Quell- oder Zielaktivitäten außerhalb des Schleifenkörpers liegen, werden nicht dupliziert. Die Funktion gibt das Duplikat $a_{\mathfrak{p}}$ der direkten Kindaktivität der Schleife zurück, das die ausgerollte Teilnehmeriteration repräsentiert. Beim Ausrollen der Schleife in Abbildung 6.9 würde die Funktion zum Beispiel den Flow

$lp^A_{bodyCp1}$ bzw. den Flow $lp^A_{bodyCp2}$ zurück geben.

Für jede Teilnehmeriteration muss der Nachrichtenfluss zwischen ihr und dem Teilnehmer $\mathfrak{p}_{partner}$, mit dem sie interagiert, emuliert werden. Dies wird durch den Aufruf der Prozedur CreateDataflowFromPartIteration realisiert, die in Abschnitt 6.3.2 erläutert wird.

Nachdem die Schleife ausgerollt wurde, wird sie in Zeile 35 durch den Aufruf der Prozedur ReplaceActivity (siehe Anhang A.2) aus p_μ entfernt.

6.3.2. Emulation des Datenflusses zwischen ausgerollten Teilnehmeriterationen und den Teilnehmer-Scopes

Sendende Schleifen übertragen während jeder Iteration Daten mittels Nachrichten an einen anderen Empfänger. Dieser Datenfluss muss auch in den ausgerollten Teilnehmeriterationen abgebildet werden. Dazu muss, wie für die Materialisierung beschrieben, die Synchronisationsaktivität, die in der Teilnehmeriteration das Senden emuliert, das Teilnehmerduplikat der Eingabevariablen des Invokes in das Duplikat der Ausgabevariable des Receives bzw. des Nachrichtenereignisses kopieren, die dem Teilnehmer-Scope des Empfängers zugeordnet ist. In Abbildung 6.10 kopiert daher die Synchronisationsaktivität syn^A_{sndCp1} in der Teilnehmeriteration, die das Senden an Teilnehmer $\mathfrak{p}_{B1}$ emuliert, den Inhalt der Variable v_{sndCp} (Teilnehmerduplikat der Variable v_{snd} des Invokes) in die Variable v^{B1}_{rcvCp} (Teilnehmerduplikat der Variable v_{rcv} des Receives).

Für empfangende Schleifen sind keine Anpassungen nach dem Ausrollen notwendig, da der Datenfluss durch die Synchronisationsaktivität im Teilnehmer-Scopes des Senders emuliert wird. Diese Aktivitäten werden bei der Materialisierung der Basisinteraktionen in den sendenden Teilnehmer-Scopes so erstellt, dass sie die Daten in das Duplikat der Ausgabevariable des Receives kopieren, das sich in der empfangenden Schleife befindet. Auf dieses global deklarierte Duplikat können alle Teilnehmeriterationen zugreifen.

Algorithmus 6.3 Ausrollen einer sequentiellen Schleife

1: **procedure** UNROLLSEQLOOP($a_{loopCp}, \mathfrak{p}_{loop}, \mathfrak{P}_{partner}$)
2: $ul \leftarrow$ CREATEACTIVITY(`flow`)
3: $a_{nxt} \leftarrow$ CREATEACTIVITY(`empty`)
4: CREATEHR($ul, \bot, a_{nxt}$)
5: $a_{exit} \leftarrow$ CREATEACTIVITY(`empty`)
6: CREATEHR($ul, \bot, a_{exit}$)
7: **for all** $\mathfrak{p}_{partner} \in \mathfrak{P}_{partner}$ **do**
8: $\mathfrak{P}_{partner} \leftarrow \mathfrak{P}_{partner} \setminus \{\mathfrak{p}_{partner}\}$
9: $a_{\mathfrak{p}} \leftarrow$ DUPLICATELOOPBODY($a_{loopCp}, \mathfrak{p}_{loop}, \mathfrak{p}_{partner}$)
10: $\mathfrak{c}_{nxt} = new$ `Condition`
11: $\mathfrak{c}_{skip} = new$ `Condition`
12: **if** $a_{loopCp} \in A_{\texttt{while}}$ **then**
13: $ex_{nxt} \leftarrow \mathsf{expr}_{\mathcal{B}}(\mathsf{cond}_{\texttt{while}}(a_{loopCp}))$
14: **else**
15: $ex_{nxt} \leftarrow \mathfrak{p}_{partner} \in \mathfrak{p}^{set}_{partner}$ mit $\mathfrak{p}^{set}_{partner} = \mathsf{partSet}(a_{loopCp})$
16: **end if**
17: $\mathsf{expr}_{\mathcal{B}}(\mathfrak{c}_{nxt}) \leftarrow ex_{nxt}$
18: $\mathsf{expr}_{\mathcal{B}}(\mathfrak{c}_{skip}) \leftarrow \neg\, ex_{nxt}$
19: $l_{nxt} \leftarrow$ CREATECONTROLLINK($a_{nxt}, a_{\mathfrak{p}}, \mathfrak{c}_{nxt}$)
20: **if** $|\mathfrak{P}_{partner}| > 0$ **then**
21: $a_{nxtPrev} \leftarrow a_{nxt}$
22: $a_{nxt} \leftarrow$ CREATEACTIVITY(`empty`)
23: CREATEHR($ul, \bot, a_{nxt}$)
24: **if** $a_{loopCp} \in A_{\texttt{while}}$ **then**
25: $l_{skip} \leftarrow$ CREATECONTROLLINK($a_{nxtPrev}, a_{exit}, \mathfrak{c}_{skip}$)
26: **else**
27: $l_{skip} \leftarrow$ CREATECONTROLLINK($a_{nxtPrev}, a_{nxt}, \mathfrak{c}_{skip}$)
28: **end if**
29: $l_{exit} \leftarrow$ CREATECONTROLLINK($a_{\mathfrak{p}}, a_{nxt}, \bot$)
30: **else**
31: $l_{exit} \leftarrow$ CREATECONTROLLINK($a_{\mathfrak{p}}, a_{exit}, \bot$)
32: **end if**
33: CREATEDATAFLOWFROMPARTITERATION($a_{loopCp}, \mathfrak{p}_{loop}, \mathfrak{p}_{partner}$)
34: **end for**
35: REPLACEACTIVITY(a_{loopCp}, ul)
36: **end procedure**

Die Prozedur CreateDataflowFromPartIteration in Algorithmus 6.4 beschreibt, wie der Datenfluss zwischen der Teilnehmeriteration und dem Teilnehmer-Scope erstellt wird, mit dem die Teilnehmeriteration interagiert. Der Prozedur wird dazu die auszurollende Schleife a_{loopCp}, der Teilnehmer $\mathfrak{p}_{loop}$, in dessen Teilnehmer-Scope sich a_{loopCp} befindet und der Teilnehmer $\mathfrak{p}_{partner}$ mit dessen Teilnehmer-Scope die Teilnehmeriteration die Kommunikation in p_μ emulieren soll, übergeben. Beim Ausrollen der Iteration der Schleife fe^A_{sndCp}, die mit $\mathfrak{p}_{B1}$ interagiert, würde also zum Beispiel die Prozedur mit den Werten fe^A_{sndCp}, $\mathfrak{p}_A$ und $\mathfrak{p}_{B1}$ aufgerufen werden.

Die Menge SYN_{snd} enthält alle Synchronisationsaktivitäten in a_{loopCp}, die das Senden einer Nachricht emulieren. In dem Beispiel würde die Menge nur aus der Aktivität syn^A_{snd} bestehen, da sich in dem Schleifenkörper von fe^A_{sndCp} nur eine Synchronisationsaktivität befindet.

Für jede dieser Aktivitäten wird ihr Teilnehmerduplikat syn_{sndCp} in der ausgerollten Iteration über die Abbildung actCopy ermittelt. Das Teilnehmerduplikat syn_{sndCp} wird in den Zeilen 6 und 7 so angepasst, dass es die Daten des Duplikats der Eingabevariable des Invokes *snd* im Teilnehmer-Scope der Schleife in das Duplikat der Ausgabevariable des Empfangskonstrukts co_{rcv} im Teilnehmer-Scope von $\mathfrak{p}_{partner}$ kopiert. Die Eingabevariable des Invokes wurde bereits im Materialisierungsschritt der Aktivität syn_{snd} zugewiesen und kann daher direkt auf syn_{sndCp} als Quellvariable übertragen werden. Das Duplikat der Ausgabevariable im Teilnehmer-Scope von $\mathfrak{p}_{partner}$ wird über das Empfangskonstrukt co_{rcv} bestimmt. Das Konstrukt co_{rcv} wird wiederum aus der Nachrichtenkante ermittelt, die von der Sendeaktivität *snd* während der Materialisierung der Basisinteraktion syn_{snd} erstellt wurde.

Algorithmus 6.4 Herstellen des Datenflusses zwischen Teilnehmeriteration und Teilnehmer-Scopes

1: **procedure** CREATEDATAFLOWFROMPARTITERATION($a_{loopCp}, \mathfrak{p}_{loop}, \mathfrak{p}_{partner}$)
2: $SYN_{snd} = \{syn_{snd} \in (A_{\texttt{assign}} \cap \text{DESCENDANTS}(a_{loopCp}))$
$\quad \mid \exists snd \in A_{\texttt{invoke}} : \text{com2Syn}(syn_{snd}) = snd)\}$
3: **for all** $syn_{snd} \in SYN_{snd}$ **do**
4: $syn_{sndCp} \leftarrow \text{actCopy}(\mathfrak{p}_{loop}, \mathfrak{p}_{partner}, syn_{snd})$
5: $co_{rcv} \leftarrow \pi_4(ml)$ mit $ml \in ML \wedge \pi_2(ml) = \text{com2Syn}(syn_{snd})$
6: $\text{assign}_{src}(syn_{sndCp}) \leftarrow \text{assign}_{src}(syn_{snd})$
7: $\text{assign}_{trg}(syn_{sndCp}) \leftarrow \text{varCopy}(\mathfrak{p}_{loop}, \text{outputVar}(co_{rcv}))$
8: **end for**
9: **end procedure**

6.3.3. Verbinden der Synchronisationsaktivitäten im ausgerollten Schleifenkörper mit Synchronisationsaktivitäten in den Teilnehmer-Scopes

Im vorherigen Abschnitt wurde beschrieben, wie der durch den Nachrichtenfluss implizierte Datenfluss zwischen den Duplikaten der Synchronisationsaktivitäten in den Teilnehmeriterationen und denen in den Teilnehmer-Scopes emuliert wird. Der durch den Nachrichtenfluss implizierte Kontrollfluss wird zwischen diesen Aktivitäten aufgrund der Nichtberücksichtigung der grenzverletzenden Kontrollflusskanten beim Ausrollen nicht abgebildet. In der ausgerollten Schleife ul^A in Abbildung 6.10 existieren zum Beispiel keine Kontrollflusskanten zwischen den Aktivitäten syn^A_{sndCp1} und syn^{B1}_{rcv} bzw. den Aktivitäten syn^A_{sndCp1} und syn^{B2}_{rcv}. Die ausgerollte Schleife kann daher in den Teilnehmeriterationen nicht die Kommunikation mit den jeweiligen Teilnehmern emulieren, Eigenschaft (iv) ist also verletzt.

Um diese Eigenschaft zu erfüllen, müssen die Kontrollflusskanten zwischen den Synchronisationsaktivitäten der Teilnehmer und den Teilnehmeriterationen anlog zur Materialisierung wiederhergestellt werden. In dem Beispiel muss also jeweils eine Kontrollflusskante zwischen den Aktivitäten syn^A_{sndCp1} und syn^{B1}_{rcv} sowie eine zwischen syn^A_{sndCp2} und syn^{B2}_{rcv} erstellt werden.

Algorithmus 6.5 Verbinden von Synchronisationsaktivitäten eines ausgerollten Schleifenkörpers mit denen in den Teilnehmer-Scopes

procedure ConnectUnrolledSynchActs(a_{loopCp}, $\mathfrak{P}_{partner}$)
  $L_{in} = \{l \in L \mid \pi_1(l) \notin \text{Descendants}(a_{loopCp})$
    $\wedge\, \pi_2(l) \in \text{Descendants}(a_{loopCp})\}$
  $L_{out} = \{l \in L \mid \pi_1(l) \in \text{Descendants}(a_{loopCp})$
    $\wedge\, \pi_2(l) \notin \text{Descendants}(a_{loopCp})\}$
  for all $\mathfrak{p}_{partner} \in \mathfrak{P}_{partner}$ **do**
    for all $l_{in} \in L_{in}$ **do**
      $syn_{rcv} \leftarrow \pi_1(l_{in})$
      $syn_{rcvCp} \leftarrow \text{actCopy}(\mathfrak{p}, \mathfrak{p}_{partner}, syn_{rcv})$
      $syn_{snd} \leftarrow \pi_2(l_{in})$
      $syn_{sndCp} \leftarrow \text{actCopy}(\mathfrak{p}_{partner}, \mathfrak{p}, syn_{snd})$
      if $syn_{sndCp} \neq \perp$ **then**
        $l_{syn} \leftarrow \text{CreateControlLink}(syn_{sndCp}, syn_{rcvCp}, \perp)$
      else
        $l_{syn} \leftarrow \text{CreateControlLink}(syn_{snd}, syn_{rcvCp}, \perp)$
      end if
      $\text{joinCond}(syn_{rcvCp}) \leftarrow \text{joinCond}(syn_{rcvCp}) \wedge \mathtt{1}_{syn}$
    end for
    for all $l_{out} \in L_{out}$ **do**
      $syn_{snd} \leftarrow \pi_1(l_{out})$
      $syn_{sndCp} \leftarrow \text{actCopy}(\mathfrak{p}, \mathfrak{p}_{partner}, syn_{snd})$
      $syn_{rcv} \leftarrow \pi_2(l_{out})$
      $syn_{rcvCp} \leftarrow \text{actCopy}(\mathfrak{p}_{partner}, \mathfrak{p}, syn_{rcv})$
      $l_{syn} \leftarrow \text{CreateControlLink}(syn_{sndCp}, syn_{rcvCp}, \perp)$
      if $syn_{rcvCp} \neq \perp$ **then**
        $l_{syn} \leftarrow \text{CreateControlLink}(syn_{sndCp}, syn_{rcvCp}, \perp)$
        $\text{joinCond}(syn_{rcvCp}) \leftarrow \text{joinCond}(syn_{rcvCp}) \wedge \mathtt{1}_{syn}$
      else
        $l_{syn} \leftarrow \text{CreateControlLink}(syn_{snd}, syn_{rcv}, \perp)$
        $\text{joinCond}(syn_{rcv}) \leftarrow \text{joinCond}(syn_{rcv}) \wedge \mathtt{1}_{syn}$
      end if
    end for
  end for
end procedure

Das Verbinden der Synchronisationsaktivitäten einer ausgerollten Schleife mit den entsprechenden Synchronisationsaktivitäten in den Teilnehmer-Scopes ist durch die Prozedur ConnectUnrolledSynchActs in Algorithmus 6.5 formal beschrieben. Der Prozedur werden als Parameter die ausgerollte Schleife a_{loopCp} und die potentiellen Teilnehmer $\mathfrak{P}_{partner}$, mit denen die Schleife interagiert, übergeben. Für die Schleife fe_{snd} bzw. fe_{sndCp} in Abbildung 6.9 gilt zum Beispiel $\mathfrak{P}_{partner} = \{\mathfrak{p}_{snd1}, \mathfrak{p}_{snd2}\}$.

Die Menge L_{in} umfasst alle bei der Materialisierung erstellten grenzverletzenden Kanten von empfangenden Schleifen, deren Quellaktivitäten sich außerhalb und deren Zielaktivitäten sich innerhalb von a_{loopCp} befinden. Die Menge L_{out} enthält wiederum alle grenzverletzenden Kanten von sendenden Schleifen, deren Zielaktivitäten sich außerhalb und deren Quellaktivitäten sich innerhalb von a_{loopCp} befinden. Für die Schleife fe_{snd} gilt also $L_{out} = \{l1_{syn}, l2_{syn}\}$ und $L_{in} = \{\}$.

Für jede ausgerollte Teilnehmeriteration werden die Duplikate der während der Materialisierung erstellten Synchronisationsaktivitäten in der ausgerollten Schleife mit den Synchronisationsaktivitäten außerhalb dieser Schleife verbunden. Letztere befinden sich daher im Teilnehmer-Scope, mit dem die Iteration die Interaktion emuliert. Dazu wird in der ersten inneren For-Schleife (Zeilen 5–16) für jedes Duplikat syn_{rcvCp} von syn_{rcv} in einer Teilnehmeriteration, das beim Ausrollen von a_{loopCp} erstellt wurde, mit der Quellaktivität syn_{snd} oder dem Duplikat der zugehörigen Quellaktivität syn_{sndCp} verbunden. Die Aktivität syn_{rcvCp} wird mit dem Duplikat syn_{sndCp} von syn_{snd} verbunden, falls sich syn_{snd} ebenfalls in einer Schleife befunden hat, die ausgerollt wurde und somit durch syn_{sndCp} ersetzt wurde[1]. Die Aktivität syn_{rcvCp} fungiert in beiden Fällen als parallele Vereinigungsaktivität.

In den Zeilen 17 bis 30 werden analog dazu die Duplikate der Synchronisationsaktivitäten einer ausgerollten sendenden Schleife mit den Dupli-

[1] Daher darf die Prozedur ConnectUnrolledSynchActs auch immer erst nach dem Ausrollen aller Schleifen in p_μ ausgeführt werden, da die Synchronisationsaktivitäten in ausgerollten Schleifen entfernt und durch ihre Duplikate in den Teilnehmeriterationen ersetzt werden.

katen der Synchronisationsaktivitäten außerhalb der Schleife verbunden. Für die Aktivität syn_{snd} wurden in Abbildung 6.10 beim Ausrollen die Duplikate syn^{A}_{sndCp1} (Teilnehmeriteration $\mathfrak{p}_{B1}$) und syn^{A}_{sndCp2} (Teilnehmeriteration $\mathfrak{p}_{B2}$) erstellt. Es gilt also in Zeile 19 $\mathsf{actCopy}(\mathfrak{p}_A, \mathfrak{p}_{B1}, syn^{A}_{snd}) = syn^{A}_{sndCp1}$ bzw. $\mathsf{actCopy}(\mathfrak{p}_A, \mathfrak{p}_{B2}, syn^{A}_{snd}) = syn^{A}_{sndCp2}$. Die Synchronisationsaktivitäten innerhalb der Empfänger-Scopes befinden sich nicht in einer Schleife, daher gibt $\mathsf{actCopy}(\mathfrak{p}_{B1}, \mathfrak{p}_A, syn^{B1}_{rcv})$ und $\mathsf{actCopy}(\mathfrak{p}_{B2}, \mathfrak{p}_A, syn^{B2}_{rcv})$ hier $\perp$ zurück. Die Kontrollflusskanten werden deshalb, wie in Abbildung 6.10 dargestellt, zwischen den Aktivitäten syn^{A}_{sndCp1} und syn^{B1}_{rcv} bzw. syn^{A}_{sndCp2} und syn^{B2}_{rcv} erstellt.

Die vorgestellten Algorithmen wurden bisher an dem Szenario I veranschaulicht. Abbildung 6.12 zeigt die ausgerollten interagierenden ForEach-Schleifen aus Abbildung 6.4 (Szenario II), nachdem die Prozedur ConnectUnrolledSynchActs deren Synchronisationsaktivitäten mit den Kontrollflusskanten $l_{syn1} - l_{syn4}$ verknüpft hat. Da sich alle Synchronisationsaktivitäten in einer ausgerollten Schleife befinden, wurden hier für jede Aktivität ein Duplikate erstellt und die Prozedur ConnectUnrolledSynchActs wird für jede ausgerollte Schleife aufgerufen. Daher würde zum Beispiel die Kontrollflusskante l_{syn1} in Abbildung 6.12 zweimal erstellt werden. Das erste Mal, wenn die Prozedur für die Schleife fe^{A1}_{sndCp} und das zweite Mal, wenn die Prozedur für die Schleife fe^{B1}_{rcvCp} aufgerufen wird. Zwei Kontrollflusskanten repräsentieren allerdings per Definition dasselbe Element aus der Menge L (siehe Abschnitt 3.2.3.2), wenn sie die gleichen Aktivitäten verbinden und dieselbe Transitionsbedingung besitzen.

In den bisherigen Beispielen wurde nur das Verbinden von Synchronisationsaktivitäten ausgerollter Schleifen gezeigt, die durch die Materialisierung von Invoke- und Receive-Interaktionen erstellt wurden. Ein Beispiel für das Verbinden von Synchronisationsaktivitäten die von Invoke-Aktivitäten und Nachrichtenereignissen erstellt wurden, findet sich in Abschnitt 7.3.2, wo die Konsolidierung des Interaktionsmusters *„Contingent Request“* beschrieben wird. Die Prozedur UnrollSeqLoop erstellt in diesem Fall, wie in Abbildung 7.12 illustriert, für jede Teilnehmeriteration ein Duplikat der Throw-Aktivität, die das Auslösen des Nachrichtenereignisses emuliert. Throws sind

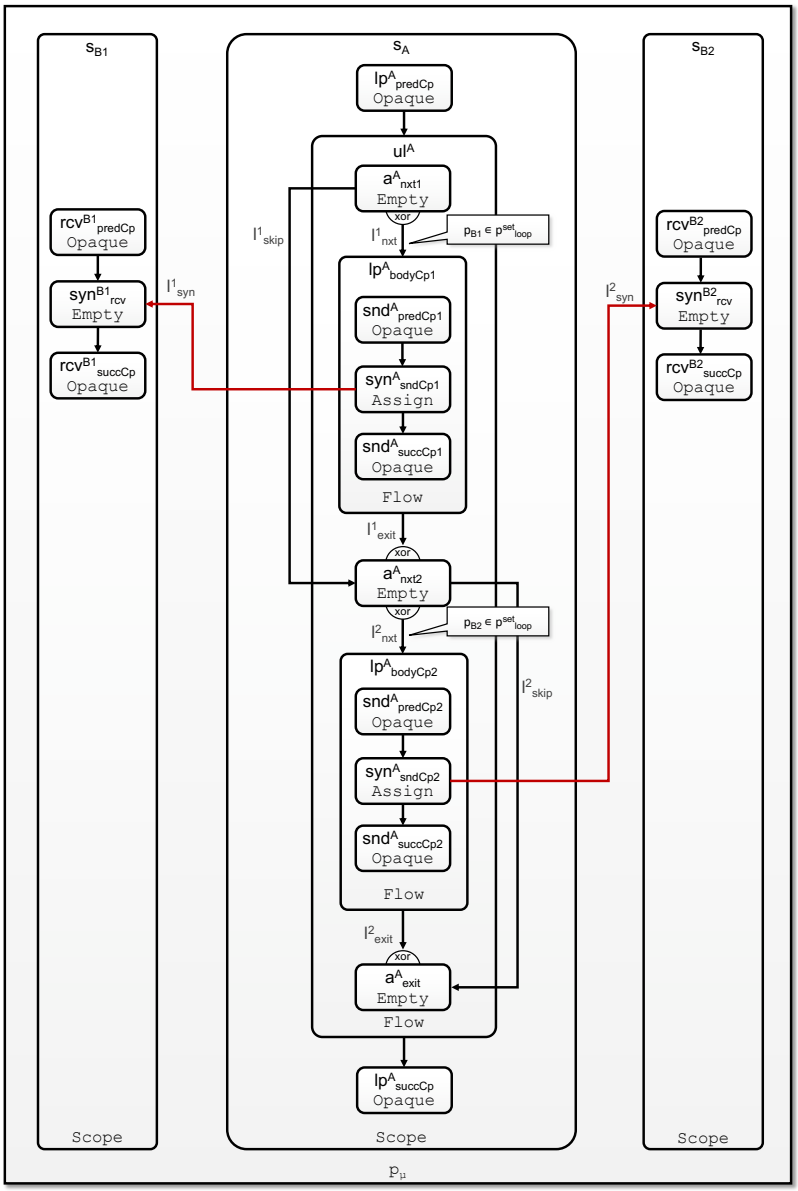

Abbildung 6.11.: Ausgerollte ForEach-Schleife aus Abbildung 6.10, deren Synchronisationsaktivitäten mit denen in den Empfänger-Scopes über Kontrollflusskanten verbunden sind

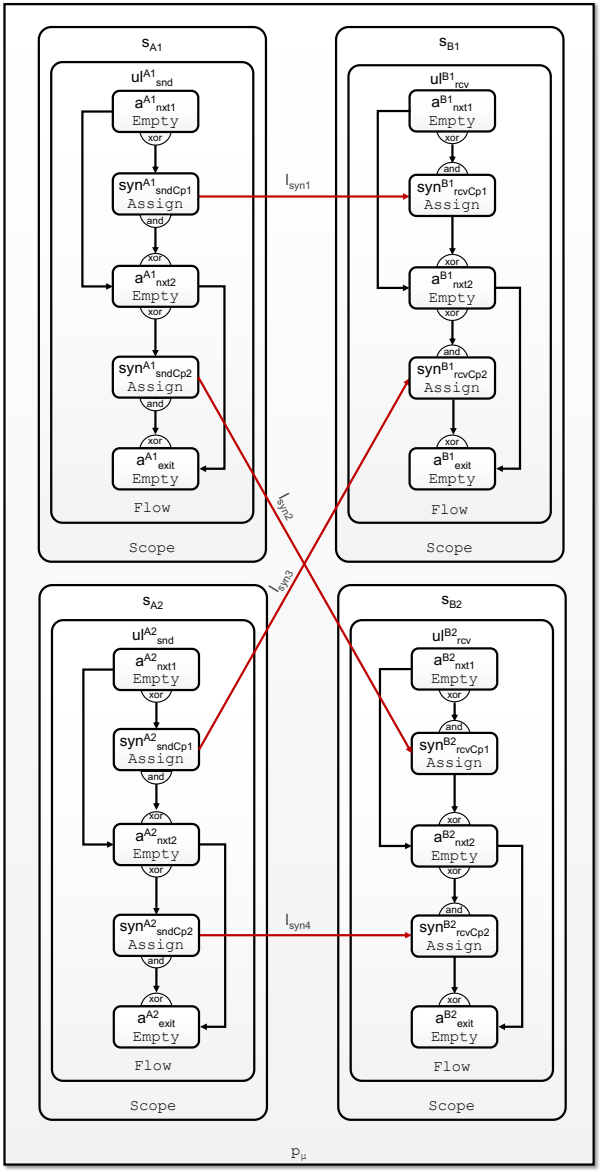

Abbildung 6.12.: Ausgerollte interagierende ForEach-Schleifen aus Abbildung 6.4 mit verbundenen Synchronisationsaktivitäten

ebenfalls Synchronisationsaktivitäten und werden daher auch von der Prozedur ConnectUnrolledSynchActs mit den zugehörigen Assigns verbunden, die das Senden der Nachricht emulieren.

6.4. Algorithmus zum Auflösen der grenzverletzenden Kontrollflusskanten von Schleifen im konsolidierten Prozessmodell

Die Prozedur ResolveControlFlowViolations in Algorithmus 6.6 führt die in den vorherigen Abschnitten erläuterten Schritte, d.h. das Ausrollen der Schleifen und das Verbinden ihrer Synchronisationsaktivitäten, für all jene Schleifen in p_μ aus, die grenzverletzenden Kontrollflusskanten besitzen. Ihr werden dazu die durch die Materialisierung erstellten Kontrollflusskanten L_{syn} übergeben (siehe Abschnitt 5.2). In Zeile 2 werden dann alle Schleifen mit eingehenden bzw. ausgehenden Kontrollflusskanten aus L_{syn} ermittelt und zur Menge $A^{vio}_{\texttt{loop}}$ hinzugefügt. Für jede Schleife a_{lpCp} in der Menge $A^{vio}_{\texttt{loop}}$ wird überprüft, ob ihre maximale Iterationsanzahl bestimmbar ist. Dazu wird in Zeile 5 die originale Schleife a_{lpOrg} ermittelt, aus der das Duplikat a_{lpCp} erstellt wurde (die Funktion GetMaxIter benötigt die originale Schleife, um die Iterationsanzahl anhand der Nachrichtenkanten zu ermitteln). Die Schleife a_{lpCp} wird nur ausgerollt, wenn die Iterationsanzahl von GetMaxIter bestimmt werden kann[1]. Um die Schleife auszurollen, werden in Zeile 8 die Teilnehmer ermittelt, mit denen die Schleife interagiert. Die Abbildung $\text{loop}_{\mathfrak{P}} : A_{\texttt{loop}} \rightarrow \mathfrak{P}^{set}$ dient dazu, diese Teilnehmer a_{lpCp} zuzuordnen, um diese im nächsten Schritt, bei dem die Synchronisationsaktivitäten innerhalb der Schleifen mit denen außerhalb der Schleifen verbunden werden, nicht erneut bestimmen zu müssen. Dieser Schritt ist in den Zeilen 14–17 umgesetzt.

[1] Da die Fusion von Schleifen in Abschnitt 6.6 informell erläutert wird, wird hier dazu kein Algorithmus aufgerufen.

Algorithmus 6.6 Auflösen von Kontrollflussverletzungen in Schleifen

procedure ResolveControlFlowViolations(L_{syn})
  $A^{vio}_{\texttt{loop}} = \{a_{loop} \in A_{\texttt{loop}} \mid \exists l_{syn} \in L_{syn} :$
    $a_{loop} \in (\text{Ancestors}(\pi_1(l_{syn})) \cup \text{Ancestors}(\pi_2(l_{syn})))\}$
  $A^{unrollCp}_{\texttt{loop}} = \{\}$
  for all $a_{lpCp} \in A^{vio}_{\texttt{loop}}$ **do**
    $a_{lpOrg} \in A_{\texttt{loop}}$ mit $\exists \mathfrak{p} \in \mathfrak{P} : \text{actCopy}(\mathfrak{p}, \bot, a_{lpOrg}) = a_{lpCp}$
    $it \leftarrow$ GetMaxIter(a_{lpOrg})
    if $it \neq \bot$ **then**
      $\mathfrak{P}_{peer} =$ GetInteractingParticipants(a_{lpOrg})
      $\text{loop}_{\mathfrak{P}}(a_{lpCp}) \leftarrow \mathfrak{P}_{peer}$
      UnrollSeqLoop($a_{lpCp}, \mathfrak{P}_{peer}$)
      $A^{unrollCp}_{\texttt{loop}} \leftarrow A^{unrollCp}_{\texttt{loop}} \cup \{a_{lpCp}\}$
    end if
  end for
  for all $a_{lpCp} \in A^{unroll}_{\texttt{loop}}$ **do**
    $\mathfrak{P}_{peer} =$ GetInteractingParticipants($\text{loop}_{\mathfrak{P}}(a_{lpCp})$)
    ConnectUnrolledSynchActs($a_{lpCp}, \mathfrak{P}_{peer}$)
  end for
end procedure

6.5. Verhaltensäquivalenz von ausgerollten Schleifen zu den originalen Schleifen

Die Eigenschaften, die eine ausgerollte Schleife in p_μ erfüllen muss, um verhaltensäquivalent zur originalen Schleife in der Choreographie c zu sein, wurden bereits in Abschnitt 6.3 beschrieben. Es wurde ebenfalls diskutiert, wie die Algorithmen, die das Ausrollen und die Verknüpfung der Teilnehmeriterationen implementieren, diese Eigenschaften in p_μ umsetzen. Im Folgenden wird noch einmal kurz zusammengefasst, wie die Eigenschaften erfüllt werden, die sich nicht über Zustandsbeziehungen diskutieren lassen. Danach wird im weiteren Verlauf dieses Abschnitts die Verhaltensäquivalenz von ausgerollten Schleifen zu originalen Schleifen detailliert diskutiert.

Die ausgerollten Teilnehmeriterationen müssen die Kommunikation mit jedem in p_μ abgebildeten Teilnehmer emulieren können, mit denen auch die Schleife in der Choreographie potentiell kommunizieren kann (Eigenschaft (i)). Die potentiellen Teilnehmer mit denen die Schleife kommunizieren kann, werden durch Algorithmus 6.2 bestimmt und Algorithmus 6.3 erstellt für jeden potentiellen Teilnehmer eine entsprechende Teilnehmeriteration. Eigenschaft (i) ist somit erfüllt.

Aus der Menge der potentiellen Teilnehmer darf zur Laufzeit nur die Kommunikation mit den Teilnehmern emuliert werden, die sich in der Menge der Teilnehmer befinden, mit denen zur Laufzeit tatsächlich kommuniziert werden soll (Eigenschaft (ii)). Bei ForEach-Schleifen wird diese Menge zum Beispiel über die in Abschnitt 3.2.4.10 definierte Abbildung partSet festgelegt. Die optionale Ausführung der Teilnehmeriterationen wird durch die in Algorithmus 6.3 als a_{nxt} bezeichnete exklusive Verzweigungsaktivität realisiert, die die Aktivitäten der Teilnehmeriteration synchronisiert. Befindet sich der Teilnehmer in der Menge der Teilnehmer, mit denen die Kommunikation emuliert werden soll, wird die in Algorithmus 6.3 als l_{nxt} bezeichnete Kante aktiviert und damit die Iteration ausgeführt. Andernfalls wird die als l_{skip} bezeichnete Kante aktiviert und die Ausführung der Teilnehmeriteration übersprungen.

Die sequentielle Ausführung der Teilnehmeriterationen (Eigenschaft (iii)) wird ebenfalls durch die Aktivität(en) a^i_{nxt} realisiert, da diese die ihr zugeordnete Iteration i und alle im Kontrollfluss nachfolgenden Teilnehmeriterationen synchronisieren.

Eine detaillierte Diskussion der Zustandsbeziehungen zwischen den Geschäftsaktivitäten innerhalb der Schleife, in deren Umgebung und den mit der Schleife kommunizierenden Teilnehmern erfolgt in den nächsten Abschnitten. Dabei werden diese Beziehungen in der Choreographie und in p_μ zuerst für sendende und dann für empfangende Schleifen diskutiert.

		A_{lpDesc}			A_{lpEnv}			A_{rcvEnv}		
		snd_{pred}	snd_{succ}	snd_{par}	a_{lpPred}	a_{lpPred}	a_{lpPred}	rcv_{pred}	rcv_{succ}	rcv_{par}
A_{lpDesc}	snd_{pred}		✓(1)	✓(1)	✓(6)	✓(6)	✓(6)	✓(3)	✗(4)	✓(3)
	snd_{succ}	✓(1)		✓(1)	✓(6)	✓(6)	✓(6)	✓(3)	✓(4)	✓(3)
	snd_{par}	✓(1)	✓(1)	✓(1)	✓(6)	✓(6)	✓(6)	✓(3)	✓(4)	✓(3)
A_{lpEnv}	a_{lpPred}	✓(6)	✓(6)	✓(6)	✓(8)	✓(8)	✓(8)	✓(5)	✓(7)	✓(5)
	a_{lpSucc}	✓(6)	✓(6)	✓(6)	✓(8)	✓(8)	✓(8)	✓(5)	✓(7)	✓(5)
	a_{lpPar}	✓(6)	✓(6)	✓(6)	✓(8)	✓(8)	✓(8)	✓(5)	✓(7)	✓(5)
A_{rcvEnv}	rcv_{pred}	✓(3)	✓(3)	✓(3)	✓(5)	✓(5)	✓(5)	✓(2)	✗(2)	✓(2)
	rcv_{succ}	✗(4)	✓(4)	✓(4)	✓(7)	✓(7)	✓(7)	✗(2)	✓(2)	✓(2)
	rcv_{par}	✓(3)	✓(3)	✓(3)	✓(5)	✓(5)	✓(5)	✓(2)	✓(2)	✓(2)

Abbildung 6.13.: Verhaltensäquivalenz der von der Materialisierung und dem Ausrollen einer sendenden Schleife betroffenen Aktivitäten

6.5.1. Zustandsbeziehungen bei sendenden Schleifen

Abbildung 6.13 zeigt die von der Materialisierung und dem Ausrollen betroffenen Aktivitäten und ob deren Zustandsbeziehungen in p_μ erhalten bleiben. Dabei handelt es sich um die Aktivitäten innerhalb und in der Umgebung der sendenden Schleife a_{loop} sowie um die Aktivitäten in der Umgebung der Empfangsaktivitäten. Im Folgenden werden die Zustandsbeziehungen zwischen diesen Aktivitäten in der Choreographie erläutert und mit denen ihrer Duplikate in p_μ nach dem Ausrollen der Schleife verglichen.

Die Menge A_{lpDesc} repräsentiert die direkten und indirekten Kindaktivitäten der Schleife, die untersucht werden. Da die Sendeaktivität *snd* in der Schleife wie bei Invoke-Receive-Interaktionen in Abschnitt 5.4.1.3 durch eine Synchronisationsaktivität ersetzt wird, umfasst die Menge wieder die Aktivitäten in der Umgebung der Schleife:

$$A_{lpDesc} := \{snd_{pred}, snd_{succ}, snd_{par}\}$$

Dabei ist snd_{pred} die direkte Vorgänger- und snd_{succ} die direkte Nachfolgeaktivität von *snd*. Die Aktivität snd_{par} ist die zu *snd* parallele Aktivität. Die Aktivitäten befinden sich in derselben Elternaktivität wie *snd*[1] und es existieren keine Kontrollflusskanten zwischen ihnen.

Die Menge A_{lpEnv} repräsentiert alle Aktivitäten in der Umgebung der Schleife, die sich in derselben Elternaktivität wie a_{loop} befinden:

$$A_{lpEnv} := \{a_{lpPred}, a_{lpSucc}, a_{lpPar}\}$$

Diese Aktivitäten werden betrachtet, da sie aufgrund ihrer direkten Zustandsbeziehungen zu a_{loop} vom Ausrollen der Schleife betroffen sind. Die Aktivität a_{lpPred} ist die direkte Vorgänger- und a_{lpSucc} die direkte Nachfolgeaktivität der Schleife. Die Aktivität a_{lpPar} ist die zu a_{loop} parallele Aktivität. Es existieren auch zwischen den Aktivitäten a_{lpPred}, a_{lpSucc} und a_{lpPar} keine Kontrollflusskanten.

Die Menge A_{rcvEnv} umfasst analog dazu die Aktivitäten in der Umgebung der Aktivität *rcv* in der Verhaltensbeschreibung der empfangenden Teilnehmer:

$$A_{rcvEnv} := \{rcv_{pred}, rcv_{succ}, rcv_{par}\}$$

Die Duplikate der Aktivitäten in p_{μ}, für die die Zustandsbeziehungen diskutiert werden, werden wieder mit dem Suffix *cp* im Subskript gekennzeichnet. Zusätzlich wird den Duplikaten der Kindaktivitäten in der ausgerollten Schleife *ul* im Subskript das Präfix *ul* vorangestellt. Die Duplikate der Kindaktivitäten in a_{loopCp} werden also durch die Menge $A_{lpDescCp}$ und die in der ausgerollten Schleife *ul* durch die Menge $A_{ulDescCp}$ repräsentiert. $A_{ulDescCp}$ be-

[1] Es wird zur Vereinfachung der Verifikation wieder davon ausgegangen, dass die Sendeaktivität und die Schleife nur jeweils eine parallele Aktivität sowie eine Vorgänger- und Nachfolgeaktivität besitzen. Diese können natürlich weitere Aktivitäten enthalten.

steht aus den Mengen $A^i_{ulDescCp}$ (mit $i \in |\mathfrak{P}_{loop}|$[1]), von denen jede die Duplikate der Kindaktivitäten in der jeweiligen Teilnehmeriteration enthält. Es gilt also:

$$A_{ulDescCp} := \bigcup_{1 \leq i \leq |\mathfrak{P}_{loop}|} A^i_{ulDescCp}$$

Die Duplikate der Umgebung der Empfangsaktivitäten in den Teilnehmer-Scopes der Empfänger werden durch die Menge $A_{rcvEnvCp}$ repräsentiert:

$$A_{rcvEnvCp} := A_{rcvPredCp} \cup A_{rcvParCp} \cup A_{rcvSuccCp}$$

Die Mengen $A_{rcvPredCp}$, $A_{rcvParCp}$ und $A_{rcvSuccCp}$ repräsentieren die Teilnehmerduplikate der Vorgängeraktivität rcv_{pred}, der parallelen Aktivität rcv_{par} und der Nachfolgeaktivität rcv_{succ} der Empfangsaktivität:

$$\begin{aligned} A_{rcvPredCp} &:= \{rcv^1_{predCp}, \ldots, rcv^n_{predCp}\} \\ A_{rcvParCp} &:= \{rcv^1_{parCp}, \ldots, rcv^n_{parCp}\} \qquad \text{mit } n = |\mathfrak{P}_{loop}| \\ A_{rcvSuccCp} &:= \{rcv^1_{succCp}, \ldots, rcv^n_{succCp}\} \end{aligned}$$

Das Element rcv^i_{predCp} repräsentiert zum Beispiel das Duplikat der Vorgängeraktivität von rcv, im Teilnehmer-Scope des Teilnehmers $\mathfrak{p}_i$.

