View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Online Publikationen der Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3421

Choreographie-basierte Konsolidierung von BPEL Prozessmodellen

Peter Debicki

Studiengang: Softwaretechnik

Prüfer: Prof. Dr. Frank Leymann

Betreuer: Dipl.-Inf. Sebastian Wagner

begonnen am: 16. August 2012

beendet am: 15. Februar 2013

CR-Klassifikation: H.4.1, K.1

Kurzfassung

Wagner et al. zeigen ein Konzept zur Choreographie-basierten Konsolidierung von Prozessmodellen. Die vorliegende Diplomarbeit konkretisiert das technische Vorgehen in Form eines erweiterbaren Prototyps. Als Eingabe dient eine BPEL4Chor Choreographie sowie die zugehörigen technischen Fragmente in Form von WSDL-Dateien. Die Kommunikationsmuster der Choreographieteilnehmer werden anhand eines Katalogs von Konsolidierungsmustern analsysiert und in einen neuen ausführbaren BPEL-Prozess zusammengeführt. Hierbei werden der ursprüngliche Kontrollfluss der Aktivitäten der Choreographie sowie die Datenflussabhängigkeiten im neuen erzeugten BPEL-Prozess weitestgehend erhalten. Je nach verwendetem Kommunikationsmuster, synchron oder asynchron, werden verschiedene Konsolidierungsoperationen an den teilnehmenden Aktivitäten durchgeführt. Das Ergebnis ist ein BPEL-Prozess der eine äquivalente Kontroll- sowie Datenflusssemantik, wie die ursprüngliche Choreographie besitzt, jedoch bezüglich Laufzeit und Speicherverbrauch eine optimierte Leistung aufweist.

Inhalt

Abbildungsverzeichnis 7
Auflistungsverzeichnis 9
Tabellenverzeichnis 10
1 Einleitung 11
1.1 Ziele der vorliegenden Diplomarbeit 12
1.2 Kapitelübersicht und Aufbau 13
1.3 Motivation 13
1.4 Verwandte Arbeiten 13
1.5 Aufgabenstellung 14
2 Grundlagen und Technologien 15
2.1 Web Services 15
2.2 Web Services Business Process Execution Language 2.0 17
2.2.1 Grundlegende Konzepte von WS-BPEL 2.0 17
2.2.1.1 Abstrakte und ausführbare Prozesse 18
2.2.1.2 Basis Aktivitäten von WS-BPEL 2.0 20
2.2.1.3 Strukturierte Aktivitäten von WS-BPEL 2.0 21
2.2.1.4 Scopes und Handler 23
2.3 BPEL4Chor 24
2.3.1 Beispielchoreographie 25
2.4 Allen-Kalkül 27
3 Konsolidierung von BPEL4Chor-Choreographien 28
3.1 Zustandsmodell für WS-BPEL 2.0 Prozesse sowie Aktivitäten 28
3.1.1 Prozess Instanz Zustandsmodell 28
3.1.2 Aktivitäts-Zustandsmodell 29
3.1.3 <scope>-Aktivitäts-Zustandsmodell 31
3.1.4 <invoke>-Aktivitäts-Zustandsmodell 34
3.1.5 Schleifen-Zustandsmodell 35
3.1.6 Link-Zustandsmodell 35
3.2 Formales Vorgehen bei der choreographiebasierten Konsolidierung von BPEL-Prozessen ... 36
3.2.1 Anlegen des konsolidierten BPEL-Prozesses 37
3.2.1.1 Übernehmen der Fault Handler in konsolidierten Prozess 39
3.2.2 Generierung des Kontrollflusses 40
3.2.2.1 Anpassung der `join`- und `transitionCondition` während der Konsolidierung 50
3.2.2.2 Peer-Scope-Dependency Problematik 52
3.2.3 Generierung des Datenflusses 53
3.2.3.1 Voraussetzungen für den korrekten Datenfluss 56
3.2.3.2 Auswirkungen der Konsolidierung auf die verwendeten CorrelationSets 58

3.3 Taxonomie der Konsolidierungsmuster („Merge-Patterns“) 61
3.3.1 Asynchrone Merge-Patterns 61
3.3.1.1 AsyncPattern1.1 62
3.3.1.2 AsyncPattern1.2 65
3.3.1.3 AsyncPattern1.3 66
3.3.1.4 AsyncPattern1.4 68
3.3.1.5 AsyncPattern1.5 68
3.3.1.6 AsyncPattern1.6 69
3.3.1.7 AsyncPattern1.7 („Khalaf Split“) 70
3.3.1.8 AsyncPattern1.8 (Asynchrones n-zu-1 Senden auf `<receive>`) 72
3.3.1.9 AsyncPattern2.1 74
3.3.1.10 AsyncPattern2.2 (Asynchrones n-zu-1 Senden auf `<pick>`) 77
3.3.1.11 AsyncPattern2.3 (Asynchrones n-zu-1 Senden auf einen `<onMessage>`-Zweig) 79
3.3.1.12 AsyncPattern3.0 („Non-Merge-Pattern-Async“) 81
3.3.2 Synchrone Merge-Patterns 85
3.3.2.1 SyncPattern1.1 86
3.3.2.2 SyncPattern1.2 87
3.3.2.3 SyncPattern1.3 88
3.3.2.4 SyncPattern1.4 (Multiple `<reply>`-Aktivitäten) 89
3.3.2.5 SyncPattern1.5 (Sendende `<invoke>`-Aktivität innerhalb von Handlern) 91
3.3.2.6 SyncPattern2.1 (`<onMessage>`-Zweig als `receiveActivity`) 92
3.3.2.7 SyncPattern2.2 93
3.3.2.8 SyncPattern2.3 93
3.3.2.9 SyncPattern2.4 93
3.3.2.10 SyncPattern3.0 („Non-Merge-Pattern-Sync“) 94
3.4 Vervollständigung der technischen Artefakte im neuen konsolidierten Prozess und Übernahme der WSDLs 95
3.4.1 Einfügen der WSDL-Dateien per `import`-Statements 96
3.4.2 Anpassung der Korrelationsmengen bei mehreren initialen Startaktivitäten 96
3.4.3 Erzeugen und Hinzufügen der `PartnerLinks` für die nicht konsolidierten Message Links aus *NMML* 97
3.4.4 Technische Vervollständigung der initialen Startaktivitäten sowie der inter-prozess kommunizierenden 98
4 Implementierung 100
4.1 Eingesetzte Technologien 100
4.1.1 StAX 100
4.1.2 Eclipse IDE 100
4.1.3 Eclipse Modeling Framework (EMF) 100
4.2 Vorgehen und Architektur 101
4.3 Erweiterbarkeit der Patterns 105

5 Zusammenfassung und Ausblick 106
5.1 Ausblick 107
Literaturverzeichnis 108
Erklärung 112

Abbildungsverzeichnis

1.1 Herstellung eines Sportwagens........ 12
1.2 Versuchsaufbau in der Community Cloud........ 12
1.3 BPEL4Chor Konsolidierung........ 14
2.1 Inhalt einer WSDL-Datei........ 16
2.2 Aufbau eine SOAP-Nachricht........ 16
2.3 Artefakte eines BPEL Prozesses........ 18
2.4 Verbindung WSDL → BPEL........ 18
2.5 Kommunikationsmuster zwischen Prozesspartnern........ 19
2.6 Externe Sicht eines Prozesses........ 19
2.7 Grundgerüst Computerkauf mit ausführbarer Vervollständigung........ 19
2.8 Links und ihre Semantik........ 22
2.9 BPEL4Chor Artefakte........ 24
2.10 Choreographiebeispiel........ 25
2.11 Die 13 Relationen des Allen-Kalküls........ 27
2.12 Vergleich der Kontrollflussrelationen zweier Beispielfragmente........ 27
3.1 Prozess Instanz Zustandsmodell........ 28
3.2 Aktivitäts-Zustandsmodell........ 29
3.3 `<scope>`-Aktivitäts-Zustandsmodell........ 31
3.4 `<invoke>`-Aktivitäts-Zustandsmodell........ 34
3.5 Schleifen-Zustandsmodell........ 35
3.6 Link-Zustandsmodell........ 35
3.61 Anlegen des konsolidierten BPEL-Prozesses........ 37
3.61b Verändertes Fehlerverhalten in konsolidiertem Prozess........ 39
3.61c Ein `<catchAll>`-Fault Handler mit einer `<compensate>`-Aktivität........ 40
3.7 Asynchrone Konsolidierung Variante 1........ 42
3.9 AsyncMerge Variante 1........ 43
3.10 AsyncMerge Variante 1 Beispiel 2........ 43
3.11 Vervielfachung der Links........ 43
3.12 Veränderter Kontrollfluss bei Variante 1........ 44
3.13 Propagieren des `suppressJoinFailure`-Attribut-Wertes........ 45
3.14 Asynchrone Konsolidierung Variante 2........ 45
3.15 AsyncMerge Variante 2........ 46

3.16 Synchrone Konsolidierung Variante 1 … 47
3.17 SyncMerge Variante 1 … 48
3.18a Variante 1 mit nur einer <assign>-Aktivität Teil 1 … 49
3.18b Variante 1 mit nur einer <assign>-Aktivität Teil 2 … 49
3.19 SyncMerge Variante 2 … 50
3.20 Übernahme der <source>-Elemente aus ursprünglichen Aktivitäten … 51
3.21 Anpassungen der joinCondition beim Konsolidieren … 52
3.22 Kontrollflussabhängigkeit zweier Scopes … 53
3.23 Zyklus in Partnerscopeabhängkeiten … 53
3.24 Austausch der kommunizierenden Aktivitäten durch <assign>-Aktivitäten … 54
3.25 Optimierungen des Datenflusses im konsolidierten Prozess … 55
3.25b Race Condition … 56
3.25c Behebung des Lost Update Problems … 56
3.26 Syntax einer Korrelationseigenschaft … 58
3.27 Verwendung mehrerer initialer Startaktivitäten … 59
3.28a Beispielfragmente aus Korrelationsbeispielchoreographie … 59
3.29 Anpassung der initiate-Attribute … 61
3.30 Anwendung des Merge-Algorithmus … 62
3.31 AsyncPattern1.1 … 62
3.32 <invoke> mit FH und CH … 63
3.32b <empty>-Optimierung … 63
3.32c Fallbeispiele einiger <empty>-Optimierungen … 65
3.33 AsyncPattern1.2 … 66
3.34 AsyncPattern1.3 … 66
3.35 AsyncPattern1.4 … 68
3.36 AsyncPattern1.5 … 68
3.37 AsyncPattern1.6 … 69
3.38 Erweiterung von <assign>-Aktivität *a* durch zusätzlichen <copy>-Block … 70
3.39 AsyncPattern1.7: Aufspalten eines Kontrollflusslinks … 70
3.40 AsyncPattern1.7 angewendet auf Cui's Kontrollflusslinkfragmentierung … 71
3.41 n-zu-1 Senden sowie das zugehörige Topology-Fragment … 72
3.42 Anwendung des AsyncPattern1.1 auf mehrere Message Links … 72
3.43 Asynchrones n-zu-1 Senden … 73
3.44 Ergebnis der Konsolidierung des n-zu-1 AsyncPattern1.8 … 73
3.45 AsyncPattern2.1 mit einem <onMessage>-Zweig … 74
3.46 AsyncPattern2.1 mit Schutzvariable $v_{pick_activated}$ … 76
3.47 AsyncPattern2.1 mit einer <pick>-Aktivität mit weiteren <onMessage>-Zweigen … 76
3.48 Syntaktische Umwandlung eines <onAlarm>-Zweigs in eine <wait>-Aktivität … 76
3.49 AsyncPattern2.1 für <pick>-Aktivität mit <onAlarm>-Zweig … 77

3.50	AsyncPattern2.2 mit zwei sendenden PBDs………………………………………..	77
3.51	AsyncPattern2.2 mit einem weiteren choreographie-extern kommunizierenden `<on-Message>`-Zweig………………………………………………………………..	78
3.52	AsyncPattern2.3 Konsolidierung einer Choreographie mit zwei sendenden PBDs auf den gleichen `<onMessage>`-Zweig *msg1* in `<pick>` *p*…………………………….	79
3.53	Konsolidierung einer Choreographie mit •*p*=∅ sowie `createInstance="yes"` von *p* mit Async-Pattern2.1…………………………………………………………	80
3.54	Syntaktische Umformung eines `<onMessage>`-Zweigs in eine äquivalente `<receive>`-Aktivität………………………………………………………………………..	81
3.55	AsyncPattern3.0 Muster…………………………………………………………	82
3.56	Rekursive Untersuchung von *par(s)*……………………………………………...	83
3.57	Sendende Aktivität `<invoke>` *s* und empfangende Aktivität `<receive>` *r* und eine Konsolidierung ohne Kontrollflusslink……………………………………………..	84
3.58	Sendende Aktivität `<invoke>` *s* und empfangende Aktivität `<receive>` *r* und eine Konsolidierung ohne Kontrollflusslink mit zusätzlicher Variable v_t…………………..	85
3.59	Anwendung des Merge-Algorithmus…………………………………………….	86
3.60	SyncPattern1.1……………………………………………………………………	86
3.62	SyncPattern1.2……………………………………………………………………	88
3.63	SyncPattern1.3……………………………………………………………………	88
3.64	SyncPattern1.4 mit zwei `<reply>`-Aktivitäten und einem Fault……………………..	89
3.65	SyncPattern1.4 mit zwei `<reply>`-Aktivitäten……………………………………….	91
3.66	SyncPattern1.5……………………………………………………………………	92
3.67	SyncPattern2.1……………………………………………………………………	93
3.68	SyncPattern2.4……………………………………………………………………	94
3.69	Hinzufügen der WSDL-`imports` in den konsolidierten Prozess……………………...	96
3.70	Hinzufügen der `PartnerLinks` sowie der technischen Attribute `portType`, `operation` sowie `partnerLink` in die asynchron intra-prozess kommunizierenden Aktivitäten………………………………………………………………………..	97
3.71	Hinzufügen der technischen Artefakte bei synchron intra-prozess kommunizierenden Aktivitäten………………………………………………………………………..	98
4.1	Schritte beim Vorgehen der Konsolidierung einer BPEL4Chor-Choreographie……...	101
4.2	Die mergeChoreography-Komponente und ihre Abhängigkeiten……………………..	101
4.3a	ChoreographyMerger-Klassendiagramm…………………………………………...	102
4.3b	Async- bzw. SyncMatcher sowie die Beziehungen zu den Async- bzw. SyncPatterns und den entsprechenden Schnittstellen (Interfaces) und der abstrakten Klassse MergePattern………………………………………………………………………...	103
4.4	Sequenzdiagramm für das Auffinden der MergePatterns……………………………...	104

Auflistungsverzeichnis

2.1	PBD des Reisebüros……………………………………………………………...	25
2.2	Participant Topology……………………………………………………………..	25

2.3 Participant Grounding……………………………………………………………. 25
3.0 Kopieren der PBDs in neue <scope>-Aktivitäten des neuen BPEL-Prozesses……… 38
3.1 Pseudocode für Merge-Algorithmus…………………………………………………... 41
3.2 Pseudocode asyncMerge-Methode Variante 1………………………………………... 42
3.3 Pseudocode asyncMerge-Methode Variante 2………………………………………... 46
3.4 Pseudocode syncMerge-Methode Variante 1…………………………………………. 48
3.5 Pseudocode <empty>-Optimierungsalgorithmus…………………………………….. 64
3.6 Pseudocode *jc(act)*-Funktion……………………………………………………….. 65
3.7 Pseudocode *replaceVar* (v_1, v_2, act)-Funktion………………………………………… 68
3.8 Pseudocode *isActivityInHandler*(a_{invoke})-Funktion…………………………………….. 83

Tabellenverzeichnis

2.1 Die 13 Intervallrelationen und ihre Bedeutungen……………………………………... 27
3.1 Zustände und Übergänge des Prozess Instanz Modells……………………………….. 29
3.2 Zustände und Übergänge des Aktivitäts-Zustandsmodells……………………………. 31
3.3 Zustände und Übergänge des <scope>-Aktivitäts-Zustandsmodells………………… 34
3.4 Zustände und Übergänge des Link-Zustandsmodells…………………………………. 36
3.5 Notation für Funktionen und Mengen des Konsolidierungs-Algorithmus……………. 41
3.6 Notation für Funktionen und Mengen des Konsolidierungs-Algorithmus (Erweiterungen)……………………………………………………………………………… 47

1 Einleitung

Cloud-Computing [MT11] bietet Unternehmen neue Möglichkeiten zur Realisierung ihrer Geschäftsprozesse. Durch die Verlagerung von Teilen der Geschäftsprozesse in die *Cloud* reduziert sich der Bedarf an firmeninternem, qualifiziertem technischem Knowhow zum Betrieb, der Wartung sowie Pflege der IT-Ressourcen. Das Unternehmen kann sich wieder auf seine eigenen nicht IT-getriebenen Geschäftsziele konzentrieren. So kann man mittlerweile in zahlreichen großen Unternehmen die Auslagerung von Teilen der Geschäftsprozesse beobachten, wie z.B. der Gehaltsabrechnung oder Supportprozessen. Ohne diese nicht wettbewerbsdifferenzierenden Teilprozesse kann sich das Unternehmen wieder auf sein Hauptgeschäft fokussieren. Hinzu kommen weitere Kostenvorteile, wie sie beispielsweise das *Pay-Per-Use* Kostenmodell anbietet, bei dem das Unternehmen nur für die Kapazitäten bezahlt, die auch effektiv genutzt werden. Man kann Parallelen zur Ablösung des Mainframes durch die Client/Server Architektur in den 1980ern erkennen, die auf einen Bedarf nach einer bereichs- bzw. branchenorientierten organisatorischen Struktur zurückzuführen war.

Ein weiteres vielversprechendes Konzept des Cloud-Computing ist die *Community Cloud* [MT11]: Ein Zusammenschluss von Unternehmen oder Organisationen der gleichen Branche mit dem Ziel, aus dem Verbund ihrer *Private Clouds* [MT11] eine gemeinsame Nutzung der Infrastruktur zu ermöglichen. In diesem Verbund können IT-Ressourcen dynamisch bereitgestellt (*Elastizität*), bedarfsgerecht abgerechnet (*Pay-Per-Use*) und durch Virtualisierung standardisiert werden. Durch die Nutzung der gemeinsamen Infrastruktur können die Kosten aller beteiligten Unternehmen gesenkt werden, da diese unter den Mitgliedern der Community Cloud aufgeteilt werden. Zusätzlich kann die Community Cloud Werkzeuge anbieten, die der Zusammenarbeit der Unternehmen dienen [WKL11], wie z.B. Content Management Services.

Um eine möglichst effiziente Zusammenarbeit in der Community Cloud zu ermöglichen, müssen auch die Geschäftsprozesse der beteiligten Unternehmen miteinander interagieren. Diese Interaktion kann durch Choreographien der Geschäftsprozesse dargestellt werden, die den Kontroll- sowie Datenfluss zwischen den Parteien modellieren. Natürlich wollen die Unternehmen die wettbewerbsdifferenzierenden Arbeitsabläufe ihrer Geschäftsprozesse auch in einem solchen Zusammenschluss nicht jedermann zugänglich machen. Somit müssen zunächst die für eine Nutzung in der Community Cloud geeigneten Prozessfragmente isoliert werden.

Wir wollen im folgenden Abschnitt anhand eines kleinen Szenarios für einen solchen Anwendungsfall aus der Automobilbranche das Vorgehen skizzieren (vgl. [WKL11]). Abbildung 1.1 zeigt eine Choreographie als *BPMN 2.0 Collaboration Diagramm* [OMG11] zwischen zwei Geschäftsprozessen zur Herstellung eines Sportwagens. Der eine Prozess stellt den Arbeitsablauf auf Seiten des Autoherstellers (AH) dar, der zweite den eines Zulieferers, der für die Herstellung des Motors zuständig ist (MH). Zu Beginn bestellt der AH einen Motor beim MH. Anschließend entwickeln beide Beteiligten einen Prototyp. Der AH einen Chassis-Prototyp und der MH einen Motorenprototyp. Diese Aktivitäten werden durch die aufklappbaren Teilprozesse dargestellt. Nach der Entwicklung des Chassis sowie des Motors wird letzterer an den AH geliefert, der in Zusammenarbeit mit Ingenieuren des MH einen ersten Prototyp des Sportwagens baut. Nun wird der Prototyp in einer Versuchsreihe beim AH auf seine Verkehrstauglichkeit geprüft. Hierzu können beispielsweise Versuche im Windkanal oder Crashtests gehören. Die Ergebnisse dieser Versuchsreihe werden an den MH geschickt. Beide Parteien analysieren die Ergebnisse und führen mögliche Optimierungen am Chassis und am Motor durch (dies ist in der Abbildung nicht dargestellt). Anschließend wird die Versuchsreihe wiederholt. Nach Fertigstellung der Versuchsreihen mit optimalen Ergebnissen, wird der Sportwagen hergestellt.
In diesem Herstellungsprozess testen und entwickeln beide Parteien den Prototyp beim MH. Um Ingenieuren beider Parteien einen möglichst effizienten Zugriff auf die IT Infrastruktur sowie den Zugang

zu den Testergebnissen zu ermöglichen, entscheiden sich die Unternehmen die für die Entwicklung sowie den Test des Sportwagenprototyps notwendigen IT Ressourcen in einer Community Cloud zusammenzuschließen. Diese Prozessfragmente sind in Abbildung 1.1 mit einem roten Kasten hervorgehoben (AH2 sowie MH2). Die Community Cloud stellt hierfür eine Workflow Engine zur Verfügung.

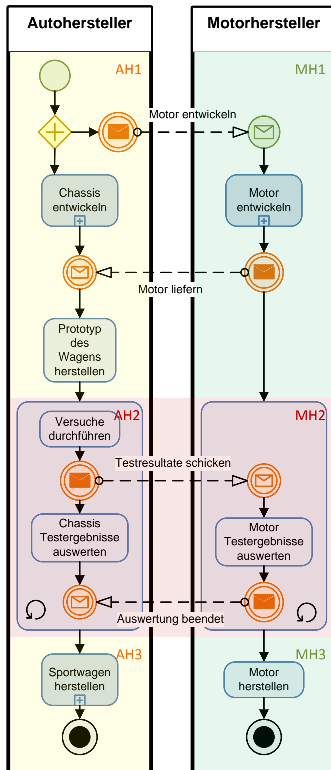


Abbildung 1.1 Herstellung eines Sportwagens (vgl. [WKL11])

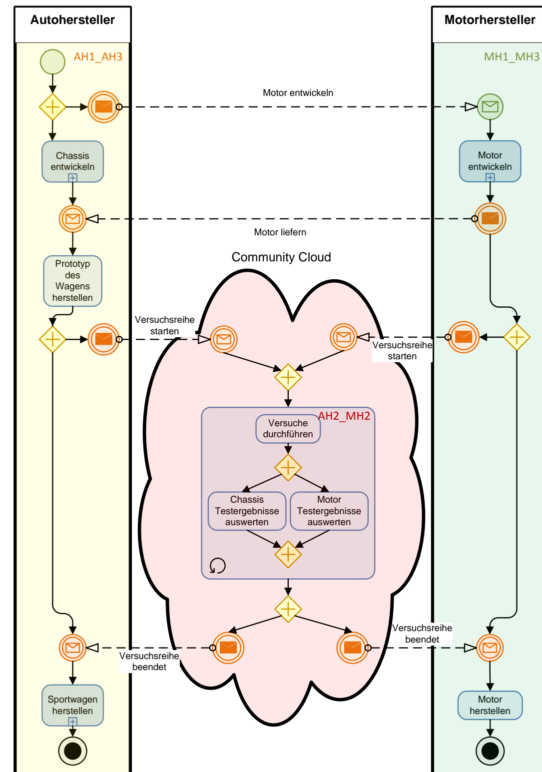


Abbildung 1.2 Versuchsaufbau in der Community Cloud

Da die beiden Parteien die Herstellungsprozesse ihrer jeweiligen spezifischen Produkte nicht füreinander zugänglich machen wollen, werden diese vor Ort auf der unternehmenseigenen IT Infrastruktur ausgeführt. Diese Prozessfragmente sind durch die vier Bereiche AH1, AH3, MH1 sowie MH3 dargestellt.

Abbildung 1.2 zeigt die Choreographie aus vorhergehendem Beispiel mit Einbeziehung einer Community Cloud: Die neuen Prozesse AH1_AH3 sowie M1_MH3 werden auf der unternehmenseigenen IT Infrastruktur ausgeführt, die Prozessfragmente die für die gemeinsame Versuchsreihe zuständig sind, wurden dagegen aus den ursprünglichen Geschäftsprozessen abgespalten („*split*“) und in einem neuen Prozesse zusammengelegt („*merged*“), der in der Community Cloud bereitgestellt und betrieben wird. Die vorliegende Diplomarbeit befasst sich mit dem konsolidieren oder zusammenlegen von Prozessfragmenten, die in Form einer BPEL4Chor Choreographie (Abschnitt 2.3) vorliegen, zu einem neuen ausführbaren Geschäftsprozess.

1.1 Ziele der vorliegenden Diplomarbeit

Das Ziel der vorliegenden Diplomarbeit ist die Erarbeitung und Umsetzung eines Konsolidierungsalgorithmus für BPEL4Chor [DKLW07] Choreographien sowie seiner anschließenden Implementierung als Eclipse Plugin [ECL12]. Das Ergebnis dieser Konsolidierung soll ein bereitstellbarer sowie ausführbarer WS-BPEL 2.0 Prozess [OAS07] sein. Zusätzlich soll gezeigt werden, dass nach Anwendung

der Konsolidierung der Daten- als auch der Kontrollfluss des neuen Prozesses dem der ursprünglichen Choreographie weitestgehend entspricht.

1.2 Kapitelübersicht und Aufbau

In Kapitel 1 wurde zunächst ein kleines Beispiel eines praktischen Konsolidierungsfalls gegeben. Anschließend wurden die Ziele der vorliegenden Diplomarbeit (1.1) verdeutlicht. Es folgt die Motivation (1.3), die zu dieser Arbeit geführt hat sowie verwandte Arbeiten (1.4). Abschnitt 1.5 skizziert nochmals die Aufgabenstellung. Kapitel 2 geht auf die technischen Grundlagen ein, die zum Verständnis dieser Arbeit notwendig sind. Abschnitt 2.1 erklärt hierfür die Web Service Grundlagen. Anschließend stellt Abschnitt 2.2 den WS-BPEL 2.0 Standard vor, der zur Implementierung der hier vorgestellten Geschäftsprozesse dient. Abschnitt 2.3 erklärt die WS-BPEL 2.0 Erweiterung BPEL4Chor, die zur Implementierung von Geschäftsprozesschoreographien entwickelt wurde und Ausgangspunkt unseres Konsolidierungsalgorithmus ist. Kapitel 3 zeigt das Vorgehen und die Überlegungen, die zum Konsolidierungsalgorithmus geführt haben. Kapitel 4 erläutert die technische Implementierung, die hierfür eingesetzten Technologien sowie Frameworks. Kapitel 5 gibt eine kurze Zusammenfassung der Arbeit und einen Ausblick auf laufende und mögliche Weiterentwicklungen der hier vorgestellten Konzepte.

1.3 Motivation

Die Beweggründe, die zu dieser Diplomarbeit geführt haben basieren auf der Idee eine BPEL4Chor [DKLW07] Choreographie direkt ohne eine Zwischentransformation in andere Modelle, wie z.B. Ereignisgesteuerte Prozessketten (EPKs) [KNS92] oder Petri-Netze [PET62], in einen neuen ausführbaren WS-BPEL 2.0 [OAS07] Geschäftsprozess zu konsolidieren. Das Beispiel aus Abschnitt 1 liefert hierfür ein praktisches Anwendungsszenario: Ein naiver Ansatz würde zunächst die beiden Geschäftsprozesse in die Prozessfragmente AH1, AH2, AH3, MH1, MH2 sowie MH3 zerlegen [CUI12]. Ohne anschließende Konsolidierung würden dann AH1 und AH3 auf der IT Infrastruktur des Autoherstellerunternehmens betrieben, MH1 sowie MH3 entsprechend auf der IT Infrastruktur des Motorherstellerunternehmens. Die Prozesse AH2 sowie MH2 würden in der Community Cloud bereitgestellt. Dies hätte mindestens sechs Prozessinstanzen zur Folge anstatt der ursprünglichen zwei. Durch Konsolidierung können hier auf Seiten der Community Cloud Prozesse Kosten gespart werden, da Cloud Infrastruktur Anbieter für gewöhnlich eine bedarfsgerechte (*Pay-Per-Use*) Kostenabrechnung anbieten. Zusätzlich können ohne Konsolidierung Performanceeinbußen mit steigender Anzahl der Prozessinstanzen auf Seiten der beiden Unternehmen auftreten, beispielsweise im Hinblick auf die Workflow Engines oder die benutzten Datenbanken.
Auf der technischen Seite wird durch die Konsolidierung zusätzlich die choroegraphieinterne Kommunikation zwischen den Parteien vermieden: Es ist keine Serialisierung und Deserialisierung der SOAP-Nachrichten mehr notwendig, die Korrelation entfällt ebenfalls.

1.4 Verwandte Arbeiten

Das Beispiel aus Abschnitt 1 spaltet zunächst die ursprünglichen Geschäftsprozesse in verschiedene miteinander kommunizierende Prozessfragmente auf. Cui's [CUI12] Diplomarbeit zeigt hierfür einen Ansatz der auf Arbeiten von Khalaf [KKL08][KL06] sowie Kopp et. al. basiert [KKL08]. Ein weiterer Ansatz, der für die im Kapitel 3 vorgestellte Datenflussanalyse entnommen wurde basiert ebenfalls auf Kopp et. al. [KKL08]. Die dort gezeigte Strategie der Datenflussanalyse bezieht die Nebenläufigkeit von WS-BPEL 2.0 Prozessen sowie die Dead Path Elimination [OAS07] mit ein.

Vorliegende Diplomarbeit mit dem Schwerpunkt der Konsolidierung von BPEL4Chor Choreographien basiert auf den Ansätzen aus Wagner et. al. [WKL11]. Verwandte Arbeiten befassen sich überwiegend mit dem Zusammenführen semantisch äquivalenter Prozesse. Mendling et. al. konsolidieren in ihrer Arbeit [MS06] semantisch äquivalente Ereignisgesteuerte Prozessketten (EPKs) [KNS92]. Küster et. al. zeigen in ihrem Ansatz [KGF+08] die Möglichkeit Prozesse zusammenzuführen, die von einem gemeinsamen Basisprozess abgeleitet wurden. Basis für das in [KGF+08] vorgestellte Vorgehen ist eine Art Logbuch indem die Änderungsschritte festgehalten werden, die vom Basisprozess zu den neuen Prozessen geführt haben. Sun et. al. stellen in [SKY06] ein Vorgehen zur Konsolidierung vor, das auf Petri-Netzen [PET62] basiert. Im Gegensatz zu dem in der vorliegenden Diplomarbeit verwendeten Vorgehen, müssen in [SKY06] die Stellen an denen die Prozesse zusammengeführt werden manuell angeben werden.
Die hier verwendete Erweiterung von WS-BPEL 2.0 zur Modellierung von Choreographien wird in [DKLW07] von Decker et. al. vorgestellt. Weitere Ausführungen sowie der Vergleich zu anderen Choreographieerweiterungen sind in [DKLW09] von Decker et. al. zu finden. Barros et. al. [BDH05] gehen in ihrer Arbeit auf die grundsätzlichen Interaktionsmuster zwischen Web Services ein.

1.5 Aufgabenstellung

Abbildung 1.3 zeigt die in dieser Diplomarbeit umgesetzte Aufgabenstellung: Das hier entwickelte Eclipse Plugin [ECL12] nimmt eine BPEL4Chor Choreographie als Input (in Form einer ZIP-Datei), analysiert die dort enthaltenen Prozessfragmente bezüglich ihrer Daten- und Kontrollflussabhängigkeiten und erzeugt einen ausführbaren WS-BPEL 2.0 Prozess mit der zugehörigen WSDL-Datei als Output. Dieser konsolidierte Prozess ist auf jeder WS-BPEL 2.0 konformen BPEL-Engine bereitstell- sowie ausführbar und besitzt eine weitestgehend gleiche Daten- und Kontrollflusssemantik, wie die ursprüngliche BPEL4Chor Choreographie.

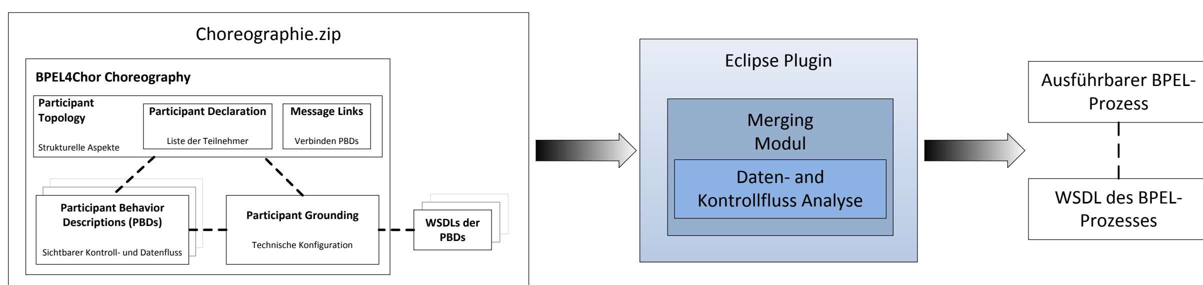


Abbildung 1.3 BPEL4Chor Konsolidierung: Eingabe für das Eclipse Merging Modul ist eine BPEL4Chor Choreographie sowie die WSDL Dateien der enthaltenen Prozessfragmente, die Ausgabe ist ein neuer ausführbarer BPEL-Prozess sowie die zu diesem gehörige WSDL Datei

2 Grundlagen und Technologien

In diesem Kapitel werden die Technologien und Grundlagen vorgestellt, die zum Verständnis der Konsolidierung von *BPEL-Choreographien* notwendig sind. Das Vorgehen in Kapitel 3 setzt voraus, dass der Leser mit den hier vorgestellten Technologien vertraut ist. Zunächst werden die Ideen hinter Web Services, Geschäftsprozessen sowie Geschäftsprozesschoreographien erklärt. Hierzu werden auch die relevanten Spezifikationen der zugrundeliegenden Protokolle, wie beispielsweise *SOAP* [W3C07], *WS-BPEL* [OAS07] sowie *BPEL4Chor* [DKLW07] erläutert. Auf die *WS-BPEL 2.0* [OAS07] Spezifikation wird etwas detaillierter eingegangen, da diese für das Verständnis der Konsolidierung einer BPEL4Chor-Choreographie essenziell ist.
Die vorgestellten Technologien und Konzepte werden nicht in voller Ausführlichkeit erklärt. Stattdessen konzentrieren wir uns auf die Details, die für das Verständnis der Diplomarbeit von Nutzen sind, ohne den Text mit zu vielen Kleinstdetails vollzustopfen. Den interessierten Lesern sei die referenzierte Literatur ans Herz gelegt.

2.1 Web Services

Web Services ermöglichen die Realisierung einer *Service-Oriented Architecture* (SOA) [WCL+05]. Web Services und die diese umgebenden Technologien werden vom *World Wide Web Consortium* (W3C) gefördert. Die Grundlage einer SOA sind lose gekoppelte Service. Ein Service ist eine Software Komponente, die über eine beliebige Netzwerktechnologie zur Verfügung gestellt wird. Dieser Service kann anschließend von anderen Programmen verwendet und mit diesen kombiniert werden. Ein Web Service stellt eine Schnittstelle zu einem Programm dar, welches die eigentliche Arbeit verrichtet und anschließend das Ergebnis über diesen Web Service zurückliefert, sofern es eines gibt. Web Service sind zustandslos unterstützen jedoch auch zustandsbehaftete langläufige Konversationen. Im Gegensatz zu anderen verteilten Technologien liegt der Vorteil von Web Services in der Tatsache, dass sie auf veröffentlichten und akzeptierten Standards basieren anstatt auf proprietären Lösungen. Diese Standards wurden in einem gemeinsamen Bestreben verschiedener Firmen erstellt und reflektieren so die Bedürfnisse der Industrie. Bedürfnisse wie beispielsweise Sicherheit, Ausfallsicherheit und Interoperabilität. Alle relevanten Standards im Bereich der Web Services basieren auf der *Extensible Markup Language* (XML) [W3C12]. Diese reichen von der Definition der Schnittstellen über das Format der transferierten Daten bis hin zu *Quality of Service* (QoS) Definitionen. Zwei der wichtigsten Standards ist die *Web Service Description Language* (WSDL) [W3C01], die die Schnittstelle des Web Service definiert sowie das *Simple Object Access Protocol* (SOAP) [W3C07], welches die verschickten und empfangenen Nachrichten eines Web Services definiert als auch deren Bearbeitung entlang des Nachrichtenkanals.

Wie in Abbildung 2.1 gezeigt enthält eine WSDL-Datei folgende Elemente, die hier kurz erläutert werden:

- **Types** (nicht in der Abbildung): Definitionen der benötigten Datentypen
- **Message**: Abstrakte Beschreibungen der ausgetauschten Daten
- **Operation**: Abstrakte Beschreibungen der vom Service unterstützten Aktionen. Man kann diese mit Methoden oder Funktionen vergleichen. Nachrichten bieten Eingabe- und Ausgabedaten für Operationen an.

- **PortType**: Abstrakte Menge der vom Service angebotenen Operationen

- **Binding**: Definieren ein konkretes Protokoll, wie beispielsweise SOAP über HTTP [W3C07], als auch die Datenformate der Nachrichten und Operationen eines PortType

- **Port**: Definieren die Adresse oder einen Verbindungspunkt zu dem Service, oftmals in Form einer URI

- **Service**: Eine Sammlung von Ports

Da WS-BPEL 2.0 die Version 1.1 von WSDL benutzt bezieht sich folgende Arbeit auf diese.

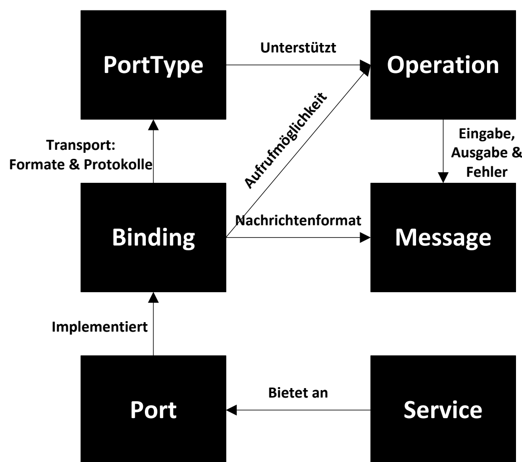


Abbildung 2.1 Inhalt einer WSDL-Datei (vgl. [LEY10b])

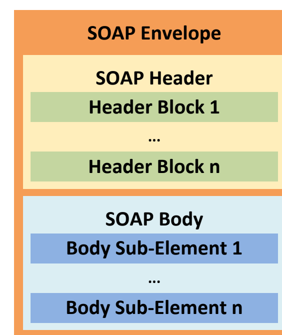


Abbildung 2.2 Aufbau einer SOAP Nachricht (vgl. [LEY10b])

Abbildung 2.2 zeigt schematisch den Aufbau einer SOAP Nachricht mit den folgenden Elementen:

- **SOAP Envelope**: Dies ist das Wurzelelement der XML-Dokuments („Briefumschlag"). Es enthält *SOAP-Header* sowie *SOAP-Body* Elemente.

- **SOAP Header**: Dies ist ein optionales Feld und muss bei Verwendung als erstes Feld im SOAP-Envelope definiert werden. Die SOAP-Spezifikation definiert nicht den Inhalt eines SOAP-Header. Der Header dient der Übertragung der Nutzdaten der Middleware, die für die Übertragung der Nachricht zuständig ist. Hier können beispielsweise Informationen, die für die sichere und ausfallfreie Übertragung der Nachricht relevant sind enthalten sein.

- **SOAP Body**: Dieser Teil enthält die eigentlichen Nutzdaten und muss in jeder Nachricht vorhanden sein.

Web Services verwenden zur Kommunikation Nachrichten. Aus diesem Grunde werden alle Daten zwischen Aufrufer und Aufgerufenem in Nachrichten verpackt. Abhängig vom gewählten Kommunikationsmuster werden die Nachrichten entsprechend ausgetauscht. Die Definitionen der üblichen Kommunikationsmuster können im Abschnitt 2.4 „Port Types" der WSDL Spezifikation [W3C01] nachgelesen werden. Die geläufigsten sind das Anfrage-Antwort Muster (Request-Response im Folgenden) bestehend aus einer Anfrage- sowie einer zugehörigen Antwortnachricht sowie das Einweg-Muster (One-Way im Folgenden) bestehend aus nur einer aufrufenden Anfragenachricht.

Ein Teil der Flexibilität, die man durch die Benutzung von Web Services erlangt, basiert auf der Tatsache, dass diese plattform- sowie transportunabhängig sind.

2.2 Web Services Business Process Execution Language 2.0

Die Web Services Business Process Execution Language 2.0, häufig auch als BPEL abgekürzt, oder WS-BPEL ist eine XML-basierte Sprache zur Beschreibung von Geschäftsprozessen basierend auf Web Services [WCL+05]. Da Web Services an sich keine Möglichkeit besitzen eine Logik zur Interaktion zu definieren wurde BPEL entwickelt. Sie erlaubt die Definition von Prozessen, die die Interaktionslogik als und mit Web Services erlauben. „*WS-BPEL definiert ein Model und eine Grammatik zur Beschreibung des Verhaltens eines Geschäftsprozesses basierend auf Interaktionen zwischen dem Prozess und seinen Partnern*“ [OAS07]. Der Kontrollfluss in einem BPEL Prozess ist explizit durch die Kontrolllinks zwischen den Aktivitäten modelliert, während der Datenfluss implizit durch die Benutzung der globalen und lokalen Variablen dargestellt wird. In BPEL werden die zu einem Prozess gehörigen Daten standardmäßig durch XPath Ausdrücke [W3C99a] gelesen und verändert. Sie bietet außerdem Parallelität sowie Death-Path-Elimination [OAS07] an. Seine Wurzeln hat BPEL in der graphbasierten Sprache Web Services Flow Language (IBM WSFL [LEY01]) sowie der blockbasierten Sprache XLANG (Microsoft XLANG [THA01]) und bietet hier eine hybride Alternative, da sie fluss- sowie operatorbasierte Modellierung unterstützt [WCL+05].

2.2.1 Grundlegende Konzepte von WS-BPEL 2.0

Der folgende Abschnitt bezieht sich überwiegend auf die Spezifikation aus [OAS07]. Eine BPEL Prozessbeschreibung besteht neben der WSDL Datei, die den Prozess nach Außen als Web Service zur Verfügung stellt, aus der eigentlichen Definition in XML, die folgende Grundelemente enthält (siehe Abbildung 3.3):

- **Partner Links und Partner Link Types**: Ein Partner Link stellt einen Kommunikationskanal zwischen zwei Partnern dar. Eine Partner Link Definition beinhaltet einen Partner Link Type und mindestens eine Rolle. Ein Partner Link Type beschreibt die Art des Nachrichtenaustauschs den der Prozess als Service ausführt und wird durch die Definition der Rollen, die jeder Service in der Interaktion einnimmt sowie durch die Spezifikation des Port Type, der durch den Service zur Verfügung gestellt wird um entsprechende Nachrichten zu empfangen, charakterisiert. Der Partner Link Type ist eine Erweiterung der WSDL. Abbildung 2.4 veranschaulicht den Zusammenhang zwischen WSDL und BPEL Definition.

- **Variables**: Ein Prozess empfängt, verändert und sendet Daten mithilfe von Variablen, die einen Teil des Laufzeitzustands des Prozesses speichern. Es gibt drei Möglichkeiten von Variablen: WSDL Message Type, XML Schema Type (Simple oder Complex) sowie XML Schema Elemente. Der Inhalt von Nachrichten, die zwischen Prozesspartnern ausgetauscht werden, wird im WSDL Message Type gespeichert.

- **Correlation Sets**: Correlation Sets dienen der Zuordnung einer Menge von Nachrichten zwischen Partnern, die an einer Interaktion beteiligt sind. Da Geschäftsprozesse durch die Repräsentation als Web Service zustandslos sind, benötigt die BPEL-Engine im Falle von mehreren nebenläufigen Instanzen eines Prozesses eine Möglichkeit der Zuordnung der Nachrichten an die entsprechende Instanz. Sie stellen eine Art Identifikationsschlüssel dar und sind nach ihrer einmaligen Initialisierung unveränderlich über die ganze Kommunikation hinweg.

- **Handler**: Ein BPEL Prozess kann zwei Arten von Handlern beinhalten um nach seiner Instanziierung auf Fehlersituationen sowie eingehende Nachrichten oder mögliche zeitgesteuerte Ereignisse zu reagieren: Fault Handler (FH) sowie für die letzten beiden Fälle Event Handler (EH). Im Abschnitt 2.2.1.4 wird die Bedeutung beider näher beleuchtet.

- **Activity**: Jeder Prozess muss mindestens eine Aktivität enthalten. Diese muss eine der aus Ausschnitt 2.2.1.2 beschriebenen Basisaktivitäten sein, oder ein Verbund aus den Strukturierten aus Abschnitt 2.2.1.3 sowie 2.2.1.4. Da der Life-Cycle eines BPEL Prozesses durch den Empfang einer initialen Nachricht beginnt, muss dieser mindestens eine Aktivität des Typs `<receive>` oder `<pick>` enthalten, welche das Attribut `createInstance` auf `true` gesetzt hat.

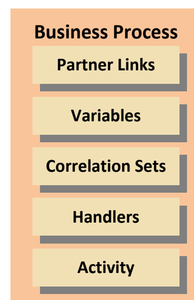


Abbildung 2.3 Artefakte eines BPEL Prozesses (vgl. [LEY10a])

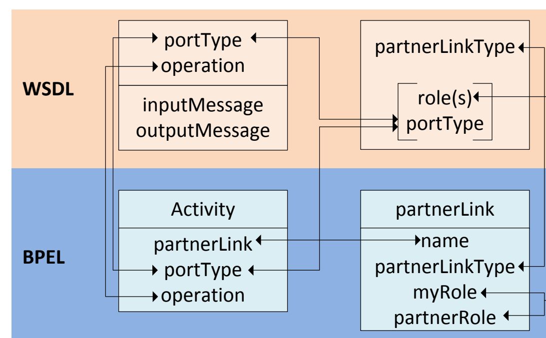


Abbildung 2.4 Verbindung WSDL → BPEL

2.2.1.1 Abstrakte und ausführbare Prozesse

WS-BPEL 2.0 erlaubt die Definition von ausführbaren sowie abstrakten Prozessen. Ausführbare Prozesse müssen alle in der WS-BPEL 2.0 Spezifikation [OAS07] definierten Attribute sowie Aktivitäten enthalten. In Verbindung mit ihren WSDL Dateien sind sie auf jeder standardkonformen BPEL-Engine bereitstell- und voll ausführbar. Ein abstrakter Prozess ist ein partiell definierter Prozess, der nicht ausführbar ist und verschiedene Details der konkreten operationalen Details, welche in den ausführbaren Prozessen enthalten sein müssen, unspezifiziert lässt [OAS07]. Da die im Abschnitt 2.3 beschriebene Erweiterung von WS-BPEL 2.0 auf abstrakten Prozessen basiert, werden diese hier etwas genauer beschrieben.
Abstrakte Prozesse müssen als solche explizit deklariert werden. Sie bieten zwei Möglichkeiten um operationale Details zu verstecken: Die Benutzung expliziter opaker Symbole („*opaque tokens*“) sowie die Auslassung ganzer Artefakte. Ein abstrakter Prozess kann die gesamte Menge aller Artefakte, die auch ein ausführbarer Prozess definiert, enthalten. Durch seine Deklaration als „*abstract*“ wird jedoch angezeigt, dass weitere Schritte notwendig sind um ihn zu einem voll ausführbaren Prozess zu konvertieren. Hierzu wird dem interessierten Leser der Abschnitt „*Executable Completion*“ (ausführbare Vervollständigung) der WS-BPEL 2.0 Spezifikation [OAS07] empfohlen. Es gibt verschiedene Anwendungsgebiete für abstrakte Prozesse:

- Sollen Bedingungen und Einschränkungen für den Nachrichtenaustausch zwischen Interaktionspartner von Geschäftsprozessen aufgezeigt werden, so werden nur die Kommunikationsmuster der einzelnen Partner angegeben und die internen Details der eigentlichen Prozesse bezüglich Datenhaltung sowie Verarbeitung bleiben versteckt (Abbildung 2.5). Die in

dieser Diplomarbeit verwendete Erweiterung von WS-BPEL 2.0 BPEL4Chor basiert auf dem *Abstract Process Profile for Observable Behavior* (vgl. Abschnitt 13.3 aus [OAS07]).

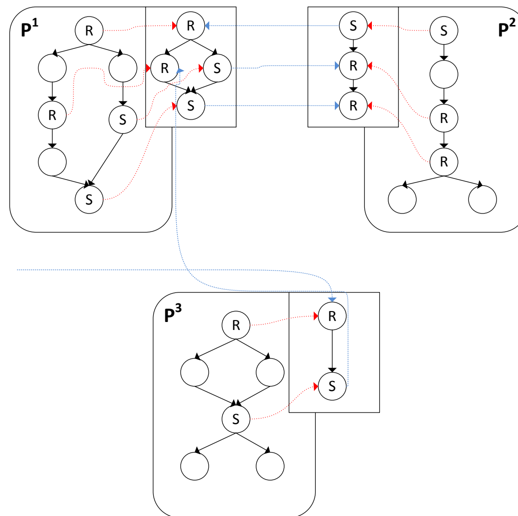


Abbildung 2.5 Kommunikationsmuster zwischen Prozesspartnern

- Eine externe Sicht auf den ausführbaren Prozess wird präsentiert, die genauen internen Details bleiben verborgen. So können technische Details ausgeblendet werden oder beispielsweise firmeninterne Vorgehensbausteine geheim bleiben (Abbildung 2.6).

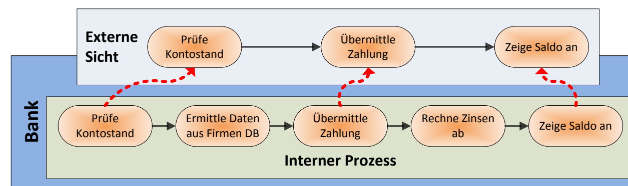


Abbildung 2.6 Externe Sicht eines Prozesses

- Ein Gerüst für eine allgemeine oder bewährte Vorgehensweise wird erstellt in der später die ausgelassenen Aktivitäten und Attribute für einen spezifischen Anwendungsfall hinzugefügt werden (Abbildung 2.7).

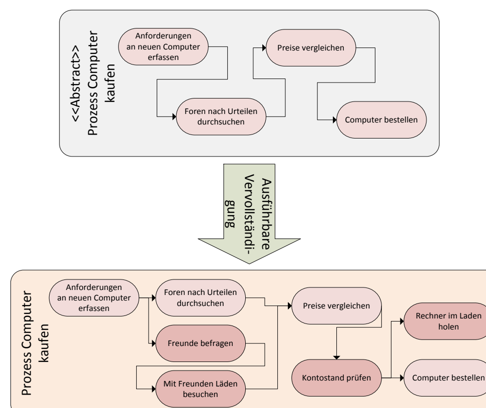


Abbildung 2.7 Grundgerüst Computerkauf mit ausführbarer Vervollständigung

Der für diese Diplomarbeit relevante Anwendungsbereich von abstrakten Prozessen ist die Projektion von Nachrichtenmustern zwischen den Kommunikationsteilnehmern. So entsteht eine externe Sicht auf den Kontroll- und Datenfluss, die wir im Folgenden als Choreographie bezeichnen.

2.2.1.2 Basis Aktivitäten von WS-BPEL 2.0

BPEL bietet eine Fülle von Basisaktivitäten zur Daten- und Kontrollflusssteuerung an. Der folgende Abschnitt beschreibt diese in kurzer Form. Für eine detaillierte Beschreibung sei hier auf die Spezifikation verwiesen [OAS07].

Jede Aktivität besitzt zwei optionale Standard Attribute: Ein `name` Attribute sowie `suppressJoinFailure`. Letzteres gibt an ob ein *joinFault* unterdrückt werden soll, wenn dieser Auftritt (siehe hierzu Abschnitt *11.6 Parallel and Control Dependencies Processing – Flow* der WS-BPEL 2.0 Spezifikation [OAS07]). Zusätzlich besitzt jede Aktivität zwei Container `<sources>` und `<targets>`, die die entsprechende Elemente `<source>` und `<target>` enthalten. Sie dienen der Kontrollflusssteuerung.

2.2.1.2.1 <invoke>

Mithilfe der `<invoke>`-Aktivität kann ein BPEL-Prozess einen anderen Prozess oder Web Service aufrufen. Der Aufruf kann sowohl asynchron (*One-Way*) erfolgen, als auch synchron (*Request-Response*). Im asynchronen Fall läuft der Kontrollfluss direkt nach dem Versenden weiter, im synchronen dagegen wartet der aufrufende Prozess zuerst auf die Antwort bevor er im Kontrollfluss fortschreitet. Eine `<invoke>`-Aktivität kann eigene *Compensation Handler* und *Fault Handler* definieren.

2.2.1.2.2 <receive> und <reply>

Mittels der `<receive>`-Aktivität wartet ein Prozess auf das Eintreffen einer Nachricht. Ist diese Aktivität die erste im Kontrollfluss, so muss das Attribute `createInstance="yes"` gesetzt werden, da hier der initiale Einstiegspunkt für den Prozesslebenszyklus ist. Mithilfe der `<reply>`-Akitvität kann der Prozess eine Antwortnachricht auf eine zuvor empfangene `<receive>`-Nachricht (*Request-Response*) senden. Die `<reply>`-Aktivität kann durch ein `messageExchange`-Attribut mit einer `<receive>`-Aktivität assoziiert werden.

2.2.1.2.3 <assign>

Die `<assign>`-Aktivität erlaubt die Zuweisung von Daten an Variablen mithilfe von XPath-Ausdrücken [W3C99a] bzw. die durch das `expressionLanguage`-Attribut des BPEL-Prozesses definierte Sprache.

2.2.1.2.4 <validate>

Die `<validate>`-Aktivität wird zur Validierung von Variablen bezüglich der mit ihnen assoziierten XML oder WSDL Datendefinitionen verwendet.