Die Synchronisationsaktivitäten die in den Teilnehmer-Scopes den Empfang emulieren, werden durch die Menge SYN_{rcv} dargestellt.

[1] Im Gegensatz zu den vorherigen Abschnitten wird das Superskript in diesem Abschnitt dazu verwendet, die Ordnung zwischen Teilnehmeriterationen zu kennzeichnen.

(1) Beziehungen zwischen den Aktivitäten A_{lpDesc} der sendenden Schleife

Die Zustandsbeziehungen, die zwischen den Aktivitäten A_{lpDesc} eines Schleifenkörpers auf der Ebene einer einzelnen Iteration gelten, werden von den in der Schleife modellierten Kontrollflussbeziehungen bestimmt. Diese Beziehungen bleiben bei der Generierung der Teilnehmer-Container im erstellten Schleifenduplikat a_{loopCp} zwischen den Duplikaten der Kindaktivitäten $A_{lpDescCp}$ erhalten. Die Schleife a_{loopCp} ist daher verhaltensäquivalent zur Schleife a_{loop} (siehe Abschnitt 5.3.1). Beim Ausrollen von a_{loopCp} und der Erstellung einer Teilnehmeriterationen werden durch die Prozedur DuplicateLoopBody neben den Aktivitäten auch die Kontrollflusskanten sowie die Hierarchiebeziehungen zwischen ihnen dupliziert. Die Duplikate in $A^i_{ulDescCp}$ nutzen weiterhin die Variablen und Ereignisse in p_μ, auf die auch die Aktivitäten in a_{loopCp} zugegriffen haben. Da *ul* von einer sendenden Schleife erstellt wurde, erzeugt die Prozedur ConnectUnrolledSynchActs nur ausgehende Kanten, deren Zielaktivitäten in den Teilnehmer-Scopes liegen, die den Empfang von Nachrichten emulieren. Es werden also keine neuen Synchronisationsabhängigkeiten zwischen den Aktivitäten in $A^i_{ulDescCp}$ erzeugt. Folglich bleiben die Duplikate in $A^i_{ulDescCp}$ auf Iterationsebene zu den Aktivitäten in A_{lpDesc} verhaltensäquivalent (Eigenschaft (iv)).

Betrachtet man mehrere Iterationen, gelten zwischen den iterierbaren Aktivitäten in A_{lpDesc} bzw. $A_{lpDescCp}$ die Zustandsbeziehungen aus Abbildung 4.43. Um diese Zustandsbeziehungen mit denen der Duplikate in *ul* vergleichen zu können, werden die Duplikate einer Aktivität aus $A_{lpDescCp}$ in den jeweiligen Teilnehmeriterationen von *ul* wieder zu einer Aktivität abstrahiert. Die Aktivitäten snd^i_{predCp} und snd^{i+1}_{predCp} werden zum Beispiel als einzelne Aktivität betrachtet, da es sich bei ihnen um Duplikate der Aktivität snd_{predCp} handelt, die für die Teilnehmeriterationen i bzw. $i+1$ erstellt wurden. Daher gilt zwischen ihnen und den anderen Aktivitäten, mit denen ihre Zustände verglichen werden, z.B. hier den Aktivitäten aus *ul*, die mengentheoretische Vereinigung der Zustandsbeziehungen von $snd^1_{predCp}, \ldots, snd^n_{predCp}$ mit der zu vergleichenden Aktivität. Es gilt daher zwischen den Zuständen des regulären

Kontrollflusses von zwei Aktivitäten in *ul*, die kein Duplikat derselben Aktivität aus $A_{ulDescCp}$ sind, die Beziehung ||. Es ist also keine Reihenfolge zwischen den regulären Zuständen vorgegeben. Dies liegt, wie bei den Zustandsbeziehungen von Schleifen in Abschnitt 4.2.16 erläutert, darin begründet, dass während jeder Iteration die Instanzen der Aktivitäten alle Zustände des regulären Kontrollflusses erreichen können. Löst eine Instanz der Kindaktivitäten von *ul* einen Fehler aus oder wird sie terminiert, werden auch alle anderen Instanzen terminiert. Dies führt dann ebenfalls zur Terminierung der Instanzen der Aktivitäten aus $A_{ulDescCp}$ in den Nachfolgeiterationen. Die Instanzen der Aktivitäten $A_{ulDescCp}$ können also so lange jeden regulären Zustand erreichen, bis eine Instanz einen Fehlerzustand erreicht hat. Durch die Abstraktion gelten zwischen den Aktivitäten in $A_{ulDescCp}$ dieselben Zustandsbeziehungen wie für die originalen Aktivitäten in A_{lpDesc}.

(2) Beziehungen zwischen den Aktivitäten A_{rcvEnv} der Empfänger

Die Aktivitäten A_{rcvEnv} in der Umgebung der Receive-Aktivität sind mehreren Empfängern zugeordnet. Zwischen den Aktivitäten A_{rcvEnv} gelten die Zustandsbeziehungen, die in Abschnitt 5.4.1.3 für Invoke-Receive-Interaktionen erläutert wurden.

Die Generierung des Teilnehmer-Containers des Empfängers und die durch die ConnectUnrolledSynchActs generierte Kontrollflusskante l_{syn} bei der syn_{rcvCp} Zielaktivität ist, führen zu denselben Kontrollfluss- und damit auch Zustandsbeziehungen zwischen den Aktivitäten in $A^{i}_{envRcvCp}$ wie bei der materialisierten Invoke-Receive-Interaktion. Folglich können auch hier die Zustandsbeziehungen zwischen rcv^{i}_{predCp} und rcv^{i}_{succCp} nicht vollständig erhalten werden.

(3) Beziehungen zwischen den Aktivitäten A_{lpDesc} und rcv_{pred} bzw. rcv_{par}

Die Aktivitäten rcv_{pred} und rcv_{par} der empfangenden Teilnehmer, besitzen keine Synchronisationsabhängigkeit zur Empfangsaktivität *rcv*. Sie sind daher

		rcv_{succ}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
A_{lpDesc}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	‖	‖	‖	‖	‖	‖
	ABORTED	←	‖	‖	←	←	←	←
	EXECUTING	←	‖	‖	‖	‖	‖	‖
	TERMINATED	←	‖	‖	←	←	←	←
	FAULTED	←	‖	‖	←	←	←	←
	COMPLETED	←	‖	‖	‖	‖	‖	‖

Abbildung 6.14.: Zustandsprofil zwischen den Aktivitäten A_{lpDesc} im Schleifenkörper einer sendenden Schleife und der Aktivität rcv_{succ} in den Empfängern

wie bei Invoke-Receive-Interaktionen vollständig von allen Kindaktivitäten der Schleife A_{lpDesc} isoliert.

Die Duplikate $A_{ulDescCp}$ in den ausgerollten Iterationen befinden sich im Teilnehmer-Scope des sendenden Teilnehmers, während sich die Duplikate von rcv_{pred} bzw. rcv_{par}, d.h. $A_{rcvPredCp}$ und $A_{rcvParCp}$ im Teilnehmer-Scope des jeweiligen Empfängers befinden. Die von der Prozedur ConnectUnrolledSynchActs erstellten Kontrollflusskanten zwischen den Teilnehmer-Scopes führen ebenfalls zu keiner Synchronisationsabhängigkeit zwischen den Aktivitäten aus $A_{ulDescCp}$ und $A_{rcvPredCp}$ bzw. $A_{rcvParCp}$, da diese keine Synchronisationsabhängigkeit zu den den Empfang emulierenden Aktivitäten SYN_{rcv} haben. Es gilt also zwischen den Aktivitäten $A_{ulDescCp}$ und $A_{rcvPredCp}$ bzw. $A_{rcvParCp}$ das Profil für vollständig isolierte Aktivitäten.

(4) Beziehungen zwischen den Aktivitäten A_{lpDesc} und rcv_{succ}

Über die Aktivitäten *snd* und *rcv* besteht eine Zustandsabhängigkeit zwischen den Aktivitäten A_{lpDesc} und rcv_{succ}. Das aus dieser Abhängigkeit resultierende Profil zwischen den Aktivitäten A_{lpDesc} und rcv_{succ} ist in Abbildung 6.14 dargestellt. Die Aktivitäten *snd* und A_{lpDesc} befinden sich in derselben Flow-Aktivität, die wiederum die direkte Kindaktivität der Schleife ist. Daher wird der Zustand der Instanz von *snd* durch die Zustände der Instanzen von A_{lpDesc}

beeinflusst (siehe Abschnitt 4.2.5). Enden diese während einer Iteration in einem regulären Endzustand, kann die Instanz von *snd* in dieser oder in der darauf folgenden Iteration (falls die Instanz in *dead* endet) eine Nachricht senden. Die Instanzen von *rcv* in den Empfängern können *completed* erreichen und die Instanzen von rcv_{succ} (Axiom *Sq1*) können in *executing* gehen. Somit gilt aufgrund der Abstraktion von den Iterationen auch zwischen A_{lpDesc} und rcv_{succ} die Beziehung $\|$ (Axiom $Lp_{SND}1$). Die Zustände *dead* und *aborted* können die Instanzen von rcv_{succ} weiterhin unabhängig von denen von A_{lpDesc} erreichen. Endet eine Instanz von A_{lpDesc} während einer Iteration in einem Fehlerzustand, wird die Schleife beendet. Die Instanzen von *rcv* in den Empfängern, die noch keine Nachricht empfangen haben, können somit nicht mehr *completed* erreichen (Axiom $Lp_{SND}2$). Sie müssen entweder terminiert oder in den Zustand *dead* gesetzt werden. Damit können auch die Instanzen von rcv_{succ} der Empfänger nicht mehr den Zustand *executing* und dessen Nachfolgezustände erreichen (Axiom *Sq2* bzw. *Sq3*). Daher gilt zwischen den Fehlerzuständen von A_{lpDesc} und allen Zuständen von rcv_{succ} (außer *initial*) die Beziehung $\leftarrow$.

Nach dem Ausrollen der Schleife befinden sich in jeder Teilnehmeriteration i die Teilnehmerduplikate A^i_{lpDesc} von A_{lpDesc}. Die Synchronisationsaktivität syn^i_{snd} in der Teilnehmeriteration ist dabei über die Kontrollflusskante l^i_{syn} mit der Synchronisationsaktivität syn^i_{rcv} im Empfänger-Scope verbunden. Es besteht also eine materialisierte Invoke-Receive-Interaktion zwischen der Teilnehmeriteration i und dem Empfänger-Scope. Folglich gelten zwischen den regulären und den Fehlerzuständen der Aktivitäten in A^i_{lpDesc} und der Aktivität rcv^i_{succ} die Zustandsbeziehungen, die in Abschnitt 5.4.1.3 beschrieben wurden. Das impliziert auch, dass die Beziehungen zwischen snd^i_{pred} und rcv^i_{succ} nicht erhalten werden können.

Tritt bei der Ausführung der Instanzen $A^i_{ulDescCp}$ in Teilnehmeriteration i ein Fehler auf, werden die Instanzen von $A^{i+1}_{ulDescCp}, \ldots, A^n_{ulDescCp}$ der nachfolgenden Teilnehmeriterationen terminiert, da sie sich in derselben Elternaktivität *ul* befinden. Folglich werden auch die Kontrollflusskanten $l^{i+1}_{syn}, \ldots, l^n_{syn}$ deaktiviert und damit auch die Instanzen der Synchronisationsaktivitäten

$syn^{i+1}_{rcv}, \ldots, syn^{n}_{rcv}$ (Axiom $AndI_{Join}3$). In diesem Fall können die Instanzen von $rcv^{i+1}_{succCp}, \ldots, rcv^{n}_{succCp}$ den Zustand *executing* nicht mehr erreichen (Axiom *Sq2*). Die Aktivitäten aus $A_{rcvSuccCp}$ können also wie in der originalen Schleife den Zustand *executing* nur so lange erreichen, bis eine der Aktivitäten aus $A_{ulDescCp}$ einen Fehlerzustand erreicht hat. Aufgrund der Isolation des Sender- von den Empfänger-Scopes können die Instanzen von $A_{rcvSuccCp}$ weiterhin unabhängig von denen der Aktivitäten $A_{ulDescCp}$ in *aborted* gehen. Wird von konkreten Teilnehmeriterationen und Empfängern abstrahiert, gilt zwischen den regulären Zuständen der Aktivitäten $A_{ulDescCp}$ und Zuständen der Aktivitäten $A_{rcvSuccCp}$ weiter die Beziehung $\|$. Zwischen den Fehlerzuständen von $A_{ulDescCp}$ und dem Zustand *executing* sowie dessen Nachfolgezuständen der Aktivitäten in $A_{rcvSuccCp}$ besteht weiter die Beziehung $\leftarrow$.

(5) Beziehungen zwischen den Aktivitäten A_{lpEnv} und rcv_{pred} bzw. rcv_{par}

Da A_{lpDesc} und rcv_{pred} bzw. rcv_{par} vollständig voneinander isoliert sind und A_{lpEnv} nur Zustandsbeziehungen zu A_{lpDesc} besitzt, sind die Aktivitäten in A_{lpEnv} von den Aktivitäten in rcv_{pred} und rcv_{par} ebenfalls vollständig isoliert.

Wie oben erläutert, herrscht dadurch nach dem Ausrollen ebenfalls vollständige Isolation zwischen den Duplikaten $A_{ulDescCp}$ und Teilnehmerduplikaten von rcv_{pred}, also $A_{rcvPredCp}$ bzw. denen von rcv_{par}, also $A_{rcvParCp}$. Daher kann der Zustand von $A_{lpEnvCp}$ über $A_{ulDescCp}$ weder an die Aktivitäten $A_{rcvPredCp}$ noch $A_{rcvParCp}$ weiter propagiert werden. Somit sind auch die Aktivitäten in $A_{lpEnvCp}$ vollständig von denen in A_{PredCp} und $A_{rcvParCp}$ isoliert (und umgekehrt).

(6) Beziehungen zwischen den Aktivitäten A_{lpDesc} und A_{lpEnv}

Zwischen ihrer Vorgängeraktivität a_{lpPred} und der Schleife a_{loop} gelten die Zustandsbeziehungen der sequentiellen Ausführung (siehe Abbildung 4.18). Da es sich bei a_{loop} um eine strukturierte Aktivität handelt, propagiert deren Instanz ihren Zustand direkt an die Instanz ihrer Kindaktivität weiter (siehe Abschnitt 4.2.5), die diese wiederum an ihre Nachfahren weiter propagieren.

Folglich gelten zwischen ihren Nachfahren A_{lpDesc} und der Aktivität a_{lpPred} dieselben Zustandsbeziehungen, die zwischen A_{lpDesc} und a_{loop} gelten. Der Flow *ul*, der die ausgerollte Schleife repräsentiert, enthält von der Prozedur UnrollSeqLoop die Duplikate von A_{lpDesc} als Nachfahren und das Duplikat von a_{lpPred} als direkten Vorgänger. Zusätzlich bekommt *ul* die Eintrittsbedingung von a_{loop} zugewiesen. Daher bestehen auch zwischen $a_{lpPredCp}$ und $A_{ulDescCp}$ in p_μ dieselben Aktivitätszustandsbeziehungen wie zwischen a_{lpPred} und A_{lpDesc}.

Die Instanzen der Nachfahren von A_{lpDesc} müssen im regulären Kontrollfluss den Zustand *dead* oder *completed* erreichen, bevor die Instanz von a_{loop} in den Zustand *completed* gehen kann. Erst dann können die von a_{loop} ausgehenden Kontrollflusskanten evaluiert und die Instanz von a_{lpSucc} kann entweder in *dead* oder *executing* gehen. Endet eine Instanz von A_{lpDesc} in einem Fehlerzustand, geht danach auch die Instanz von a_{loop} in einen Fehlerzustand (Axiom $St_{CH}4$ bzw. $St_{CH}5$) und die Instanz von a_{lpSucc} wird vorzeitig terminiert (Axiom $St_{CH2CH}2$). Dieses Verhalten wird zwischen $A_{ulDescCp}$ und $a_{lpSuccCp}$ beibehalten, da $a_{lpSuccCp}$ die Nachfolgeaktivität der ausgerollten Schleife *ul* wird. Ein Fehler in den Instanzen von $A_{ulDescCp}$ führt also ebenfalls zur vorzeitigen Terminierung von $a_{lpSuccCp}$. Erreichen alle Instanzen von $A_{ulDescCp}$ einen regulären Endzustand, wird die Instanz von *ul* im Zustand *completed* oder *dead* beendet und das Duplikat $a_{lpSuccCp}$ kann ebendiese Zustände erreichen.

Die Aktivitäten a_{lpPar} und a_{loop} haben dieselbe Elternaktivität. Es gilt damit zwischen ihnen das Profil der Kindaktivitäten einer strukturierten Aktivität aus Abbildung 4.10. Dieses Profil gilt auch zwischen a_{lpPar} und a_{lpDesc}, da keine Synchronisationsabhängigkeit zwischen den Aktivitäten besteht, d.h. auch hier beeinflussen sich die Zustände des regulären Kontrollflusses nicht. Fehlerzustände der Instanz von a_{lpPar} werden über a_{loop} an die Instanzen von a_{lpDesc} propagiert (und umgekehrt). Sie führen damit zu deren Terminierung, falls sie keinen anderen Endzustand erreicht haben. Zwischen den Duplikaten $a_{lpParCp}$ und $A_{ulDescCp}$ gelten dieselben Zustandsbeziehungen, da in p_μ weiterhin keine Synchronisationsabhängigkeit zwischen diesen Aktivitäten besteht und die Fehlerzustände über *ul* propagiert werden.

(7) Beziehungen zwischen den Aktivitäten in A_{lpEnv} und rcv_{succ}

Die Aktivität a_{lpPred} beeinflusst den Zustand der Aktivität rcv_{succ} transitiv über a_{loop} und die Nachrichtenkante. Wie im vorherigen Abschnitt erläutert, muss a_{lpPred} einen Endzustand des regulären Kontrollflusses erreichen, dass die Schleife a_{loop} abhängig von ihrer Eintrittsbedingung ausgeführt wird und damit Nachrichten an die Empfänger gesendet werden können. Erreicht also a_{lpPred} einen regulären Endzustand, können die Instanzen von rcv_{succ} den Zustand *executing* und seine Nachfolgezustände erreichen. Geht a_{lpPred} in einen Fehlerzustand, müssen die Instanzen von rcv_{succ} terminiert oder deaktiviert werden. Daher gilt zwischen a_{lpPred} und rcv_{succ} dasselbe Profil wie zwischen snd_{pred} und rcv_{succ} aus Abbildung 5.10.

Die zur Schleife parallele Aktivität a_{lpPar} hat keine Synchronisationsabhängigkeit zu a_{loop} und ihren Nachfahren A_{lpDesc}. Daher beeinflusst die Instanz von a_{lpPar} im regulären Kontrollfluss weder den Zustand der Instanz von A_{lpDesc} noch den Zustand der Instanzen von rcv und rcv_{succ}. Löst die Instanz von a_{lpPar} einen Fehler aus oder wird sie terminiert, führt dies zur Terminierung der Instanz von a_{loop}, falls deren Ausführung vorher nicht beendet wurde. In diesem Fall können die Instanzen von rcv in den Empfängern, die noch keine Nachricht erhalten haben, nicht *completed* und die zugehörigen Instanzen von rcv_{succ} nicht den Zustand *executing* erreichen. Tritt der Fehler oder die Terminierung der Instanz von a_{lpPar} erst nach der Ausführung von a_{loop} auf, hat dies keine Auswirkungen auf die Zustände der Instanzen von rcv_{succ}. Es gilt also auch zwischen a_{lpPar} und rcv_{succ} dasselbe Profil, das für Invoke-Receive-Interaktionen in Abbildung 5.14 dargestellt ist.

Die Nachfolgeaktivität a_{lpSucc} hat keinen Einfluss auf den Zustand der Instanzen von rcv_{succ}, da deren Instanz erst ausgeführt wird, nachdem die Schleife a_{loop} mit den Instanzen der Sendeaktivitäten ausgeführt wurde. Es besteht also zwischen a_{lpSucc} und rcv_{succ} die vollständige Isolation.

Diese Zustandsbeziehungen bleiben in p_{μ} auch zwischen den Duplikaten $A_{rcvSuccCp}$ und $A_{lpEnvCp}$ (also $a_{lpPredCp}$, $a_{lpParCp}$ bzw. $a_{lpSuccCp}$) erhalten. Dies liegt

zum einen darin begründet, dass, wie im vorherigen Abschnitt erläutert, die Zustandsbeziehungen zwischen $A_{lpEnvCp}$ und $A_{ulDescCp}$ dieselben sind, wie zwischen A_{lpEnv} und A_{lpDesc}. Zum anderen bleiben in jeder Teilnehmeriteration die Zustände zwischen $A^{i}_{sndPredCp}$ und rcv^{i}_{SuccCp} erhalten. Über die Aktivitäten $A_{sndPredCp}$ ($A_{sndPredCp} \subset A_{ulDescCp}$) haben die Aktivitäten in $A_{lpEnvCp}$ eine implizite Zustandsbeziehung zu den Aktivitäten in $A_{rcvSuccCp}$.

(8) Beziehungen zwischen den Aktivitäten A_{lpEnv} der Schleifenumgebung

Die Aktivitäten a_{lpPar} und a_{lpPred} sowie a_{lpPar} und a_{lpSucc} haben keine Synchronisationsabhängigkeiten zueinander und sie befinden sich in derselben strukturierten Aktivität. Daher gilt zwischen ihnen das Profil für die Kindaktivitäten einer strukturierten Aktivität, das in Abbildung 4.10 dargestellt ist. Weder das Ausrollen der Schleife, noch die neuen Kontrollflusskanten zwischen den Synchronisationsaktivitäten in der ausgerollten Schleife und den Teilnehmer-Scopes erzeugen Zustandsabhängigkeiten zwischen deren Duplikaten. Also gilt auch zwischen den Aktivitäten $a_{lpParCp}$ und $a_{lpSuccCp}$ sowie zwischen $a_{lpPredCp}$ und $a_{lpParCp}$ das Profil aus Abbildung 4.10.

Die Aktivitäten a_{lpPred} und a_{lpSucc} haben über a_{loop} eine Synchronisationsabhängigkeit zueinander. In p_{μ} besteht die Synchronisationsabhängigkeit zwischen den Duplikaten $a_{lpPredCp}$ und $a_{lpSuccCp}$ über die ausgerollte Schleife *ul*, die die Eintrittsbedingung von a_{loop} zugewiesen bekommt. Es bleiben also auch hier die originalen Zustandsbeziehungen erhalten.

(9) Beziehungen zwischen den Aktivitäten unterschiedlicher empfangender Teilnehmer[1]

Zwischen den Instanzen der Aktivitäten A_{rcvEnv} der unterschiedlichen empfangenden Teilnehmer bestehen durch die Teilnehmerisolation keine Zustandsabhängigkeiten. Die Duplikate von A_{rcvEnv} in den verschiedenen Teilnehmer-

[1] Die Beziehungen zwischen unterschiedlichen Teilnehmern sind in Abbildung 6.13 aus Platzgründen nicht dargestellt.

Scopes, die die empfangenden Teilnehmer repräsentieren, haben ebenfalls keine Zustandsabhängigkeiten zueinander (siehe Abschnitt 5.3). Die Materialisierung der Interaktion und das Ausrollen der Schleife erzeugt nur Synchronisations- und somit Zustandsabhängigkeiten zwischen den Aktivitäten im Teilnehmer-Scope des Senders und denen in den Teilnehmer-Scopes der Empfänger. Es werden aber keine Zustandsabhängigkeiten zwischen den Aktivitäten des Empfänger-Scopes erzeugt. Daher gilt zwischen $A^1_{rcvEnvCp}, \ldots, A^n_{rcvEnvCp}$ weiterhin die vollständige Isolation.

6.5.2. Zustandsbeziehungen bei empfangenden Schleifen

Analog zum vorherigen Abschnitt werden hier die Zustandsbeziehungen zwischen den Geschäftsaktivitäten untersucht, die eine direkte Kontrollflussbeziehung zur empfangenden Schleife a_{loop} und zu der den Sendern zugeordneten Sendeaktivität *snd* haben.

Es werden dieselben Mengennotationen wie im vorherigen Abschnitt verwendet. Die Menge A_{lpDesc} repräsentiert die Nachfahren der empfangenden Schleife und die Menge A_{lpEnv} die Aktivitäten in ihrer Umgebung. Die Aktivität A_{sndEnv} umfasst die Aktivitäten in der Umgebung der Sendeaktivität:

$$A_{sndEnv} := \{snd_{pred}, snd_{succ}, snd_{par}\}$$

Die Kindaktivitäten der ausgerollten empfangenden Schleife *ul* werden durch die Menge $A_{ulDescCp}$ und die Duplikate der Umgebung der Sendeaktivitäten in den Teilnehmer-Scopes durch die Menge $A_{sndEnvCp}$ repräsentiert:

$$A_{sndEnvCp} := A_{sndPredCp} \cup A_{sndParCp} \cup A_{sndSuccCp}$$

Die Mengen $A_{sndPredCp}$, $A_{sndParCp}$ und $A_{sndSuccCp}$ repräsentieren die Teilnehmerduplikate der Vorgängeraktivität snd_{pred}, der parallelen Aktivität snd_{par} und der Nachfolgeaktivität snd_{succ} der Sendeaktivität.

		A_{lpDesc}			A_{lpEnv}			A_{sndEnv}		
		rcv_{pred}	rcv_{succ}	rcv_{par}	a_{lpPred}	a_{lpSucc}	a_{lpPar}	snd_{pred}	snd_{succ}	rcv_{par}
A_{lpDesc}	rcv_{pred}		***X***(1)	✓(1)	✓(6)	✓(6)	✓(6)	✓(3)	✓(3)	✓(3)
	rcv_{succ}	***X***(1)		✓(1)	✓(6)	✓(6)	✓(6)	***X***(4)	✓(4)	✓(4)
	rcv_{par}	✓(1)	✓(1)	✓(1)	✓(6)	✓(6)	✓(6)	✓(3)	✓(3)	✓(3)
A_{lpEnv}	a_{lpPred}	✓(6)	✓(6)	✓(6)	✓(7)	✓(7)	✓(7)	✓(5)	✓(5)	✓(5)
	a_{lpSucc}	✓(6)	✓(6)	✓(6)	✓(7)	✓(7)	✓(7)	***X***(5)	✓(5)	✓(5)
	a_{lpPar}	✓(6)	✓(6)	✓(6)	✓(7)	✓(7)	✓(7)	✓(5)	✓(5)	✓(5)
A_{sndEnv}	snd_{pred}	✓(3)	***X***(4)	✓(3)	✓(5)	***X***(5)	✓(5)	✓(2)	✓(2)	✓(2)
	snd_{succ}	✓(3)	✓(4)	✓(3)	✓(5)	✓(5)	✓(5)	✓(2)	✓(2)	✓(2)
	snd_{par}	✓(3)	✓(4)	✓(3)	✓(5)	✓(5)	✓(5)	✓(2)	✓(2)	✓(2)

Abbildung 6.15.: Verhaltensäquivalenz der von der Materialisierung und dem Ausrollen einer empfangenden Schleife betroffenen Aktivitäten

Inwieweit die Zustandsbeziehungen zwischen diesen Aktivitäten in p_μ erhalten werden können, ist in Abbildung 6.15 dargestellt.

(1) Beziehungen zwischen den Aktivitäten A_{lpDesc} der empfangenden Schleife

Auf Ebene einer einzelnen Iteration der empfangenden Schleife gelten zwischen den Kindaktivitäten A_{lpDesc} die gleichen Zustandsbeziehungen wie für die Aktivitäten eines Empfängers bei einer Invoke-Receive-Interaktion (siehe Abschnitt 5.4.1.3). Über Iterationen hinweg gelten zwischen den Aktivitäten in A_{lpDesc}, wie bei sendenden Schleifen, die Zustandsbeziehungen, die im Profil in Abbildung 4.43 dargestellt sind.

Wie für die Kindaktivitäten einer sendenden Schleife diskutiert, bleiben die Zustandsbeziehungen im Duplikat der Schleife in p_μ und beim Ausrollen des Schleifenkörpers erhalten. Um die Invoke-Receive-Interaktion zwischen einem Sender und der ihm zugeordneten Teilnehmeriteration der empfangenden Schleife zu materialisieren, erzeugt die Prozedur CONNECT-

UnrolledSynchActs eine Kante, deren Zielaktivität sich in der Teilnehmeriteration befindet. Es gelten auf Ebene einer ausgerollten Iteration i zwischen rcv^i_{predCp}, rcv^i_{parCp} und rcv^i_{succCp} die Beziehungen der materialisierten Invoke-Receive-Interaktion. Daher werden die Beziehungen zwischen dem Zustand *dead* von rcv^i_{predCp} und den Zuständen von rcv^i_{succCp}, verglichen mit den originalen Aktivitäten, eingeschränkt (siehe Profil in Abbildung 5.11). Werden Duplikate in den Iterationen und den Sendern abstrahiert, ergeben sich dieselben Zustandsbeziehungen wie zwischen den Duplikaten in der originalen Schleife.

(2) Beziehungen zwischen den Aktivitäten A_{sndEnv} der Sender

Die Aktivitäten A_{sndEnv} in der Umgebung der Invoke-Aktivität *snd* sind mehreren Sendern zugeordnet. Es gelten die Zustandsbeziehungen, die in Abschnitt 5.4.1.3 für die Geschäftsaktivitäten des sendenden Teilnehmers in Invoke-Receive-Interaktionen erläutert wurden.

Die Generierung der Teilnehmer-Container der Sender erhalten die Zustandsbeziehungen zwischen den Aktivitätsduplikaten $A_{sndEnvCp}$ eines Senders. Die bei der Materialisierung der Invoke-Receive-Interaktionen und durch die Prozedur ConnectUnrolledSynchActs erstellten Kontrollflusskanten haben ihre Quellaktivität im Teilnehmer-Scope des jeweiligen Senders und die Zielaktivität in der zugeordneten Teilnehmeriteration von *ul*. Folglich werden durch die Kanten keine neuen Zustandsbeziehungen zwischen den Aktivitäten $A_{sndEnvCp}$ in einem Sender impliziert. Die Zustandsbeziehungen von $A_{sndEnvCp}$ sind also zu denen von A_{sndEnv} äquivalent.

(3) Beziehungen zwischen den Aktivitäten A_{sndEnv} und rcv_{pred} bzw. rcv_{par}

Auf Ebene einer einzelnen Iteration bzw. Interaktion gilt zwischen den Zuständen der Aktivitäten A_{sndEnv} in der Umgebung der Sendeaktivität und den Aktivitäten rcv_{pred} bzw. rcv_{par} die Beziehung ||. Analog zur Invoke-Receive-Interaktion liegt der Grund darin, dass rcv_{pred} bzw. rcv_{par} und A_{sndEnv} ver-

schiedenen Teilnehmer zugeordnet sind und erstere nicht von der Empfangsaktivität *rcv* synchronisiert werden. Fehler in den Instanzen rcv_{pred} bzw. rcv_{par} können aufgrund der Teilnehmerisolation nicht zu den Instanzen der Aktivitäten aus A_{sndEnv} propagiert werden. Da die Instanzen von rcv_{pred} und rcv_{par} während jeder Iteration alle regulären und Fehlerzustände durchlaufen können, unabhängig davon, welche Zustände die Instanzen von A_{sndEnv} in den verschiedenen Teilnehmern erreichen, besteht auch über mehrere Iterationen hinweg zwischen den Zuständen (ausgenommen *initial*) dieser Aktivitäten die Beziehung ∥.

Auf Ebene einer einzelnen Teilnehmeriteration gilt zwischen den Zuständen der Duplikate rcv^{i}_{predCp} bzw. rcv^{i}_{parCp} und $A^{i}_{sndEnvCp}$ weiterhin die Beziehung ∥. Zum einen weil sich die Duplikate $A^{i}_{sndEnvCp}$ in anderen Teilnehmer-Scopes befinden als rcv^{i}_{predCp} bzw. rcv^{i}_{parCp} und zum anderen weil diese Aktivitäten keine Synchronisationsabhängigkeiten zu $A^{i}_{sndEnvCp}$ haben.

Allerdings haben die Aktivitäten rcv^{i+1}_{predCp} und rcv^{i+1}_{parCp} eine Synchronisationsabhängigkeit zu $snd^{i}_{predCp} \in A_{sndEnvCp}$. Die Instanzen der Aktivitäten rcv^{i+1}_{predCp} und rcv^{i+1}_{parCp} können nämlich erst ausgeführt werden, wenn alle Aktivitäten der Teilnehmeriteration *i* einen regulären Endzustand erreicht haben. Dies ist nur möglich, wenn die Synchronisationsaktivitäten, die die Interaktion mit der Iteration *i* emulieren, einen Endzustand erreicht haben. Was wiederum impliziert, dass die Vorgängeraktivität snd^{i}_{succCp} der Synchronisationsaktivität, die das Senden emuliert, ebenfalls einen Endzustand erreicht haben muss. Es kommt also auch hier aufgrund der Materialisierung zur verzögerten Ausführung von Aktivitätsinstanzen, die in Abschnitt 5.4.1.3 bereits für Invoke-Receive-Interaktionen beschrieben wurde. Durch die Abstraktion der Duplikate in den Teilnehmeriterationen in eine einzelne Aktivität geht deren Reihenfolge verloren. Daher gilt zwischen den Zuständen der Duplikate rcv_{predCp} bzw. rcv_{parCp} und $A_{sndEnvCp}$ ebenfalls die Beziehung ∥.

		rcv_{succ}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
snd_{pred} / snd_{par}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	‖	‖	‖	‖	‖	‖
	ABORTED	←	‖	‖	‖	‖	‖	‖
	EXECUTING	←	‖	‖	‖	‖	‖	‖
	TERMINATED	←	‖	‖	‖	‖	‖	‖
	FAULTED	←	‖	‖	‖	‖	‖	‖
	COMPLETED	←	‖	‖	‖	‖	‖	‖

Abbildung 6.16.: Zustandsprofil zwischen den Aktivitäten snd_{pred} und snd_{par} des Senders und der Aktivität rcv_{succ} in der empfangenden Schleife

(4) Beziehungen zwischen den Aktivitäten A_{sndEnv} und rcv_{succ}

Die Instanzen der Senderaktivität snd_{succ} haben keinen Einfluss auf die Zustände der Instanzen von rcv_{succ}. Die Instanzen der Aktivitäten snd_{pred} und snd_{par} können hingegen aufgrund der Nachrichtenkante zwischen *snd* und *rcv* den Zustand der Instanzen von rcv_{succ} beeinflussen. Die Zustandsbeziehungen zwischen der Aktivität *snd* der Sender und der Aktivität *rcv* des Empfängers wurden in Abschnitt 4.2.16.2 erläutert. Auf Basis dieser Beziehungen und der Beziehungen zwischen snd_{pred} und snd_{par} mit *snd* bzw. *rcv* resultiert das Profil in Abbildung 6.16.

Durch die Abstraktion von den Iterationen gilt, abgesehen vom Zustand *initial*, zwischen allen Zuständen von snd_{pred} bzw. snd_{par} und rcv_{succ} die Beziehung ‖. Falls die Instanz von snd^{i}_{pred} in einem Sender einen anderen Endzustand als *completed* erreicht, kann keine Nachricht gesendet werden. Die Instanz von rcv^{i}_{succ} in der empfangenden Teilnehmeriteration i kann dann nur deaktiviert oder terminiert werden, da die Instanz von *rcv* nicht *completed* erreichen kann. Eine Instanz von snd^{i}_{par} kann das Senden im Fehlerfall verhindern, falls sie einen Fehlerzustand erreicht hat, bevor die Nachricht gesendet wurde, da dieser Fehlerzustand auch die Instanz der Sendeaktivität terminieren würde.

Aufgrund der Isolation der Sender voneinander können aber Instanzen der

Sendeaktivität in den anderen Sendern unabhängig davon Nachrichten an die empfangende Schleife senden. Folglich können Instanzen von rcv_{succ} in anderen Iterationen weiter potentiell ausgeführt werden.

Es gelten auf Ebene einer ausgerollten Iteration i zwischen snd^{i}_{predCp} bzw. snd^{i}_{parCp} und rcv^{i}_{succCp} die Beziehungen der materialisierten Invoke-Receive-Interaktion. Folglich sind die Beziehungen zwischen dem Zustand *dead* von rcv^{i}_{succCp} und den Zuständen von snd^{i}_{predCp} (siehe Profil in Abbildung 5.11) verglichen mit den originalen Aktivitäten eingeschränkt. Werden Duplikate in den Iterationen und den Sendern abstrahiert, ergeben sich wieder dieselben Zustandsbeziehungen wie zwischen snd_{pred} und rcv_{succ}.

(5) Beziehungen zwischen den Aktivitäten A_{sndEnv} und A_{lpEnv}

Die Aktivitäten a_{lpPred} bzw. a_{lpPar} aus A_{lpEnv} befinden sich in einem anderen Teilnehmer als die Aktivitäten in A_{sndEnv}. Sie haben keine Synchronisationsabhängigkeit zur Schleife a_{loop} und damit auch nicht transitiv zu den Aktivitäten in A_{sndEnv}. Die Instanzen von a_{lpPred} bzw. a_{lpSucc} können somit unabhängig von den Instanzen der Aktivitäten in A_{sndEnv} alle Nachfolgezustände von *initial* erreichen. Es gilt daher zwischen den Zuständen der Aktivitäten a_{lpPred} bzw. a_{lpSucc} und den Zuständen der Aktivitäten in A_{sndEnv} (außer *initial*) die Beziehung ||.

Die Aktivität a_{lpSucc} kann als Nachfolgeaktivität der empfangenden Schleife den Zustand *executing* erst erreichen, wenn alle Iterationen der Schleife beendet wurden. Allerdings können die Instanzen der Kindaktivitäten der Schleife unabhängig von den Instanzen der Aktivitäten in A_{sndEnv} terminiert und deaktiviert werden, z.B. wenn keine Iteration von a_{loop} ausgeführt wird. Das heißt, die Instanzen von a_{lpSucc} können auch vor den Instanzen aus A_{sndEnv} *executing* oder einen Endzustand erreichen. Es gilt also zwischen den Zuständen von a_{lpSucc} und A_{sndEnv} ebenfalls die Beziehung ||.