2.2.1.2.5 <throw>

Die `<throw>`-Aktivität wird zum Werfen einer Ausnahme innerhalb des BPEL-Prozesses verwendet.

2.2.1.2.6 <wait>

Die `<wait>`-Aktivität veranlasst einen BPEL-Prozess für eine definierte Zeitspanne oder bis zu einem definierten Zeitpunkt zu warten.

2.2.1.2.7 <empty>
Die `<empty>`-Aktivität macht effektiv nichts. Sie wird zur Synchronisation von nebenläufigen Aktivitäten verwendet oder wenn ein *Fault* gefangen und unterdrückt werden soll.

2.2.1.2.8 <extensionActivity>
BPEL erlaubt die Erweiterung der Spezifikation durch neue Aktivitäten. Die BPEL-Engine muss jedoch in der Lage sein diese zu verstehen, ansonsten können diese wie eine `<empty>`-Aktivität gehandhabt werden.

2.2.1.2.9 <exit>
Die `<exit>`-Aktivität beendet augenblicklich die Instanz eines Prozesses.

2.2.1.2.10 <rethrow>
Der `<rethrow>`-Aktivität ermöglicht das Weiterreichen einer Ausnahme, aus einem *Fault Handler* und nur aus einem solchen, an den ihn umgebenden Gültigkeitsbereich.

2.2.1.2.11 <compensate>
Die `<compensate>`-Aktivität veranlasst die Kompensation (siehe Abschnitt 2.2.1.4) aller erfolgreich ausgeführter Subscopes. Sie darf nur innerhalb eines `<catch>`, `<catchAll>`, `<compensation-Handler>` oder `<terminationHandler>` verwendet werden.

2.2.1.2.12 <compensateScope>
Die `<compensateScope>`-Aktivität veranlasst die Kompensation (siehe Abschnitt 2.2.1.4) eines spezifischen erfolgreich ausgeführten Subscopes. Sie darf nur innerhalb eines `<catch>`, `<catch-All>`, `<compensationHandler>` oder `<terminationHandler>` verwendet werden.

2.2.1.3 Strukturierte Aktivitäten von WS-BPEL 2.0

BPEL besitzt weiterhin strukturierte Aktivitäten, die sich aus den zuvor erläuterten Basisaktivitäten zusammensetzen und so eine sequentielle oder parallele Ausführung erlauben.

2.2.1.3.1 <sequence>
Die `<sequence>`-Aktivität führt die in ihr aufgeführten Aktivitäten sequentiell aus. Die Reihenfolge entspricht der Anordnung der Aktivitäten innerhalb des `<sequence>`-Elements.

2.2.1.3.2 <if>
Die `<if>`-Aktivität erlaubt die bedingte Ausführung von Aktivitäten. Die Aktivität besteht aus einer Liste von einer oder mehreren Verzweigungen, die durch `<if>` und optionale `<elseif>`-Elemente definiert sind, gefolgt von einem optionalen `<else>`-Zweig.

2.2.1.3.3 <while>
Die `<while>`-Aktivität erlaubt die wiederholte Ausführung der in ihr enthaltenen Aktivität. Die enthaltene Aktivität wird solange ausgeführt, wie die boolesche Bedingung (`<condition>`) zu Beginn jeder Iteration wahr ist.

2.2.1.3.4 <repeatUntil>
Ähnlich der `<while>`-Aktivität erlaubt die `<repeatUntil>`-Aktivität die wiederholte Ausführung der in ihr enthaltenen Aktivität. Die enthaltene Aktivität wird solange ausgeführt bis die angegebene

boolesche Bedingung (`<condition>`) wahr wird. Im Gegensatz zur `<while>`-Aktivität führt die `<repeatUntil>`-Aktivität die enthaltene Aktivität mindestens einmal aus.

2.2.1.3.5 <pick>

Die `<pick>`-Aktivität wartet auf das Eintreffen genau eines Ereignisses aus einer Menge von Ereignissen und führt anschließend eine mit diesem assoziierte Aktivität aus. Nachdem ein Ereignis eingetroffen ist, werden die restlichen verworfen. Ein solches Ereignis kann durch eine Nachricht (`<onMessage>`) oder einen zeitlichen *TimeOut* (`<onAlarm>`) repräsentiert werden. Jede `<pick>`-Aktivität muss mindestens ein `<onMessage>`-Ereignis enthalten.

2.2.1.3.6 <flow>

Die `<flow>`-Aktivität unterstützt Nebenläufigkeit und Synchronisation. Mithilfe der enthaltenen `<links>` können Abhängigkeiten zwischen den eingeschlossenen Aktivitäten festgelegt werden. Durch Angabe des `<source>`-Elements kann jede Aktivität der Ausgangspunkt eines Links werden. Durch das entsprechende `<target>`-Element kann jede Aktivität zum Endpunkt eines Links werden.

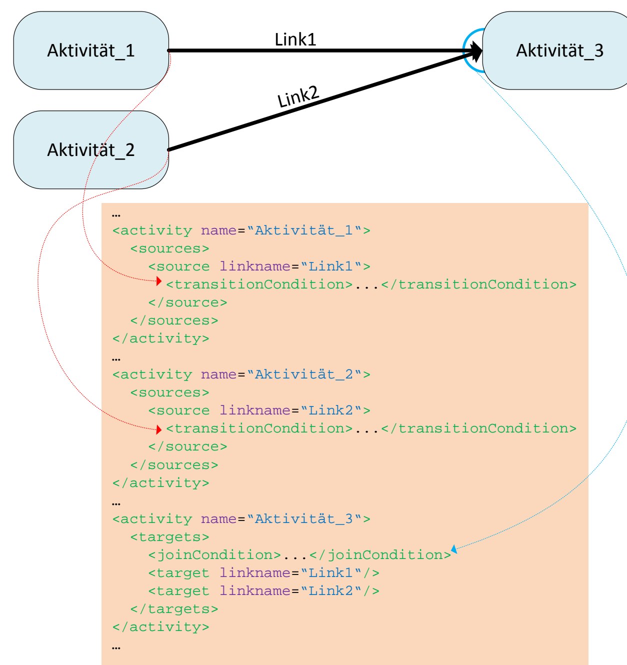


Abbildung 2.8 Links und ihre Semantik

Abbildung 2.8 zeigt den Zusammenhang zwischen `<links>`, `<targets>` und `<sources>`: Sobald Aktivität_1 beendet wurde, wird ihre optionale `<transitionCondition>` evaluiert. Ist keine angegeben so wird angenommen, dass sie zu `true` ausgewertet wird. Das Gleiche gilt für Aktivität_2 sowie Link2. Kommt die Ausführung zur Aktivität_3 so wird ihre optionale `<joinCondition>`, die für beide `<targets>` gilt ausgewertet. Ist diese nicht vorhanden, so ist die implizite `<joinCondition>` die Disjunktion von Link1 und Link2. Das optionale `suppressJoinFailure`-Attribut jeder Aktivität gibt an, ob im Falle einer negativen Auswertung der `<joinCondition>` ein `joinFailure`-Fault ausgelöst oder die *Dead-Path-Elimination* ausgeführt wird. Ist das Attribut nicht gesetzt, erbt die Aktivität dieses von ihrer umgebenden Aktivität (vgl. Abschnitt *11.6.3 Dead-Path-Elimination* der WS-BPEL 2.0 Spezifikation [OAS07]).

2.2.1.3.7 <forEach>

Die `<forEach>`-Aktivität führt den in ihr enthaltenen `<scope>` genau `N+1` mal aus, wobei `N` dem Wert des `<finalCounterValue>` minus dem Wert des `<startCounterValue>` entspricht. Durch

Angabe einer Abschlussbedingung (<completionCondition>) kann dieser Mechanismus schon vor Beendigung der Gesamtanzahl der Durchläufe der Schleife durchbrochen werden. Wird das parallel-Attribut auf „yes" gesetzt, so werden die Schleifendurchläufe parallel ansonsten sequentiell ausgeführt.

2.2.1.4 Scopes und Handler

Die <scope>-Aktivität bietet den in ihr enthaltenen Aktivitäten einen eigenen Gültigkeitsbereich mit der Möglichkeit der Definition eigener Variablen, PartnerLinks, messageExchange-Attributen sowie Handlern, die nur in diesem Kontext gültig sind. Gültigkeitsbereiche von Scopes können hierarchisch verschachtelt sein, wobei der „*root*"-Kontext der BPEL-Prozess selbst ist. Im Gegensatz zum <process>-Element können Scopes noch zusätzlich *Compensation Handler* (CH) sowie *Termination Handler* (TH) enthalten.

2.2.1.4.1 Compensation Handler (CH)

Wurde eine <scope>-Aktivität erfolgreich beendet, so wird ihr *Compensation Handler* installiert. Die im *CH* enthaltenen Aktivitäten werden ausgeführt, wenn es zu einer Kompensation (*Compensation*) in einem übergeordneten Gültigkeitsbereich kommt, beispielsweise durch einen transaktionsbedingten Rollback [KRL09].

2.2.1.4.2 Fault Handler (FH)

In jedem BPEL-Prozess und jeder <scope>-Aktivität können *Fault Handler* definiert werden. Mit ihrer Hilfe können durch <throw>-Aktivitäten geworfene Ausnahmen abgefangen und bearbeitet werden.

2.2.1.4.3 Termination Handler (TH)

<scope>-Aktivitäten können *Termination Handler* definieren, die im Falle einer erzwungenen Terminierung oder einer Ausnahme noch vor den *Fault Handlern* ausgelöst werden. Wird kein *TH* explizit definiert, so wird der *Default Termination Handler* installiert, der lediglich die Kompensation auslöst. Ein *TH* darf keine Ausnahmen werfen.

2.2.1.4.4 Event Handler (EH)

Jeder BPEL-Prozess und jede <scope>-Aktivität kann optional *Event Handler* definieren, die aus beliebig vielen <onEvent>- und <onAlarm>-Zweigen, jedoch aus mindestens einem von beiden, bestehen können. Die Aktivität in diesen Zweigen muss eine <scope>-Aktivität sein, die durch Eintreten eines bestimmten Ereignisses ausgeführt wird. <onEvent>-Zweige werden durch das Eintreffen einer bestimmten Nachricht, <onAlarm>-Zweige dagegen durch das Eintreffen eines *TimeOuts* ausgelöst. Die Ausführung eines *Event Handlers* findet nebenläufig zur umgebenden <scope>-Aktivität bzw. zum umgebenden BPEL-Prozess statt. *Event Handler* können auf die Daten der sie umgebenden Gültigkeitsbereiche zugreifen.

2.2.1.4.5 Isolierte Scopes

<scope>-Aktivitäten bieten zusätzlich durch das isolated-Attribut Kontrolle über den nebenläufigen Zugriff auf gemeinsame Ressourcen: Variablen, Partner Links sowie Kontrollfluss Links von <flow>-Aktivitäten. Der interessierte Leser findet hierzu im Abschnitt *12.8 Isolated Scopes* der WS-BPEL 2.0 [OAS07] Spezifikation eine detaillierte Erläuterung der Funktionsweise.

2.3 BPEL4Chor

Das folgende Kapitel basiert auf der Quelle [DKLW07]. WS-BPEL 2.0 eignet sich primär zur Orchestrierung aus Sicht eines einzelnen Prozessteilnehmers und weniger zur Darstellung einer Choreographie zwischen mehreren Geschäftsprozessen, die miteinander kommunizieren. Um eine globale Sicht auf alle Teilnehmer, die an einer solchen Interaktion teilnehmen zu ermöglichen, wurde BPEL4Chor entwickelt [DKLW07][DKLW09]. BPEL4Chor besteht aus drei verschiedenen Artefakten auf die im folgenden Abschnitt anhand eines kleinen Beispiels näher eingegangen wird.

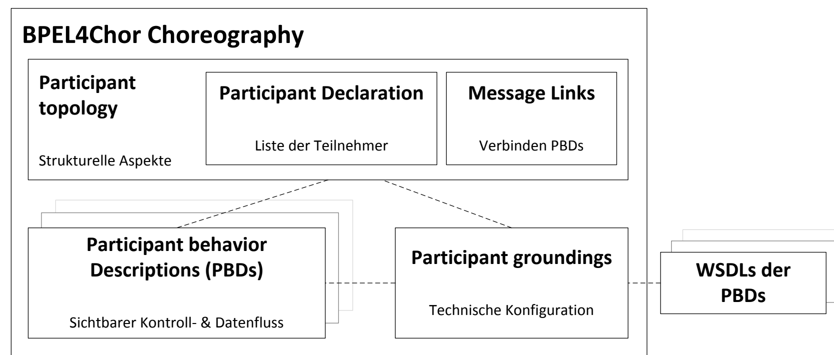


Abbildung 2.9 BPEL4Chor Artefakte (vgl. [DKLW07])

Abbildung 2.9 zeigt den Aufbau sowie die in einer BPEL4Chor Choreographie enthaltenen Artefakte:

1. **Participant Behavior Description (PBD)**:
   PBDs beschreiben den Kontrollfluss zwischen den Aktivitäten der teilnehmenden Prozesse einer Choreographie. Eine PBD kann derart abstrahiert sein, dass sie nur die nachrichtensendeden- sowie empfangenden Aktivitäten enthält, aber auch alle Aktivitäten zwischen diesen. Der in der vorliegenden Diplomarbeit beschriebene Konsolidierungsalgorithmus erhebt den Anspruch einer möglichst vollständigen Beschreibung der PBDs um ein möglichst optimales Ergebnis zu liefern. So kann z.B. bei Benutzung einzelner opaker Aktivitäten keine Datenflussanalyse durchgeführt werden, da nicht sicher ist, dass diese Aktivitäten später möglicherwiese durch `<assign>`-Aktivitäten ersetzt werden. WS-BPEL 2.0 definiert ein Profil für abstrakte Prozesse, das *Abstract Process Profile for Observable Behavior* [OAS07], das die Benutzung von Elementen, wie beispielsweise *partnerLinks*- oder *operation*-Attributen von nachrichtensendenden- und empfangenden Aktivitäten verbietet, um so eine Trennung von der technischen Konfiguration in den WSDL-Dateien zu ermöglichen. Dieses Profil wird mit einigen wenigen Änderungen für die PBDs verwendet: Ein zusätzliches Attribut *wsu:id* vom Typ *xsd:id* wird in jede Aktivität, die an einer Kommunikation teilnimmt hinzugefügt, um eine Identifikation der sendenden und empfangenden Aktivitäten in verschiedenen Interaktionsteilnehmern zu gewährleisten.

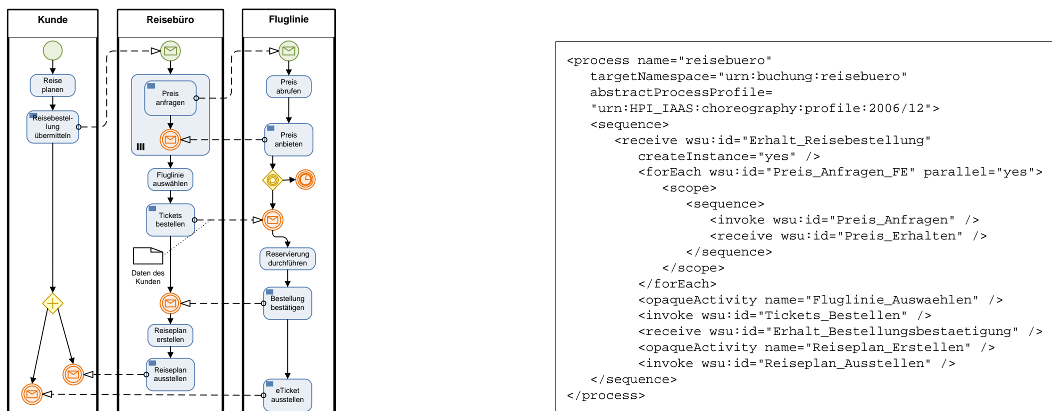
2. **Participant Topology**:
   Die *Participant Topology*, im Folgenden kurz *Topology* genannt, beschreibt die Kommunikation zwischen den Teilnehmern der Choreographie. Typen von Teilnehmern (*Participants*), Teilnehmerreferenzen (*Participant References*) sowie Nachrichtenlinks (*Message Links*) werden hier definiert. Jeder Teilnehmer ist mit einem Typ assoziiert sowie die Relationen zwischen verschiedenen Teilnehmern definiert. Die Nachrichtenlinks beschreiben schließlich die

Verbindung zwischen Kommunikationsaktivitäten in verschiedenen Teilnehmern. (Die Syntax einer Topology ist in [KOP11b] zu finden.)

3. **Participant Grounding**:
   Alle technischen Aspekte wurden aus den vorhergehenden Artefakten herausgenommen und werden nun in dem *Participant Grounding* definiert. Web Service spezifische *Port Types*, Operationen sowie XML Schema Typen sind hier spezifiziert. Die Verbindung mit der WSDL-Definition wird hier realisiert. (Die Syntax eines Grounding ist in [KOP11a] zu finden.)

2.3.1 Beispielchoreographie

Das folgende Beispiel ist [DKLW07] entnommen.




```
<process name="reisebuero"
   targetNamespace="urn:buchung:reisebuero"
   abstractProcessProfile=
   "urn:HPI_IAAS:choreography:profile:2006/12">
   <sequence>
      <receive wsu:id="Erhalt_Reisebestellung"
         createInstance="yes" />
         <forEach wsu:id="Preis_Anfragen_FE" parallel="yes">
            <scope>
               <sequence>
                  <invoke wsu:id="Preis_Anfragen" />
                  <receive wsu:id="Preis_Erhalten" />
               </sequence>
            </scope>
         </forEach>
         <opaqueActivity name="Fluglinie_Auswaehlen" />
         <invoke wsu:id="Tickets_Bestellen" />
         <receive wsu:id="Erhalt_Bestellungsbestaetigung" />
         <opaqueActivity name="Reiseplan_Erstellen" />
         <invoke wsu:id="Reiseplan_Ausstellen" />
   </sequence>
</process>
```

Abbildung 2.10 Choreographiebeispiel (vgl. [DKLW07])

Auflistung 2.1 PBD des Reisebüros (vgl. [DKLW07])

```
<topology name="buchungstopology"
   targetNamespace="urn:buchung"
   xmlns:reisebuero="urn:buchung:reisebuero">
   <participantTypes>
      <participantType name="Reisebuero"
         participantBehaviorDescription="reisebuero:reisebuero" />
      <participantType name="Kunde" ... />
      <participantType name="Fluglinie" ... />
   </participantTypes>
   <participants>
      <participant name="kunde" type="Kunde" selects="reisebuero" />
      <participant name="reisebuero" type="Reisebureo" selects="fluglinien" />
      <participantSet name="fluglinien" type="Fluglinie"
         forEach="reisebuero:Preis_Anfragen_FE">
         <participant name="aktuelleFluglinie"
            forEach="reisebuero:Preis_Anfragen_FE" />
         <participant name="gewaehlteFluglinie" />
      </participantSet>
   </participants>
   <messageLinks>
      <messageLink name="reiseBestellungUebermittelnLink"
         sender="kunde"
         sendActivity="Reisebestellung_Uebermitteln"
         receiver="reisebuero"
         receiveActivity="Erhalt_Reisebestellung"
         messageName="reiseBestellung" />
      <!-- ... -->
      <messageLink name="ticketsBestellenLink"
         sender="reisebuero"
         sendActivity="TicketsBestellen"
         receiver="gewaehlteFluglinie"
         receiveActivity="Erhalt_Bestellung"
         messageName="ticketBestellung"
         participantRefs="kunde" />
      <messageLink name="eTicketAusstellenLink"
         sender="gewaehlteFluglinie"
         sendActivity="eTicketAusstellen"
         receiver="kunde"
         receiveActivity="Erhalt_eTicket"
         messageName="eTicket" />
   </messageLinks>
</topology>
```

```
<grounding topology="top:buchungstopology"
   xmlns:top="urn:buchung" xmlns:...>
   <messageLinks>
      <messageLink name="reiseBestellungUebermittelnLink"
         portType="ag1:reiseBuero_pt"
         operation="getTripRequest" />
      <messageLink name="ticketsBestellenLink"
         portType="lhx:web_pt"
         operation="getOrder" />
      <!-- ... -->
   </messageLinks>
   <participantRefs>
      <participantRef name="kunde"
          WSDLproperty="msgs:kundeProp" />
   </participantRefs>
</grounding>
```

Auflistung 2.2 Participant Topology (vgl. [DKLW07])

Auflistung 2.3 Participant Grounding (vgl. [DKLW07])

Ein Kunde möchte einen Flug buchen und setzt sich hierzu mit einem Reisebüro in Verbindung. Das Reisebüro kontaktiert daraufhin verschiedene Fluglinien, um angebotene Preise für die vom Kunden gewünschten Flugdaten zu akquirieren. Nach eingegangenen Antworten der Fluglinien wählt das Reisebüro die für den Kunden günstigste Alternative und bucht mit dessen Daten (Email-Adresse des Kunden) einen Flug bei der Fluglinie mit den besten Konditionen. Die anderen Fluglinien beenden nach einer gewissen Zeit das Warten auf eine mögliche Bestellung. Nach Bestätigung der Buchung

durch die ausgewählte Fluglinie, sendet das Reisebüro dem Kunden einen Reiseplan und die Fluglinie ein eTicket an Selbigen.
Auflistung 2.1 zeigt die PBD des beteiligten Reisebüros. Alle technischen Details wurden entfernt, lediglich die kommunizierenden Aktivitäten werden in ihrem Kontrollfluss aufgezeigt. Auch der Vorgang des Auswählens der günstigsten Fluglinie sowie das Erzeugen des Reiseplans werden als opake Aktivitäten dargestellt.
Auflistung 2.2 zeigt die Participant Topology für das Reisebuchungsbeispiel. Im Mittelpunkt dieses Artefakts stehen die Message Links die miteinander kommunizierende Aktivitäten in verschiedenen Choreographie Teilnehmern miteinander verbinden. Somit können die strukturellen Aspekte einer Choreographie genau definiert werden. Auflistung 2.3 zeigt das Participant Grounding für das Beispiel aus [DKLW07]. In diesem Artefakt werden die technischen Verbindungen zur WSDL definiert, die dann die Choreographie Web-Service-spezifisch machen. Für den in dieser Diplomarbeit vorgestellten Konsolidierungsalgorithmus sind vor allem die Message Links aus der Topology von zentraler Bedeutung, da mit ihrer Hilfe die zu verbindenden Aktivitäten der zu konsolidierenden Teilnehmer definiert werden. Für diese gelten die folgenden Regeln (vgl. [DKLW07]):

I. Nur eine `<receive>`-Aktivität, ein `<onMessage>`-Zweig oder eine `<invoke>`-Aktivität sind für das `receiveActivity`-Attribut gültig. Falls eine `output`-Variable in der `<invoke>`-Aktivität definiert wird, muss diese `<invoke>`-Aktivität als `receiveActivity` in einem anderen Message Link auftauchen. Angenommen ein Message Link *l* enthält ein `<invoke>` *i* als `sendActivity` und ein `<receive>` *r* als `receiveActivity`. Falls *r* nicht mit einer `<reply>`-Aktivität *y* über ein `messageExchange`-Attribut assoziiert ist, darf das `<invoke>` *i* nicht als `receiveActivity` in einem anderen Message Link auftauchen.

II. `<reply>`- und `<invoke>`-Aktivitäten sind gültig als `sendActivity` in einem Message Link. Nehmen wir *i*, *r* und *y* wie in I. definiert. Der Message Link, der *y* als `sendActivity` enthält muss *i* als `receiveActivity` enthalten. Somit kann ein `<reply>` nur einem synchronen Aufruf eines `<invoke>` antworten und nicht weiteren `<invoke>`-oder `<receive>`-Aktivitäten oder einem `<onMessage>`-Zweig.

III. Jede `<invoke>`- und `<reply>`-Aktivität ist in genau einem Message Link die `sendActivity`. Ausgenommen hiervon sind Aktivitäten, die in eine Kommunikation mit Prozessen oder Web Services involviert sind, die nicht in der Choreographie definiert sind. Daraus folgt: Existieren mehrere `<receive>`-Aktivitäten für die gleiche `<invoke>`-oder `<reply>`-Aktivität (z.B. durch Verzweigung auf der Empfängerseite), so müssen diese alle denselben Bezeichner haben.

IV. Für jede `<receive>`-Aktivität und jeden `<onMessage>`-Zweig gibt es genau einen Message Link der diese Aktivität bzw. diesen Zweig als `receiveActivity` enthält. Im Fall von mehreren sendenden Aktivitäten für ein `<receive>` (oder `<onMessage>`-Zweig) ist eine Menge von Sendern im Message Link angegeben (`senders`-Attribut).

V. Falls das `senders`-Attribut im Message Link definiert ist, muss jeder der dort vorkommenden Sender vom gleichen Typ sein.

VI. Falls das `senders`-Attribut in einem Message Link *l* definiert ist und die `<receive>`-Aktivität *r* ist mit einer `<reply>`-Aktivität *p* über ein `messageExchange`-Attribut assoziiert, muss auch `bindSenderTo` im Link *l* definiert sein.

VII. Die Typen der Variablen von miteinander kommunizierenden Aktivitäten auf Sender- und Empfängerseite müssen identisch sein.

2.4 Allen-Kalkül

Das Allen-Kalkül, auch Allens Intervallalgebra genannt, ist eine Logik zur Repräsentation von zeitlichen Zusammenhängen. Sie definiert hierfür 13 Relationen mit denen es möglich ist alle möglichen Zusammenhänge zwischen zwei Intervallen zu beschreiben (vgl. Abbildung 2.11).

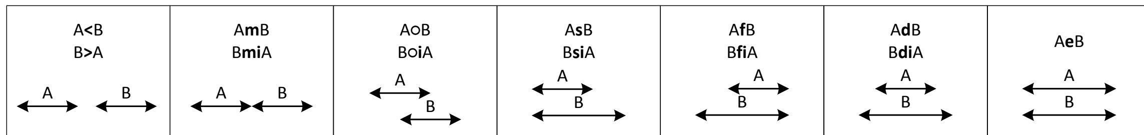
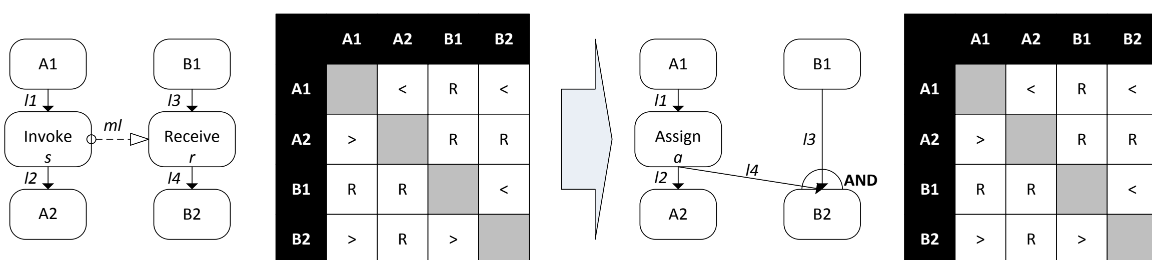


Abbildung 2.11 Die 13 Relationen des Allen-Kalküls

Tabelle 2.1 zeigt die Bedeutung der 13 Relationen aus Abbildung 2.11:

Intervallrelation	Bedeutung
A<B	A findet vor B statt
B>A	B findet nach A statt
A**m**B	A trifft B
B**mi**A	B wird von A getroffen (**i** steht für invers)
A○B	A überschneidet sich mit B
B○**i**A	B wird von A überschnitten
A**s**B	A fängt mit B an
B**si**A	invers zu A**s**B
A**f**B	A hört mit B auf
B**fi**A	invers zu A**f**B
A**d**B	B findet während B statt
B**di**A	invers zu A**d**B
A**e**B	A ist gleich B.

Tabelle 2.1 Die 13 Intervallrelationen und ihre Bedeutungen



	A1	A2	B1	B2
A1		<	R	<
A2	>		R	R
B1	R	R		<
B2	>	R	>	

	A1	A2	B1	B2
A1		<	R	<
A2	>		R	R
B1	R	R		<
B2	>	R	>	

Abbildung 2.12 Vergleich der Kontrollflussrelationen zweier Beispielfragmente dargestellt als Intervallrelationen

Abbildung 2.12 zeigt zwei Beispielfragmente sowie die zugehörigen Intervallrelationen: Die Reihen der Tabellen stehen für die linke Seite der Relation, die Spalten für die rechte Seite. Die grauen Felder stehen für die leere Menge Ø da eine Aktivität keine Intervallrelation zu sich selbst hat. *R* steht für die Menge aller Relationen, somit gilt *R={<, >, **m**, **mi**,* ○, ○***i**, **s**, **si**, **f**, **fi**, **d**, **di**, **e**}*. Aktivität *A1* wird vor Aktivität *A2* ausgeführt (*A1<A2*), jeweils im linken als auch im rechten Fragment. Das gleiche gilt für die Aktivitäten B1 sowie B2 (*B1<B2*). Wichtig an diesem Beispiel sind die Relationen zwischen *A1* und *B2* in beiden Fragmenten: Im linken wird zuerst *A1* ausgeführt und nachdem `<invoke>` *s* eine Nachricht an das empfangende `<receive>` *r* gesendet hat, wird *B2* ausgeführt. Diese Kontrollflussabhängigkeit wird im rechten Fragment durch eine `<assign>`-Aktivität *a* simuliert. Durch *a* wird *B2* erst ausgeführt, wenn *B1* <u>und</u> *a* ausgeführt wurden.

3 Konsolidierung von BPEL4Chor-Choreographien

In diesem Kapitel werden wir das Vorgehen beim Konsolidieren einer BPEL4Chor-Choreographie genauer erklären und einen Algorithmus hierfür präsentieren. Um zu zeigen, dass der so erzeugte neue Prozess den gleichen Kontrollfluss, wie die ursprüngliche Choreographie hat, werden wir zunächst ein Zustandsmodell einführen. Weidlich et. al. zeigen in ihrer Arbeit [WDW07] eine Möglichkeit der Kompatibilitätsanalyse zwischen BPEL 2.0 Prozessen basierend auf dem π-Kalkül. Wagner et. al. [WKL12] zeigen dagegen einen Ansatz zur Überprüfung der Konsolidierung von BPEL4Chor-Choreographien, basierend auf dem Allen-Kalkül [AL83]. Im folgenden Kapitel werden wir den letzteren Ansatz verwenden.

3.1 Zustandsmodell für WS-BPEL 2.0 Prozesse sowie Aktivitäten

Der folgende Abschnitt veranschaulicht die möglichen Zustände eines WS-BPEL 2.0 Prozesses sowie seiner involvierten Aktivitäten. Die hier vorgestellten Zustandsmodelle sind [KHK+11] entnommen. Die Modelle wurden auf der *Apache ODE* Engine [AODE11] implementiert. Da Zustandsmodelle nicht explizit durch die WS-BPEL 2.0 Spezifikation [OAS07] definiert werden, können die Zustände einzelner BPEL-Engine Implementierungen abweichen. Die vorgestellten Modelle werden in UML Zustandsdiagrammen [OMG10] dargestellt.
Das Zustandsmodell aus Abschnitt 3 *Process State Model* [KHK+11] wird hier nicht verwendet, da die technischen Fähigkeiten einer BPEL-Engine zum Bereitstellen eines BPEL Prozesses oder seine Ersetzung durch einen neuen für diese Arbeit nicht relevant sind.

3.1.1 Prozess Instanz Zustandsmodell

Abbildung 3.1 zeigt das Prozess Instanz Zustandsmodell, Tabelle 3.1 beschreibt die einzelnen Zustände und ihre Übergangsbedingungen:

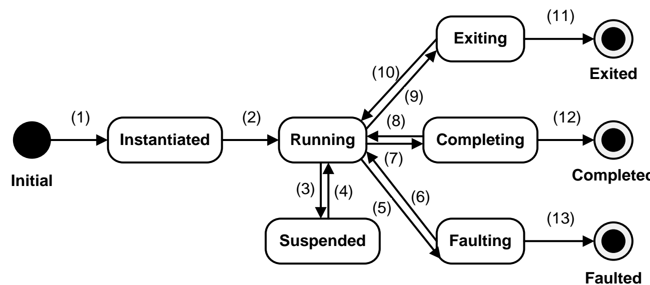


Abbildung 3.1 Prozess Instanz Zustandsmodell

Zustand	Bedeutung	Übergang	Bedeutung
Initial	Der Prozess wurde bereitgestellt und steht zur Instanziierung bereit.	(1)	Instanziierende Nachricht erfolgreich erhalten.

Instantiated	Der Prozess wurde instanziiert.	(2)	Ausführung der Prozessinstanz hat begonnen.
Running	Die Prozessinstanz wird ausgeführt.	(3)	Prozessinstanz wird in den Ruhemodus versetzt.
		(5)	Auftreten eines nichtbehandelten Fault während der Prozessausführung.
		(7)	Prozessinstanz wird erfolgreich abgeschlossen.
		(9)	Prozessinstanz wird beendet. (Hier durch `<exit>`-Aktivität)
Exiting	Die Prozessinstanz wird beendet. (Hier durch `<exit>`-Aktivität)	(10)	Prozessinstanz wird zur Wiederausführung angeregt.
		(11)	Prozessinstanz wird in Endzustand versetzt.
Suspended	Die Prozessinstanz wurde in den Ruhemodus versetzt. (Achtung: Nicht in der WS-BPEL 2.0 Spezifikation [OAS07] definiert.)	(4)	Ausführung der Prozessinstanz wird wieder aufgenommen.
Completing	Prozessinstanz wird erfolgreich abgeschlossen.	(8)	Prozessinstanz wird zur Wiederausführung angeregt.
		(12)	Prozessinstanz wird in Endzustand versetzt.
Faulting	Ein Fault hat die Grenze der Prozessinstanz erreicht. („root“-Kontext)	(6)	Prozessinstanz wird zur Wiederausführung angeregt.
		(13)	Prozessinstanz wird in Endzustand versetzt.
Exited	Prozessinstanz wurde erfolgreich beendet. (Hier durch `<exit>`-Aktivität)		
Completed	Prozessinstanz wurde erfolgreich beendet.		
Faulted	Prozessinstanz wurde aufgrund eines Fault beendet.		

Tabelle 3.1 Zustände und Übergänge des Prozess Instanz Modells

3.1.2 Aktivitäts-Zustandsmodell

Abbildung 3.2 zeigt das Aktivitäts-Zustandsmodell, Tabelle 3.2 beschreibt die einzelnen Zustände und ihre Übergangsbedingungen:

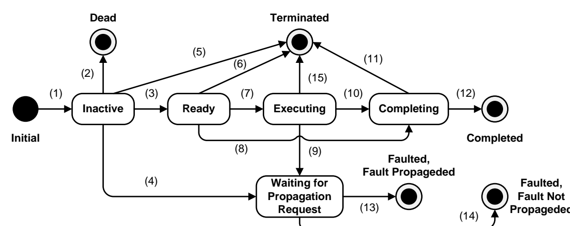


Abbildung 3.2 Aktivitäts-Zustandsmodell

Zustand	Bedeutung	Übergang	Bedeutung
Initial	Der initiale Zustand der Aktivität. Der Übergang von von **Initial** zu **Inactive** ist abhängig von der Implementierung der BPEL-Engine (vgl. [KHK+11]).	(1)	Die Aktivität wird vorbereitet.
Inactive	Die Aktivität wird noch nicht ausgeführt, sondern für die Ausführung vorbereitet.	(2)	Dead-Path-Elimination wird angewendet.
		(3)	Aktivität wird in den **Ready**-Zustand versetzt. Dies tritt ein sobald die Aktivität inaktiv ist, alle eingehenden Links ausgewertet wurden und die `joinCondition=true` ist.
		(4)	`joinCondition=false` und `suppressJoin-Failure=no`.
		(5)	Prozess wird beendet (hier durch <exit>-Aktivität) oder umschließende Vateraktivität hat Fault geworfen.
Ready	Die Aktivität steht zur Ausführung bereit.	(6)	Prozess wird beendet (hier durch <exit>-Aktivität) oder umschließende Vateraktivität hat Fault geworfen.
		(7)	Ausführung der Aktivität wird begonnen.
		(8)	Aktivität wird übersprungen.
Executing	Die Aktivität wird ausgeführt.	(9)	`joinCondition=false` und `suppressJoin-Failure=no`. Aktivität wirft einen Fault.
		(10)	Ausführung wird beendet.
		(15)	Prozess wird beendet (hier durch <exit>-Aktivität) oder umschließende Vateraktivität hat Fault geworfen.
Waiting for Propagation Request	Die Aktivität wartet auf weitere Anweisungen, ob der Fault an den umschließenden `Scope` weitergereicht oder unterdrückt (`suppress`) werden soll. Das Weiterreichen ist das Standardverhalten von BPEL.	(13)	Den Fault an umschließenden `Scope` weiterreichen.
		(14)	Den Fault wird verworfen und nicht an umschließenden `Scope` weitergereicht.
Completing	Ausführung der Aktivität wird erfolgreich abgeschlossen.	(11)	Prozess wird beendet (hier durch <exit>-Aktivität) oder umschließende Vateraktivität hat Fault geworfen.
		(12)	Ausführung der Aktivität vollständig abschließen und in Endzustand versetzen.
Dead	Die Aktivität wurde durch Dead-Path-Elemination in den Endzustand **Dead** versetzt.		
Terminated	Die Aktivität wurde terminiert.		

Completed	Die Ausführung der Aktivität wurde erfolgreich beendet. In diesem Zustand werden die ausgehenden Links der Aktivität ausgewertet.		
Faulted, Fault Propageded	Die Aktivität hat einen Fault geworfen und diesen an umschließenden `Scope` weitergereicht.		
Faulted, Fault Not Propageded	Die Aktivität hat einen Fault geworfen, diesen jedoch nicht an den umschließenden `Scope` weitergereicht.		

Tabelle 3.2 Zustände und Übergänge des Aktivitäts-Zustandsmodells

3.1.3 <scope>-Aktivitäts-Zustandsmodell

Abbildung 3.3 zeigt das `<scope>`-Aktivitäts-Zustandsmodell, Tabelle 3.3 beschreibt die einzelnen Zustände und ihre Übergangsbedingungen. Das `<scope>`-Aktivitäts-Zustandsmodell ist eine Erweiterung des Aktivitäts-Zustandsmodells.

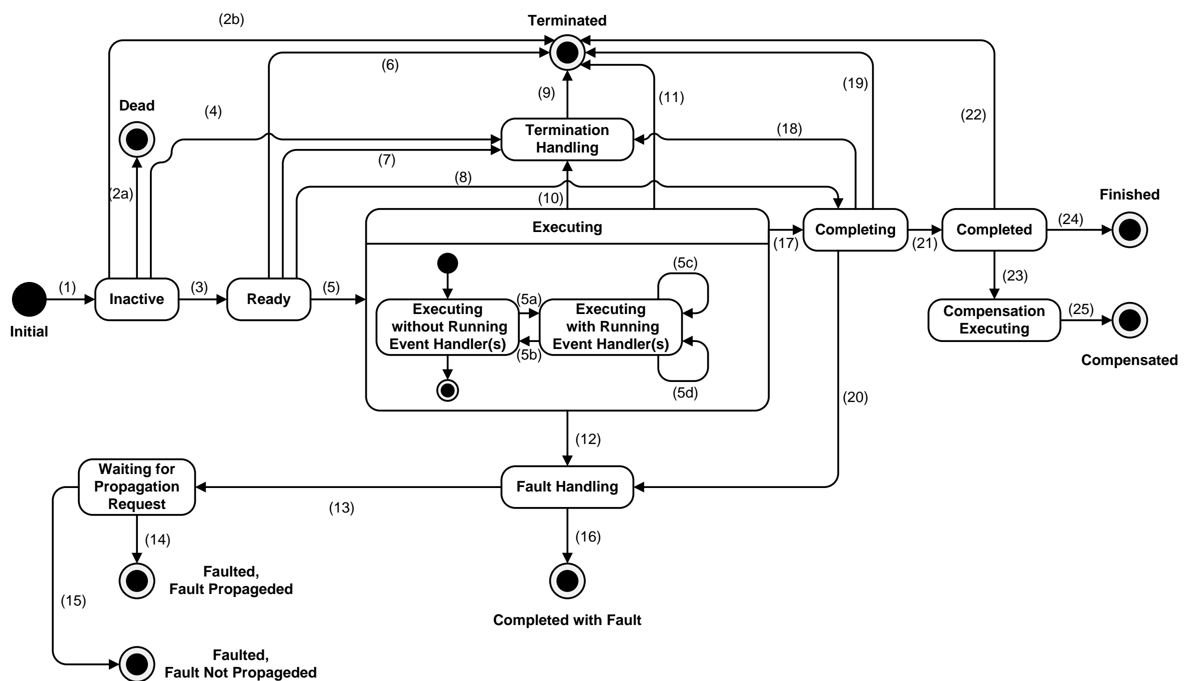


Abbildung 3.3 `<scope>`-Aktivitäts-Zustandsmodell

Zustand	Bedeutung	Übergang	Bedeutung
Initial	Der initiale Zustand der Aktivität. Der Übergang von von **Initial** zu **Inactive** ist abhängig von der Implementierung der BPEL-Engine (vgl. [KHK+11]).	(1)	Die Aktivität wird vorbereitet.

<table>
<tr><td colspan="2" rowspan="5">Inactive</td><td rowspan="5">Die Aktivität wird noch nicht ausgeführt, sondern für die Ausführung vorbereitet.</td><td>(2a)</td><td>Dead-Path-Elimination wird angewendet.</td></tr>
<tr><td>(2b)</td><td>Prozess wird beendet (hier durch <code><exit></code>-Aktivität).</td></tr>
<tr><td>(3)</td><td>Aktivität wird in den Ready-Zustand versetzt. Dies tritt ein sobald die Aktivität inaktiv ist, alle eingehenden Links ausgewertet wurden und die <code>joinCondition=true</code> ist.</td></tr>
<tr><td>(4)</td><td>Prozess wird beendet (hier durch <code><exit></code>-Aktivität) oder umschließende Vateraktivität hat Fault geworfen. Kontrolle wird an <code><terminationHandler></code> weitergereicht.</td></tr>
<tr></tr>
<tr><td colspan="2" rowspan="4">Ready</td><td rowspan="4">Die Aktivität steht zur Ausführung bereit.</td><td>(5)</td><td>Ausführung der Aktivität wird begonnen.</td></tr>
<tr><td>(6)</td><td>Prozess wird beendet (hier durch <code><exit></code>-Aktivität) oder umschließende Vateraktivität hat Fault geworfen.</td></tr>
<tr><td>(7)</td><td>Prozess wird beendet (hier durch <code><exit></code>-Aktivität) oder umschließende Vateraktivität hat Fault geworfen. Kontrolle wird an <code><terminationHandler></code> weitergereicht.</td></tr>
<tr><td>(8)</td><td>Aktivität wird übersprungen.</td></tr>
<tr><td colspan="2">Termination Handling</td><td>Die <code><scope></code>-Aktivität führt ihren <code><terminationHandler></code> aus.</td><td>(9)</td><td>Aktivität wird terminiert.</td></tr>
<tr><td rowspan="4">Executing</td><td>Executing without Running Event Handler(s)</td><td>Die <code><scope></code>-Aktivität wird ausgeführt. Es sind keine Event Handler installiert.</td><td>(5a)</td><td>Event Handler wird aufgrund eingehender Nachricht installiert.</td></tr>
<tr><td rowspan="3">Executing with Running Event Handler(s)</td><td rowspan="3">Ein Event Handler wurde durch eingehende Nachricht installiert und wird parallel zur <code><scope></code>-Aktivität ausgeführt.</td><td>(5b)</td><td>Event Handler hat seine Ausführung beendet.</td></tr>
<tr><td>(5c)</td><td>Ein weiterer Event Handler wird aufgrund eingehender Nachricht installiert.</td></tr>
<tr><td>(5d)</td><td>Event Handler hat seine Ausführung beendet. Weitere Event Handler werden noch ausgeführt.</td></tr>
<tr><td colspan="2" rowspan="3">Executing</td><td rowspan="3">Die <code><scope></code>-Aktivität wird ausgeführt.</td><td>(10)</td><td>Fault wurde während der Ausführung einer Kindaktivität geworfen.</td></tr>
<tr><td>(11)</td><td>Prozess wird beendet (hier durch <code><exit></code>-Aktivität).</td></tr>
<tr><td>(12)</td><td>Fault wurde während der Ausführung</td></tr>
</table>

			einer Kindaktivität geworfen.
		(17)	Ausführung wird beendet.
Completing	Ausführung der Aktivität wird erfolgreich abgeschlossen.	(18)	Fault wurde während der Ausführung einer Kindaktivität geworfen
		(19)	Prozess wird beendet (hier durch `<exit>`-Aktivität).
		(20)	Fault wurde während der Ausführung einer Kindaktivität geworfen.
		(21)	Ausführung der Aktivität vollständig abschließen.
Completed	Die Ausführung der Aktivität wurde erfolgreich beendet. In diesem Zustand werden die ausgehenden Links der Aktivität ausgewertet.	(22)	Prozess wird beendet (hier durch `<exit>`-Aktivität).
		(23)	`<compensationHandler>` wird aufgerufen (z.B. durch `<compensateScope>`).
		(24)	CH wurde nicht aufgerufen und Prozessinstanz kommt in den Endzustand.
Compensation Executing	Die `<scope>`-Aktivität wird kompensiert.	(25)	Kompensation wird beendet.
Fault Handling	Die `<scope>`-Aktivität verarbeitet einen aufgetretenen Fault.	(13)	Fault wurde nicht verarbeitet oder weitergereicht.
		(16)	Fault wurde verarbeitet und nicht weitergereicht.
Waiting for Propagation Request	Die Aktivität wartet auf weitere Anweisungen, ob der Fault an den umschließenden `Scope` weitergereicht oder unterdrückt (`suppress`) werden soll. Das Weiterreichen ist das Standardverhalten von BPEL.	(14)	Den Fault an den umschließenden `Scope` weiterreichen.
		(15)	Den Fault wird verworfen und nicht an umschließenden `Scope` weitergereicht.
Dead	Die Aktivität wurde durch Dead-Path-Elemination in den Endzustand **Dead** versetzt.		
Terminated	Die Aktivität wurde terminiert.		
Finished	Die `<scope>`-Aktivität kann nicht mehr kompensiert werden. Prozessinstanz hat Endzustand erreicht.		
Compensated	Der `Scope` wurde erfolgreich kompensiert.		
Completed with Fault	Ein Fault wurde in einem `<faultHandler>` bearbeitet und nicht weitergereicht.		
Faulted, Fault Propageded	Die Aktivität hat einen Fault geworfen und diesen an umschließenden `Scope` weiter-		

	gereicht.		
Faulted, Fault Not Propageded	Die Aktivität hat einen Fault geworfen, diesen jedoch nicht an den umschließenden `Scope` weitergereicht.		

Tabelle 3.3 Zustände und Übergänge des `<scope>`-Aktivitäts-Zustandsmodells

3.1.4 <invoke>-Aktivitäts-Zustandsmodell

Abbildung 3.4 zeigt das `<invoke>`-Aktivitäts-Zustandsmodell. Die `<invoke>`-Aktivität kann explizite Fault Handler sowie Compensation Handler definieren. Eine solche `<invoke>`-Aktivität ist semantisch äquivalent zu einer `<scope>`-Aktivität, die eine `<invoke>`-Aktivität enthält [OAS07].

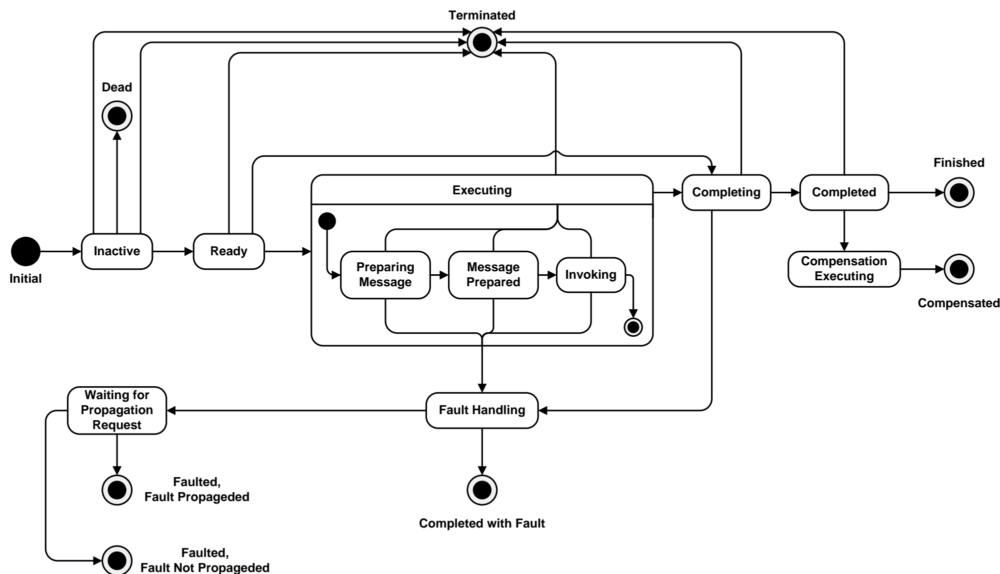


Abbildung 3.4 `<invoke>`-Aktivitäts-Zustandsmodell

Die in Abbildung 3.4 dargestellten drei neuen Unterzustände innerhalb des **Executing**-Zustands haben folgende Bedeutungen:

- **Preparing Message**:
  Die zu versendende Nachricht wird vorbereitet.

- **Message Prepared**:
  Die zu versendende Nachricht steht zum Versandt bereit.

- **Invoking**:
  Der externe Service wird aufgerufen. Im synchronen Fall verbleibt die `<invoke>`-Aktivität in diesem Zustand bis die `reply`-Nachricht erhalten wurde.

3.1.5 Schleifen-Zustandsmodell

Abbildung 3.5 zeigt das Schleifen-Zustands-Modell für die sequenzielle `<forEach>`-, die `<repeat-Until>`- sowie die `<while>`-Aktivität:

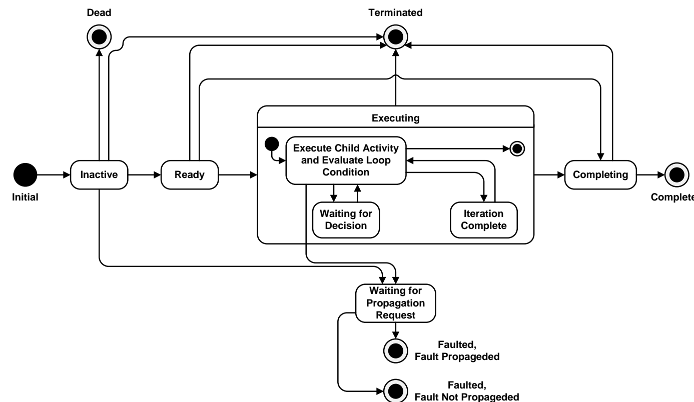


Abbildung 3.5 Schleifen-Zustandsmodell

Die in Abbildung 3.5 dargestellten drei Unterzuständen des **Executing**-Zustands haben folgende Bedeutungen:

- **Execute Child Activity and Evaluate Loop**:
  Die Aktivität innerhalb der Schleife wird ausgeführt.

- **Waiting for Decision**:
  Die Schleifen-Aktivität wartet auf das Ergebnis der Auswertung der Abbruchbedingung.

- **Iteration Complete**:
  Die Iteration wurde beendet.

3.1.6 Link-Zustandsmodell

Abbildung 3.6 zeigt das Link-Zustandsmodell, Tabelle 3.4 beschreibt die einzelnen Zustände und ihre Übergänge:

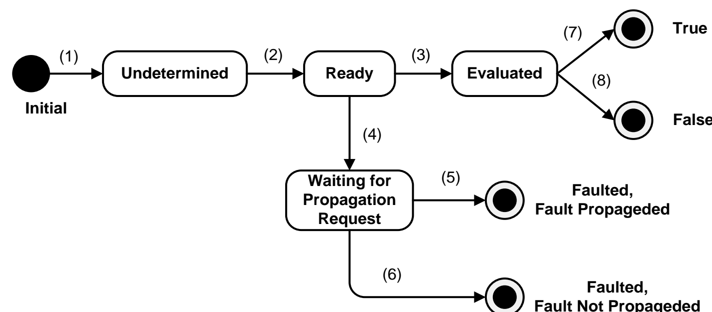


Abbildung 3.6 Link-Zustandsmodell

Zustand	Bedeutung	Übergang	Bedeutung
Initial	Der Link wurde noch nicht instanziiert.	(1)	Der Link wird instanziiert.
Undetermined	Der Status des Links wurde noch nicht ausgewertet.	(2)	Die Auswertung des Links wird begonnen.
Ready	Der Link steht zur Auswertung bereit.	(3)	`<transitionCondition>` des Links wird ausgewertet.
		(4)	Die Auswertung des Links hat zu einem Fault geführt.
Evaluated	Der Link wurde ausgewertet.	(7)	Der Linkstatus wird auf `true` gesetzt.
		(8)	Der Linkstatus wird auf `false` gesetzt.
Waiting for Propagation Request	Ein Fehler ist während der Auswertung aufgetreten.	(5)	Den Fault an den umschließenden `Scope` weiterreichen.
		(6)	Den Fault wird verworfen und nicht an umschließenden `Scope` weitergereicht.
Faulted, Fault Propageded	Die Auswertung des Links hat einen Fault geworfen und diesen an den umschließenden `Scope` weitergereicht.		
Faulted, Fault Not Propageded	Die Auswertung des Links hat einen Fault geworfen und diesen nicht an den umschließenden `Scope` weitergereicht.		
True	Der Linkstatus ist `true`.		
False	Der Linkstatus ist `false`.		

Tabelle 3.4 Zustände und Übergänge des Link-Zustandsmodells

3.2 Formales Vorgehen bei der choreographiebasierten Konsolidierung von BPEL-Prozessen

Der folgende Abschnitt beschreibt das Vorgehen beim Konsolidieren der BPEL4Chor-Choreographie.

Definition 3.2.1: *PBD* ist die Menge der in einer BPEL4Chor-Choreographie enthaltenen PBDs (Participant Behavior Descriptions). Jede PBD ist durch einen QName eindeutig bezeichnet.

Definition 3.2.2: *PT* ist die Menge der in einer Topology enthaltenen Participant Types. Jeder Participant Type ist durch einen NCName eindeutig bezeichnet. Durch das Attribut `participantBehaviorDescription`, das einen QName darstellt, wird die Verbindung zur zugehörigen PBD hergestellt.

Definition 3.2.3: *P* ist die Menge der in einer Topology enthaltenen Participants sowie Participant Sets. Jeder Participant sowie jedes Participant Set ist durch einen NCName eindeutig bezeichnet.

Durch das Attribut `type`, das einen NCName darstellt, wird die Verbindung zum zugehörigen Participant Type hergestellt.

Definition 3.2.4: *ML* ist die Menge der in einer Topology enthaltenen Message Links. Jeder Message Link ist durch einen NCName eindeutig bezeichnet.

Definition 3.2.5: Als *choreographie-intern kommunizierende* Aktivitäten bezeichnen wir alle sendenden und empfangenden Paare von Aktivitäten, die ihre Kommunikation auf die Choreographie beschränken.

Die Menge *PBD* erhalten wir, indem wir die gegebene Choreographie entpacken und die dort enthaltenen PBDs als abstrakte BPEL-Prozesse einlesen. Die Mengen *PT, P* sowie *ML* erhalten wir durch Einlesen der Topology-Datei aus der Choreographie.

3.2.1 Anlegen des konsolidierten BPEL-Prozesses

Bevor wir die eigentliche Konsolidierung durchführen, überführen wir zunächst die PBDs der Choreographie in einen neuen ausführbaren BPEL-Prozess. Hierzu wird ein neuer leerer BPEL-Prozess angelegt in den anschließend eine `<flow>`-Aktivität eingefügt wird. In diese `<flow>`-Aktivität werden nun alle PBDs als neue `<scope>`-Aktivitäten eingefügt, inklusive ihrer Variablen, MessageExchanges, CorrelationSets (Korrelationsmengen), Event Handler sowie Fault Handler. Wir verwenden hier eine `<flow>`-Aktivität um die Parallelität der ursprünglichen Choreographie in den neuen `<scope>`-Aktivitäten beizubehalten. Abbildung 3.61 zeigt dies anhand einiger schematischer Beispielfragmente: Die Choreographie enthält die drei PBDs *PBD1*, *PBD2* sowie *PBD3*, die samt ihrer enthaltenen Daten- sowie Kontrollflusselemente in den neuen Prozess *ProcessMerged* kopiert werden. Diese kommunizieren über die drei Message Links *ml1*, *ml2* sowie *ml3* miteinander. Auflistung 3.0 zeigt die algorithmischen Schritte beim Kopieren der PBDs in den neuen ausführbaren Prozess *ProcessMerged*.

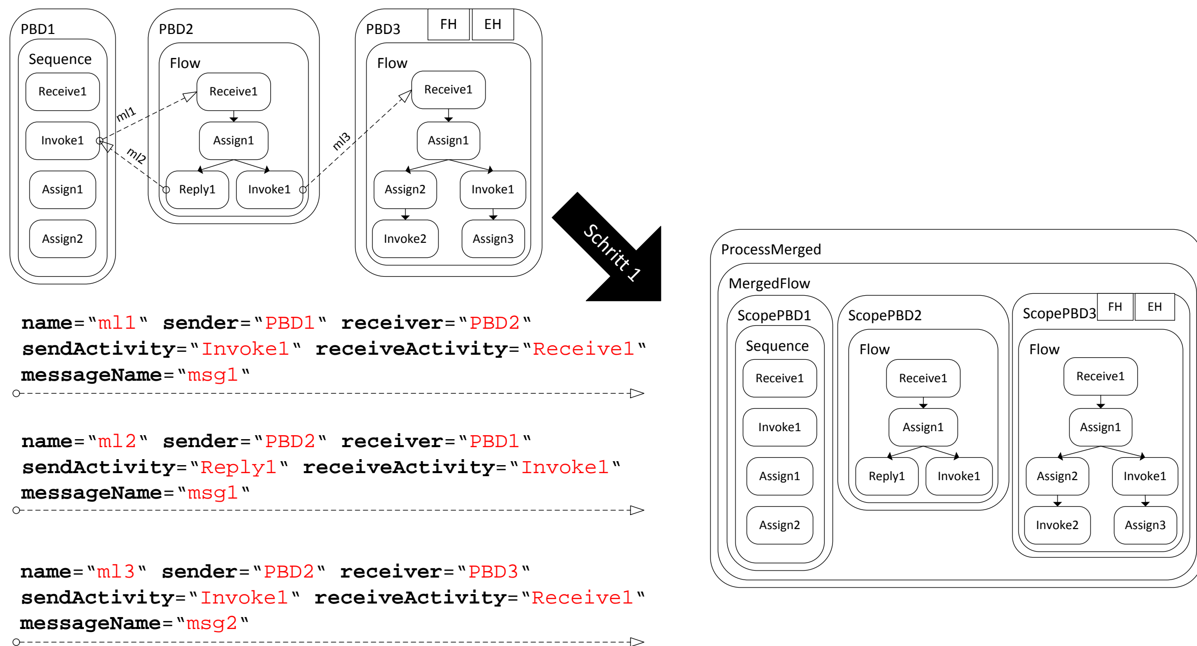


Abbildung 3.61 Anlegen des konsolidierten BPEL-Prozesses aus PBDs der ursprünglichen Choreographie (nicht dargestellt sind die Datenelemente Variablen, MessageExchanges sowie CorrelationSets)

Tabelle 3.4b erklärt die in Auflistung 3.0 verwendeten Funktionen:

Notation	Definition
createNewBPELProcess(name)	Funktion die einen neuen leeren BPEL Prozess mit dem Namen *name* zurückliefert
createNewActivity(actType, actName)	Funktion die eine neue Aktivität vom Typ *actType* und dem Namen *actName* zurückliefert
copyMessageExchanges(act1, act2)	Funktion die alle `MessageExchanges` aus Aktivität *act1* nach *act2* kopiert (*act1* sowie *act2* können vom Typ `<scope>` oder `<process>` sein)
copyCorrelationSets(act1, act2)	Funktion die alle `CorrelationSets` aus Aktivität *act1* nach *act2* kopiert (*act1* sowie *act2* können vom Typ `<scope>` oder `<process>` sein)
copyVariables(act1, act2)	Funktion die alle `Variables` aus Aktivität *act1* nach *act2* kopiert (*act1* sowie *act2* können vom Typ `<scope>` oder `<process>` sein)
copyFaultHandlers(act1, act2)	Funktion die alle FHs aus Aktivität *act1* nach *act2* kopiert (*act1* sowie *act2* können vom Typ `<scope>` oder `<process>` sein)
copyEventHandlers(act1, act2)	Funktion die alle EHs aus Aktivität *act1* nach *act2* kopiert (*act1* sowie *act2* können vom Typ `<scope>` oder `<process>` sein)
copyActivity(act1, act2)	Funktion die Aktivität *act1* sowie alle Subaktivitäten nach *act2* kopiert
createNPCatchAllFH(act)	Funktion die neuen `<catchAll>`-Fault Handler in Aktivität *act* anlegt, der nur eine `<compensate>`-Aktivität enthält (*NP* steht für „*Non-Propagating*“ vgl. Abschnitt 3.2.1.1)
hasCatchAllFH(act)	Funktion die überprüft ob Aktivität *act* einen `<catchAll>`-Fault Handler definiert

Tabelle 3.4b Notation für Funktionen und Mengen des Konsolidierungs-Algorithmus

```
(1)  MergedProcess = CreateNewBPELProcess(„ProcessMerged“)
       // Erzeuge neuen leeren BPEL Prozess
(2)  MergedProcess.setActivity(CreateNewActivity(<flow>, „MergedFlow“))
       // Erzeuge neue <flow>-Aktivität mit dem Namen „MergedFlow“

(3)  foreach (pbd in PBD) do
(4)    newScope = CreateNewActivity(<scope>, „Scope“+pbd.name)   // Lege neue <scope>-Aktivität an
(5)    copyMessageExchanges(pbd, newScope)   // Kopiere MessageExchanges aus pbd in neue
                                             //  <scope>-Aktivität
(6)    copyCorrelationSets(pbd, newScope)   // Kopiere CorrelationSets aus pbd in neue <scope>-Aktivität
(7)    copyVariables(pbd, newScope)          // Kopiere Variablen aus pbd in neue <scope>-Aktivität
(8)    copyFaultHandlers(pbd, newScope)     // Kopiere FaultHandler aus pbd in neue <scope>-Aktivität
(9)    if ( not hasCatchAllFaultHandler(pbd))  // Falls pbd keine <catchAll>-Fault Handler hat
(10)     createNPCatchAllFH(newScope)       // Lege neuen „Non-Propagating“-<catchAll>-FH an
(11)   fi
(12)   copyEventHandler(pbd, newScope)      // Kopiere EventHandler aus pbd in neue <scope>-Aktivität
(13)   copyActivity(pbd, newScope)          // Kopiere Aktivität (und alle Subaktivitäten) aus pbd in neue
                                            // <scope>-Aktivität
(14)   MergedProcess.getActivity().add(newScope)  // Füge neue <scope>-Aktivität in <flow>-Aktivität ein
(15) od
```

Auflistung 3.0 Kopieren der PBDs in neue `<scope>`-Aktivitäten des neuen BPEL-Prozesses

Da es sich bei den PBDs um abstrakte BPEL-Prozesse handelt und technische Eigenschaften, wie beispielsweise die `PartnerLinks` nicht in den kommunizierenden Aktivitäten enthalten sind, müssen diese nach der Generierung des Kontroll- sowie Datenflusses in den neuen ausführbaren BPEL-Prozess aus der Grounding-Datei übernommen werden (vgl. Abschnitt 3.4). Doch zunächst folgt in den nächsten beiden Abschnitten die Beschreibung der Generierung des Daten- sowie Kontrollflusses aus den Message Links.

3.2.1.1 Übernehmen der Fault Handler in konsolidierten Prozess

Beim Übertragen der ursprünglichen PBDs in den neuen konsolidierten Prozess in Form neuer `<scope>`-Aktivitäten müssen wir darauf achten, dass mögliche nicht gefangene Faults aus den `<scope>`-Aktivitäten nicht in den Prozessscope durchdringen, da diese den Ablauf des gesamten Prozesses beinflussen können (Zeile 9-11 Auflistung 3.0). Diese hätte eine veränderte Ausführungssemantik des konsolidierten Prozesses um Vergleich zur ursprünglichen Choreographie zur Folge.

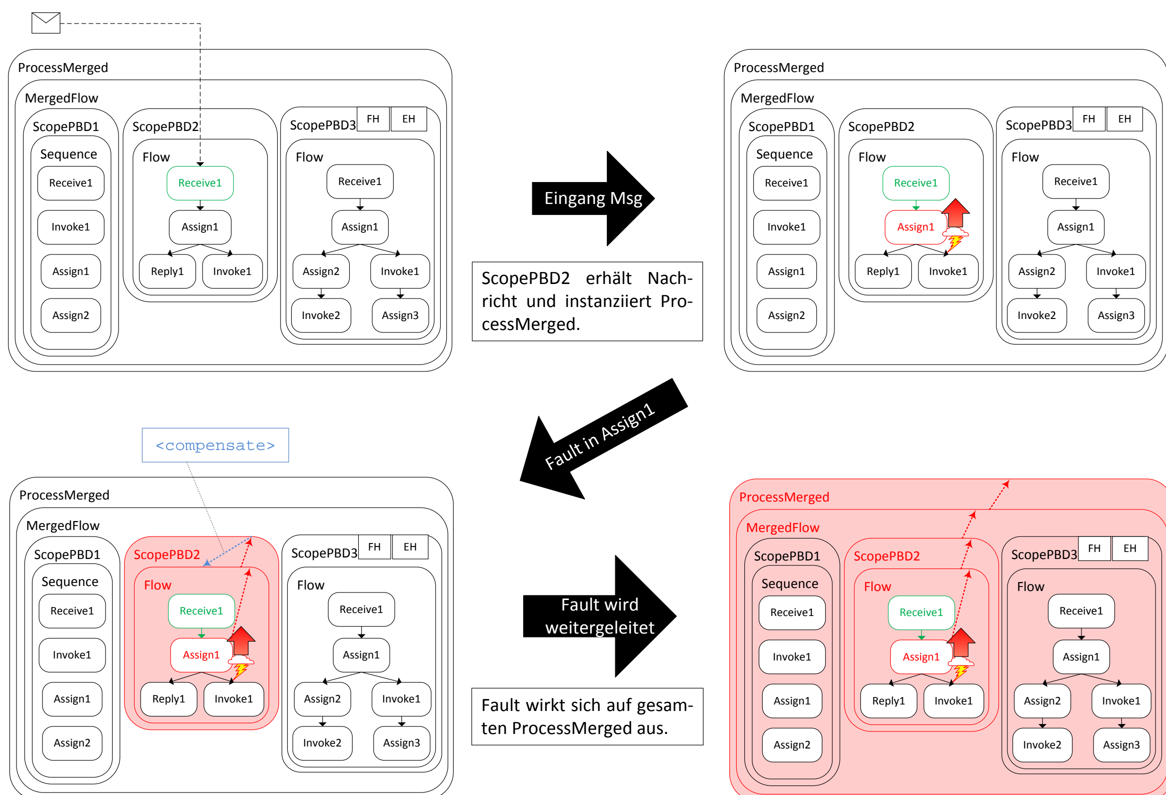


Abbildung 3.61b Verändertes Fehlerverhalten in konsolidiertem Prozess durch nicht gefangenen Fault

Abbildung 3.61b skizziert das Problem der nicht gefangenen Faults: Nachdem die drei PBDs *PBD1*, *PBD2* und *PBD3* in die `<scope>`-Aktivitäten *ScopePBD1*, *ScopePBD2* sowie *ScopePBD3* des konsolidierten Prozesses *ProcessMerged* kopiert wurden, erhält *ScopePBD2* eine Nachricht und instanziiert den Prozess. Anschließend tritt ein Fehler in der `<assign>`-Aktivität *Assign1* auf und ein Fault wird ausgelöst. In *PBD2* würde dieser Fehler lediglich bis zum Prozessscope gelangen und dort den Prozess terminieren, falls er nicht zuvor aufgefangen wurde. Im konsoldierten Prozess dagegen wirkt sich ein solcher möglicher Fehler auf den gesamten Prozessscope von *ProcessMerged* aus und terminiert im ungünstigsten Fall auch die beiden anderen `<scope>`-Aktivitäten *ScopePBD1* sowie *ScopePBD2*. Um einem solchen veränderten Fehlerverhalten entgegenzuwirken, werden die zu konsolidierenden PBDs der Choreographie auf `<catchAll>`-Zweige in den Fault Handlern untersucht, die jeden nicht in einem separaten `<catch>`-Zweig definierten Fault auffangen. Existieren solche Zweige bereits müssen keine Änderungen vorgenommen werden, andernfalls wird in die neue `<scope>`-Akti-

vität ein neuer <catchAll>-Handler hinzugefügt, der lediglich eine <compensate>-Aktivität und keine <rethrow>-Aktivität enthält (vgl. [OAS07] Abschnitt 12.5.1 *Default FCT-Handlers*).

Abbildung 3.61c Ein <catchAll>-Fault Handler mit einer <compensate>-Aktivität sowie die entsprechende XML-Syntax

3.2.2 Generierung des Kontrollflusses

Die grundlegende Idee zur Generierung des expliziten Kontrollflusses im konsolidierten Prozess basieren auf der Überlegung den ehemaligen Nachrichtenfluss zwischen den Teilnehmern der Choreographie mit Hilfe des Austauschs der nachrichtenversendenden- bzw. empfangenden Aktivitäten durch Synchronisationsaktivitäten zu emulieren. Wir werden hierzu <assign>- sowie <empty>-Aktivitäten verwenden. Der folgende Abschnitt beschreibt das korrekte Einbinden dieser Aktivitäten in den Kontrollfluss und der Abschnitt 3.2.2 zeigt welche Daten durch die neuen <assign>-Aktivitäten kopiert werden.

Zuerst werden wir einige grundsätzliche Funktionen und Mengen definieren (vgl. Tabelle 3.5), deren genaue Implementierungen im Kapitel 4 gezeigt werden.

Notation	Definition
null	Steht für die leere Menge ∅
pbd(name)	Funktion die aus der Menge *PBD* die PBD mit dem QName *name* liefert
var(x,name)	Funktion die aus der PBD mit dem Namen *name* die Variable *x* liefert
out(a)	Funktion die zu einer gegebenen Aktivität *a* die Menge aller ausgehenden Links liefert
in(a)	Funktion die zu einer gegebenen Aktivität *a* die Menge aller eingehenden Links liefert
tc(l,a)	Funktion die zu einem gegebenen ausgehenden Link *l* einer Aktivität *a* die transitionCondition liefert
jc(a)	Funktion die zu einer gegebenen Aktivität *a* die joinCondition liefert
prel(a)	Funktion die zu einer gegebenen Aktivität *a* die Menge aller Vorgängeraktivitäten zurückliefert, die mit dieser über die eingehenden Links verbunden sind
succl(a)	Funktion die zu einer gegebenen Aktivität *a* die Menge aller Nachfolgeraktivitäten zurückliefert, die mit dieser über die ausgehenden Links verbunden sind
pre(a)	Funktion die zu einer gegebenen Aktivität *a* die Menge der Vorgängeraktivitäten zurückliefert, die nicht per eingehende Links mit dieser verbunden sind, jedoch vor Beginn von *a* beendet sein müssen
succ(a)	Funktion die zu einer gegebenen Aktivität *a* die Menge der Nachfolgeraktivitäten zurückliefert, die nicht per ausgehende Links mit dieser verbunden sind, jedoch nach dem Beenden von *a* ausgeführt werden
typeof(a)	Funktion die zu einer gegebenen Aktivität *a* deren Aktivitätstyp zurückliefert (mögliche Aktivitätstypen sind hier: <invoke>, <receive>, <reply>, <assign>, <validate>, <throw>, <wait>, <empty>, <exit>, <extensionActivity>, <rethrow>, <compensate>, <compensateScope>, <sequence>, <if>, <while>, <repeatUntil>, <pick>, <flow>, <forEach>, <scope>, <process>)
par(a)	Funktion die zu einer gegebenen Aktivität *a* die umgebende Aktivität zurückliefert (Mögliche Typen sind hier: <scope>, <flow>, <sequence>, <if>, <while>, <repeatUntil>, <pick>, <forEach>, *CompensationHandler (CH)*, *TerminationHandler (TH)*, *EventHandler (EH)*, *FaultHandler (FH)*, <process>)
getSendActivity(ml)	Funktion die zu einem gegebenen Message Link *ml* aus der Menge *ML* die senden-de Aktivität liefert

getReceiveActivity(ml)	Funktion die zu einem gegebenen Message Link *ml* aus der Menge *ML* die empfangende Aktivität liefert
isInvokeAsync(inv)	Funktion die zu einer gegebenen `<invoke>`-Aktivität *inv* prüft, ob diese asynchron kommuniziert (sie definiert nur die `inputVariable`)
isInvokeSync(inv)	Funktion die zu einer gegebenen `<invoke>`-Aktivität *inv* prüft, ob diese synchron kommuniziert (sie definiert `input`- als auch `outputVariable`)
createSource(l,a)	Funktion die zu einem gegebenen Link *l* und einer gegebenen Aktivität *a* in *a* eine neue `<source>` für diesen Link anlegt
createTarget(l,a)	Funktion die zu einem gegebenen Link *l* und einer gegebenen Aktivität *a* in *a* einen neuen `<target>` für diesen Link anlegt
connector(a,b)	Funktion die zu zwei gegebenen Aktivitäten *a* und *b* den von *a* ausgehenden und in *b* eingehenden Link zurückliefert
setJC(act, jc)	Funktion die in der gegebenen Aktivität *act* die `<joinCondition>` *jc* setzt
replaceVar(v_1, v_2, a)	Funktion die alle Vorkommen der Variable v_1 durch v_2 in Aktivität *a* und allen Nachfolgeraktivitäten ersetzt.

Tabelle 3.5 Notation für Funktionen und Mengen des Konsolidierungs-Algorithmus (Erweiterungen)

```
(1)  foreach (ml in ML) do
(2)    s = getSendActivity(ml)                    // sendenden Aktivität
(3)    r = getReceiveActivity(ml)                 // empfangende Aktivität

(4)    if (typeof(s) == <reply>)                  // (1)
(5)      continue                                 // fahre mit nächstem Message Link fort
(6)    fi

(7)    if (typeof(s) == <invoke> && isInvokeAsync(s))  // Wir haben einen asynchronen <invoke>-Aufruf
(8)      asyncMerge(s, r)                         // (2)
(9)    fi

(10)   if (typeof(s) == <invoke> && isInvokeSync(s))   // Wir haben einen synchronen <invoke>-Aufruf
(11)     syncMerge(s, r)                          // (3)
(12)   fi
(13)od
```

Auflistung 3.1 Pseudocode für Merge-Algorithmus

Anmerkungen zum Pseudocode aus Auflistung 3.1:
(1): Falls *s* eine `<reply>`-Aktivität ist, überspringe den Message Link *ml*: Der Message Link *ml* bezieht sich auf eine `<reply>`-Aktivität einer synchronen `<invoke>`-Aktivität (vgl. Abschnitt 2.3.1 Regel II. der Message Links), die zwei Message Links mit letzterer verbindet. Ein Message Link enthält hierbei die `<invoke>`-Aktivität als sendende Aktivität und der hier gefundene selbige als empfangende. Die `<invoke>`-Aktivität wird nur dann behandelt, wenn sie als sendende Aktivität definiert wurde.
(2): Falls *s* eine asynchrone `<invoke>`-Aktivität ist, diese also nur die `inputVariable` definiert, liegt ein asynchrones Kommunikationsmuster vor. Für diesen Fall werden wir nun zwei Konsolidierungsvarianten vorstellen.

Variante 1: Wir entfernen *s* und *r* und fügen eine neue `<assign>`-Aktivität *a* ein. Die eingehenden Links von *s* werden zu den eingehenden Links von *a* (vgl. Abbildung 3.7). Die Menge der eingehen-

den Links von *r* wird zu der Menge der eingehenden Links der Nachfolgeaktivitäten von *r* (hier B2) hinzugefügt. Die Menge der ausgehenden Links von *s* und *r* wird zur Menge der ausgehenden Links von *a*. Die `joinConditions` der Nachfolgeaktivitäten von *r* sind die Konjunktion der zuvor in *r* eingehenden Links (hier *l3*) sowie der aus *r* ausgehenden und in diese Aktivitäten eingehenden Links (hier *l4*). Auflistung 3.2 zeigt den Pseudocode für die *asyncMerge*-Methode.

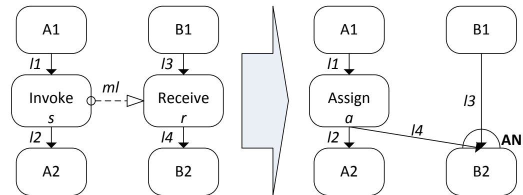


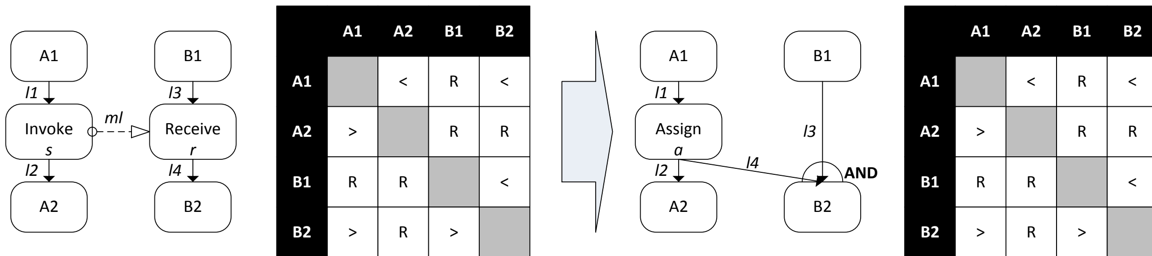
Abbildung 3.7 Asynchrone Konsolidierung Variante 1

```
(1)  asyncMerge(s, r)
(2)  begin
(3)    a = createNewActivity(<assign>, „a”)            // Erzeuge neue <assign>-Aktivität a
(4)    succsR = succl(r) + succ(r)
(5)    foreach (act in succsR) do
(6)      inActFromR = connector(r,act)
(7)      foreach (l in in(r)) do
(8)        createTarget(l, act)
(9)        setJC(act, (l AND inActFromR) + (jc(act) != null ? OR jc(act) : null))
                               // Setze joinConditions
(10)     od
(12)   od
(13)   foreach (l in in(s)) do
(14)     createTarget(l, a)
(15)   od
(16)   foreach (l in out(s)) do
(17)     createSource(l, a)
(18)   od
(19)   foreach (l in out(r)) do
(20)     createSource(l, a)
(21)   od
(22)  remove(s)                        // Entferne s
(23)  remove(r)                        // Entferne r
(24)end
```

Auflistung 3.2 Pseudocode asyncMerge-Methode Variante 1

Variante 1 verwendet hier nur eine `<assign>`-Aktivität und verzichtet auf eine synchronisierende `<empty>`-Aktivität auf der Empfängerseite. Wir wollen kurz anhand des Allen-Kalküls [AL83] zeigen, dass der Kontrollfluss der ursprünglichen simplen Beispielchoreographie aus Abbildung 3.7 erhalten bleibt und anschließend auf die Problematik eingehen, die mit dieser Variante einhergeht. Abbildung 3.9 zeigt die Intervallrelationen vor und nach der Anwendung der Variante 1 des Konsoli-

dierungsalgorithmus für asynchrone `<invoke>`-Aktivitäten. Da die beiden Tabellen gleich sind, bleibt der Kontrollfluss der ursprünglichen Choroegraphiefragmente in diesem simplen Beispiel erhalten.



	A1	A2	B1	B2
A1		<	R	<
A2	>		R	R
B1	R	R		<
B2	>	R	>	

	A1	A2	B1	B2
A1		<	R	<
A2	>		R	R
B1	R	R		<
B2	>	R	>	

Abbildung 3.9 AsyncMerge Variante 1: Die Beispielfragmente aus Variante 1 sowie die dazugehörige Intervallrelationen vor und nach der Konsolidierung.

In der ursprünglichen Choreographie wird auf der sendenden Seite zuerst *A1* ausgefürt und anschließend `<invoke>` *s* und dann *A2*. Auf der Empfängerseite wird zuerst *B1* ausgeführt und nach dem Erhalt der Nachricht in `<receive>` *r B2*. Im konsolidierten Fragment der rechten Seite bleibt diese Kontrollflussabhängigkeit erhalten: Zuerst wird *A1* ausgeführt und anschließend `<assign>` *a* und dann *A2*. *B2* wird erst ausgeführt nachdem *B1* und *a* ausgeführt wurden.
Wir haben jetzt jedoch ein etwas komplexeres Beispiel, bei dem es mit der vorherigen Variante zu Problemen bzw. Änderungen des Kontrollflusses kommt.

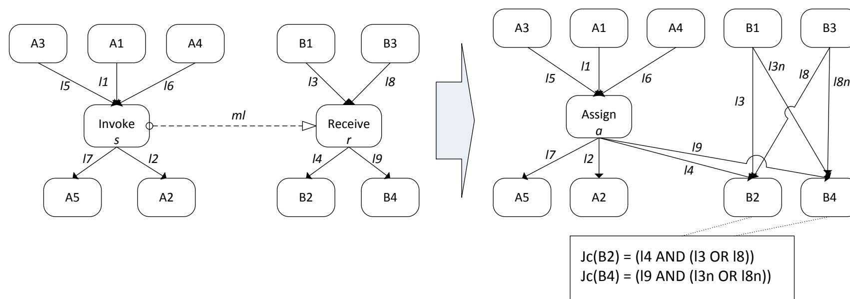


Abbildung 3.10 AsyncMerge Variante 1 Beispiel 2

Abbildung 3.10 zeigt ein erweitertes Beispiel für die Variante 1 des asynchronen Konsolidierungsalgorithmus. Im Gegensatz zum ersten simplen Beispiel müssen hier weitere Links hinzugefügt werden um die ehemals in die empfangende Aktivität *r* eingehenden Links für ihre Nachfolgeaktivitäten zur Verfügung zu stellen (die Links *l3n* sowie *l8n*). Je mehr Nachfolgeaktivitäten *r* besitzt, umso mehr neue Verlinkungen müssen hergestellt und die entsprechenden `joinConditions` angepasst werden (vgl. Abbildung 3.11).

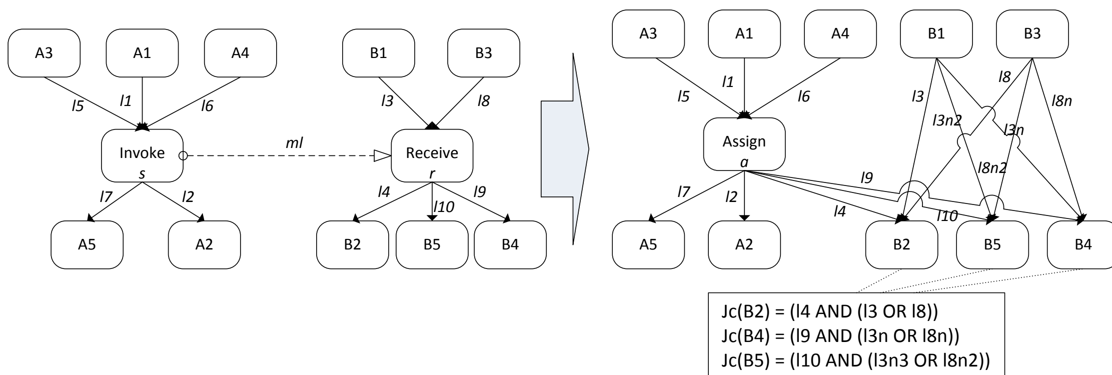


Abbildung 3.11 Vervielfachung der Links

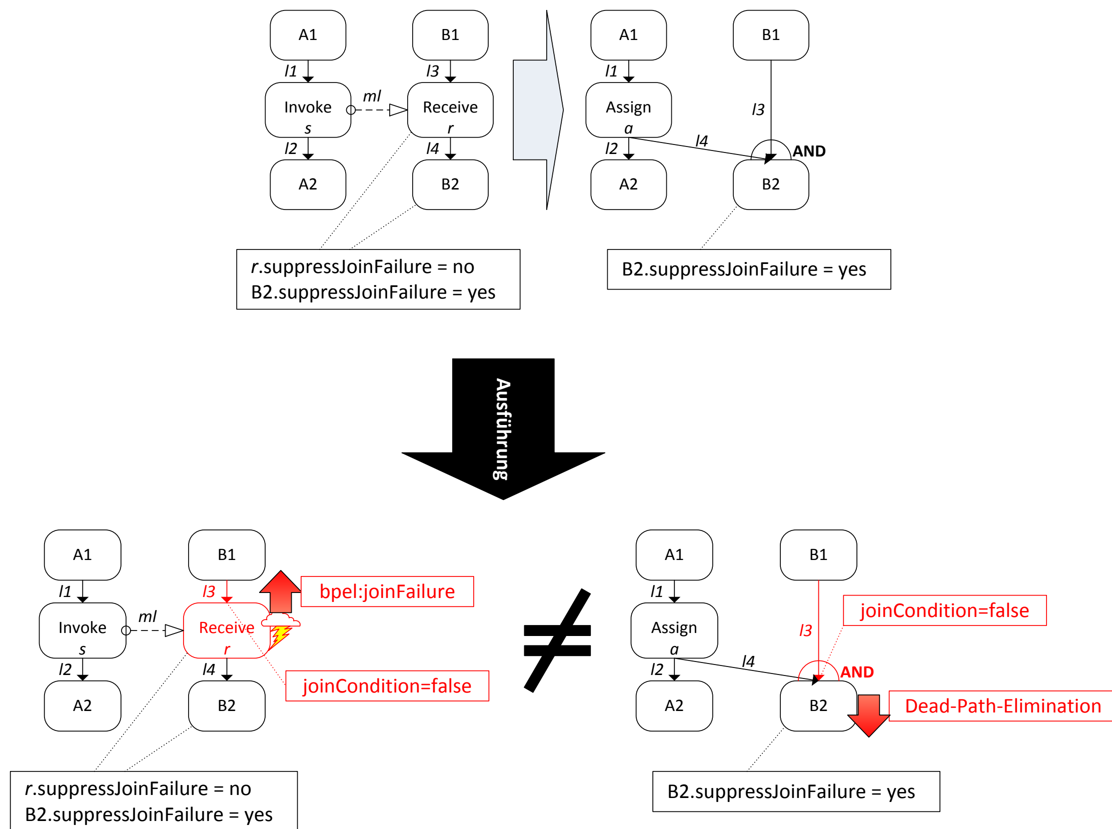


Abbildung 3.12 Veränderter Kontrollfluss bei Variante 1

Das eigentliche Problem stellt jedoch der hier veränderte Kontrollfluss dar, wenn wir uns das Attribut `suppressJoinFailure` anschauen: Angenommen die `<receive>`-Aktivität *r* aus dem ersten Beispiel aus Abbildung 3.7 hat das Attribut `suppressJoinFailure` auf „`no`" gesetzt. Das Vorgehen zum Evaluieren der von Aktivität B1 ausgehenden und in Aktivität *r* eingehenden Links sieht folgendermaßen aus (vgl. WS-BPEL 2.0 Spezifikation [OAS07] Abschnitt *11.6.2 Link Semantics*):

1. Sobald Aktivität B1 beendet und kein Fault an den umschließenden `Scope` weitergereicht wurde, werden die `transitionConditions` der ausgehenden Links evaluiert. Dies ist der Übergang aus dem **Completing**-Zustand zum **Completed**-Zustand des Aktivitäts-Zustandsmodells aus Abschnitt 3.1.2. Nun werden die ausgehenden Links (hier *l3*) instanziiert und evaluiert. Wird keine explizite `transitionCondition` angegeben, so wird der Link zu `true` evaluiert (dargestellt durch den Übergang von **Undetermined** → **Ready** → **Evaluated** aus dem Link-Zustandsmodell Abschnitt 3.1.6).
2. Für jede Aktivität, die eingehende Links von B1 hat (hier *r*) wird überprüft, ob diese Aktivität für die Ausführung bereitsteht. Dies wird durch den Zustandsübergang von **Initial** → **Inactive** im Aktivitäts-Zustandsmodell dargestellt. Trifft dies zu, so wird die `joinCondition` von *r* ausgewertet. Ist das Ergebnis `true` so wird *r* ausgeführt (dargestellt durch den Übergang **Inactive** → **Ready** → **Executing** im Aktivitäts-Zustandsmodell). Bei einer Auswertung zu `false` wird jedoch ein `bpel:joinFailure`-Fault geworfen, da wir das Attribut `suppressJoinFailure` auf „`no`" gesetzt haben und somit keine *Dead-Path-Elimination* durchgeführt wird (dargestellt durch den Übergang **Executing** → **Waiting For Propagation Request**).

Wenden wir jetzt unsere Variante 1 der Konsolidierung an, so wird der Kontrollfluss verändert. Dies veranschaulicht Abbildung 3.12.
Ein naiver Lösungsansatz für dieses Problem wäre das Attribut `suppressJoinFailure` der Nachfolgeraktivitäten von *r* auf „`no`" zu setzen, doch auch dies führt zu einer Veränderung des Kontroll-

flusses, wenn z.B. die Nachfolgeraktivitäten noch andere Vorgängeraktivitäten haben. Abbildung 3.13 veranschaulicht diesen Sachverhalt:

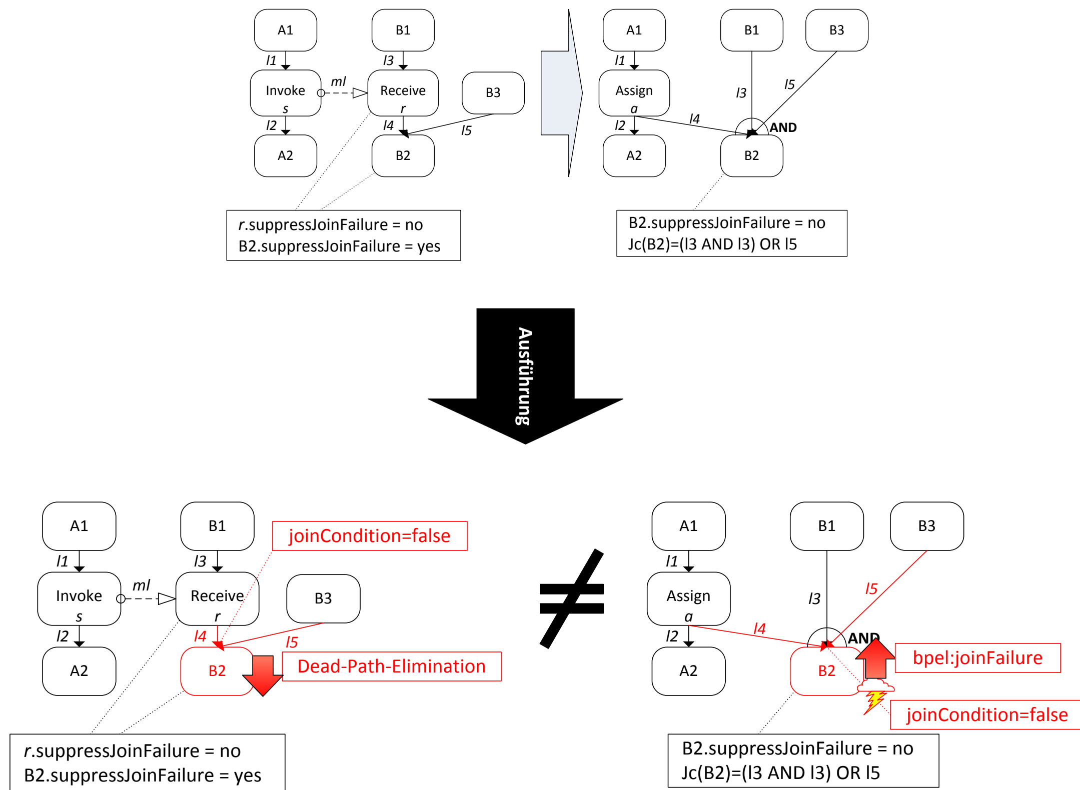


Abbildung 3.13 Propagieren des `suppressJoinFailure`-Attribut-Wertes an Nachfolgeaktivitäten von *r*

Um die zuvor beschriebenen Probleme der Konsolidierung der Variante 1 zu vermeiden werden wir jetzt die zweite Variante vorstellen:

Variante 2: Wir entfernen *s* und *r* und fügen eine neue `<assign>`-Aktivität *a* sowie eine neue `<empty>`-Aktivität *b* auf der empfangenden Seite ein. Die eingehenden Links von *s* werden zu den eingehenden Links von *a*. Die ausgehenden Links von *s* werden zu den ausgehenden Links von *a*. Entsprechendes gilt für die Empfängerseite: Die eingehenden Links von r werden zu den eingehenden Links von *b* und die ausgehenden von *r* zu den ausgehenden von *b*. Abbildung 3.14 zeigt dies an einem simplen Beispiel. Um den Kontrollfluss der beiden Fragmente herzustellen, fügen wir einen neuen Link (hier *l5*) ausgehend von *a* und eingehend in *b* ein. Die `joinCondition` von *b* ist die Konjunktion der eingehenden Links *l3* und *l5* für den Fall, dass noch keine `joinCondition` gesetzt wurde. Ist bereits eine `joinCondition` vorhanden, so wird diese per Konjunktion mit dem neuen Link *l5* kombiniert (vgl. Abschnitt 3.2.2.1.2 `<targets>` und ihre `<joinCondition>`). Mit dieser Variante entfällt die Vervielfachung der Links in den Nachfolgeaktivitäten von *r*, wie sie bei der Variante 1 auftritt. Auch das Problem des veränderten Kontrollflusses durch das Propagieren des `suppressJoinFailure`-Attributs entfällt, da die neue `<empty>`-Aktivität *b* dessen Wert nun von der ursprünglichen `<receive>`-Aktivität *r* übernimmt.

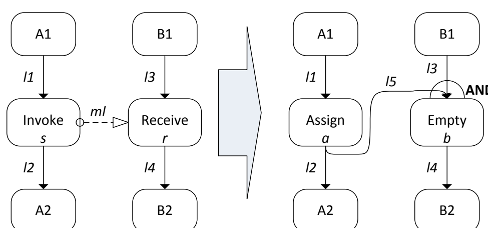
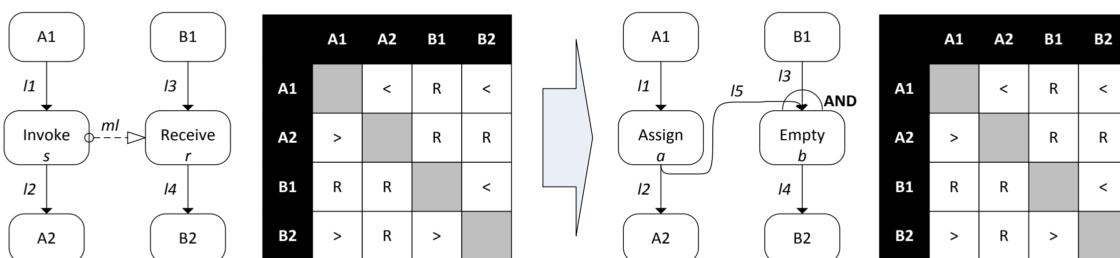


Abbildung 3.14 Asynchrone Konsolidierung Variante 2

```
(1)  asyncMerge(s, r)
(2)  begin
(3)    a = createNewActivity(<assign>, „a“)        // Erzeuge neue <assign>-Aktivität
(4)    b = createNewActivity(<empty>, „b“)         // Erzeuge neue <empty>-Aktivität
(5)    foreach (l in in(s)) do
(6)      createTarget(l, a)
(7)    od
(8)    foreach (l in out(s)) do
(9)      createSource(l, a)
(10)   od
(11)   foreach (l in in(r)) do
(12)     createTarget(l, b)
(13)   od
(14)   foreach (l in out(r)) do
(15)     createSource(l, b)
(16)   od
(17)   nl = new Link()
(18)   createSource(nl, a)
(19)   createTarget(nl, b)
(20)   setJC(b,(nl AND jc(r)))                     // Setze joinCondition
(21)   remove(r)                                   // Entferne r
(22)   remove(s)                                   // Entferne s
(23) end
```

Auflistung 3.3 Pseudocode asyncMerge-Methode Variante 2

Auflistung 3.3 zeigt den Pseudocode für die Variante 2 des asynchronen Konsolidierungsalgorithmus. Hierzu sei angemerkt, dass dieser nur einen ersten Überblick über die nötigen Schritte geben soll und noch nicht vollständig ist. Wir werden ihn im Verlauf des Kapitels vervollständigen und besonders das genaue Definieren der neuen `joinConditions` sowie auf den Einfluss der umschließenden Aktivitäten und Gültigkeitsbereiche der sendenden und empfangenden Aktivitäten eingehen. Zusätzlich wird das Behandeln der `CorrelationSets` beim Konsolidieren gezeigt (vgl. Abschnitt 3.2.3.2).
Abbildung 3.15 zeigt die Intervallrelationen vor und nach der Anwendung der Variante 2 des Konsolidierungsalgorithmus für asynchrone `<invoke>`-Aktivitäten. Die Kontrollflussrelation in der ursprünglichen Situation auf der linken Seite ist identisch mit der in Abbildung 3.9. Durch das Hinzufügen der `<empty>`-Aktivität *b* im konsolidierten Prozess wird die empfangende `<receive>`-Aktivität *r* emuliert und der Wert des `suppressJoinFailure`-Attributs dieser in *b* übernommen.



	A1	A2	B1	B2
A1		<	R	<
A2	>		R	R
B1	R	R		<
B2	>	R	>	

	A1	A2	B1	B2
A1		<	R	<
A2	>		R	R
B1	R	R		<
B2	>	R	>	

Abbildung 3.15 AsyncMerge Variante 2: Die Beispielfragmente aus Variante 2 sowie die dazugehörige Intervallrelationen vor und nach der Konsolidierung.

Nun werden wir den Algorithmus aus Auflistung 3.1 für das Konsolidieren einer synchronen `<invoke>`-Aktivität zeigen:
(3): Falls *s* eine synchrone `<invoke>`-Aktivität ist, diese also `input`- als auch `outputVariable` definiert, existiert eine zusätzlicher Link *ml'* für die zugehörige `<reply>`-Aktivität. In diesem Link ist *s* die empfangende Aktivität und eine `<reply>`-Aktivität *y* die sendende (vgl. Abschnitt 2.3.1 Regel II. der Message Links). Somit liegt ein synchrones Kommunikationsmuster vor. Für diesen Fall werden wir nun ebenfalls zwei Konsolidierungsvarianten vorstellen.

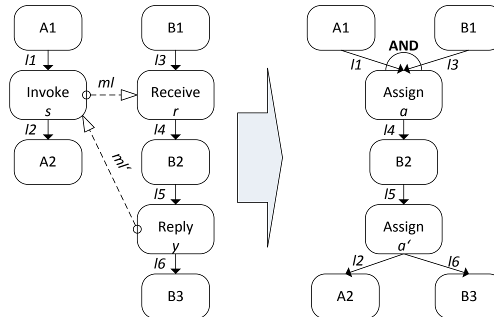


Abbildung 3.16 Synchrone Konsolidierung Variante 1

Variante 1: Eine neue `<assign>`-Aktivität *a* wird hinzugefügt und *s*, *r* und *y* werden entfernt. Wie in Abbildung 3.16 gezeigt, wird die `<reply>`-Aktivität *y* durch eine neue `<assign>`-Aktivität *a'* ersetzt. Analog zum asynchronen Fall, wird die Menge der eingehenden Links und die entsprechenden `joinConditions` von *s* und *r* in einer Konjunktion zu *a* hinzugefügt. Die ausgehenden Links müssen jedoch anders als im asynchronen Fall bearbeitet werden: Die `<receive>`-Aktivität *r* wird beendet sobald die Nachricht der `<invoke>`-Aktivität *s* empfangen wurde. Somit werden die ausgehenden Links von *r* zu den ausgehenden Links von *a*. Die `<invoke>`-Aktivität *s* wird beendet sobald die Nachricht der `<reply>`-Aktivität *y* empfangen wurde, daher werden die ausgehenden Links von *s* und *y* zu den ausgehenden Links von *a'* hinzugefügt. Die eingehenden Links von *y* werden zu den eingehenden Links von *a'*.

Notation	Definition
findReplyML(inv,pbd)	Funktion die zu einer gegebenen `<invoke>`-Aktivität *inv* und einer gegebenen PBD *pbd* den Message Link aus der Menge *ML* liefert, in dem *inv* die empfangende Aktivität ist und *pbd* die sendende PBD zu dieser
getPBD(a)	Funktion die zu einer gegebenen Aktivität *a* die PBD liefert in der diese Aktivität enthalten ist

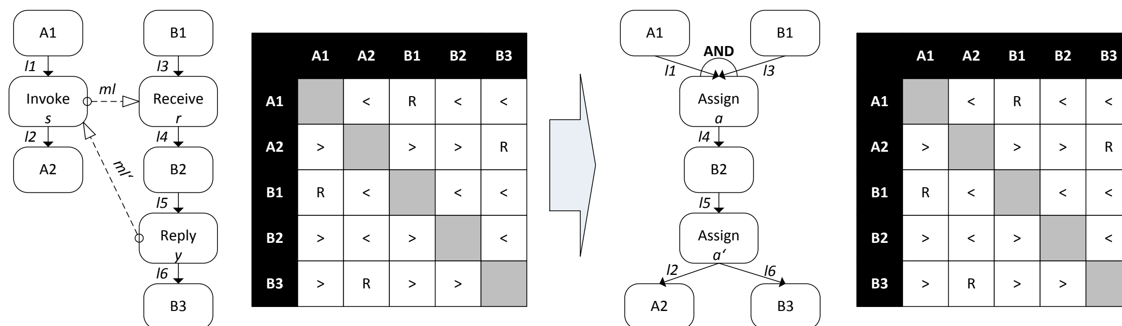
Tabelle 3.6 Notation für Funktionen und Mengen des Konsolidierungs-Algorithmus (Erweiterungen)

```
(1)  syncMerge(s, r)
(2)  begin
(3)    a = createNewActivity(<assign>, „a“)       // Erzeuge neue <assign>-Aktivität a
(4)    a' = createNewActivity(<assign>, „a'“)     // Erzeuge neue <assign>-Aktivität a'
(5)    replyML = findReplyML(s, getPBD(r))
(6)    y = getSendActivity(replyML)
(7)    foreach (l in in(r)) do
(8)      createTarget(l, a)
(9)    od
```

```
(10)  foreach (l in in(s)) do
(11)      createTarget(l, a)
(12)  od
(13)  foreach (l in in(a)) do
(14)      setJC(a, (l + (jc(a) != null ? AND jc(a) : null)))     // Setze joinCondition von a
(15)  od
(16)  foreach (l in out(s)) do
(17)      createSource(l, a')
(18)  od
(19)  foreach (l in out(y)) do
(20)      createSource(l, a')
(21)  od
(22)  setJC(a', jc(y))                          // joinCondition von y nach a' übernehmen
(23)  remove(s)                                 // Entferne s
(24)  remove(r)                                 // Entferne r
(25)  remove(y)                                 // Entferne y
(26) end
```

Auflistung 3.4 Pseudocode syncMerge-Methode Variante 1

Auflistung 3.4 zeigt den Pseudocode für die Variante 1 des synchronen Konsolidierungsalgorithmus. Wir wollen nun zeigen, dass der Kontrollfluss der ursprünglichen Choreographiefragmente gleich dem des neuen Prozessmodell ist. Hierfür verwenden wir wieder die bekannten Intervallrelationen (vgl. Abbildung 3.17). In den Fragmenten der ursprünglichen Choreographie auf der linken Seite wird zuerst *A1* ausgeführt und anschließend `<invoke>` *s*. Im Gegensatz zum asynchronen Fall wird der Kontrollfluss vor *A2* nun blockiert bis die Anwortnachricht von `<reply>` *y* erhalten wurde. Im konsolidierten Fragment auf der rechten Seite emulieren wird die Kontrollflussabhängigkeit zwischen *A1*, `<invoke>` *s* sowie `<receive>` *r* durch eine `<assign>`-Aktivität *a*. *B2* wird erst ausgeführt, wenn *A1*, *B1* sowie *a* ausgeführt wurden. *y* wird durch eine weitere `<assign>`-Aktivität *a'* ersetzt nach deren Beendigung der Kontrollfluss, wie in der ursprünglichen Choreographie, in *A2* und *B3* fortgesetzt wird.



	A1	A2	B1	B2	B3
A1		<	R	<	<
A2	>		>	>	R
B1	R	<		<	<
B2	>	<	>		<
B3	>	R	>	>	

	A1	A2	B1	B2	B3
A1		<	R	<	<
A2	>		>	>	R
B1	R	<		<	<
B2	>	<	>		<
B3	>	R	>	>	

Abbildung 3.17 SyncMerge Variante 1: Die Beispielfragmente aus Variante 1 sowie die dazugehörige Intervallrelationen vor und nach der Konsolidierung. Die Reihen der Tabellen stehen für die linke Seite der Relation, die Spalten für die rechte Seite. „<“ sowie „>“ stehen für die zeitlichen Relationen „vor“ bzw. „nach“ (vgl. **Abbildung 3.8**). „**R**“ steht für alle möglichen Relationen (<, >, m,mi, o, oi, s, si, f, fi, d, di, e).