Zwischen den Zuständen der Duplikate $a_{lpPredCp}$ bzw. $a_{lpParCp}$ und den Aktivitäten in A_{sndEnv} gilt in p_{μ} weiterhin die Beziehung ||, da diese keine Zu-

		$a_{lpSuccCp}$						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
snd_{predCp}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	→	‖	→	→	→	→
	ABORTED	←	→	‖	→	→	→	→
	EXECUTING	←	→	‖	→	→	→	→
	TERMINATED	←	→	‖	→	→	→	→
	FAULTED	←	→	‖	→	→	→	→
	COMPLETED	←	→	‖	→	→	→	→

Abbildung 6.17.: Zustandsprofil zwischen der Aktivität snd_{predCp} des Senders und der Nachfolgeaktivität $a_{lpSuccCp}$ der ausgerollten empfangenden Schleife

standsbeziehungen zu a_{loopCp} besitzen. Es ist allerdings nicht mehr möglich, dass die Instanz von $a_{lpSuccCp}$ in den Zustand *dead* oder *executing* geht, bevor die Instanzen von snd_{predCp} in den Sendern einen Endzustand erreicht haben. Dies liegt an den von der Prozedur ConnectUnrolledSynchActs erstellten Kontrollflusskanten zwischen den Synchronisationsaktivitäten in den Teilnehmer-Scopes der Sender und denen in den ausgerollten Teilnehmeriterationen der Schleife *ul*. Die Synchronisationsaktivitäten müssen einen Endzustand erreichen, bevor die Kanten deaktiviert werden (siehe Axiom *SqI3* bzw. Abschnitt 5.4.1.3) und somit auch alle Aktivitätsinstanzen in *ul* einen Endzustand erreichen können. Dies ist wiederum die Voraussetzung dafür, dass die Instanz von *ul* in den Zustand *completed* (Axiom $St_{CH}6$) und die Instanz von $rcv_{lpSuccCp}$ in *executing* oder *dead* gehen kann.

Daher gilt zwischen den Aktivitäten das Profil in Abbildung 6.17. Im Vergleich zur Choreographie kann es also zu einer zeitlich verzögerten Ausführung von $rcv_{lpSuccCp}$ kommen. Das Problem der verzögerten Ausführung durch die bei der Materialisierung erzeugten Kontrollflusskanten wurde bereits für Invoke-Receive-Interaktionen diskutiert.

		a_{lpSucc}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{lpDesc}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	→	→	→	→	→	→
	ABORTED	←	⊕	‖	⊕	⊕	⊕	⊕
	EXECUTING	←	⊕	→	→	→	→	→
	TERMINATED	←	⊕	‖	⊕	⊕	⊕	⊕
	FAULTED	←	⊕	→	⊕	⊕	⊕	⊕
	COMPLETED	←	⊕	→	→	→	→	→

Abbildung 6.18.: Zustandsprofil zwischen den Nachfahren $a_{lpDesc} \in A_{lpDesc}$ und der Nachfolgeaktivität a_{lpSucc} der empfangenden Schleife

(6) Beziehungen zwischen den Aktivitäten A_{lpDesc} und A_{lpEnv}

Die Aktivitäten in A_{lpEnv} befinden sich in derselben strukturierten Aktivität wie a_{loop}. Im regulären Kontrollfluss bestehen keine Zustandsbeziehungen zwischen der Aktivität a_{lpPred} bzw. a_{lpPar} und den Aktivitäten in A_{lpDesc}, da diese keine Synchronisationsabhängigkeit zueinander haben. Fehler in den Instanzen von a_{lpPred} oder a_{lpPar} werden über a_{loop} zu den Instanzen von A_{lpDesc} propagiert und umgekehrt. Es gelten also über a_{loop} zwischen a_{lpPred} bzw. a_{lpPar} und A_{lpDesc} dieselben Beziehungen wie zwischen den Kindaktivitäten einer strukturierten Aktivität, die in Abbildung 4.10 dargestellt sind.

Da a_{lpSucc} eine direkte Nachfolgeaktivität von a_{loop} ist, muss die Instanz von a_{loop} den Zustand *completed* erreichen, bevor die Instanz von a_{lpSucc} in *executing* oder *dead* gehen kann (Axiom *Sq1* bzw. *Sq2*). Dies ist wiederum nur möglich, wenn die Instanzen der Nachfahren von A_{lpDesc} einen regulären Endzustand erreicht haben (Axiom $St_{CH}6$). Folglich gilt zwischen den Zuständen *completed* und *dead* der Aktivitäten in A_{lpDesc} und den Zuständen *executing* und *dead* der Aktivität a_{lpSucc} die Beziehung →. Aufgrund der sequentiellen Ausführung der Aktivitäten in A_{lpDesc} und a_{lpSucc} wird, falls eine Instanz von A_{lpDesc} einen Fehler auslöst, die Instanz von a_{lpSucc} erst danach terminiert. Es gilt daher zwischen den Aktivitäten das Profil in Abbildung 6.18.

Die Kontrollflussbeziehungen zwischen $A_{lpEnvCp}$ und *ul* sind dieselben, wie

zwischen A_{lpEnv} und a_{loop}. Die Duplikate $A_{ulDescCp}$ sind Kindaktivitäten von ul. Es gelten folglich die gleichen Zustandsbeziehungen zwischen $A_{lpEnvCp}$ und $A_{ulDescCp}$ wie zwischen A_{lpEnv} und A_{lpDesc}.

(7) Beziehungen zwischen den Aktivitäten in A_{lpEnv}

Die Zustandsbeziehungen zwischen den Aktivitäten in A_{lpEnv} sind dieselben, die für A_{lpEnv} bei sendenden Schleifen beschrieben wurden. Aus den gleichen Gründen wie bei den sendenden Schleifen bleiben die Beziehungen zwischen den Duplikaten in $A_{lpEnvCp}$ erhalten.

6.6. Fusionieren von interagierenden While-Schleifen

Die Materialisierung des Szenarios IV erzeugt grenzverletzende Kontrollflusskanten, deren Quell- und Zielaktivitäten sich jeweils in unterschiedlichen While-Schleife befinden. Wie in Abschnitt 6.2 erläutert, kann deren maximale Anzahl an Iterationen nicht bestimmt und die Schleifen können somit nicht ausgerollt werden. Wagner, Kopp und Leymann [WKL15] beschreiben, wie die grenzverletzenden Kontrollflusskanten dennoch eliminiert werden können, indem die interagierenden While-Schleifen in eine einzelne Schleife *fusioniert* werden.

Das Fusionieren von Schleifen ist auch eine Technik aus dem Compiler-Bau [Muc97], die zum Beispiel dazu verwendet wird, um die Laufzeit von eingebetteten Systemen zu optimieren [QCS02]. Die Schleifenfusion wird dort ebenfalls als Alternative zum Ausrollen von Schleifen verwendet [Dar99]. Die Grundidee dahinter ist, dass der Code von zwei Schleifenkörpern in einen einzelnen Schleifenkörper zusammengefasst wird, um so die Iterationsanzahl und die Häufigkeit der Evaluierung von Bedingungen zu minimieren.

Der eingangs erwähnte Ansatz von Wagner, Kopp und Leymann wird hier nur informell skizziert, da dieser auf einem einfacheren Metamodell basiert, das keine (vorzeitige) Terminierung von Aktivitäten kennt. Der Ansatz kann

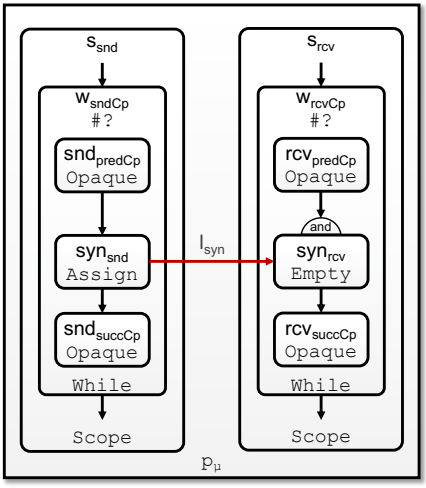

Abbildung 6.19.: Szenario IV nach der Materialisierung

damit, wie im weiteren Verlauf erläutert, bestimmte Fehlerszenarien nicht abdecken. Die Ausgangsposition für das Fusionieren von zwei interagierenden While-Schleifen ist wie beim Ausrollen, dass die Teilnehmerduplikate der While-Schleifen in p_μ erstellt wurden und dass die Synchronisationsaktivitäten innerhalb der Schleifen über grenzverletzende Kontrollflusskanten miteinander verbunden sind. Die ist im Beispiel in Abbildung 6.19 dargestellt, wo die Synchronisationsaktivitäten in zwei While-Schleifen über die grenzverletzende Kontrollflusskante l_{syn} verbunden sind.

Die grenzverletzenden Kontrollflusskanten werden bei der Fusion von Schleifen dadurch aufgelöst, dass in p_μ eine neue While-Schleife w_{fused} erstellt wird, in die die Schleifenkörper der beiden Schleifen eingefügt werden. Das heißt, es werden alle Aktivitäten der beiden Schleifenkörper inklusive deren Kontrollflussbeziehungen dem Schleifenkörper von w_{fused} hinzugefügt. Die Schleifenkörper werden dabei analog zu den Teilnehmer-Scopes wieder über einen Scope voneinander isoliert, so dass Fehler in einem Schleifenkörper nicht zur Terminierung der Aktivitäten im anderen Schleifenkörper führen.

Die Scopes fungieren also als Teilnehmer-Container innerhalb der Schleife. Dies ist in Abbildung 6.20 illustriert (die Fault-Handler der Scopes sind der Übersichtlichkeit wegen nicht dargestellt). Die Schleife w_{fused} enthält die Schleifenkörper von w_{sndCp} und w_{rcvCp} aus Abbildung 6.19 sowie die Kontrollflusskante l_{syn} zwischen ihnen. Die Schleifenbedingung von w_{fused} ist die Disjunktion der Schleifenbedingungen der fusionierten Schleifen, im Beispiel also die Disjunktion der Schleifenbedingungen von w_{sndCp} und w_{rcvCp}. Dies stellt zur Laufzeit sicher, dass von w_{fused} immer eine Iteration ausgeführt wird, wenn vom Schleifenkörper w_{sndCp} oder w_{rcvCp} eine Iteration ausgeführt werden muss, weil deren jeweilige Schleifenbedingung wahr ist. Es darf dabei nur der Schleifenkörper ausgeführt werden, dessen Schleifenbedingung wahr ist. Dazu erhält jeder Schleifenkörper eine exklusive Verzweigungsaktivität a_{entry} als direkten Vorgänger und eine exklusive Vereinigungsaktivität a_{exit} als direkten Nachfolger. Aktivität a_{entry} ist mit der Eingangsaktivät des Schleifenkörpers über die Kontrollflusskante l_{nxt} verbunden. Die Aktivitäten a_{entry} und a_{exit} sind über eine Kontrollflusskante l_{skip} miteinander verbunden. Die Kante l_{nxt} erhält als Transitionsbedingung die Schleifenbedingung der Schleife, aus der der Schleifenkörper stammt. Folglich wird die Kante aktiviert, wenn die Schleifenbedingung wahr ist. Andernfalls wird die Kante l_{skip} aktiviert, die als Transitionsbedingung die Negation der Schleifenbedingung zugewiesen bekommt.

Im regulären Kontrollfluss kommt es in der fusionierten Schleife w_{fused} zur *Iterationsabhängigkeit* zwischen Aktivitäten, die ursprünglich aus verschiedenen Schleifen stammen. Das bedeutet, dass deren Instanzen über mehrere Iterationen hinweg nicht mehr unabhängig voneinander alle Zustände durchlaufen können. Das impliziert wiederum, dass es verglichen mit der Choreographie zur verzögerten Ausführung von Aktivitätsinstanzen kommt, da die nächste Iteration von w_{fused} erst ausgeführt werden kann, wenn alle Aktivitätsinstanzen in ihrem Körper einen Endzustand erreicht haben. Haben in dem Beispiel in Abbildung 6.19 die Aktivitäten in w_{sndCp} eine kürzere Ausführungszeit als die in w_{rcvCp}, werden die Iterationen von w_{sndCp} und damit die Aktivitätsinstanzen in ihrem Körper schneller durchlaufen als die von

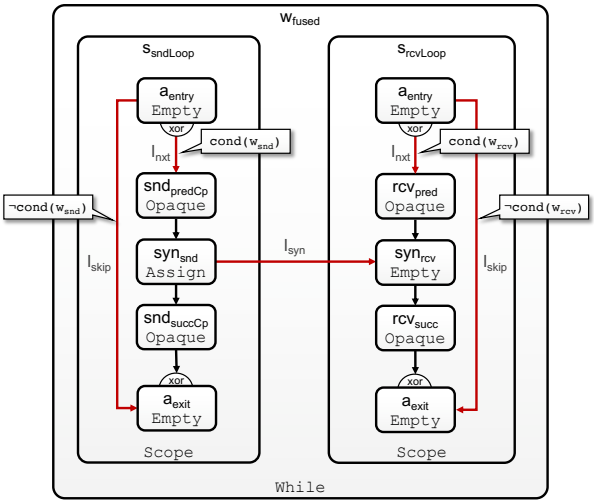

Abbildung 6.20.: Szenario aus Abbildung 6.6 nach Materialisierung

w_{rcvCp}. In der fusionierten Schleife müssen die Instanzen der Aktivitäten, die aus w_{sndCp} stammen, mit der Ausführung „warten“, bis alle Instanzen aus w_{rcvCp} in der vorherigen Iteration einen Endzustand erreicht haben.

Es wurde bisher nicht darauf eingegangen, wo w_{fused} eingefügt wird. Sie kann keinem Teilnehmer-Scope in p_μ zugeordnet werden, da sie das Verhalten von zwei unterschiedlichen Teilnehmern emuliert. Außerdem haben die Aktivitäten in w_{fused} verschiedene Elternaktivitäten, zu denen sie jeweils eine direkte Zustandsabhängigkeit besitzen. Angenommen in dem Beispiel wird w_{fused} dem Teilnehmer-Scope von s_{snd} zugeordnet, der das Verhalten von $\mathfrak{p}_{snd}$ emuliert. Müssen aufgrund eines Fehler auch die Aktivitäten terminiert werden, die aus der Schleife w_{sndCp} stammen, muss dazu w_{fused} terminiert werden. Dies hätte zur Folge, dass neben diesen Aktivitäten auch die aus w_{rcvCp} terminiert werden würden. Das widerspricht dem in der Choreographie modellierten Verhalten, dass die Terminierung von w_{snd} nicht die Terminierung von Aktivitäten aus w_{rcv} zur Folge hat. Die Fehlerisolation von w_{snd} und w_{rcv} ist für deren Duplikate in p_μ also nicht gegeben. Zusätzlich besteht in diesem

Fall das Problem, dass ein Fehler in der Elternaktivität der Aktivitäten aus w_{rcvCp} nicht von dieser bzw. vom Teilnehmer-Scope s_{rcv} an die Aktivitäten in w_{fused} propagiert werden kann.

Alternativ kann die Schleife w_{fused} auch außerhalb der Teilnehmer-Scopes platziert werden. Allerdings müssten dann Terminierungsereignisse von der jeweiligen Elternaktivität an w_{fused} propagiert werden. Dies könnte zum Beispiel über einen Fault- und Termination-Handler geschehen, der wiederum eine Throw-Aktivität auslöst, die zur Terminierung von w_{fused} führt. Aber auch dies würde zur Terminierung von allen Aktivitäten innerhalb w_{fused} führen und nicht nur zu denen, die dem Teilnehmer zugeordnet sind.

Für den Fall, dass innerhalb von w_{fused} ein Fehler auftritt, muss dieser zur korrekten Elternaktivität propagiert werden. Um das zu erreichen, muss w_{fused} ebenfalls einen Fehler auslösen und somit abgebrochen werden. Damit würden auch in diesem Fall die Aktivitäten des von dem Fehler nicht betroffenen Teilnehmers terminiert werden. Die Fehlerisolation zwischen den Aktivitäten, die aus verschiedenen Teilnehmern stammen, wäre also in p_μ nicht mehr gegeben. In der Choreographie in Abbildung 6.19 kann zum Beispiel die Schleife w_{snd} weiter iteriert werden, auch wenn aufgrund eines Fehlers im Schleifenkörper von w_{rcv} die Ausführung von w_{rcv} abgebrochen wurde. Dies ist in w_{fused} nicht mehr möglich, wenn der Fehler weiter propagiert werden muss.

Die Fusionierung von Schleifen ist daher nur bedingt dazu geeignet, die grenzverletzenden Kontrollflusskanten zwischen zwei interagierenden Schleifen in p_μ aufzulösen. Ihre Eignung hängt grundsätzlich davon ab, ob es tolerierbar ist, dass im Fehlerfall die komplette Schleife, d.h. auch die Logik des nicht von dem Fehler direkt betroffenen Teilnehmers terminiert wird.

Ist dies nicht tolerierbar, kann die Choreographie nicht konsolidiert werden, da die durch Materialisierung entstandenen grenzverletzenden Kontrollflusskanten nicht eliminiert werden können. Falls die Nachrichtenübertragung zwischen den Schleifen nicht materialisiert werden würde, um die Erstellung dieser Kontrollflusskanten zu vermeiden, müsste p_μ Nachrichten an

sich selbst schicken, was laut der in Kapitel 5 formulierten Anforderung (ii) ebenfalls nicht erlaubt ist.

6.7. Zusammenfassung

In diesem Kapitel wurden verschiedene Modellierungsszenarien für interagierende Schleifen vorgestellt und wie deren Materialisierung zu grenzverletzenden Kontrollflusskanten und damit zu einem ungültigen Prozessmodell p_μ führt. Für alle Szenarien bis auf eins können die grenzverletzenden Kontrollflusskanten durch das Ausrollen der Schleifen eliminiert werden. Bei diesem Ansatz wird ein Duplikat des Schleifenkörpers der aufzurollenden Schleife für jeden potentiellen Teilnehmer, mit dem die Schleifen interagieren kann, in p_μ erstellt. Diese Duplikate werden zur Laufzeit abhängig davon ausgeführt oder übersprungen, ob sich der Teilnehmer in der Menge der Teilnehmer befindet, mit denen eine Instanz von p_μ tatsächlich kommunizieren soll. Das Ausrollen einer Schleife, die potentiell mit sehr vielen Teilnehmern interagiert, kann folglich die Komplexität von p_μ stark erhöhen, da der Schleifenkörper entsprechend häufig dupliziert wird.

Es wurde hier nur das Ausrollen von sequentiellen Schleifen diskutiert, die nacheinander während jeder Iteration mit einem anderen Teilnehmer interagieren. Der Ansatz lässt sich aber leicht auf parallele Schleifen übertragen, bei denen die Schleifenkörper gleichzeitig ausgeführt werden und die Schleife somit auch gleichzeitig mit mehreren Teilnehmern interagiert. Dazu müssen die ausgerollten Körper einer Schleife anstatt als sequentielle als zueinander parallele Aktivitäten in p_μ eingefügt werden. Aus Platzgründen wurde in diesem Kapitel allerdings darauf verzichtet, das parallele Ausrollen detailliert zu erläutern.

Bei der Diskussion der Verhaltensäquivalenz zwischen der originalen und der ausgerollten Schleife hat sich herausgestellt, dass nicht alle Zustandsbeziehungen zwischen den Geschäftsaktivitäten erhalten werden können. Auch hier kommt es, wie nach der Materialisierung von Basisaktivitäten in

p_μ, zur verzögerten Ausführung von Geschäftsaktivitäten verglichen mit der originalen Choreographie.

Für zwei interagierende While-Schleifen kann deren potentielle Anzahl an Iterationen nicht bestimmt werden, da diese auch in mehreren Iterationen mit dem gleichen Teilnehmer kommunizieren können. Kontrollflussverletzungen, die durch die Materialisierung von zwei interagierenden While-Schleifen entstehen, lassen sich daher nicht durch das Ausrollen beheben. Deshalb wurde als Alternative zum Ausrollen ein Ansatz zur Fusion von zwei While-Schleifen in eine einzelne Schleife vorgestellt. Der Ansatz hat allerdings den Nachteil, dass durch die Fusion die Aktivitäten in den Körpern der fusionierten Schleife nicht mehr fehlerisoliert voneinander sind. In zukünftigen Arbeiten können die in Abschnitt 6.2 erwähnten Datenanalysetechniken genutzt werden, um für bestimmte While-Schleifen deren maximale Anzahl an Iterationen zu bestimmen. Diese könnten dann anlog zu den ForEach-Schleifen ausgerollt werden.

Mit den in diesem Kapitel vorgestellten Ansätzen zur Auflösung von grenzverletzenden Kontrollflusskanten kann die Interaktion von mehreren Teilnehmern in p_μ emuliert werden. Damit können auch die komplexen Interaktionen einer Choreographie in p_μ abgebildet werden, auf die im nächsten Kapitel eingegangen wird.

Kapitel 7

Validierung der Konsolidierung mittels Interaktionsmuster

Barros, Dumas und ter Hofstede [BDtH05] haben Interaktionen zwischen Diensten bzw. Prozessen untersucht, die in wissenschaftlichen Arbeiten oder in Anwendungsfällen aus der Industrie dokumentiert wurden, und daraus Interaktionsmustern destilliert. Diese Muster kategorisieren Interaktionen danach, wie viele Teilnehmer involviert sind, nach der Anzahl an Nachrichten, die diese austauschen, in welcher Reihenfolge der Nachrichtenaustausch stattfindet und ob Nachrichten von Teilnehmern an andere Teilnehmer weitergeleitet werden. Die Muster sind abstrakt formuliert und dienen als Benchmark für Choreographiesprachen, die dahingehend evaluiert werden können, inwieweit sie die Muster unterstützen. Eine Umsetzung der Muster mit BPEL4Chor wurde von Kopp [Kop16a, S. 100–139] erläutert.

In diesem Kapitel wird aufbauend auf der Arbeit von Kopp die Modellierung der Muster mit dem Metamodell aus Kapitel 3 beschrieben. Um die Zweckmäßigkeit des in den vorherigen Kapiteln vorgestellten Konsolidierungsansatzes zu validieren, wird diskutiert, inwieweit die Muster mit den dort beschrieben Schritten konsolidiert werden können. Der Aufbau dieses Kapitels orientiert sich dabei an der von Barros, Dumas und ter Hofstede vorgenommenen Kategorisierung der Interaktionsmuster.

7.1. Single-transmission Bilateral Interaction Patterns

In diese Kategorie fallen alle Muster, in denen zwei Teilnehmer involviert sind, zwischen denen genau eine Interaktion stattfindet. Dies sind die Muster „Send“, „Receive“ und „Send/Receive“ [BDtH05]. Das Muster *„Send“* wird direkt durch die Invoke-Aktivität unterstützt. Das Muster *„Receive“* kann durch die Receive-Aktivität und die Nachrichtenereignisse der Pick-Aktivität modelliert werden.

Beim Muster *„Send/Receive“* sendet ein Teilnehmer eine Nachricht an einen anderen Teilnehmer. Diese Interaktionen können direkt, wie in Abschnitt 5.4.1 beschrieben, durch Invoke- und Receive-Basisinteraktionen modelliert werden. Anschaulich ist dieses Muster in Abbildung 5.4 dargestellt. In Abschnitt 5.4.1 wurde die Materialisierung dieser Interaktionen ausführlich beschrieben.

7.2. Single-transmission Multilateral Interaction Patterns

Diese Kategorie fasst die Muster zusammen, in denen ein Teilnehmer mit einer Menge von anderen Teilnehmern interagiert, wobei mit jedem Teilnehmer genau ein Nachrichtenaustausch stattfindet. Dies umfasst die Muster „Racing Incoming Messages“ , „One-to-many Send“, „One-from-many Receive“ und „One-to-many Send/Receive“.

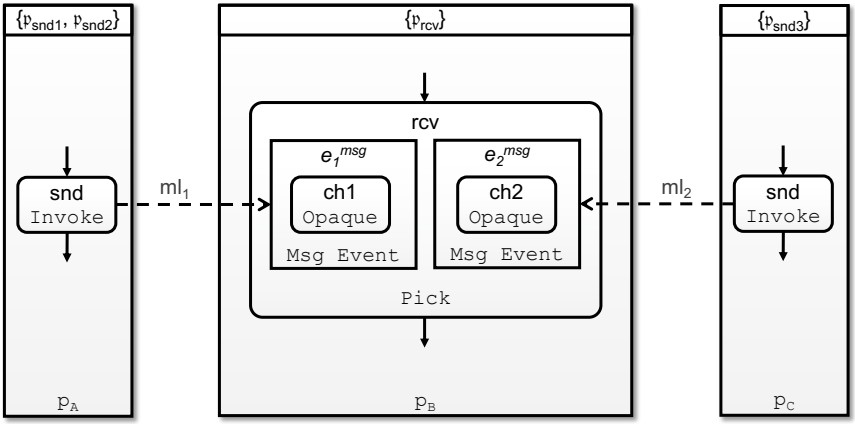

Abbildung 7.1.: Interaktionsmuster „Racing Incoming Messages“

7.2.1. Racing Incoming Messages

Im Falle des Musters *„Racing Incoming Messages“* empfängt ein Teilnehmer genau eine Nachricht aus einer Menge von potentiellen Nachrichten, die von anderen Teilnehmern gleichen oder unterschiedlichen Teilnehmertyps gesendet werden können. Dieses Interaktionsmuster kann durch eine Pick-Aktivität modelliert werden, die eine Menge von Nachrichtenereignissen enthält. Jedes Nachrichtenereignis ist dabei über eine Nachrichtenkante mit einem Invoke verbunden. Für jede Nachrichtenkante können ein oder mehrere potentielle Sender und genau ein Empfänger spezifiziert werden. In Abbildung 7.1 sind zum Beispiel die Teilnehmer $\mathfrak{p}_{snd1}$ und $\mathfrak{p}_{snd2}$ dem Ereignis e_1^{msg} und $\mathfrak{p}_{snd3}$ dem Ereignis e_3^{msg} über die Nachrichtenkanten ml_1 bzw. ml_2 zugeordnet. Sendet einer der Teilnehmer aus den Teilnehmermengen eine Nachricht, wird das entsprechende Ereignis aktiviert und damit die zugehörige Kindaktivität des Pick ausgeführt.

Die Konsolidierung erstellt für jeden an der Interaktion beteiligten Teilnehmer einen Teilnehmer-Scope in p_μ. Da das Muster durch eine Menge von Basisinteraktionen zwischen Invoke-Aktivitäten und Nachrichtenereignissen

modelliert wird, wird der in Abschnitt 5.5.3 beschriebene Algorithmus zu deren Materialisierung angewandt. Dieser ersetzt im Teilnehmer-Scope des Empfängers das Pick durch den Scope s_{pick}, der das Verhalten des Pick über die Fault-Handler emuliert. In den Teilnehmer-Scopes, die die sendenden Teilnehmer repräsentieren, ersetzt der Algorithmus das Invoke durch eine Assign-Aktivität. Im Scope s_{pick} wird für jeden sendenden Teilnehmer eine Throw-Aktivität erstellt, die mit dem Assign im entsprechenden Teilnehmer-Scope über eine Kontrollflusskante verbunden wird. Das Throw löst, sobald das Assign in einem der Teilnehmer-Scopes ausgeführt wurde und den Nachrichtentransfer emuliert hat, über ein Fehlerereignis die Ausführung der jeweiligen Kindaktivität aus. Ein Fault-Handler ist dazu über ein Fehlerereignis mit einem oder mehreren Throws assoziiert, deren Ausführung den Fault-Handler aktiviert. In Abbildung 7.2 ist die konsolidierte Interaktion aus Abbildung 7.1 dargestellt. Die Throws $syn_{rcvSnd1}$ und $syn_{rcvSnd2}$ sind dort zum Beispiel mit dem Fault-Handler $fh1^{msg}$ assoziiert. Wird das Senden einer Nachricht über die Synchronisationsaktivitäten von Teilnehmer $\mathfrak{p}_{snd2}$ emuliert, wird zuerst syn_{snd2} ausgeführt und dadurch kommt durch das von $syn_{rcvSnd1}$ ausgelöste Fehlerereignis, wie im Pick, die Aktivität $ch1_{cp}$ zur Ausführung. Die Eigenschaft, dass das Pick nur eine der potentiellen Nachrichten empfangen kann, wird dadurch emuliert, dass zur Laufzeit eines Scopes nur einer seiner Fault-Handler aktiviert werden kann. Eine detaillierte Diskussion der Zustandsbeziehungen zwischen den Geschäftsaktivitäten des Pick fand bereits in Abschnitt 5.5.4 statt.

7.2.2. One-to-many Send

Beim Muster *„One-to-many Send“* sendet ein Teilnehmer eine Menge von Nachrichten an unterschiedliche Empfänger. Dieses Muster wird durch eine Nachrichtenkante modelliert, die ein Invoke und ein Receive miteinander verbindet. Für die Nachrichtenkante muss ein sendender Teilnehmer $\mathfrak{p}_{snd}$ und die Teilnehmermenge $\mathfrak{p}_{rcv}^{set}$ der Empfänger spezifiziert werden. Das Invoke befindet sich in einem ForEach, das über die Teilnehmermenge $\mathfrak{p}_{rcv}^{set}$ der

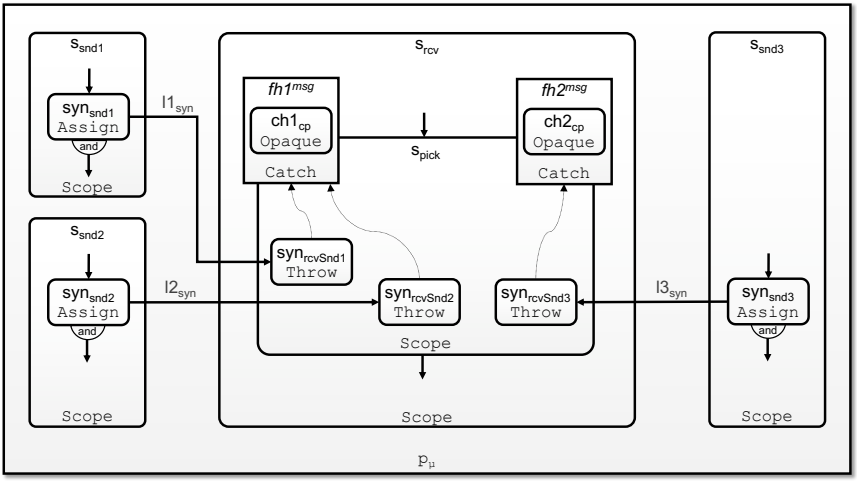

Abbildung 7.2.: Konsolidiertes Interaktionsmuster „Racing Incoming Messages“

Empfänger iteriert und so während jeder Iteration eine Nachricht an einen anderen Teilnehmer aus $\mathfrak{p}_{rcv}^{set}$ sendet. Dabei können die Empfänger in $\mathfrak{p}_{rcv}^{set}$ zur Laufzeit vor der Ausführung des ForEach bestimmt werden. In Abbildung 7.3 ist eine „One-to-many Send“ Interaktion dargestellt, in der Teilnehmer $\mathfrak{p}_{snd}$ eine Nachricht an die Teilnehmer $\mathfrak{p}_{rcv1}$ und $\mathfrak{p}_{rcv2}$ sendet.

Die Konsolidierung erstellt in p_μ Teilnehmer-Scopes für den sendenden und für die empfangenden Teilnehmer. Im nächsten Schritt wird die der „One-to-many Send“-Interaktion zugrunde liegende Invoke-Receive-Interaktion materialisiert. Dadurch entstehen grenzverletzende Kontrollflusskanten, da sich das während der Materialisierung erstellte Assign syn_{snd} innerhalb und das Empty syn_{rcv} außerhalb der ForEach-Schleife befindet. Um diese Verletzungen aufzulösen, werden die Teilnehmeriterationen der ForEach-Schleife mit dem Ansatz aus Abschnitt 6.2 ausgerollt. Das Resultat der Konsolidierung der „One-to-many Send“-Interaktion aus Abbildung 7.3 ist in Abbildung 7.4 dargestellt. Da die Schleife *fe* mit den Teilnehmern $\mathfrak{p}_{rcv1}$ und $\mathfrak{p}_{rcv2}$ kommunizieren kann, wurde der Schleifenkörper zweimal ausgerollt.

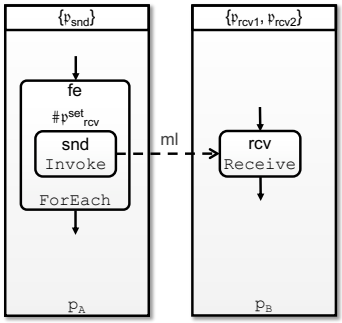

Abbildung 7.3.: Interaktionsmuster „One-to-many Send“

Die Voraussetzung dafür, dass die Schleife ausgerollt werden kann, ist, dass die potentiellen Teilnehmer, mit denen über die Schleife kommuniziert werden kann, zur Entwurfszeit bekannt sein müssen. Dies ist eine Einschränkung zur originalen Interaktion, wo die Empfänger zur Laufzeit bestimmt werden können. Durch die Einschränkung muss zwar die Menge der potentiellen Teilnehmer bekannt sein, allerdings muss die ausgerollte Schleife zur Laufzeit nicht die Kommunikation mit allen diesen Teilnehmern emulieren. So wird die Kommunikation mit Teilnehmer $\mathfrak{p}_{rcv1}$ nur emuliert, wenn sich dieser zur Laufzeit von p_μ auch in der Menge der Teilnehmer befindet, die dem ForEach über die Abbildung partSet (siehe Abschnitt 3.2.4.10) zugewiesen wurde.

7.2.3. One-from-many Receive

Beim Muster *„One-from-many Receive“* empfängt ein Teilnehmer $\mathfrak{p}_{rcv}$ so lange eine Menge von Nachrichten von unterschiedlichen sendenden Teilnehmern, bis er ausreichend Informationen erhalten hat, um die Kommunikation abzubrechen. Dieses Muster wird ebenfalls durch eine Nachrichtenkante zwischen einem Invoke und einem Receive modelliert, für die die sendenden Teilnehmer über eine Teilnehmermenge $\mathfrak{p}^{set}_{snd}$ und der empfangender Teilnehmer $\mathfrak{p}_{rcv}$ spezifiziert werden muss. Kopp [Kop16a] schlägt die Modellierung die-

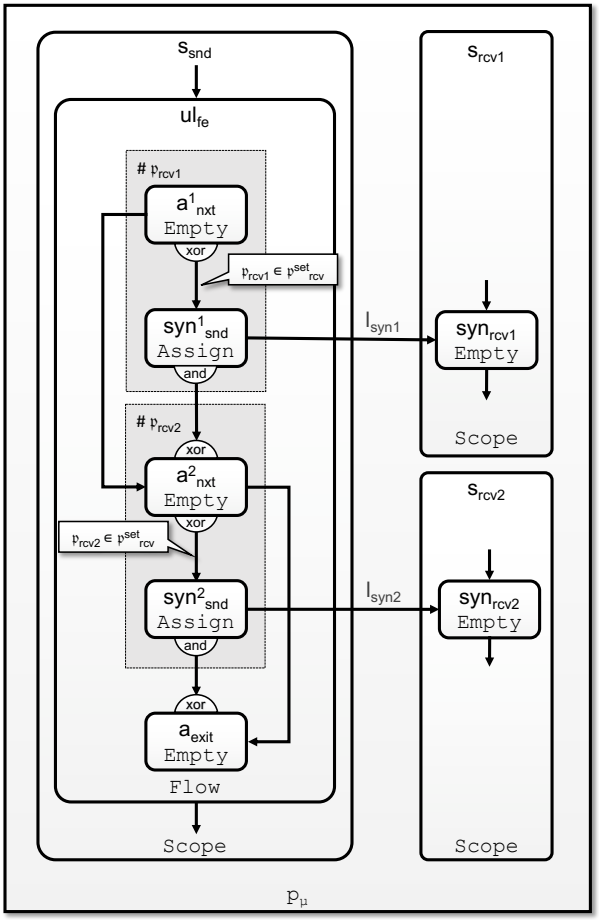

Abbildung 7.4.: Konsolidiertes Interaktionsmuster „One-to-many Send“

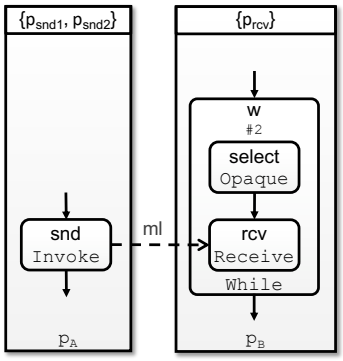

Abbildung 7.5.: Interaktionsmuster „One-from-many Receive“

ses Musters mittels einer While-Schleife in der Verhaltensbeschreibung des Empfängers vor, da so auch während der Ausführung der Schleife weitere sendende Teilnehmer zur Teilnehmermenge $\mathfrak{p}_{snd}^{set}$ hinzukommen können. Bei einer ForEach-Schleife müssten die Sender hingegen schon vor ihrer Ausführung bekannt sein. Ein Beispiel für diese Interaktion findet sich in Abbildung 7.5. Über die Schleife *w* kann der Teilnehmer $\mathfrak{p}_{rcv}$ sukzessive die Nachrichten von den Teilnehmern aus der Teilnehmermenge $\mathfrak{p}_{snd}^{set}$ (hier $\mathfrak{p}_{snd}^{set} = \{\mathfrak{p}_{snd1}, \mathfrak{p}_{snd2}\}$) empfangen. Dabei bestimmt die Aktivität *select* den Teilnehmer, von dem in einer Iteration der Schleife eine Nachricht empfangen werden soll.

Analog zur Konsolidierung des Interaktionsmuster „One-to-many Send“ wird zuerst die zugrunde liegende Invoke-Receive-Interaktion materialisiert. Daraus resultieren Assign-Aktivitäten in den Teilnehmer-Scopes der Sender, die mit einer Empty-Aktivität in der While-Schleife im Teilnehmer-Scope des Empfängers über Kontrollflusskanten verbunden sind. Diese grenzverletzenden Kontrollflusskanten werden ebenfalls durch den in Abschnitt 6.3 beschriebenen Ansatz zum Ausrollen von While-Schleifen aufgelöst. Das Ergebnis der Konsolidierung der Interaktion aus Abbildung 7.5 ist in Abbildung 7.6 dargestellt. Da es sich um eine While-Schleife handelt, die

ausgerollt wurde, wird für jede ausgerollte Teilnehmeriteration anhand der Schleifenbedingung geprüft, ob sie ausgeführt werden kann.

Auch hier muss die Anzahl der potentiellen Teilnehmer, mit denen über die While-Schleife kommuniziert werden kann (d.h. die maximale Iterationsanzahl), bekannt sein, um die Schleife auszurollen. Dies ist ebenfalls eine Einschränkung zur originalen Interaktion, wo die Teilnehmer zur Ausführungszeit der Schleife bestimmt werden können.

7.2.4. One-to-many Send/Receive

Beim Interaktionsmuster *„One-to-many Send/Receive“* sendet ein Teilnehmer eine Anfragenachricht an eine Menge von unterschiedlichen Empfängern und wartet dann für eine bestimmte Zeitspanne auf deren Antwortnachrichten. Das Muster wird durch zwei Nachrichtenkanten ml_{req} und ml_{res} modelliert, die jeweils ein Invoke und ein Receive verbinden. Die Kante ml_{req} bildet das Senden der Anfragenachricht ab. Für sie muss der sendende Teilnehmer $\mathfrak{p}_{snd}$ und die Teilnehmermenge $\mathfrak{p}_{rcv}^{set}$ der Empfänger spezifiziert werden. Die Kante ml_{res} bildet wiederum das Senden der Antwortnachrichten ab. Daher muss für sie als Sender die Teilnehmermenge $\mathfrak{p}_{rcv}^{set}$ und als Empfänger der Teilnehmer $\mathfrak{p}_{snd}$ spezifiziert werden. Die Verhaltensbeschreibung von $\mathfrak{p}_{snd}$ enthält eine ForEach-Schleife fe_{snd}, in der sich das Invoke der Kante ml_{req} befindet. Damit kann die Anfragenachricht wie beim Muster „One-to-many Send“ an die verschiedenen Empfänger gesendet werden. Um die Nachrichten nur innerhalb der spezifizierten Zeitspanne zu erhalten, enthält die Verhaltensbeschreibung zusätzlich einen Scope s_{fe} mit einer weiteren ForEach-Schleife[1] fe_{rcv} als Kindaktivität und einem Event-Handler der einer Throw-Aktivität zugeordnet ist. Der Empfang der Antwortnachrichten wird dabei über das Receive der Nachrichtenkante ml_{res} innerhalb des ForEach realisiert. Im Timer-Ereignis des Event-Handlers wird die Zeitspanne festgelegt, nach der

[1] Hier wird ein ForEach anstatt einer While-Schleife genutzt, da als Sender der Antwortnachrichten nur die Teilnehmer aus $\mathfrak{p}_{rcv}^{set}$ infrage kommen, die vor der Ausführung von fe_{rcv} bereits bekannt sind.

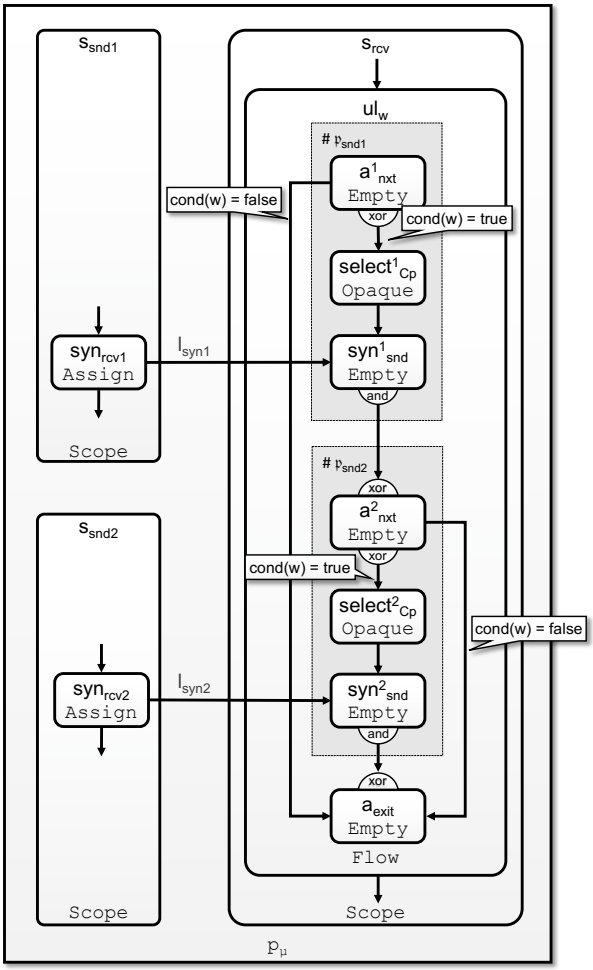

Abbildung 7.6.: Konsolidiertes Interaktionsmuster „One-from-many Receive“

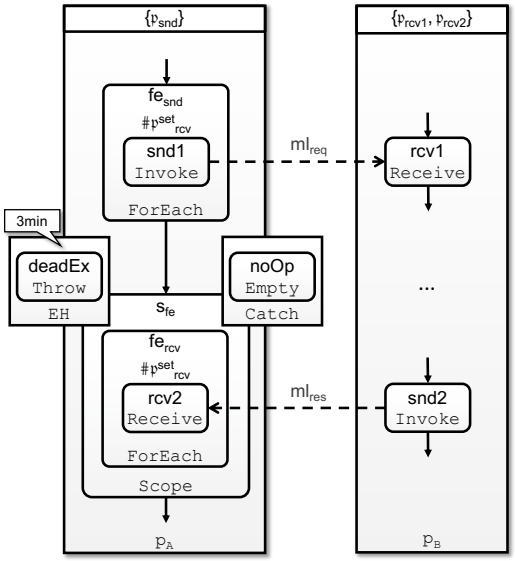

Abbildung 7.7.: Interaktionsmuster „One-to-many Send/Receive“

das Throw ausgeführt und damit die Ausführung des Scopes und des Receive über einen Fehler beendet wird. Eine „One-to-many Send/Receive“ Interaktion in der ein Teilnehmer die Anfragenachricht an zwei andere Teilnehmer sendet und die Antwortnachricht von diesen Teilnehmern innerhalb von drei Minuten erwartet, ist in Abbildung 7.7 dargestellt.

Bei der Konsolidierung dieses Interaktionsmusters entstehen durch die Materialisierung wie bei „One-to-many Send“ bzw. „One-from-many Receive“ Interaktionen ebenfalls grenzverletzende Kontrollflusskanten im Teilnehmer-Scope von $\mathfrak{p}_{snd}$, da Kontrollflusskanten aus dem sendenden ForEach heraus bzw. in das empfangende ForEach hinein zeigen. Daher werden beide Schleifen im Teilnehmer-Scope von $\mathfrak{p}_{snd}$ ausgerollt. In Abbildung 7.8 ist die konsolidierte Interaktion aus Abbildung 7.7 dargestellt. Die Flows $ul_{fe_{snd}}$ bzw. $ul_{fe_{rcv}}$

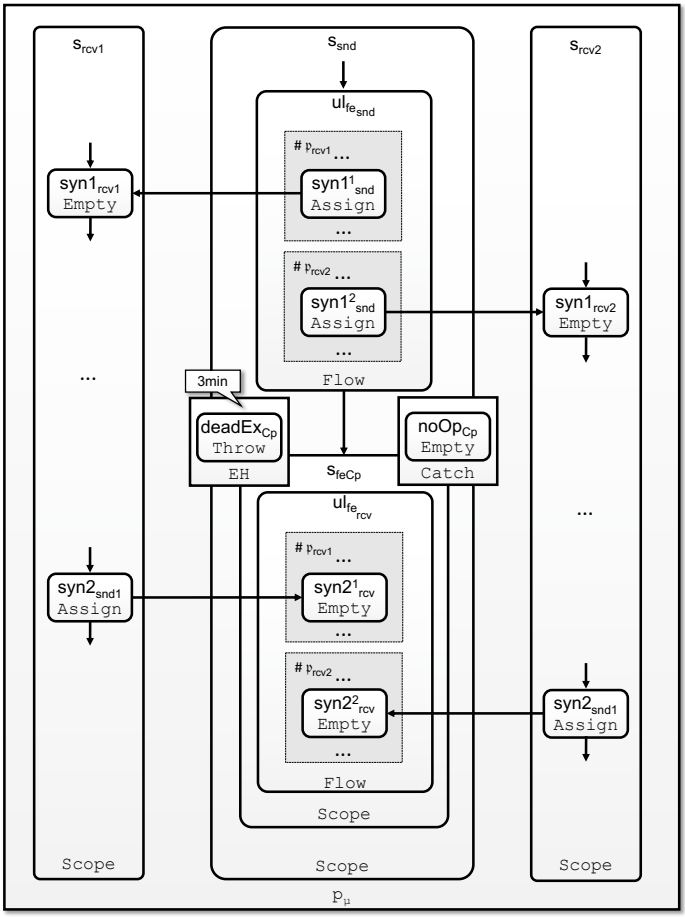

Abbildung 7.8.: Konsolidierte „One-to-many Send/Receive“ Interaktion

repräsentieren die beiden ausgerollten ForEach-Schleifen[1]. Der Flow $ul_{fe_{rcv}}$ befindet sich im Duplikat des Scopes s_{fe}. Der Empfang der Antwortnachricht wird also weiterhin nur für die angegebene Zeitspanne emuliert.

[1]Der Kontrollfluss zwischen den Teilnehmeriterationen wurde der Übersichtlichkeit wegen nicht dargestellt.

7.3. Multi-Transmission Interaction Patterns

Diese Kategorie umfasst die Muster „Multi-Responses“, „Contingent Request“ und „Atomic Multicast Notification“ [BDtH05], in denen ein Teilnehmer mit denselben Teilnehmern mehrere Nachrichten austauscht.

7.3.1. Multi-Responses

In *„Multi-Responses“* Interaktionen sendet ein Teilnehmer $\mathfrak{p}_{req}$ eine Anfragenachricht an einen anderen Teilnehmer $\mathfrak{p}_{res}$ und empfängt dann so lange Antwortnachrichten von diesem Teilnehmer, bis er ausreichend Informationen erhalten hat. Das Muster wird, wie im Beispiel Abbildung 7.9 dargestellt, durch zwei Nachrichtenkanten ml_{req} und ml_{res} modelliert, die jeweils ein Invoke mit einem Receive verbinden. Die Kante ml_{req} bildet das Senden der Anfragenachricht und die Kante ml_{req} das Senden der Antwortnachricht ab. Folglich wird für ml_{req} der Teilnehmer $\mathfrak{p}_{req}$ als sendender und $\mathfrak{p}_{res}$ als empfangender Teilnehmer spezifiziert bzw. für ml_{res} der Teilnehmer $\mathfrak{p}_{res}$ als sendender und $\mathfrak{p}_{req}$ als empfangender Teilnehmer. Um das Senden und Empfangen einer beliebigen Anzahl von Antwortnachrichten umzusetzen, befinden sich die Invoke und Receive-Aktivität von ml_{res} jeweils in einer While-Schleife. Die Nutzung von ForEach-Schleifen ist hier nicht möglich, da diese während jeder Iteration mit einem anderen Teilnehmer kommunizieren müssen.

Die Konsolidierung materialisiert die Interaktionen, die mittels der beiden Nachrichtenkanten ml_{req} bzw. ml_{res} modelliert sind. Die Materialisierung von ml_{res} erzeugt eine grenzverletzende Kontrollflusskante, deren Quellaktivität sich in der While-Schleife im Teilnehmer-Scope von $\mathfrak{p}_{req}$ und deren Zielaktivität sich in der While-Schleife von Teilnehmer-Scope $\mathfrak{p}_{res}$ befindet. Da die Anzahl der Iterationen für zwei kommunizierende While-Schleifen nicht bestimmt werden kann, werden sie, wie in Abschnitt 6.6 erläutert, in ein einzelnes While fusioniert. Anschaulich ist die Fusion der Schleifen aus Abbildung 7.9 in Abbildung 7.10 dargestellt.

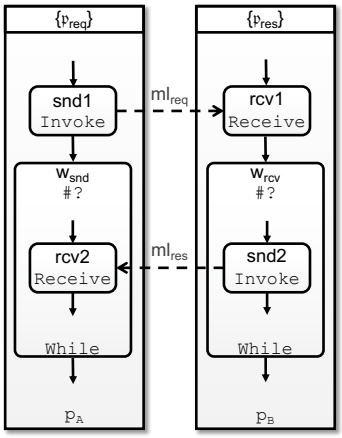

Abbildung 7.9.: Interaktionsmuster „Multi-Responses“

Verglichen mit der originalen Interaktion ergibt sich durch die Fusionierung die Einschränkung der Iterationsabhängigkeit zwischen den Aktivitäten in dem fusionierten Schleifenkörper. Außerdem ist in der fusionierten Schleife die Fehlerisolation zwischen den Aktivitäten, die aus den zwei While-Schleifen stammen, nicht mehr gegeben. Folglich lässt sich dieses Muster nicht komplett verhaltensäquivalent zur Choreographie konsolidieren.

7.3.2. Contingent Request

Beim Interaktionsmuster *„Contingent Request“* sendet ein Teilnehmer $\mathfrak{p}_{req}$ eine Anfragenachricht an einen Teilnehmer. Erhält $\mathfrak{p}_{req}$ von diesem innerhalb einer bestimmten Zeitpanne keine Antwortnachricht, sendet er die Nachricht so lange an weitere Teilnehmer, bis er von einem dieser Teilnehmer eine Antwortnachricht erhält. Das Senden der Anfragenachricht wird über die Nachrichtenkante ml_{req} und das Senden der Antwortnachricht über die Nachrichtenkante ml_{res} modelliert. Die Kante ml_{req} verbindet ein Invoke mit einem Receive und die Kante ml_{res} ein Invoke mit einem Nachrichtenereignis. Für die Kante ml_{req} wird Teilnehmer $\mathfrak{p}_{req}$ als Sender und als Empfänger

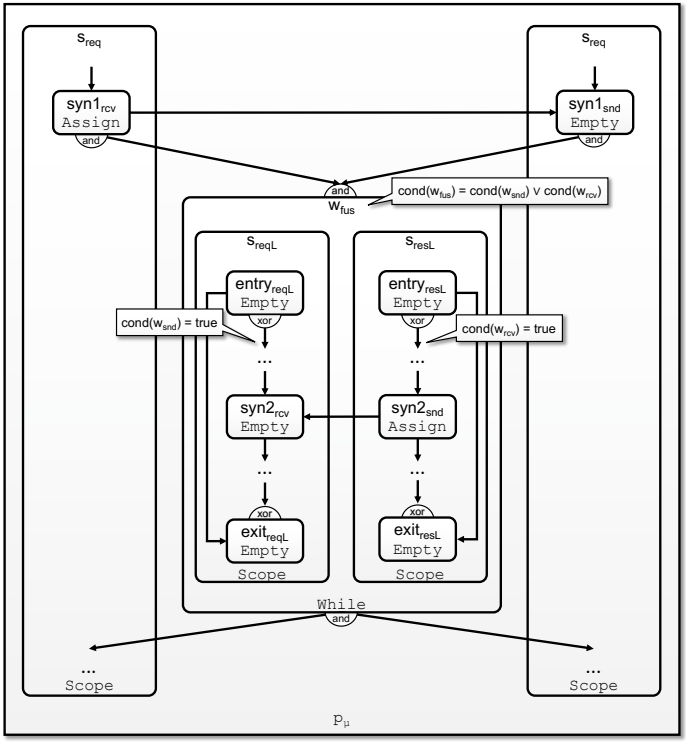

Abbildung 7.10.: Konsolidiertes Interaktionsmuster „Multi-Responses“

die Teilnehmermenge $\mathfrak{p}_{alt}^{set}$ spezifiziert, die die alternativen Empfänger der Nachricht umfasst. Für die Kante ml_{res} werden wiederum $\mathfrak{p}_{alt}^{set}$ als Sender und $\mathfrak{p}_{req}$ als Empfänger spezifiziert. Um die Anfragenachricht sukzessive an die alternativen Empfänger zu senden, befindet sich das Invoke wie beim Muster „One-to-many Send“ in einer ForEach-Aktivität. In diesem ForEach befindet sich außerdem eine Pick-Aktivität, die wiederum eine Throw- und eine Empty-Aktivität enthält. Das Throw ist dem Nachrichtenereignis der Kante ml_{res} und das Empty einem Timer-Ereignis zugeordnet, in dem die Zeitspanne für den Empfang der Antwortnachricht spezifiziert ist. Wurde die Zeitspanne überschritten, wird das Empty ausgeführt, die Ausführung des

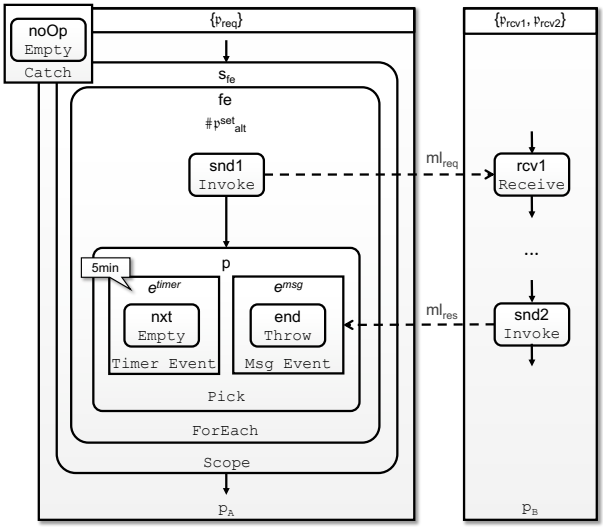

Abbildung 7.11.: Interaktionsmuster „Contingent Request“

Pick beendet und das ForEach geht in die nächste Iteration. Beim Empfang einer Antwortnachricht in der vorgegebenen Zeitspanne wird das Throw ausgeführt und beendet die Ausführung des ForEach. Um sicherzustellen, dass der Fehler nicht weiter propagiert wird, hat der Scope einen Fault-Handler, der den durch das Throw ausgelösten Fehler fängt. Abbildung 7.11 zeigt eine „Contingent Request“ Interaktion, in der die Anfragenachricht an zwei alternative Teilnehmer $\mathfrak{p}_{rcv1}$ und $\mathfrak{p}_{rcv2}$ (Teilnehmermenge $\mathfrak{p}_{alt}^{set}$) gesendet werden kann. Die Zeitspanne, in der diese Teilnehmer eine Antwortnachricht senden können, beträgt 5 Minuten.

Die Konsolidierung materialisiert die Basisinteraktion zwischen dem Invoke und dem Receive sowie die Basisinteraktion zwischen dem Invoke und dem Nachrichtenereignis. Dadurch wird, wie bei der Materialisierung der Interaktion „Racing Incoming Messages“, das Pick durch einen Scope s_{rcv} ersetzt, der das Verhalten des Pick emuliert. Der Scope besitzt zwei Fault-Handler. Einer wird aktiviert, falls die Zeitspanne für den Nachrichtenempfang von einem

Teilnehmer abgelaufen ist. Dazu enthält der Scope ein Wait, das über eine Kontrollflusskante mit einem Throw verbunden ist, das die Ausführung dieses Fault-Handlers anstößt (siehe Abschnitt 5.5). Der andere Fault-Handler wird aktiviert, wenn der Empfang der Antwortnachricht von einem der Teilnehmer emuliert wird. Da sich beim Interaktionsmuster „Contingent Request“ das Invoke, das die Anfragenachricht sendet, und das Pick, das die Antwortnachricht empfängt, in einer ForEach-Schleife befinden, kommt es auch hier durch die Materialisierung zu grenzverletzenden Kontrollflusskanten im Teilnehmer-Scope von $\mathfrak{p}_{req}$. Diese werden wieder durch das Ausrollen des ForEach aufgelöst. In Abbildung 7.12 ist die konsolidierte „Contingent Request“ Interaktion aus Abbildung 7.11 dargestellt. In jeder Teilnehmeriteration des Scopes s_{rcv} wird durch den Algorithmus aus Abschnitt 6.3.3 genau eine Throw-Aktivität mit dem Assign, das das Senden der Antwortnachricht des jeweiligen Teilnehmers emuliert, verbunden.

7.3.3. Atomic Multicast Notification

Beim Muster *„Atomic Multicast Notification“*, das auch als *„Transactional Notification“* bezeichnet wird, sendet ein Teilnehmer Nachrichten an mehrere andere Teilnehmer, die diese in einem bestimmten Zeitraum annehmen müssen. Die Annahme der Nachrichten soll dabei atomar geschehen, d.h. die Nachrichten müssen entweder von allen oder von keinem Empfänger angenommen werden. Da BPEL4Chor Transaktionalität bzw. Atomizität nicht unterstützt [Kop16a], kann dieses Muster mit dem Metamodell nicht umgesetzt werden.

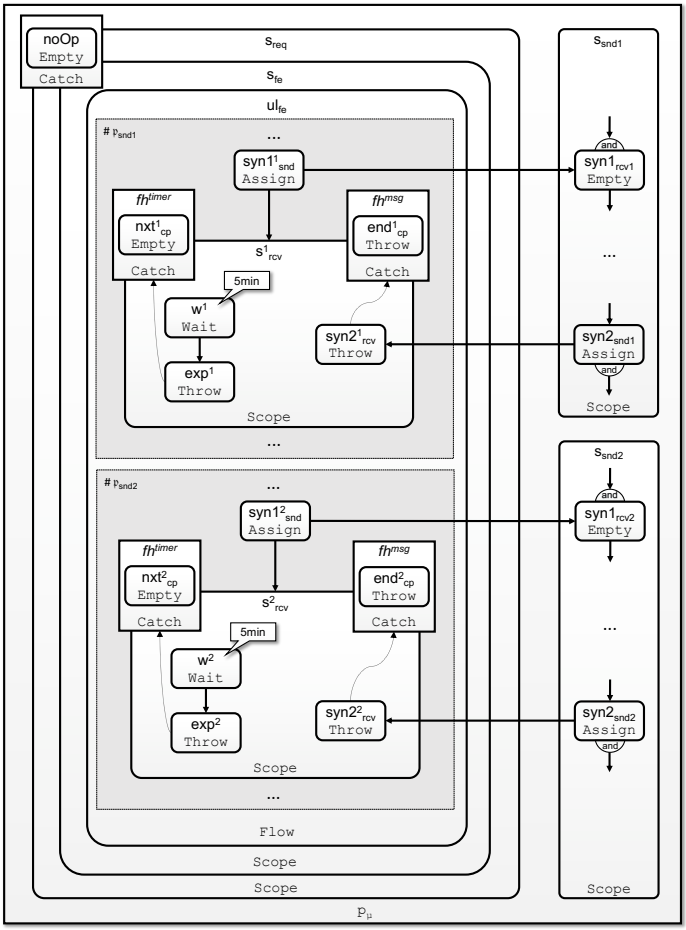

Abbildung 7.12.: Konsolidiertes Interaktionsmuster „Contingent Request“

7.4. Routing Patterns

Zu dieser Kategorie gehören die Interaktionsmuster, bei denen ein Teilnehmer eine Nachricht an einen Teilnehmer sendet, der diese wiederum an andere Teilnehmer weiterleitet. Konkret sind das die Muster „Request with Referral“, „Relayed Request“ und „Dynamic Routing“ [BDtH05].

7.4.1. Request with Referral

Beim Muster *„Request with Referral“* sendet ein Teilnehmer $\mathfrak{p}_{req}$ eine Anfragenachricht an einen Teilnehmer $\mathfrak{p}_{ref}$ mit dem Hinweis, dass dieser die Antwortnachricht nicht an $\mathfrak{p}_{req}$, sondern an eine Menge von anderen Teilnehmern $\{\mathfrak{p}_1, \ldots, \mathfrak{p}_n\}$ senden soll. Das Senden der Anfragenachricht wird analog zum Muster „Send/Receive“ über eine Nachrichtenkante ml_{req} modelliert, in der als Sender $\mathfrak{p}_{req}$ und als Empfänger $\mathfrak{p}_{ref}$ spezifiziert werden. Die Nachricht, die über diese Kante gesendet wird, muss u.a. die Menge der Empfänger der Antwortnachricht enthalten. Da in dem Metamodell u.a. Teilnehmermengen für Variablen und Nachrichten erlaubt sind (siehe Abschnitt 3.2.1), kann mittels der Teilnehmerreferenzübermittlung [DKLW09] die Übertragung der Empfänger realisiert werden.

Die Antwortnachrichten werden über die Nachrichtenkante ml_{res} vom Teilnehmer $\mathfrak{p}_{ref}$ an diese Empfänger gesendet. Da die Antwortnachricht an mehrere Empfänger gesendet werden soll, befindet sich das Invoke wie beim Muster „One-to-many Send“ in einem ForEach. Der Teilnehmermenge, über die das ForEach iteriert, müssen vor dessen Ausführung die Empfänger der Antwortnachricht zugewiesen werden. Abbildung 7.13 zeigt eine „Request with Referral“ Interaktion, in der die Antwortnachrichten an die Teilnehmer $\mathfrak{p}_{rcv1}$ und $\mathfrak{p}_{rcv2}$ gesendet werden können. Die Aktivität *rcv1* empfängt die Anfragenachricht von $\mathfrak{p}_{req}$, die die Teilnehmermenge der möglichen Empfänger der Antwortnachricht enthält und kopiert diese in die Variable v. Das Assign *setRcv* kopiert diese wiederum in die Teilnehmermenge $\mathfrak{p}^{set}_{rcv}$, über die das ForEach *fe* iteriert, um die Antwortnachrichten zu senden.

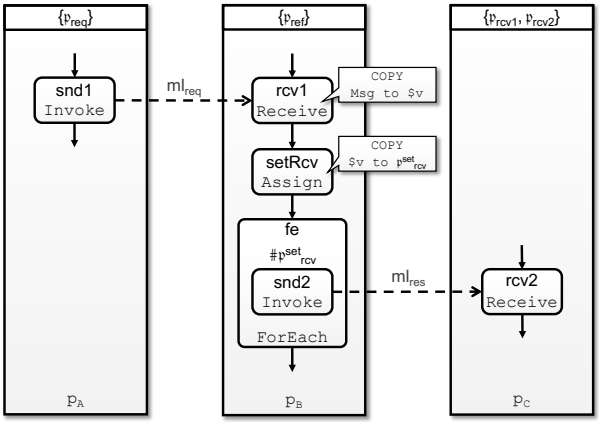

Abbildung 7.13.: Interaktionsmuster „Request with Referral“

Da es sich bei diesem Interaktionsmuster um eine Kombination der Muster „Send/Receive“ und „One-to-many Send“ handelt, kann es ebenfalls mit dem in dieser Arbeit vorgestellten Ansatz konsolidiert werden. Das Ergebnis der Konsolidierung ist in Abbildung 7.14 dargestellt. Das ForEach muss wieder ausgerollt werden. Die Aktivität $setRcv_{cp}$ in p_μ weist der Variablen v weiterhin den Inhalt der Teilnehmermenge $\mathfrak{p}^{set}_{rcv}$ zu. Damit führt die ausgerollte Schleife, wie das ForEach in der originalen Interaktion nur die Teilnehmeriterationen aus, deren Teilnehmer sich in $\mathfrak{p}^{set}_{rcv}$ befinden.

7.4.2. Relayed Request

Im Falle des Musters *„Relayed Request“* sendet der Teilnehmer $\mathfrak{p}_{req}$ eine Anfragenachricht an den Teilnehmer $\mathfrak{p}_{ref}$, der die Nachricht wiederum an eine Menge von Teilnehmern $\{\mathfrak{p}_1 \dots \mathfrak{p}_n\}$ weiterleitet, mit dem Hinweis, dass diese mit dem Teilnehmer $\mathfrak{p}_{req}$ kommunizieren sollen. Das heißt, dass die Antwortnachricht sowie alle weiteren Nachrichten direkt zwischen dem Teilnehmer $\mathfrak{p}_{req}$ und $\mathfrak{p}_1 - \mathfrak{p}_n$ ausgetauscht werden müssen. Teilnehmer $\mathfrak{p}_{ref}$ beobachtet dabei die gesamte Kommunikation zwischen diesen Teilnehmern.

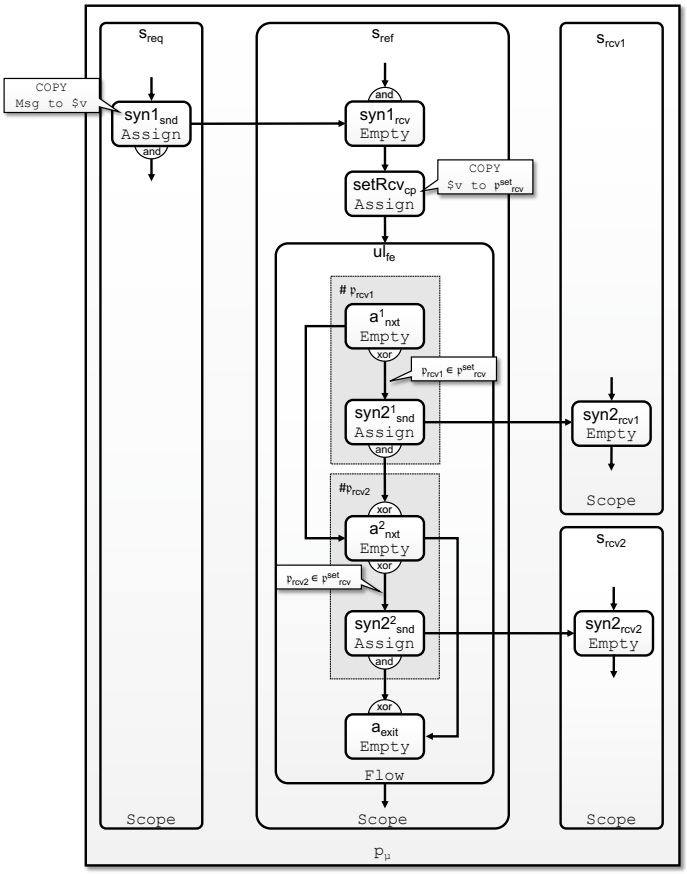

Abbildung 7.14.: Konsolidiertes Interaktionsmuster „Request with Referral“

Abbildung 7.15 zeigt ein Beispiel einer „Relayed Request“ Interaktion, in der der Teilnehmer $\mathfrak{p}_{req}$ direkt mit den Teilnehmern $\mathfrak{p}_{rcv1}$ und $\mathfrak{p}_{rcv2}$ kommuniziert und Teilnehmer $\mathfrak{p}_{ref}$ diese Kommunikation beobachtet. Das Muster wird ähnlich zum Muster „Request with Referral“ umgesetzt. Das Senden der Anfragenachricht von Teilnehmer $\mathfrak{p}_{req}$ an $\mathfrak{p}_{ref}$ wird über eine „Send/Receive“ Interaktion modelliert. Im Gegensatz zum Muster „Request with Referral“ ist für dieses Muster nicht spezifiziert, ob die Teilnehmer, mit denen $\mathfrak{p}_{req}$ kommunizieren soll, in der Anfragenachricht übertragen werden oder ob dies bereits zur Entwurfszeit in den jeweiligen Teilnehmerbeschreibungen festgelegt wird. Die initiale Kommunikation zwischen Teilnehmer $\mathfrak{p}_{ref}$ und den Teilnehmern $\mathfrak{p}_1 - \mathfrak{p}_n$ wird durch eine „One-to-many Send“ Interaktion modelliert. Der Austausch der Antwortnachricht zwischen den Teilnehmern $\mathfrak{p}_1 - \mathfrak{p}_n$ und dem Teilnehmer $\mathfrak{p}_{req}$ wird über eine „One-from-many Receive“ Interaktion realisiert. Auf diese Interaktion können weitere multilateralen Interaktionen folgen, bei denen der Teilnehmer $\mathfrak{p}_{req}$ Nachrichten mit den Teilnehmern $\mathfrak{p}_1 - \mathfrak{p}_n$ austauscht. Während dieser multilateralen Interaktionen muss jeder involvierte Teilnehmer eine Nachrichtenkopie, d.h. eine Nachricht mit demselben Inhalt, an $\mathfrak{p}_{ref}$ senden. In Abbildung 7.15 ist dies durch die Aktivität *snd3* umgesetzt, die parallel zur Aktivität *snd3* eine Kopie der Nachricht an $\mathfrak{p}_{ref}$ sendet.

„Relayed Request“ Interaktionen können ebenfalls konsolidiert werden, da sie immer aus einer „Send/Receive“ und mehreren multilateralen Interaktionen bestehen[1].

7.4.3. Dynamic Routing

Das Muster „*Dynamic Routing*“ ist eine Erweiterung der Muster „Request with Referral“ bzw. „Relayed Request“, bei dem die Teilnehmer, an die die Anfragenachrichten von Teilnehmer $\mathfrak{p}_{ref}$ weitergeleitet werden, zur Laufzeit anhand einer Routing-Bedingung bestimmt werden. Die Routing-Bedingung

[1] Da die Konsolidierung analog zu der von „Request with Referral“ Interaktionen ist, wird hier darauf verzichtet, die konsolidierte Interaktion aus Abbildung 7.15 darzustellen.

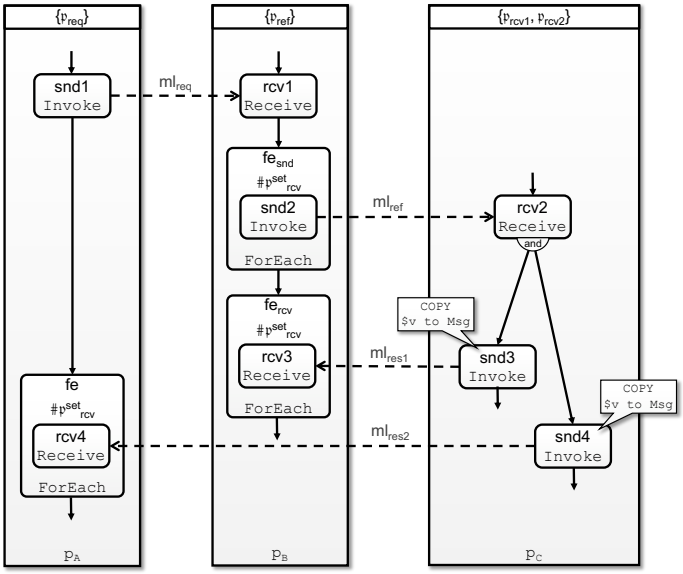

Abbildung 7.15.: Interaktionsmuster „Relayed Request“

und die Ermittlung der Teilnehmermenge, an die $\mathfrak{p}_{ref}$ die Nachricht weiterleitet, kann abhängig vom Anwendungsfall auf unterschiedliche Art und Weise modelliert werden. Um dieses Muster in der „Relayed Request“ Interaktion in Abbildung 7.15 umzusetzen, können zum Beispiel vor der Schleife fe_{snd} Aktivitäten eingefügt werden, die die Teilnehmer in der Teilnehmermenge $\mathfrak{p}^{set}_{rcv}$ abhängig vom Inhalt der Anfragenachricht ermitteln. Wird die Logik zur Ermittlung der Teilnehmermenge ohne Interaktionen oder nur mit den in diesem Kapitel beschriebenen Interaktionsmustern umgesetzt, kann sie folglich ebenfalls konsolidiert werden.

7.5. Zusammenfassung

In diesem Kapitel wurde die Prozesskonsolidierungsmethode anhand von verschiedenen Interaktionsmustern [BDtH05] validiert, die komplexe In-

teraktionen zwischen zwei oder mehreren Teilnehmern beschreiben. Dazu wurde grafisch gezeigt, wie die Muster mit den Elementen des Metamodells als Choreographiemodelle umgesetzt werden können. Basierend auf dieser Umsetzung wurden die Konsolidierungsschritte erläutert, die nötig sind, um das jeweilige Modell zu konsolidieren.

Es können alle Muster bis auf „Atomic Multicast Notification“ und „Multi-Responses“ konsolidiert werden. Die Konsolidierung von „Atomic Multicast Notification“ ist nicht möglich, da das Metamodell keine Transaktionalität unterstützt. Das Muster „Multi-Responses“ kann zwar durch die Fusion von While-Schleifen konsolidiert werden. Aufgrund der Limitation der Fusion, dass in der fusionierten Schleife die Fehlerisolation zwischen den Aktivitäten nicht beibehalten werden kann, die aus verschiedenen Schleifen stammen, kann jedoch im Fehlerfall das Verhalten des Musters „Multi-Responses“ im konsolidierten Prozessmodell nicht äquivalent zur Choreographie emuliert werden.

Da die Interaktionsmuster aus einer Vielzahl von Beispielen aus Industrie und Wissenschaft abstrahiert wurden und da bis auf die eben genannten Ausnahmen alle Muster konsolidiert werden können, kann festgestellt werden, dass die Prozesskonsolidierungsmethode ihre praktische Anwendbarkeit bewiesen hat.

Kapitel 8

Wiederherstellung von fragmentierten BPEL-Prozessmodellen

Um das Outsourcing von Unternehmensteilen zu unterstützen, müssen u.a. Teile aus existierenden Geschäftsprozessen in die neuen organisatorischen Einheiten ausgelagert werden. Dazu beschreibt Khalaf in ihren Arbeiten [KL06; Kha08; KL10; KL12] für BPEL-Prozesse, wie deren auszulagernden Aktivitäten zur Entwurfszeit verschiedenen Teilnehmern zugeordnet und automatisiert in eigenständige miteinander interagierende Prozessmodelle, im Folgenden auch Prozessfragmente[1] genannt, transformiert werden können. Ziel dieser *Prozessfragmentierung* ist es, dass die interagierenden Prozesse zur Laufzeit die operationale Semantik des originalen Prozessmodells

[1] In diesem Kapitel wird unter dem Begriff Prozessfragment ein vollständiges durch die Fragmentierung generiertes Prozessmodell verstanden, im Gegensatz zu Abschnitt 3.3, wo ein Prozessfragment als unterspezifiziertes Prozessmodell definiert ist.

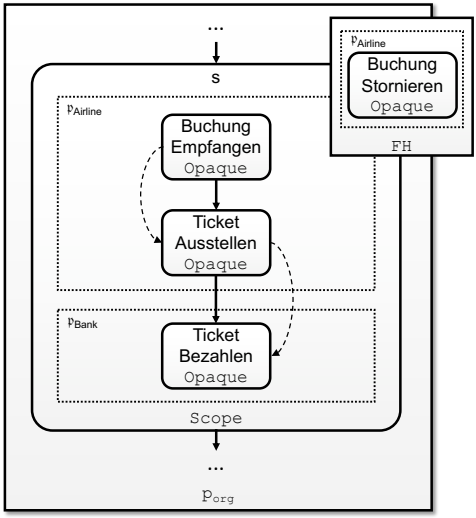

Abbildung 8.1.: Beispielprozess zur Buchung eines Flugtickets

emulieren. In diesem Kapitel wird untersucht, ob und inwieweit die in den vorherigen Kapiteln beschriebene Prozesskonsolidierung die Fragmentierung eines Prozessmodells rückgängig machen kann, d.h. ob die Fragmentierung die Umkehroperation zur Konsolidierung ist. Dazu wird in Abschnitt 8.2 untersucht, ob das konsolidierte Prozessmodell, das aus den Prozessfragmenten erstellt wurde, verhaltensäquivalent zum fragmentierten Prozessmodell ist. In Abschnitt 8.3 wird skizziert, wie die Prozesskonsolidierung erweitert werden kann, um Prozessfragmente korrekt zu konsolidieren.

Als Illustrationsszenario für dieses Kapitel dient das Prozessmodell p_{org} in Abbildung 8.1, das die Schritte einer Flugticketbuchung vereinfacht darstellt[1]. Die Datenabhängigkeiten zwischen den Aktivitäten sind durch gestrichelte Pfeile gekennzeichnet und die Fluggesellschaft möchte den Bezahlvorgang, d.h. die Aktivität *Ticket Bezahlen*, komplett an eine Bank auslagern, die durch den Teilnehmer $\mathfrak{p}_{Bank}$ repräsentiert wird.

[1] Flow-Aktivitäten sind in der Abbildung der Übersichtlichkeit wegen nicht dargestellt.

8.1. Grundlagen der Prozessfragmentierung

Die Fragmentierungsoperation erhält als Eingabe das zu fragmentierende Prozessmodell $p_{org} \in P$ und die *Fragmentspezifikation* F_{spec}, die jedem Teilnehmer eine oder mehrere Aktivitäten aus p_{org} zuweist.

$$F_{spec} := \{(\mathfrak{p}, A) \mid \mathfrak{p} \in \mathfrak{P}_\sigma \wedge A \subset \pi_1(p_{org})\}$$

Dabei müssen die Mengen der den Teilnehmern $\mathfrak{P}_\sigma \subseteq \mathfrak{P}$ zugeordneten Aktivitäten disjunkt sein, es gilt also $\bigcap_{\mathfrak{p} \in \mathfrak{P}_\sigma} \pi_2(\mathfrak{p}, A) = \emptyset$.

Die Ausgabe der Prozessfragmentierung ist eine Choreographie $c_\phi \in C$, die die Teilnehmer der Fragmentspezifikation enthält[1]. Formal ist die Fragmentierungsoperation ϕ also wie folgt definiert:

$$\phi : P \times \wp(F_{spec}) \to C$$

In diesem Abschnitt werden hauptsächlich die Kontrollflussaspekte der Fragmentierung betrachtet, die für die Prüfung der Verhaltensäquivalenz relevant sind und auf das hier verwendete Metamodell übertragen.