Bei dieser Variante tritt ein ähnliches Problem, wie auch schon bei der Variante 1 der asynchronen Konsolidierung auf: Wir verschmelzen hier die beiden kommunizierenden Aktivitäten *s* und *r* zu einer neuen `<assign>`-Aktivität *a*. Angenommen *s* hat das `suppressJoinFailure`-Attribut auf „`yes`“

gesetzt *r* jedoch auf „`no`“. Welchen Wert soll nun für dieses Attribut in *a* gesetzt werden ohne den ursprünglichen Kontrollfluss zu verändern? Weiterhin besteht die Möglichkeit, dass *s* einen Compensation sowie Fault Handler definieren kann (vgl. WS-BPEL 2.0 Spezifikation [OAS07] Abschnitt 10.3 *Invoking Web Service Operations – Invoke*). Auf diese Variante der `<invoke>`-Aktivität werden wir im Abschnitt 3.3 eingehen.
Die Abbildungen 3.18a sowie 3.18b zeigen die Problematik am Beispiel unserer Fragmente:

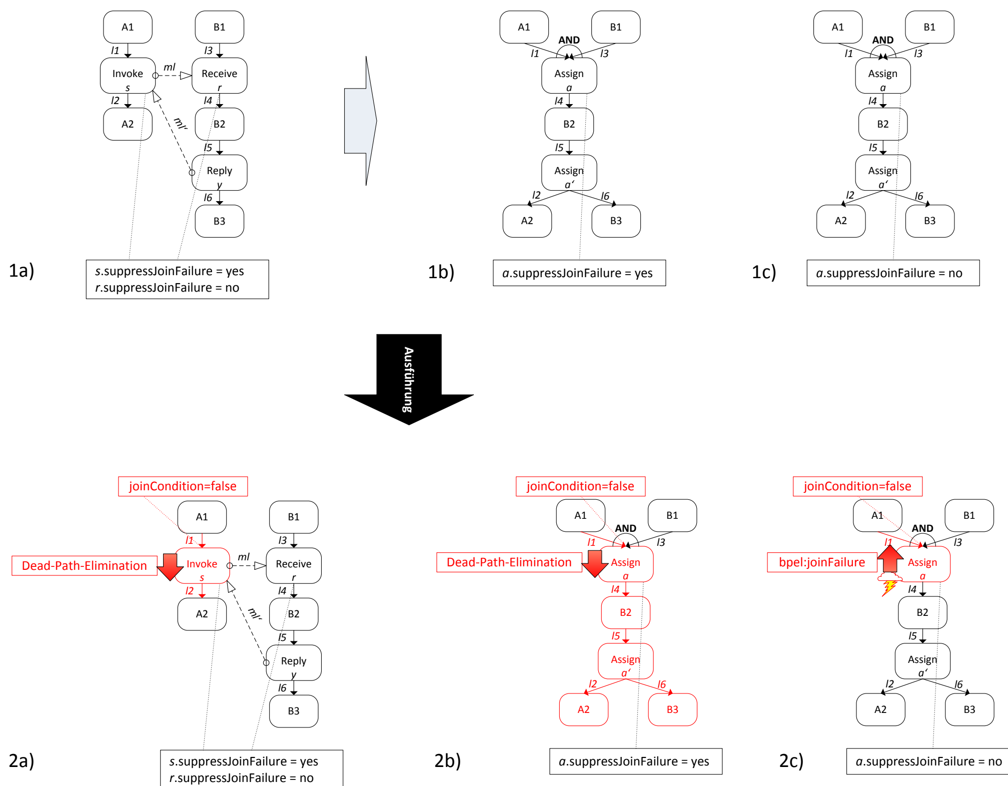


Abbildung 3.18a Variante 1 mit nur einer `<assign>`-Aktivität

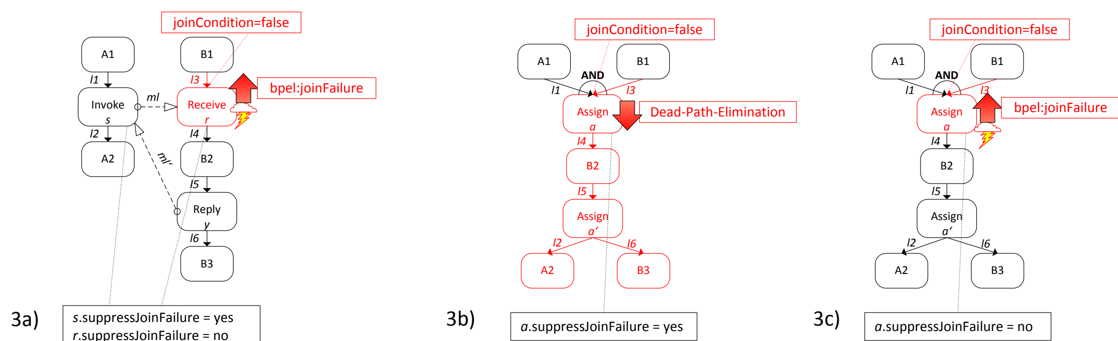


Abbildung 3.18b Variante 1 mit nur einer `<assign>`-Aktivität: 1a) Zeigt die Prozessfragmente in der ursprünglichen Choreographie, 1b) zeigt das Ergebnis nach der Konsolidierung mit Variante 1 und dem Setzen des `suppressJoinFailure`-Attributs auf „`yes`“, 1c) zeigt das Ergebnis nach der Konsolidierung mit Variante 1 und dem Setzen des `suppressJoinFailure`-Attributs auf „`no`“. 2a) zeigt den Kontrollfluss bei Ausführung der ursprünglichen Choreographie und der Auswertung des Links *l2* zu `false`: Die *Dead-Path-Elimination* wird durchgeführt. 2b) zeigt den Kontrollfluss bei Ausführung von 1b) und der Auswertung des Links *l2* zu `false`: Auch hier wird die *Dead-Path-Elimination* durchgeführt. 2c) zeigt den Kontrollfluss von 1c) mit selbigen Bedingungen: Hier wird jedoch eine `bpel:joinFailure` an die Umgebung propagiert. 3a) zeigt der Kontrollfluss der ursprünglichen Choreographie bei Auswertung des Links *l3* zu `false`: Auf der Empfängerseite wird ein `bpel:joinFailure` an die Umgebung propagiert, wogegen in 3b) die *Dead-Path-Elimination* durchgeführt wird. 3c) hat wieder denselben Kontrollfluss, wie die ursprünglichen Choreographiefragmente.

Variante 2: Zwei neue `<assign>`-Aktivitäten *a* und *a‘* werden hinzugefügt und *s*, *r* und *y* werden entfernt. Zusätzlich wird eine synchronisierende `<empty>`-Aktivität auf der Empfängerseite für *r* ein-

gefügt. Die eingehenden Links von *r* werden zu den eingehenden Links von *b*. Wie in Abbildung 3.19 gezeigt, wird die `<reply>`-Aktivität *y* durch eine neue `<assign>`-Aktivität *a‘* ersetzt. Die eingehenden Links von *s* werden zu den eingehenden Links von *a*. Ein neuer Link wird ausgehend von *a* und eingehend in *b* hinzugefügt. Die `<receive>`-Aktivität *r* wird beendet sobald die Anfragenachricht der `<invoke>`-Aktivität *s* empfangen wurde. Somit werden die ausgehenden Links von *r* zu den ausgehenden Links von *b*. Die eingehenden Links von *y* werden zu den eingehenden Links von *a‘*. Die `<invoke>`-Aktivität *s* wird beendet sobald die Nachricht der `<reply>`-Aktivität *y* empfangen wurde, daher werden die ausgehenden Links von *s* und *y* zu den ausgehenden Links von *a‘* hinzugefügt.

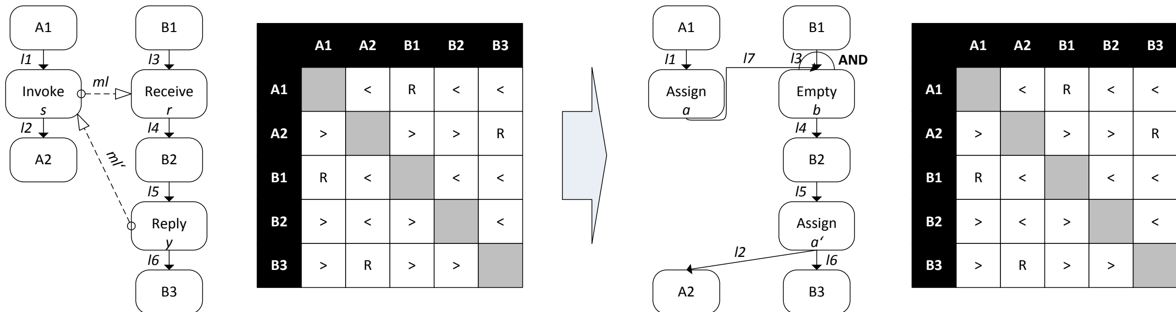


Abbildung 3.19 SyncMerge Variante 2: Die Beispielfragmente aus Variante 2 sowie die dazugehörige Intervallrelationen vor und nach der Konsolidierung. Die Reihen der Tabellen stehen für die linke Seite der Relation, die Spalten für die rechte Seite. „<“ sowie „>“ stehen für die zeitlichen Relationen „vor“ bzw. „nach“ (vgl. **Abbildung 3.8**). „**R**“ steht für alle möglichen Relationen (<, >, m,mi, o, oi, s, si, f, fi, d, di, e).

Wir werden in der vorliegenden Arbeit die Variante 2 für das synchrone Konsolidieren verwenden und diese im Verlauf des Kapitels, parallel zur Variante 2 des asynchronen Konsolidierungsalgorithmus, schrittweise verfeinern.

Der folgende Unterabschnitt beschreibt das genaue Vorgehen beim Setzen der `join`- sowie `transitionConditions` während der Konsolidierung. Daraufhin werden wir auf das Problem der *Peer-Scope-Dependency* [OAS07] eingehen, auf das wir mit unserem vorliegenden Ansatz treffen werden. Abschnitt 3.2.2 Generierung des Datenflusses erläutert die konkreten Details die bei Einfügen der neuen `<assign>`-Aktivitäten beachtet werden müssen.

3.2.2.1 Anpassung der `join`- und `transitionCondition` während der Konsolidierung

In WS-BPEL 2.0 besitzt jede Aktivität die optionalen Standardelemente `<sources>` und `<targets>` mit denen diese Synchronisationsbeziehungen über Links herstellen kann (vgl. Abbildung 2.8). Die dort verwendeten Links müssen in einer umschließenden `<flow>`-Aktivität deklariert werden.

3.2.2.1.1 <sources> und ihre <transitionCondition>s

Jedes `<source>`-Element kann eine optionale `transitionCondition` definieren. Diese wird im **Completed**-Zustand der Aktivität ausgewertet (vgl. Abschnitt 3.1.2 Aktivitäts-Zustandsmodell). Sie muss in der durch das `expressionLanguage`-Attribut spezifizierten Sprache des BPEL-Prozesses vorliegen. Wir fordern für spätere Analysezwecke unseres Konsolidierungsalgorithmus, dass diese die Standardsprache XPath 1.0 [W3C99a] ist. Wird keine angegeben, so wird angenommen, dass der Link des zugehörigen `<source>`-Elements zu `true` gewertet wird. In unseren Konsolidierungsalgorithmen für asynchrone sowie synchrone Kommunikationsmuster aus dem vorangegangenem Abschnitt werden die ausgehenden Links sowie die zugehörigen `transitionConditions` der ursprünglichen sendenden und empfangenden Aktivitäten in die neuen `<assign>`-sowie `<empty>`-Aktivitäten beim Erstellen der neuen Sourcen (*createSource(l,a)*-Funktion) übernommen. Wird eine neuer Link zwi-

schen der <assign>-Aktivität und der entsprechenden synchronisierenden <empty>-Aktivität hinzugefügt, so müssen wir hier keine explizite transitionCondition angeben. Dieser Link emuliert die Nachrichtenflussrelation durch eine Kontrollflussrelation. Abbildung 3.20 zeigt diesen Zusammenhang.

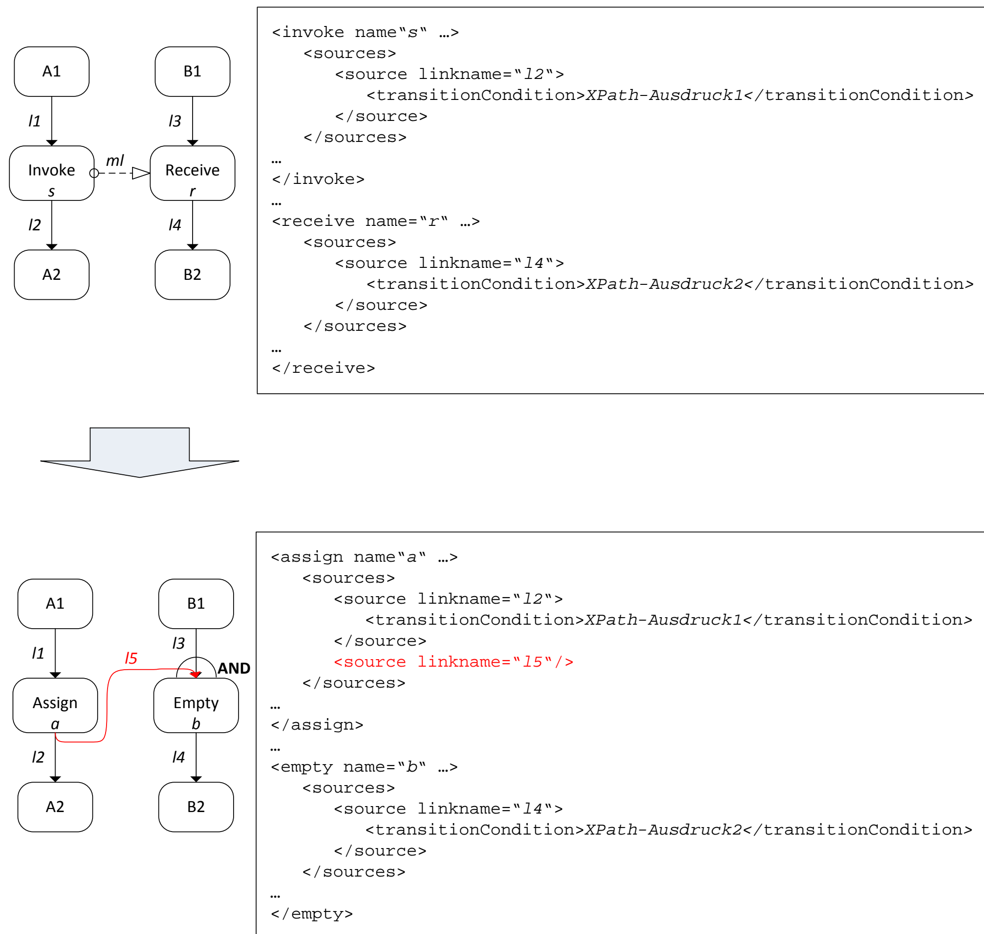
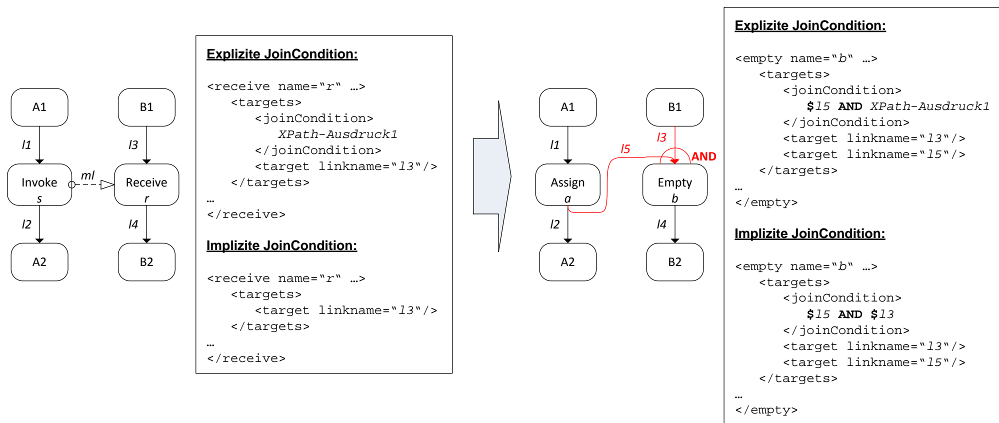


Abbildung 3.20 Übernahme der <source>-Elemente aus ursprünglichen Aktivitäten

3.2.2.1.2 <targets> und ihre <joinCondition>

Das <targets>-Element kann eine optionale joinCondition für alle ihre enthaltenen <target>-Elemente definieren. Diese wird im **Ready**-Zustand der Aktivität ausgewertet (vgl. Abschnitt 3.1.2 Aktivitäts-Zustandsmodell). Wird keine explizite joinCondition angegeben so ist das Ergebnis der Auswertung die Disjunktion über alle Status der in den <target>-Elementen definierten Links. In unseren Konsolidierungsalgorithmen für asynchrone sowie synchrone Kommunikationsmuster aus dem vorangegangenem Abschnitt werden die eingehenden Links sowie die zugehörige joinCondition, falls vorhanden, der ursprünglichen sendenden und empfangenden Aktivitäten in die neuen <assign>-sowie <empty>-Aktivitäten beim Erstellen der neuen Targets (*createTarget(l,a)*-Funktion) übernommen. Im Gegensatz zu den <sources> müssen wir hier jedoch beim Anlegen des neuen Links zwischen <assign>- und synchronisierender <empty>-Aktivität darauf achten, dass die logische Semantik der ursprünglichen joinCondition erhalten bleibt und mit dem neuen Link per Konjunktion verbunden wird. Haben wir beispielsweise mehrere eingehende Links und keine explizite joinCondition, so muss trotzdem sichergestellt werden, dass die *ODER-Semantik* auch in der neuen Konjunktion gültig ist. Abbildung 3.21 veranschaulicht diesen Sachverhalt.



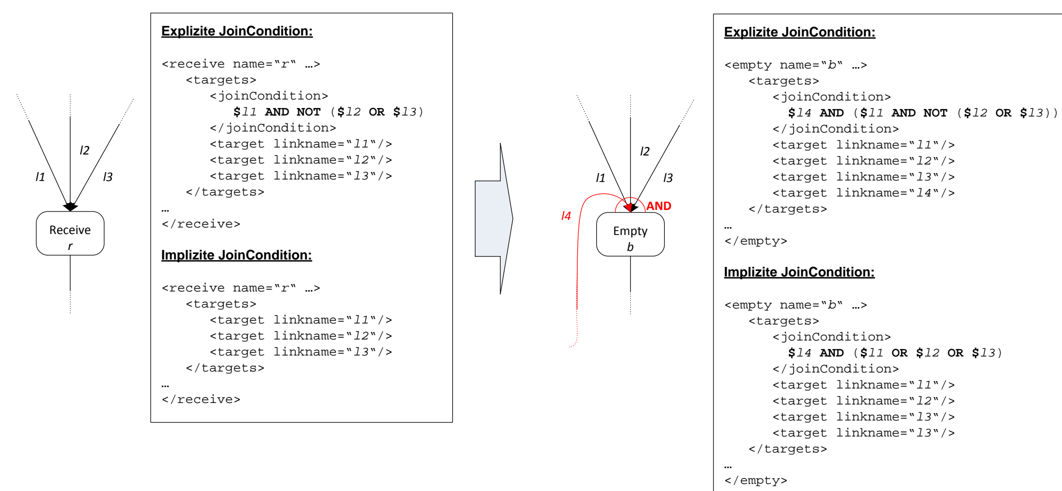


Abbildung 3.21 Anpassungen der `joinCondition` beim Konsolidieren: Das obere Modell zeigt das Anpassen der `joinCondition` an den Choreographiefragmenten im asynchronen Fall. Das untere Modell zeigt das Anpassen der `joinCondition` bei mehreren `targets`. Ist eine `joinCondition` vorhanden so wird diese per Konjunktion mit dem neuen eingehenden `target`-Link verbunden. Gibt es keine, muss darauf geachtet werden die standardmäßige *ODER-Semantik* der ursprünglichen `target`-Links in die Konjunktion einzubinden.

3.2.2.2 Peer-Scope-Dependency Problematik

Definition 3.2.6 (*Control Dependency* vgl. [OAS07]): Gibt es eine Abhängigkeit zwischen einer Aktivität *a* und einer Aktivität *b* aufgrund einer Link Verbindung innerhalb einer `<flow>`-Aktivität oder der Reihenfolge innerhalb einer `<sequence>`-Aktivität, so dass Aktivität *a* vor der Ausführung von Aktivität *b* beendet sein muss, sprechen wir von einer Kontrollflussabhängigkeit (*Control Dependency*).

Definition 3.2.7 (*Peer-Scopes* vgl. [OAS07]): Zwei Scopes *s1* und *s2* heißen *Partnerscopes* (*Peer-Scopes*), wenn sich beide innerhalb desselben unmittelbar umgebenden Scopes bzw. Root-Scope des BPEL-Prozesses befinden.

Definition 3.2.8 (*Scope-Controlled-Set* vgl. [OAS07]): Eine Aktivität *a* ist innerhalb der Menge der *scope-gesteuerten* Aktivitäten des Scopes *s*, falls *a* der Scope *s* selbst oder eine innerhalb *s* eingeschlossene Aktivität ist.

Definition 3.2.9 (*Peer-Scope Dependency* vgl. [OAS07]): Falls *s1* und *s2* Partnerscopes sind und es gibt eine Aktivität *b* in der Menge der scope-gesteuerten Aktivitäten von *s2* sowie eine Aktivität *a* in

der Menge der scope-gesteuerten Aktivitäten von *s1*, so dass *b* eine Kontrollflussabhängigkeit auf *a* hat, besitzt *s2* eine *direkte* Partnerscopeabhängigkeit auf *s1*. Die Partnerscopeabhängigkeitsrelation ist die transitive Hülle der *direkten* Partnerscopeabhängigkeitsrelation.

Die **Regel 1** des WS-BPEL 2.0 Standards [OAS07] aus dem Abschnitt 12.5.2 *Default Compensation Order* schreibt vor, dass zwei Scopes *a* und *b*, von denen *b* eine Kontrollflussabhängigkeit auf *a* hat, im Fall einer Kompensierungsausführung nach der erfolgreichen Ausführung beider Scopes, der Compensation Handler von *b* vor dem von *a* ausgeführt werden muss. Abbildung 3.22 verdeutlicht diesen Zusammenhang.

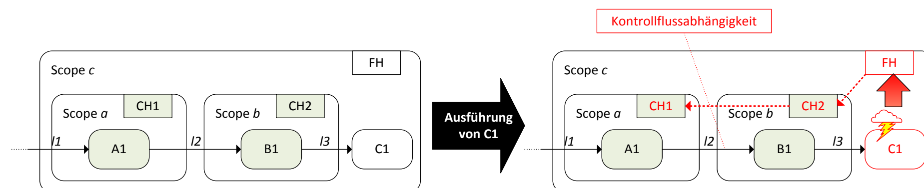


Abbildung 3.22 Kontrollflussabhängigkeit zweier Scopes und die Auswirkung auf das Compensation Handler Verhalten: Scope *a* sowie Scope *b* wurden erfolgreich ausgeführt und die Compensation Handler CH1 und CH2 installiert. Während der Ausführung von Aktivität C1 tritt ein Fault auf der die Kompensation aktiviert. Durch die Kontrollflussabhängigkeit bedingt durch den Link *l2* muss CH2 vor CH1 ausgeführt werden.

Die **Regel 2** des WS-BPEL 2.0 Standards [OAS07][SA00082] verbietet Zyklen in Partnerscopeabhängigkeiten. Somit sind zwei Scopes *s1* und *s2* von denen jeweils *s1* eine Partnerscopeabhängigkeit auf *s2* und *s2* eine auf *s1* hat verboten (vgl. Abbildung 3.23).
Um die zuvor in diesem Kapitel erarbeiteten Vorgehen zur Konsolidierung von synchronen sowie asynchronen Kommunikationsmustern zu ermöglichen werden wir in der vorliegenden Diplomarbeit die Regel 2 auflockern und erlauben Zyklen in Partnerscopeabhängigkeiten. Zukünftige und laufende Arbeiten werden hierzu eine angemessene Lösung liefern.
Die in dieser Arbeit zum Testen der erzeugten konsolidierten Prozesse verwendeten BPEL-Engines *Apache ODE* in Version 1.3.5 [AODE11] sowie *bpel-g* in Version 5.3 [BPLG12] erlauben zusätzlich die Ausführung von Prozessen mit Zyklen in Partnerscopeabhängigkeiten.

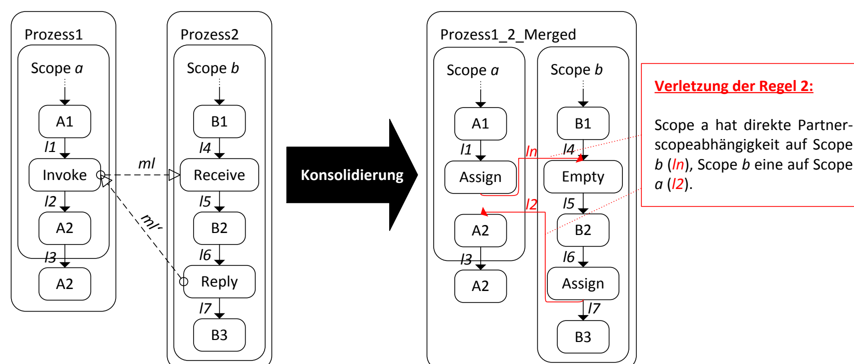


Abbildung 3.23 Zyklus in Partnerscopeabhängkeiten: Wird im konsolidierten Beispiel auf der rechten Seite in der Aktivität A2 eine Fault geworfen und verursacht die Ausführung einer Kompensation, so wird, bedingt durch die Kontrollflussabhängigkeit aufgrund des Links *l2*, zunächst der Compensation Handler von Scope *a* ausgeführt und anschließend der von Scope *b*. Doch dieser hat wieder eine Kontrollflussabhängigkeit mit Scope *a*, bedingt durch den Link *ln*.

3.2.3 Generierung des Datenflusses

Im vorangegangenen Abschnitt wurden die sendenden Aktivitäten durch `<assign>`-Aktivitäten ersetzt, um auf diese Weise die korrekte Synchronisation der ursprünglichen beiden Prozessmodelle im

neuen konsolidierten Prozessmodell zu gewährleisten. Neben der Synchronisation des Kontrollflusses werden die neuen <assign>-Aktivitäten zum Transformieren des ehemaligen Nachrichtenflusses in einen Datenfluss verwendet. Vor der Konsolidierung werden die zu versendenden Nachrichten in einer Variablen gespeichert. Wir nennen diese Variable im Folgenden v_s. Die Nachricht wird anschließend übertragen, empfangen und in einer anderen Variable des empfangenden Prozesses gespeichert. Im Folgenden sei diese Variable v_r. Um den Nachrichtenaustausch in einen Datenaustausch umzuwandeln stehen uns zwei Varianten zu Verfügung:

1. Wir kopieren den Inhalt der Variable v_s in die Variable v_r.

2. Wir ersetzen jedes Vorkommen von v_r nach dem Erhalt der Nachricht durch v_s.

Nehmen wir zum Beispiel an, der Teilnehmer *PBD1* sendet die Variable *varInit* über den Message Link *ml1* in Form der Nachricht *msg1*, die der Empfänger *PBD2* in der Variablen *varReceiveInit* speichert. Somit sehen die kommunizierenden Nachrichten hierfür folgendermaßen aus:

s: <invoke inputVariable="varInit">

r: <receive variable="varReceiveInit">

Benutzen wir die erste Variante so kopiert die neue <assign>-Aktivität *varInit* nach *varReceiveInit* Diese Variante verhindert das Überschreiben von Daten, da die beiden Variablen v_s und v_r immer getrennt behandelt werden und der ehemalige Nachrichtenaustausch durch ein Kopieren der Daten von v_s nach v_r emuliert wird. Abbildung 3.24 veranschaulicht das Vorgehen anhand des asynchronen sowie des synchronen Falls mit zugehöriger <reply>-Aktivität.

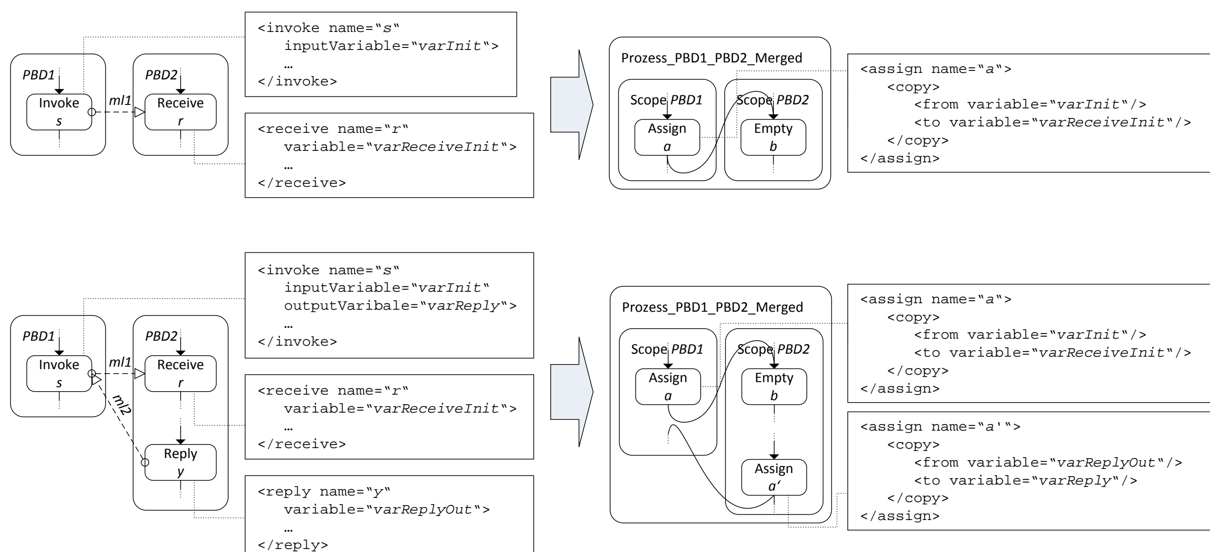


Abbildung 3.24 Austausch der kommunizierenden Aktivitäten durch <assign>-Aktivitäten im konsolidierten Prozess: Die obere Abbildung zeigt den Austausch der sendenden <invoke>- sowie der empfangenden <receive>-Aktivität durch die neue <assign>-Aktivität und das Kopieren der Variablen im asynchronen Fall. Die untere Abbildung zeigt den entsprechenden Austausch der <invoke>-und <reeive>-Aktivitäten sowie der <reply>-Aktivität durch zwei neue <assign>-Aktivitäten im synchronen Fall.

Der Nachteil der ersten Variante ist die Tatsache, dass die Daten nach der <assign>-Aktivität doppelt gespeichert werden. Im Falle komplexer oder großer Datenstrukturen kann dies zu Performanceeinbußen führen. Die zweite Variante bietet hier eine Alternative: Anstatt die Daten von v_s nach v_r zu kopieren, werden alle Zugriffe auf v_r durch v_s ersetzt. Die <assign>-Aktivität kann durch eine <empty>-Aktivität ersetzt werden, die lediglich zum Synchronisieren der Links eingesetzt wird. Problematisch wird die zweite Variante jedoch wenn v_s nach dem Datenaustausch erneut verändert wird: Die zuvor unabhängigen Variablen sind nun durch eine einzelne ersetzt worden. Dies hat zur Folge, dass sich die

Änderungen von v_s auf das Verhalten der lesenden und schreibenden Aktivitäten sowie `join`- und `transitionConditons` auswirken, die zuvor v_r verwendet haben. Das Problem ist als das *Lost Update Problem* bekannt (vgl. [BN09]). Um dieser ungewollten Änderung des Datenflusses entgegenzuwirken verwenden wir zunächst die erste Variante und ermitteln anschließend den expliziten Datenfluss, wie von Kopp et al. in [KKL08] gezeigt. Gibt es keine nachfolgenden schreibenden Zugriffe auf v_s, die auf die `<receive>`-Aktivität folgen, ersetzen wir alle Zugriffe von v_r durch Zugriffe auf v_s. Anschließend ersetzen wir die `<assign>`-Aktivität durch eine `<empty>`-Aktivität. Der Algorithmus zum Analysieren des Datenflusses sowie dem anschließenden möglichen Ersetzen der `<assign>`-Aktivitäten wird in Kapitel 4 Implementierung beschrieben.

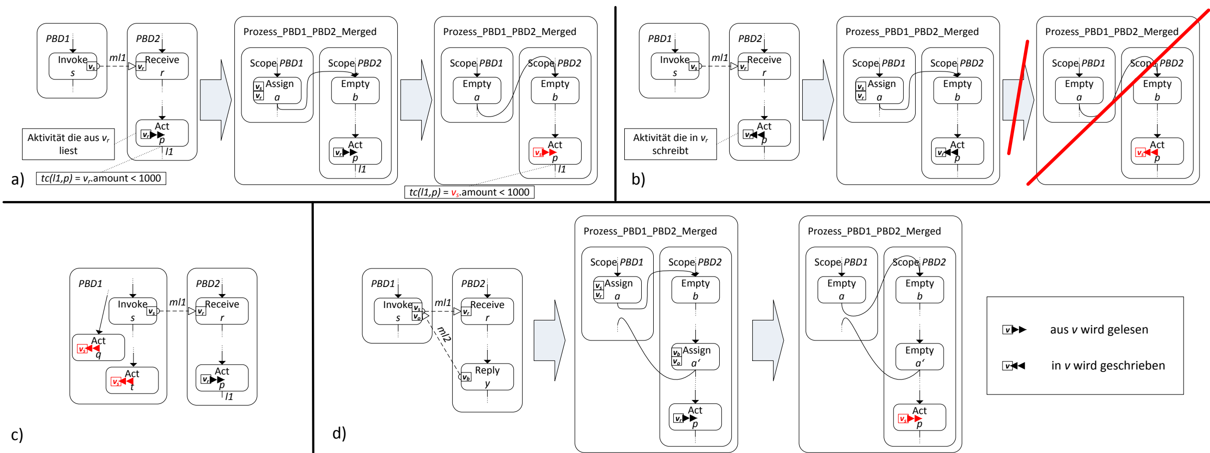


Abbildung 3.25 Optimierungen des Datenflusses im konsolidierten Prozess

Abbildung 3.25 zeigt die möglichen Optimierungen des Datenflusses nach Anwendung des Analysealgorithmus:

Abbildung 3.25 a): Nachdem der Konsolidierungsalgorithmus für asynchrone Kommunikation angewendet wurde, folgt die Analyse des Datenflusses. Dieser muss überprüfen ob auf der ehemals sendenden Seite in v_s schreibende Aktivitäten, die auf die neue `<assign>`-Aktivitäten *a* folgen, Verwendung finden. Dies beinhaltet auch Aktivitäten die z.B. in einem parallelen `<flow>`-Zweig der `<assign>`-Aktivität *a* schreibenden Zugriff auf v_s haben. Hierbei können folgende Aktivitäten schreibenden Zugriff ausüben: `<assign>`s, synchrone `<invoke>`s, `<receive>`s sowie `<pick>`s.

Diese Überprüfung muss auch auf der ehemaligen Empfängerseite nach der neuen `<empty>`-Aktivität *b* sowie möglichen parallelen Kontrollflusszweigen stattfinden, da es sonst durch das Ersetzen von v_r durch v_s zu Seiteneffekten auf der ehemaligen Senderseite kommt. Wurden keine möglichen schreibenden Aktivitäten gefunden, so werden alle Vorkommen von v_r durch v_s ersetzt. Hierbei müssen auch die entsprechenden `transitionCondition`-XPath-Ausdrücke entsprechend angepasst werden.

Abbildung 3.25 b): In diesem Beispielfragment darf die Ersetzung von v_r durch v_s nicht angewendet werden, da sonst der Datenfluss von Scope *PBD1* durch Seiteneffekte der Aktivität *p* verändert wird.

Abbildung 3.25 c): Auch in diesem Beispielfragment darf es zu keiner Ersetzung kommen, da v_s in einer Aktivität *q* eines parallelen Kontrollflusszweigs von *PBD1* sowie einer Nachfolgeaktivität *t* der `<invoke>`-Aktivität *s* beschrieben wird.

Abbildung 3.25 d): Das Beispielfragment zeigt das optimale Einsparungspotenzial von Kopiervorgängen im synchronen Fall. Die zwei `<assign>`-Aktivitäten *a* sowie *a'* können durch zwei synchronisierende `<empty>`-Aktivitäten ersetzt werden.

3.2.3.1 *Voraussetzungen für den korrekten Datenfluss*

Die im vorherigen Abschnitt beschriebene Methode zur Generierung des Datenflusses setzt voraus, dass der sendende Kommunikationsteilnehmer den empfangenden nicht „*überholt*". Somit sollte das Ergebnis des Datenflusses vor und nach dem senden der Daten nicht durch Wettlaufsituationen (*Race Conditions*) bedingt sein. Der WS-BPEL 2.0 Standard definiert hierzu (vgl. [OAS07] Seite 92): „*… Während der Ausführung eines Geschäftsprozesses können Wettlaufsituationen auftreten. Nachrichten, die für eine bestimmte Prozessinstanz bestimmt sind können eintreffen bevor die empfangende* `<receive>`*-Aktivität aktiviert wurde. […] Prozess Engines KÖNNEN verschiedene Mechanismen zur Behandlung einer solchen Wettlaufsituation anwenden. […]*"

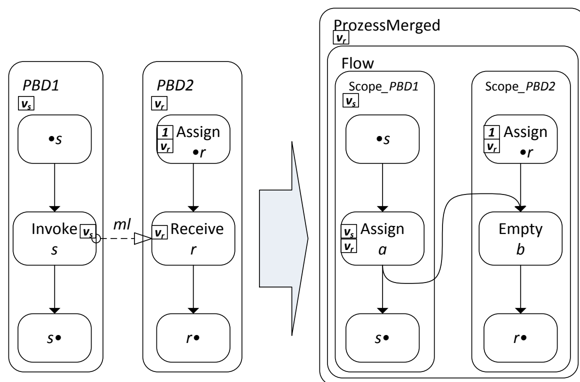


Abbildung 3.25b Race Condition

Abbildung 3.25b zeigt eine solche Wettlaufsituation an einem Beispielfragment mit zwei kommunizierenden PBDs *PBD1* und *PBD2*. Hierbei ist *•s=prel(s)+pre(s)* die Menge aller direkten Vorgängeraktivität von *s*, *s•=succl(s)+succ(s)* die Menge aller direkten Nachfolgeraktivitäten von *s*, *•r= prel(r)+pre(r)* die Menge aller direkten Vorgängeraktivitäten von *r*, mit der `<assign>`-Aktivität *Assign_•r∈ •r* sowie *r•= succl(r)+succ(r)* die Menge aller direkten Nachfolgeraktivitäten von *r*. Die rechte Seite zeigt den konsolidierten Prozess nach Anwendung des AsyncPattern1.1 aus Abschnitt 3.3.1.1. Wenn nun in der linken Beispielchoreographie die sendende Aktivität *s* die Nachricht v_s schickt bevor die `<receive>`-Aktivität *r* aktiviert wurde, ist das Ergebnis der Verarbeitung dieser Nachricht in *PBD2* implementierungsabhängig. Sie kann verworfen oder in eine Warteschlange eingereiht werden (vgl. [LEY10a]). Derselbe Ablauf hat im rechten konsolidierten Prozess *ProzessMerged* unter denselben Bedingungen einen möglichen Datenverlust zur Folge: Die `<assign>`-Aktivität *a* wird ausgeführt bevor *Scope_-PBD2* die `<assign>`-Aktivität *Assign_•r* aktiviert. Da die `<receive>`-Aktivität *r* durch eine synchronisierende `<empty>`-Aktivität *b* ersetzt wurde, wird v_r im Datenfluss nach *Assign_•r* nichtmehr neu gelesen. Wenn jetzt *Assign_•r* einen Wert in v_r schreibt (hier mit ***1*** dargestellt) nachdem *a* ausgeführt wurde, ist das Ergebnis von v_s verloren (*Lost Update*). Einen ähnlichen problematischen Fall stellt folgender Ablauf dar: *Assign_•r* hat seine Ausführung beendet und den Wert ***1*** nach v_r geschrieben. Im selben Moment schreibt *a* v_s nach v_r und überschreibt nun den Wert ***1***. Nun wird der von *Assign_•r* ausgehende und in *b* eingehende Link ausgewertet. Hat dieser eine `transitionCondition`, die v_r verwendet, so ist auch in diesem Fall die Kontrollflusssemantik verändert worden.

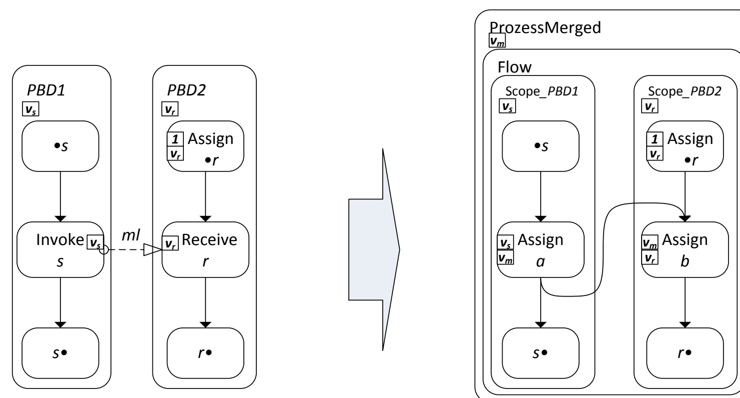


Abbildung 3.22c Behebung des *Lost Update* Problems durch Hinzufügen neuer Nachrichtenvariable v_m

Abbildung 3.25c zeigt eine Alternative zur Lösung des *Lost Update* Problems: Anstatt durch eine synchronisierende `<empty>`-Aktivität wird *r* nun durch eine `<assign>`-Aktivität ersetzt. Zusätzlich hierzu wurde eine neue Nachrichtenvariable v_m eingeführt, die den ursprünglichen Datenfluss emuliert. Somit bleiben die Änderungen von v_r auch im Falle einer Wettlaufsituation unabhängig von möglichen vorhergehenden schreibenden Aktivitäten (*Assign_•r*). Die in den folgenden Abschnitten beschriebenen Konsolidierungsalgorithmen setzen jedoch Choreographien ohne derartige Wettlaufsituationen voraus.

3.2.3.2 Auswirkungen der Konsolidierung auf die verwendeten CorrelationSets

PBDs können, genau wie ausführbare BPEL-Prozesse auch, Korrelationsmengen (CorrelationSets) zur korrekten Zuordnung von Nachrichten an die an einer Kommunikation teilnehmenden Instanzen von Prozessen verwenden. Während der Konsolidierung kommt es jedoch vor, dass einige der Korrelationsmengen obsolet werden, da diese zur Zuordnung von Nachrichten an die korrekte Prozessinstanz dienen und wir den Nachrichtenfluss in einen expliziten Kontrollfluss aus Kombinationen aus `<assign>`- und `<empty>`-Aktivitäten umwandeln. Dies trifft jedoch nicht auf die Korrelationsmengen zu, die in nicht-choreographie-intern kommunizierenden Aktivitäten verwendet werden, also solchen die mit choreographie-externen Partnern in Verbindung stehen. Leider deckt die vorliegende Arbeit nicht alle Kommunikationsmuster der an einer Choreographie teilnehmenden Partner vollständig ab, hierzu gehören z.B. kommunizierende Aktivitäten innerhalb von Schleifen und Event Handlern, sodass einige Aktivitäten auch nach der Konsolidierung an einem prozessinternen Nachrichtenaustausch beteiligt sind und wir daher auch hier nicht alle Korrelationsmengen entfernen können.
Für die technischen Details zur Definition einer Korrelationsmenge sei dem Leser [OAS07] empfohlen. Wir werden das genaue Vorgehen der Anpassung der in der Choreographie definierten und im später konsolidierten Prozess verwendeten Korrelationsmengen im Kapitel 4 Implementierung genauer beschreiben. Da wir keine neuen Korrelationsmengen einführen, sondern lediglich auf die schon definierten zugreifen bzw. diese im optimalen Fall sogar entfernen können, zeigt Abbildung 3.26 die wichtigsten Attribute bei der Verwendung einer Eigenschaft einer Korrelationsmenge.

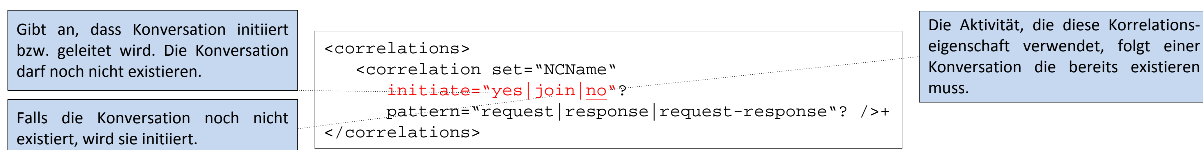


Abbildung 3.26 Syntax einer Korrelationseigenschaft (vgl. [LEY10a])

Korrelationseigenschaften werden nur von kommunizierenden Aktivitäten verwendet. Zu diesen gehören: `<invoke>`-, `<receive>`- sowie `<reply>`-Aktivitäten als auch die `<onMessage>`-Zweige einer `<pick>`-Aktivität und die `<onEvent>`-Zweige eines Event Handler. Da wir keine neuen Kommunikationsaktivitäten einführen, sondern vorhandene choreographie-intern kommunizierende durch `<assign>`- und synchronisierende `<empty>`-Aktivitäten ersetzen, müssen wir lediglich bei choreographie-extern kommunizierenden Aktivitäten in manchen Fällen das `initiate`-Attribut anpassen. Das `pattern`-Attribut, das im Falle synchroner Kommunikation die Korrelationsmenge mit den eingehenden, ausgehenden oder beiden Nachrichten assoziiert, wird nicht verändert.
Wir werden im Folgenden einige kleine Beispielkonversationen modellieren um das Anpassen der Korrelationseigenschaften in choreographie-extern kommunizierenden Aktivitäten genauer zu verdeutlichen.

3.2.3.2.1 Mehrere initiale Startaktivitäten

Eine initiale Startaktivität ist die Startaktivität, die die Instanziierung einer Prozess Instanz auslöst. Dies kann eine `<receive>`- oder eine `<pick>`-Aktivität sein, die das `createInstance`-Attribut auf „`yes`“ gesetzt hat. Bei der Konsolidierung kann es vorkommen, dass zwei oder mehrere ehemals unabhängige PBDs, die jeweils alle eine initiale Startaktivität enthalten, in den neuen verschmolzenen Prozess überführt werden. Verwenden diese Aktivitäten zusätzlich Korrelationsmengen, so müssen sie im neuen konsolidierten Prozess mindestens eine gemeinsame Korrelationsmenge benutzen. Hierzu definiert die WS-BPEL 2.0 Spezifikation folgende Regel (vgl. [SA00075] in [OAS07]): *„Wenn ein Prozess mehrere Startaktivitäten mit Korrelationsmengen beinhaltet, so müssen diese Aktivitäten mindestens eine gemeinsame Korrelationsmenge verwenden. Diese gemeinsamen Korrelationsmengen müssen zusätzlich das `initiate`-Attribut auf „`join`“ gesetzt haben.“*

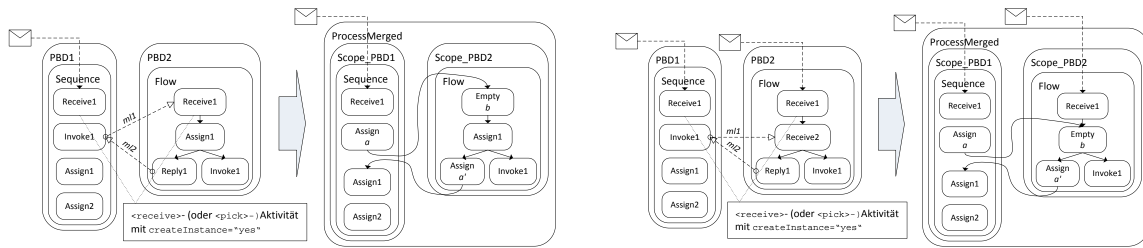


Abbildung 3.27 Verwendung mehrerer initialer Startaktivitäten bei Teilnehmern einer Choreographie: Die linken Beispielfragmente zeigen eine Choreographie mit zwei *PBDs*, die beide eine initiale Startaktivität beinhalten. In PBD1 wird die Instanziierung durch eine externe Nachricht ausgelöst, in PBD2 dagegen durch eine Nachricht, die über den MessageLink *ml1* von PBD1 gesendet wird. Die rechten Beispielfragmente zeigen eine ähnliche Choreographie, in der jedoch die Instanziierung beider PBDs durch externe Nachrichten ausgelöst wird.

Abbildung 3.27 veranschaulicht die Zusammenhänge der initialen Startaktivitäten und der Auswirkungen bei Benutzung von Korrelationsmengen im konsolidierten Prozess: In der linken Beispielchoreographie besitzen beiden Teilnehmer initiale Startaktivitäten, die zusätzlich (nicht im Bild dargestellt) die beiden Korrelationsmengen `corSet1` (PBD1) sowie `corSet2` (PBD2) verwenden. Nach der Konsolidierung werden diese übernommen, müssen jedoch nicht angepasst werden, da der Lebenszyklus des Teilnehmers PBD2 direkt an den Lebenszyklus des Teilnehmers PBD1 gekoppelt ist. PBD2 wird über den Message Link *ml1* und nicht über eine externe Nachricht instanziiert wird. In diesem einfachen Fall kann sogar das `corSet2` aus dem neuen konsolidierten Prozess `ProcessMerged` entfernt werden.

Im rechten Beispiel liegt eine ähnliche Konfiguration vor: Auch hier besitzen beide Teilnehmer initiale Startaktivitäten und verwenden die beiden Korrelationsmengen (nicht im Bild dargestellt) `corSet1` (PBD1) sowie `corSet2` (PBD2) in diesen Aktivitäten. Im Gegensatz zum linken Beispiel wird jedoch die Instanziierung beider Teilnehmer durch externe Nachrichten ausgelöst. Daher muss in diesem Fall eine neue Korrelationsmenge eingeführt werden, da im konsolidierten Prozess nun zwei durch externe Nachrichten ausgelöste instanziierende Startaktivitäten vorliegen, die Korrelationsmengen verwenden (vgl. zuvor definierte Regel der WS-BPEL 2.0 Spezifikation [OAS07] für das Verwenden mehrerer initialer Startaktivitäten). Wir werden hierzu eine neue Korrelationseigenschaft einführen und diese in den konsolidierten Startaktivitäten sowie den dort verwendeten Nachrichten einfügen.

Abbildung 3.28a Beispielfragmente aus Korrelationsbeispielchoreographie: Fragmente aus den beiden PBDs des rechten Beispiels aus Abbildung 3.27 im Zusammenspiel mit den verwendeten Korrelationsmengen.

Abbildung 3.28b Die konsolidierte Beispielchoreographie: Das Ergebnis der Konsolidierung mit angepassten Korrelationsmengen in den initialen Startaktivitäten.

Abbildung 3.28a sowie Abbildung 3.28b zeigen die Anpassungen der Korrelationsmengen- und eigenschaften der rechten Beispielchoreographie aus Abbildung 3.27: Die ursprünglichen technischen Informationen aus den beiden WSDL-Dateien `pbd1.wsdl` sowie `pbd2.wsdl` wurden in die neue `processmerged.wsdl`-Datei des konsolidierten Prozesses übernommen. Zu den bereits verwendeten Korrelationseigenschaften `correlProperty1` sowie `correlProperty2` wurde die neue gemeinsame `commonInitCorrelProperty` hinzugefügt um der Regel für multiple initiale Startaktivitäten der WS-BPEL 2.0 Spezifikation gerecht zu werden. Diese wurde ebenfalls in die von den Startaktivitäten verwendeten Nachrichten sowie als neue Korrelationsmengen in den `<receive>`-Aktivitäten des BPEL-Prozesses (`processmerged.bpel`) eingefügt.

3.2.3.2.2 Anpassung der Korrelationsmengen in choreographie-extern kommunizierenden Aktivitäten

Enthalten die zu konsolidierenden Choreographien zusätzlich Korrelationsmengen, die in den choreographie-extern kommunizierenden Aktivitäten verwendet werden, so müssen diese in einigen Fällen angepasst werden um die korrekte Initialisierung der Konversationen zu gewährleisten. Abbildung 3.29 zeigt einige Beispielfragmente sowie die notwendigen Anpassungen bei der Konsolidierung:

Abbildung 3.29**a**): Die beiden Prozesse *P1* und *P2* sind über die Message Links *ml1*, *ml2* sowie *ml3* an einer choreographie-internen Kommunikation beteiligt. Die sendende Aktivität *S1* von *P1* verwendet hierbei die Korrelationsmengen *corSet1*, *corSet2* sowie *corSet3* und initialisiert *corSet2* und *corSet3*, da *corSet1* bereits in R_1 von *P1* initialisiert wurde. Auf der Empfängerseite *P2* werden *corSet1* sowie *corset2* in R_1 initialisiert und anschließend *corSet3* in der Antwortaktivität S_1 der synchronen Kommunikation, die *ml1* und *ml2* repräsentieren. In den beiden darauffolgenden Aktivitäten S_2 von *P1* sowie R_2 von *P2* sind bereits alle drei Korrelationsmengen initialisiert. In diesem simplen Bespiel entfällt die Anpassung des `initiate`-Attributs, da die Korrelationsmengen *corSet2* sowie *corSet3* nur für choreographie-interne Kommunikation verwendet werden und nach der Ersetzung der ursprünglichen kommunizierenden Aktivitäten (S_1 von *P1*, R_1 von *P2*, S_1 von *P2*, S_2 von *P1* sowie R_2 von *P2*) durch die entsprechenden `<assign>`- und `<empty>`-Aktivitäten (A_1, A_2, A_3, E_1 sowie E_2) obsolet werden.

Abbildung 3.29**b**): Es liegt eine ähnliche Konfiguration wie in a) vor, doch zusätzlich wird eine ausgehende Nachricht in S_3 von *P1* bereitgestellt. Diese sendende Aktivität verwendet die Korrelationsmengen *corSet1*, *corSet2* sowie *corSet3*. Durch die Konsolidierung wird die *corSet2* und *corSet3* initialisierende Aktivität S_1 von *P1* durch A_1 ersetzt. Um die beiden Korrelationsmengen korrekt initialisiert in die gesendete Nachricht von S_3 zu überführen muss das `initiate`-Attribut dieser Aktivität für

corSet2 sowie *corSet3* auf `join` (oder `yes`) geändert werden.
Abbildung 3.29**c**): Die dargestellte Beispielchoreographie (dargestellt durch die Fragmente *P1* und *P2*) kommuniziert mit den externen Prozesspartnern P_{Ext1} sowie P_{Ext2} über die empfangende Aktivität R_1 von *P1* und die sendenden Aktivitäten S_1 von *P2*, S_2 und S_3 von *P1* sowie S_3 von *P2*. Durch die Konsolidierung entfallen die *corSet3* und *corSet4* initialisierenden Aktivitäten S_1 von *P1* sowie S_2 von *P2*, die durch die `<assign>`-Aktivitäten A_1 und A_2 ersetzt wurden. Nun muss wieder sichergestellt werden, dass die Korrelationsmengen *corSet3* und *corSet4* korrekt initialisiert an P_{Ext1} sowie P_{Ext2} versendet werden. Zu diesem Zweck müssen die `initiate`-Attribute von S_2 sowie S_3 im konsolidierten Prozess entsprechend angepasst werden. In diesem Fall verwenden wir `initiate="join"`, da nicht sicher ist, ob zunächst S_2 oder S_3 ausgeführt werden.

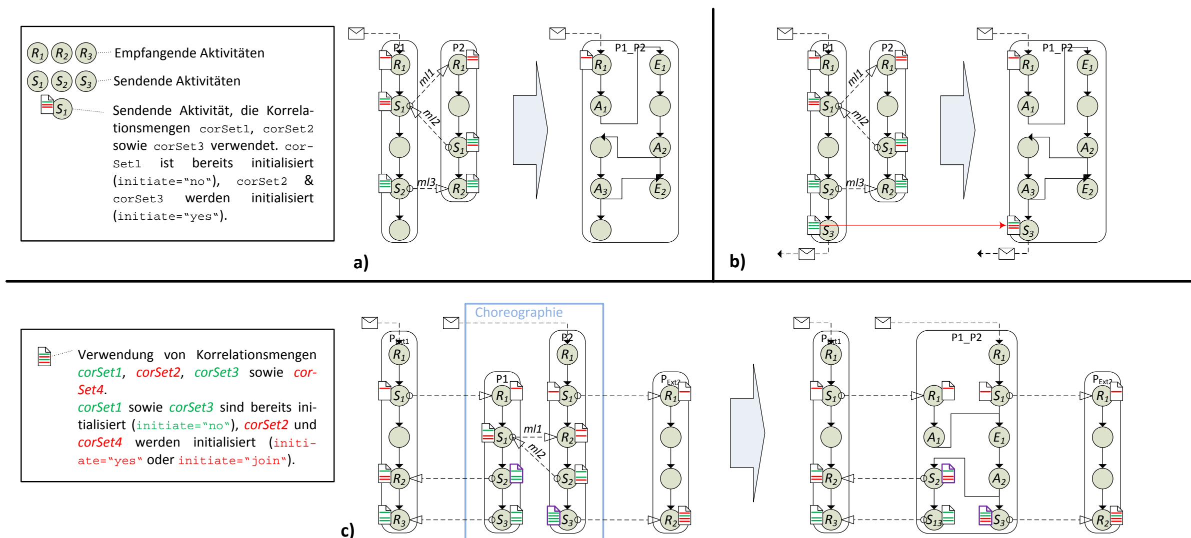


Abbildung 3.29 Anpassung der `initiate`-Attribute in den choreographie-extern kommunizierenden Aktivitäten

3.3 Taxonomie der Konsolidierungsmuster („Merge-Patterns“)

Der folgende Abschnitt zeigt die vom Konsolidierungsalgorithmus verwendeten Muster, im Folgenden *Merge-Patterns* genannt, die zum Auffinden der verschiedenen Senden/Empfangen-Paare und anschließendem Ersetzen durch Synchronisations-Aktivitäten verwendet werden.