Die Fragmentierungsoperation besteht dabei aus den folgenden Schritten, die im Detail in der Dissertation von Khalaf [Kha08, S. 91–134] und den der Dissertation zugrunde liegenden Arbeiten beschrieben werden:

1. Ermittlung der Datenabhängigkeiten zwischen den Aktivitäten in p_{org}
2. Erstellung der Prozessmodelle der Teilnehmer anhand der Fragmentspezifikation
3. Synchronisierung der Zustände von fragmentierten strukturierten Aktivitäten

[1] Khalaf nutzt nicht explizit BPEL4Chor als Ausgabeformat, sondern einen daran angelehnten Formalismus. Cui beschreibt in [Cui12], wie dieser Formalismus in eine BPEL4Chor-Choreographie transformiert werden kann.

4. Realisierung der Kommunikation zwischen den Fragmenten

In BPEL-Prozessmodellen kann der Datenfluss zwischen den Aktivitäten eines Prozessmodells nur implizit über Variablen und den darauf lesend bzw. schreibend zugreifenden Kontrollflusskonstrukten modelliert werden. Um über Nachrichtenaustausch den Datenfluss zwischen Aktivitäten zu emulieren, die sich nach der Fragmentierung von p_{org} in verschiedenen Prozessmodellen befinden, müssen die expliziten Datenabhängigkeiten allerdings bekannt sein. Die Aktivität *Ticket Bezahlen* hat zum Beispiel über den Ticketpreis eine Datenflussabhängigkeit zur Aktivität *Ticket Ausstellen*, diese muss auch nach der Fragmentierung erhalten bleiben. Die expliziten Datenflussabhängigkeiten zwischen den Paaren von Aktivitäten eines Prozessmodells können mit dem von Khalaf, Kopp und Leymann [KKL08] beschriebenen Algorithmus ermittelt werden.

Im zweiten Schritt wird für jeden Teilnehmer der Fragmentspezifikation ein neues Prozessmodell (Fragment) erstellt, in das die dem Teilnehmer zugeordneten Aktivitäten eingefügt werden. Neben diesen Aktivitäten werden auch deren Vorfahren (z.B. Scopes und Schleifen) in die neu erstellten Prozessmodelle eingefügt, um die Verschachtelung und damit den Kontrollflusskontext der jeweiligen Aktivität beizubehalten (*Rubber Band Effect* [KL10]). Das bedeutet, dass alle strukturierten Aktivitäten, die direkte oder indirekte Kindaktivitäten enthalten, die verschiedenen Fragmenten zugeordnet sind, ebenfalls fragmentiert werden.

Durch die Fragmentierung können strukturierte Aktivitäten entstehen, die keine Kindaktivität mehr haben. Wird zum Beispiel die primäre Kindaktivität eines Scopes mit Fault-Handler dem Teilnehmer $\mathfrak{p}_1$ und die Fehlerbehandlungsaktivität dem Teilnehmer $\mathfrak{p}_2$ zugeordnet, würde das Prozessmodell, das Teilnehmer $\mathfrak{p}_2$ implementiert, einen Scope ohne Kindaktivität enthalten. Dies würde die BPEL-Syntax verletzen. Diese Verletzungen werden dadurch aufgehoben, dass ebenfalls im zweiten Schritt die „leeren“ strukturierten Aktivitäten mit Empty-Aktivitäten aufgefüllt werden. In den Scope würde zum Beispiel ein Empty als dessen primäre Kindaktivität eingefügt werden.

Zwischen den zusammengehörigen Prozessmodellen die aus p_{org} erstellt wurden, werden zur Laufzeit über einen Koordinator (siehe Abschnitt 8.1.1) Informationen über die Zustände der fragmentierten strukturierten Aktivitäten ausgetauscht. Damit wird zum Beispiel sichergestellt, dass die Terminierung des Fragments einer strukturierten Aktivität in einem Prozessmodell ebenfalls zur Terminierung der zugehörigen Aktivitätsfragmente in den anderen Prozessmodellen führt. Dazu werden im dritten Schritt zusätzliche Informationen generiert, die es dem Koordinator ermöglichen, zusammengehörige Aktivitätsfragmente in unterschiedlichen Prozessmodellen zu identifizieren.

Im vierten Schritt wird die Kommunikation zwischen den Prozessmodellen über den Austausch von Nachrichten realisiert. Dazu werden die Kommunikationsaktivitäten zu den Prozessen hinzugefügt und die zugehörigen Artefakte, wie zum Beispiel WSDL-Dateien, generiert. Damit kann zwischen den Prozessen der im ersten Schritt ermittelte Datenfluss von p_{org} emuliert und der Zustand von Kontrollflusskanten propagiert werden. Letzteres ist notwendig, wenn Aktivitäten unterschiedlichen Prozessmodellen zugeordnet werden, die in p_{org} über eine Kontrollflusskante miteinander verbunden waren. Im Detail wird die Fragmentierung von Kontrollflusskanten in Abschnitt 8.1.2 diskutiert.

8.1.1. Synchronisierung von fragmentieren strukturierten Aktivitäten

Werden die Kindaktivitäten einer strukturierten Aktivität auf verschiedene Prozessmodelle bzw. Teilnehmer aufgeteilt, wird die strukturierte Aktivität ebenfalls fragmentiert. Der Scope s aus Abbildung 8.1 wird zum Beispiel in den Scope s_{Frag1} und s_{Frag2} aufgeteilt (siehe Abbildung 8.2).

Zwischen den Fragmenten derselben strukturierten Aktivität müssen die Zustände synchronisiert werden, um das originale Verhalten zu emulieren. So müssen alle Fragmente der strukturierten Aktivität gleichzeitig in den selben Zustand, wie zum Beispiel *executing* oder *faulted*, gesetzt werden. Ist

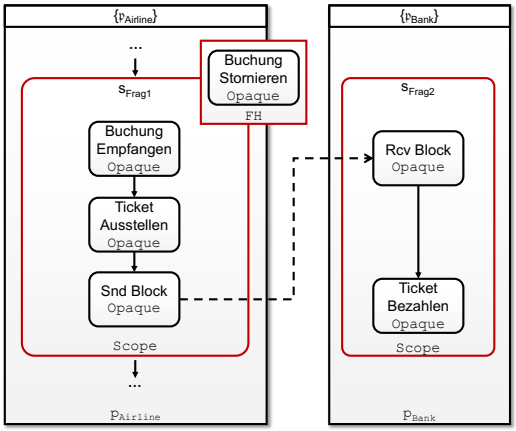

Abbildung 8.2.: Eine fragmentierte Scope-Aktivität

ein Scope *s* so fragmentiert, dass sich die primäre Kindaktivität in einem anderen Fragment befindet als die Fehlerbehandlungsaktivitäten, müssen letztere über den Fehler informiert werden, um ihn zu verarbeiten. Der Fault-Handler mit der Aktivität *Buchung stornieren* muss also, obwohl er einem anderen Teilnehmer zugeordnet ist, ausgeführt werden, wenn eine Instanz von *Ticket bezahlen* in *faulted* geht. Bei einer fragmentierten Schleife muss sichergestellt werden, dass alle Fragmente eine Iteration abgeschlossen haben, bevor ein Fragment die nächste Iteration starten kann [KL12].

Da die Synchronisation der Zustände zwischen den Fragmenten der strukturierten Aktivitäten über Nachrichten zu komplex wäre, schlagen Khalaf und Leymann [KL10] einen Ansatz vor, dies mittels eines zentralen Koordinators umzusetzen, der den OASIS Standard WS-Coordination [OAS07a] implementiert. Dieser Koordinator ist eine zusätzliche externe Middleware, die während der Ausführung die Fragmente überwacht, in dem er zur Laufzeit Zustandsinformationen über die Aktivitätsinstanzen der Fragmente von der Workflow-Engine erhält. Abhängig vom Zustand einer oder mehrerer Fragmente instruiert der Koordinator wiederum die Engines, die Instanzen in den anderen Fragmenten in einen bestimmten Zustand zu setzen. So

instruiert der Koordinator zum Beispiel das Fragment eines Scopes, das die Fehlerbehandlungsaktivität des Scopes enthält, diese auszuführen, falls dessen Kindaktivität in einem anderen Fragment einen Fehler auslöst.

Das Protokoll, das der Koordinator verwendet, hängt vom Typ der strukturierten Aktivität ab. Der Koordinator wird von der Workflow-Engine aufgerufen, sobald diese während der Orchestrierung eine fragmentierte strukturierte Aktivität ausführt. Um fragmentierte Aktivitäten zu kennzeichnen, werden die BPEL-Aktivitätsbeschreibungen mit entsprechenden Attributen erweitert [Pal07]. Die verwendete Workflow-Engine wird ebenfalls erweitert, um diese Attribute zu interpretieren und den Koordinator aufzurufen.

8.1.2. Fragmentierung von Kontrollfluss- und Datenkanten

Befinden sich zwei Aktivitäten, die in p_{org} über eine Kontrollfluss- oder Datenkante verbunden waren, in verschiedenen Fragmenten, muss der Zustand der Kontrollflusskante und die Daten zwischen den Fragmenten über Nachrichten ausgetauscht werden. Dies ist zum Beispiel bei den Aktivitäten a_{pred} und a_{succ} in Abbildung 8.3 der Fall. Dort muss nach der Fragmentierung von p_{org} der Zustand der Transitionsbedingung l_{split} vom Prozessfragment des Teilnehmers $\mathfrak{p}_A$ zur Eintrittsbedingung von a_{succ} im Prozessfragment von Teilnehmer $\mathfrak{p}_B$ propagiert werden.

Der Austausch von Transitionsbedingungszuständen bzw. Daten über Nachrichten wurde durch Khalaf und Leymann [KL06] beschrieben. Dieser Ansatz wird auf das hier verwendete Metamodell übertragen. Da der Fokus dieser Arbeit auf dem Kontrollfluss liegt, wird hier hauptsächlich auf die Übertragung des Zustands von Kontrollflusskanten eingegangen. Eine Kontrollflusskante, deren Quell- und Zielaktivität sich in verschiedenen Fragmenten befindet, wird als fragmentierte Kante l_{split} bezeichnet. Um den Nachrichtenaustausch zu realisieren, wird im Fragment, in dem sich die Quellaktivität a_{pred} der Kante befindet, ein *sendender Block* und in dem Fragment, in dem sich die Zielaktivität a_{succ} befindet, ein *empfangender Block* erstellt.

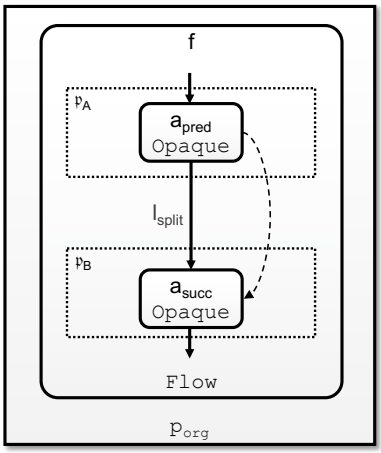

Abbildung 8.3.: Die Quell- und Zielaktivität der zu fragmentierenden Kontrollflusskante l_{split} sind Teilnehmer $\mathfrak{p}_A$ bzw. $\mathfrak{p}_B$ zugeordnet

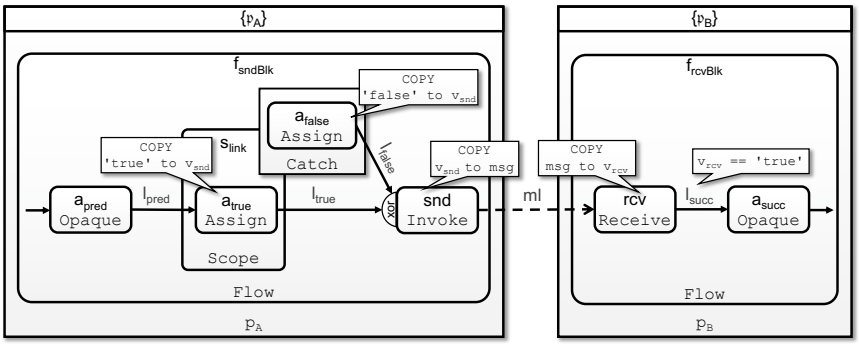

Abbildung 8.4.: Propagierung des Zustands der fragmentierten Kontrollflusskante zwischen sendendem und empfangendem Block

Der sendende Block ist links in Abbildung 8.4 dargestellt und besteht formal aus den Aktivitäten A_{sndBlk}, den Kontrollflusskanten L_{sndBlk} und den Hierarchiebeziehungen HR_{sndBlk}:

- $A_{sndBlk} := \{a_{pred}, a_{true}, a_{false}, snd, s_{link}, f_{snd}\}$:
  - $a_{pred} \in A_{\texttt{opaque}} \wedge a_{pred} = \pi_1(l_{split})$
  - $a_{true} \in A_{\texttt{assign}}$:
    - $\mathsf{joinFault}(a_{true}) \leftarrow \texttt{true}$
    - $\mathsf{assign}_{src}(a_{true}) \leftarrow v_{true}$ mit $v_{true} \in V \wedge \mathsf{value}_V(v_{true}) \leftarrow \texttt{true}$
    - $\mathsf{assign}_{trg}(a_{true}) \leftarrow v_{snd}$ mit $v_{snd} \in V$
  - $a_{false} \in A_{\texttt{assign}}$:
    - $\mathsf{assign}_{src}(a_{false}) \leftarrow v_{false}$ mit $v_{false} \in V \wedge \mathsf{value}_V(v_{false}) \leftarrow \texttt{false}$
    - $\mathsf{assign}_{trg}(a_{false}) \leftarrow v_{snd}$
  - $snd \in A_{\texttt{invoke}}$ mit $\mathsf{inputVar}(snd) \leftarrow v_{snd}$
  - $s_{link} \in A_{\texttt{scope}}$
  - $f_{sndBlk} \in A_{\texttt{flow}}$

- $L_{sndBlk} := \{l_{pred}, l_{true}, l_{false}\}$:
  - $l_{pred} = (a_{pred}, a_{true}, \pi_3(l_{split}))$
  - $l_{true} = (a_{true}, snd, \texttt{true})$
  - $l_{false} = (a_{false}, snd, \texttt{true})$

- $HR_{sndBlk} := \{hr_{a_{pred}}, hr_{s2true}, hr_{s2false}, hr_{sPar}, hr_{sndPar}\}$:
  - $hr_{a_{pred}} = (f_{sndBlk}, \mathfrak{t}^{ch}_{HR}, a_{pred})$
  - $hr_{s2true} = (s_{link}, \mathfrak{t}^{ch}_{HR}, a_{true})$
  - $hr_{s2false} = (s_{link}, e^{joinFault}, a_{false})$ mit $e^{joinFault} \in E^{joinFault}$
  - $hr_{sPar} = (f_{sndBlk}, \mathfrak{t}^{ch}_{HR}, s_{link})$
  - $hr_{sndPar} = (f_{sndBlk}, \mathfrak{t}^{ch}_{HR}, snd)$

Im sendenden Block wird die Kontrollflusskante l_{split} durch die Kante l_{pred} repräsentiert, deren Zustand an den empfangenden Block gesendet wird. Der Zustand von l_{pred} wird dabei wie folgt ermittelt. Evaluiert l_{pred} zur Laufzeit zu `false`, löst die Aktivität a_{true} einen Join-Fehler aus, der vom Fault-Handler

des Scopes s_{link} gefangen wird. Die Fehlerbehandlungsaktivität a_{false} weist der Variable v_{snd} daraufhin den Wert `false` zu. Evaluiert l_{pred} zu `true`, weist die Aktivität a_{true} der Variablen v_{snd} den Wert `true` zu. Der Inhalt der Variable v_{snd} wird dann über das Invoke *snd* an das Receive *rcv* im empfangenden Block gesendet.

Der empfangende Block (rechts in Abbildung 8.4) empfängt den Zustand von l_{pred} über die Nachrichtenkante *ml* und prüft abhängig davon, ob die Nachfolgeaktivität a_{succ} ausgeführt werden kann. Dies ist durch das Receive *rcv* und die Kontrollflusskante l_{pred} realisiert, die nur aktiviert wird, wenn die Variable v_{link} den Wert `true` hat:

- $A_{rcvBlk} := \{a_{succ}, rcv, f_{rcvBlk}\}$:
  - $a_{succ} \in A_{\texttt{opaque}} \wedge a_{pred} = \pi_2(l_{split})$
  - $rcv \in A_{\texttt{receive}}$ mit $\mathsf{outputVar}(rcv) = v_{rcv} \wedge v_{rcv} \in V$
  - $f_{rcvBlk} \in A_{\texttt{flow}}$
- $L_{rcvBlk} := \{l_{succ}\}$:
  - $l_{succ} = (rcv, a_{succ}, \mathrm{v}_{rcv} = \texttt{true})$
- $HR_{rcvBlk} := \{hr_{parent}\}$
  - $hr_{a_{succ}} = (f_{rcvBlk}, \mathfrak{t}^{ch}_{HR}, a_{succ})$
  - $hr_{rcvPar} = (f_{rcvBlk}, \mathfrak{t}^{ch}_{HR}, rcv)$
- $ML_{sndRcvBlk} := \{ml\}$ mit $ml = (\{\mathfrak{p}_A\}, snd, \{\mathfrak{p}_B\}, rcv)$

8.1.2.1. Zustandsbeziehungen zwischen a_{pred} und a_{succ}

In diesem Abschnitt werden die Zustandsbeziehungen betrachtet, die zwischen a_{pred} und a_{succ} gelten, wenn diese sich in dem oben formalisierten sendenden bzw. empfangenden Block befinden. Die Aktivitäten a_{pred} und a_{succ} haben eine indirekte Kontrollflussbeziehung über die Kontrollflusskanten, die Hierarchiebeziehungen und der Nachrichtenkante. Da sie sich in unterschiedlichen Prozessmodellen befinden, sind a_{pred} und a_{succ} eigentlich fehlerisoliert voneinander (siehe Abschnitt 4.2.7). Die Isolation wird allerdings dadurch aufgehoben, dass der Koordinator die Zustände zwischen

		a_{succ}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{pred}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	‖	‖	⊕	⊕	⊕	⊕
	ABORTED	←	‖	‖	⊕	⊕	⊕	⊕
	EXECUTING	←	‖	‖	→	→	→	→
	TERMINATED	←	‖	‖	⊕	⊕	⊕	⊕
	FAULTED	←	‖	‖	⊕	⊕	⊕	⊕
	COMPLETED	←	‖	‖	→	→	→	→

Abbildung 8.5.: Zustandstransitionsprofil zwischen Quellaktivität a_{pred} und Zielaktivität a_{succ} einer fragmentierten Kontrollflusskante ohne Koordinator

den Fragmenten ihrer gemeinsamen Elternaktivität synchronisiert. Löst die Instanz von a_{pred} einen Fehler aus, führt dies auch dazu, dass die Instanz des Flow f_{sndBlk} in *faulted* geht. Dies veranlasst den Koordinator zur direkten Terminierung der Instanz von f_{rcvBlk} und seiner Kindaktivitäten, da f_{sndBlk} und f_{rcvBlk} Fragmente des Flow f sind.

Die Konsolidierung berücksichtigt den Koordinator nicht. Daher zeigt Abbildung 8.5 das Profil zwischen a_{pred} und a_{succ} nach der Fragmentierung von l_{split} ohne die Zustandssynchronisationen des zentralen Koordinators. Geht a^I_{pred}, also die Instanz von a_{pred}, in den Zustand *completed*, erreicht im regulären Kontrollfluss die Instanz $a^I_{a_{true}}$ danach ebenfalls den Zustand *completed*[1] (Axiom *Sq1*) und die Instanz $a^I_{a_{false}}$ den Zustand *dead* (Axiom $Sc_{PCH2FH}2$). Erreicht a^I_{pred} hingegen den Zustand *dead*, wird aufgrund des Join-Fehlers die Instanz $a^I_{a_{true}}$ in *faulted* gesetzt und $a^I_{a_{false}}$ erreicht danach den Zustand *executing* (Axiom $Sc_{PCH2FH}1$) und *completed*. In beiden Fällen geht snd^I, nachdem $a^I_{a_{true}}$ bzw. $a^I_{a_{false}}$ einen Endzustand erreicht haben, in den Zustand *executing* (Axiom $Xor_{Join}1$). Aufgrund der Nachrichtenkante zwischen *snd* und *rcv* sowie der Kontrollflusskante l_{succ} zwischen *rcv* und a_{succ}, kann a^I_{succ} in *executing* oder

[1] Es wird davon ausgegangen, dass während der Fragmentierung der sendende bzw. empfangende Block korrekt modelliert werden und somit keine Fehler bei der Ausführung von $a^I_{a_{true}}$ bzw. $a^I_{a_{false}}$ auftreten können.

dead gehen, nachdem snd^I und somit rcv^I den Zustand *completed* erreicht haben (Axiom $Ml_{InvRcv}1$). Enthält v_{rcv} den Wert `true`, wird die Instanz von l_{succ} aktiviert und a^I_{succ} geht, wie für die sequentielle Ausführung in Axiom *Sq1* definiert, in den Zustand *executing* nachdem a^I_{pred} *completed* erreicht hat. Enthält v_{rcv} hingegen den Wert `false`, wird a^I_{succ} in *dead* gesetzt, da die Instanz von l_{succ} deaktiviert wird. Den Zustand *dead* kann a^I_{succ} also ebenfalls weiterhin erreichen, nachdem a^I_{pred} den Zustand *dead* erreicht hat. Allerdings kann a^I_{succ} den Zustand *dead* auch erreichen, bevor a^I_{pred} den Zustand *dead* erreicht. Dies liegt an der Isolation von f_{sndBlk} und f_{rcvBlk}, da so die Instanzen $a^I_{f_{sndBlk}}$ und $a^I_{f_{rcvBlk}}$ des fragmentierten Flow f und deren Kindaktivitäten unabhängig voneinander in den Zustand *dead* gesetzt werden können.

Erreicht a^I_{pred} einen Fehlerzustand, wird a^I_{snd} in den Zustand *aborted* gesetzt (Axiom $St_{CH2CH}2$) und kann somit keine Nachricht an a^I_{rcv} senden. Daher müssen a^I_{rcv} und a^I_{succ} terminiert werden oder den Zustand *dead* erreichen, so dass kein Deadlock entsteht (siehe Abschnitt 4.2.14). Auch hier kann aufgrund der Isolation von a_{pred} und a_{succ} keine Reihenfolge abgeleitet werden, ob a^I_{rcv} und damit a^I_{succ} diese Zustände erreichen, bevor oder nachdem a^I_{pred} einen Fehlerzustand erreicht hat. Selbst wenn a^I_{pred} den Zustand *completed* erreicht, kann a^I_{succ} aufgrund der Isolation vorher schon terminiert werden oder den Zustand *dead* erreicht haben.

Das Profil in Abbildung 8.5 entspricht daher nicht dem Profil der sequentiellen Ausführung von a_{pred} und a_{succ} aus Abbildung 4.18. Werden allerdings die Zustandssynchronisationen des Koordinators zwischen den Fragmenten f_{sndBlk} und f_{rcvBlk} mit berücksichtigt, ergibt sich das Profil in Abbildung 8.6, das dem der sequentiellen Ausführung entspricht. Die Instanzen $a^I_{f_{sndBlk}}$ und $a^I_{f_{rcvBlk}}$ können nicht mehr unabhängig voneinander in *dead* gehen, da der Koordinator, um das Verhalten des unfragmentierten Flow f zu emulieren, beide gleichzeitig in *dead* setzt. Erreicht $a^I_{f_{sndBlk}}$ einen Fehlerzustand, wird dieser direkt an $a^I_{f_{rcvBlk}}$ weitergeleitet und umgekehrt. Zum Beispiel kann a^I_{succ} dann nicht mehr *dead* erreichen, wenn a^I_{pred} terminiert wurde.

		a_{succ}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{pred}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	→	→	⊕	⊕	⊕	⊕
	ABORTED	←	⊕	‖	⊕	⊕	⊕	⊕
	EXECUTING	←	⊕	→	→	→	→	→
	TERMINATED	←	⊕	‖	⊕	⊕	⊕	⊕
	FAULTED	←	⊕	→	⊕	⊕	⊕	⊕
	COMPLETED	←	⊕	→	→	→	→	→

Abbildung 8.6.: Zustandstransitionsprofil zwischen Quellaktivität a_{pred} und Zielaktivität a_{succ} einer fragmentierten Kontrollflusskante mit Koordinator

8.2. Wiederherstellung von fragmentierten Prozessmodellen

In diesem Abschnitt wird untersucht, ob die Konsolidierung der Fragmente von p_{org} ein Prozessmodell $p_\mu = \mu(\phi(p_{org}))$ erzeugt, das verhaltensäquivalent zu p_{org} ist, d.h. ob zwischen den Choreographien c_{org} und c_μ, denen sie zugeordnet sind, gilt:

$$\overbrace{(\{\mathfrak{p}_{org}\}, \emptyset, \emptyset, p_{org})}^{c_{org}} \approx_{\texttt{task}} \overbrace{(\{\mathfrak{p}_\mu\}, \emptyset, \emptyset, p_\mu)}^{c_\mu}$$
$$\text{mit } \mathsf{type}_{\mathfrak{p}}(\mathfrak{p}_{org}) = p_{org} \wedge \mathsf{type}_{\mathfrak{p}}(\mathfrak{p}_\mu) = p_\mu$$

8.2.1. Generierung der Teilnehmer-Scopes aus den Fragmenten

Die aus der Fragmentierung von p_{org} resultierende Choreographie c_ϕ ist also die Eingabe der Konsolidierungsoperation. Im ersten Schritt der Konsolidierung wird für jeden Teilnehmer in c_ϕ bzw. für jedes Fragment von p_{org} ein Teilnehmer-Scope in p_μ zusammen mit dem Kontroll- und Datenfluss des jeweiligen Fragments eingefügt. Dieser Konsolidierungsschritt erhält, wie in Abschnitt 5.3 erläutert, die Kontrollflussbeziehungen zwischen den Aktivitäten in den jeweiligen Fragmenten und isoliert die Aktivitäten der unterschiedlichen Fragmente voneinander. Dies resultiert allerdings in der

Isolation von Aktivitäten, die in p_{org} nicht voneinander isoliert waren. Das führt wiederum dazu, dass Fehler nicht mehr zwischen Aktivitäten propagiert werden, wie dies in p_{org} der Fall war. Wird die durch die Fragmentierung entstandene Choreographie des Illustrationsszenarios konsolidiert, sind die Aktivitäten $s_{Airline}$ und s_{Bank} voneinander isoliert. Löst jetzt zum Beispiel die Instanz der Aktivität *Ticket Ausstellen* einen Fehler aus, wird die Instanz der Aktivität *Ticket Bezahlen* nicht terminiert. In den verschiedenen Prozessmodellen sind die Aktivitäten zwar ebenfalls voneinander isoliert, Fehler werden aber über den Koordinator propagiert, der bei der Konsolidierung nicht berücksichtigt wird. Folglich ist die Verhaltensäquivalenz zwischen p_{org} und p_{μ} im Fehlerfall nicht gegeben.

Die zusammengehörenden Fragmente von strukturierten Aktivitäten können ohne Koordinator auch nicht gleichzeitig in den Zustand *executing*, *completed* oder *dead* gesetzt werden. Folglich ist in p_{μ} aufgrund der Nichtberücksichtigung des Koordinators auch im regulären Kontrollfluss die Verhaltensäquivalenz nicht gegeben.

8.2.2. Materialisierung der Interaktion zwischen sendendem und empfangendem Block

Die Fragmentierung erstellt zwischen a_{pred} und a_{succ} ausschließlich sendende und empfangende Blöcke, die über Invoke-Receive-Interaktionen kommunizieren. Diese Interaktionen werden bei der Konsolidierung, wie in Abschnitt 5.4.1 beschrieben, materialisiert. Abbildung 8.7 zeigt das Ergebnis der Materialisierung des sendenden und empfangenden Blocks.

In p_{μ} wird bei der Konsolidierung das Duplikat a_{predCp} der Aktivität a_{pred} und das Duplikat a_{succCp} der Aktivität a_{succ} zusammen mit den Duplikaten der anderen Aktivitäten des sendenden und empfangenden Blocks eingefügt. Formal ändert sich im Duplikat des sendenden Blocks nur, dass das Invoke *snd* durch das Assign syn_{snd} ersetzt wird, das die Nachrichtenübertragung des Zustands von l_{split} emuliert:

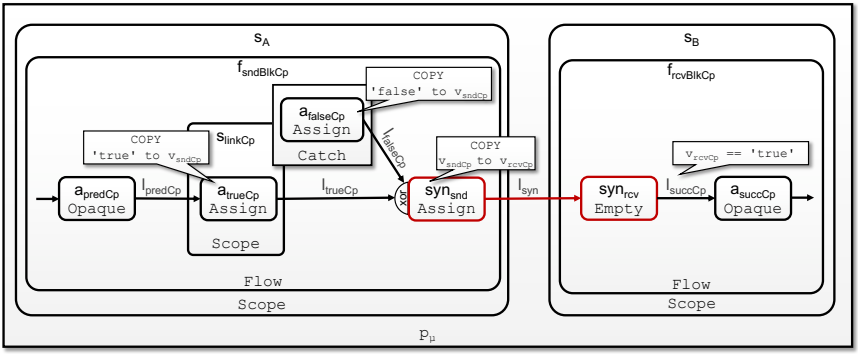

Abbildung 8.7.: Materialisierte Interaktion zwischen sendendem und empfangendem Block aus Abbildung 8.4

- $A_{sndBlkCp} := \{a_{predCp}, a_{trueCp}, a_{falseCp}, syn_{snd}, s_{linkCp}, f_{sndBlkCp}\}$[1]:
  - ...
  - $syn_{snd} \in A_{\texttt{assign}}$ mit:
    - $\text{assign}_{src}(syn_{snd}) = v_{sndCp}$ mit $v_{sndCp} \in V$ (Duplikat von v_{snd})
    - $\text{assign}_{trg}(syn_{snd}) = v_{rcvCp}$ mit $v_{rcvCp} \in V$ (Duplikat von v_{rcv})
  - ...
- $L_{sndBlkCp} := \{l_{predCp}, l_{trueCp}, l_{falseCp}\}$:
  - ...
  - $l_{trueCp} = (a_{trueCp}, syn_{snd}, \texttt{true})$
  - $l_{falseCp} = (a_{falseCp}, syn_{snd}, \texttt{true})$
  - ...
- $HR_{sndBlkCp} := \{hr_{s2trueCp}, hr_{s2falseCp}, hr_{sParCp}, hr_{synPar}\}$:
  - ...
  - $hr_{synPar} = (f_{sndBlk}, \mathfrak{t}^{ch}_{HR}, syn_{snd})$
  - ...

Im Duplikat des empfangenden Blocks wird das Receive *rcv* durch die Syn-

[1] Es werden hier nur die Kontrollflusselemente angegeben, die Beziehungen zu *snd* haben, da die Beziehungen der Aktivitätsduplikate unverändert zum sendenden Block im Prozessmodellfragment sind.

chronisationsaktivität syn_{rcv} ersetzt, die über eine Kontrollflusskante l_{syn} mit der Aktivität syn_{snd} im sendenden Block verbunden ist:

- $A_{rcvBlkCp} := \{syn_{rcv}, f_{rcvBlkCp}, a_{succCp}\}$
  - $syn_{rcv} \in A_{\texttt{empty}}$
- $L_{rcvBlkCp} := \{l_{syn}, l_{succCp}\}$:
  - $l_{syn} = (syn_{snd}, syn_{rcv}, \texttt{true})$
  - $l_{succCp} = (syn_{rcv}, a_{succCp}, \mathrm{v}_{rcvCp} = \texttt{true})$[1]
- $HR_{rcvBlkCp} := \{hr_{rcvPar}\}$
  - $hr_{rcvPar} = (f_{rcvBlkCp}, \mathfrak{t}^{ch}_{HR}, syn_{rcv})$

8.2.2.1. Zustandsbeziehungen zwischen a_{predCp} und a_{succCp}

Im Folgenden wird untersucht, welche Zustandsbeziehungen zwischen a_{predCp} und a_{succCp} nach der Materialisierung bestehen. Die Aktivitäten a_{predCp} und a_{succCp} befinden sich in verschiedenen Scopes und sind transitiv über die Kontrollflusskanten l_{predCp}, $l_{falseCp}$, l_{trueCp}, l_{syn} und l_{succCp} verbunden. Die Quell- und Zielaktivität von l_{syn} befinden sich in verschiedenen Scopes und Flows. Daher findet hier neben den Axiomen *Sq1 – Sq3* auch das Axiom *SqI3* Anwendung.

Die Zustandsbeziehungen zwischen a_{predCp} und a_{succCp} sind in Abbildung 8.8 dargestellt. Aufgrund der transitiven Verbindung von a_{predCp} mit a_{succCp} über die Kontrollflusskanten gilt weiterhin, dass $a^I_{succ_{Cp}}$ im regulären Kontrollfluss in *executing* geht, nachdem $a^I_{pred_{Cp}}$ den Zustand *completed* erreicht hat. Um in den Zustand *dead* gehen zu können, muss die eingehende Kontrollflusskante von $a^I_{succ_{Cp}}$ evaluiert wurden sein. Dies ist aufgrund der transitiven Kontrollflussverbindung von a_{predCp} und a_{succCp} nur möglich, nachdem $a^I_{pred_{Cp}}$ einen Endzustand erreicht hat. Das ist ein Unterschied zum Profil zwischen a_{predCp} und a_{succCp} in Abbildung 8.5 (die abweichenden Zustandsbeziehungen sind in Abbildung 8.8 rot markiert), der bereits bei der Materialisierung

[1] Als Optimierungsmöglichkeit kann die Kante l_{predCp} entfernt werden und die Kante l_{syn} verbindet direkt syn_{snd} und syn_{rcv}. In diesem Fall würde l_{syn} die Transitionsbedingung von l_{predCp} erhalten.

		a_{succCp}						
		INITIAL	DEAD	ABORTED	EXECUTING	TERMINATED	FAULTED	COMPLETED
a_{predCp}	INITIAL	‖	→	→	→	→	→	→
	DEAD	←	→	‖	⊕	⊕	⊕	⊕
	ABORTED	←	→	‖	⊕	⊕	⊕	⊕
	EXECUTING	←	→	‖	→	→	→	→
	TERMINATED	←	→	‖	⊕	⊕	⊕	⊕
	FAULTED	←	→	‖	⊕	⊕	⊕	⊕
	COMPLETED	←	→	‖	→	→	→	→

Abbildung 8.8.: Zustandstransitionsprofil zwischen dem Duplikat der Quellaktivität a_{predCp} und der Zielaktivität a_{succCp} nach der Materialisierung einer fragmentierten Kontrollflusskante

von Invoke-Receive-Interaktionen in Abschnitt 5.4.1.3 diskutiert wurde. Ein Fehler in der Instanz $a^I_{pred_{Cp}}$ oder deren Terminierung führt ebenfalls zur Terminierung von $a^I_{syn_{snd}}$ und somit dazu, dass l_{syn} zu `false` evaluiert wird. Dies führt wiederum dazu, dass $a^I_{syn_{rcv}}$ und a^I_{succ} danach in *dead* gehen.

Die Zustandsbeziehungen zwischen a_{predCp} und a_{succCp} entsprechen also mit Einschränkung denen, die zwischen a_{pred} und a_{succ} ohne Koordinator gelten. Um die Zustandsbeziehungen wiederherzustellen, die zwischen a_{pred} und a_{succ} im unfragmentierten Prozessmodell p_{org} gelten, muss die Konsolidierungsoperation speziell für die Konsolidierung von fragmentierten strukturierten Aktivitäten erweitert werden.

Es ist also nicht möglich, mit der Prozesskonsolidierung mit dem Ansatz von Khalaf fragmentierte Prozessmodelle in ein Prozessmodell zu überführen, das verhaltensäquivalent zum originalen unfragmentierten Prozessmodell ist.

8.3. Erweiterung des Konsolidierungsansatzes zur Wiederherstellung von fragmentierten Prozessmodellen

Dieser Abschnitt beschreibt, wie die Konsolidierungsschritte angepasst werden müssen, um eine Choreographie c_ϕ, die durch die Fragmentierung eines Prozessmodells p_{org} erstellt wurde, so zu konsolidieren, dass das erzeugte Prozessmodell p_μ verhaltensäquivalent zu p_{org} ist.

Der erste Schritt der Konsolidierung (*Erstellung von p_μ und die Generierung der Teilnehmer-Container* - vgl. Abbildung 5.1) muss so modifiziert werden, dass keine Teilnehmer-Container generiert werden. Damit wird die Isolation der Aktivitäten in den Prozessmodellen von c_ϕ unterbunden, weil sich diese vor der Fragmentierung alle im selben Prozessmodell p_{org} befunden haben und somit nicht isoliert voneinander waren.

Mit der Kontrollflussmaterialisierung werden die fragmentierten Kontrollflusskanten in c_ϕ wiederhergestellt. Sie muss nicht angepasst werden, da sie in p_μ, wie im vorherigen Abschnitt erläutert, die fragmentierte Kontrollflusskante zwischen zwei Aktivitäten verhaltensäquivalent zu p_{org} wiederherstellt. Es bleiben zwar durch die Fragmentierung und die Konsolidierung in p_μ „Narben“ zurück (zum Beispiel Synchronisationsaktivitäten – siehe Abbildung 8.4), das Verhalten zwischen den Aktivitäten bleibt unter Berücksichtigung der durch den Koordinator realisierten Zustandsbeziehungen erhalten.

Der Koordinator ist nicht mehr notwendig, wenn die fragmentierten strukturierten Aktivitäten wieder zu einer Aktivität zusammengefügt werden. Dazu muss der dritte Schritt der Konsolidierung (*Auflösen von Kontrollflussverletzungen*), so angepasst werden, dass er die zusammengehörigen Aktivitätsfragmente identifiziert und in p_μ in eine einzelne Aktivität überführt. Die Identifizierung der verschiedenen Fragmente wird dadurch ermöglicht, dass diese während der Fragmentierung für den Koordinator als zusammengehörig gekennzeichnet wurden (siehe Abschnitt 8.1.1). Die wieder zusammengeführte strukturierte Aktivität muss die Eigenschaften der struk-

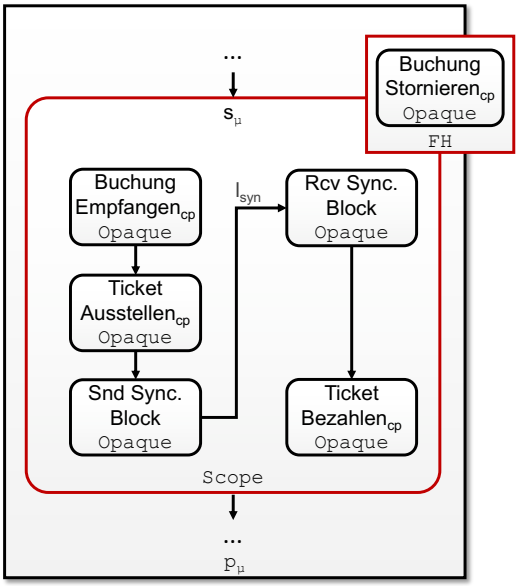

Abbildung 8.9.: Der fragmentierte Beispielprozess aus Abbildung 8.2 nach der Konsolidierung mit den aus den Fragmenten erstellten Scope s_μ

turierten Aktivitätsfragmente besitzen (Typ, Attribute etc.) und alle ihre Kindaktivitäten. Abbildung 8.9 zeigt die Choreographie aus Abbildung 8.2 nach der Konsolidierung[1] mit dem aus den Scope-Fragmenten s_{Frag1} und s_{Frag1} erstellten Scope s_μ.

Da damit auch Fragmente zusammengehöriger Schleifen identifiziert und zusammengefügt werden können, kommt es zu keinen grenzverletzenden Kontrollflusskanten in p_μ. Daher ist es nicht nötig, während der Konsolidierung von c_ϕ , die Schleifenfragmente mit den in Abschnitt 6.3 bzw. Abschnitt 6.6 beschriebenen Verfahren Auszurollen bzw. zu Fusionieren.

[1] Die Kontrollflusskonstrukte zum Lesen des Kantenstatus und die Synchronisationsaktivitäten sind der Übersichtlichkeit wegen in *Snd Sync. Block* und *Rcv Sync. Block* gekapselt.

8.4. Zusammenfassung

In diesem Kapitel wurde die von Khalaf und Leymann [KL06] beschriebene Prozessfragmentierung erläutert, bei der aus einem einzelnen Prozessmodell eine Choreographie erzeugt wird, in der die Aktivitäten dieses Prozessmodells auf unterschiedliche interagierende Teilnehmerprozesse verteilt sind. Es wurde untersucht, ob die in dieser Arbeit beschriebene Konsolidierung die Umkehroperation zur Prozessfragmentierung ist. Dazu wurde analysiert, ob Prozessmodelle, die durch die Konsolidierung einer durch die Fragmentierung erzeugten Choreographie generiert wurden, verhaltensäquivalent zum ursprünglich unfragmentierten Prozessmodell sind. Es wurde festgestellt, dass dies nicht der Fall ist, da die Fragmentierung neben der Choreographie zusätzliche Koordinationsinformationen generiert, um zusammengehörige fragmentierte Aktivitäten zu identifizieren, die sich in unterschiedlichen Prozessmodellen befinden. Diese Informationen werden zusammen mit Anpassungen an der Workflow-Engine dazu genutzt, um die Choreographie verhaltensäquivalent zum ursprünglichen Prozessmodell auszuführen.

Die Prozesskonsolidierungsmethode hat zum Ziel, alle mit BPEL4Chor modellierten Choreographien zu konsolidieren, also auch solche, die nicht mit dem Fragmentierungsansatz von Khalaf erstellt wurden. Wäre die Methode speziell für die Konsolidierung der von Khahlaf fragmentierten Prozessmodelle entwickelt worden, wäre es nicht möglich gewesen, Aspekte wie die Isolation der Aktivitäten aus verschiedenen Teilnehmern in p_μ zu emulieren, was ihre Anwendbarkeit eingeschränkt hätte.

Die Konsolidierungsschritte können allerdings angepasst werden, um die fragmentierungsspezifischen Informationen von Khalaf zu berücksichtigen und damit aus der Choreographie ein zum ursprünglichen Prozessmodell verhaltensäquivalentes Prozessmodell zu generieren.

Kapitel 9

Prototypische Implementierung mittels Cuvée

Dieses Kapitel beschreibt das Tool *Cuvée*, das die Schritte zur Konsolidierung von interagierenden Prozessen aus Kapitel 5 und Kapitel 6 implementiert. Cuvée erhält als Eingabe eine BPEL4Chor-Choreographie [DKLW07] und generiert daraus ein einzelnes BPEL-Prozessmodell, das das Verhalten und die Interaktionen der Teilnehmer emuliert. Technisch ist das Tool als Standalone-Java-Anwendung implementiert, die entweder über die Kommandozeile oder eine Java-API genutzt werden kann. Abbildung 9.1 zeigt die Komponenten des Tools und von oben nach unten deren Ausführungsreihenfolge.

Im ersten Schritt wird die Choreographie von der *Reader-Komponente* eingelesen. Der *Container-Creator* erstellt dann das konsolidierte Prozessmodel mit den Teilnehmer-Scopes und die *Materialization-Komponente* materialisiert

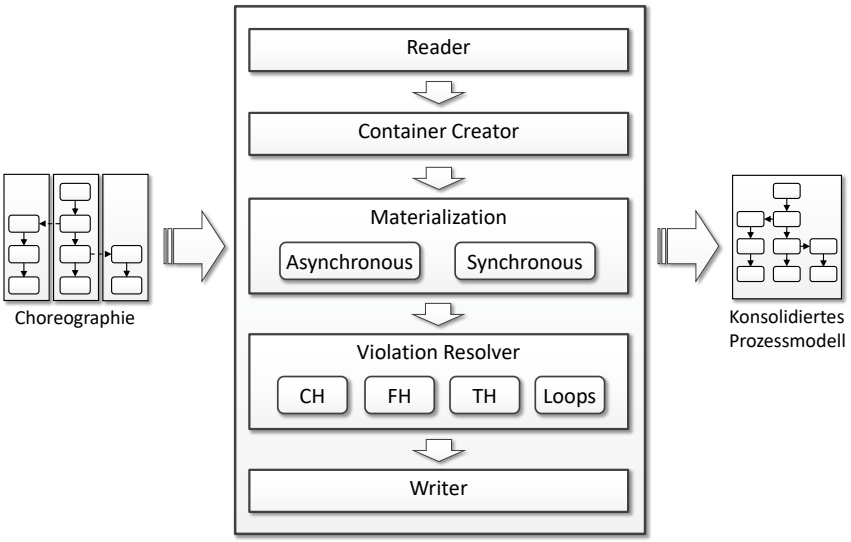

Abbildung 9.1.: Komponenten von Cuvée

die Basisinteraktionen. Der *Violation-Resolver* löst die potentiell durch die Materialisierung entstandenen grenzverletzenden Kontrollflusskanten auf. Die *Writer-Komponente* erstellt dann die BPEL-Datei und die WSDL-Dateien. Die Komponenten werden in den folgenden Abschnitten dieses Kapitels erläutert.

9.1. Reader

Der Reader-Komponente wird eine ZIP-Datei übergeben, die die unten beschriebenen Artefakte der zu konsolidieren BPEL4Chor-Choreographie enthalten muss. Die Komponente wandelt die Artefakte, die als XML-Dateien vorliegen, in ein Objektmodell[1] um. Diese Umwandlung wurde im Rahmen der Diplomarbeit von Cui [Cui12] implementiert.

[1] Das Modell basiert auf dem Eclispe Modeling Framework https://www.eclipse.org/modeling/emf/

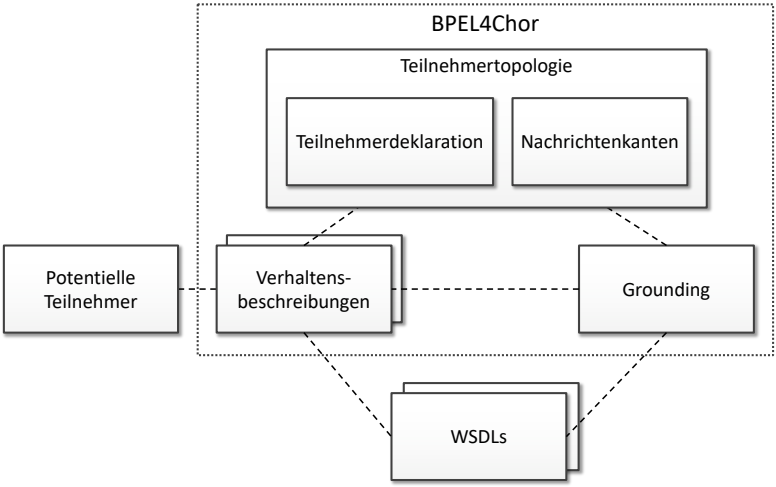

Abbildung 9.2.: Für die Konsolidierung notwendige Choreographieartefakte (adaptiert aus [DKLW07])

Zusätzlich normalisiert die Komponente die Verhaltensbeschreibungen der Choreographie. Die Normalisierung dient hauptsächlich dazu, Fallunterscheidungen bei der Materialisierung der Basisinteraktionen zu vermeiden (siehe Abschnitt 9.3). So werden zum Beispiel BPEL-Sequence-Aktivitäten durch Flows ersetzt, die über Kontrollflusskanten die sequentielle Ausführung der Kindaktivitäten der Sequence-Aktivitäten emulieren.

Die für die Konsolidierung benötigten Artefakte sind in Abbildung 9.2 dargestellt. Diese können zum Beispiel mit dem BPEL4Chor-Designer von Weiß et al. [WAS+13] modelliert werden.

Die *Verhaltensbeschreibungen* sind, wie in Abschnitt 3.1 erläutert, BPEL-Prozessmodelle, mit denen das öffentliche Verhalten der Teilnehmer modelliert wird. Diese Prozessmodelle müssen dem Syntaxprofil *Abstract Process Profile for Participant Behavior Descriptions* entsprechen [DKLW07]. Das Profil verlangt unter anderem, dass alle Kommunikationsaktivitäten einen eindeutigen Bezeichner besitzen und dass für sie keine technischen Details,

wie WSDL-PortTypen oder -Operationen angegeben werden. In der *Teilnehmertopologie* werden die Teilnehmer der Choreographie deklariert, die zur Entwurfszeit bekannt sind. Dabei wird für jeden Teilnehmer sein Typ, d.h. seine Verhaltensbeschreibung festgelegt. Die Topologie enthält außerdem die Teilnehmermengen, die in der Choreographie genutzt werden, sowie die Nachrichtenkanten, die die Kommunikationsaktivitäten miteinander verbinden.

Das *Grounding* enthält die technischen Details zu den WSDL-Schnittstellen, wie zum Beispiel PortTypes und Operationen, über die die Teilnehmer kommunizieren[1]. Das oben genannte Profil für die Verhaltensbeschreibungen erlaubt keine WSDL-spezifischen Informationen in den Kommunikationsaktivitäten. Der Prototyp hingegen akzeptiert für Kommunikationsaktivitäten, die die Kommunikation mit externen Services realisieren, die Spezifikation von WSDL-Informationen. Damit können auch Teilnehmer konsolidiert werden, deren Verhaltensbeschreibungen als ausführbare Prozessmodelle modelliert sind (siehe Abschnitt 9.5). Daher können die Verhaltensbeschreibungen ebenfalls auf *WSDL*-Dateien verweisen.

Wie oben erwähnt, müssen in der Teilnehmertopologie nur die Teilnehmer deklariert werden, die bereits zur Entwurfszeit bekannt sind. Da zum Erstellen der Teilnehmer-Container alle potentiellen Teilnehmer bekannt sein müssen (siehe Abschnitt 5.3), die an den Konversationen der Choreographie teilnehmen können, muss zusätzlich zur BPEL4Chor-Choreographie eine Liste angegeben werden (in Abbildung 9.2 *Potentielle Teilnehmer* genannt), die diese Teilnehmer und deren Typ deklariert.

9.2. Container-Creator

Der Container-Creator erstellt das Objektmodell des konsolidierten BPEL-Prozessmodells p_μ mitsamt den zugehörigen technischen Artefakten, wie

[1] Da das Grounding nur technische Informationen enthält, wurde es im Metamodell nicht berücksichtigt.

zum Beispiel WSDLs, Schemata usw. Analog zu dem in Abschnitt 5.3 beschriebenen Vorgehen, werden in p_μ für jeden potentiellen Teilnehmer entsprechende Teilnehmer-Scopes erstellt. In diesen Scopes werden wiederum die Duplikate des Kontroll- und Datenflusses aus der Verhaltensbeschreibung eingefügt, die den jeweiligen Teilnehmern zugewiesen sind.

9.3. Materialization

Die Materialisierung der Basisinteraktionen ist durch eine Menge von *Materialisierungsmustern* implementiert, die die Kontroll- und Datenflusskonstrukte in p_μ abhängig von der Art der Basisinteraktion und dem Kontrollflusskontext der Kommunikationsaktivitäten erzeugen. Die Muster lassen sich dabei, wie in Abbildung 9.3 skizziert, in verschiedene Kategorien unterteilen.

Neben den asynchronen, kann Cuvée auch die in dieser Ausarbeitung ausgeklammerten synchronen Varianten dieser Interaktionen materialisieren.

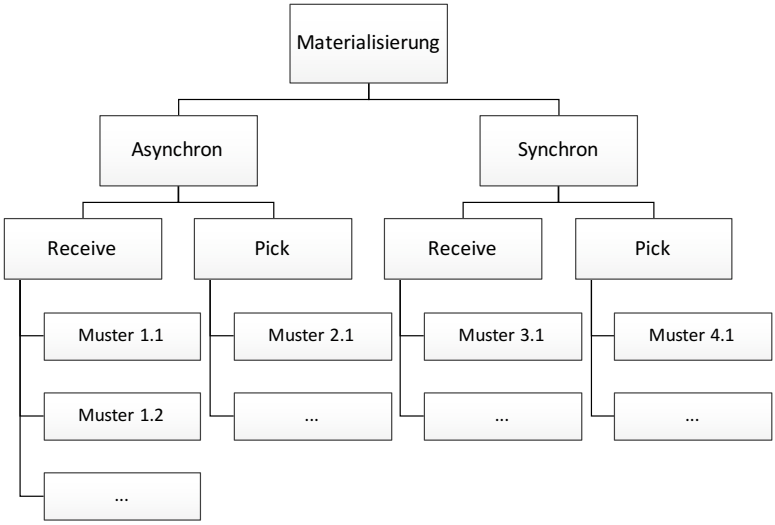

Abbildung 9.3.: Kategorisierung der Materialisierungsmuster

Bei diesen Varianten wird das Invoke, über das die Anfragenachricht gesendet wurde, erst beendet, wenn es eine Antwortnachricht von einer im Kontrollfluss auf das Receive bzw. auf das Pick folgenden Reply-Aktivität erhalten hat [OAS07b]. Formal ist die Materialisierung von synchronen Interaktionen in Wagner et al. [WBK+17] und auf Implementierungsebene in der Diplomarbeit von Debicki [Deb13] beschrieben. Zusätzlich können für asynchrone und synchrone Interaktionen Muster unterschieden werden, bei denen der Empfang der Anfragenachricht mittels eines Receives oder eines Picks modelliert werden kann. Die Blätter des Kategorisierungsbaums sind die eigentlichen Muster, die durch konkrete Klassen implementiert sind, die das Objektmodel von p_μ manipulieren. Welches Muster bzw. welche Klasse während der Materialisierung einer Basisinteraktion aufgerufen wird, hängt dabei vom Kontrollflusskontext der Kommunikationsaktivitäten ab, da dieser unterschiedliche Manipulationen des Objektmodels erfordert. Die Klasse, die das Muster 1.1 implementiert, wird zum Beispiel aufgerufen, wenn eine asynchrone Invoke-Receive Interaktion materialisiert werden soll, bei der das Invoke und das Receive eine Vorgänger- und eine Nachfolgeaktivität besitzen. Besitzen weder das Invoke noch das Receive eine Nachfolgeaktivität, wird die Klasse, die das Muster 1.4 implementiert, aufgerufen. Diese Klasse erzeugt keine Synchronisationsaktivitäten, da es in diesem Fall nicht notwendig ist, den durch die Interaktion implizierten Kontroll- bzw. Datenfluss zu emulieren. Die Muster dienen also auch dazu, dass in p_μ keine unnötigen Kontrollflusskonstrukte eingefügt werden. Eine komplette Übersicht dieser Muster ist ebenfalls in der Diplomarbeit von Debicki [Deb13] enthalten.

9.4. Violation-Resolver

Die Aufgabe des Violation-Resolvers ist es, mögliche Kontrollflussverletzungen in strukturierten Aktivitäten zu beheben, die durch die Materialisierung der Kontrollflusskanten entstehen können (vgl. Kapitel 6). Die Komponente kann dabei Kontrollflussverletzungen auflösen, die durch eingehende bzw.

ausgehende Kanten in Fault-, Termination- und Compensation-Handlern sowie ForEach- und While-Schleifen entstehen. Dazu wird der Kontrollfluss in p_μ vom Violation-Resolver so angepasst, dass das Verhalten der strukturierten Aktivität emuliert bzw. approximiert wird und gleichzeitig die Kontrollflussabhängigkeiten zwischen den Synchronisationsaktivitäten, die durch grenzverletzende Kontrollflusskanten abgebildet werden sollen, erhalten bleiben. Im folgenden wird eine kurze Übersicht über die Kontrollflussverletzungen gegeben, die vom Violation-Resolver aufgelöst werden können. Eine detaillierte Beschreibung findet sich in den jeweiligen studentischen Arbeiten.

Der Ansatz zum Auflösen von grenzverletzenden Kontrollflusskanten in Fault-Handlern wurde in der Diplomarbeit von Berger implementiert [Ber13]. Berger hat den Ansatz auf Termination-Handler, für die ebenfalls keine eingehenden Kontrollflusskanten zulässig sind, übertragen und den Prototypen entsprechend erweitert. Das Vorgehen zum Auflösen der grenzverletzenden Kontrollflusskanten ist analog zu dem bei Fault-Handlern, d.h. auch da wird die Logik des Termination-Handlers in einen separaten Scope transferiert.

Compensation-Handler erlauben weder eingehende noch ausgehende Kontrollflusskanten. Um Kontrollflussverletzungen zu eliminieren, die durch die Materialisierung von Interaktionen entstehen, bei denen sich Kommunikationsaktivitäten im Compensation-Handler eines Scopes befinden, wurde zusammen mit Panis [Pha15] ein entsprechender Ansatz entwickelt und in Cuvée umgesetzt. Die Grundidee dieses Ansatzes ist es, dass die Kompensationslogik in die Fault- bzw. Termination-Handler des jeweiligen Eltern-Scopes des betroffenen Scopes transferiert wird. Enthält ein Eltern-Scope mehrere Kind-Scopes, wird anhand des von Khalaf, Roller und Leymann [KRL09] vorgestellten *Compensation Order Graphs* die Ausführungsreihenfolge der Kompensationslogik der Kind-Scopes bestimmt und im Fault- und Termination-Handler entsprechend abgebildet.

In Abschnitt 6.3 wurde das Ausrollen von ForEach- bzw. While-Schleifen

diskutiert, um grenzverletzende Kontrollflusskanten zu eliminieren. Außerdem wurde an einem Beispiel beschrieben, wie zwei interagierende While-Schleifen fusioniert werden können. Beide Ansätze wurden von Kukillaya implementiert [Kuk17].

9.5. Writer

Diese Komponente wurde ebenfalls in der Diplomarbeit von Cui [Cui12] umgesetzt. Sie transformiert das Objektmodell von p_μ in ein abstraktes BPEL-Prozessmodell und die WSDL-Objektmodelle in entsprechende WSDL-Dateien. Das Prozessmodell p_μ folgt dann also dem Profil *Abstract Process Profile for Observable Behaviour* und enthält daher nur BPEL-Standardelemente, d.h. keine BPEL4Chor-Erweiterungen. Es kann somit mit den Regeln der „executable completion"' [OAS07b] in ein ausführbares BPEL-Prozessmodell transformiert werden.

Sind die Verhaltensbeschreibungen aller Teilnehmer bereits als ausführbare Prozesse modelliert, ist p_μ ebenfalls ausführbar. Es muss nur im Namespace von p_μ die URI des abstrakten BPEL-Profils, durch die von ausführbarem BPEL ersetzt werden. Ein ausführbares Prozessmodell enthält keine unterspezifizierten Elemente (Aktivitäten, Bedingungen, Variablen etc.) und für alle Aktivitäten, die mit externen Services kommunizieren, liegen die zur Kommunikation notwendigen Informationen in den Aktivitäten und den zugehörigen WSDL-Dateien vor.

9.6. Zusammenfassung

In diesem Kapitel wurden die Funktionen und die Architektur des Tools Cuvée beschrieben, mit dem eine BPEL4Chor-Choreographie in ein einzelnes Prozessmodell konsolidiert werden kann. Wie auch schon in den vorherigen Kapiteln erläutert, müssen alle potentiellen Teilnehmer der Choreographie bekannt sein. Es ist also nicht möglich, mit Cuvée Choreographien zu konso-

lidieren, in denen Teilnehmer erst zur Laufzeit, zum Beispiel von externen Web-Services, ermittelt werden. Cuvée unterstützt Verhaltensbeschreibungen, die mit abstrakten oder ausführbaren BPEL modelliert sind und ist damit in der Lage, auch ausführbare Prozessmodelle zu erzeugen. Dabei unterstützt Cuvée neben den in dieser Arbeit beschriebenen Interaktionen zusätzlich die Konsolidierung von Choreographien, deren Teilnehmer über Event-, Fault-, Termination- und Compensation-Handler miteinander kommunizieren.

Kapitel 10

Zusammenfassung und Ausblick

10.1. Zusammenfassung

Mittels Choreographien können die Interaktionen und das Verhalten von Teilnehmern spezifiziert werden. Die in dieser Ausarbeitung betrachteten Choreographien modellieren das Verhalten der Teilnehmer mittels Prozessmodellen. Die Interaktionen werden über den Austausch von Nachrichten zwischen deren Kommunikationsaktivitäten realisiert.

Organisatorischen Gründe, wie das Insourcing von Geschäftsbereichen oder technische Anforderungen, zum Beispiel der Verzicht auf nachrichtenbasierte Kommunikation aus Performanzgründen, können es nötig machen, das Verhalten der Teilnehmer über ein einziges Prozessmodell abzubilden. Dazu wurde in dieser Ausarbeitung als erster Forschungsbeitrag die *Prozesskonsolidierungsmethode* vorgestellt, mit der die Prozessmodelle, die das Teilnehmer-

verhalten einer Choreographie beschreiben, in ein einzelnes Prozessmodell konsolidiert werden können. Die Schritte der Prozesskonsolidierungsmethode wurden an einem in dieser Arbeit definierten Metamodell erläutert, das auf der Choreographiesprache BPEL4Chor [DKLW07] basiert.

Im ersten Schritt wird das konsolidierte Prozessmodell erstellt und es wird ihm der Kontrollfluss und der Datenkontext jedes Teilnehmers der Choreographie hinzugefügt. Dazu müssen alle potentiellen Teilnehmer einer Choreographie bekannt sein, auch wenn diese nicht an jeder Choreographieausführung teilnehmen müssen. Es können also keine Choreographien konsolidiert werden, deren Teilnehmer dynamisch zur Laufzeit bestimmt werden.

Die in der Choreographie modellierten Interaktionen implizieren durch den Nachrichtenaustausch Kontroll- und Datenflussabhängigkeiten zwischen den Aktivitäten der Teilnehmer. Im zweiten Schritt werden dem konsolidierten Prozessmodell Kontrollflusskonstrukte hinzugefügt, die dort die von Basisinteraktionen, also den Versand oder Empfang einer einzelnen Nachricht, implizierten Abhängigkeiten ohne die Verwendung von Kommunikationsaktivitäten emulieren. Im dritten Schritt werden im konsolidierten Prozessmodell darauf aufbauend auch komplexe Interaktionen, die mittels Schleifen modelliert sind und somit den Austausch von mehreren Nachrichten realisieren, emuliert.

Die Konsolidierung hat das Ziel, dass das konsolidierte Prozessmodell dasselbe Verhalten abbildet, das für die verschiedenen Teilnehmer in der originalen Choreographie spezifiziert wurde. Dies wurde mittels des zweiten Forschungsbeitrags *Verifikation über die Äquivalenz von Aktivitätszuständen* überprüft. Bei der Verifikation wird für jeden Konsolidierungsschritt überprüft, ob alle Geschäftsaktivitäten des konsolidierten Prozessmodells, also die Aktivitäten, die eine bestimmte Geschäftsfunktionen implementieren und kein Interaktionsverhalten umsetzen, dieselben Ausführungsreihenfolgen von Zustandstransitionen erzeugen können, wie in der originalen Choreographie. Grundlage der Verifikation bilden die Ausführungsreihenfolgen

von Zustandstransitionen, die zwischen zwei Aktivitäten bzw. deren Instanzen gelten, die eine direkte Kontrollflussbeziehung, zum Beispiel über eine gemeinsame Elternaktivität, haben. Diese Ausführungsreihenfolgen wurden als Teil des Forschungsbeitrags auf Basis der operationalen Semantik von BPEL4Chor und BPEL definiert. Um die Ausführungsreihenfolge während der Verifikation auch für Aktivitäten zu bestimmen, die eine indirekte Kontrollflussabhängigkeit besitzen, werden diese transitiv angewandt. Die Verifikation prüft nur, ob die Choreographie und das konsolidierte Prozessmodell dieselbe Menge an Ausführungsreihenfolgen von Zustandstransitionen generieren können, nicht aber, ob die Instanzen des Prozessmodells und die der Choreographie unter demselben Datenkontext die gleichen Ausführungsreihenfolgen erzeugen. Dies wurde daher zusätzlich bei der Diskussion der Konsolidierungsschritte informell untersucht.

Die Verifikation über die Ausführungsreihenfolgen von Zustandstransitionen hat gezeigt, dass die Geschäftsaktivitäten im konsolidierten Prozessmodell in der gleichen Reihenfolge ihre Zustände durchlaufen wie in der Choreographie. Die Materialisierung von Interaktionen kann aber dazu führen, dass im konsolidierten Prozessmodell einige Aktivitäten bestimmte Zustände verglichen mit der Choreographie zeitlich verzögert erreichen. Dies hat keinen Einfluss auf die Korrektheit des Kontroll- oder Datenflusses, führt aber dazu, dass zum Beispiel Geschäftsaktivitäten später ausgeführt, und folglich die Geschäftsziele später erreicht werden, als in der Choreographie.

Zwischen den Teilnehmern einer Choreographie können unterschiedliche Interaktionen modelliert werden, die sich in eine Menge von Interaktionsmustern kategorisieren lassen [BDtH05]. Ein Muster repräsentiert zum Beispiel Interaktionen, in denen ein Teilnehmer eine Nachricht an genau einen anderen Teilnehmer sendet, während ein anderes Muster Interaktionen umfasst, in denen ein Teilnehmer mehrere Nachrichten von unterschiedlichen Teilnehmern empfängt. Im Forschungsbeitrag *Validierung der Konsolidierung mittels Interaktionsmustern* wurde die Anwendbarkeit der Prozesskonsolidierung auf diese Muster untersucht. Dazu wurde erläutert, wie sich die Interaktionsmuster mit dem Metamodell modellieren und dann mit dem in dieser Arbeit

vorgestellten Ansatz konsolidieren lassen. Abgesehen von den Mustern „Atomic Multicast Notification“ und „Multi-Responses“ ist die Konsolidierung von allen Mustern möglich. Vom Muster „Atomic Multicast Notification“ kann das transaktionale Verhalten nicht mit dem Metamodell abgebildet werden kann. Das Muster „Multi-Responses“ kann zwar konsolidiert werden, allerdings nicht vollständig verhaltensäquivalent zur Choreographie. Wie oben erwähnt, müssen zur Konsolidierung aller Muster die potentiellen Teilnehmer der Choreographie bekannt sein.

Eine mögliche Umkehroperation zur Konsolidierung – die Fragmentierung von Prozessmodellen – wurde von Khalaf beschrieben [KL06; KL10; Kha08; KL12]. Bei dieser Operation werden die Aktivitäten des zu fragmentierenden Prozessmodells unterschiedlichen Teilnehmern zugeordnet. Die Fragmentierung erzeugt, basierend auf dieser Zuordnung eine Choreographie bestehend aus den Teilnehmern und deren Prozessmodellen, mit den dem jeweiligen Teilnehmer zugeordneten Aktivitäten. Um die Kontroll- und Datenflussbeziehungen beizubehalten, die zwischen den Aktivitäten im fragmentierten Prozessmodell modelliert wurden, spezifiziert die erstellte Choreographie die Kommunikationsreihenfolge zwischen den Prozessmodellen. Im Forschungsbeitrag *Wiederherstellung fragmentierter Prozessmodelle* wurde diskutiert, inwieweit die Prozesskonsolidierung die Umkehroperation zur Prozessfragmentierung ist. Dazu wurde untersucht, ob konsolidierte Prozessmodelle, die aus Choreographien erstellt wurden, die wiederum durch Fragmentierung erzeugt wurden, verhaltensäquivalent zu den fragmentierten Prozessmodellen sind. Der Fragmentierungsansatz erzeugt Prozessmodelle mit speziellen Spracherweiterungen, um Fragmente einer strukturierten Aktivität zu identifizieren, so dass eine Koordinations-Middleware die Zustände zwischen den Aktivitätsfragmenten synchronisieren kann. Wird eine Schleife fragmentiert, weil ihre Kindaktivitäten verschiedenen Teilnehmern zugeordnet werden, müssen zum Beispiel die Iterationen der Schleifenfragmente in den verschiedenen Prozessen synchronisiert werden. Die Prozesskonsolidierung nutzt hingegen ein Metamodell, das auf Standard-BPEL bzw. BPEL4Chor basiert. Sie kann somit zusammengehörige Aktivitätsfragmente und ihre Kindakti-

vitäten nicht identifizieren, um diese wieder in einer einzelne Aktivität zusammenführen. Es ist daher nicht möglich, ein konsolidiertes Prozessmodell zu erzeugen, das verhaltensäquivalent zum fragmentierten Prozessmodell ist.

Der Beitrag *prototypische Implementierung der Konsolidierung* beschreibt das entwickelte Tool Cuvée, mit dem in BPEL4Chor modellierte Choreographien mittels der in dieser Arbeit beschriebenen Methode konsolidiert werden können [Deb13; Kuk17]. Das Tool implementiert alle Konsolidierungsschritte. Zusätzlich können mit Cuvée auch Choreographieren konsolidiert werden, deren Teilnehmer über Fault-, Termination- und Compensation-Handler kommunizieren [Ber13; Pha15]. Über Handler-kommunizierende Choreographien wurden in dieser Ausarbeitung nicht diskutiert, da sie für die Konsolidierung der Interaktionsmuster nicht relevant sind.

10.2. Ausblick

BPEL4Chor wurde als Grundlage für das Metamodell und die Konsolidierung verwendet, da es die Anforderungen an eine Choreographiesprache aus [KLW11] erfüllt. BPMN bietet mit Kollaborationsdiagrammen einen zu BPEL4Chor [KLW11; DKL+08] analogen Modellierungsansatz für Choreographien. Mit diesen Diagrammen kann das Teilnehmerverhalten ebenfalls mit Prozessmodellen und die Teilnehmerinteraktionen durch Nachrichtenaustausch über Kommunikationsaktivitäten modelliert werden. Aufgrund der Popularität von BPMN [Har16] sollte in zukünftigen Arbeiten geprüft werden, inwieweit sich die Schritte der Konsolidierungsmethode auf BPMN Kollaborationsdiagramme anwenden lassen.

Das Metamodell und die Konsolidierung berücksichtigen nicht alle Sprachkonstrukte von BPEL4Chor, mit denen die Interaktionsmuster implementiert werden können. In BPEL4Chor kann zum Beispiel das Interaktionsmuster „Send/Receive“ zusätzlich über eine synchrone Sendeaktivität implementiert werden. Ein Ansatz zur Konsolidierung von synchronen Interaktionen wurde

bereits von Wagner, Kopp und Leymann in [WKL12] beschrieben und müsste auf das hier verwendete Metamodell angepasst werden. Dasselbe gilt für die Kommunikation über Event-Handler, mit der zum Beispiel das Muster „Racing Incoming Messages“ implementiert werden kann. Die Grundlagen der Konsolidierung von über Event-Handler kommunizierenden Teilnehmern wurden in der Masterarbeit von Milutinovic [Mil14] diskutiert.

In Abschnitt 6.6 wurde ein Ansatz zur Fusion von zwei kommunizierenden Schleifen beschrieben, deren maximale Anzahl an Iterationen nicht bestimmt und die somit nicht ausgerollt werden können. Der Ansatz hat gegenüber dem Ausrollen von Schleifen den Nachteil, dass die fusionierte Schleife, die die Schleifenkörper der beiden Schleifen enthält, nicht alle Fehlerszenarien der originalen Choreographie emulieren kann. Daher sollte das Ausrollen von Schleifen immer der Fusion vorgezogen werden. Um dies auch für While-Schleifen zu ermöglichen, wo die maximale Iterationsanzahl implizit im Kontrollfluss kodiert ist, können die Datenflussanalysetechniken von Heinze, Amme und Moser [HAM12] sowie von Monakova et al.[MKL+09] genutzt werden. Dies würde auch eine verhaltensäquivalente Konsolidierung des oben erwähnten Musters „Multi-Responses“ ermöglichen.

Wie bereits in der Zusammenfassung erläutert, kann die Prozesskonsolidierung ein mit dem Ansatz von Khalaf und Leymann [KL06; KL10] fragmentiertes Prozessmodell nicht verhaltensäquivalent wiederherstellen. Dazu muss sie in zukünftigen Arbeiten so erweitert werden, dass sie fragmentierte Aktivitäten und deren Kindaktivitäten identifiziert, um diese wieder in eine einzelne Aktivität zusammenführen zu können. Die notwendigen Erweiterungen wurden bereits in dieser Ausarbeitung skizziert.

Aktivitäten in Choreographien können unterschiedliche nicht-funktionale Anforderungen an die Ausführungsumgebung, zum Beispiel an deren Verfügbarkeit, haben. Geschäftskritische Aktivitäten können beispielsweise in hochverfügbaren Umgebungen und Aktivitäten mit niedrigeren Verfügbarkeitsanforderungen in günstigeren und weniger zuverlässigen Umgebungen ausgeführt werden. Wagner et al. [WFKS12] haben dazu ein Konzept be-

schrieben, wie die Prozesskonsolidierung und -fragmentierung zusammen genutzt werden können, eine Choreographie c_{org} in eine Choreographie c_{depl} zu transformieren, in der die Aktivitäten abhängig von ihren nicht-funktionalen Anforderungen den jeweiligen Laufzeitumgebungen zugeordnet werden, auf denen sie provisioniert werden sollen. In c_{depl} entsprechen die Laufzeitumgebungen also den Teilnehmern. Während mittels der Fragmentierung die Verteilung der Aktivitäten aus c_{org} auf verschiedene Teilnehmer in c_{depl} realisiert wird, werden mit der Konsolidierung alle Aktivitäten, die der gleichen Laufzeitumgebung zugeordnet sind, in ein einzelnes Prozessmodell zusammengefasst. In zukünftigen Arbeiten sollten die existierenden Tools zur Fragmentierung und Konsolidierung von Prozessmodellen integriert werden, um den Ansatz zur Erstellung von c_{depl} holistisch umzusetzen. Dabei sollte Prozessverantwortlichen weiterhin die Überwachung der originalen Choreographie c_{org} ermöglicht werden. Um dies zu erreichen, müssen die integrierten Tools, wie von Wagner et al. [WFKS12] beschrieben, Regeln generieren, mit denen ein Monitoring-Tool (zum Beispiel das von Wetzstein [WKK+10]) basierend auf dem Ausführungszustand von c_{depl} den Zustand von c_{org} ableiten kann.

Literaturverzeichnis

[AA12] Y. Aït-Ameur und I. Aït-Sadoune. „Stepwise Development of Formal Models for Web Services Compositions: Modelling and Property Verification“. In: *Database and Expert Systems Applications - 23rd International Conference, DEXA 2012, Vienna, Austria, September 3-6, 2012. Proceedings, Part I*. Hrsg. von S. W. Liddle, K.-D. Schewe, A. M. Tjoa und X. Zhou. Bd. 7446. Lecture Notes in Computer Science. Springer, 2012, S. 9 (Zitiert auf S. 40).

[Abr10] J.-R. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010 (Zitiert auf S. 40).

[AFK04] J. Arias-Fisteus, L. S. Fernández und C. D. Kloos. „Formal Verification of BPEL4WS Business Collaborations“. In: *E-Commerce and Web Technologies, 5th International Conference, EC-Web 2004, Zaragoza, Spain, August 31-September 3, 2004, Proceedings*. Hrsg. von K. Bauknecht, M. Bichler und B. Pröll. Bd. 3182. Lecture Notes in Computer Science. Springer, 2004, S. 76–85 (Zitiert auf S. 39).

[AFK05] J. Arias-Fisteus, L. S. Fernández und C. D. Kloos. „Applying model checking to BPEL4WS business collaborations“. In: *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC), Santa Fe, New Mexico, USA, March 13-17, 2005*. Hrsg. von H. Haddad, L. M. Liebrock, A. Omicini und R. L. Wainwright. ACM, 2005, S. 826–830 (Zitiert auf S. 39).

[AM08] F. Abouzaid und J. Mullins. „A Calculus for Generation, Verification and Refinement of BPEL Specifications“. In: Bd. 200. 3. 2008, S. 43–65 (Zitiert auf S. 35).

[Bae73] J.-L. Baer. „A Survey of Some Theoretical Aspects of Multiprocessing“. In: *ACM Comput. Surv.* 5.1 (1973), S. 31–80 (Zitiert auf S. 160).

[BAP17] G. Babin, Y. A. Ameur und M. Pantel. „Web Service Compensation at Runtime: Formal Modeling and Verification Using the Event-B Refinement and Proof Based Formal Method“. In: *IEEE Trans. Serv. Comput.* 10.1 (2017), S. 107–120 (Zitiert auf S. 40).

[BBB+11] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen und J. Sifakis. „Rigorous Component-Based System Design Using the BIP Framework“. In: *IEEE Softw.* 28.3 (2011), S. 41–48 (Zitiert auf S. 40).

[BBGK17] M. C. Beutel, V. Borozanov, S. Gökay und K.-H. Krempels. „Semi-automated Business Process Model Matching and Merging Considering Advanced Modeling Constraints“. In: *ICEIS 2017 - Proceedings of the 19th International Conference on Enterprise Information Systems, Volume 1, Porto, Portugal, April 26-29, 2017*. Hrsg. von S. Hammoudi, M. Smialek, O. Camp und J. Filipe. SciTePress, 2017, S. 324–331 (Zitiert auf S. 43).

[BCE+06] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu und M. Sabetzadeh. „A Manifesto for Model Merging“. In: *Proceedings of the 2006 International Workshop on Global Integrated Model Management*. GaMMa '06. Shanghai, China: Association for Computing Machinery, 2006, S. 5–12 (Zitiert auf S. 42).

[BDtH05] A. Barros, M. Dumas und A. ter Hofstede. „Service Interaction Patterns“. In: *BPM*. Springer, 2005 (Zitiert auf S. 23, 25, 31, 32, 43, 62, 136, 277, 278, 289, 295, 299, 333).

[Ber13] P. Berger. „Konsolidierung von BPEL-Prozessmodellen im Kontext von Interaktionen über Fehlerbehandlungskontrukte“. Deutsch. Diplomarbeit. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Nov. 2013, S. 83 (Zitiert auf S. 223, 327, 335).

[BF04] M. J. Butler und C. Ferreira. „An Operational Semantics for StAC, a Language for Modelling Long-Running Business Transactions“. In: *Coordination Models and Languages, 6th International Conference, COORDINATION 2004, Pisa, Italy, February 24-27, 2004, Proceedings*. Hrsg. von R. D. Nicola, G. L. Ferrari und G. Meredith. Bd. 2949. Lecture Notes in Computer Science. Springer, 2004, S. 87–104 (Zitiert auf S. 36).

[BFN05] M. J. Butler, C. Ferreira und M. Y. Ng. „Precise Modelling of Compensating Business Transactions and its Application to BPEL“. In: *J. Univers. Comput. Sci.* 11.5 (2005), S. 712–743 (Zitiert auf S. 36).

[BGVC20] R. Belchior, S. Guerreiro, A. Vasconcelos und M. Correia. „A Survey on Business Process View Integration“. In: *CoRR* abs/2011.14465 (2020). arXiv: 2011.14465 (Zitiert auf S. 43).

[BLN86] C. Batini, M. Lenzerini und S. B. Navathe. „A Comparative Analysis of Methodologies for Database Schema Integration“. In: *ACM Comput. Surv.* 18.4 (1986), S. 323–364 (Zitiert auf S. 47).