3.3.1 Asynchrone Merge-Patterns

Die asynchronen Merge-Patterns sind charakterisiert durch eine sendende und eine empfangende Aktivität, die über einen Message Link miteinander kommunizieren. Als sendende Aktivität steht in BPEL für den asynchronen Fall hierfür die `<invoke>`-Aktivität zur Verfügung. Als empfangende Aktivitäten kommen `<receive>`-Aktivitäten, `<onMessage>`-Zweige der `<pick>`-Aktivität sowie `<onEvent>`-Zweige der Event Handler einer `<scope>`-Aktivität oder des Prozess-Scopes in Frage. Wir werden nun die asynchronen Merge-Patterns ausgehend von den Allgemeinen hin zu den speziellen mit besonderen Umgebungsbedingungen präsentieren. Die Idee zum Auffinden des passenden Merge-Patterns basiert auf dem Iterationsalgorithmus aus Auflistung 3.1, der nach dem Kopieren der ursprünglichen PBDs in neue `<scope>`-Aktivitäten die Message Links aus der Topology untersucht und das passende Konsolidierungsmuster liefert. Abbildung 3.30 zeigt den schematischen Aufbau dieses Vorgehens. Es wird jedoch auch Fälle geben in denen es kein passendes Merge-Pattern gibt. In diesem Fall werden die kommunizierenden Aktivitäten nicht ersetzt. Da es sich anschließend um intra-Prozess-kommunizierende Aktivitäten handelt, teilen wir der verwendeten BPEL-Engine stattdessen mit das SOAP-Message-Handling in diesen Fällen zu umgehen (*SOAP-Bypassing*). Die verwendeten

BPEL-Engines *Apache ODE* [AODE11] sowie *bpel-g* [BPLG12] bieten hierfür spezielle Konfigurationsoptionen an, die beim Bereitstellen gesetzt werden. Wir werden diese am Ende des Abschnitts kurz vorstellen.

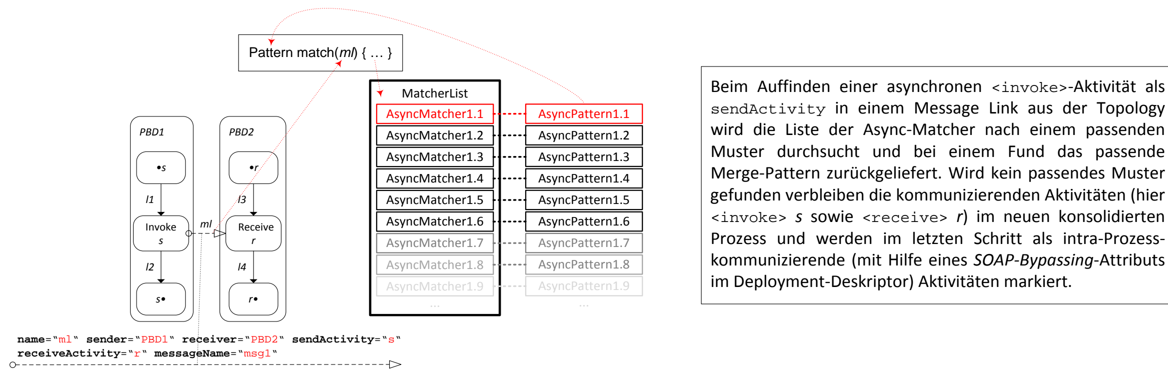


Abbildung 3.30 Anwendung des Merge-Algorithmus: Für jeden asynchron kommunizierenden Link wird die Liste der bekannten AsyncMatcher nach einem Muster durchsucht und im Falle einer Übereinstimmung das entsprechende AsyncPattern zurückgeliefert.

3.3.1.1 AsyncPattern1.1

Das AsyncPattern1.1 ist das einfachste Merge-Pattern und basiert auf den Überlegungen aus der Variante 2 der asynchronen Kommunikationsmuster aus Abschnitt 3.2.2. Die sendende <invoke>-Aktivität als auch die empfangende <receive>-Aktivität können Vorgänger- sowie Nachfolgeaktivitäten im Kontrollfluss haben.

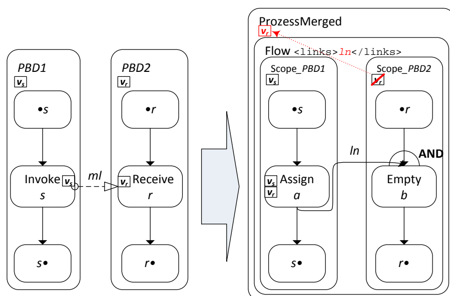


Abbildung 3.31 AsyncPattern1.1

Abbildung 3.31 zeigt die Beschaffenheit der zwei Teilnehmer des AsyncPattern1.1 sowie die Änderungen bei der Konsolidierung: *PBD1* kommuniziert per <invoke>-Aktivität *s* mit der <receive>-Aktivität *r* in *PBD2*. Beide Aktivitäten haben Vorgänger- als auch Nachfolgeraktivitäten im Kontrollfluss. *•s* bezeichnet hier die Menge aller direkten Vorgängeraktivitäten von *s*, *s•* die Menge aller direkten Nachfolgeraktivitäten von *s*. Entsprechend steht *•r* für die Menge aller direkten Vorgängeraktivitäten von *r*, *r•* für die Menge aller direkten Nachfolgeraktivitäten von *r*. Es gilt hierbei *•s=prel(s)+pre(s)*, *s•=succl(s)+succ(s)*, *•r=prel(r)+pre(r)* und *r•=succl(r)+succ(r)* (vgl. Tabelle 3.5). Wir ersetzen im konsolidierten Prozess *s* durch die <assign>-Aktivität *a*, die v_s an v_r zuweist. Da v_r im Scope *Scope_PBD2* definiert wurde und nur dort sichtbar ist, müssen wir hierzu v_r in den globalen Prozessscope verschieben. Hierbei müssen wir aufpassen, dass es nicht schon eine globale Variable mit demselben Namen gibt. Ist dies der Fall benennen wir v_r in $v_{r'}$ um und ändern alle Verweise von v_r in *Scope_PBD2* sowie *a* auf $v_{r'}$. Der neue Kontrollflusslink *ln*, der nun *a* mit der synchronisierenden <empty>-Aktivität *b* verbindet, wird zu den Links in der <flow>-Aktivität des konsolidierten Prozesses hinzugefügt. Auch hier muss wieder auf mögliche Namenskollisionen geprüft werden. Wie in den Abschnitten 3.2.2.1.1 sowie 3.2.2.1.2 erläutert, müssen die <transitionCondition>s von *a* nicht angepasst werden, jedoch die <joinCondition> von *b* im Falle anderer eingehender Links als *ln*.

3.3.1.1.1 <invoke> mit FHs und CHs

WS-BPEL 2.0 erlaubt die direkte Definition von Compensation sowie Fault Handlern innerhalb einer <invoke>-Aktivität. Eine solche <invoke>-Aktivität ist semantisch äquivalent zu einer <scope>-

Aktivität, die diese <invoke>-Aktivität enthält und die den CH sowie die FHs dieser definiert. Sie trägt denselben Namen, wie die enthaltene <invoke>-Aktivität.

Abbildung 3.32 zeigt das Beispiel aus dem Async-Pattern1.1 mit einem solchen <invoke> und die nötigen Anpassungen für das Konsolidierungsmuster: Im neuen konsolidierten Prozess *ProzessMerged* wird nun eine neue <scope>-Aktivität *s* definiert, die die neue <assign>-Aktivität *a* umschließt und die FHs sowie den CH von <invoke> *s* enthält. Zusätzlich werden die eingehenden Links von <invoke> *s* zu den eingehenden Links von <scope> *s* (hier *l1*) und die ausgehenden Links von <invoke> *s* zu den ausgehenden Links von <scope> *s* (hier *l2*).

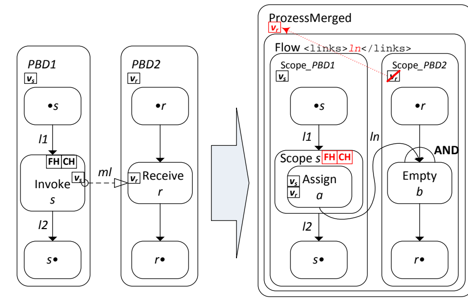


Abbildung 3.32 <invoke> **mit FH und CH**

3.3.1.1.2 <empty>-Optimierer

Wie in Abschnitt 3.2.2 beschrieben, gibt es in der Variante eins der Konsolidierung asynchroner Kommunikationsmuster die Möglichkeit auch die synchronisierende <empty>-Aktivität zu entfernen. Der folgende Abschnitt beschreibt eine Optimierungsfunktion, die unter bestimmten Bedingungen den Kontrollfluss des konsolidierten Prozesses so anpasst, dass wir auf das <empty> verzichten können. Hierzu werden wir einige der Funktionen aus Tabelle 3.5 definieren.

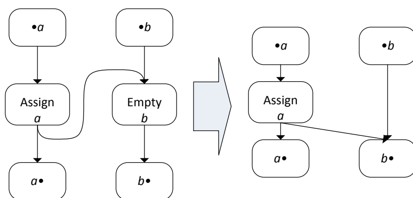


Abbildung 3.32b <empty>**-Optimierung**

Abbildung 3.32b zeigt das Ergebnis der Optimierung nach Ausführung des Algorithmus. Um mögliche bpel:joinFailure-Faults nicht zu unterdrücken, setzen wir voraus, dass in *b* suppressJoinFailure auf „yes" gesetzt ist. Wir werden im Optimierungsschritt die <transitionCondition>s sowie <joinCondition>s der Vorgänger sowie Nachfolgeraktivitäten von *b* anpassen und neue Links in diesen anlegen um den Kontrollfluss mit dem ursprünglichen identisch zu halten. Hierbei gilt wieder: $\bullet b = prel(b) + pre(b)$, $b\bullet = succl(b) + succ(b)$, mit $a \in prel(b)$.

```
(1)  optimizeEmpty(act_empty)
(2)  begin
(3)    if (act_empty.suppressJoinFailure == "yes")
(4)      if ( size(prel(act_Empty))>0 || size(succl(act_Empty))>0 )
(5)        foreach ( succ_Act in act_Empty• ) do
(6)          Link e2succ = connector(act_empty, succ_Act)
(7)          TC tc_succAct = tc(e2succ, act_Empty)
(8)          JC jc_succAct = jc(succ_Act)
(9)          JC jc_Empty = jc(act_empty)
(10)         foreach ( pre_Act in •act_Empty ) do
(11)           Link pre2e = connector(pre_Act, act_empty)
(12)           TC tc_preAct = tc(pre2e, pre_Act)
(13)           Link newLink = createLink(pre_Act)
(14)           Source newSource = createSource(newLink, pre_Act)
(15)           if ( tc_succAct ≠ null || tc_preAct ≠ null )
(16)             newSource.tc = combine(tc_succAct, tc_preAct)
(17)           fi
(18)           removeSource(pre2e, pre_Act)
(19)           Target newTarget = createTarget(newLink, succ_Act)
(20)           if ( jc_Empty == null )
```

```
(21)            jc_succAct = replaceLink(e2succ, newLink, jc_succAct)
(22)          else
(23)            jc_Empty = replaceLink(pre2e, newLink, jc_Empty)
(24)          fi
(25)        od
(26)        JC jc_New = null
(27)        if ( e2succ == null )
(28)          jc_New = combine(jc_Empty, jc_succAct)
(29)        else
(30)          jc_New = replaceLinkWithJC(e2succ, jc_Empty, jc_succAct)
(31)        fi
(32)        setJC(succ_Act, jc_New)
(33)        removeTarget(e2succ, succ_Act)
(34)      od
(35)    fi
(36)    remove(act_empty)
(37)  fi
(38) end
```

Auflistung 3.5 Pseudocode `<empty>`-Optimierungsalgorithmus

Auflistung 3.5 zeigt den Pseudocode für den Optimierungsalgorithmus für `<empty>`-Aktivitäten. Nach Eingabe einer `<empty>`-Aktivität (act_{Empty}) wird geprüft ob die Dead-Path-Elimination für diese Aktivität gesetzt wurde (Zeile 3). Anschließend werden für alle Nachfolgeraktivitäten in $act_{Empty}\bullet$ folgende Schritte durchgeführt: Wir ermitteln den Link *e2succ*, der $succ_{Act}$ und act_{Empty} verbindet (Zeile 5). Handelt es sich bei $succ_{Act}$ um eine Aktivität aus $succ(act_{Empty})$, die nicht per Link mit act_{Empty} verbunden ist, so bleibt dieser Link leer. Daraufhin speichern wir, falls vorhanden, die `<transitionCondition>` dieses Links in $tc_{succAct}$, die `<joinCondition>` von $succ_{Act}$ in $jc_{succAct}$ sowie die `<joinCondition>` von act_{Empty} in jc_{Empty} (Zeilen 6-8), da wir sie in den nächsten Schritten zu neuen kombinierten `<joinCondition>` sowie `<transitionCondition>`s transformieren. In der folgenden Schleife (Zeilen 10-25) kombinieren wir die `<transitionCondition>`s aller Vorgängeraktivitäten von act_{Empty} ($\bullet act_{Empty}$) mit den `<transitionCondition>`s des Links, der act_{Empty} mit $succ_{Act}$ verbindet, falls vorhanden. So stellen wir sicher, dass die `<transitionCondition>`s von act_{Empty} in die Vorgängeraktivitäten übertragen werden und der ursprüngliche Kontrollfluss erhalten bleibt. Anschließend wird ein neuer Link mit Ausgang in pre_{Act} und Eingang in $succ_{Act}$ und der kombinierten `<transitionCondition>` aus $tc_{succAct}$ und tc_{preAct} angelegt. Hierzu werden $tc_{succAct}$ und tc_{preAct} per UND-Verknüpfung miteinander verbunden (Zeile 15 $combine(tc_{succAct}, tc_{preAct})$). Besitzt act_{Empty} keine eingehenden Links und ist jc_{Empty} somit leer, ersetzen wir alle Vorkommen von *e2succ* in $jc_{succAct}$ durch den neuen Link *newLink* (Zeile 21). Andernfalls wird die `<joinCondition>` jc_{Empty} angepasst, indem der alte Link *pre2e* durch den neuen *newLink* im XPath-Ausdruck ersetzt wird (Zeile 23 $replaceLink(pre2e, newLink, jc_{Empty})$). Nachdem alle Aktivitäten aus $\bullet act_{Empty}$ angepasst wurden, wird die neue `<joinCondition>` jc_{New} von $succ_{Act}$ angelegt. Existiert kein Link *e2succ*, der act_{Empty} mit $succ_{Act}$ verbindet, so ist jc_{New} die UND-Verknüpfung aus jc_{Empty} und $jc_{SuccAct}$ (Zeile 28). Andernfalls werden alle Vorkommen des Links *e2succ* mit der neuen kombinierten jc_{Empty} `<joinCondition>` in $jc_{succAct}$ ersetzt (Zeile 30 $replaceLinkWithJC(e2succ, jc_{Empty}, jc_{succAct})$). Nach erfolgreicher Zusammenführung der Vorgänger- und Nachfolgeraktivitäten von act_{Empty} im Kontrollfluss, wird act_{Empty} entfernt. Sei $m=size(act_{Empty}\bullet)$ die Anzahl der Aktivitäten in $act_{Empty}\bullet$ und $n=size(\bullet act_{Empty})$ die Anzahl der Aktivitäten in $\bullet act_{Empty}$, dann müssen wir in unserem Optimierungsalgorithmus $n*m$ neue Links anlegen. Auflistung 3.6 zeigt den Pseudocode für das Ermitteln der `<joinCondition>` einer Aktivität. Hierzu wird lediglich geprüft, ob es eine explizite `<joinCondition>` in dieser Aktivität gibt, ansonsten werden die eingehenden Links der `<targets>` per ODER-Verknüpfung miteinander kombiniert (implizite `<joinCondition>`).

```
(1)  JC jc(act)
(2)  begin
(3)    JC jc = null
(4)    if ( act.jc ≠ null )
(5)      return act.jc
(6)    else
(7)      for ( i = 0; i < size(act.<targets>); i++ ) do
(8)        jc += "$" + act.<targets>[i]
(9)        jc = ( i < (size(act.<targets>)-1) ? jc + " OR " : jc )
(10)     od
(11)   fi
(12)   return jc
(13) end
```

Auflistung 3.6 Pseudocode ***jc(act)*****-Funktion**

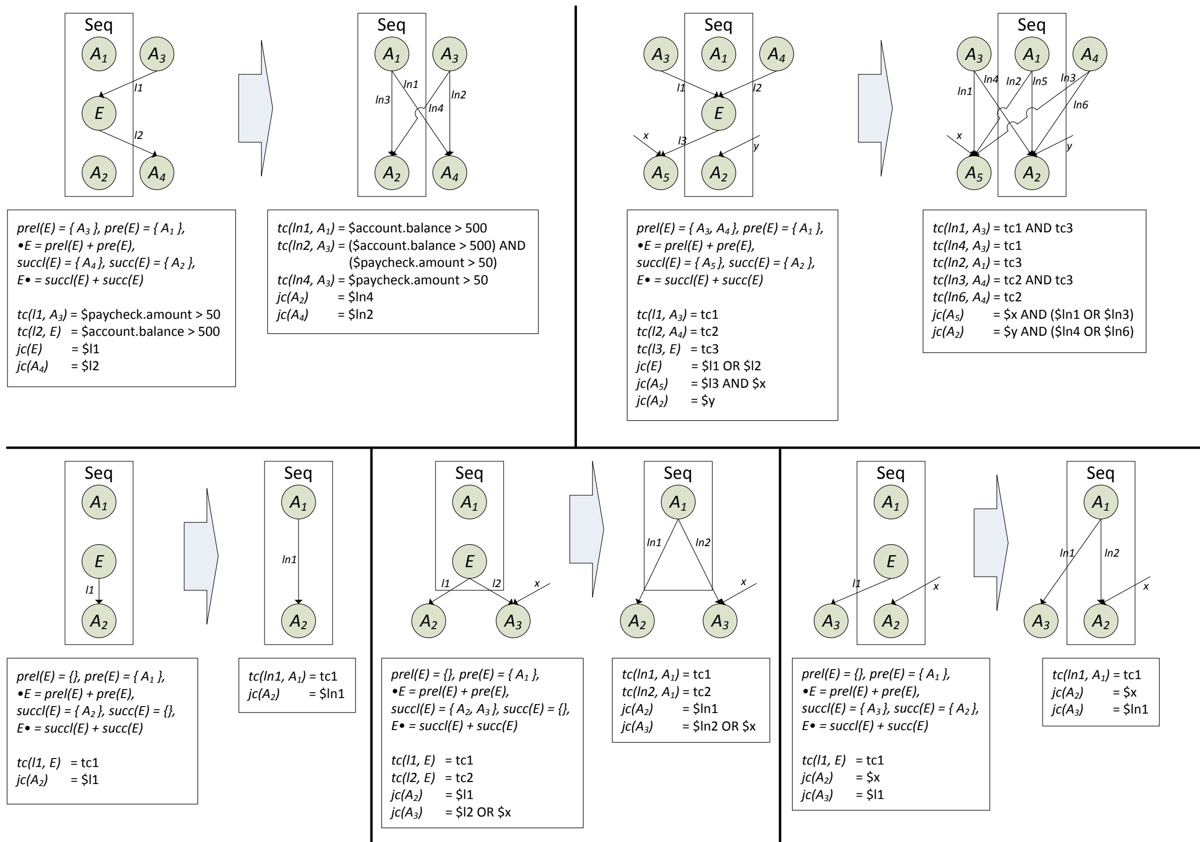


Abbildung 3.32c Fallbeispiele einiger `<empty>`-Optimierungen

Abbildung 3.32c zeigt den Optimierungsalgorithmus an einigen simplen Beispielfragmenten.

3.3.1.2 AsyncPattern1.2

Das AsyncPattern1.2 stellt eine Spezialisierung des AsyncPattern1.1 dar, mit der Bedingung, dass für die Menge $r\bullet = \emptyset$ gilt, es somit keine Nachfolgeaktivitäten auf die empfangende Aktivität gibt. Abbildung 3.33 zeigt einige Beispielfragmente einer Choreographie mit der gegebenen Ausgangssituation.

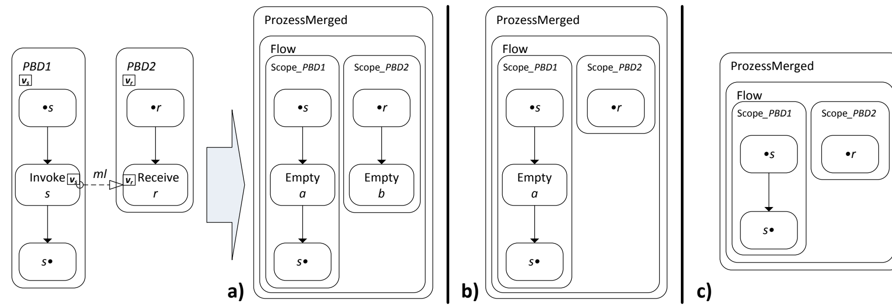


Abbildung 3.33 AsyncPattern1.2

Gegeben sind zwei PBDs *PBD1* und *PBD2* die über einen Message Link *ml* miteinander kommunizieren. Da in *PBD2* keine Nachfolgeaktivitäten auf die `<receive>`-Aktivität *r* folgen, wird hier die Zuweisung von v_s an v_r obsolet, da die Daten von v_r in *PBD2* nichtmehr verwendet werden. Stattdessen ersetzen wir *s* und *r* durch zwei synchronisierende `<empty>`-Aktivitäten *a* und *b* wie in **a)** gezeigt. Ist zusätzlich für *r* das `suppressJoinFailure`-Attribut auf „`yes`" gesetzt, so können wir *r* vollständig entfernen (**b)**). Andernfalls würden wir einen möglichen `bpel:joinFailure`-Fault unterdrücken. **c)** zeigt eine weitere Optimierung für den Fall, dass auch für *s* das `suppressJoinFailure`-Attribut auf „`yes`" gesetzt ist: Nach Anwendung des `<empty>`-Optimierungsalgorithmus aus Abschnitt 3.3.1.1.2 wurde auch die *s* ersetzende `<empty>`-Aktivität *a* entfernt und der Kontrollfluss zwischen •*s* sowie *s*• zusammengeführt.

3.3.1.3 AsyncPattern1.3

Das AsyncPattern1.3 ist ein weiterer Spezialfall des AsyncPattern1.1. Im Gegensatz zum AsyncPattern1.2 gilt hier nun: $s\bullet = \emptyset$ gilt, es existieren somit keine weiteren Nachfolgeaktivitäten auf die sendende. Abbildung 3.34 zeigt das Kommunikationsmuster an einem Beispielfragment.

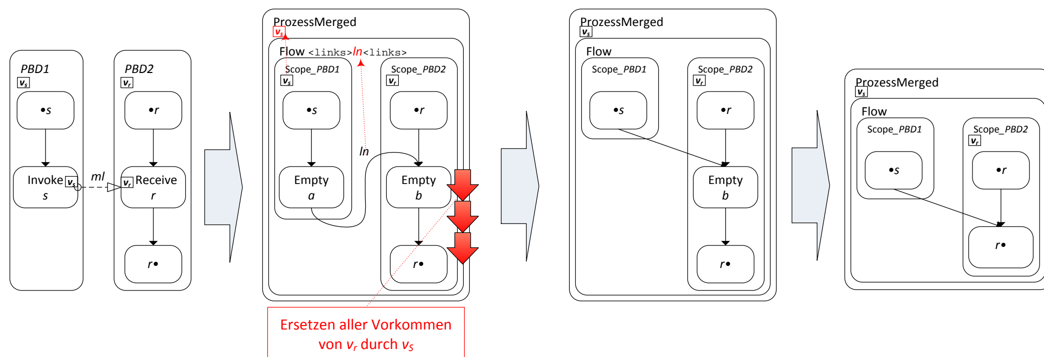


Abbildung 3.34 AsyncPattern1.3

Gegeben sind die beiden PBDs *PBD1* und *PBD2*. Da *PBD1* die Daten von v_s nicht mehr verwendet, wird bei der Konsolidierung *s* durch die synchronisierende `<empty>`-Aktivität *a* ersetzt und *r* durch die `<empty>`-Aktivität *b*. Da wir hier auf die Datenzuweisung von v_s an v_r durch die übliche `<assign>`-Aktivität, die *s* ersetzt, verzichtet haben, werden wir nun ausgehend von *b* alle Nachfolgeaktivitäten untersuchen und alle Vorkommen von v_r durch v_s ersetzen. v_s wird zusätzlich in den Prozessscope übertragen. Auflistung 3.7 zeigt den Pseudocode für das Ersetzen der Variablen. Zur Übersichtlichkeit und um den Umfang der vorliegenden Arbeit nicht zu sprengen, wurde auf Implementierungsdetails verzichtet. Diese werden im Kapitel 4 Implementierung genauer beschrieben.
Nachdem v_r durch v_s in *Scope_PBD2* ersetzt wurde, führen wir den `<empty>`-Optimierungsalgorithmus für *a* und *b* aus und können im Optimalfall (beide `suppressJoinFailure`-Attribute sind auf „`yes`" gesetzt) beide Synchronisationsaktivitäten entfernen (Abbildung 3.34 rechte Fragmente).

```
(1)  replaceVar(v_r, v_s, act)
(2)  begin
(3)    foreach ( source in act.<sources> ) do
(4)      if ( source.tc ≠ null )
(5)        replaceVariableInXPathExpr(v_r, v_s, source.tc)
(6)      fi
(7)    od
(8)    if ( typeof(act) == <invoke> )
(9)      … ersetze alle Vorkommen von v_r durch v_s in inputVariable und outputVariable, <from-
(10)     Parts>, <toParts>, <compensationHandler>, Fault Handler sowie allen Unteraktivitäten
(11)      (Rekursion) …
(12)   fi
(13)   if ( typeof(act) == <receive> )
(14)     … ersetze alle Vorkommen von v_r durch v_s in variable  oder <fromParts> …
(15)   fi
(16)   if ( typeof(act) == <reply> )
(17)     … ersetze alle Vorkommen von v_r durch v_s in variable  oder <toParts> …
(18)   fi
(19)   if ( typeof(act) == <assign> )
(20)     … ersetze alle Vorkommen von v_r durch v_s in allen <copy>s → from-specs sowie to-specs
(21)     mit variable und expressionLanguage …
(22)   fi
(23)   if ( typeof(act) == <validate> )
(24)     … ersetze alle Vorkommen von v_r durch v_s in variable  …
(25)   fi
(26)   if ( typeof(act) == <sequence>  ||  typeof(act) == <flow>  )
(27)     foreach ( sub_Act in act.activities ) do
(28)       replaceVariable(v_r, v_s, sub_Act)
(29)     od
(30)   fi
(31)   if ( typeof(act) == <if> )
(32)     … ersetze alle Vorkommen von v_r durch v_s in condition,  allen <elseif>s und <else>  sowie
(33)     allen enthaltenen Unteraktivitäten (Rekursion) …
(34)   fi
(35)   if ( typeof(act) == <while>  ||  typeof(act) == <repeatUntil>  )
(36)     replaceVariableInXPathExpr(v_r, v_s, act.<condition>)
(37)     replaceVariable(v_r, v_s, act.activity)
(38)   fi
(39)   if ( typeof(act) == <pick> )
(40)     … ersetze alle Vorkommen von v_r durch v_s in allen <onMessage> sowie <onAlarm>-Zweigen
(41)     sowie Unteraktivitäten (Rekursion) …
(42)   fi
(43)   if ( typeof(act) == <forEach> )
(44)     … ersetze alle Vorkommen von v_r durch v_s in startCounterValue-, finalCounterValue-,
(45)     completionCondition-expressions sowie der enthaltenen <scope>-Aktivität (Rekursion) …
(46)   fi
(47)   if ( typeof(act) == <scope> )
(48)     … überprüfe ob eine Variable v_r definiert wurde (diese überdeckt die ursprüngliche Variable v_r),
(49)     wenn nicht, ersetze alle Vorkommen von v_r durch v_s in allen CHs, FHs, EHs, THs sowie der
(50)     Unteraktivität (Rekursion)
(51)   fi
```

```
(52)    foreach ( succAct in act• ) do
(53)       replaceVariable(vr, vs, succAct)
(54)    od
(55) end
```

Auflistung 3.7 Pseudocode *replaceVar(v_1, v_2, act)***-Funktion**

Das AsyncPattern1.3 setzt voraus, dass es keine parallelen schreibenden oder lesenden Aktivitäten im sendenden Prozess gibt, die nach der Ausführung von *s* v_s verwenden, da v_s nach der Konsolidierung im empfangenden Prozessfragment (*Scope_PBD2*) verändert werden kann (vgl. Abbildung 3.25).

3.3.1.4 AsyncPattern1.4

Das AsyncPattern1.4 ist charakterisiert durch eine leere Menge an direkten Nachfolgeaktivitäten von *s* sowie *r*, es gilt somit $r\bullet=\emptyset$ und $s\bullet=\emptyset$. Abbildung 3.35 zeigt ein Beispielfragment zweier miteinander kommunizierender Prozessfragmente, die ein solches Muster enthalten.

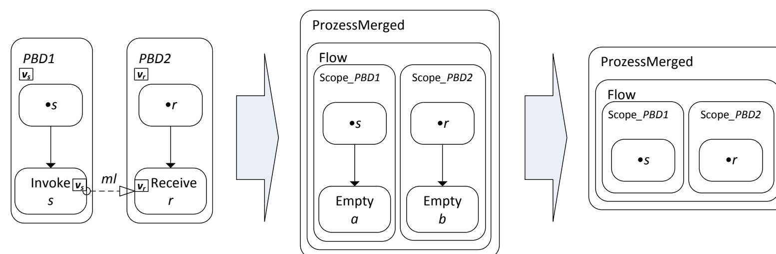


Abbildung 3.35 AsyncPattern1.4

Da beide Choreographieteilnehmerfragmente *PBD1* sowie *PBD2* die Daten nach der Kommunikation nicht mehr verwenden, entfällt das Zuweisen von v_s an v_r per `<assign>`-Aktivität. Ist zusätzlich die *Dead-Path-Elimination* (beide `suppressJoinFailure`-Attribute sind auf „`yes`" gesetzt) für *s* und *r* aktiviert, können die synchronisierenden `<empty>`-Aktivitäten *a* und *b* ebenfalls entfernt werden.

3.3.1.5 AsyncPattern1.5

Das AsyncPattern1.5 ist ein Kommunikationsmuster bei dem es keine direkten Vorgängeraktivitäten auf die empfangende `<receive>`-Aktivität gibt. Daher ist das `createInstance`-Attribut der empfangenden `<receive>`-Aktivität auf „`yes`" gesetzt und es gilt $\bullet r=\emptyset$. Abbildung 3.36 zeigt das AsyncPattern1.5 an einem Beispielfragment zweier kommunizierender PBDs.

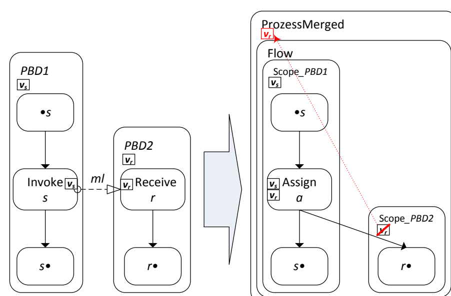


Abbildung 3.36 AsyncPattern1.5

PBD1 und *PBD2* kommunizieren über einen Message Link ml miteinander. Hierbei ist jedoch *PBD1* der Initiator von *PBD2* ($\bullet r=\emptyset$). Da die `<receive>`-Aktivität *r* keine direkten Vorgängeraktivitäten besitzt kann in diesem Fall auf eine synchronisierende `<empty>`-Aktivität verzichtet werden. *s* wird durch eine `<assign>`-Aktivität *a* ersetzt, die v_s nach v_r kopiert. Zusätzlich muss der Kontrollfluss zwischen *Scope_PBD1* mit dem von *Scope_PBD2* verbunden werden. Hierzu werden alle ausgehenden Links von *r*, inklusive möglicher `<transitionCondition>`s zu den ausgehen-

den Links von *a* hinzugefügt und in die Prozess-<flow>-Aktivität übertragen. Existieren keine ausgehenden Links, es gilt somit *succl(r)=∅*, wird ein neuer Link *ln* (nicht abgebildet) ausgehend von *a* und eingehend in die Nachfolgeraktivität *succ(r)* hinzugefügt.

3.3.1.6 AsyncPattern1.6

Das AsyncPattern1.6 ist eine weitere Spezialisierung des AsyncPattern1.1. Im Gegensatz zu diesem überprüft es zusätzlich, ob es in den direkten Nachfolgeraktivitäten der sendenden <invoke>-Aktivität *s* weitere asynchrone <invoke>-Aktivitäten gibt, die choreographie-intern kommunizieren. Hierzu untersucht es die übrigen Message Links aus *ML* und kann die involvierten <invoke>-Aktivitäten zu einer neuen <assign>-Aktivität mit mehreren atomaren <copy>-Blöcken zusammenführen und anschließend den Kontrollfluss synchronisieren. Dieses Verfahren wird sukzessiv für die direkten Nachfolgeaktivitäten durchgeführt.

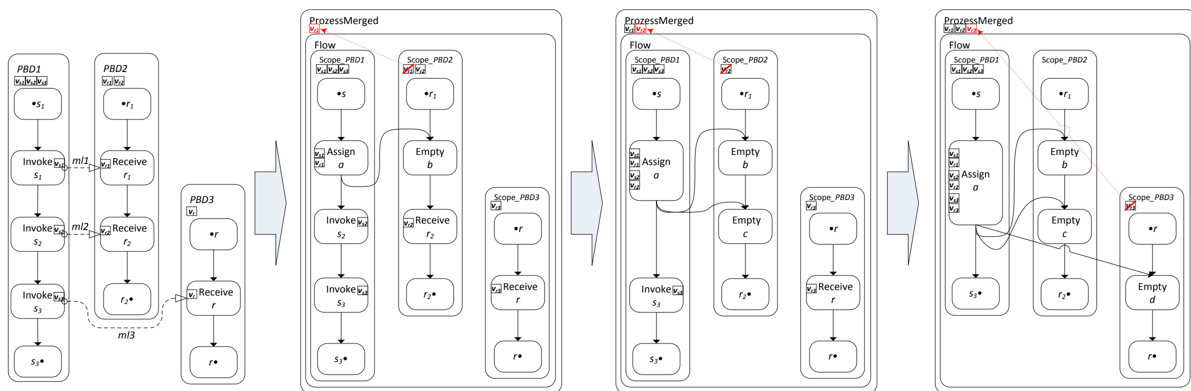


Abbildung 3.37 AsyncPattern1.6

Abbildung 3.37 zeigt die drei Beispielsfragmente einer Choreographie in denen ein solches Kommunikationsmuster auftritt. Die drei PBDs *PBD1*, *PBD2* sowie *PBD3* kommunizieren über die drei Message Links *ml1*, *ml2* und *ml3* miteinander. Nachdem die PBDs in die entsprechenden <scope>-Aktivitäten im neuen konsolidierten Prozess *ProzessMerged* kopiert wurden, untersucht der Algorithmus die Message Links. In der asynchronen Kommunikation zwischen *PBD1* und *PBD2*, dargestellt durch den Message Link *ml1* und die <invoke>-Aktivität s_1 in *PBD1* sowie die <receive>-Aktivität r_1 in *PBD2*, wird nun das AsyncPattern1.1 angewendet und anschließend die direkten Nachfolgeaktivitäten von s_1 in $s_1\bullet=succl(s_1)+succ(s_1)$ untersucht: Hierbei findet sich erneut eine asynchrone *choreographie-intern* kommunizierende <invoke>-Aktivität s_2, dargestellt durch den Message Link *ml2*. Nun werden folgende Bedingungen für s_2 überprüft:

1. Liegt s_2 in $succl(s_1)$ überprüfen wir zunächst, ob s_1 und s_2 die gleichen Werte für das suppressJoinFailure-Attribute enthalten. Sind diese verschieden, bricht der Algorithmus ab und der nächste Message Link wird untersucht andernfalls wird überprüft, ob es eine explizite Kontrollflussabhängigkeit durch eine <transitionCondition> gibt, für den entsprechenden, beide Aktivitäten verbindenden Link. Ist dies der Fall, so bricht der Algorithmus ab, da erst zur Laufzeit entschieden werden kann ob eine <transitionCondition> zu wahr ausgewertet wird (vgl. [BFG05]) und wir bei einer Zusammenführung beider Aktivitäten den Kontrollfluss verändern würden. So könnte es sein, dass vor der Zusammenführung die Auswertung der <transitionCondition> von s_1 die Ausführung von s_2 verhindert. Liegt keine <transitionCondition> für diesen Link vor wird untersucht, ob s_2 weitere eingehende Links besitzt (<targets>). Ist dies der Fall bricht der Algorithmus ab. Würden wir diese zu den eingehenden Links der aus AsyncPattern1.1 für die s_1 ersetzende <assign>-Aktivität *a* hinzufügen, so wäre der Kontrollfluss von *a* abhängig von den Vorgängeraktivitäten von s_2 aus $prel(s_2)$. Sind beide vorhergehenden

Überprüfungen negativ, so wird anschließend überprüft, ob s_1 und s_2 in derselben <scope>-Aktivität liegen, da wir ansonsten durch eine Zusammenführung den Kontrollfluss verändern würden. Dies wäre beispielsweise dann der Fall, wenn s_1 in einem <scope> mit einem Compensation Handler liegt, der nach der Zusammenführung erst nach Beendigung von s_2 installiert werden würde.

2. Liegt s_2 in *succ*(s_1) überprüfen wir, ob s_1 und s_2 die gleichen Werte für das suppressJoinFailure-Attribute enthalten. Anschließend wird geprüft, ob s_2 eingehende Links besitzt (<targets>). Ist dies der Fall bricht der Algorithmus ab, da wir durch Zusammenführung beider Aktivitäten einen veränderten Kontrollfluss zur Folge hätten. Nun wird überprüft ob s_1 und s_2 in derselben <scope>-Aktivität liegen.

Wurden die vorhergehenden Überprüfungen bestanden so werden s_1 und s_2 in die <assign>-Aktivität *a* zusammengeführt, indem eine neuer <copy>-Block für das Kopieren der Variable v_{s2} nach v_{r2} zu dem schon vorhandenen <copy>-Block hinzugefügt wird. Abbildung 3.38 zeigt einen solchen <copy>-Block.

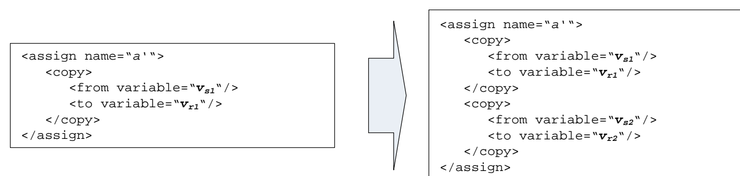


Abbildung 3.38 Erweiterung von <assign>-Aktivität *a* durch zusätzlichen <copy>-Block

Nachdem s_1 und s_2 in *a* zusammengeführt wurden, werden die <sources> von s_2, falls vorhanden, inklusive <transitionCondition>s zu *a* hinzugefügt und der Algorithmus wiederholt die Überprüfung in den direkten Nachfolgeraktivitäten s_2• (Abbildung 3.37 dritte Iteration). Im Optimalfall können die <empty>-Aktivitäten *b*, *c* und *d* nach Anwendung des <empty>-Optimierers auch hier entfernt werden (nicht dargestellt).

3.3.1.7 AsyncPattern1.7 („Khalaf Split")

In ihrer Dissertation [KHA08] beschreibt Khalaf eine Methode zum Aufspalten eines Kontrollflusslinks über Prozessgrenzen hinweg. Das AsyncPattern1.7 erkennt solche Kommunikationsmuster und stellt den ursprünglichen Kontrollfluss im konsolidierten Prozess wieder her. Abbildung 3.39 zeigt die Idee hinter dem Fragmentierungsvorgang von Khalaf.

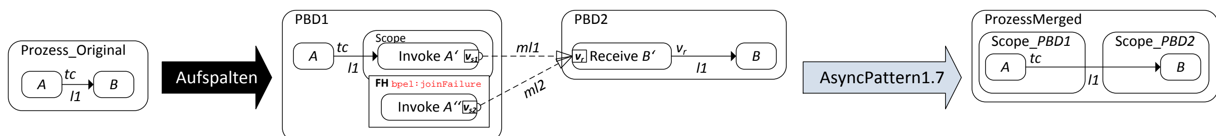


Abbildung 3.39 Aufspalten eines Kontrollflusslinks (vgl. [KHA08]) und anschließendes Konsolidieren mit AsyncPattern1.7

Im unfragmentierten Prozess *Prozess_Original* sind die beiden Aktivitäten *A* und *B* über einen Link miteinander verbunden. Für diesen Link gilt die <transitionCondition> *tc(l1, A)=tc*. Der Kontrollflusslink wird nach Khalaf folgendermaßen aufgespalten: Um den Status der <transitionCondition> *tc* aus *PBD1* nach *PBD2* zu übertragen wird der Kontrollfluss in einen Nachrichtenfluss transformiert. Hierzu wird eine <scope>-Aktivität mit einem Fault Handler zum Auffangen des bpel:joinFailure-Faults hinzugefügt. In dieser <scope>-Aktivität befindet sich eine <invoke>-Aktivität *A'*, die das suppressJoinFailure-Attribute auf „no" gesetzt hat und die Variable v_{s1} mit dem Standardwert „true" verwendet. Zusätzlich enthält der Fault Handler eine <invoke>-Aktivität *A''*, die Variable v_{s2} mit dem Standardwert „false" verwendet. Beide <invoke>-Aktivitäten kom-

munizieren mit derselben `<receive>`-Aktivität *B‘* in *PBD2* über die beiden Message Links *ml1* sowie *ml2*. Je nach Auswertung der `<transitionCondition>` *tc* zur Laufzeit, sendet entweder *A‘* oder im `bpel:joinFailure`-Fault Fall *A‘‘*. Die Empfängerseite *PBD2* erhält den Wert des ehemaligen Kontrollflusslinks in der Variablen v_r und wertet diese in der `<transitionCondition>` *tc(l2, B‘)= “*v_r*=true()“* aus. Da der Link zwischen *B‘* und *B* im fragmentierten Prozess *PBD2* weiterhin l1 heißt, muss die `<joinCondition>` von *B* nicht angepasst werden.
In seiner Diplomarbeit [CUI12] setzt Cui diesen Fragmentierungsvorgang um. Wir werden AsyncPattern1.7 so konstruieren, dass dieses das dort gezeigte Fragmentierungsmuster erkennt und den ursprünglichen Kontrollfluss wiederherstellt.

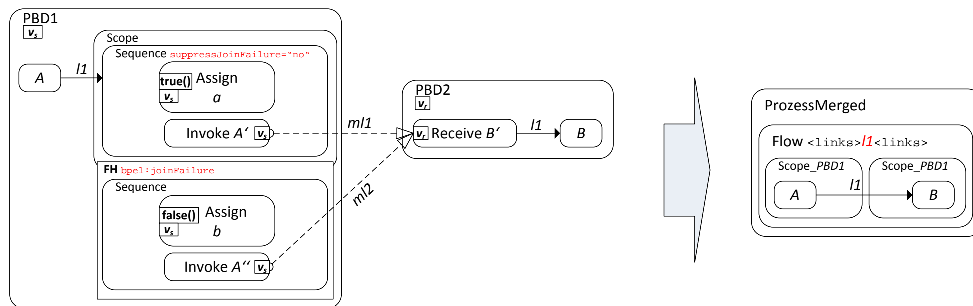
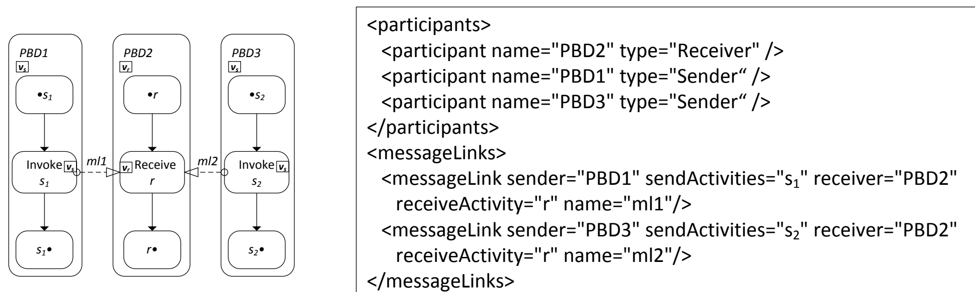


Abbildung 3.40 AsyncPattern1.7 angewendet auf Cui’s Kontrollflusslinkfragmentierung (vgl. [CUI12])

Abbildung 3.40 zeigt die Umsetzung der Kontrollflusslinkfragmentierung nach Cui mit anschließender Konsolidierung mit AsyncPattern1.7. Für das AsyncPattern1.7 sucht der Konsolidierungsalgorithmus nach zwei Message Links aus *ML ml1* und *ml2*, die beide die gleiche `<receive>`-Aktivität, beispielswese *B‘*, als `receiveActivity` enthalten. Zusätzlich befinden sich die beiden `<invoke>`-Aktivitäten, z.B. *A‘* und *A‘‘*, in derselben *PBD*. Jetzt wird überprüft, ob sich jeweils *A‘* sowie *A‘‘* als zweite Aktivität innerhalb einer `<sequence>`-Aktivität befinden (*typeof(par(A‘)=`<sequence>`* sowie *typeof(par(A‘‘))=`<sequence>`*). Trifft dies zu wird überprüft, ob die Vorgängeraktivität beider `<invoke>`s eine `<assign>`-Aktivität ist. Diese `<assign>`-Aktivität weist in einem Fall den festen Wert „`true()`“, sei diese *a*, im anderen Fall „`false()`“, sei diese *b*, an die von der `<invoke>`-Aktivität verwendete Variable v_s zu. Anschließend wird überprüft, ob die `<sequence>`-Aktivität, die *a* enthält innerhalb einer `<scope>`-Aktivität (*typeof(par(par(A‘)))=`<scope>`*) mit einem Fault Handler für den `bpel:joinFailure`-Fault liegt und das `suppressJoinFailure`-Attribut auf „no“ gesetzt hat. Zusätzlich darf diese Aktivität nur einen eingehenden Link haben, sei dieser hier *l1*. Die `<sequence>`-Aktivität, die *b* enthält muss in diesem Fault Handler enthalten sein (*typeof(par(par(A‘‘)))=`<catch faultName="bpel:joinFailure">`*). Beide Sequenzen dürfen nur die zuvor erwähnten `<assign>`- und `<invoke>`-Aktivitäten enthalten. Wurden alle vorherigen Muster erkannt, wird die Empfängerseite überprüft: Die `<receive>`-Aktivität, sei diese hier *B‘*, enthält nur einen ausgehenden Link, der denselben Namen, wie der in die `<sequence>`-Aktivität von *A‘* eingehende besitzt (*l1*). Zusätzlich prüft die `<transitionCondition>` dieses Links den Wert der empfangenen Variable v_r auf „`true`“ (*tc(l1, B‘)=“*v_r*=true()“*).
Wurden alle zuvor genannten Bedingungen im sendenden und empfangenden Fragment positiv überprüft, so wird das Konsolidierungsmuster angewendet und der Kontrollfluss der beiden PBDs im neuen Prozess *ProzessMerged* zusammengeführt. Hierzu wird die `<scope>`-Aktivität in *Scope_PBD1* entfernt sowie die `<receive>`-Aktivität in *Scope_PBD2*. Anschließend wird der Link *l1*, der zuvor in *Scope_PBD1* sowie *Scope_PBD2* enthalten war in die Prozess-`<flow>`-Aktivität übertragen. Da auf Sender sowie Empfängerseite derselbe Linkname *l1* während der Fragmentierung verwendet wurde, müssen bei der Konsolidierung keine Anpassungen an der `<joinCondition>` von *B* durchgeführt werden.
Das von Cui in [CUI12] implementierte Fragmentierungsmuster für Kontrollflusslinks lässt sich auch ohne eine `<sequence>`-Aktivität innerhalb der `<scope>`-Aktivität auf der sendenden Seite realisieren, beispielsweise durch zwei Variablen, die nur einmal zu Beginn des Prozesses mit „`true`“ und „`false`“ initialisiert werden (v_{true} und v_{false}). Der gezeigte Erkennungs- und Konsolidierungsalgorithmus ist hierfür leicht anpassbar.

3.3.1.8 AsyncPattern1.8 (Asynchrones n-zu-1 Senden auf `<receive>`)

Das AsyncPattern1.8 dient dem Aufspüren und anschließendem Konsolidieren einer n-zu-1 Kommunikation innerhalb einer BPEL4Chor Choreographie. Hierbei können mehrere Choreographieteilnehmer Nachrichten zu einem anderen einzelnen Teilnehmer senden. Dieses Muster wird durch mehrere verschiedene Message Links aus *ML* mit jeweils derselben `receiveActivity` repräsentiert, wobei im Gegensatz zu AsyncPattern1.7 die `sendActivity` in verschiedenen PBDs enthalten ist. Abbildung 3.41 zeigt ein solches Beispiel und die zugehörigen Message Links in der Topology.



```
<participants>
  <participant name="PBD2" type="Receiver" />
  <participant name="PBD1" type="Sender“ />
  <participant name="PBD3" type="Sender“ />
</participants>
<messageLinks>
  <messageLink sender="PBD1" sendActivities="s1" receiver="PBD2"
    receiveActivity="r" name="ml1"/>
  <messageLink sender="PBD3" sendActivities="s2" receiver="PBD2"
    receiveActivity="r" name="ml2"/>
</messageLinks>
```

Abbildung 3.41 n-zu-1 Senden sowie das zugehörige Topology-Fragment

Um ein korrektes Verbinden des Kontrollflusses zwischen den sendenden und den empfangenden Choreographieteilnehmern zu gewährleisten, müssen wir unseren Ansatz aus AsyncPattern1.1 etwas verändern. Wenn wir in diesem Beispiel das AsyncPattern1.1 auf die drei Teilnehmer und die zwei Message Links anwenden kann es passieren, dass Informationen überschrieben werden.

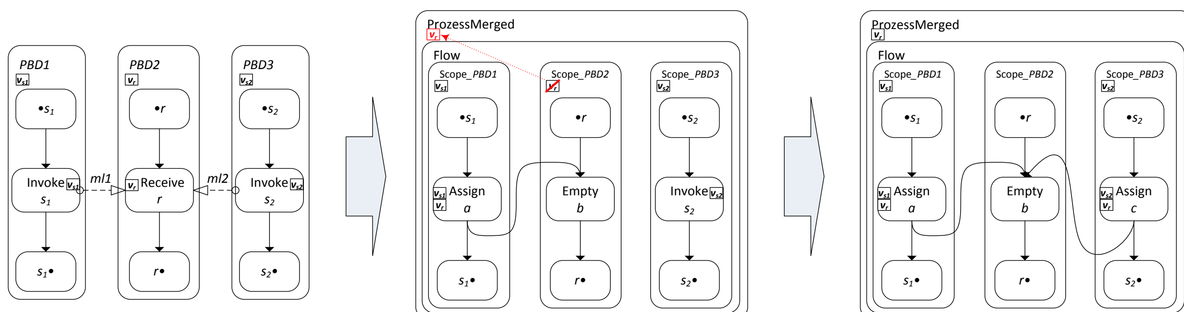


Abbildung 3.42 Anwendung des AsyncPattern1.1 auf mehrere Message Links mit gleicher `<receive>`-Aktivität

Abbildung 3.42 zeigt die Beispielchoreographie mit den drei PBDs *PBD1*, *PBD2* und *PBD3*. *PBD1* und *PBD3* senden jeweils beide auf die gleiche `<receive>`-Aktivität *r* in *PBD2*. Nach sukzessiver Anwendung des AsyncPattern1.1 wird zunächst s_1 in *PBD1* durch eine `<assign>`-Aktivität *a* ersetzt, die v_{s1} nach v_r kopiert, sowie *r* durch eine synchronisierende `<empty>`-Aktivität *b*. Anschließend wird s_2 in *PBD3* durch eine `<assign>`-Aktivität *c* ersetzt, die v_{s2} nach v_r kopiert. Da sich nun v_r im globalen Prozessscope befindet, kann es passieren, dass zunächst v_{s1} nach v_r kopiert wird, falls *Scope_PBD1* im Ablauf schneller voranschreitet und anschließend nochmals *Scope_PBD2* v_{s2} nach v_r kopiert und die Daten von v_{s1} überschreibt. Zusätzlich kommt das Problem des veränderten Kontrollflusses hinzu: In der ursprünglichen Choreographie sendet beispielsweise s_1 aus *PBD1* eine Nachricht an *r* in *PBD2* und nach Erhalt dieser läuft *PBD2* weiter. In der konsolidierten muss jedoch `<empty>` *b* auch nachdem v_{s1} nach v_r kopiert wurde auf das Aktivieren den Kontrollflusslinks aus *c* in *Scope_PBD3* warten, wodurch eine andere Synchronisierung der Abläufe stattfindet als in der Choreographie.
Eine Lösung für das erste Problem des Überschreibens der Daten in v_r wäre das Einführen einer weiteren temporären Variable v_m zum emulieren des Datenflusses wie in Abschnitt 3.2.3.1 beschrieben. Je nach unterliegender Struktur der Daten von v_s und v_r kann dieser zusätzliche Kopiervorgang jedoch auf Kosten der Performance gehen. Stattdessen werden wir einen Ansatz wählen, der zwar eine

zusätzliche Variable $v_{r\text{-}written}$ verwendet, die jedoch lediglich als eine Art Schutzvariable (*Guard Variable*) dient, um zu signalisieren, ob v_r bereits durch eine der beiden `<assign>`-Aktivitäten *a* oder *c* beschrieben wurde und als Datentyp `xsd:boolean` verwendet sowie bei der Deklaration *inline* mit `false` initialisiert wird (vgl. WS-BPEL 2.0 Spezifikation [OAS07] Abschnitt 8.1 *Variables*). Diese Variable wird in einer `<if>`-Aktivität verwendet, die das ursprüngliche s_1 ersetzende `<assign>` *a* umgibt.