[BMB20] L. Boumlik, M. Mejri und H. Boucheneb. „Toward a Formalization of BPEL 2.0 : An Algebra Approach“. In: *International Journal on Web Service Computing* 11 (März 2020), S. 1–20 (Zitiert auf S. 32, 37, 40).

[BS03] E. Börger und R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003 (Zitiert auf S. 38).

[Che77] P. P. Chen. „The entity-relationship model: a basis for the enterprise view of data“. In: *American Federation of Information Processing Societies: 1977 National Computer Conference, June 13-16, 1977, Dallas, Texas, USA*. Bd. 46. AFIPS Conference Proceedings. AFIPS Press, 1977, S. 77–84 (Zitiert auf S. 47).

[CLS+05] F. Curbera, F. Leymann, T. Storey, D. Ferguson und S. Weerawarana. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, 2005 (Zitiert auf S. 157).

[CMNR06] J. S. Cardoso, J. Mendling, G. Neumann und H. A. Reijers. „A Discourse on Complexity of Process Models“. In: *Business Process Management Workshops, BPM 2006 International Workshops, BPD, BPI, ENEI, GPWW, DPM, semantics4ws, Vienna, Austria, September 4-7, 2006, Proceedings*. Hrsg. von J. Eder und S. Dustdar. Bd. 4103. Lecture Notes in Computer Science. Springer, 2006, S. 117–128 (Zitiert auf S. 173).

[CS17] J. Colaço und P. Sousa. „View Integration of Business Process Models“. In: *Information Systems - 14th European, Mediterranean, and Middle Eastern Conference, EMCIS 2017, Coimbra, Portugal, September 7-8, 2017, Proceedings*. Hrsg. von M. Themistocleous und V. Morabito. Bd. 299. Lecture Notes in Business Information Processing. Springer, 2017, S. 619–632 (Zitiert auf S. 46).

[CS20] D. Cardoso und P. Sousa. „Generation of Stakeholder-Specific BPMN Models“. In: *Advances in Enterprise Engineering XIII*. Hrsg. von A. David, G. Guizzardi und J. Borbinha. Springer International Publishing, 2020, S. 15–32 (Zitiert auf S. 46).

[Cui12] D. Cui. „Splitting BPEL Processes“. Diplomarbeit. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Mai 2012, S. 99 (Zitiert auf S. 303, 322, 328).

[Dar99] A. Darte. „On the complexity of loop fusion“. In: *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*. 1999, S. 149–157 (Zitiert auf S. 270).

[DBLL04] Z. Duan, A. J. Bernstein, P. M. Lewis und S. Lu. „A model for abstract process specification, verification and composition“. In: *Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, NY, USA, November 15-19, 2004, Proceedings*. Hrsg. von M. Aiello, M. Aoyama, F. Curbera und M. P. Papazoglou. ACM, 2004, S. 232–241 (Zitiert auf S. 37).

[DD04] R. Dijkman und M. Dumas. „Service-oriented Design: A Multi-viewpoint Approach“. In: *International Journal of Cooperative Information Systems* 13.4 (2004), S. 337–368 (Zitiert auf S. 30).

[Deb13] P. Debicki. „Choreographie-basierte Konsoliedierung von BPEL-Prozessmodellen“. German. Diploma Thesis. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, und Information Technology, Germany, Feb. 2013, S. 112 (Zitiert auf S. 326, 335).

[DKB08] G. Decker, O. Kopp und A. P. Barros. „An Introduction to Service Choreographies (Servicechoreographien - eine Einführung)“. In: *it - Information Technology* 50.2 (2008), S. 122–127 (Zitiert auf S. 30, 49).

[DKL+08] G. Decker, O. Kopp, F. Leymann, K. Pfitzner und M. Weske. „Modeling Service Choreographies using BPMN and BPEL4Chor“. In: *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE '08)*. Bd. 5074. Lecture Notes in Computer Science. Springer-Verlag, Juni 2008, S. 79–93 (Zitiert auf S. 335).

[DKLW07] G. Decker, O. Kopp, F. Leymann und M. Weske. „BPEL4Chor: Extending BPEL for Modeling Choreographies“. In: *2007 IEEE International Conference on Web Services (ICWS 2007), July 9-13, 2007, Salt Lake City, Utah, USA*. IEEE Computer Society, 2007, S. 296–303 (Zitiert auf S. 49, 52, 321, 323, 332).

[DKLW09] G. Decker, O. Kopp, F. Leymann und M. Weske. „Interacting services: From specification to execution“. In: *Data & Knowledge Engineering* 68.10 (Apr. 2009), S. 946–972 (Zitiert auf S. 19, 30, 31, 295).

[DW11] G. Decker und M. Weske. „Interaction-centric modeling of process choreographies“. In: *Inf. Syst.* 36.2 (2011), S. 292–312 (Zitiert auf S. 31).

[DWM99] I. Düntsch, H. Wang und S. McCloskey. „Relations Algebras in Qualitative Spatial Reasoning“. In: *Fundamenta Informatiae* 39 (Aug. 1999), S. 229–248 (Zitiert auf S. 41).

[ELS+10] H. Eberle, F. Leymann, D. Schleicher, D. Schumm und T. Unger. „Process Fragment Composition Operations“. In: *5th IEEE Asia-Pacific Services Computing Conference, APSCC 2010, 6-10 December 2010, Hangzhou, China, Proceedings*. IEEE Computer Society, 2010, S. 157–163 (Zitiert auf S. 46, 72).

[ELU10] H. Eberle, F. Leymann und T. Unger. „Implementation Architectures for Adaptive Workflow Management“. In: *ADAPTIVE 2010*. Xpert Publishing Services, Nov. 2010, S. 1–6 (Zitiert auf S. 46).

[Fad04] M. Fadlisyah. „Using the π-Calculus for Modeling and Verifying Processes on Web Services“. Master Thesis. Dresden University of Technology, Germany, Okt. 2004 (Zitiert auf S. 34).

[Fah05] D. Fahland. *Complete Abstract Operational Semantics for the Web Service Business Process Execution Language*. Techn. Ber. 2005 (Zitiert auf S. 38).

[FBS04a] X. Fu, T. Bultan und J. Su. „Analysis of interacting BPEL web services“. In: *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*. Hrsg. von S. I. Feldman, M. Uretsky, M. Najork und C. E. Wills. ACM, 2004, S. 621–630 (Zitiert auf S. 38).

[FBS04b] X. Fu, T. Bultan und J. Su. „WSAT: A Tool for Formal Analysis of Web Services“. In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*. Hrsg. von R. Alur und D. A. Peled. Bd. 3114. Lecture Notes in Computer Science. Springer, 2004, S. 510–514 (Zitiert auf S. 38).

[Fer04] A. Ferrara. „Web services: a process algebra approach“. In: *Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, NY, USA, November 15-19, 2004, Proceedings*. Hrsg. von M. Aiello, M. Aoyama, F. Curbera und M. P. Papazoglou. ACM, 2004, S. 242–251 (Zitiert auf S. 35).

[FGV05] R. Farahbod, U. Glässer und M. Vajihollahi. „A Formal Semantics for the Business Process Execution Language for Web Services“. In: *Web Services and Model-Driven Enterprise Information Services, Proceedings of the Joint Workshop on Web Services and Model-Driven Enterprise Information Services, WSMDEIS 2005, In conjunction with ICEIS 2005, Miami, USA, May 2005*. Hrsg. von S. Bevinakoppa, L. F. Pires und S. Hammoudi. INSTICC Press, 2005, S. 122–133 (Zitiert auf S. 38).

[FR05] D. Fahland und W. Reisig. „ASM-based Semantics for BPEL: The Negative Control Flow“. In: *Proceedings of the 12th International Workshop on Abstract State Machines, ASM 2005, March 8-11, 2005, Paris, France*. 2005, S. 131–152 (Zitiert auf S. 38).

[FUMK03] H. Foster, S. Uchitel, J. Magee und J. Kramer. „Model-based Verification of Web Service Compositions“. In: *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*. IEEE Computer Society, 2003, S. 152–163 (Zitiert auf S. 34).

[FUMK05] H. Foster, S. Uchitel, J. Magee und J. Kramer. „Tool Support for Model-Based Engineering of Web Service Compositions“. In: *2005 IEEE International Conference on Web Services (ICWS 2005), 11-15 July 2005, Orlando, FL, USA*. IEEE Computer Society, 2005, S. 95–102 (Zitiert auf S. 34).

[GLC13] K. Görlach, F. Leymann und V. Claus. „Unified Execution of Service Compositions“. In: *Proceedings of the 6th IEEE International Conference on Service Oriented Computing & Applications (SOCA 2013)*. IEEE Computer Society Conference Publishing Services, Dez. 2013, S. 162–167 (Zitiert auf S. 47).

[Gur18] Y. Gurevich. „Evolving Algebras 1993: Lipari Guide“. In: *CoRR* abs/1808.06255 (2018). arXiv: 1808.06255 (Zitiert auf S. 38).

[GvdAJ08] F. Gottschalk, W. M. P. van der Aalst und M. H. Jansen-Vullers. „Merging Event-Driven Process Chains“. In: *On the Move to Meaningful Internet Systems: OTM 2008, OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008, Monterrey, Mexico, November 9-14, 2008, Proceedings, Part I*. Hrsg. von R. Meersman und Z. Tari. Bd. 5331. Lecture Notes in Computer Science. Springer, 2008, S. 418–426 (Zitiert auf S. 44).

[Gyn94] P. Gyngell. „Reengineering the corporation: A manifesto for business revolution: Michael Hammer and James Champy HarperCollins Publishers USA published in UK by Nicholas Brealey Publishing London (1993) 216 pp“. In: *J. Strateg. Inf. Syst.* 3.4 (1994), S. 339–345 (Zitiert auf S. 15).

[HAM12] T. Heinze, W. Amme und S. Moser. „Control flow Unfolding of Workflow Graphs Using Predicate Analysis and SMT Solving“. In: *ZEUS*. 2012 (Zitiert auf S. 232, 336).

[Har16] P. Harmon. *The State of Business Process Management 2016*. März 2016 (Zitiert auf S. 335).

[HAS21] T. S. Heinze, W. Amme und A. Schäfer. „Detecting Semantic Business Process Model Clones“. In: *Proceedings of the 13th European Workshop on Services and their Composition (ZEUS 2021), Bamberg, Germany, February 25-26, 2021*. Hrsg. von J. Manner, S. Haarmann, S. Kolb, N. Herzberg und O. Kopp. Bd. 2839. CEUR Workshop Proceedings. CEUR-WS.org, 2021, S. 25–28 (Zitiert auf S. 43).

[HDvdA+05] J. Hidders, M. Dumas, W. M. P. van der Aalst, A. H. M. ter Hofstede und J. Verelst. „When are two Workflows the Same?“ In: *Theory of Computing 2005, Eleventh CATS 2005, Computing: The Australasian Theory Symposium, Newcastle, NSW, Australia, January/February 2005*. Hrsg. von M. D. Atkinson und F. K. H. A. Dehne. Bd. 41. CRPIT. Australian Computer Society, 2005, S. 3–11 (Zitiert auf S. 31, 34).

[HKM06] T. Hornung, A. Koschmider und J. Mendling. „Integration of heterogeneous BPM Schemas: The Case of XPDL and BPEL“. In: *The 18th Conference on Advanced Information Systems Engineering (CAiSE '06), Forum Proceedings, Theme: Trusted Information Systems, Luxembourg, June 5-9, 2006*. Hrsg. von N. Boudjlida, D. Cheng und N. Guelfi. Bd. 231. CEUR Workshop Proceedings. CEUR-WS.org, 2006 (Zitiert auf S. 47).

[Hoa69] C. A. R. Hoare. „An Axiomatic Basis for Computer Programming“. In: *Commun. ACM* 12.10 (1969), S. 576–580 (Zitiert auf S. 37).

[Hoa78] C. A. R. Hoare. „Communicating Sequential Processes“. In: *Commun. ACM* 21.8 (1978), S. 666–677 (Zitiert auf S. 36).

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985 (Zitiert auf S. 37, 152).

[HOH17] P. Hartman, J. Ogden und B. Hazen. „Bring it back? An examination of the insourcing decision“. In: *International Journal of Physical Distribution & Logistics Management* 47 (Jan. 2017) (Zitiert auf S. 19).

[Hol93] G. J. Holzmann. „Design and Validation of Protocols: A Tutorial“. In: *Comput. Networks ISDN Syst.* 25.9 (1993), S. 981–1017 (Zitiert auf S. 38).

[Hol97] G. J. Holzmann. „The Model Checker SPIN“. In: *IEEE Trans. Software Eng.* 23.5 (1997), S. 279–295 (Zitiert auf S. 38).

[HSS05] S. Hinz, K. Schmidt und C. Stahl. „Transforming BPEL to Petri Nets“. In: *Business Process Management, 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005, Proceedings*. Hrsg. von W. M. P. van der Aalst, B. Benatallah, F. Casati und F. Curbera. Bd. 3649. 2005, S. 220–235 (Zitiert auf S. 33).

[KELU10] O. Kopp, H. Eberle, F. Leymann und T. Unger. „The Subprocess Spectrum“. In: *Proceedings of the Business Process and Services Computing Conference: BPSC 2010*. GI e.V., 2010 (Zitiert auf S. 16).

[KEvL+10] O. Kopp, L. Engler, T. van Lessen, F. Leymann und J. Nitzsche. „Interaction Choreography Models in BPEL: Choreographies on the Enterprise Service Bus“. In: *Subject-Oriented Business Process Management - Second International Conference, S-BPM ONE 2010, Karlsruhe, Germany, October 14, 2010. Selected Papers*. Hrsg. von A. Fleischmann, W. Schmidt, R. Singer und D. Seese. Bd. 138. Communications in Computer and Information Science. Springer, 2010, S. 36–53 (Zitiert auf S. 31).

[KGFE08a] J. Küster, C. Gerth, A. Förster und G. Engels. „A Tool for Process Merging in Business-Driven Development“. In: *Proceedings of the Forum at the CAiSE*. 2008 (Zitiert auf S. 43).

[KGFE08b] J. M. Küster, C. Gerth, A. Förster und G. Engels. „Detecting and Resolving Process Model Differences in the Absence of a Change Log“. In: *Business Process Management, 6th International Conference, BPM 2008, Milan, Italy, September 2-4, 2008. Proceedings*. Hrsg. von M. Dumas, M. Reichert und M.-C. Shan. Bd. 5240. Lecture Notes in Computer Science. Springer, 2008, S. 244–260 (Zitiert auf S. 43).

[Kha08] R. Khalaf. „Supporting business process fragmentation while maintaining operational semantics: a BPEL perspective“. Doctoral Thesis. University of Stuttgart, Faculty of Computer Science, Electrical Engi-

neering, und Information Technology, Germany, 2008 (Zitiert auf S. 301, 303, 334).

[KHK+11] O. Kopp, S. Henke, D. Karastoyanova, R. Khalaf, F. Leymann, M. Sonntag, T. Steinmetz, T. Unger und B. Wetzstein. *An Event Model for WS-BPEL 2.0*. Technischer Bericht Informatik 2011/07. Universität Stuttgart, Institut für Architektur von Anwendungssystemen: Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Sep. 2011, S. 55 (Zitiert auf S. 32, 84, 99).

[KI14] A. Kheldoun und M. Ioualalen. „Transformation BPEL Processes to RECATNet for Analysing Web Services Compositions“. In: *MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7 - 9 January, 2014*. Hrsg. von L. F. Pires, S. Hammoudi, J. Filipe und R. C. das Neves. SciTePress, 2014, S. 425–430 (Zitiert auf S. 33).

[KKL08] R. Khalaf, O. Kopp und F. Leymann. „Maintaining Data Dependencies Across BPEL Process Fragments“. In: *International Journal of Cooperative Information Systems (IJCIS)* 17.3 (Sep. 2008), S. 259–282 (Zitiert auf S. 304).

[KL03] R. Khalaf und F. Leymann. „On Web Services Aggregation“. In: *Technologies for E-Services, 4th International Workshop, TES 2003, Berlin, Germany, September 8, 2003, Proceedings*. Hrsg. von B. Benatallah und M.-C. Shan. Bd. 2819. Lecture Notes in Computer Science. Springer, 2003, S. 1–13 (Zitiert auf S. 30).

[KL06] R. Khalaf und F. Leymann. „Role-based Decomposition of Business Processes using BPEL“. In: *ICWS 2006*. IEEE, 2006 (Zitiert auf S. 17, 24, 25, 301, 307, 320, 334, 336).

[KL10] R. Khalaf und F. Leymann. „Coordination for Fragmented Loops and Scopes in a Distributed Business Process“. In: *BPM*. Springer, 2010 (Zitiert auf S. 301, 304, 306, 334, 336).

[KL12] R. Khalaf und F. Leymann. „Coordination for fragmented loops and scopes in a distributed business process“. In: *Inf. Syst.* 37.6 (2012), S. 593–610 (Zitiert auf S. 301, 306, 334).

[KLW11] O. Kopp, F. Leymann und S. Wagner. „Modeling Choreogaphies: BPMN 2.0 versus BPEL-based Approaches“. In: *Proceedings of the 4th International Workshop on Enterprise Modelling and Information Systems Architectures — EMISA 2011*. Lecture Notes in Informatics (LNI). Gesellschaft für Informatik e.V. (GI), 2011 (Zitiert auf S. 31, 335).

[KM93] K. Kennedy und K. S. McKinley. „Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution.“ In: *LCPC*. 1993, S. 301–320 (Zitiert auf S. 233).

[KML08] O. Kopp, R. Mietzner und F. Leymann. *Abstract Syntax of WS-BPEL 2.0*. Techn. Ber. University of Stuttgart, Sep. 2008, S. 75 (Zitiert auf S. 50, 57, 58, 60).

[KMWL09] O. Kopp, D. Martin, D. Wutke und F. Leymann. „The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages“. In: *Enterprise Modelling and Information Systems* 4.1 (Juni 2009). Hrsg. von U. Frank, S. 3–13 (Zitiert auf S. 56).

[Kop16a] O. Kopp. „Partnerübergreifende Geschäftsprozesse und ihre Realisierung in BPEL“. Deutsch. Dissertation. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Feb. 2016, S. 242 (Zitiert auf S. 31, 157, 159, 277, 282, 293).

[Kop16b] O. Kopp. „Partnerübergreifende Geschäftsprozesse und ihre Realisierung in BPEL“. Diss. University of Stuttgart, 2016 (Zitiert auf S. 49).

[KRL09] R. Khalaf, D. Roller und F. Leymann. „Revisiting the Behavior of Fault and Compensation Handlers in WS-BPEL“. In: *On the Move to Meaningful Internet Systems: OTM 2009, Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009, Vilamoura, Portugal, November 1-6, 2009, Proceedings, Part I*. Hrsg. von R. Meersman, T. S. Dillon und P. Herrero. Bd. 5870. Lecture Notes in Computer Science. Springer, 2009, S. 286–303 (Zitiert auf S. 327).

[Kuk17] S. V. Kukillaya. „Choreography-based Consolidation of Interacting BPEL Processes having Loops“. Masterarbeit. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, März 2017, S. 87 (Zitiert auf S. 222, 328, 335).

[KvB04] M. Koshkina und F. van Breugel. „Modelling and verifying web service orchestration by means of the concurrency workbench“. In: *ACM SIGSOFT Softw. Eng. Notes* 29.5 (2004), S. 1–10 (Zitiert auf S. 35).

[KW17] C. Klinkmüller und I. Weber. „Control Flow Information Analysis in Process Model Matching Techniques“. In: *CoRR* abs/1707.01089 (2017). arXiv: 1707.01089 (Zitiert auf S. 43).

[KYYH17] J. Kunchala, J. Yu, S. Yongchareon und Y. Han. „Towards merging collaborating processes for artifact lifecycle synthesis“. In: *Proceedings of the Australasian Computer Science Week Multiconference, ACSW 2017, Geelong, Australia, January 31 - February 3, 2017*. ACM, 2017, 50:1–50:8 (Zitiert auf S. 42).

[KYYL20] J. Kunchala, J. Yu, S. Yongchareon und C. Liu. „An approach to merge collaborating processes of an inter-organizational business process for artifact lifecycle synthesis“. In: *Computing* 102.4 (2020), S. 951–976 (Zitiert auf S. 42).

[LA94] F. Leymann und W. Altenhuber. „Managing business processes as an information resource“. In: *IBM Syst. J.* 33.2 (1994), S. 326–348 (Zitiert auf S. 16, 86).

[Ley01] F. Leymann. *Web Services Flow Language (WSFL 1.0)*. IBM Software Group. 2001 (Zitiert auf S. 30).

[Ley10] F. Leymann. „BPEL vs. BPMN 2.0: Should You Care?“ In: *BPMN*. Hrsg. von J. Mendling, M. Weidlich und M. Weske. Bd. 67. Lecture Notes in Business Information Processing. Springer, 2010, S. 8–13 (Zitiert auf S. 31).

[LJ98] R. Lai und A. Jirachiefpattana. „Lotos“. In: *Communication Protocol Specification and Verification*. Boston, MA: Springer US, 1998, S. 81–109 (Zitiert auf S. 35).

[LKLR07] N. Lohmann, O. Kopp, F. Leymann und W. Reisig. „Analyzing BPEL4Chor: Verification and Participant Synthesis“. In: *Web Services and Formal Methods, 4th International Workshop, WS-FM 2007, Brisbane, Australia, September 28-29, 2007. Proceedings*. Hrsg. von M. Dumas und R. Heckel. Bd. 4937. Lecture Notes in Computer Science. Springer, 2007, S. 46–60 (Zitiert auf S. 33).

[LM07] R. Lucchi und M. Mazzara. „A pi-calculus based semantics for WS-BPEL“. In: *J. Log. Algebraic Methods Program.* 70.1 (2007), S. 96–118 (Zitiert auf S. 35).

[Loh07a] N. Lohmann. „A Feature-Complete Petri Net Semantics for WS-BPEL 2.0“. In: *Web Services and Formal Methods, 4th International Workshop, WS-FM 2007, Brisbane, Australia, September 28-29, 2007. Proceedings*. Hrsg. von M. Dumas und R. Heckel. Bd. 4937. Lecture Notes in Computer Science. Springer, 2007, S. 77–91 (Zitiert auf S. 33).

[Loh07b] N. Lohmann. *A Feature-Complete Petri Net Semantics for WS-BPEL 2.0 and its Compiler BPEL2oWFN*. Techn. Ber. Humboldt-Universität zu Berlin, 2007 (Zitiert auf S. 33).

[Loh10] N. Lohmann. „Communication models for services“. In: *2nd Central-European Workshop on Services and their Composition, Services und ihre Komposition, ZEUS 2010, Berlin, Germany, February 25-26, 2010. Proceedings*. Hrsg. von C. Gierds und J. Sürmeli. Bd. 563. CEUR Workshop Proceedings. CEUR-WS.org, 2010, S. 9–16 (Zitiert auf S. 135).

[LON+13] L. A. F. Leite, G. A. Oliva, G. M. Nogueira, M. A. Gerosa, F. Kon und D. S. Milojicic. „A systematic literature review of service choreography adaptation“. In: *Serv. Oriented Comput. Appl.* 7.3 (2013), S. 199–216 (Zitiert auf S. 30).

[LR00] F. Leymann und D. Roller. *Production Workflow – Concepts and Techniques*. Prentice Hall PTR, 2000 (Zitiert auf S. 15, 49, 77, 195).

[LRW09] C. Li, M. Reichert und A. Wombacher. „Discovering Reference Models by Mining Process Variants Using a Heuristic Approach“. In: *Business Process Management, 7th International Conference, BPM 2009, Ulm, Germany, September 8-10, 2009. Proceedings*. Hrsg. von U. Dayal, J. Eder, J. Koehler und H. A. Reijers. Bd. 5701. Lecture Notes in Computer Science. Springer, 2009, S. 344–362 (Zitiert auf S. 43).

[LZSS04] M. Liu, Q. Zhuge, Z. Shao und E. H.-M. Sha. „General loop fusion technique for nested loops considering timing and code size“. In: *CASES*. 2004, S. 190–201 (Zitiert auf S. 233).

[Men02] T. Mens. „A State-of-the-Art Survey on Software Merging“. In: *IEEE Trans. Software Eng.* 28.5 (2002), S. 449–462 (Zitiert auf S. 47).

[Mil14] A. Milutinovic. *Konsolidierung mittels Event-Handler kommunizierender BPEL-Prozesse*. Deutsch. Studienarbeit: Universität Stuttgart, Institut für Architektur von Anwendungssystemen. Studienarbeit. Juli 2014 (Zitiert auf S. 223, 336).

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Bd. 92. Lecture Notes in Computer Science. Springer, 1980 (Zitiert auf S. 35).

[Mil83] R. Milner. „Calculi for Synchrony and Asynchrony“. In: *Theor. Comput. Sci.* 25 (1983), S. 267–310 (Zitiert auf S. 152).

[MK06] J. Magee und J. Kramer. *Concurrency: State Models and Java Programs*. 2nd. Wiley Publishing, 2006 (Zitiert auf S. 34).

[MKL+09] G. Monakova, O. Kopp, F. Leymann, S. Moser und K. Schäfers. „Verifying Business Rules Using an SMT Solver for BPEL Processes“. In: *Proceedings of the Business Process and Services Computing Conference: BPSC'09*. Lecture Notes in Informatics. Gesellschaft für Informatik e.V. (GI), März 2009 (Zitiert auf S. 33, 232, 336).

[MRS05] P. Massuthe, W. Reisig und K. Schmidt. „An Operating Guideline Approach to the SOA“. In: *Annals of Mathematics, Computing & Teleinformatics* 1.3 (2005), S. 35–43 (Zitiert auf S. 33).

[MS06] J. Mendling und C. Simon. „Business Process Design by View Integration“. In: *BPM Workshops*. Springer, 2006 (Zitiert auf S. 44).

[Muc97] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997 (Zitiert auf S. 270).

[MWJ99] P. Mitra, G. Wiederhold und J. Jannink. „Semi-automatic Integration of Knowledge Sources“. In: *2nd International Conference on Information Fusion (FUSION 1999)*. 1999 (Zitiert auf S. 47).

[Nak05] S. Nakajima. „Lightweight formal analysis of web service flows“. In: *Progress in Informatics* 2.57-76 (2005), S. 5–5 (Zitiert auf S. 39).

[NBF+12] A. Nowak, T. Binz, C. Fehling, O. Kopp, F. Leymann und S. Wagner. „Pattern-driven green adaptation of process-based applications and their runtime infrastructure“. In: *Computing* 94.6 (2012), S. 463–487 (Zitiert auf S. 19).

[OAS07a] OASIS. *WS-TX 1.2 OASIS Standards.* 2007 (Zitiert auf S. 306).

[OAS07b] OASIS. *Web Services Business Process Execution Language Version 2.0 – OASIS Standard*. 2007 (Zitiert auf S. 15, 30, 86, 326, 328).

[OAS13] OASIS. *TOSCA Primer v1.0*. http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.html. 2013 (Zitiert auf S. 20).

[Obj11] Object Management Group (OMG). *Business Process Model and Notation (BPMN) Version 2.0*. OMG Document Number: formal/2011-01-03. 2011 (Zitiert auf S. 15, 30, 31).

[OVvdA+05] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas und A. H. M. ter Hofstede. „WofBPEL: A Tool for Automated Analysis of BPEL Processes“. In: *Service-Oriented Computing - ICSOC 2005, Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005, Proceedings*. Hrsg. von B. Benatallah, F. Casati und P. Traverso. Bd. 3826. Lecture Notes in Computer Science. Springer, 2005, S. 484–489 (Zitiert auf S. 33).

[OVvdA+07] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas und A. H. M. ter Hofstede. „Formal semantics and analysis of control flow in WS-BPEL“. In: *Sci. Comput. Program.* 67.2-3 (2007), S. 162–198 (Zitiert auf S. 33).

[Pal07] M. Paluszek. „Coordinating Distributed Loops and Fault Handling, Transactional Scopes using WS- Coordination protocols layered on WS-BPEL services“. Diplomarbeit. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, März 2007, S. 155 (Zitiert auf S. 307).

[PB03] R. Pottinger und P. A. Bernstein. „Merging Models Based on Given Correspondences“. In: *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*. Hrsg. von J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger und A. Heuer. Morgan Kaufmann, 2003, S. 826–873 (Zitiert auf S. 42).

[Pel03] C. Peltz. „Web Services Orchestration and Choreography“. In: *IEEE Computer* 36.10 (2003), S. 46–52 (Zitiert auf S. 16, 30).

[Pet62] C. A. Petri. „Kommunikation mit Automaten“. ger. Diss. Universität Hamburg, 1962 (Zitiert auf S. 33).

[Pha15] P. R. Phadnis. „Consolidation of Process Models that Interact via Compensation Handlers“. Masterarbeit. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Aug. 2015, S. 78 (Zitiert auf S. 223, 327, 335).

[Pnu77] A. Pnueli. „The Temporal Logic of Programs“. In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, S. 46–57 (Zitiert auf S. 38).

[PZQ+06] G. Pu, H. Zhu, Z. Qiu, S. Wang, X. Zhao und J. He. „Theoretical Foundations of Scope-Based Compensable Flow Language for Web Service“. In: *Formal Methods for Open Object-Based Distributed Systems, 8th IFIP WG 6.1 International Conference, FMOODS 2006, Bologna, Italy, June 14-16, 2006, Proceedings*. Hrsg. von R. Gorrieri und H. Wehrheim. Bd. 4037. Lecture Notes in Computer Science. Springer, 2006, S. 251–266 (Zitiert auf S. 36).

[PZWQ06] G. Pu, X. Zhao, S. Wang und Z. Qiu. „Towards the Semantics and Verification of BPEL4WS“. In: *Electron. Notes Theor. Comput. Sci.* 151.2 (2006), S. 33–52 (Zitiert auf S. 36).

[QCS02] Y. Qian, S. Carr und P. H. Sweany. „Loop fusion for clustered VLIW architectures“. In: *LCTES-SCOPES*. 2002, S. 112–119 (Zitiert auf S. 233, 270).

[RB01] E. Rahm und P. A. Bernstein. „A survey of approaches to automatic schema matching“. In: *VLDB J.* 10.4 (2001), S. 334–350 (Zitiert auf S. 47).

[RDUD10] M. L. Rosa, M. Dumas, R. Uba und R. M. Dijkman. „Merging Business Process Models“. In: *On the Move to Meaningful Internet Systems: OTM 2010 - Confederated International Conferences: CoopIS, IS, DOA and ODBASE, Hersonissos, Crete, Greece, October 25-29, 2010, Proceedings, Part I*. Hrsg. von R. Meersman, T. S. Dillon und P. Herrero. Bd. 6426. Lecture Notes in Computer Science. Springer, 2010, S. 96–113 (Zitiert auf S. 44).

[RDUD13] M. L. Rosa, M. Dumas, R. Uba und R. M. Dijkman. „Business Process Model Merging: An Approach to Business Process Consolidation“. In: *ACM Trans. Softw. Eng. Methodol.* 22.2 (2013), 11:1–11:42 (Zitiert auf S. 45, 47).

[Rei94] A. J. Reich. „Intervals, Points, and Branching Time“. In: *TIME*. 1994, S. 121–133 (Zitiert auf S. 41).

[RKDL08] P. Reimann, O. Kopp, G. Decker und F. Leymann. *Generating WS-BPEL 2.0 Processes from a Grounded BPEL4Chor Choreography*. Technischer Bericht Informatik 2008/07. Universität Stuttgart, Institut für Architektur von Anwendungssystemen; Universität Stuttgart, Institut für Parallele und Verteilte Systeme, Anwendersoftware: Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Oktober 2008, S. 95 (Zitiert auf S. 48, 238).

[RvdA07] M. Rosemann und W. M. P. van der Aalst. „A configurable reference modelling language“. In: *Inf. Syst.* 32.1 (2007), S. 1–23 (Zitiert auf S. 45).

[SCC19] P. Sousa, D. Cardoso und J. Colaço. „Managing Multi-view Business Processes Models in the Atlas Tool“. In: *Proceedings of the 19th CIAO! Doctoral Consortium, Enterprise Engineering Working Conference Forum 2019, and Enterprise Engineering Working Conference Posters 2019, co-located with the 9th Enterprise Engineering Working Conference (EEWC 2019), Lisbon, Portugal, May 20-24, 2019*. Hrsg. von S. Guerreiro, P. Sousa, D. Aveiro, G. Guizzardi, J. Borbinha und R. Winter. Bd. 2408. CEUR Workshop Proceedings. CEUR-WS.org, 2019 (Zitiert auf S. 46).

[SFC04] G. Salaün, A. Ferrara und A. Chirichiello. „Negotiation Among Web Services Using LOTOS/CADP“. In: *Web Services, European Conference, ECOWS 2004, Erfurt, Germany, September 27-30, 2004, Proceedings*. Hrsg. von L.-J. Zhang. Bd. 3250. Lecture Notes in Computer Science. Springer, 2004, S. 198–212 (Zitiert auf S. 36).

[SK13] M. Sonntag und D. Karastoyanova. „Model-as-you-go: An Approach for an Advanced Infrastructure for Scientific Workflows“. In: *Journal of Grid Computing* 11.3 (Sep. 2013), S. 553–583 (Zitiert auf S. 46).

[SK18] E. Stachtiari und P. Katsaros. „Compositional execution semantics for business process verification“. In: *J. Syst. Softw.* 137 (2018), S. 217–238 (Zitiert auf S. 40).

[SKK+11] D. Schumm, D. Karastoyanova, O. Kopp, F. Leymann, M. Sonntag und S. Strauch. „Process Fragment Libraries for Easier and Faster Development of Process-based Applications“. In: *Journal of Systems Integration* 2 (2011), S. 39–55 (Zitiert auf S. 72).

[SKY06] S. Sun, A. Kumar und J. Yen. „Merging workflows: A new perspective on connecting business processes“. In: *Decis. Support Syst.* 42.2 (2006), S. 844–858 (Zitiert auf S. 43).

[SLM+10] D. Schumm, F. Leymann, Z. Ma, T. Scheibler und S. Strauch. „Integrating Compliance into Business Processes“. In: *Multikonferenz Wirtschaftsinformatik, MKWI 2010, Göttingen, Deutschland, 23.-25.2.2010, Proceedings*. Hrsg. von M. Schumann, L. M. Kolbe, M. H. Breitner und A. Frerichs. Universitätsverlag Göttingen, 2010, S. 2125–2137 (Zitiert auf S. 45).

[Sta04] C. Stahl. „Transformation von BPEL4WS in Petrinetze“. German. Master Thesis. Humboldt-Universität zu Berlin, Apr. 2004 (Zitiert auf S. 33).

[STA05] A.-W. Scheer, O. Thomas und O. Adam. „Process Aware Information Systems“. In: Wiley-Interscience, 2005. Kap. Process Modeling Using Event-Driven Process Chains (Zitiert auf S. 44).

[vBK06] F. van Breugel und M. Koshkina. „Models and Verification of BPEL“. `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf`. 2006 (Zitiert auf S. 31).

[vdAal11] W. M. P. van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011 (Zitiert auf S. 42).

[vdAal98] W. M. P. van der Aalst. „The application of Petri nets to workflow management“. In: *Journal of Circuits, Systems and Computers* 8.1 (1998), S. 21–66 (Zitiert auf S. 43).

[vdAtHKB03] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski und A. P. Barros. „Workflow Patterns“. In: *Distributed and Parallel Databases* 14.1 (2003), S. 5–51 (Zitiert auf S. 85, 110).

[vdAtHW03] W. M. P. van der Aalst, A. H. M. ter Hofstede und M. Weske. „Business Process Management: A Survey“. In: *Business Process Management, International Conference, BPM 2003, Eindhoven, The Netherlands, June 26-27, 2003, Proceedings*. Hrsg. von W. M. P. van der Aalst, A. H. M. ter Hofstede und M. Weske. Bd. 2678. Lecture Notes in Computer Science. Springer, 2003, S. 1–12 (Zitiert auf S. 80).

[vGla90] R. J. van Glabbeek. „The Linear Time-Branching Time Spectrum (Extended Abstract)“. In: *CONCUR*. 1990, S. 278–297 (Zitiert auf S. 152).

[Vir04] M. Viroli. „Towards a Formal Foundation to Orchestration Languages“. In: *Electron. Notes Theor. Comput. Sci.* 105 (2004), S. 51–71 (Zitiert auf S. 36).

[WAHK15] A. Weiß, V. Andrikopoulos, M. Hahn und D. Karastoyanova. „Enabling the Extraction and Insertion of Reusable Choreography Fragments“. In: *Proceedings of the 22nd IEEE International Conference on Web Services*. New York: IEEE, Juni 2015, S. 686–694 (Zitiert auf S. 72).

[WAHK17] A. Weiß, V. Andrikopoulos, M. Hahn und D. Karastoyanova. „Model-as-you-go for Choreographies: Rewinding and Repeating Scientific Choreographies“. In: *IEEE Transactions on Services Computing* PP.99 (Juli 2017), S. 1–1 (Zitiert auf S. 46, 49, 73).

[WAS+13] A. Weiß, V. Andrikopoulos, S. G. Sáez, D. Karastoyanova und K. Vukojevic-Haupt. *Modeling Choreographies using the BPEL4Chor Designer: an Evaluation Based on Case Studies*. Technischer Bericht Informatik 2013/03. Universität Stuttgart, Institut für Architektur von Anwendungssystemen: Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Aug. 2013, S. 23 (Zitiert auf S. 323).

[WBK+17] S. Wagner, U. Breitenbücher, O. Kopp, A. Weiß und F. Leymann. „Fostering the Reuse of TOSCA-based Applications by Merging BPEL Management Plans“. In: *Cloud Computing and Services Science: 6th International Conference (CLOSER 2016) - Revised Selected Papers*. Bd. 740. Communications in Computer and Information Science.

Springer International Publishing, Juli 2017, S. 232–254 (Zitiert auf S. 20, 26, 326).

[WBL16] S. Wagner, U. Breitenbücher und F. Leymann. „A Method For Reusing TOSCA-based Applications and Management Plans“. In: *Proceedings of the 6th International Conference on Cloud Computing and Service Science (CLOSER 2016)*. Rome: SciTePress, Apr. 2016, S. 181–191 (Zitiert auf S. 26).

[WDW07] M. Weidlich, G. Decker und M. Weske. „Efficient Analysis of BPEL 2.0 Processes Using p-Calculus“. In: *Proceedings of The 2nd IEEE Asia-Pacific Services Computing Conference, APSCC 2007, December 11-14, 2007, Tsukuba Science City, Japan*. Hrsg. von J. Li, M. Guo, Q. Jin, Y. Zhang, L.-J. Zhang, H. Jin, M. Mambo, J. Tanaka und H. Hayashi. IEEE Computer Society, 2007, S. 266–274 (Zitiert auf S. 35, 36).

[Wei11] M. Weidlich. „Verhaltensprofile - Ein Relationaler Ansatz zur Verhaltenskonsistenzanalyse“. In: *Ausgezeichnete Informatikdissertationen 2011*. Hrsg. von S. Hölldobler. Bd. D-12. LNI. GI, 2011, S. 261–270 (Zitiert auf S. 41).

[Wer07] D. Werth. *Modellierung unternehmensübergreifender Geschäftsprozesse – Modelle, Notation und Vorgehen für Geschäftsprozesse*. Salzwasser Verlag, 2007 (Zitiert auf S. 15).

[Wes12] M. Weske. *Business Process Management - Concepts, Languages, Architectures, 2nd Edition*. Springer, 2012 (Zitiert auf S. 77).

[WFKS12] S. Wagner, C. Fehling, D. Karastoyanova und D. Schumm. „State Propagation-based Monitoring of Business Transactions“. In: *Proceedings of the 2012 IEEE International Conference on Service-Oriented Computing and Applications (SOCA 2012)*. Taipeh: IEEE Xplore, Dezember 2012 (Zitiert auf S. 17, 27, 336, 337).