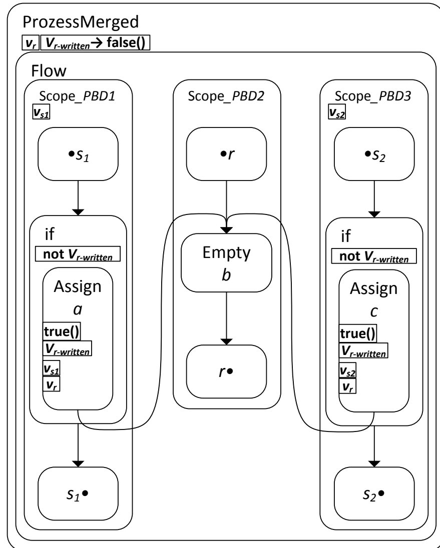


Abbildung 3.43 Asynchrones n-zu-1 Senden

Abbildung 3.43 zeigt die Anwendung der `<if>`-Aktivität in Verbindung mit der neuen Schutzvariable $v_{r\text{-}written}$: Die `<assign>`-Aktivitäten *a* und *c* wurden durch einen neuen `<copy>`-Block erweitert, der noch vor dem Kopieren der Variable v_{s1} bzw. v_{s2} nach v_r den Wert der Schutzvariable $v_{r\text{-}written}$ auf `true` setzt. Die `<assign>`-Aktivitäten sind nun beide von einer `<if>`-Aktivität umgeben, die jeweils nur dann die enthaltene Aktivität ausführt, wenn $v_{r\text{-}written}$ noch nicht beschrieben wurde ($v_{r\text{-}written}$=`false`). Andernfalls werden *a* und *c* nicht aktiviert und der Kontrollfluss schreitet in *Scope_PBD1* sowie *Scope_PBD2* weiter voran. Der Kontrollflusslink zwischen *a* und *b* sowie *c* und *b* wird auch bei negativer Auswertung der `<if>-condition` und anschließender Auslassung von *a* bzw. *c* zu `false` evaluiert. Der hier dargestellte Ansatz löst zwar das Problem des Überschreibens der Daten von v_r, doch leider bleibt auch hier der Ablauf von *Scope_PBD2* von den beiden anderen *Scope_PBD1* sowie *Scope_PBD3* abhängig, da *b* weiterhin erst ausgeführt wird wenn beide Kontrollflusslinks aus *a* als auch *c* aktiviert wurden. Wir werden nun die `<receive>` *r* ersetzende `<empty>`-Aktivität *b* so anpassen, dass diese bereits nach Aktivierung nur einer der beiden Kontrollflusslinks ausgeführt wird. Hierzu ersetzen wir *b* durch eine `<scope>`-Aktivität, die einen Fault Handler für den `bpel:joinFailure`-Fault enthält mit einer leeren `<empty>`-Aktivität. Zusätzlich enthält der `<scope>` für jeden der beiden Kontrollflusslinks von *a* und *c*, seien diese *l1* und *l2*, eine `<empty>`-Aktivität deren `suppressJoinFailure`-Attribut auf „no" gesetzt ist. Die `<joinCondition>`s der beiden `<empty>`-Aktivitäten sind einmal „`not $l1`" sowie entsprechend für *l2* „`not $l2`". Wird nun einer der beiden Links aktiviert, so wirft die entsprechende `<empty>`-Aktivität einen `bpel:joinFailure`-Fault der im Fault Handler des `<scope>` aufgefangen wird. Da dieser nur eine `<empty>`-Aktivität enthält, wird *Scope_PBD2* direkt nach diesem weiter ausgeführt. Wird nun auch der andere Link aktiviert, so hat dieser keine Auswirkung mehr auf den Kontrollfluss von *Scope_PBD2*, da der `<scope>`, der die beiden `<empty>`-Aktivitäten enthält, bereits einen Fault geworfen hat. Abbildung 3.44 zeigt das Ergebnis dieses Vorgehens an der Beispielchoreographie.

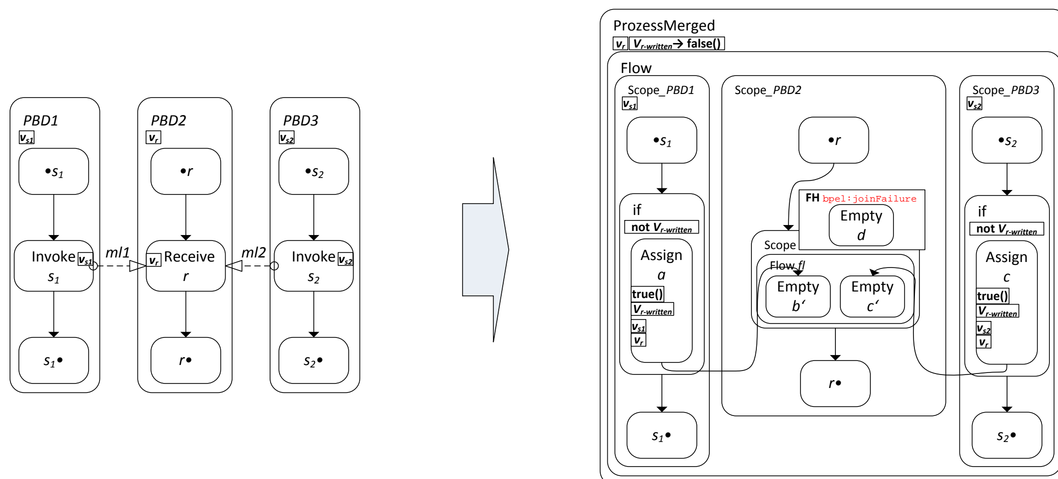


Abbildung 3.44 Ergebnis der Konsolidierung des n-zu-1 AsyncPattern1.8

Sei *sendActs* die Menge aller zur `<receive>`-Aktivität *r* sendenden Aktivitäten (im vorherigen Beispiel gilt somit *sendActs* = { $PBD1 \rightarrow s_1$, $PBD3 \rightarrow s_2$ }). Diese Menge wird ermittelt indem über alle noch nicht behandelten Message Links aus *ML* iteriert wird und diejenigen gesucht werden, die auf dieselbe `<receive>`-Aktivität *r* aus jeweils verschiedenen PBDs senden (`Participants` der Topology). Die `Participants` müssen dabei nicht zwangsweise vom selben `ParticipantType` sein. Anschließend wird die empfangende Aktivität *r* durch die zuvor erwähnte `<scope>`-Aktivität, sei diese hier *b*, ersetzt. *b* enthält eine `<flow>`-Aktivität *fl*. Besitzt *r* eingehende und ausgehende Links, so werden diese in *b* übertragen. Hierbei werden die eingehenden Links von *r* (`<targets>`), inklusive der `<joinCondition>`, zu den eingehenden Links der neuen `<scope>`-Aktivität *b*. Die ausgehenden Links (`<sources>`) von *r*, inklusive `<transitionCondition>`s, werden zu den ausgehenden Links von *b*. Zusätzlich wird für jede Aktivität in *sendActs* eine neue synchronisierende `<empty>`-Aktiviät in *fl* hinzugefügt (mit `suppressJoinFailure="no"`). Nun wird jede Aktivität aus *sendActs* durch eine `<if>`-Aktivität $if_{s\text{-}r}$ ersetzt, wie im Beispiel aus Abbildung 3.44, und eine Schutzvariable $v_{r\text{-}written}$ im Prozessscope des konsolidierten Prozesses eingeführt, die per *inline*-Deklaration mit `false` initialisiert wird. Die eingehenden Links der ehemaligen *sendActs*-Aktivität werden zu den eingehenden Links der neuen `<if>`-Aktivität, inklusive `<joinCondition>`. Selbiges gilt für die ausgehenden Links und deren `<transitionCondition>`s. Die in $if_{s\text{-}r}$ enthaltene neue `<assign>`-Aktivität (mit den `<copy>`-Blöcken zum Kopieren von „`true()`“ nach $v_{r\text{-}written}$ sowie v_s nach v_r) wird mit der entsprechenden `<empty>`-Aktivität in *b* mit einem Kontrollflusslink l_{new} verbunden. Anschließend wird die `<joinCondition>` dieser `<empty>`-Aktivität auf „`not` $\$l_{new}$“ gesetzt (*jc(b‘)=“not* $\$l_{new1}$“ sowie *jc(c‘)=“not* $\$l_{new2}$“).

3.3.1.9 AsyncPattern2.1

Neben der `<receive>`-Aktivität bietet WS-BPEL 2.0 auch die `<pick>`-Aktivität zum Empfangen von Nachrichten an. Diese muss mindestens einen `<onMessage>`-Zweig zum Empfangen enthalten und kann optionale zeitbasierte `<onAlarm>`-Zweige definieren. Ist zusätzlich das `createInstance`-Attribut auf „`yes`“ gesetzt, dürfen nur `<onMessage>`-Zweige enthalten sein. Wurde eine der in den `<onMessage>`-Zweigen enthaltenen Aktivitäten durch Eingang einer entsprechenden Nachricht aktiviert bzw. einer der zeitbasierten `<onAlarm>`-Zweig durch Eintreten eines zeitlichen Ereignisses, so müssen die verbliebenen Zweige deaktiviert werden. Das AsyncPattern2.1 dient dem Erkennen und Konsolidieren von choreographie-internen Kommunikationen mit einem `<onMessage>`-Zweig einer `<pick>`-Aktivität als empfangende Aktivität. Da die `<onMessage>`-Zweige kein `name`-Attribut besitzen wurde in BPEL4Chor der `wsu:id`-Bezeichner eingeführt, der zum Identifizieren eines solchen Zweiges als `receiveActivity` in einem Message Link aus *ML* dient.

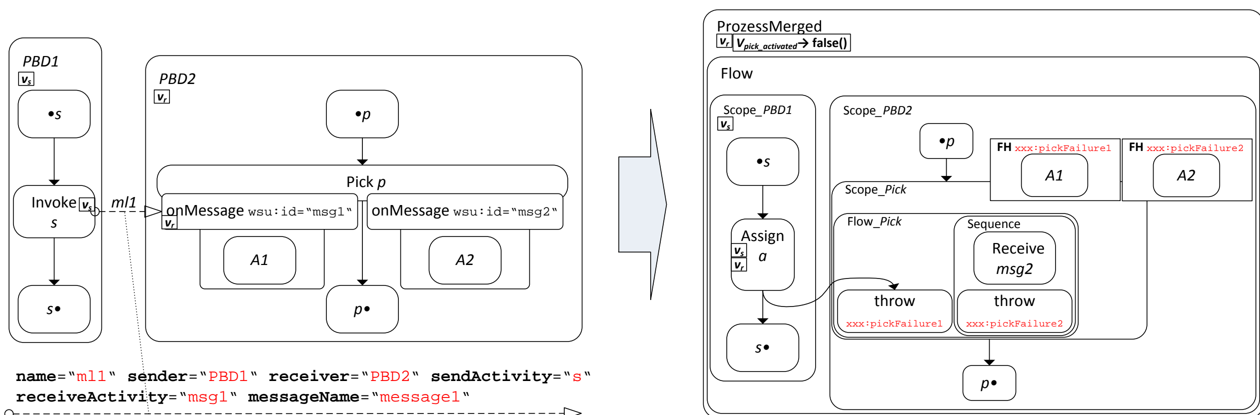


Abbildung 3.45 AsyncPattern2.1 mit einem `<onMessage>`-Zweig zum Empfang einer Nachricht eines choreographie-internen Senders

Abbildung 3.45 zeigt ein Beispielfragment einer Choreographie mit zwei kommunizierenden PBDs. *PBD1* sendet eine Nachricht an den `<onMessage>`-Zweig *msg1* der `<pick>`-Aktivität *p* in *PBD2*. Sobald diese Nachricht empfangen wurde, schreitet *PBD2* mit der Ausführung der Aktivitäten in *p•= succ(p)+succl(p)* voran. Wir werden nun den Konsolidierungsvorgang genauer beschreiben, der zu dem Ergebnis auf der rechten Seite der Abbildung 3.45 führt.
Zunächst findet der Erkennungsalgorithmus einen Message Link *ml1* aus *ML* der einen `<onMessage>`-Zweig einer `<pick>`-Aktivität als `receiveActivity` enthält (in unserem Fall *msg1*). Nachdem die beiden PBDs in ihre neuen `<scope>`-Aktivitäten im neuen konsoldierten Prozess *ProzessMerged* kopiert wurden, wird die sendende Aktivität *s* durch eine `<assign>`-Aktivität *a* ersetzt, die v_s nach v_r kopiert. v_r wurde zuvor aus *Scope_PBD2* in den Prozessscope übertragen, damit *Scope_PBD1* Zugriff auf diese hat. Nun wird die `<pick>`-Aktivität *p* durch eine neue `<scope>`-Aktivität *Scope_Pick* ersetzt. Hierbei werden die eingehenden Links und ihre `<joinCondition>` von *p* zu den eingehenden Links von *Scope_Pick* und entsprechend die ausgehenden Links und ihre möglichen expliziten `<transitionCondition>`s. *Scope_Pick* wird folgendermaßen aufgebaut: Eine neue `<flow>`-Aktivität *Flow_Pick* wird hinzugefügt. Zusätzlich wird eine `<throw>`-Aktivität eingefügt, die einen neuen benutzerdefinierten Fault wirft, in diesem Fall nennen wir ihn `xxx:PickFailure1`. Dieser ist über einen Kontrollflusslink mit der `<assign>`-Aktivität *a* verbunden. Anschließend fügen wir einen Fault Handler in die *Scope_Pick*-Aktivität hinzu, der den zuvor definierten `xxx:PickFailure1`-Fault auffängt. Dieser Fault Handler enthält die Aktivität *A1*, die zuvor mit *msg1* assoziiert war. Damit sichergestellt wird, dass der `<scope>` *Scope_Pick* auch beim Empfang einer choreographie-externen Nachricht, die zuvor den `<onMessage>`-Zweig *msg2* von *p* aktivieren konnte, deaktiviert wird, ersetzen wir *p* und *msg2* durch eine neue `<receive>`-Aktivität *msg2*, die sich in einer `<sequence>`-Aktivitäten vor einer weiteren `<throw>`-Aktivität befindet. Diese `<throw>`-Aktivität wirft einen weiteren benutzerdefinierten Fault `xxx:PickFailure2`, welcher über einen Fault Handler von *Scope_Pick* aufgefangen wird. Dieser Fault Handler enthält wiederum die Aktivität *A2*, die zuvor im `<onMessage>`-Zweig *msg2* enthalten war. Wir emulieren die `<pick>`-Aktivität durch eine `<scope>`-Aktivität, die entsprechend der ersten empfangenen Nachricht den passenden Fault Handler mit den zuvor in den `<onMessage>`-Zweigen enthaltenen Aktivitäten ausführt. Diese Ersetzung von `<pick>`-Aktivitäten lässt sich auf weitere enthaltene `<onMessage>`-Zweige erweitern, indem weitere benutzerdefinierte Faults innerhalb der neuen `<scope>`-Aktivitäten hinzugefügt werden. Zur Laufzeit werden zunächst die Aktivitäten aus *•p* ausgeführt und sobald der Kontrollfluss *Scope_Pick* aktiviert, wird entweder die `<assign>`-Aktivität *a* ausgeführt, die daraufhin den *Scope_Pick* durch das Werfen des `xxx:PickFailure1` deaktiviert und den entsprechenden Fault Handler mit der enthaltenen Aktivität *A1* ausführt oder eine eingehende Nachricht wird über die `<receive>`-Aktivität *msg2* empfangen, die entsprechendes mit dem hierfür vorgesehenen Fault Handler `xxx:PickFailure2` durchführt. Kommt es hier zu einer Wettlaufsituation zwischen *a* und *msg2* ist die Ausführung implementierungsabhängig, wie auch schon der WS-BPEL 2.0 Standard für die ursprüngliche `<pick>`-Aktivität definiert (vgl. [OAS07] Abschnitt 11.5).
Die vorher beschriebene Konsolidierungsvariante hat jedoch noch ein ähnliches Problem, wie in AsyncPattern1.8 geschildert: Wir deaktivieren zwar den Kontrollfluss in *Scope_Pick* nach Aktivierung des entsprechenden Kontrollflusslinks aus `<assign>` *a* oder durch den Empfang einer Nachricht in *msg2*, es kann jedoch vorkommen, dass *msg2* eine Nachricht empfängt den *Scope_Pick* deaktiviert *a* jedoch weiterhin v_s nach v_r kopiert und dies möglicherweise einen nicht gewünschten Nebeneffekt auf *Scope_PBD2* hat (*Lost Update Problem*). Um dieses Problem zu verhindern werden wir ähnlich der Lösung für AsyncPattern1.8 eine Schutzvariable $v_{pick_activated}$ in den Prozessscope einführen, die per *inline*-Deklaration mit `false` initialisiert wird. Zusätzlich wird *a* durch eine Kombination aus einer `<if>`-Aktivität sowie einer in dieser enthaltenen `<assign>`-Aktivität ersetzt. Um sicherzustellen, dass die `<if>`-Aktivität nur dann ausgeführt wird, wenn $v_{pick_activated}$ auf `false` gesetzt ist, fügen wir weiterhin vor die `<throw>`-Aktivität für den benutzerdefinierten Fault `xxx:PickFailure2` eine weitere `<assign>`-Aktivität hinzu, die $v_{pick_activated}$ auf `true` setzt, sobald *msg2* eine Nachricht empfängt. Dieses Vorgehen lässt sich auf weitere choreographie-extern kommunizierende `<onMessage>`-Zweige aus *p* erweitern, indem die entsprechenden `<sequence>`-Aktivitäten in *Scope_Pick* angepasst werden. Abbildung 3.46 zeigt die angepasste Konsolidierung der Beispielfragmente aus Abbildung 3.45.

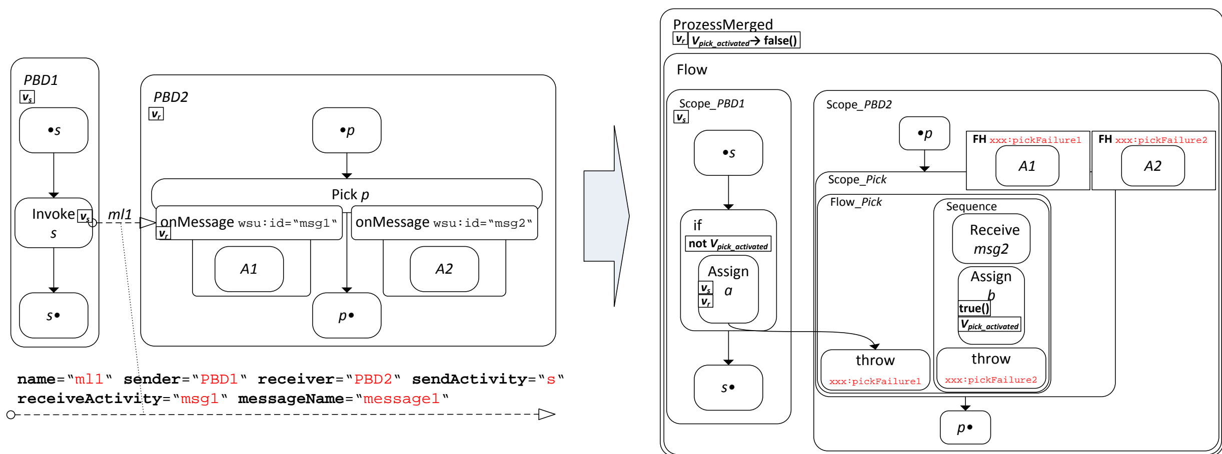


Abbildung 3.46 AsyncPattern2.1 mit Schutzvariable $v_{pick_activated}$ zur Vermeidung des *Lost Update Problems*

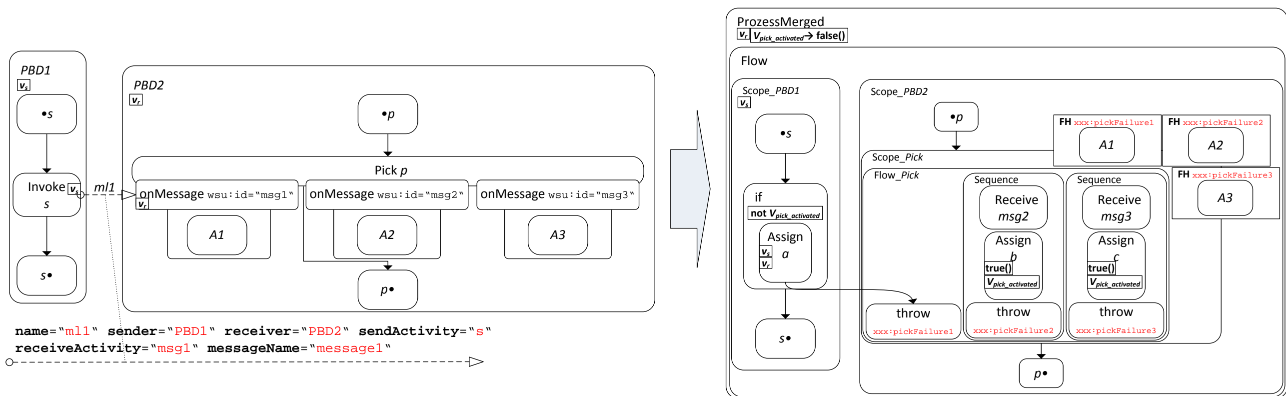
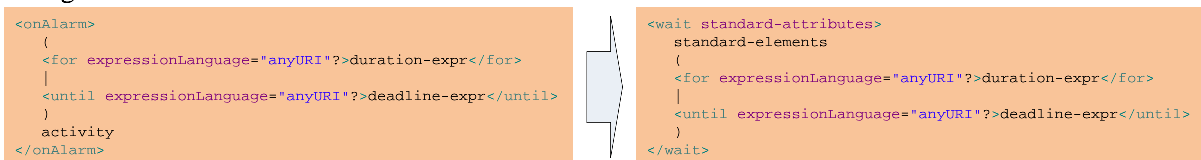


Abbildung 3.47 AsyncPattern2.1 mit einer `<pick>`-Aktivität mit weiteren `<onMessage>`-Zweigen (hier *msg3*)

Abbildung 3.47 zeigt die Anwendung des AsyncPattern2.1 auf eine `<pick>`-Aktivität mit einem weiteren `<onMessage>`-Zweig *msg3*: Eine weitere `<receive>`-Aktivität wird hinzugefügt, die *msg3* ersetzt und nach der Zuweisung von `true` an die Schutzvariable $v_{pick_activated}$ einen benutzerdefinierten Fault `xxx:PickFailure3` wirft, der im dafür vorgesehenen Handler von *Scope_Pick* aufgefangen wird und die ursprüngliche Aktivität *A3* ausführt.

Zu den schon gezeigten `<onMessage>`-Zweigen bietet die `<pick>`-Aktivität `<onAlarm>`-Zweige, die nach Ablauf einer gegebenen Zeitperiode oder Erreichen eines entsprechenden Zeitpunkts die enthaltene Aktivität ausführen. Da wir in unserem Konsolidierungsvorgang die `<pick>`-Aktivität durch eine `<scope>`-Aktivität mit entsprechendem emulierendem Verhalten nachbauen, müssen wir im Falle eines definierten `<onAlarm>`-Zweigs auch diesen in den `<scope>` integrieren. Hierzu werden wir zu den vorhandenen `<receive>`/`<throw>`-Kombinationen innerhalb des *Flow_Pick* eine `<wait>`-Aktivität mit anschließender `<throw>`-Aktivität einfügen und erneut einen benutzerdefinierten Fault Handler zum *Scope_Pick* hinzufügen, der die Aktivität des ursprünglichen `<onAlarm>`-Zweigs enthält.


```
<onAlarm>
   (
   <for expressionLanguage="anyURI"?>duration-expr</for>
   |
   <until expressionLanguage="anyURI"?>deadline-expr</until>
   )
   activity
</onAlarm>
```

```
<wait standard-attributes>
   standard-elements
   (
   <for expressionLanguage="anyURI"?>duration-expr</for>
   |
   <until expressionLanguage="anyURI"?>deadline-expr</until>
   )
</wait>
```

Abbildung 3.48 Syntaktische Umwandlung eines `<onAlarm>`-Zweigs in eine `<wait>`-Aktivität (vgl. [OAS07])

Abbildung 3.48 zeigt die syntaktische Umwandlung eines <onAlarm>-Zweigs in eine entsprechende <wait>-Aktivität. Die enthaltene Aktivität des <onAlarm>-Zweigs wird bei der Konsolidierung in den entsprechenden Fault Handler von *Scope_Pick* übertragen.

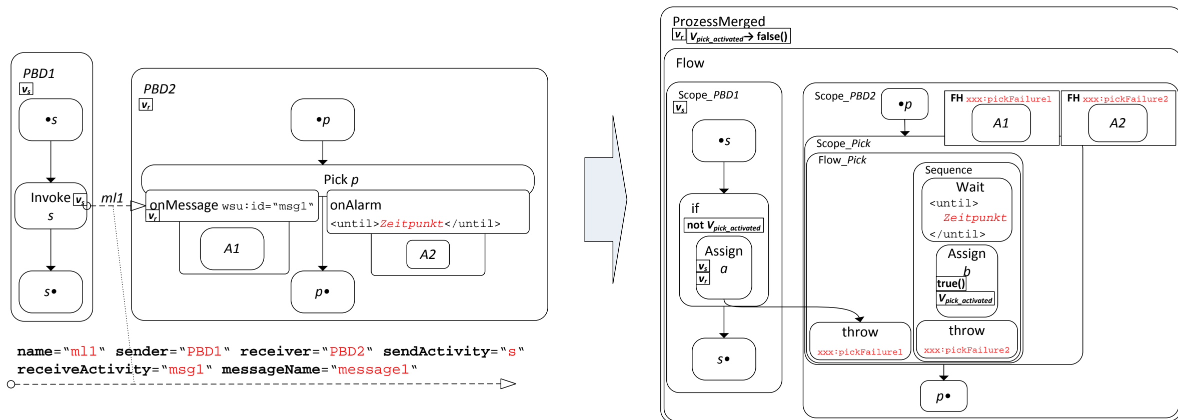


Abbildung 3.49 AsyncPattern2.1 für <pick>-Aktivität mit <onAlarm>-Zweig

Abbildung 3.49 zeigt die Anwendung des AsyncPattern2.1 auf eine Beispielchoreographie mit einem <onAlarm>-Zweig im <pick> *p* der Empfängerseite *PBD2*. *A2* wurde in den Fault Handler für den benutzerdefinierten xxx:PickFailure2-Fault von *Scope_Pick* übertragen. Die <until>-Bedingung ist nun in der neuen <wait>-Aktivität enthalten. Die gleiche Umwandlung funktioniert auch für <for>-Ausdrücke innerhalb des <onAlarm>-Zweigs.

3.3.1.10 AsyncPattern2.2 (Asynchrones n-zu-1 Senden auf <pick>)

Das AsyncPattern2.2 ist eine Spezialisierung des AsyncPattern2.1 mit der Besonderheit, dass hier mehrere Choreographieteilnehmer auf dieselbe <pick>-Aktivität, jedoch verschiedene <onMessage>-Zweige dieser, eines anderen Choreographieteilnehmers asynchron Nachrichten senden. Hierzu untersucht der Mustererkennungsalgorithmus beim Fund eines <onMessage>-Zweigs als receiveActivity in einem Message Link zusätzlich die anderen noch nicht konsolidierten Message Links aus *ML* auf eine receiveActivity mit der wsu:id eines der anderen <onMessage>-Zweige der entsprechenden <pick>-Aktivität.

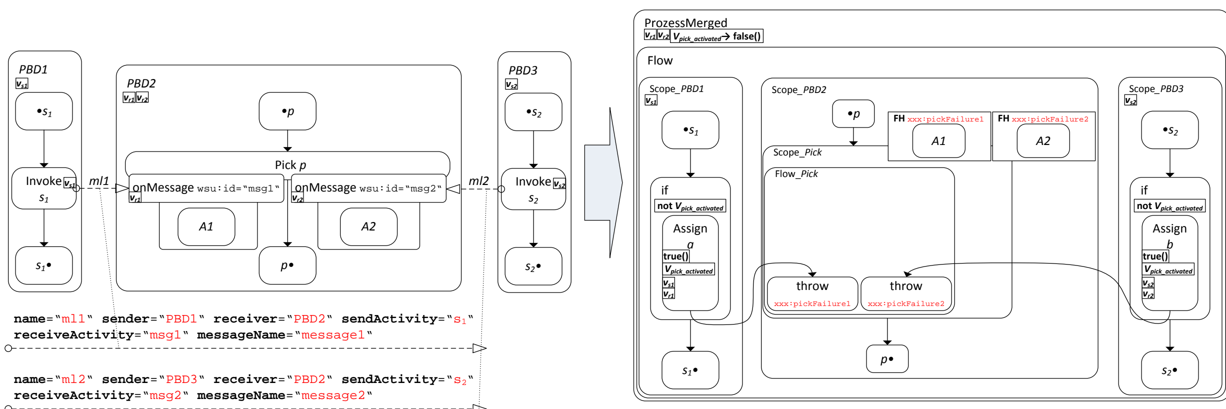


Abbildung 3.50 AsyncPattern2.2 mit zwei sendenden PBDs auf zwei <onMessage>-Zweige einer <pick>-Aktivität

Abbildung 3.50 zeigt die Anwendung des AsyncPattern2.2 auf drei miteinander kommunizierende PBDs *PBD1*, *PBD2* sowie *PBD3*. Die Konsolidierung erfolgt analog zu AsyncPattern2.1, indem die `<pick>`-Aktivität *p* durch die *Scope_Pick* Aktivität mit den benutzerdefinierten Faults für jeden konsolidierten `<onMessage>`-Zweig ersetzt wird. Im Gegensatz zu AsyncPattern2.1 wird die Zuweisung des `true` in die $v_{pick_activated}$ Schutzvariable jedoch bereits in den beiden `<assign>`-Aktivitäten *a* und *b* durchgeführt um zu verhindern, dass sobald einer der beiden ehemals sendenden Aktivitäten aktiviert wird, die entsprechend andere keine Werte mehr in die empfangende Variable kopiert (v_{r1} bzw. v_{r2}), da in der ursprünglichen Choreographie die `<pick>`-Aktivität ebenfalls bei Eingang einer Nachricht in den entsprechenden `<onMessage>`-Zweig deaktiviert wird. Dies sichert auch einen korrekten Kontroll- und Datenfluss falls die beiden `<onMessage>`-Zweige dieselbe Variable v_r verwenden.

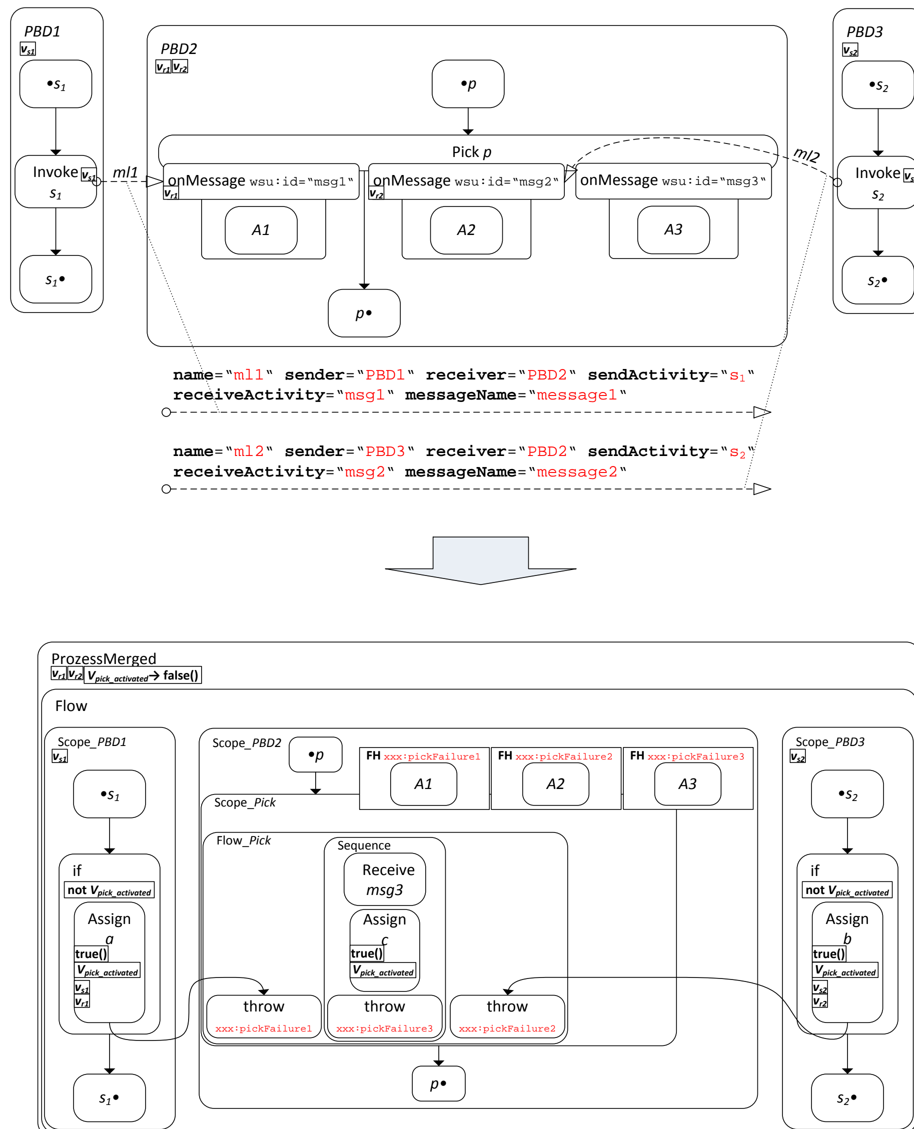
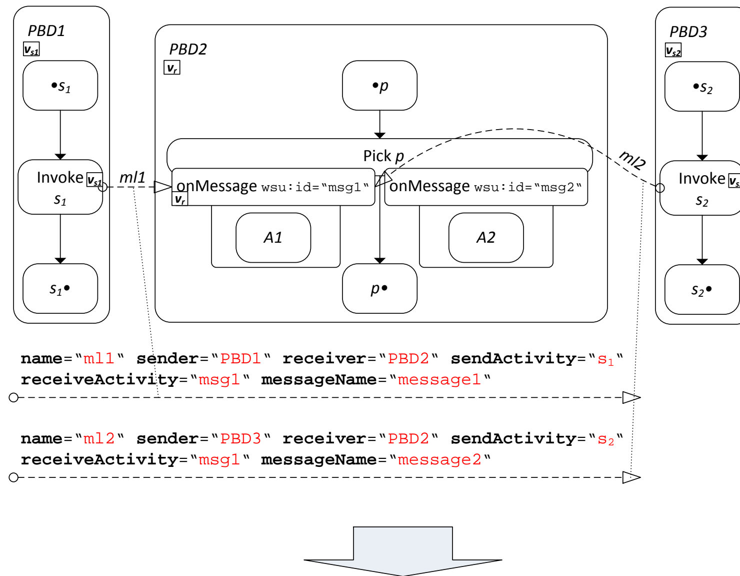


Abbildung 3.51 AsyncPattern2.2 mit einem weiteren choreographie-extern kommunizierenden `<onMessage>`-Zweig *msg3* in `<pick>` *p* sowie das Ergebnis der Konsolidierung

Abbildung 3.51 zeigt die vorhergehende Konsolidierung der drei PBDs mit einem weiteren choreographie-extern kommunizierenden `<onMessage>`-Zweig in der `<pick>`-Aktivität *p*. Hierbei wird die gleiche Kombination aus `<receive>`/`<throw>`-Aktivitäten sowie entsprechendem Fault Handler für *msg3* eingesetzt, wie in AsyncPattern2.1. Sind weiterhin `<onAlarm>`-Zweige definiert, so werden diese analog zu AsyncPattern2.1 in die neue *Scope_Pick* Aktivität konsolidiert (vgl. Abbildung 3.49).

3.3.1.11 AsyncPattern2.3 (Asynchrones n-zu-1 Senden auf einen `<onMessage>`-Zweig)

Das AsyncPattern2.3 dient dem Erkennen und anschließendem Konsolidieren einer BPEL4Chor Choreographie, in der mehrere verschiedene PBDs auf den gleichen `<onMessage>`-Zweig einer anderen PBD Nachrichten asynchron senden. Hierzu müssen die vorhergehenden AsyncPattern2.1 sowie AsyncPattern2.3 erweitert werden, um im Falle einer möglichen Wettlaufsituation den Kontroll- sowie Datenfluss im konsolidierten Prozess mit der ursprünglichen Choreographie synchron zu halten.



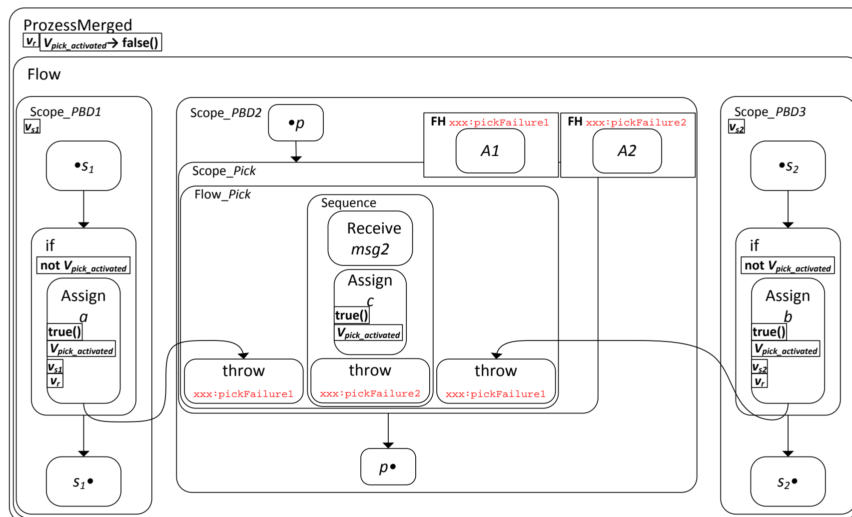


Abbildung 3.52 AsyncPattern2.3 Konsolidierung einer Choreographie mit zwei sendenden PBDs auf den gleichen `<onMessage>`**-Zweig *msg1* in** `<pick>` ***p***

Abbildung 3.52 zeigt Beispielfragmente einer Choreographie mit zwei sendenden PBDs *PBD1* sowie *PBD3*. Beiden senden Nachrichten auf den gleichen `<onMessage>`-Zweig *msg1* in der `<pick>`-Aktivität *p* von *PBD2*. Um dieses Muster zu erkennen sucht der Algorithmus nach weiteren Message Links aus *ML*, die als `receiveActivity` den gleichen `<onMessage>`-Zweig der entsprechenden `<pick>`-Aktivität enthalten, wie der bereits behandelte. Die Konsolidierung erfolgt analog zu AsyncPattern2.2 mit dem Unterschied, dass die beiden `<throw>`-Aktivitäten, die per Kontrollflusslink mit den neuen `<assign>`-Aktivitäten *a* und *b* verbunden sind, nun denselben benutzerdefinierten Fault werfen (`xxx:PickFailure2`). Sind weitere choreographie-extern kommunizierende `<onMes-`

sage>-Zweige bzw. <onAlarm>-Zweige enthalten, werden diese analog zu AsyncPattern2.1 konsolidiert.

3.3.1.11.1 Asynchrones Senden auf initiale <pick>-Aktivität (•p=∅)

Die AsyncPattern2.1-2.3 lassen sich auch in dem Fall anwenden, wenn die empfangende <pick>-Aktivität *p* keine Vorgängeraktivitäten besitzt: es gilt somit $\bullet p=\emptyset$ und zusätzlich ist das createInstance-Attribut auf „yes" gesetzt (daher dürfen keine <onAlarm>-Zweige enthalten sein). Nehmen wir das Beispiel aus Abbildung 3.45 und verändern es nun so, dass $\bullet p=\emptyset$ gilt und das createInstance-Attribut entsprechend gesetzt wird (createInstance="yes"). Die einzige Änderung am Konsolidierungsalgorithmus ist das Übernehmen des createInstance-Attributs für die <receive>-Aktivitäten, die die choreographie-extern kommunizierenden <onMessage>-Zweige im neuen <scope> *Scope_Pick* ersetzen. Abbildung 3.53 zeigt die Änderung an der Beispielchoreographie. Sei r_{pbd1_init} eine initiale <receive>-oder <pick>-Vorgängeraktivität von <invoke> *s* in *PBD1*. Diese Aktivität muss existieren, da sonst die Choreographie nicht korrekt ist (jeder BPEL-Prozess muss mindestens eine initiale <receive>-oder <pick>-Aktivität besitzen, vgl. [OAS07] Abschnitt 5.5 *The Lifecycle of an Executable Business Process*). Das gleiche gilt für *PBD2* und wird durch <pick> *p* erfüllt. Im konsolidierten Prozess *ProzessMerged* wird *p* durch die bekannte *Scope_Pick* Aktivität ersetzt und die choreographie-extern kommunizierenden <onMessage>-Zweige durch entsprechende <receive>-Aktivitäten, in unserem Beispiel *msg2* mit dem aus *p* übernommenen createInstance-Attribut. Je nachdem welche Aktivität, sei es r_{pbd1_init} oder *msg2*, die initiale Nachricht zuerst empfängt, erzeugt diese eine neue Prozessinstanz von *ProzessMerged* und die nächsten eintreffenden Nachrichten für r_{pbd1_init} bzw. *msg2* werden mit dieser Instanz korreliert. Hätte die in Abbildung 3.53 dargestellte <throw>-Aktivität, die den benutzerdefinierten Fault xxx:PickFailure1 wirft, keine Kontrollflussabhängigkeit auf <assign> *a* in *Scope_PBD1* so wäre der konsolidierte Prozess fehlerhaft. Kommunizieren alle <onMessage>-Zweige nur choreographie-intern, wird die Initiierung von *Scope_PBD2* durch die sendenden PBDs gesteuert. Im Gegensatz zu AsyncPattern1.5 werden wir deshalb kein gesondertes Erkennungs- und Konsolidierungsmuster für den Fall $\bullet p=\emptyset$ einführen, sondern lediglich das createInstance-Attribut von *p* für die choreographie-extern kommunizierenden <receive>-Aktivitäten übernehmen, wie im unterem Beispiel *msg2*.

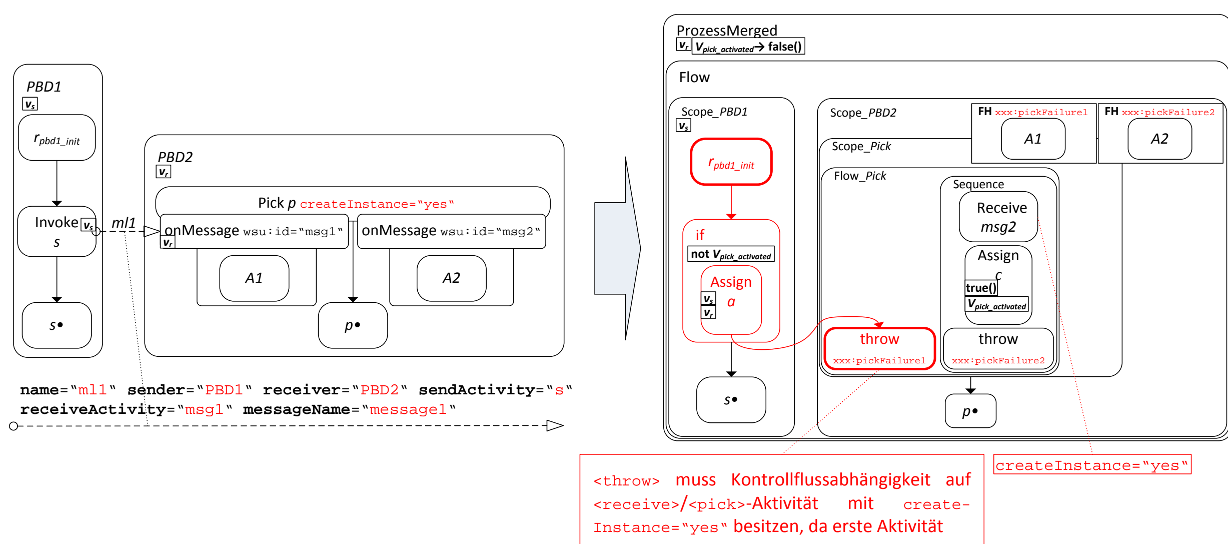


Abbildung 3.53 Konsolidierung einer Choreographie mit $\bullet p=\emptyset$ sowie createInstance="yes" von *p* mit AsyncPattern2.1

3.3.1.11.2 Syntaktische Transformation eines <onMessage>-Zweigs in eine <receive>-Aktivität

Der folgende kleine Abschnitt soll die syntaktische Transformation eines <onMessage>-Zweigs in eine äquivalente <receive>-Aktivität veranschaulichen. Hierzu sei angemerkt, dass die in einer PBD enthaltenen kommunizierenden Aktivitäten (<invoke>, <receive>, <reply>, <pick> sowie <onEvent>) die technischen Attribute partnerLink, portType sowie operation nicht verwenden dürfen. Stattdessen werden diese über die Grounding-Datei der Choreographie sowie die für die Eingabe der Konsolidierung notwendigen WSDL-Dateien bereitgestellt. Wir werden diese Attribute erst nach der Konsolidierung des Daten- und Kontrollflusses in der neuen WSDL-Datei für den konsolidierten Prozess zusammenführen und in die verbliebenen intra- sowie inter-Prozess kommunizierenden Aktivitäten im Prozess ProzessMerged einfügen. Die intra-Prozess kommunizierenden Aktivitäten sind hierbei diejenigen, die zwar vor der Konsolidierung choreographie-intern kommunizieren, für die es jedoch keine passenden Merge-Patterns gibt und diese daher nach der Konsolidierung rekursiv mit dem eigenen Prozess kommunizieren. Die inter-Prozess kommunizierenden Aktivitäten sind dagegen diejenigen, die nach der Konsolidierung für andere Prozesse oder Web Services zur Kommunikation zur Verfügung stehen. Abbildung 3.54 veranschaulicht die Transformation eines <onMessage>-Zweigs einer <pick>-Aktivität in eine äquivalente <receive>-Aktivität, anhand der XML-Syntaxen (vgl. [OAS07]).

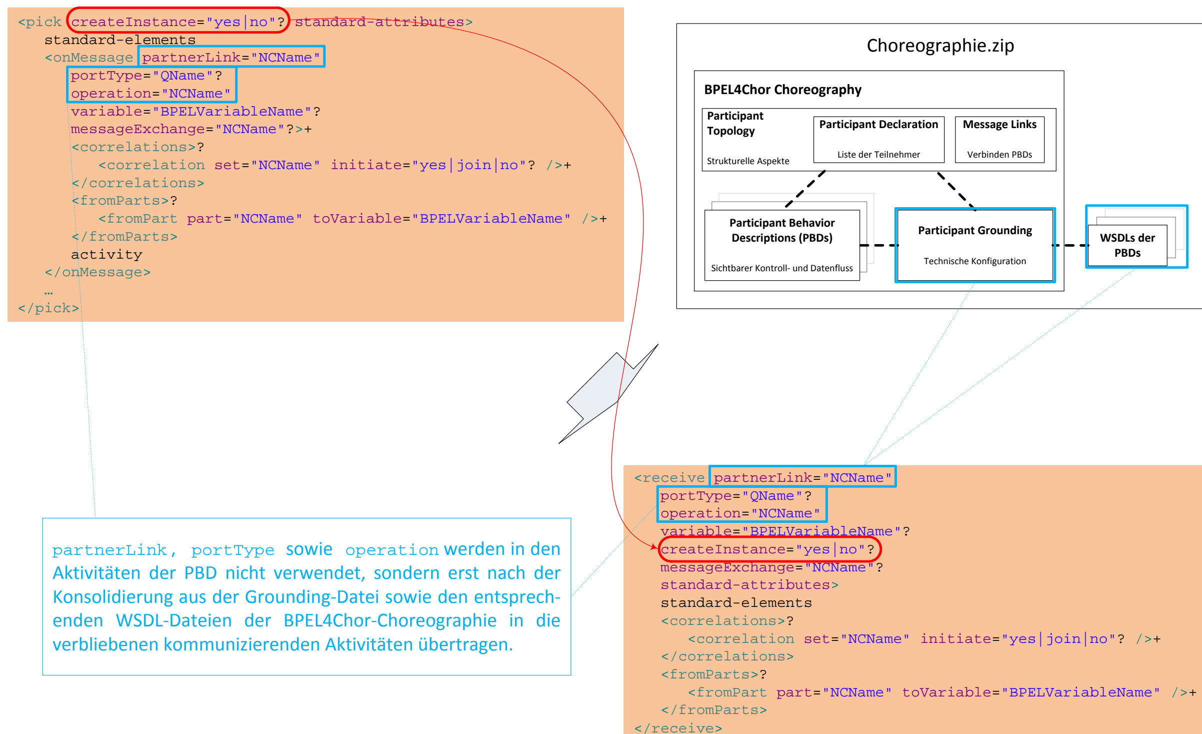


Abbildung 3.54 Syntaktische Umformung eines <onMessage>-Zweigs in eine äquivalente <receive>-Aktivität: Der wsu:id-Bezeichner des <onMessage>-Zweigs (nicht abgebildet) wird zum name-Attribut der <receive>-Aktivität, das createInstance-Attribut der umschließenden <pick>-Aktivität wird in das neue <receive> übernommen. Erst nach der Konsolidierung werden die technischen Attribute partnerLink, portType sowie operation aus der Grounding-Datei sowie den entsprechenden WSDL-Dateien der BPEL4Chor-Choreographie übernommen und in die kommunizierenden Aktivitäten übertragen.

3.3.1.12 AsyncPattern3.0 („Non-Merge-Pattern-Async“)

Das AsyncPattern3.0 stellt eine besondere Form der Merge-Patterns dar: Da wir in der vorliegenden Arbeit nicht alle möglichen Kombinationen und Konstellationen aus einer asynchron sendenden und

einer entsprechenden empfangenden Aktivität erkennen und konsolidieren, wird das AsyncPattern3.0 eingeführt um diese Sonderfälle bzw. nicht implementierten Kommunikationsmuster abzufangen und gesondert zu behandeln.
Sei *PBD1* eine PBD die über den Message Link *ml* aus *ML* mit einer weiteren PBD *PBD2* kommuniziert. Trifft nun der hier erwähnte Erkennungsalgorithmus auf ein nicht-konsolidierbares asynchrones Kommunikationsmuster so gibt es folgende Möglichkeiten der Bearbeitung einer solchen Situation:

1. Der Algorithmus bricht ab und signalisiert per Fehlermeldung, dass die vorliegende BPEL4-Chor-Choreographie ein nicht konsolidierbares Muster enthält und beendet die Verarbeitung. Diese Variante hat den Nachteil, dass selbst wenn die Choreographie andere konsolidierbare Muster enthält, diese ebenfalls verworfen werden.
2. Der Algorithmus speichert den betroffenen Message Link *ml* in einer speziellen Liste *NMML* (*Non-Mergeable-Message-Links*) für nicht konsolidierbare Muster und fährt mit der Erkennung und Konsolidierung der verbleibenden Message Links aus *ML* fort. Diese Liste enthält alle choreographie-intern kommunizierenden Message Links, deren Aktivitäten nach der Untersuchung aller verbliebenen Message Links als intra-Prozess kommunizierende Aktivitäten im neuen konsolidierten Prozess konfiguriert werden. Wie in Abschnitt 3.3.1 erwähnt, bieten die beiden verwendeten BPEL-Engines *Apache ODE* [AODE11] sowie *bpel-g* [BPLG12] spezielle Konfigurationsoptionen an um diese rekursiven Aufrufe ohne SOAP-Message-Handling zu optimieren.

Wir werden in der vorliegenden Arbeit die zweite Variante verwenden um die bereits implementierten Merge-Patterns anwenden zu können. Zusätzlich kann der Katalog der Merge-Patterns durch zukünftige Erweiterungen auch die noch nicht umsetzbaren Muster konsolidierbar machen.
In den folgenden Unterabschnitten werden die Kombinationen und Konstellationen aus sendender und empfangender Aktivität erläutert in denen das AsyncPattern3.0 den betreffenden Message Link *ml* erkennt und in die Liste *NMML* einfügt.

3.3.1.12.1 `<onEvent>`-Zweig (EH) als empfangende Aktivität

In einem Message Link *ml* einer BPEL4Chor-Choreographie kann auch die `wsu:id` eines Event Handlers eines BPEL-Prozesses bzw. einer `<scope>`-Aktivität (`<onEvent>`-Zweig) als `receive-Activity` auftreten. Ein solcher `<onEvent>`-Zweig steht dann zur Verfügung sobald der diesem Event Handler zugehörige BPEL-Prozess instanziiert bzw. die entsprechende `<scope>`-Aktivität aktiviert wurde. Er wird aktiviert sobald eine entsprechende Nachricht eingegangen ist und erlaubt die mehrfache nebenläufige Ausführung der in ihm enthaltenen `<scope>`-Aktivität bei Eintreffen mehrerer dieser Nachrichten. Die in dieser Arbeit implementierten Konsolidierungsmuster unterstützen keine Event Handler als `receiveActivity`, daher wird ein solcher Message Link *ml* der eine `wsu:id` eines solchen `<onEvent>`-Zweigs enthält in die Liste *NMML* eingefügt und die Kommunikation der enthaltenen `sendActivity` sowie des Event Handlers nach Untersuchung der verbliebenen Message Links aus *ML* in eine intra-Prozess kommunizierende umgewandelt.