[WFN04] A. Wombacher, P. Fankhauser und E. J. Neuhold. „Transforming BPEL into Annotated Deterministic Finite State Automata for Service Discovery“. In: *Proceedings of the IEEE International Conference on Web Services (ICWS'04), June 6-9, 2004, San Diego, California, USA*. IEEE Computer Society, 2004, S. 316–323 (Zitiert auf S. 39).

[WKK+10] B. Wetzstein, D. Karastoyanova, O. Kopp, F. Leymann und D. Zwink. „Cross-organizational process monitoring based on service choreographies“. In: *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*. Hrsg. von S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal und C.-C. Hung. ACM, 2010, S. 2485–2490 (Zitiert auf S. 337).

[WKL11] S. Wagner, O. Kopp und F. Leymann. „Towards Choreography-based Process Distribution In The Cloud“. In: *Proceedings of the 2011 IEEE International Conference on Cloud Computing and Intelligence Systems*. Bejing: IEEE Xplore, Sep. 2011, S. 490–494 (Zitiert auf S. 27).

[WKL12] S. Wagner, O. Kopp und F. Leymann. „Towards Verification of Process Merge Patterns with Allen's Interval Algebra“. In: *Proceedings of the 4th Central-European Workshop on Services and their Composition (ZEUS 2012)*. Bamberg: CEUR Workshop Proceedings, März 2012, S. 81–88 (Zitiert auf S. 27, 336).

[WKL13] S. Wagner, O. Kopp und F. Leymann. „Consolidation of Interacting BPEL Process Models with Fault Handlers“. In: *Proceedings of the 5th Central-European Workshop on Services and their Composition (ZEUS 2013)*. Rostock: CEUR Workshop Proceedings, Feb. 2013, S. 9–16 (Zitiert auf S. 27, 223).

[WKL14] S. Wagner, O. Kopp und F. Leymann. „Choreography-based Consolidation of Multi-Instance BPEL Processes“. In: *Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER 2014)*. Hrsg. von SciTePress. Barcelona: SciTePress, Apr. 2014, S. 287–298 (Zitiert auf S. 26, 166).

[WKL15] S. Wagner, O. Kopp und F. Leymann. „Choreography-based Consolidation of Interacting Processes Having Activity-based Loops“. In: *Proceedings of the 5th International Conference on Cloud Computing and Service Science (CLOSER 2015)*. Stuttgart: SciTePress, Mai 2015, S. 284–296 (Zitiert auf S. 26, 222, 270).

[WMW11] M. Weidlich, J. Mendling und M. Weske. „Efficient Consistency Measurement Based on Behavioral Profiles of Process Models“. In: *IEEE Trans. Software Eng.* 37.3 (2011), S. 410–429 (Zitiert auf S. 41).

[WPMW11] M. Weidlich, A. Polyvyanyy, J. Mendling und M. Weske. „Causal Behavioural Profiles - Efficient Computation, Applications, and Evaluation“. In: *Fundam. Inform.* 113.3-4 (2011), S. 399–435 (Zitiert auf S. 41).

[WRK+13] S. Wagner, D. Roller, O. Kopp, T. Unger und F. Leymann. „Performance Optimizations for Interacting Business Processes“. In: *Proceedings of the first IEEE International Conference on Cloud Engineering (IC2E 2013)*. San Francisco: IEEE Computer Society, März 2013, S. 210–216 (Zitiert auf S. 19, 27).

[YTYL05] Y. Yang, Q. Tan, J. Yu und F. Liu. „Transformation BPEL to CP-Nets for Verifying Web Services Composition“. In: Sep. 2005, 6 pp. (Zitiert auf S. 33).

[YZ14] C. Yang und F. Zhong. „Towards the formal foundation of orchestration process“. In: *Fifth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*. 2014, S. 1–6 (Zitiert auf S. 33).

[Zac87] J. A. Zachman. „A Framework for Information Systems Architecture“. In: *IBM Syst. J.* 26.3 (1987), S. 276–292 (Zitiert auf S. 46).

[ZBDtH06] J. M. Zaha, A. P. Barros, M. Dumas und A. H. M. ter Hofstede. „Let's Dance: A Language for Service Behavior Modeling“. In: *OTM Conferences (1)*. Hrsg. von R. Meersman und Z. Tari. Bd. 4275. Lecture Notes in Computer Science. Springer, 2006, S. 145–162 (Zitiert auf S. 31).

[ZHZ17] Y. Zhu, Z. Huang und H. Zhou. „Modeling and verification of Web services composition based on model transformation“. In: *Softw. Pract. Exp.* 47.5 (2017), S. 709–730 (Zitiert auf S. 37).

[ZMH16] M. A. Zemni, A. Mammar und N. B. Hadj-Alouane. „An automated approach for merging business process fragments“. In: *Comput. Ind.* 82 (2016), S. 104–118 (Zitiert auf S. 45).

Alle URLs wurden zuletzt am 10.12.2021 geprüft.

Abbildungsverzeichnis

ANHANG

In diesem Kapitel werden die unterstützenden Algorithmen erläutert, die von den Konsolidierungsalgorithmen in Kapitel 5 bzw. Kapitel 6 verwendet werden.

A.1. Erstellung von Aktivitäten, Hierarchiebeziehungen und Kontrollflusskanten

Algorithmus A.1 Hinzufügen einer Aktivität zum Kontrollfluss von p_μ

```
1: function CreateActivity(t_A)
2:     a = new Activity
3:     type_A(a) ← t_A
4:     π_1(p_μ) ← π_1(p_μ) ∪ {a}
5:     return a
6: end function
```

Die Funktion CreateActivity in Algorithmus A.1 erstellt eine neue Aktivität, weist ihr den im Parameter übergebenen Typ zu und fügt sie zum Prozessmodell p_μ und der Choreographie hinzu.

Eine neue Hierarchiebeziehung wird in p_μ durch die Funktion CreateHR in Algorithmus A.2 erstellt.

Algorithmus A.2 Erstellen einer Hierarchiebeziehung in p_μ

1: **function** CreateHR(a_{parent}, $\mathfrak{t}_{HR}$, a_{child})
2: $hr = (a_{parent}, \mathfrak{t}_{HR}, a_{child})$
3: $\pi_2(p_\mu) \leftarrow \pi_2(p_\mu) \cup \{hr\}$ **return** hr
4: **end function**

Die Funktion addChildActivities in Algorithmus A.3 fügt zu einer strukturierten Flow-Aktivität $a_{flow} \in A_{\texttt{flow}}$ eine Menge von Kindaktivitäten A_{child} hinzu.

Algorithmus A.3 Hinzufügen von Kindaktivitäten zu Flow-Aktivität

1: **procedure** addChildActivities(a_{flow}, A_{child})
2: **for all** $a_{child} \in A_{child}$ **do**
3: CreateHR(a_{flow}, $\mathfrak{t}^{ch}_{HR}$, a_{child})
4: **end for**
5: **end procedure**

Die Funktion in Algorithmus A.4 erstellt eine Kontrollflusskante und fügt sie zu p_μ hinzu. Wird der Funktion keine Transitionsbedingung übergeben, wird diese standardmäßig auf `true` gesetzt.

Algorithmus A.4 Erstellung von Kontrollflusskanten

1: **function** CreateControlLink(a_{src}, a_{trg}, $\mathfrak{c}$)
2: **if** $\mathfrak{c} = \bot$ **then**
3: $\mathfrak{c} \leftarrow$ `true`
4: **end if**
5: $l = (a_{src}, a_{trg}, \mathfrak{c})$
6: $\pi_3(p_\mu) \leftarrow \pi_3(p) \cup \{l\}$ **return** l
7: **end function**

A.2. Ersetzen und Entfernen von Aktivitäten im Kontrollfluss

Die Prozedur ReplaceActivity in Algorithmus A.5 ersetzt im Kontrollfluss die Aktivität a_{org} durch die Aktivität a_{new}. Die Aktivität a_{new} wird dazu zum Prozessmodell p_μ hinzugefügt und erbt von a_{org} alle Kontrollflusseigenschaften, d.h. die eingehenden und ausgehenden Kanten (Zeile 3 bzw. 4), die Eintrittsbedingung (Zeile 5) sowie die Elternaktivität (Zeile 7) und Kindaktivitäten (Zeile 9). Die Aktivität a_{org} wird wiederum aus p_μ entfernt.

Algorithmus A.5 Ersetzen einer Aktivität a_{org} durch a_{new} im Kontrollfluss

1: **procedure** ReplaceActivity(a_{org}, a_{new})
2: $\quad \pi_1(p_\mu) \leftarrow \pi_1(p_\mu) \cup \{a_{new}\}) \setminus \{a_{org}\}$
3: $\quad \forall l_{in} \in \text{Incoming}_L(a_{org}) : \pi_2(l_{in}) \leftarrow a_{new}$
4: $\quad \forall l_{out} \in \text{Outgoing}_L(a_{org}) : \pi_1(l_{out}) \leftarrow a_{new}$
5: $\quad \text{joinCond}(a_{new}) \leftarrow \text{joinCond}(a_{org})$
6: $\quad hr_{parent} \leftarrow \text{ParentHR}(a_{org})$
7: $\quad \text{CreateHR}(\pi_1(hr_{parent}), \pi_2(hr_{parent}), a_{new})$
8: $\quad \pi_2(p_\mu) \leftarrow \pi_2(p_\mu) \setminus \{hr_{parent}\}$
9: $\quad \forall hr_{child} \in \{hr \in HR \mid \pi_1(hr) = a_{org}\} : \pi_1(hr_{child}) \leftarrow a_{new}$
10: **end procedure**

A.3. Algorithmen zur Duplizierung von Prozessmodellen

Dieser Abschnitt beschreibt verschiedene Algorithmen, mit denen Prozessmodelle und ihre Elemente wie Aktivitäten, Variablen usw. dupliziert werden können. Duplizieren bedeutet in diesem Kontext, dass ein neues Element erzeugt wird, dem die Eigenschaften des originalen Elements zugewiesen werden.

Die Algorithmen nutzen die untenstehenden Abbildungen, um dem originalen Element aus der Choreographie $c \in C$ sein Duplikat in p_μ zuzuweisen. Um das Duplikat für jeden Teilnehmer eindeutig zu ermitteln, d.h., um die Rechtseindeutigkeit der Abbildungen zu gewährleisten, muss dieser immer mit angegeben werden. Optional kann für Elemente, die beim Ausrollen

von Schleifen dupliziert werden, also Aktivitäten, Kontrollflusskanten und Hierarchiebeziehungen, noch der Teilnehmer $\mathfrak{P}$ mit angegeben werden, der von einer ausgerollten Teilnehmeriteration repräsentiert wird (siehe Abschnitt 6.3). Befindet sich das Element nicht in einer Teilnehmeriteration, wird anstatt des Teilnehmers das Symbol $\perp$ angegeben. Die Abbildung $\mathsf{actCopy}(\mathfrak{p}_{snd}, \mathfrak{p}_{rcv1}, a_{org}) \leftarrow a_{copy}$ gibt zum Beispiel an, dass sich das Duplikat a_{copy} der Aktivität a_{org} im Teilnehmer-Scope befindet, der Teilnehmer $\mathfrak{p}_{snd}$ repräsentiert. Das Duplikat a_{copy} ist dabei der Teilnehmeriteration einer ausgerollten Schleife zugeordnet, die die Kommunikation mit Teilnehmer $\mathfrak{p}_{rcv1}$ emuliert.

- $\mathsf{condCopy} : \mathfrak{P} \times \mathfrak{C} \rightarrow \mathfrak{C}$ weist einer Bedingung aus c das Duplikat in p_μ zu
- $\mathsf{eventCopy} : \mathfrak{P} \times E \rightarrow E$ weist einem Ereignis aus c das Duplikat in p_μ zu
- $\mathsf{varCopy} : \mathfrak{P} \times V \rightarrow V$ weist einer Variablen aus c das Duplikat in p_μ zu
- $\mathsf{participantScope} : \mathfrak{P} \rightarrow A_{\mathtt{scope}}$ weist einem Teilnehmer das Duplikat eines Prozessmodells als Teilnehmer-Scope zu
- $\mathsf{actCopy} : \mathfrak{P} \times (\mathfrak{P} \cup \{\perp\}) \times A \rightarrow A$ weist einer Aktivität aus c das Duplikat in p_μ zu
- $\mathsf{hrCopy} : \mathfrak{P} \times (\mathfrak{P} \cup \{\perp\}) \times HR \rightarrow HR$ weist einer Hierarchiebeziehung aus c das Duplikat in p_μ zu
- $\mathsf{clCopy} : \mathfrak{P} \times (\mathfrak{P} \cup \{\perp\}) \times L \rightarrow L$ weist einer Kontrollflusskante aus c das Duplikat in p_μ zu

Algorithmus A.6 Duplizierung von Prozessmodellen und ihren Elementen

1: **function** DUPLICATEPROCESSMODEL($p_{org}, \mathfrak{p}$)
2: $p_{copy} = new\texttt{process}$
3: $\pi_7(p_{copy}) \leftarrow \pi_7(p_{org})$
4: **for all** $\mathfrak{c}_{org} \in \pi_5(p_{org})$ **do**
5: $\mathfrak{c}_{copy} =$ DUPLICATECONDITION($\mathfrak{c}_{org}$)
6: condCopy($\mathfrak{p}, \perp, \mathfrak{c}_{org}$) $\leftarrow \mathfrak{c}_{copy}$
7: $\pi_5(p_{copy}) \leftarrow \pi_5(p_{copy}) \cup \{\mathfrak{c}_{copy}\}$
8: **end for**
9: **for all** $v_{org} \in \pi_4(p_{org})$ **do**
10: $v_{copy} =$ DUPLICATEVARIABLE(v_{org})
11: varCopy($\mathfrak{p}, v_{org}$) $\leftarrow v_{copy}$
12: $\pi_4(p_{copy}) \leftarrow \pi_4(p_{copy}) \cup \{v_{copy}\}$
13: **end for**
14: **for all** $e_{org} \in \pi_8(p_{org})$ **do**
15: $e_{copy} =$ DUPLICATEEVENT(p_{copy}, e_{org}, varCopy)
16: eventCopy($\mathfrak{p}, e_{org}$) $\leftarrow e_{copy}$
17: $\pi_8(p_{copy}) \leftarrow \pi_8(p_{copy}) \cup \{e_{copy}\}$
18: **end for**
19: **for all** $a_{org} \in (\pi_1(p_{org})$ **do**
20: $a_{copy} =$ DUPLICATEACTIVITY($p_{copy}, \perp, a_{org}$)
21: $\pi_1(p_{copy}) \leftarrow \pi_1(p_{copy}) \cup \{a_{copy}\}$
22: **end for**
23: **for all** $hr_{org} \in \pi_2(p_{org})$ **do**
24: $hr_{copy} =$ DUPLICATEHIERARCHYRELATION($\mathfrak{p}, \perp, hr_{org}$)
25: hrCopy($\mathfrak{p}, \perp, hr_{org}$) $\leftarrow hr_{copy}$
26: $\pi_2(p_{copy}) \leftarrow \pi_2(p_{copy}) \cup \{hr_{copy}\}$
27: **end for**
28: **for all** $l_{org} \in \pi_3(p_{org})$ **do**
29: $l_{copy} =$ DUPLICATECONTROLLINK($\mathfrak{p}, \perp, l_{org}$)
30: clCopy($\mathfrak{p}, \perp, \mathfrak{c}_{org}$) $\leftarrow l_{copy}$
31: $\pi_3(p_{copy}) \leftarrow \pi_3(p_{copy}) \cup \{l_{copy}\}$
32: **end for**
33: **return** p_{copy}
34: **end function**

Die Duplizierung eines Prozessmodells p_{org}, das das Verhalten eines Teilnehmers $\mathfrak{p}$ beschreibt, wird durch die Funktion DuplicateProcessmodel in Algorithmus A.6 implementiert. Die Funktion gibt ein neues Prozessmodell zurück, dem die Duplikate der Elemente des originalen Prozessmodells p_{org} zugewiesen sind. Alle Elemente, die nicht von den in Kapitel 5 bzw. Kapitel 6 beschriebenen Algorithmen modifiziert werden, wie zum Beispiel Bezeichner, werden nicht dupliziert. Die Reihenfolge der Duplizierung der Elemente in Algorithmus A.6 ist durch die transitiven Abhängigkeiten der Prozesselemente vorgegeben. Auf die Duplikate der Elemente kann über die entsprechenden Abbildungen, wie z.B. condCopy oder actCopy zugegriffen werden. Es wird in den Algorithmen davon ausgegangen, dass die zu konsolidierende Choreographie c global sichtbar ist.

Die Bedingungen werden mittels Algorithmus A.7 dupliziert. Der Ausdruck der Bedingung wird nicht dupliziert, da Bedingungen nur durch Zuweisung eines neuen Ausdrucks geändert werden.

Algorithmus A.7 Duplizierung von Bedingungen

```
1: function DuplicateCondition(c_org)
2:     c_copy = new Condition
3:     expr_B(c_copy) ← expr_B(c_org)
4:     return c_copy
5: end function
```

In Algorithmus A.8 werden die Variablen dupliziert. Da Datentypen von den Konsolidierungsalgorithmen nicht geändert werden, werden diese nicht dupliziert.

Algorithmus A.8 Duplizierung von Variablen

```
1: function DuplicateVariable(v_org)
2:     v_copy = new Variable
3:     value_V(v_copy) ← value_V(v_copy)
4:     return v_copy
5: end function
```

Ereignisse werden mittels Algorithmus A.9 dupliziert. Die zu duplizierenden Elemente hängen dabei vom Typ des jeweiligen Ereignisses ab. Der Algorithmus weist Duplikaten von Nachrichtenereignissen das jeweilige Variablenduplikat zu, daher muss die Duplizierung der Variablen eines Prozessmodells vor der Ausführung dieses Algorithmus erfolgen.

Algorithmus A.9 Duplizierung eines Ereignisses

```
1:  function DuplicateEvent(𝔭, e_org)
2:      e_copy = new Event
3:      type_E(e_copy) ← type_E(e_org)
4:      if e_org ∈ E^timer then
5:          for(e_copy) ← for(e_org)
6:      else if e_org ∈ E^fault then
7:          faultName(e_copy) ← faultName(e_org)
8:      else if e_org ∈ E^msg then
9:          outputVar(e_copy) ← varCopy(𝔭, outputVar(e_org))
10:     end if
11:     return e_copy
12: end function
```

Algorithmus A.10 erstellt das Duplikat einer Aktivität. Die Elemente bzw. Eigenschaften, die dem Duplikat zugewiesen werden, hängen von dem Typ der Originalaktivität a_{org} ab. Für Aktivitäten, die nicht auf andere Elemente des Prozessmodells verweisen, wie zum Beispiel die Pick-Aktivität, werden nur die Standardelemente dupliziert. Voraussetzung für den Algorithmus ist, dass vorher bereits Duplikate der Bedingungen und Variablen der Originalaktivität erstellt wurden.

Algorithmus A.10 Duplizierung einer Aktivität

```
1:  function DuplicateActivity(𝔭, 𝔭#, a_org)
2:      a_copy = new Activity
3:      actCopy(𝔭, 𝔭#, a_org) ← a_copy
4:      type_A(a_copy) ← type_A(a_org)
5:      joinCond(a_copy) ← condCopy(𝔭, joinCond(a_org))
6:      if a_copy ∈ A_while then
7:          cond_while(a_copy) ← condCopy(𝔭, cond_while(a_org))
8:      else if a_copy ∈ A_throw then
9:          throws(a_copy) ← eventCopy(𝔭, throws(a_org))
10:     else if a_copy ∈ A_invoke then
11:         inputVar(a_copy) ← varCopy(𝔭, inputVar(a_org))
12:         receiver(a_copy) ← receiver(a_org)
13:     else if a_copy ∈ A_receive then
14:         outputVar(a_copy) ← varCopy(𝔭, outputVar(a_org))
15:     else if a_copy ∈ A_assign then
16:         assign_src(a_copy) ← varCopy(𝔭, assign_src(a_org))
17:         assign_trg(a_copy) ← varCopy(𝔭, assign_trg(a_org))
18:     else if a_copy ∈ A_wait then
19:         timer(a_copy) ← eventCopy(𝔭, timer(a_org))
20:     end if
21:     return a_copy
22: end function
```

Analog zu Aktivitäten werden Hierarchiebeziehungen von Algorithmus A.11 ebenfalls typabhängig dupliziert. Da Hierarchiebeziehungen Bedingungen, Ereignisse oder Aktivitäten referenzieren können, muss der Algorithmus nach der Duplizierung dieser Elemente ausgeführt werden.

Algorithmus A.11 Duplizierung von Hierarchiebeziehungen

function DUPLICATEHIERARCHYRELATION($\mathfrak{p}, \mathfrak{p}_{\#}, hr_{org}$)
  $a_{parent} = \text{actCopy}(\mathfrak{p}, \mathfrak{p}_{\#}, \pi_1(hr_{org}))$
  $a_{child} = \text{actCopy}(\mathfrak{p}, \mathfrak{p}_{\#}, \pi_3(hr_{org}))$
  $\mathfrak{t}_{HR}^{org} = \pi_2(hr_{org})$
  $\mathfrak{t}_{HR}^{copy} = \mathfrak{t}_{HR}^{org}$
  if $\mathfrak{t}_{HR}^{org} \in \mathfrak{C}$ **then**
    $\mathfrak{t}_{HR}^{copy} \leftarrow \text{condCopy}(\mathfrak{p}, \mathfrak{p}_{\#}, \mathfrak{t}_{HR}^{org})$
  else if $e_{org} \in E$ **then**
    $\mathfrak{t}_{HR}^{copy} \leftarrow \text{eventCopy}(\mathfrak{p}, \mathfrak{t}_{HR}^{org})$
  end if
  $hr_{copy} = (a_{parent}, \mathfrak{t}_{HR}^{copy}, a_{child})$
  return hr_{copy}
end function

Kontrollflusskanten werden mit Algorithmus A.12 dupliziert. Wie Hierarchiebeziehungen referenzieren Kanten ebenfalls Aktivitäten und Bedingungen, die vor der Ausführung des Algorithmus dupliziert werden müssen.

Algorithmus A.12 Duplizierung von Kontrollflusskanten

function DUPLICATECONTROLLINK($\mathfrak{p}$, $\mathfrak{p}_{\#}$, l_{org})
  $a_{src} = \text{actCopy}(\mathfrak{p}, \mathfrak{p}_{\#}, \pi_1(l_{org}))$
  $a_{trg} = \text{actCopy}(\mathfrak{p}, \mathfrak{p}_{\#}, \pi_2(l_{org}))$
  $\mathfrak{c}_{copy} = \text{DUPLICATECONDITITION}(\pi_3(l_{org}))$
  $l_{copy} = (a_{src}, a_{trg}, \mathfrak{c}_{copy})$
  return l_{copy}
end function

Die Funktion DUPLICATELOOPBODY in Algorithmus A.13 dupliziert den Schleifenkörper der Schleife a_{loop} und gibt die Kopie der direkten Kindaktivität des duplizierten Schleifenkörpers zurück. Die Funktion ist daher ähnlich zu der Funktion DUPLICATEPROCESSMODEL aufgebaut. Allerdings werden hier nur

die Aktivitäten des Schleifenkörpers dupliziert sowie die Hierarchiebeziehungen und Kontrollflusskanten zwischen diesen. Bedingungen, Ereignisse und Variablen werden nicht dupliziert, da sie mittels des Metamodells nur global auf Prozessmodellebene deklariert werden können. Folglich greift zur Laufzeit einer Schleife jede Iteration eines Schleifenkörpers auf dieselben Bedingungen, Ereignisse und Variablen zu, d.h. es existieren keine lokalen Kopien für die jeweilige Iteration.

Algorithmus A.13 Duplizierung eines Schleifenkörpers

1: **function** DUPLICATELOOPBODY(a_{loop}, $\mathfrak{p}$, $\mathfrak{p}_{\#}$)
2: $A_{desc} = \text{DESCENDANTS}(a_{loop})$
3: **for all** $a_{desc} \in A_{desc}$ **do**
4: $a_{copy} \leftarrow \text{DUPLICATEACTIVITY}(\mathfrak{p}, \mathfrak{p}_{\#}, a_{desc})$
5: $\pi_1(p_\mu) \leftarrow \pi_1(p_\mu) \cup \{a_{copy}\}$
6: **end for**
7: $HR_{child} = \{hr \in HR \mid \pi_1(hr) \in a_{desc} \wedge \pi_3(hr) \in a_{desc}\}$
8: **for all** $hr_{child} \in HR_{child}$ **do**
9: $hr_{copy} \leftarrow \text{DUPLICATEHIERARCHYRELATION}(\mathfrak{p}, \mathfrak{p}_{\#}, hr_{child})$
10: $\mathsf{hrCopy}(\mathfrak{p}, \mathfrak{p}_{\#}, hr_{child}) \leftarrow hr_{copy}$
11: $\pi_2(p_\mu) \leftarrow \pi_2(p_\mu) \cup \{hr_{copy}\}$
12: **end for**
13: $L_{child} = \{l \in L \mid \pi_1(l) \in A_{desc} \wedge \pi_2(l) \in A_{desc}\}$
14: **for all** $l_{child} \in L_{child}$ **do**
15: $l_{copy} \leftarrow \text{DUPLICATECONTROLLINK}(\mathfrak{p}, \mathfrak{p}_{\#}, a_{loop})$
16: $\mathsf{clCopy}(\mathfrak{p}, \mathfrak{p}_{\#}, l_{child}) \leftarrow l_{copy}$
17: $\pi_3(p_\mu) \leftarrow \pi_3(p_\mu) \cup \{l_{copy}\}$
18: **end for**
19: $a_{lpChCp} = \mathsf{actCopy}(\mathfrak{p}, \mathfrak{p}_{\#}, a_{lpChild})$ mit $a_{lpChild} \in \text{CHILDREN}(a_{loop})$
20: **return** a_{lpChCp}
21: **end function**

Verzeichnis der Mengen, Abbildungen und Funktionen

Dieses Kapitel gibt einen Überblick über die in der Ausarbeitung verwendeten Symbole, Menge, Abbildungen und Funktionen. In Tabelle A.1 sind die verwendeten mathematischen Symbole dargestellt.

Tabelle A.1.: Liste der mathematischen Symbole

Symbol	Beschreibung
π_i	Projektion des Elements eines Tupels an Position i
$\wp$	Die Potenzmenge
$\dot{\vee}$	Die logische Kontravalenz
$\perp$	undefiniert

Tabelle A.2 zeigt die verwendeten Mengen und deren Elemente.

Tabelle A.2.: Liste der Mengensymbole

Menge	Element	Beschreibung
C	c	Choreographien
$\mathfrak{P}$	$\mathfrak{p}$	Teilnehmer einer Choreographie
$\mathfrak{P}^{set}$	$\mathfrak{p}^{set}$	Teilnehmermengen einer Choreographie
ML	ml	Nachrichtenkanten einer Choreographie
P	p	Prozessmodelle (Verhaltensbeschreibungen) einer Choreographie
A	a	Aktivitäten eines Prozessmodells
HR	hr	Hierarchiebeziehungen eines Prozessmodells
T_{HR}	$\mathfrak{t}_{HR}$	Hierarchiebeziehungstypen, d.h. Kindaktivitätsbeziehung $\mathfrak{t}^{ch}_{HR}$ oder eine ereignisbasierte Beziehung
L	l	Kontrollflusskanten eines Prozessmodells
V	v	Variablen eines Prozessmodells
$\mathcal{B}$	—	Booleschen Werte
O	—	Abstrakte Objekte
$\mathfrak{C}$	$\mathfrak{c}$	Bedingungen eines Prozessmodells
$Ex_{\mathcal{B}}$	$ex_{\mathcal{B}}$	Boolesche Ausrücke eines Prozessmodells
$\mathcal{L}$	l	In den Kontrollflusskonstrukten eines Prozessmodells verwendete Bezeichner
E	e	Ereignisse eines Prozessmodells, es werden Standardereignisse E^{std}, Nachrichtenereignisse E^{msg}, Timer-Ereignisse E^{msg} und Fehlerereignisse E^{fault} unterschieden
T_E	$\mathfrak{t}_E$	Ereignistypen
CO_{rcv}	co_{rcv}	Empfangskonstrukte, d.h Receive-Aktivitäten oder Nachrichtenereignisse eines Prozessmodells
S	s	Die Menge der Scopes
C^F	c^F	Choreographiefragmente
P^F	p^F	Prozessmodellfragmente
L^F	l^F	Kontrollflusskantenfragmente
HR^F	hr^F	Hierarchiebeziehungsfragmente
ML^F	ml^F	Nachrichtenkantenfragmente
$\mathcal{K}$	K	Konversationen
P^I	p^I	Prozessinstanzen
A^I	a^I	Aktivitätsinstanzen

L^I	l^I	Kontrollflusskanteninstanzen
Υ	υ	Aktivitätszustände, die Zustände in der Menge können mit einem Subskript kategorisiert werden
Λ	$-$	Zustände die Kontrollflusskanteninstanzen einnehmen können.
E^{Υ}	e^{Υ}	Zustandstransitionsereignisse von Aktivitätsinstanzen
$\mathbb{T}$	t	Zeitschritte
T	τ	Traces, die vollständigen Traces werden durch $\hat{T}$ gekennzeichnet
Δ	δ	Zustandstransitionen
Ψ	δ	Zustandstransitionspfade

Die in der Ausarbeitung verwendeten Funktionen sind in Tabelle A.3 dargestellt.

Tabelle A.3.: Liste der Funktionen

Funktion	Beschreibung
$\textsc{Ancestors} : (A \cup E^{msg}) \rightarrow \wp(A)$	Bestimmt alle Vorfahren also direkte und indirekte Elternaktivitäten der Aktivität oder des Nachrichtenereignis.
$\textsc{Children} : A_{\texttt{struc}} \rightarrow \wp(A)$	Ermittelt die direkten Kindaktivitäten einer Aktivität.
$\textsc{Descendants} : A_{\texttt{struc}} \rightarrow \wp(A)$	Ermittelt die Nachfahren, d.h. die direkten und indirekten Kindaktivitäten einer Aktivität.
$\textsc{Parent} : A \cup E^{msg} \rightarrow A \cup \{\perp\}$	Gibt die Elternaktivität einer Aktivität zurück oder $\perp$, falls die Aktivität keine Elternaktivität besitzt. Wird ein Nachrichtenereignis übergeben, wird die Pick-Aktivität zurück gegeben, die dem Nachrichtenereignis zugeordnet ist.
$\textsc{ParentScope} : (A \cup E^{msg}) \rightarrow S)$	Gibt den direkten Eltern-Scope einer Aktivität oder eines Nachrichtenereignisses zurück.
$\textsc{Incoming}_L : A \rightarrow \wp(L)$	Gibt alle eingehenden Kanten einer Aktivität zurück.

$\textsc{Outgoing}_L : A \rightarrow \wp(L)$	Gibt alle ausgehenden Kanten einer Aktivität zurück.
$\textsc{Predecessors} : A \rightarrow \wp(A)$	Ermittelt die direkten Vorgängeraktivitäten einer Aktivität.
$\textsc{Predecessors}_{all} : A \rightarrow \wp(A)$	Gibt die direkten und indirekten Vorgängeraktivitäten einer Aktivität zurück.
$\textsc{Successors} : A \rightarrow \wp(A)$	Gibt die direkten Nachfolgeaktivitäten einer Aktivität zurück.
$\textsc{Successors}_{all} : A \rightarrow \wp(A)$	Gibt die direkten und indirekten Nachfolgeaktivitäten einer Aktivität zurück.
$\textsc{child}_{E^{timer}} : S \rightarrow A$	Bestimmt die Ereignisbehandlungsaktivität im Event-Handler eines Scopes.
$\textsc{children}_{E^{fault}} : S \rightarrow \wp(A)$	Bestimmt die Fehlerbehandlungsaktivitäten eines Scopes.
$\textsc{child}_{pr} : S \rightarrow A$	Bestimmt die primäre Kindaktivität eines Scopes.
$\textsc{events} : A \times \Upsilon \times T \rightarrow \wp(E^{\Upsilon})$	Gibt die Zustandstransitionsereignisse einer Aktivität a aus einem Trace τ zurück, in denen die Instanzen der Aktivität dens Zustand υ erreicht haben.
$\mu : C \rightarrow p_{\mu}$	Die Funktion zur Konsolidierung der Prozessmodelle in einer Choreographie
$\phi : P \times \wp(F_{spec}) \rightarrow C$	Die Funktion zur Fragmentierung eines Prozessmodells in mehrere interagierende Prozessmodelle einer Choreographie basierend auf einer Fragmentspezifikation

Tabelle A.4.: Liste der mathematischen Abbildungen

Abbildung	Beschreibung
$\equiv_A \subseteq A_{\text{opaque}} \rightarrow A_{\text{opaque}}$	Definiert, dass zwei Geschäftsaktivitäten identisch sind, d.h. dass sie das gleiche Verhalten implementieren.
$\text{value}_V : V \rightarrow \mathfrak{P} \cup \mathcal{B} \cup O$	Weist einer Variablen ihren Wert zu.

$\mathsf{type}_E : E \to T_E$	Weist einem Ereignis seinen Typ zu.
$\mathsf{for} : E^{timer} \to \mathbb{T}$	Weist einem Timer-Ereignis eine Zeitdauer zu.
$\mathsf{message} : E^{msg} \to ML$	Weist einem Nachrichtenereignis einen Nachrichtenkante zu.
$\mathsf{joinCond} : A \to Ex_{\mathcal{B}}$	Weist einer Aktivität ihre Eintrittsbedingung zu.
$\mathsf{joinFault} : A \to Ex_{\mathcal{B}}$	Bestimmt, ob eine Aktivität einen Join-Fehler auslösen kann.
$\mathsf{type}_A : A \to T_A$	Weist einer Aktivität ihren Typ zu.
$\mathsf{sender} : CO_{rcv} \times \mathbb{N} \to \mathfrak{P}$	Weist einem Empfangskonstrukt einen Sender zu, von dem es ab einem bestimmten Ausführungsschritt Nachrichten empfangen soll.
$\mathsf{receiver} : A_{\texttt{invoke}} \times \mathbb{N} \to \mathfrak{P}$	Weist einem Invoke einen Empfänger zu, an den es ab einem bestimmten Ausführungsschritt Nachrichten senden soll.
$\mathsf{outputVar} : CO_{rcv} \to V$	Weist einem Empfangskonstrukt seine Ausgabevariable zu.
$\mathsf{inputVar} : A_{\texttt{invoke}} \to V$	Weist einem Invoke seine Eingabevariable zu.
$\mathsf{assign}_{src} : A_{\texttt{assign}} \to V \cup \mathfrak{P}$	Weist einem Assign sein Quellkonstrukt zu.
$\mathsf{assign}_{trg} : A_{\texttt{assign}} \to V \cup \mathfrak{P}$	Weist einem Assign sein Zielkonstrukt zu.
$\mathsf{timer} : A_{\texttt{wait}} \to E^{timer}$	Weist einem Wait sein Timer-Ereignis zu.
$\mathsf{throws} : A_{\texttt{throw}} \to E^{fault}$	Weist einem Throw sein Fehlerereignis zu.
$\mathsf{faultName} : E^{fault} \to \mathcal{L} \cup \{\bot\}$	Weist einem Fehlerereignis den Namen des Fehlers zu, auf den es reagiert.
$\mathsf{rethrow} : S \times E^{fault} \to \mathcal{B}$	Legt fest, ob ein Scope einen Fehler an den Eltern-Scope weiter propagiert.
$\mathsf{partSet} : A_{\texttt{forEach}} \to \mathfrak{P}^{set}$	Weist einem ForEach die Teilnehmermenge zu, über die es iterieren soll.
$\mathsf{cond}_{\texttt{while}} : A_{\texttt{while}} \to Ex_{\mathcal{B}}$	Weist einem While seine Schleifenbedingung zu.

$\mathsf{loop}_{\mathfrak{P}} : A_{\mathtt{loop}} \to \mathfrak{P}^{set}$	Weist einer Schleife die Teilnehmer zu, über die sie iteriert. Kann im Gegensatz zu partSet auch für While-Schleifen genutzt werden.
$\mathsf{conversations} : C \to \wp(\mathcal{K})$	Weist einer Choreographie die von ihr erstellten Konversationen zu.
$\mathsf{inst}_A : A \to \wp(A^I)$	Weist einer Aktivität die von ihr erstellten Aktivitäten zu.
$\mathsf{inst}_L : L \to \wp(L^I)$	Weist einer Aktivität die von ihr erstellten Aktivitäten zu.
$\mathsf{history}_{\mathsf{c}} : C \to \wp(\hat{T})$	Weist einer Choreographie ihre Historie zu, d.h. die Menge aller vollständigen Traces, die ihre Aktivitäten erzeugen können.
$\mathsf{history}_{\mathsf{p}} : C \to \wp(\hat{T})$	Weist einem Prozessmodell seine Historie zu, d.h. die Menge aller vollständigen Traces, die seine Aktivitäten erzeugen können.
$\mathsf{history}_{\mathsf{a}} : A \to \wp(\hat{T})$	Weist einer Aktivität ihre Historie zu, d.h. die Menge aller vollständigen Traces, die sie und ihre Nachfahren erzeugen können.
$\omega_{\Upsilon} : A \times \Upsilon \times A \times \Upsilon \to \Gamma$	Legt eine Aktivitätszustandsbeziehung zwischen den Zuständen von zwei Aktivitäten fest.
$\mathsf{profile} : A \times A \to \bigcup \omega_{\Upsilon}$	Legt die Zustandsbeziehungen zwischen allen Zuständen von zwei Aktivitäten fest.
$\mathsf{participantScope} : \mathfrak{P} \to A_{\mathtt{scope}}$	Weist einem Teilnehmer seinen Teilnehmer-Scope im konsolidierten Prozessmodell zu.
$\mathsf{condCopy} : \mathfrak{P} \times \mathfrak{C} \to \mathfrak{C}$	Weist einer Bedingung ihr Duplikat im konsolidierten Prozessmodell zu.
$\mathsf{eventCopy} : \mathfrak{P} \times E \to E$	Weist einem Ereignis sein Duplikat im konsolidierten Prozessmodell zu.
$\mathsf{varCopy} : \mathfrak{P} \times V \to V$	Weist einer Variable ihr Duplikat im konsolidierten Prozessmodell zu.

$\mathsf{actCopy} : \mathfrak{P} \times (\mathfrak{P} \cup \{\bot\}) \times A \rightarrow A$	Weist einer Aktivität ihr Duplikat im konsolidierten Prozessmodell und in der angegebenen Teilnehmeriteration zu.
$\mathsf{hrCopy} : \mathfrak{P} \times (\mathfrak{P} \cup \{\bot\}) \times \mathfrak{C} \rightarrow \mathfrak{C}$	Weist einer Hierarchiebeziehung ihr Duplikat im konsolidierten und in der angegebenen Teilnehmeriteration zu.
$\mathsf{clCopy} : \mathfrak{P} \times (\mathfrak{P} \cup \{\bot\}) \times \mathfrak{C} \rightarrow \mathfrak{C}$	Weist einer Bedingung ihr Duplikat im konsolidierten Prozessmodell und in der angegebenen Teilnehmeriteration zu.
$\mathsf{com2Syn} : A_{com} \cup CO_{rcv} \rightarrow SYN \cup \{\bot\}$	Weist einer Kommunikationsaktivität oder einem Empfangskonstrukt keine ($\bot$) oder eine Synchronisationsaktivität zu, durch die sie im konsolidierten Prozessmodell ersetzt wurde.
$\mathsf{pickScope} : A_{\texttt{pick}} \rightarrow A_{\texttt{scope}}$	Weist einer Pick-Aktivität den Scope zu, durch den sie ersetzt wurde.