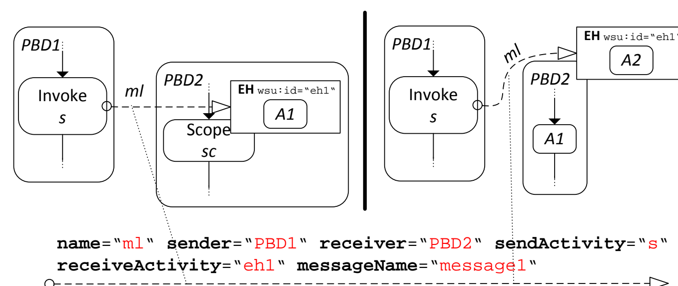


Abbildung 3.55 AsyncPattern3.0 Muster: `<onEvent>`-Zweig als `receiveActivity` in einem MessageLink *ml*

3.3.1.12.2 Sendende `<invoke>`-Aktivität innerhalb von Handlern (EH, CH, TH, FH)

Befindet sich die sendende `<invoke>`-Aktivität *s* vor der Konsolidierung der Choreographie in einem `<onEvent>`- oder `<onAlarm>`-Zweig eines Event Handlers bzw. einem Compensation Handler (CH→`<compensationHandler>`) einer `<scope>`-Aktivität, so hätten die zuvor vorgestellten Merge-Patterns einen Kontrollflusslink zur Folge der ausgehend von der *s* ersetzenden Aktivität das umgebende Konstrukt (EH oder CH) verlässt. Dies ist durch den WS-BPEL 2.0 Standard verboten (vgl. SA00070 [OAS07]). Ein Link darf niemals in einen CH oder EH eintreten bzw. von diesem ausgehen und darf nur nach Deklaration innerhalb einer `<flow>`-Aktivität, die sich selbst in diesem Handler befindet, in einem solchen verwendet werden. Daher werden wir solche Konfigurationen verbieten und falls eine derartige choreographie-intern kommunizierende `<invoke>`-Aktivität *s* als `sendActivity` in einem Message Link *ml* aus *ML* gefunden wird, den entsprechenden Message Link in die Liste *NMML* übertragen. Zu diesem Zweck wird die Funktion *isActivityInHandler*(a_{invoke}) eingeführt, die für eine gegebene Aktivität a_{invoke} prüft, ob sich diese innerhalb eines EHs oder CHs befindet (Auflistung 3.8).

```
(1)  boolean isActivityInHandler(s_invoke)
(2)  begin
(3)    while ( (typeof(s_invoke) ≠ <process>) && (typeof(s_invoke) ≠ <eventHandler>)
(4)           && (typeof(s_invoke) ≠ <compensationHandler>) ) do
(5)      s_invoke = par(s_invoke)
(6)    od
(7)    if ( typeof(s_invoke) == <process> )
(8)      return false
(9)    else
(10)     return true
(11)   fi
(12) end
```

Auflistung 3.8 Pseudocode *isActivityInHandler*(a_{invoke})-Funktion

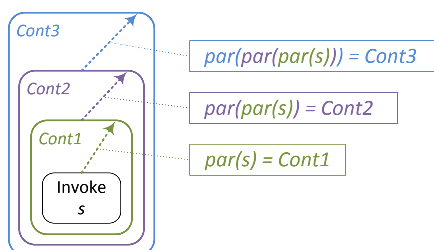


Abbildung 3.56 Rekursive Untersuchung von `par(s)`

Hierbei wird das *s*-enthaltende Modellkonstrukt *par(s)* untersucht, ob es ein `<process>`-, `<compensationHandler>`- oder `<eventHandler>`-Konstrukt ist (vgl. Abbildung 3.56). Trifft dies zu bricht die Funktion ab und prüft das entsprechende Konstrukt auf den gesuchten Typ. Ansonsten fährt die Funktion mit dem *par(s)* enthaltenden Konstrukt *par(par(s))* fort, bis der Prozessscope erreicht wurde. Da die Aktivitäten in BPEL blockstrukturiert sind, bricht diese Überprüfung spätestens hier ab.

Befindet sich *s* dagegen in einem Termination Handler (TH→`<terminationHandler>`) bzw. einem Fault Handler (FH→`<catch>`,`<catchAll>`), so ist der ausgehenden Kontrollflusslink zulässig, da für THs und FHs nur ausgehende Links zulässig sind und wir davon ausgehen können, dass die vorliegende PBD kein *s* enthält, das rekursiv in einen diesem Handler zugehörigen `<scope>` bzw. `<process>` sendet, da dieser bereits einen Fault geworfen hat oder in einem entsprechendem Konstrukt enthalten ist.

3.3.1.12.3 Empfangende `<receive>`/`<pick>`-Aktivität innerhalb von Handlern (EH, CH, TH, FH)

Für die empfangende `<receive>`- oder `<pick>`-Aktivität *r* gilt dagegen: Befindet sich *r* innerhalb eines EHs, CHs, FHs oder THs, so wird der entsprechende Message Link *ml* aus *ML*, der diese Aktivität als `receiveActivity` enthält, in die Liste *NMML* übertragen, da SA00070 sowie SA00071 des WS-BPEL 2.0 Standards [OAS07] eingehende Links in jegliche Art der vier Handler verbieten.

3.3.1.12.4 Sendende und/oder empfangende Aktivitäten innerhalb von Schleifen

Befindet sich die sendende `<invoke>`-Aktivität *s* und/oder die empfangende `<receive>`/`<pick>`-Aktivität *r* eines Message Links *ml* aus *ML* innerhalb einer Schleife *lp* (*typeof(lp)* ∈ *{`<while>`, `<repeatUntil>`, `<forEach>`}*) so wird dieser Message Link *ml*, der *s* als `sendActivity` und/oder *r* als `receiveActivity` enthält in die Liste *NMML* übertragen, da SA00070 eingehende als auch ausgehende Kontrollflusslinks in einer Schleife verbietet und wir mit den zuvor erwähnten Konsolidierungsmustern einen neuen synchronisierenden Kontrollflusslink einfügen, der den Nachrichtenaustausch zwischen *s* und *r* emuliert.
In ihrer Arbeit [KHA08] beschreibt Khalaf eine Methode der Synchronisation von fragmentierten Schleifen über Prozessgrenzen hinweg mithilfe des *WS-Coordination* Protokolls [OAS07b]. Wie zu Anfang bereits erwähnt, ist das Ziel dieser Arbeit einen ausführbaren BPEL-Prozess ohne jegliche Zusatzfragmente externer Koordinations- und Synchronisationsprotokolle als Konsolidierung einer BPEL4Chor-Choreographie bereitzustellen, daher werden wir auf diesen Ansatz verzichten. Ein weiterer möglicher Ansatz wäre eine Transformation des konsolidierten Prozesses bei Auftreten einer Schleife *lp* als umgebende Aktivität von *s* und/oder *r* in einen äquivalenten Prozess der die *WS-BPEL Extension for Sub-Process* (*BPEL-SPE* [KKL[+]05]) nutzt, doch auch hier wird eine Unterstützung dieser Erweiterung durch die verwendete BPEL-Engine vorausgesetzt und unsere Vorgabe des Verzichts externer Protokolle gebrochen. Zukünftige Arbeiten werden hierfür eine adäquate Lösung liefern können.
Eine mögliche jedoch nicht in der vorliegenden Arbeit implementierte Lösungsalternative veranschaulicht Abbildung 3.57. Die beiden PBDs *PBD1* sowie *PBD2* kommunizieren über die sendende `<invoke>`-Aktivität *s* und die empfangende `<receive>`-Aktivität *r* miteinander.

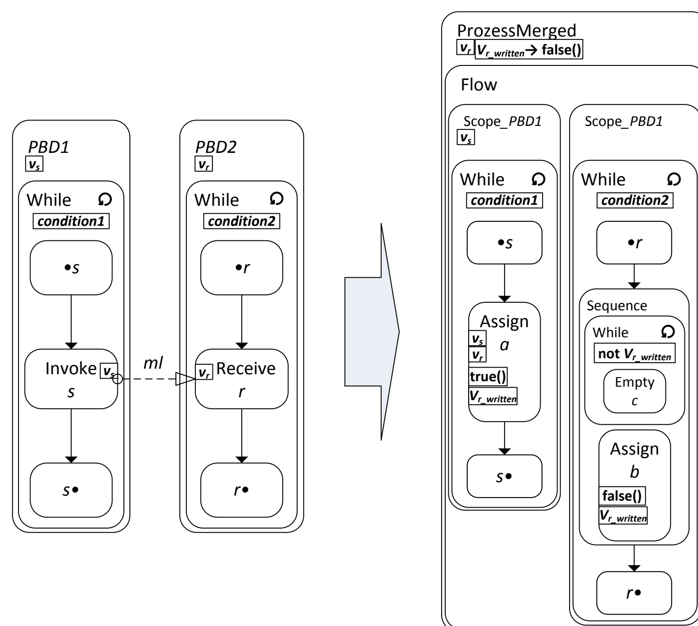


Abbildung 3.57 Sendende Aktivität `<invoke>` *s* und empfangende Aktivität `<receive>` *r* und eine Konsolidierung ohne Kontrollflusslink

Beide Aktivitäten befinden sich jeweils innerhalb einer `<while>`-Schleife. Eine mögliche Konsolidierung ohne Kontrollflusslink zwischen den beiden Schleifen im neuen *ProzessMerged* basiert erneut auf Schutzvariablen: *s* wird durch eine `<assign>`-Aktivität *a* ersetzt, die v_s nach v_r kopiert und anschließend den Wert der zuvor *inline*-initialisierten Schutzvariable $v_{r_written}$ auf `true` setzt. Auf der Empfängerseite wird *r* durch eine `<sequence>`-Aktivität ersetzt, sei diese hier *seq*, die eine weitere `<while>`-Schleife sowie eine `<assign>`-Aktivität *b* enthält. Die `<while>`-Schleife beinhaltet lediglich eine `<empty>`-Aktivität *c* und wartet bis die Variable $v_{r_written}$ mit `true` beschrieben wurde (*Polling*). Anschließend setzt sie diese in *b* wieder auf `false` und fährt mit dem Kontrollfluss nach der ursprünglichen Aktivität *r* fort (*r*•). Hierbei kann es jedoch passieren, dass die Werte von v_r bei einem erneuten Schleifendurchlauf der sendenden `<while>`-Aktivität überschrieben werden, falls die beiden Schleifen nicht synchron laufen.

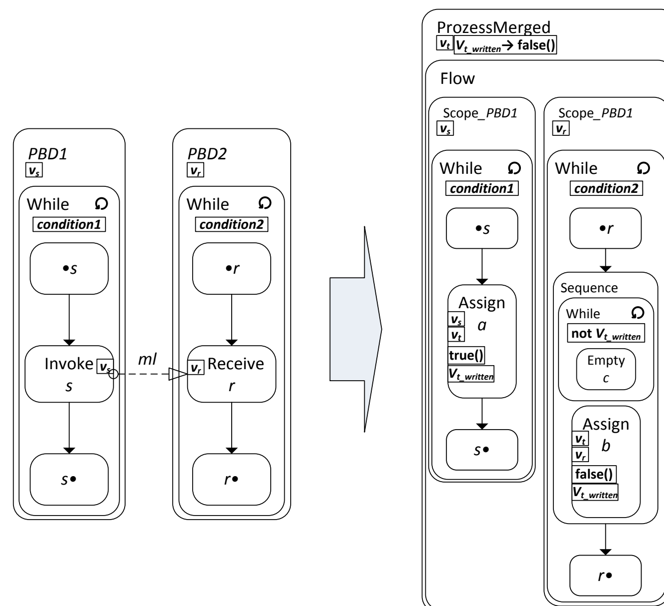


Abbildung 3.58 Sendende Aktivität `<invoke>` *s* und empfangende Aktivität `<receive>` *r* und eine Konsolidierung ohne Kontrollflusslink mit zusätzlicher Variable v_t

Abbildung 3.58 zeigt eine Erweiterung der zuvor erwähnten Variante mit einer zusätzlichen Variable v_t: *a* kopiert nun v_s nach v_t, die *b* nach v_r kopiert wird. Auch hier kann es jedoch passieren, dass wenn beispielsweise die `<while>`-Schleife in *Scope_PBD2* langsamer läuft als die `<while>`-Schleife in *Scope_PBD1* noch nicht nach v_r kopierte Werte von v_t durch neue Werte von v_s überschrieben werden. Eine weitere mögliche Lösung für dieses Problem wäre beispielsweise eine komplexe Variable v_t, die aus einer Liste von v_s-Variablen besteht und in *a* durch einen speziellen *doXslTransform*-Aufruf mit v_s-Variablen befüllt wird und aus der *b* jeweils das letzte Element dieser Liste nach v_r kopiert (vgl. [OAS07] *doXslTransform*-Beispiel im Abschnitt 8.4 *Assignment, Iterative document construction*).

3.3.2 Synchrone Merge-Patterns

Die synchronen Merge-Patterns sind analog zu den asynchronen charakterisiert durch eine sendende und eine empfangende Aktivität, die über einen Message Link miteinander kommunizieren. Als sendende Aktivität steht in BPEL für den synchronen Fall hierfür die `<invoke>`-Aktivität zur Verfügung. Als empfangende Aktivitäten kommen `<receive>`-Aktivitäten, `<onMessage>`-Zweige der `<pick>`-Aktivität sowie `<onEvent>`-Zweige der Event Handler einer `<scope>`-Aktivität oder des Prozess-Scopes in Frage. Zusätzlich zum sendenden Message Link ml_{Send} existiert im synchronen Fall noch ein weiterer Message Link ml_{Reply} indem die sendenden `<invoke>`-Aktivität *s* als `receiveActivity` definiert ist und eine `<reply>`-Aktivität *y* als `sendActivity`. Analog zu den asynchro-

nen Merge-Patterns werden wir in diesem Abschnitt zunächst die allgemeinen Konsolidierungsmuster und anschließend die speziellen mit besonderen Umgebungsbedingungen vorstellen. Auch bei den synchronen Merge-Patterns wird es Fälle geben, in denen wir kein passendes Konsolidierungsmuster liefern können, daher verbleiben auch hier die kommunizierenden Aktivitäten im neuen konsolidierten Prozess und werden am Ende zu intra-Prozess kommunizierenden konfiguriert. Analog zu den asynchronen werden wir hierfür beim Fund einer solchen Kombination aus ml_{Send} und ml_{Reply} die entsprechenden Message Links in die Liste *NMML* übertragen. Abbildung 3.59 zeigt den schematischen Aufbau der Suche nach einem passenden Merge-Pattern für die beiden Message Links ml_{Send} und ml_{Reply} analog zum asynchronen Fall.

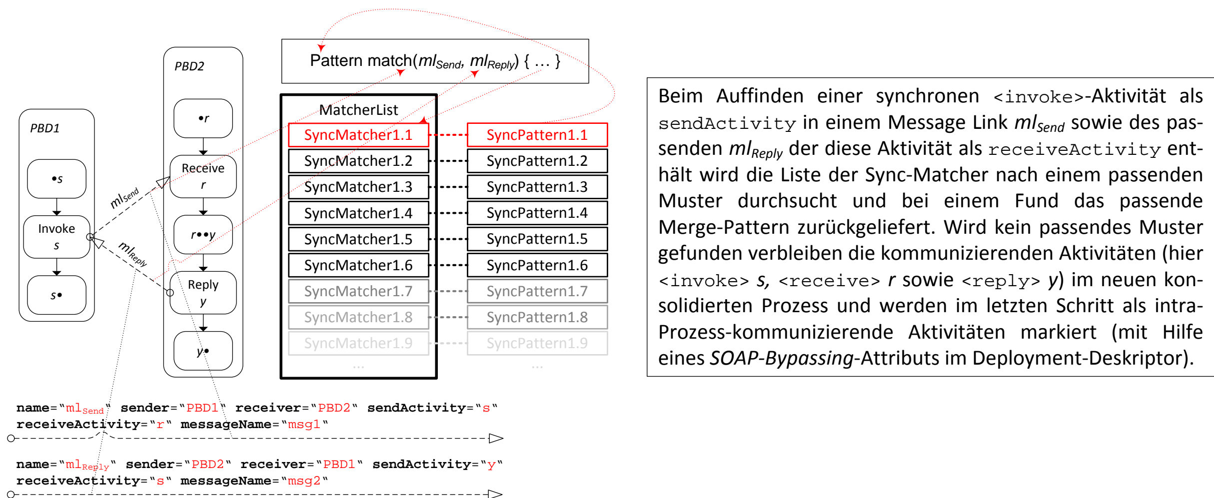


Abbildung 3.59 Anwendung des Merge-Algorithmus: Für jedes Paar synchron kommunizierender Links ml_{Send} sowie ml_{Reply} wird die Liste der bekannten SyncMatcher nach einem Muster durchsucht und im Falle einer Übereinstimmung das entsprechende SyncPattern zurückgeliefert.

3.3.2.1 SyncPattern1.1

Das SyncPattern1.1 ist das einfachste synchrone Merge-Pattern und basiert auf den Überlegungen aus der Variante 2 der synchronen Kommunikationsmuster aus Abschnitt 3.2.2. Abbildung 3.60 zeigt eine Beispielchoreographie in der die beiden PBDs *PBD1* sowie *PBD2* über die beiden Message Links ml_{Send} und ml_{Reply} synchron miteinander kommunizieren.

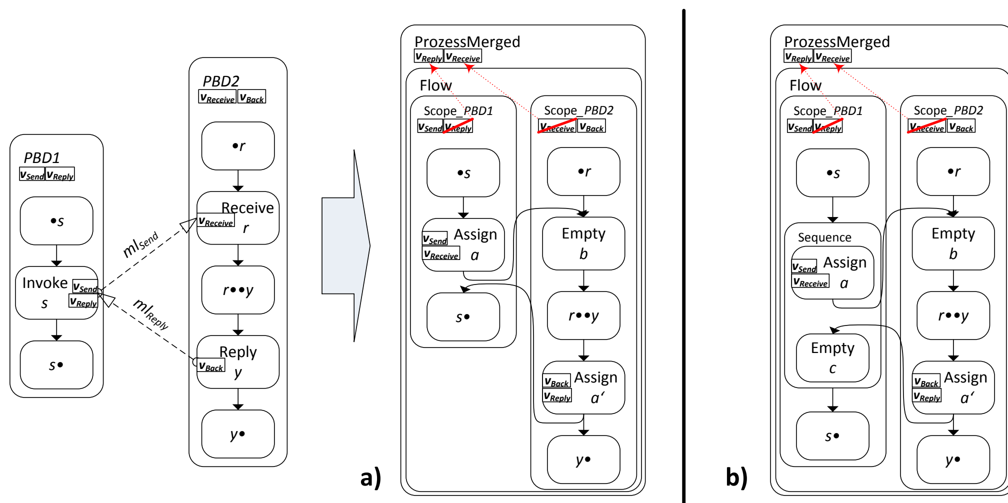


Abbildung 3.60 SyncPattern1.1

Die Beschaffenheit der beiden PBDs ist ähnlich der Beispielfragmente des AsyncPattern1.1: *PBD1* kommuniziert über die synchrone <invoke>-Aktivität *s* mit der <receive>-Aktivität *r* in *PBD2*. Im Gegensatz zum asynchronen Fall ist *s* im synchronen Fall jedoch blockierend. Der Ablauf des Kontrollflusses in *PBD1* schreitet erst voran wenn *s* die Antwort aus *PBD2* erhalten hat. Hierzu besitzt es zusätzlich zu der zu sendenden Variable v_{Send} die Variable v_{Reply}, in der die eintreffende Antwort gespeichert wird. *•s=prel(s)+pre(s)* bezeichnet hier wieder die Menge der direkten Vorgängeraktivitäten von *s*, *s•=succl(s)+succ(s)* analog hierzu die Menge der direkten Nachfolgeraktivitäten von *s*. Entsprechend steht *•r=prel(r)+pre(r)* für die Menge der direkten Vorgängeraktivitäten von *r*. Zusätzlich zu *r* gibt es im synchronen Fall jedoch noch eine antwortsendende <reply>-Aktivität *y*, die auf *s* sendet. Daher ist *s* einmal als `sendActivity` in ml_{Send} vertreten und ein weiteres Mal in ml_{Reply} als `receiveActivity`. Auf der Empfängerseite bezeichnet *r••y=succl(r)+succ(r)+r••+...+••y+ prel(y)+pre(y)* die Menge aller direkten Nachfolgeaktivitäten von *r*, die auf einem direkten Kontrollflusspfad zu *y* liegen sowie alle direkten Vorgängeraktivitäten von *y*. *y•=succl(y)+succ(y)* steht für die Menge aller direkten Nachfolgeaktivitäten der antwortsendenden Aktivität *y*. Zusätzlich definieren wir noch *r•̶•y=succl(r)+succ(r)+r••+...+•̶•y* als die Menge aller direkten Nachfolgeaktivitäten von *r*, die nicht auf einem direkten Kontrollflusspfad zu *y* liegen.
Parallel zu AsyncPattern1.1 ersetzen wir wieder *s* durch eine <assign>-Aktivität *a*, die v_{Send} nach $v_{Receive}$ kopiert. Da $v_{Receive}$ in *Scope_PBD2* definiert wurde und nur dort sichtbar ist, müssen wir diese Variable wieder in den Prozessscope verschieben. Im Falle eines Namenskonfliktes muss $v_{Receive}$ wieder umbenannt werden (vgl. AsyncPattern1.1). Zusätzlich wird ein neuer Kontrollflusslink ausgehend von *a* und eingehend in die *r* ersetzende <empty>-Aktivität *b* in der <flow>-Aktivität des konsolidierten Prozesses hinzugefügt (vgl. AsyncPattern1.1). Bis hier können wir den Ersetzungsmechanismus von *s* und *r* aus dem AsyncPattern1.1 wiederverwenden. Anschließend müssen wir jedoch auch die <reply>-Aktivität *y* durch eine neue <assign>-Aktivität *a‘* ersetzen. Diese kopiert die Antwortvariable aus v_{Back} nach v_{Reply}. Da auch v_{Reply} analog zu $v_{Receive}$ nur in *Scope_PBD1* sichtbar ist, verschieben wir auch diese Variable in den Prozessscope. Wir emulieren hier das Senden der Antwort von *y* nach *s* durch *a‘* und müssen nun wieder den Kontrollfluss in die Nachfolgeaktivitäten aus *s•* wiederherstellen. Hierzu bieten sich, je nach Konfiguration der ausgehenden Kanten von *s,* zwei Varianten an:
a): Gibt es ausgehende Kanten von *s* (<sources> mit <targets> in *succl(s)*) und besitzen diese explizite <transitionCondition>s, deren XPath-Ausdrücke möglicherweise auf Variablen aus *Scope_PBD1* lesend zugreifen, so müssen wir zunächst die entsprechenden Variablen in *Scope_PBD2* sichtbar machen, da wir in dieser Variante alle <sources> aus *s* inklusive ihrer expliziten <transitionCondition>s zu den <sources> von *a‘* hinzufügen. Gibt es keine expliziten <transitionCondition>s, so übernehmen wir alle <sources> nach *a‘* und fügen für jeden Aktivität $s_{succ(s)}$ aus *succ(s)* einen weiteren Kontrollflusslink von *a‘* nach $s_{succ(s)}$ hinzu.
b): In dieser Variante ersetzen wir *s* durch eine <sequence>-Aktivität *seq*, die *a* und zusätzlich noch eine synchronisierende <empty>-Aktivität *c* enthält. Alle <targets> und ihre <joinCondition> von *s* werden zu den <targets> von *seq*, ebenso alle <sources> und ihre <transitionCondition>s. Hier benötigen wir nur noch einen Kontrollflusslink von *a‘* nach *c*.
Wir werden im folgenden Kapitel die Variante **b)** für die synchronen Fälle verwenden.

3.3.2.2 SyncPattern1.2

Das SyncPattern1.2 ist ähnlich dem AsyncPattern1.5 aufgebaut: Die empfangende <receive>-Aktivität *r* besitzt keine direkten Vorgängeraktivitäten. Daher ist auch hier das `createInstance`-Attribut von *r* auf „`yes`“ gesetzt und es gilt *•r=∅*. Abbildung 3.62 zeigt das SyncPattern1.2 an einem Beispielfragment zweier kommunizierender PBDs. *PBD1* und *PBD2* kommunizieren synchron über die beiden Message Links ml_{Send} sowie ml_{Reply} (*s→r*, *y→s*) miteinander. Hierbei ist jedoch *PBD1* der Initiator von *PBD2* (*•r=∅*). Da die <receive>-Aktivität *r* keine direkten Vorgängeraktivitäten besitzt kann in diesem Fall auf eine synchronisierende <empty>-Aktivität verzichtet werden. *s* wird durch die <sequence>-Aktivität *seq* aus Variante **b)** des SyncPattern1.1 ersetzt. *seq* enthält die <assign>-

Aktivität *a*, die v_{Send} nach $v_{Receive}$ kopiert sowie eine synchronisierende `<empty>`-Aktivität *c*. Zusätzlich muss der Kontrollfluss zwischen *Scope_PBD1* mit dem von *Scope_PBD2* verbunden werden. Hierzu werden alle ausgehenden Links von *r*, inklusive möglicher `<transitionCondition>`s zu den ausgehenden Links von *a* hinzugefügt und in die Prozess-`<flow>`-Aktivität übertragen. Existieren keine ausgehenden Links, es gilt somit $succl(r)=\emptyset$, wird ein neuer Link *ln* (nicht abgebildet) ausgehend von *a* und eingehend in die Nachfolgeraktivität *succ(r)* hinzugefügt. Der Kontrollfluss zwischen *a‘* (zuvor *y*) und *c* wird wie in SyncPattern1.1 hergestellt.

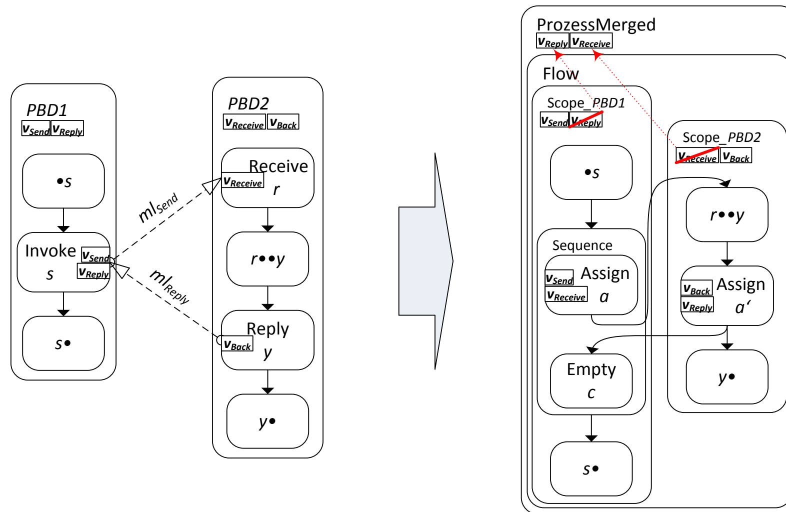


Abbildung 3.62 SyncPattern1.2

3.3.2.3 SyncPattern1.3

Das SyncPattern1.3 ist charakterisiert durch eine leere Menge an Nachfolgeraktivitäten der synchronen `<invoke>`-Aktivität *s*, es gilt somit $s\bullet=\emptyset$. Da die sendende PBD die Daten aus der Antwortnachricht von *s* nicht mehr verwendet, werden wir die antwortsendende `<reply>`-Aktivität *y* durch eine synchronisierende `<empty>`-Aktivität ersetzen.

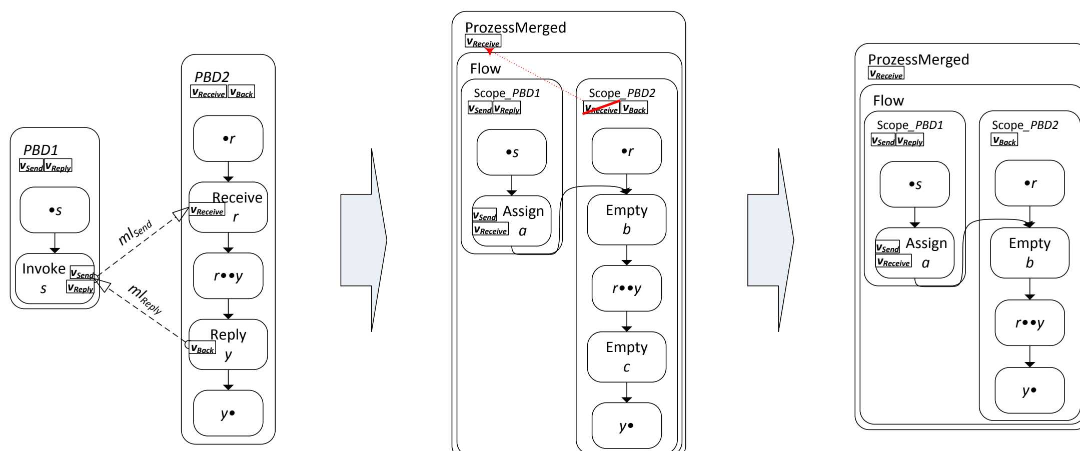


Abbildung 3.63 SyncPattern1.3

Abbildung 3.63 zeigt die Fragmente einer Beispielchoreographie mit zwei synchron miteinander kommunizierenden PBDs *PBD1* sowie *PBD2*, die über die beiden Message Links ml_{Send} sowie ml_{Reply} verbunden sind. Anstatt *s* in *Scope_PBD1* durch die bekannte `<sequence>`-Aktivität *seq* zu ersetzen,

benötigen wir in dieser Konstellation nur eine s ersetzende `<assign>`-Aktivität a, die v_{Send} nach $v_{Receive}$ kopiert und per Kontrollflusslink mit der entsprechenden synchronisierenden r ersetzenden `<empty>`-Aktivität b in *Scope_PBD2* verbunden ist. Anschließend wir y durch eine synchronisierende `<empty>`-Aktivität c ersetzt, die im günstigsten Fall bei Anwendung des `<empty>`-Optimierers aus Abschnitt 3.3.1.1.2 ebenfalls entfällt. Der Kontrollfluss zwischen $r\bullet\bullet y$ und $y\bullet$ wird danach direkt verbunden (rechtes Ergebnis der Konsolidierung).

3.3.2.4 SyncPattern1.4 (Multiple `<reply>`-Aktivitäten)

Wie in Abschnitt 3.3.1.1.1 beschrieben ist es möglich, dass eine `<invoke>`-Aktivität s direkt Compensation sowie Fault Handler definiert. Im synchronen Fall können so über die Antwortnachricht mögliche Faults signalisiert und entsprechend in s behandelt werden (vgl. [OAS07] Abschnitt 10.4). Abbildung 3.64 zeigt die Bespielfragmente einer BPEL4Chor-Choreographie in der die beiden PBDs *PBD1* sowie *PBD2* synchron miteinander kommunizieren und die sendende Aktivität s zusätzlich Fault sowie Compensation Handler definiert.

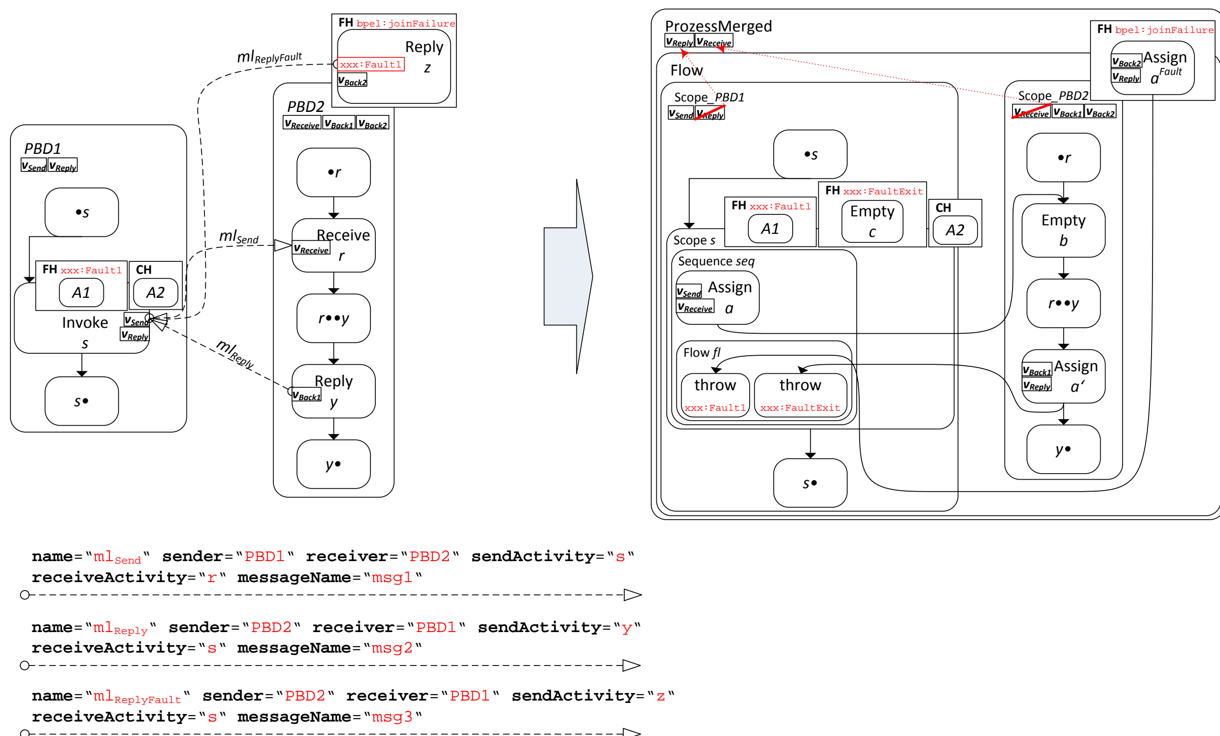


Abbildung 3.64 SyncPattern1.4 mit zwei `<reply>`-Aktivitäten und einem Fault

PBD1 sendet über die synchrone `<invoke>`-Aktivität s Nachrichten an die `<receive>`-Aktivität r in *PBD2*. Dieser Teil der Kommunikation wird über der Message Link ml_{Send} aus *ML* repräsentiert. Zusätzlich definiert s einen Compensation Handler mit der Akivität *A2* sowie einen Fault Handler für den WSDL-Fault `xxx:Fault1` mit der Aktivität *A1*. Im synchronen Fall können in WS-BPEL 2.0 direkt Fault Handler für WSDL-Faults in einer `<invoke>`-Aktivität definiert werden, die den entsprechenden Fault aus der Nachricht der antwortsendenden `<reply>`-Aktivität abfangen und verarbeiten. Jeder dieser Faults wird über einen QName identifiziert, der aus dem Namespace des entsprechenden Port Types sowie einem Fault Namen besteht. *PBD2* antwortet entweder über die `<reply>`-Aktivität y, dargestellt über den Message Link ml_{Reply} aus *ML*, oder über die `<reply>`-Aktivität z im Fault Handler von *PBD2*, dargestellt über den Message Link $ml_{ReplyFault}$ aus *ML*. Der

Erkennungsalgorithmus für das SyncPattern1.4 sucht bei jedem Auftreten einer synchronen Kommunikation neben dem sendenden Message Link aus *ML* (hier ml_{Send}) nach dem antwortenden in dem *s* die `receiveActivity` ist (hier ml_{Reply}) sowie nach weiteren MessageLinks in denen *s* ebenfalls als `receiveActivity` auftritt (hier $ml_{ReplyFault}$). Ist in einer der entsprechenden antwortsendenden `<reply>`-Aktivitäten das `faultName`-Attribut definiert, so handelt es sich im eine WSDL-Fault sendende Antwortaktivität. Im konsolidierten Prozess *ProzessMerged* wird nun `<invoke>` *s* in *Scope_PBD1* durch eine `<scope>`-Aktivität *s* ersetzt, in die der Compensation Handler mit *A2* kopiert wird. *s* enthält wieder eine `<sequence>`-Aktivität *seq*. *seq* umfasst eine `<assign>`-Aktivität *a*, die v_{Send} nach $v_{Receive}$ kopiert und per ausgehenden Kontrollflusslink mit der `<receive>` *r* ersetzenden synchronisierenden `<empty>`-Aktivität *b* in *Scope_PBD2* verbunden ist (vgl. Standardfall in SyncPattern1.1). Da wir nun zwei mögliche antwortsendende Aktivitäten ersetzen müssen, *y* und *z*, wird *seq* aus SyncPattern1.1 erweitert: Anstatt einer synchronisierenden `<empty>`-Aktivität *c*, die per eingehenden Kontrollflusslink mit der `<reply>` *y* ersetzenden `<assign>`-Aktivität verbunden ist, müssen wir nun dafür sorgen, dass nach dem Eintreffen verschiedener sich gegenseitig ausschließender Antwortnachrichten der Kontrollfluss mit *s•* korrekt synchronisiert wird. Hierzu ersetzen wir zunächst *y* und *z* durch die entsprechenden `<assign>`-Aktivitäten *a‘* sowie a^{Fault}, die im ersten Fall v_{Back1} nach v_{Reply} sowie im Fehlerfall v_{Back2} nach v_{Reply} kopieren (vgl. analog Ersetzen der `<reply>`-Aktivität in SyncPattern1.1-1.2). Als Nachfolgeaktivität von *a* wird nun eine `<flow>`-Aktivität *fl* in *seq* erzeugt, die zwei `<throw>`-Aktivitäten enthält. Wir wandeln nun in `<scope>` *s* den ursprünglichen WSDL-Fault `xxx:Fault1` in einen benutzerdefinierten gleichnamigen Fault um, der im Fehlerfall von einem entsprechenden Fault Handler in *s* aufgefangen wird. Dieser Fault Handler enthält *A1*. Ein weiterer Fault Handler behandelt den benutzerdefinierten Fault `xxx:FaultExit`, der die `<empty>`-Aktivität *c* enthält. *a‘* wird nun per ausgehenden und in die `xxx:FaultExit` werfende `<throw>`-Aktivität eingehenden Kontrollflusslink verbunden. Entsprechend wird a^{Fault} mit der `xxx:Fault1` werfenden `<throw>`-Aktivität verbunden.
Zur Laufzeit werden zunächst die Aktivitäten in *•s* von *Scope_PBD1* ausgeführt und anschließend *s*. *a* kopiert v_{Send} nach $v_{Receive}$ und setzt anschließend den Kontrollfluss in *Scope_PBD2* fort. Nachdem hier die Aktivitäten in *•r* ausgeführt wurden, schreitet die Ausführung mit *b* und anschließend *r••y* voran. Tritt hier ein `bpel:joinFailure`-Fault auf, wird dieser im entsprechenden Fault Handler von *Scope_PBD2* gefangen und a^{Fault} kopiert v_{Back2} nach v_{Reply}. Tritt ein Fehler auf der nicht gefangen wird, so war die *PBD* schon vor der Konsolidierung fehlerhaft und hätte einen `bpel:missingReply`-Fault hervorgerufen. Nachdem a^{Fault} beendet wurde, schreitet der Kontrollfluss mit der `<throw>`-Aktivität voran, die den benutzerdefinierten Fault `xxx:Fault1` wirft. Dieser wird im entsprechenden Fault Handler von *s* aufgefangen und Aktivität *A1* wird ausgeführt. Anschließend wird *s* beendet und die ausgehenden Links von *s* werden ausgewertet, falls nicht $succl(s)=\emptyset$ gilt. Danach folgen die Aktivitäten aus *s•*. Tritt kein Fehler auf so kopiert *a‘* v_{Back1} nach v_{Reply} und der Kontrollfluss schreitet nach Auswertung der ausgehenden Links von *a‘* mit der Ausführung der Aktivitäten aus *y•* sowie parallel der `<throw>`-Aktivität, die den benutzerdefinierten Fault `xxx:FaultExit` wirft in *Scope_PBD1* voran. Dieser wird im entsprechenden Fault Handler von *s* aufgefangen und *s* anschließend beendet. Die anschließende Ausführung erfolgt analog zum Fehlerfall. Mit dem gezeigten Erkennungs- und Ersetzungsmechanismus lassen sich alle ursprünglichen WSDL-Fault werfenden `<reply>`-Aktivitäten in entsprechende Kombinationen aus `<assign>`- und `<throw>`-Aktivitäten im neuen `<scope>` *s* umwandeln und der Kontrollfluss wird entsprechend per Links zwischen diesen verbunden. Der eingeführte benutzerdefinierte Fault `xxx:FaultExit` mit `<empty>`-Aktivität im entsprechenden Fault Handler von *s* dient nur zum Beenden des entsprechenden `<scope>` und anschließender Weiterführung des Kontrollflusses in *s•*.
Neben der Möglichkeit mehrerer `<reply>`-Aktivitäten für *s* bedingt durch WSDL-Faults, kann eine derartige Konstellation auch durch Verzweigungen im Kontrollfluss in der antwortsendenden PBD entstehen. Abbildung 3.65 zeigt einen solchen Fall. Die beiden PBDs aus vorigem Beispiel kommunizieren synchron miteinander. Der Übersichtlichkeit halber wurde auf Fault sowie Compensation Handler in *s* verzichtet, das vorgestellte Konsolidierungsmuster ist jedoch auch in Kombination mit dem zuvor erklärten auf solche `<invoke>`-Aktivitäten anwendbar. Auf die `<receive>`-Aktivität *r* folgt eine synchronisierende `<empty>`-Aktivität *c*, die je nach Auswertung der `<transition-Condition>`s ihrer ausgehenden Links den Kontrollfluss in `<reply>` y_1 oder y_2 fortsetzt.

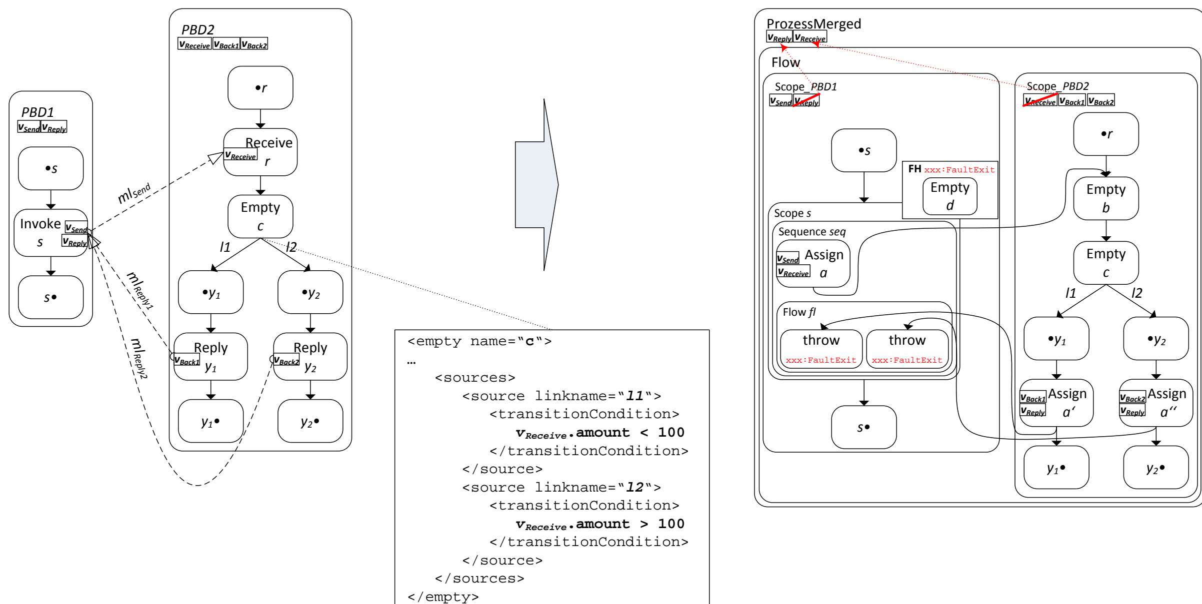


Abbildung 3.65 SyncPattern1.4 mit zwei `<reply>`-Aktivitäten

Die Konsolidierung erfolgt analog zum vorherigen Beispiel nur das hier in der `<flow>`-Aktivität *fl* für jede der beiden y_1 sowie y_2 ersetzenden `<assign>`-Aktivitäten *a'* und *a''* eine `<throw>`-Aktivität für das Werfen des benutzerdefinierten Faults `xxx:FaultExit` hinzugefügt wird, die per Kontrollflusslink jeweils mit *a'* bzw. *a''* verbunden wird. Mit diesem Fault Mechanismus wird das Warten auf die Auswertung der beiden Links zwischen *a'* bzw. *a''* und entsprechender `<throw>`-Aktivität umgangen. Sobald eine der beiden `<assign>`-Aktivitäten ausgeführt wurde, wird `<scope>` *s* im Anschluß beendet und der Kontrollfluss schreitet in den Aktivitäten *s*• voran.

3.3.2.5 SyncPattern1.5 (Sendende `<invoke>`-Aktivität innerhalb von Handlern)

Befindet sich die synchron kommunizierende `<invoke>`-Aktivität *s* innerhalb eines Handlers (FH→ Fault Handler, CH→Compensation Handler, TH→Termination Handler, EH→Event Handler), so bietet das SyncPattern1.5 in bestimmten Konstellationen der empfangende PBD eine Möglichkeit der Konsolidierung. Wie schon im AsyncPattern3.0 erklärt dürfen Links mit Ursprung in FHs sowie THs nur ausgehend und in CHs sowie EHs weder ein- noch ausgehend sein. Da im synchronen Fall von *s* die Ausführung der Nachfolgeraktivitäten solange blockiert wird bis die Antwortnachricht eingegangen ist und wir bei den vorangegangenen synchronen Konsolidierungsmustern den Kontrollfluss in die empfangende und antwortsendende PBD leiten und wieder herausführen, können wir diese in den Kontrollfluss der Aktivitäten, die sich in einem Handler befinden integrieren. Abbildung 3.66 zeigt die Beispielfragmente einer Choreographie sowie die Konsolidierung, die diese Idee umsetzt. Die synchrone `<invoke>`-Aktivität *s* befindet sich innerhalb eines Handlers (FH, CH, TH oder EH). Diese sendet Nachrichten an die `<receive>`-Aktivität *r* in der PBD *PBD2*. *PBD2* schickt die Antwortnachricht per `<reply>`-Aktivität *y* an *s*. Wichtig an dieser Konfiguration ist, dass *r* keine Vorgängeraktivitäten hat somit das `createInstance`-Attribut auf „`yes`" gesetzt ist und dass *y* keine Nachfolgeraktivitäten hat ($\bullet r=\emptyset$ sowie $y\bullet=\emptyset$). Zusätzlich muss gelten, dass es nur Nachfolgeaktivitäten von *r* gibt, die auf einem direkten Pfad zu *y* liegen und keine weiteren ($r\bullet\!\!\blacktriangle y=\emptyset$). Da *s* den Kontrollfluss in der empfangenden PBD *PBD2* initiert und dieser nach dem Senden der Antwortnachricht in *y* wieder beendet wird, erzeugen wir im neuen konsolidierten Prozess im Handler der *s* enthält eine `<flow>`-Aktivität *fl* in die wir *s*•, *s* sowie •*s* inklusive aller zugehörigen Kontrollflussrelationen hineinkopieren. Anschließend wir *Scope_PBD2* in *fl* kopiert und die Aktivitäten *s*, *r* und *y* werden durch dieselben

Aktivitäten, wie schon in SyncPattern1.1 ersetzt (*s* durch `<assign>` *a*, *r* durch `<empty>` *b* sowie *y* durch `<assign>` *a'*) und der Kontrollfluss wird per Links entsprechend hergestellt. Da in der ursprünglichen Choreographie *s* solange blockiert bis die Antwort aus *y* eintrifft, haben wir auf diese Weise den ursprünglichen Kontrollfluss erhalten. Hätte *r* noch Vorgänger, so würde der Handler nach dem inkludieren von *Scope_PBD2* einen anderen Kontrollfluss haben und könnte erst ausgeführt werden, nachdem alle Vorgängeraktivitäten beendet wurden. Hätte *y* Nachfolger, so wäre der Handler auch nach *a'* noch aktiv und der Kontrollfluss wäre abhängig von diesen Nachfolgeaktivitäten. Gäbe es Nachfolgeraktivitäten von *r* die auf keinem direkten Pfad zu *y* liegen, so könnten diese den Kontrollfluss des Handlers nach dem Konsolidieren ebenfalls verändern, da in der ursprünglichen Choreographie die Aktivitäten in *s•* direkt nach Erhalt der Antwortnachricht ausgeführt werden und der Handler anschließend beendet wird unabhängig von möglichen parallelen Aktivitäten in *r••y*. Auch hier wird zusätzlich auf das Vorhandensein eines `<catchAll>`-Fault Handlers geprüft und bei Bedarf ein neuer mit einer `<compensate>`-Aktivität angelegt, um alle möglichen Faults, die nicht in *Scope_PBD2* gefangen werden am Durchdringen in den Handler zu hindern. Der Compensation Handler wird mit einer `<empty>`-Aktivität e_1 definiert. Dies hat folgenden Grund: Ein Prozessscope hat keinen CH und *Scope_PBD2* ersetzt den Prozessscope der ehemaligen PBD *PBD2*. Würden wir keinen derartigen CH definieren, so würde während der Ausführung automatisch ein Default-CH installiert werden und könnte den ursprünglichen Kontrollfluss verändern (vgl. [OAS07] Abschnitt 12.5.1 *Default Fault, Compensation, and Termination Handlers*).

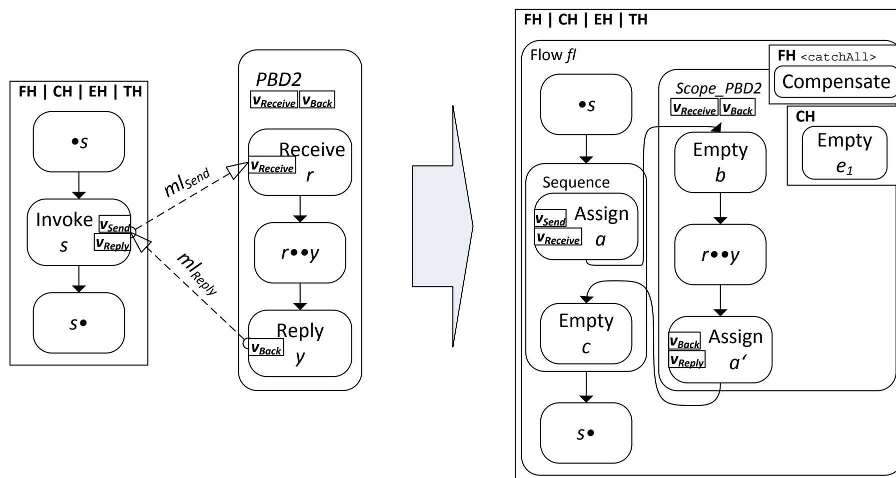


Abbildung 3.66 SyncPattern1.5

3.3.2.6 SyncPattern2.1 (`<onMessage>`-Zweig als `receiveActivity`)

Das SyncPattern2.1 stellt einen Spezialfall des SyncPattern1.1 dar, indem statt einer `<receive>`-Aktivität *r* ein `<onMessage>`-Zweig einer `<pick>`-Aktivität *p* als `receiveActivity` im Message Link ml_{Send} aus *ML* definiert wurde. In diesem Fall kann es passieren, dass die Antwort aufgrund einer Deaktivierung der entsprechenden Aktivität *p* durch eine zuvor eintreffende Nachricht eines anderen `<onMessage>`-Zweigs dieser ausbleibt. Der Kontrollfluss bleibt in diesem Fall in der sendenden Aktivität *s* hängen. Abbildung 3.67 zeigt die Fragmente einer Beispielchoreographie zweier PBDs *PBD1* und *PBD2*, die über eine synchrone `<invoke>`-Aktivität *s* in *PBD1*, einen `<onMessage>`-Zweig mit der `wsu:id` *msg1* einer `<pick>`-Aktivität *p* sowie eine `<reply>`-Aktivität *y* in *PBD2* miteinander kommunizieren. Das Ersetzen von *p* durch eine entsprechende `<scope>`-Aktivität *Scope_Pick* erfolgt analog zu AsyncPattern2.1. Die *s* ersetzende `<sequence>`-Aktivität auf der ehemaligen Senderseite wird um eine *a* umgebende `<if>`-Aktivität erweitert. *a* wird nur ausgeführt, wenn *p* noch nicht aktiviert wurde (vgl. AsyncPattern2.1 mit Schutzvariable $v_{pick_activated}$). *y* wird durch die übliche `<assign>`-Aktivität *a'* ersetzt und mit einem ausgehenden Kontrollflusslink mit *c* verbunden. Wird bei der Ausführung `<receive>` *msg2* durch eine eingehende Nachricht aktiviert bevor *a* ausgeführt

wurde und im Anschluss den entsprechenden benutzerdefinierten Fault `xxx:FaultExit` geworfen hat, so bleibt der Kontrollfluss bei der Ausführung der `<sequence>`-Akivität in *Scope_PBD1* hängen, wie in der ursprünglichen Choreographie. Dieses Konsolidierungsmuster lässt sich auch im Fall weiterer `<onMessage>`- bzw. möglicher `<onAlarm>`-Zweige einsetzen (vgl. AsyncPattern2.1) sowie entsprechend beim Auftreten mehrerer Choreographieteilnehmer die auf verschiedene `<onMessage>`-Zweige von *p* senden (vgl. AsyncPattern2.2 sowie AsyncPattern2.3).

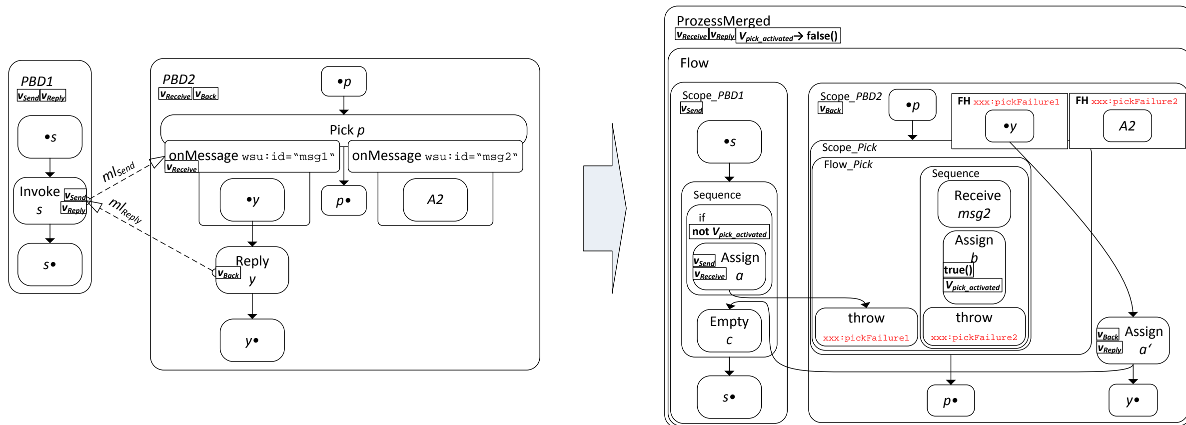


Abbildung 3.67 SyncPattern2.1

3.3.2.7 SyncPattern2.2

Das SyncPattern2.2 stellt den Sonderfall dar, dass $\bullet p=\emptyset$ gilt und das `createInstance`-Attribut auf „`yes`" gesetzt ist. Hierbei erfolgt die Konsolidierung analog zu SyncPattern2.1 mit dem Unterschied, dass die `<receive>`-Aktivitäten im *Scope_Pick* den Wert des `createInstance`-Attributs von *p* erben (vgl. AsyncPattern2.3).

3.3.2.8 SyncPattern2.3

Gilt wie im SyncPattern1.3 $s\bullet=\emptyset$ und wird ein `<onMessage>`-Zweig einer `<pick>`-Aktivität *p* auf Empfängerseite eingesetzt, so können wir auch hier die *s* ersetzende `<sequence>`-Aktivität durch eine `<assign>`-Aktivität austauschen, die lediglich v_{Send} nach $v_{Receive}$ kopiert, da die Daten der Antwortnachricht nicht mehr verwendet werden. *p* wird analog zu SyncPattern2.1 mit einer entsprechenden `<scope>`-Aktivität ersetzt und *y* durch eine synchronisierende `<empty>`-Aktivität (vgl. SyncPattern1.3).

3.3.2.9 SyncPattern2.4

Das SyncPattern2.4 erkennt und konsolidiert die synchronen Kommunikationsmuster, in denen *s* auf *p* sendet und gleichzeitig mehrere mögliche `<reply>`-Aktivitäten vorliegen. Somit gibt es einen Message Link ml_{Send} mit *s* als `sendActivity` und die `wsu:id` eines `<onMessage>`-Zweigs von *p* als `receiveActivity` und mehrere weitere Message Links $ml_{Reply1\text{-}n}$ bzw. $ml_{ReplyFault}$ mit verschiedenen `<reply>`-Aktivitäten y_1-y_n als `sendActivity` und jeweils *s* als `receiveActivity`. Wir werden in folgendem Unterabschnitt nur die Änderungen an der *s* ersetzenden Aktivität vorstellen, da die `<reply>`-Aktivitäten im antwortsendenden `<scope>`, wie schon zuvor, durch entsprechende

<assign>-Aktivitäten, die jeweils v_{Back} nach v_{Reply} kopieren, ausgetauscht werden (vgl. SyncPattern1.4).

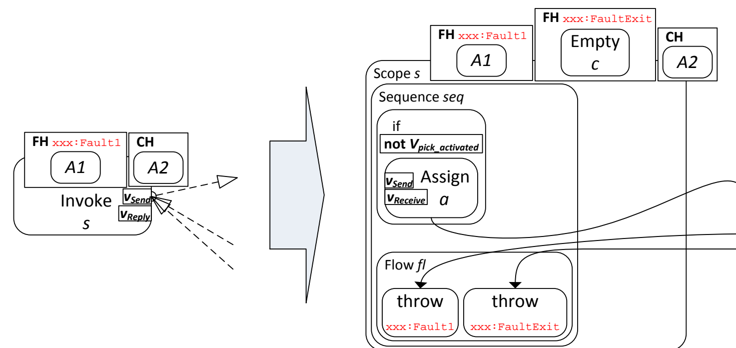


Abbildung 3.68 SyncPattern2.4

Abbildung 3.68 skizziert anhand von Beispielfragmenten das Ersetzen der synchron kommunizierenden <invoke>-Aktivität *s*, die auf einen <onMessage>-Zweig einer <pick>-Aktivität sendet (nicht abgebildet) und eine Antwortnachricht von mehreren möglichen <reply>-Aktivitäten (ebenfalls nicht abgebildet) erhält. *s* wird, wie in SyncPattern1.4, durch eine <scope>-Aktivität *s* ausgetauscht, die wiederum eine <sequence>-Aktivität *seq* enthält. Im Gegensatz zum SyncPattern1.4 ist nun die <assign>-Aktivität *a* durch die <if>-Aktivität zur Überprüfung der Schutzvariable $v_{pick_activated}$ umgeben, da hier auf einen <onMessage>-Zweig gesendet wird (vgl. SyncPattern2.1). Wie in SyncPattern1.4 wird auch hier eine weitere <flow>-Aktivität *fl* in *seq* hinzugefügt, die für mögliche WSDL-Fault sendende <reply>-Aktivitäten jeweils eine <throw>-Aktivität enthält. Diese wirft den entsprechenden umgewandelten benutzerdefinierten Fault. Für jede <reply>-Aktivität, die ohne Fault antwortet ist eine <throw>-Aktivität enthalten. Diese wirft den benutzderdefinierten Fault `xxx:FaultExit`, der *s* beendet und die Ausführung der Aktivitäten in *s*• bewirkt. Im gezeigten Beispiel gibt es somit eine <reply>-Aktivität *y*, die in der ursprünglichen Choreographie ohne Fault antwortet und eine weitere y_{Fault}, die den WSDL-Fault `xxx:Fault1` zurückliefert.

3.3.2.10 SyncPattern3.0 („Non-Merge-Pattern-Sync“)

Analog zum AsyncPattern3.0 fängt das SyncPattern3.0 die synchronen Kommunikationsmuster ab, die nicht durch einen der zuvor erwähnten Erkennungs- und Konsolidierungsmechanismen ersetzt werden können bzw. das Ziel zukünftiger Erweiterungen des Algorithmenkatalogs sind. Trifft der Erkennunsalgorithmus auf ein hier genanntes Muster, so werden die entsprechenden Message Links aus *ML* in die Liste *NMML* hinzugefügt (vgl. AsyncPattern3.0) und die betreffenden Aktivitäten im Anschluss in intra-prozess kommunizierende umgewandelt.

3.3.2.10.1 <onEvent>-Zweig (EH) als empfangende Aktivität

Da wir in der vorliegenden Arbeit die Konsolidierung synchroner Kommunikationen in denen <onEvent>-Zweige eines EHs als `receiveActivity` auftauchen nicht unterstützen, wird der entsprechende Message Link ml_{Send} aus *ML* in die Liste *NMML* übertragen. Zusätzlich suchen wir für die `sendActivity` *s* von ml_{Send} nach weiteren antwortsendenden Message Links, die *s* als `receiveActivity` enthalten und übertragen diese ebenfalls in *NMML*. Dieser Schritt ist notwendig, da bei synchroner Kommunikation immer mindestens ein antwortsendender Message Link ml_{Reply} vorliegen muss und wir von einer korrekten BPEL4Chor-Choreographie als Eingabe ausgehen.

3.3.2.10.2 Sendende <invoke>-Aktivität innerhalb von Handlern (EH, FH, TH, CH)

Befindet sich die synchron kommunizierende `<invoke>`-Aktivität s eines Message Links ml_{Send} innerhalb eines EH, FH, TH oder CH und es gilt zusätzlich für die empfangende `<receive>`-Aktivität r sowie die möglichen antwortsendenden `<reply>`-Aktivität y_1-y_n der entsprechenden Message Links ml_{Reply1}-ml_{Replyn} $\bullet r \neq \emptyset$ und/oder $y_1\bullet \neq \emptyset$ oder $y_2\bullet \neq \emptyset$ oder … $y_n\bullet \neq \emptyset$, dass heisst r hat Vorgängeraktivitäten und mindestens eine der Aktivitäten y_1-y_n haben Nachfolger im Kontrollfluss, so werden alle an der Kommunikation involvierten Message Links in *NMML* übertragen. Zusätzlich müssen alle Aktivitäten aus $r\bullet$ auf einem direkten, nichtverzweigenden Pfad zu den jeweiligen `<reply>`-Aktivitäten y_1-y_n liegen ($r\bullet \cap \bullet y_1=\emptyset$,…, $r\bullet \cap \bullet y_n=\emptyset$). Andernfalls können wir das SyncPattern1.5 anwenden.
Handelt es sich bei der empfangenden Aktivität um den `<onMessage>`-Zweig einer `<pick>`-Aktivität, so werden ebenfalls alle an der Kommunikation involvierten Message Links in *NMML* übertragen.

3.3.2.10.3 Empfangende <receive>/<pick>-Aktivität innerhalb von Handlern (EH, CH, TH, FH)

Für die empfangende `<receive>`- oder `<pick>`-Aktivität r gilt analog zu Abschnitt 3.3.1.12.3 der Asynchronen Merge-Patterns: Befindet sich r innerhalb eines EHs, CHs, FHs oder THs, so wird der entsprechende Message Link ml_{Send} aus *ML*, der diese Aktivität als `receiveActivity` enthält, in die Liste *NMML* übertragen, da SA00070 sowie SA00071 des WS-BPEL 2.0 Standards [OAS07] eingehende Links in jegliche Art der vier Handler verbieten. Zusätzlich werden alle antwortsendenden Message Links ml_{Reply1}-ml_{Replyn}, die die `sendActivity` s aus ml_{Send} als `receiveActivity` enthalten ebenfalls in *NMML* übertragen.

3.3.2.10.4 Sendende und/oder empfangende Aktivitäten innerhalb von Schleifen

Analog zu Abschnitt 3.3.1.12.4 der asynchronen Merge-Patterns werden sendende und empfangende Aktivitäten, die sich innerhalb von Schleifen befinden nicht konsolidiert und verbleiben als intraprozess kommunizierende im Prozess. Wird ein Message Link ml_{Send} entdeckt dessen `sendActivity` s und/oder `receiveActivtiy` r sich innerhalb einer Schleife befinden, wird er in die Liste *NMML* übertragen. Anschließend werden alle antwortsendenden Message Links ml_{Reply1}-ml_{Replyn}, die s als `receiveActivity` enthalten ebenfalls in *NMML* übertragen.

3.4 Vervollständigung der technischen Artefakte im neuen konsolidierten Prozess und Übernahme der WSDLs

Nachdem alle Message Links aus *ML* behandelt und die entsprechenden konsolidierbaren Kombinationen aus kommunizierenden Aktivitäten durch die jeweiligen Konstrukte ersetzt wurden, müssen die technischen Informationen in den verbliebenen kommunizierenden Aktivitäten durch Analyse der Topology, Grounding sowie den zugehörigen WSDL-Dateien in den neuen konsolidierten Prozess übernommen werden. Hierzu gehört das Anlegen der entsprechenden `partnerLinks`, das Hinzufügen dieser sowie des `operation`- und `portType`-Attributs in die jeweiligen Aktivitäten, als auch die Anpassung der Korrelationsmengen im Falle mehrerer initialer Startaktivitäten (vgl. Abschnitt 3.2.3.2.1 Mehrere initiale Startaktivitäten). Zusätzlich werden die vorhandenen WSDL-Dateien als `imports` in den Prozess eingefügt. Ein weiterer Optimierungsschritt zukünftiger Arbeiten könnte das Verschmelzen der WSDL-Dateien zu einer einzigen sein. Da dieser strukturelle Aspekt jedoch keine Auswirkungen auf die Laufzeit des ausführbaren Prozesses hat, belassen wir die technischen Informa-

tionen in den jeweiligen WSDL-Dateien und binden dieses stattdessen über mehrere `import`-Statements in den BPEL-Prozess ein.

3.4.1 Einfügen der WSDL-Dateien per `import`-Statements

Für jede PBD der Choreographie ist eine entsprechende WSDL-Datei vorhanden, die nun per `import`-Statement in den konsolidierten BPEL-Prozess hinzugefügt wird.

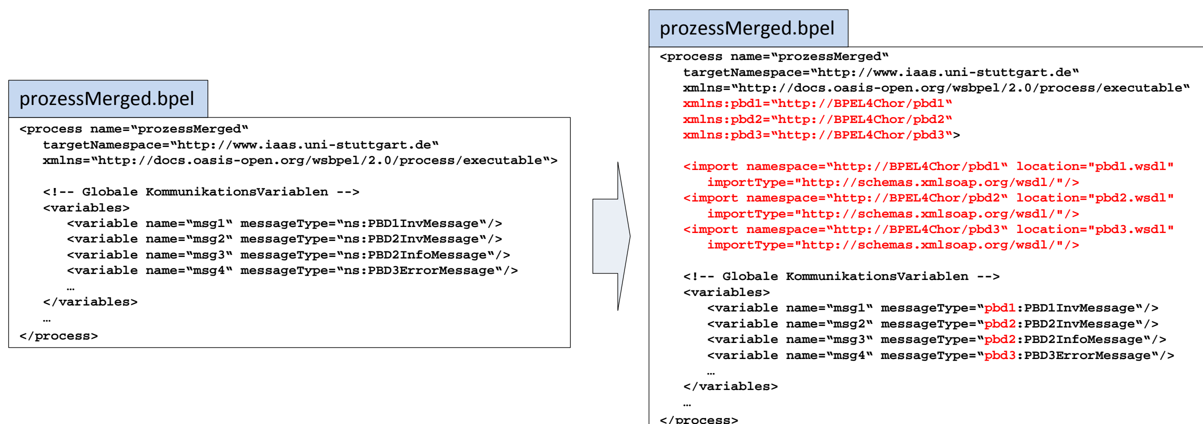


Abbildung 3.69 Hinzufügen der WSDL-`import`s in den konsolidierten Prozess

Abbildung 3.69 veranschaulicht die Schritte beim Hinzufügen der `imports`: Die drei WSDL-Dateien werden in den neuen Prozess *ProzessMerged* eingefügt sowie die Namespace-Präfixe der entsprechenden globalen Variablen angepasst. Diese Variablen sind beim Konsolidieren durch Übertragen der an einer Kommunikation teilnehmenden Daten durch das entsprechende Merge-Pattern in den globalen Prozessscope entstanden ($v_{Receive}$, v_{Reply}, etc.). Anschließend werden alle Namespace-Präfixe in den involvierten `<scope>`-Aktivitäten ebenfalls an den jeweiligen Namespace angepasst (Alle Deklarationen innerhalb *Scope_PBD1*, …, *Scope_PBDn*).

3.4.2 Anpassung der Korrelationsmengen bei mehreren initialen Startaktivitäten

Nun untersuchen wir die ersten Aktivitäten im Kontrollfluss in *Scope_PBD1-Scope_PBDn* des konsolidierten Prozesses. Handelt es sich um `<receive>`s oder `<pick>`s mit dem auf `yes` gesetzten `createInstance`-Attribut und befinden sich in mehr als einer der *Scope_PBD1-Scope_PBDn*-Aktivitäten derartige initiale Aktivitäten, so müssen wir, wie in Abschnitt 3.2.3.2.1 beschrieben, die Nachrichten der dort verwendeten Variablen um eine gemeinsame Korrelationsmenge erweitern.
Angenommen in den beiden `<scope>`-Aktivitäten *Scope_PBD1* sowie *Scope_PBD2* des konsolidierten Prozesses *ProzessMerged* befinden sich jeweils zwei initiale `<receive>`-Aktivitäten r_1 und r_2, wobei r_1 die Variable `inputPBD1` vom `messageType` `pbd1:PBD1RequestMessage` und r_2 die Variable `inputPBD2` vom `messageType` `pbd2:PBD2RequestMessage` verwendet (die Variablen können hier auch beide denselben Namen haben, da diese in verschiedenen `<scope>`s deklariert wurden). Nun werden die entsprechenden Messages in den beiden WSDL-Dateien pbd1.wsdl sowie pbd2.wsdl erweitert: In beiden Messages wird ein neuer `wsdl:part` mit dem Namen `commonCorrelProperty` vom Typ `xsd:string` hinzugefügt sowie die entsprechenden `property`-Eigenschaften definiert (`vprop:property` und `vprop:propertyAlias` vgl. Abbildung 3.28b). Anschließend werden diese `propertys` zu den Korrelationsmengen von r_1 und r_2 hinzugefügt und das `initiate`-Attribut beider wird auf „`join`“ gesetzt.

3.4.3 Erzeugen und Hinzufügen der `PartnerLinks` für die nicht konsolidierten Message Links aus *NMML*

Da die vorliegende Arbeit nicht alle vorkommenden Kommunikationsmuster- und Konstellationen in Merge-Patterns umsetzt, kann es vorkommen, dass einige der choreographie-intern kommunizierenden Aktivitäten der Message Links aus *ML* nicht durch die entsprechenden Synchronisationsaktivitäten ersetzt werden und die entsprechenden Message Links in *NMML* übertragen werden. Nun müssen diese Aktivitäten zu intra-prozess kommunizierenden im konsoldidierten Prozess *ProzessMerged* umgewandelt werden. Hierzu werden wir die Message Links aus *NMML* mit den entsprechenden zugehörigen Message Links aus der Grounding-Datei abgleichen und die technischen Informationen (`portType` sowie `operation`) aus der WSDL-Datei der PBD in die `PartnerLinks` sowie die involvierten Aktivitäten einfügen. Wir zeigen zunächst den Fall für asynchron intra-prozess kommunizierende Aktivitäten und die entsprechenden Message Links. Diese sind dadurch charakterisiert, dass ihre `sendActivity` *s* in keinem weiteren Message Link als `receiveActivity` auftaucht.

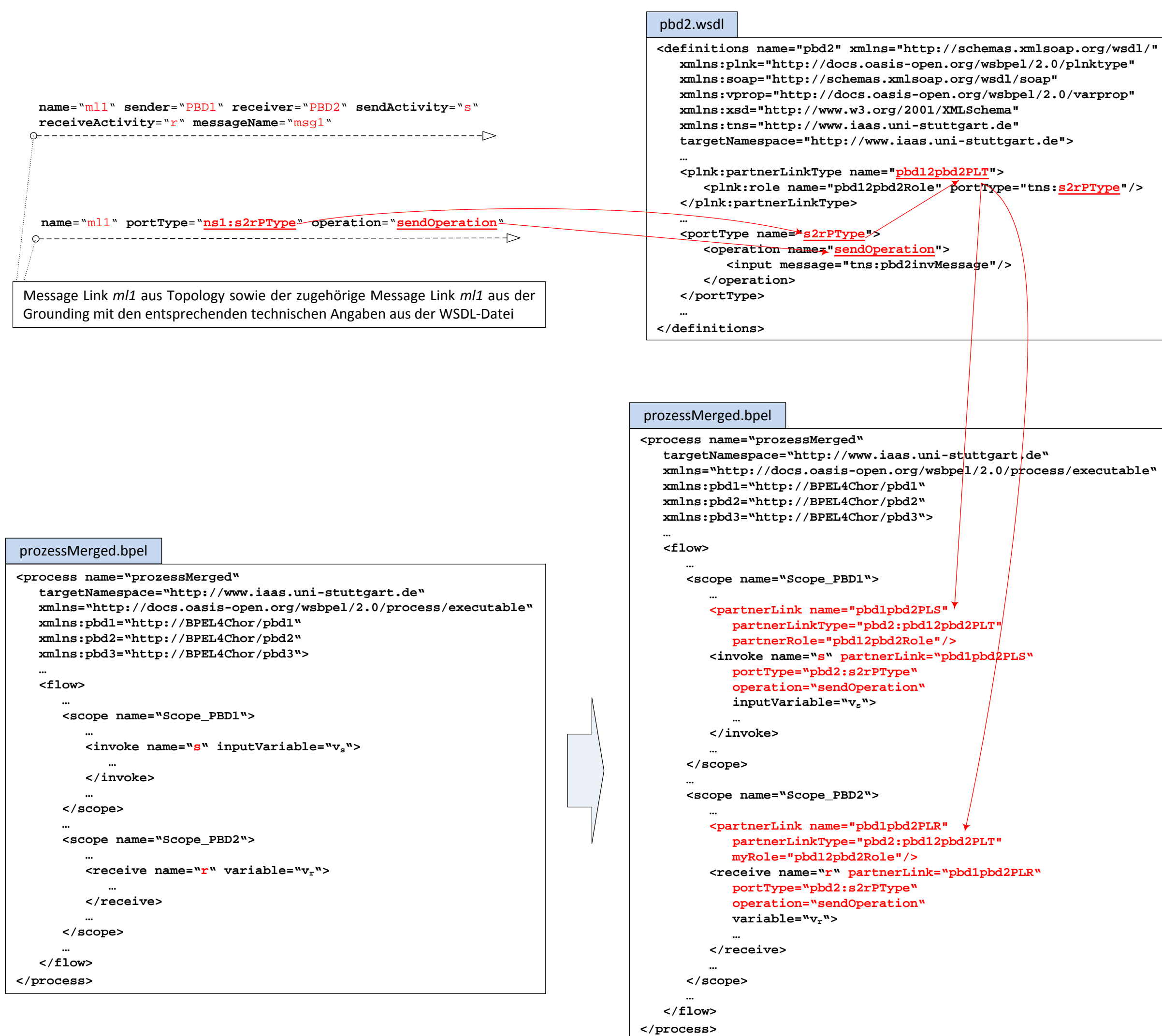


Abbildung 3.70 Hinzufügen der `PartnerLinks` **sowie der technischen Attribute** `portType`**,** `operation` **sowie** `partnerLink` **in die asynchron intra-prozess kommunizierenden Aktivitäten**

Abbildung 3.70 veranschaulicht das Vorgehen an einem Beispiel: Die beiden Aktivitäten *s* und *r* verbleiben nach der Konsolidierung als intra-prozess kommunizierende im Prozess ProzessMerged. Nachdem der Message Link *ml1* aus *NMML* mit dem zugehörigen *ml1* aus der Grounding-Datei

abgeglichen und die zugehörigen technischen Artefakte in der WSDL-Datei ausfindig gemacht wurden, werden zwei neue `PartnerLinks` in den beiden `<scope>`s *Scope_PBD1* sowie *Scope_PBD2* mit den passenden `partnerLinkTypes` angelegt und die technischen Attribute `portType`, `operation` und `partnerLink` in die beiden Aktivitäten *s* und *r* eingefügt. Dieses Vorgehen wird für alle verbleibenden asynchron intra-prozess kommunizierende Message Links aus *NMML* wiederholt und die entsprechenden Artefakte angelegt und in die beteiligten Aktivitäten hinzugefügt.
Für die synchron intra-prozess kommunizierenden Aktivitäten *s*, *r* und *y* müssen mindestens zwei Message Links ml_{Send} sowie ml_{Reply} aus *NMML* untersucht und die entsprechenden Artefakte hinzugefügt werden (vgl. Abbildung 3.71).

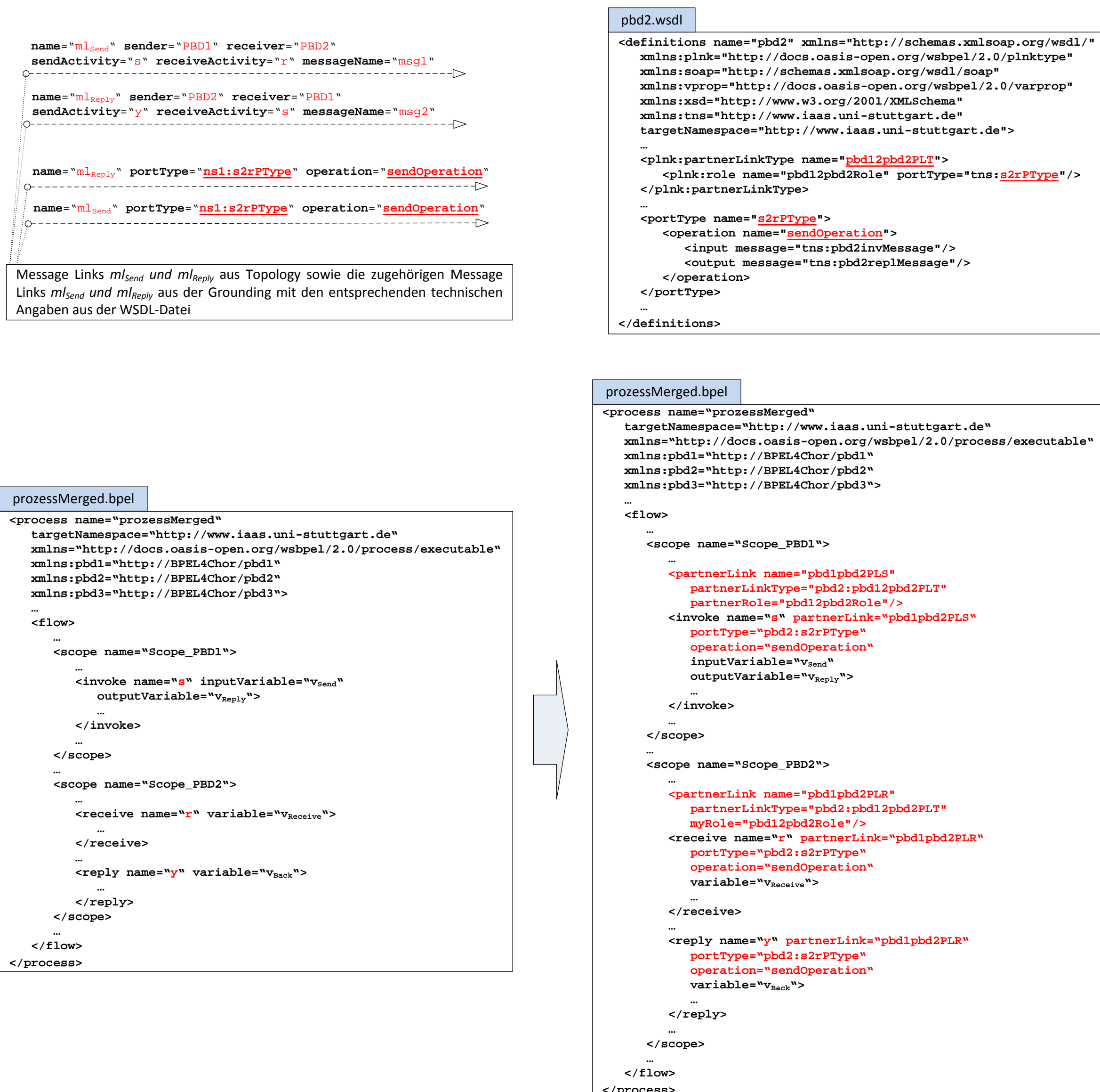


Abbildung 3.71 Hinzufügen der technischen Artefakte bei synchron intra-prozess kommunizierenden Aktivitäten

3.4.4 Technische Vervollständigung der initialen Startaktivitäten sowie der inter-prozess kommunizierenden

Da eine BPEL4Chor-Choreographie und die in der Grounding-Datei definierten Message-Links nur die choreographie-intern kommunizierenden Aktivitätenpaare mit technischen Artefakten assoziiert, müssen in einem letzten Vervollständigungsschritt die initialen `<receive>`- oder `<pick>`-Aktivitäten sowie die möglichen antwortsendenden des neuen konsolidierten Prozesses manuell vervollständigt werden (Hinzufügen der `PartnerLinks` sowie der `partnerLink`-, `operation`-, `portType`-Attri-

bute). Dies gilt auch für die ursprünglich choreographie-extern kommunizierenden Aktivitäten, die nun im konsolidierten Prozess als inter-prozess kommunizierende vorliegen.

4 Implementierung

In folgendem Kapitel werden wir die zur Umsetzung des Eclipse-Plugins zur Konsolidierung verwendeten Technologien sowie die Architektur der Implementierung kurz vorstellen.

4.1 Eingesetzte Technologien

Die eingesetzten Frameworks und Technologien wurden mit dem zum Zeitpunkt der Fertigstellung der vorliegenden Diplomarbeit neusten verfügbaren JDK 1.7.0.13 (Java Development Kit) sowohl in der 32-Bit als auch in der 64-Bit Variante implementiert und getestet.

4.1.1 StAX

Streaming API for XML (StAX) ist ein Application Programming Interface (API), um XML-Dateien aus Java zu verarbeiten. Es bietet einen Mittelweg zwischen dem Einlesen einer XML-Datei in eine Baumstruktur, wie beispielsweise das DOM (Document Object Model), und dem ereignisbasierten in dem ein Ereignis beim Auffinden eines XML-Elements in der lesenden Java-Anwendung ausgelöst wird, wie es beispielsweise SAX (Simple API for XML) unterstützt. StAX bietet stattdessen eine Cursor-basierte Variante, in der sich die Anwendung bei Bedarf über einen beweglichen Zeiger die entsprechenden Daten vom unterliegenden Parser in Form von String-Objekten holt ohne das gesamte XML-Dokument einlesen zu müssen. Zusätzlich bietet es eine Iteratorvariante in der die gelesenen Daten in Form von Objekten, die von der Klasse XMLEvent abgeleitet sind, der Anwendung zur Verfügung gestellt werden. Das von Cui in [CUI12] implementierte BPEL4Chor-Model wurde mit Hilfe von StAX um das Einlesen der Topology- sowie Grounding-Datei in das von ihm entwickelte BPEL4-Chor-Modell-Format erweitert.

4.1.2 Eclipse IDE

Die Eclipse IDE [ECL12] ist eine integrierte Entwicklungsumgebung, die mit dem Schwerpunkt der Unterstützung zur Java-Entwicklung konzipiert wurde und mittlerweile durch ihre Erweiterbarkeit für viele weitere verschiedene Entwicklungsaufgaben eingesetzt wird. Für Eclipse steht eine Vielzahl an quelloffenen als auch kommerziellen Erweiterungen (Plug-Ins) zur Verfügung. Das in der vorliegenden Arbeit entwickelte Konsolidierungsplugin wurde mit und für die Eclipse IDE Java EE Version 4.2 Service Release 1 entwickelt und getestet.

4.1.3 Eclipse Modeling Framework (EMF)

Das Eclipse Modeling Framework [EEMF12] ist ein Framework zur Generierung von Java-Code aus strukturierten Datenmodellen. Der generierte Code kann Instanzen eines Modells erzeugen, verändern, einlesen, validieren sowie serialisieren. Das EMF wird zum Einlesen, Transformieren sowie anschließendem Serialisieren der PBDs einer BPEL4Chor-Choreographie in einen ausführbaren BPEL-Prozess verwendet. Hierzu wird das EMF-Modell für BPEL-Artefakte [EBPELM] verwendet um eine möglichst einfache Navigierbarkeit innerhalb einer PBD sowie den entsprechenden Teilen im neuen konsolidierten Prozess während der Transformation zu gewährleisten.

4.2 Vorgehen und Architektur

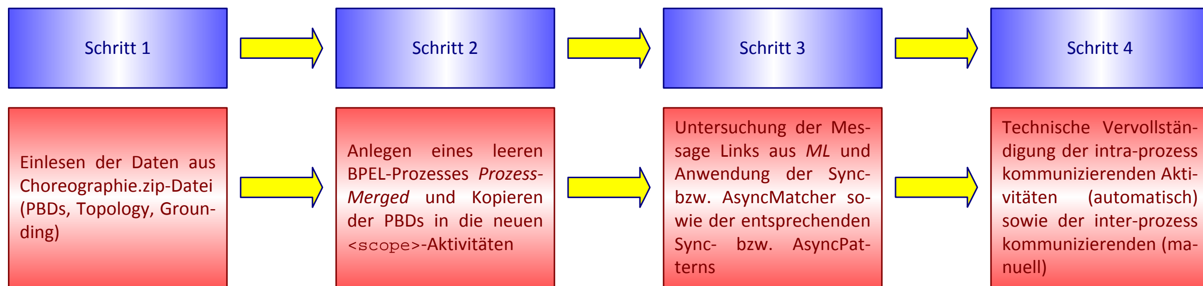


Abbildung 4.1 Schritte beim Vorgehen der Konsolidierung einer BPEL4Chor-Choreographie

Abbildung 4.1 fasst die Schritte beim Vorgehen der Konsolidierung einer BPEL4Chor-Choreographie nochmals zusammen: Zunächst wird die Choreographie-Zip-Datei eingelesen und in den entsprechenden Datenstrukturen gespeichert. Diese enthält die beteiligten PBDs, die Topology- sowie die Grounding-Datei. Anschließend wird ein neuer BPEL-Prozess *ProzessMerged* angelegt und die PBDs werden mit allen Daten in `<scope>`-Aktivitäten in diesen Prozess kopiert. Nach diesem Schritt folgt die eigentliche Konsolidierung, in dem die Message Links aus *ML* nach ihrem Kommunikationsmuster untersucht werden und der entsprechende Async- bzw. SyncMatcher mit dem passenden Async- bzw. SyncPattern angewendet wird. Gibt es für einen Message Link *ml* kein passendes Konsolidierungsmuster, so wird *ml* in die Liste *NMML* (*Non-Mergeable-Message-Links*) übertragen. Als letzter Schritt folgt die automatische technische Vervollständigung der intra-prozess kommunizierenden Aktivitäten im *ProzessMerged*, die durch die Message Links aus *NMML* identifiziert werden. Hierzu werden die technischen Konfigurationen der beteiligten Aktivitäten über die Message Links der Grounding-Datei mit den entsprechenden Informationen aus den jeweiligen WSDL-Dateien assoziiert. Die technischen Konfigurationen der inter-prozess kommunizierenden Aktivitäten im *ProzessMerged* müssen manuell erfolgen, da hierzu keine Assoziationen in der Grounding-Datei vorliegen (Diese enthält nur Angaben zu den choreographie-intern kommunizierenden Aktivitäten, die über die Message Links verbunden sind). Zu diesen Aktivitäten gehören die initialen Startaktivitäten, die den Lebenszyklus des neuen Prozesses *ProzessMerged* anstoßen, mögliche antwortsendende Aktivitäten, die dem Aufrufer entsprechende Nachrichten senden sowie die ehemals choreographie-extern kommunizierenden Aktivitäten, die nun inter-prozess kommunizierende sind.

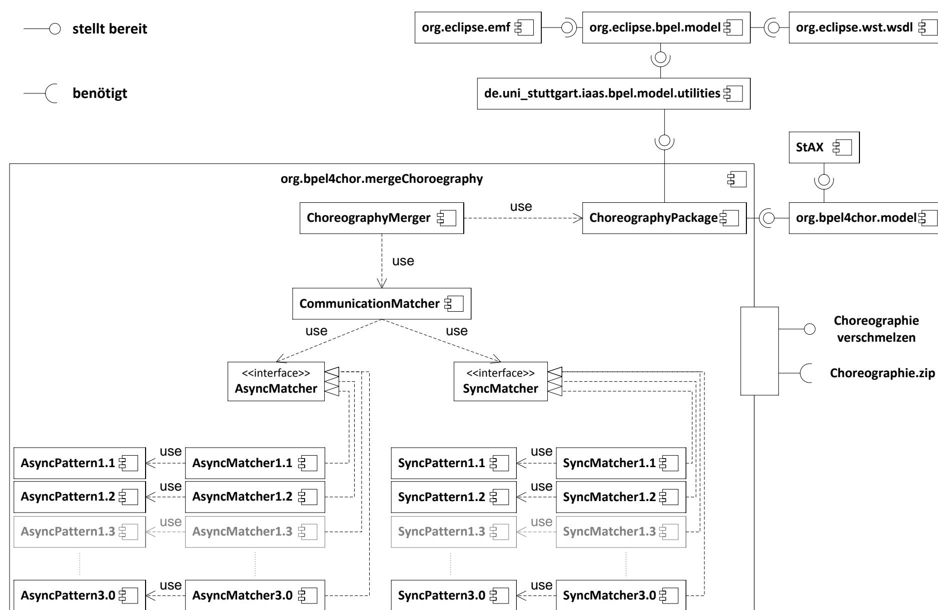


Abbildung 4.2 Die mergeChoreography-Komponente und ihre Abhängigkeiten

Abbildung 4.2 zeigt die für die Konsolidierung implementierte Komponente und ihre Abhängigkeiten zu anderen Komponenten: Die Komponente *mergeChoreography* bietet die Schnittstelle *mergeChoreography* an, die zum Konsolidieren einer BPEL4Chor-Choreographie dient. Die für die Konsolidierung benötigten Daten stellt eine Choreographie-Zip-Datei dar, die die entsprechenden PBDs, die Topology- sowie die Grounding-Datei enthält. Die Daten werden aus der Zip-Datei eingelesen und in der Komponente ChoreographyPackage gespeichert. Diese verwendet das von Cui in [CUI12] entwickelte und implementierte BPEL4Chor-Modell (*org.bpel4chor.model*), das wiederum zum Einlesen und Ausgeben des Modells *StAX* als API verwendet. Zusätzlich verwendet das *ChoreographyPackage* die in *de.uni_stuttgart.iaas.bpel.model.utilities* implementierten Hilfsmethoden zum Traversieren und Analysieren von BPEL-, PBD- sowie der zugehörigen WSDL-Dateien. Diese Hilfsmethoden basieren wiederum auf dem Eclipse BPEL-Modell (*org.eclipse.bpel.model*), das seinerseits auf dem *EMF* sowie dem *Eclipse Web Standard WSDL Tools* basiert (*org.eclipse.wst.wsdl*). Nachdem die Daten aus der Zip-Datei eingelesen wurden, wird die *ChoreographyMerger*-Komponente zum Anlegen des neuen BPEL-Prozesses *ProzessMerged* im *ChoroegraphyPackage* sowie dem anschließenden Analysieren der Message Links aus *ML* verwendet. Diese benutzt hierfür die Komponente *CommunicationMatcher*, die je nach Kommunikationsmuster des betrachteten Message Links aus *ML* die Liste der *AsyncMatcher*-Schnittstelle-implementierenden Klassen im asnychronen Fall oder die Liste der *SyncMatcher*-Schnittstelle-implementierenden Klassen im synchronen Fall traversiert und das passendste Merge-Pattern für diesen Message Link zurückliefert. Jede *SyncMatcher*- bzw. *AsyncMatcher*-Implementierungsklasse bietet hierfür ein eigenes *Sync*- bzw. *AsyncPattern* zur Konsolidierung an. Wird das *Async*- bzw. *SyncPattern3.0* als Merge-Pattern gefunden, so wird der Message Link im asynchronen bzw. werden die Message Links im synchronen Fall in die Liste *NMML* übertragen.

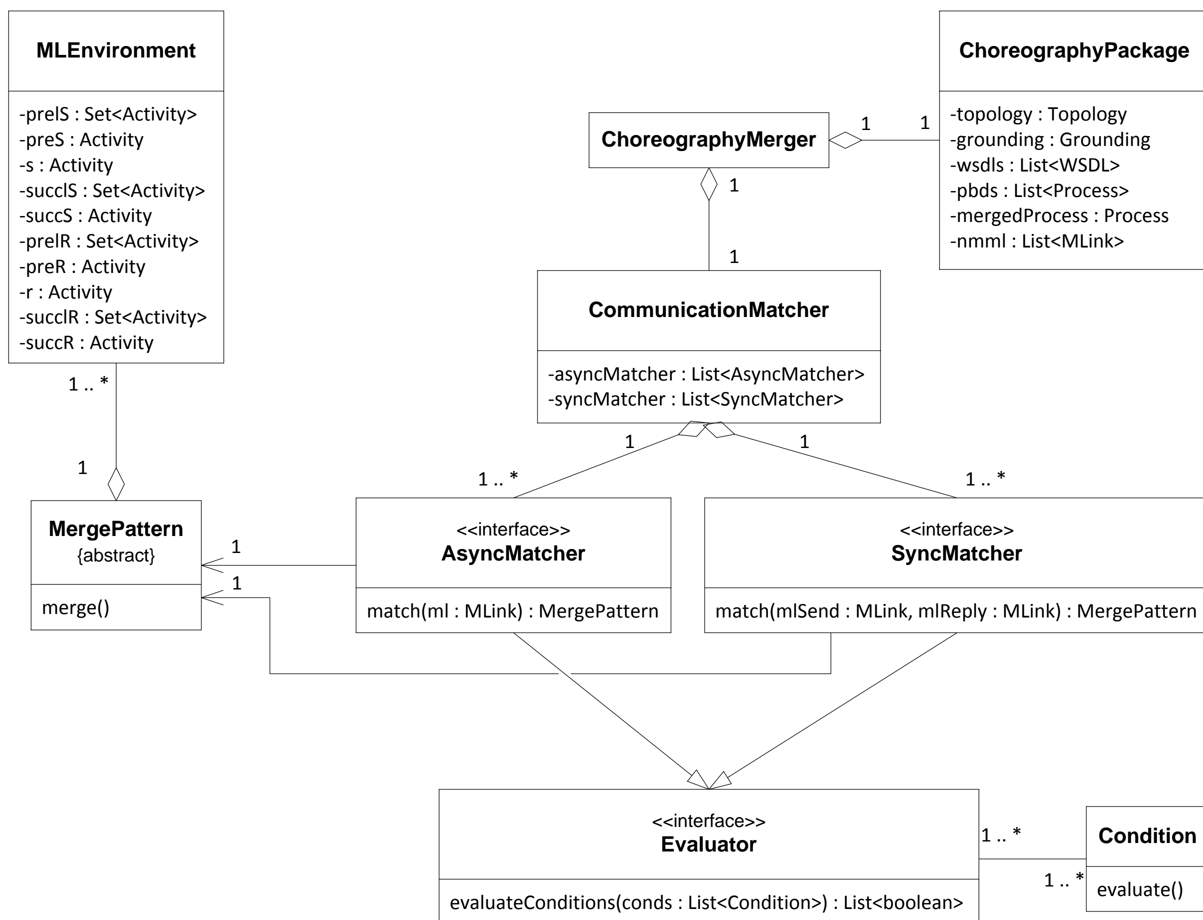


Abbildung 4.3a ChoreographyMerger-Klassendiagramm und die von dieser genutzten Klassen aus dem Paket org.bpel4chor.mergeChoreography

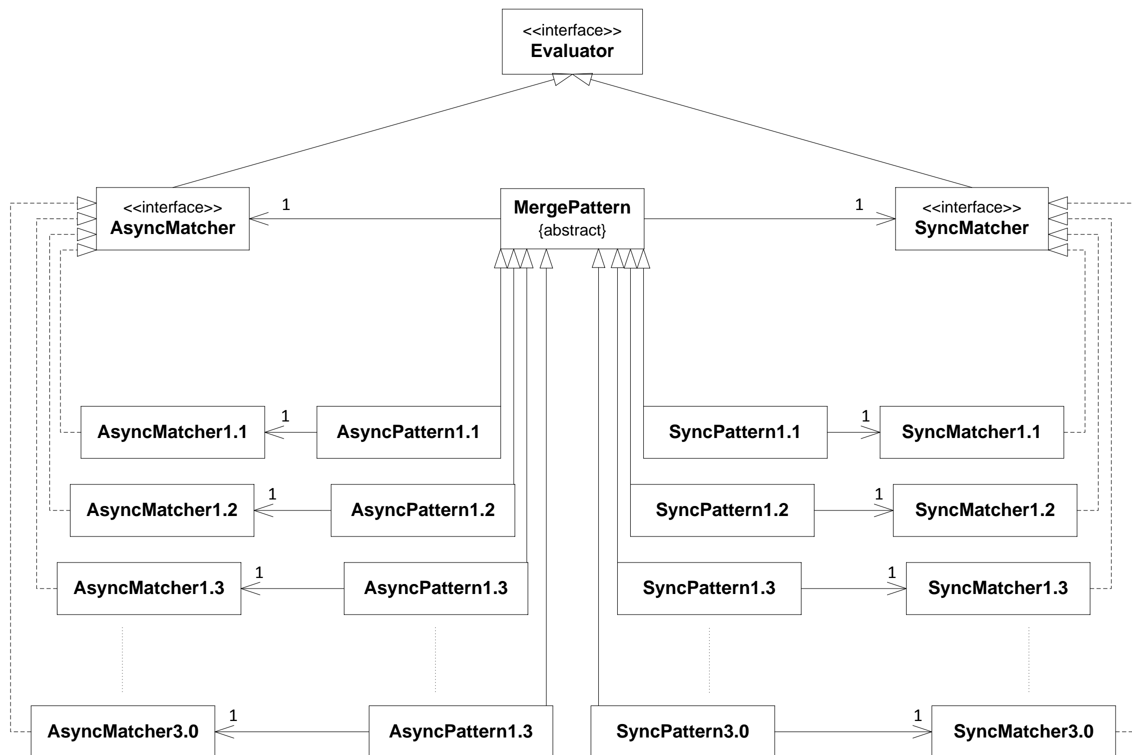


Abbildung 4.3b Async- bzw. SyncMatcher sowie die Beziehungen zu den Async- bzw. SyncPatterns und den entsprechenden Schnittstellen (Interfaces) und der abstrakten Klassse MergePattern

Abbildung 4.3a zeigt das Klassendiagramm des *ChoreographyMerger* sowie das Zusammenspiel mit den anderen Klassen in der Komponente *org.bpel4chor.mergeChoreography*. Der *ChoreographyMerger* enthält eine Instanz des *ChoreographyPackage*, welches die Daten der eingelesen BPEL4Chor-Choreographie sowie den neuen *mergedProzess* beinhaltet. Diese Daten werden vom *CommunicationMatcher* verwendet, der über die in der Topology enthaltenen Message Links iteriert und je nach asynchronem oder synchronem Kommunikationsmuster die Liste der *asyncMatcher* bzw. *syncMatcher* nach einem passendem Erkennungsmuster untersucht. Die Liste *asyncMatcher* enthält Klassen, die das *AsyncMatcher*-Interface implementieren, welches zusätzlich das *Evaluator*-Interface erweitert. Das *Evaluator*-Interface hat folgende Funktion: Es kann vorkommen, dass ein Kommunikationsmuster, das in einem Message Link und den involvierten PBDs sowie Aktivitäten verwendet wird durch die Erkennungsmuster mehrerer Matcher erkannt wird. So ist zum Beispiel das AsyncPattern1.2 ein Spezialfall des AsyncPattern1.1, daher könnten hier beide Matcher angewendet werden. Jedoch ist das AsyncPattern1.2 durch die zusätzliche Bedingung definiert, dass für die Menge $r\bullet = \emptyset$ gilt, es somit keine Nachfolgeaktivitäten auf die empfangende Aktivität gibt. Daher enthält jeder Matcher die zusätzliche Methode *evaluateConditions*, die eine Liste aus booleschen Bedingungen als Eingabe erhält und eine Liste der Übereinstimmungen als boolesche Ergebniswerte zurückliefert. Je mehr Bedingungen eine derartige Eingabeliste enthält umso länger ist die Liste der Ergebniswerte der Analyse. Sind alle Werte der Ergebnisliste `true` und ist die Liste länger als die entsprechende Liste aller möglichen weiteren Erkennungsmusters, so ist der Matcher der passendste für diesen Message Link. Der gefundene Matcher liefert anschließend ein *Async*- bzw. *SyncPattern* zurück, das die *merge*-Methode der abstrakten Klasse *MergePattern* implementiert. Die Klasse *MergePattern* enthält eine, im asynchronen Fall, oder mehrere, im synchronen Fall, Instanzen der Klasse *MLEnvironment*, die Mengen aller umgebenden Aktivitäten der kommunizierenden eines Message Links beinhaltet. Hierzu gehören die Mengen: *prelS*→Die Menge aller Vorgängeraktivitäten der sendenden Aktivität *s*, die mit dieser über die eingehenden Links verbunden sind, *succlS*→Die Menge aller Nachfolgeraktivitäten der sendenden Aktivität *s*, die mit dieser über die ausgehenden Links verbunden sind, *preS*→Die direkte Vorgängeraktivität von *s*, die nicht über einen eingehenden Link mit *s* verbunden ist, deren Ausführung jedoch vor der Aktivierung von *s* beendet sein muss, *succS*→Die direkte Nachfolgeraktivität von *s*, die nicht über einen ausgehenden Link mit *s* verbunden ist, die jedoch erst nach der Ausführung von *s* aktiviert

wird sowie *prelR*, *succlR*, *preR* und *succR* mit den entsprechenden Eigenschaften für die empfangende Aktivität *r*. Diese Mengen und Aktivitäten werden für die anschließende strukturelle Transformation mit den MergePatterns benötigt.
Abbildung 4.3b zeigt die implementierten *Async*- bzw. *SyncMatcher* sowie die zugehörigen *Async*- bzw. *SyncPattern* und ihre Ableitungs- und Implementierungsbeziehungen zu den Interfaces sowie der abstrakten Klasse *MergePattern*.

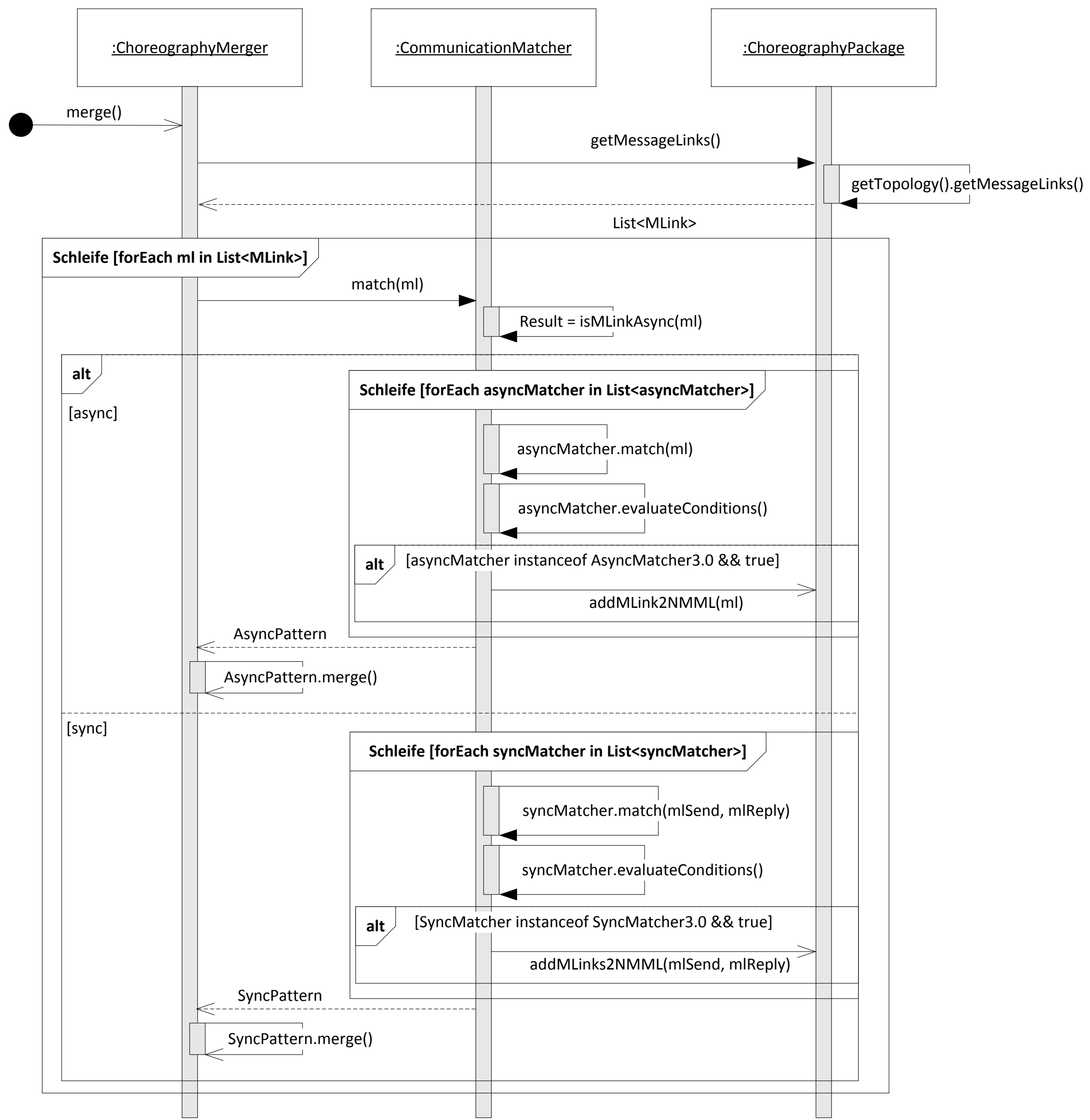


Abbildung 4.4 Sequenzdiagramm für das Auffinden der MergePatterns für die Message Links aus *ML*

Abbildung 4.4 zeigt das Sequenzdiagramm der Suche über die verfügbaren Erkennungsmuster in den Async- bzw. SyncMatchern für die Message Links aus *ML*: Je nach Kommunikationsmuster eines jeden Message Links *ml*, asynchron oder synchron, wird die entsprechende Liste der Matcher durchlaufen und die Erkennungsmuster überprüft. Wird ein Erkennungsmuster der „*Non-Mergeable-Pattern-Async/Sync*" entdeckt (Async- bzw. SyncPattern3.0) so wird der entsprechende Message Link in die Liste *NMML* eingefügt. Andernfalls wird das passendste *MergePattern* zurückgeliefert und dieses anschließend über den Aufruf der *merge*-Methode angewendet.

4.3 Erweiterbarkeit der Patterns

Eines der Ziele der vorliegenden Diplomarbeit war den Katalog der Async- sowie SyncPattern erweiterbar zu entwerfen und zu implementieren. Durch die Kombination aus den Matcher-Schnittstellen implementierenden Erkennungsmusterklassen sowie den zugehörigen Ableitungen der MergePattern-Klasse muss hierfür für neue Muster jeweils eine entsprechende Matcher-Klasse sowie das zugehörige MergePattern je nach Kommunikationsmuster (asynchron oder synchron) implementiert werden. Die vorliegende Arbeit und die zugehörige Implementierung sind derart konfiguriert, dass neue Matcher-Klassen, die im Paket *org.bpel4chor.mergechoreography.matcher.communication.async/sync* gespeichert werden, automatisch vom *CommunicationMatcher* geladen werden. Werden neue Patterns für bestimmte Konfigurationen und Konstellationen, die durch die beiden *Non-Mergeable-Pattern-Async/ Sync*" (Async- bzw. SyncPattern3.0) abgefangen werden implementiert, so müssen zusätzlich die Bedingungen dieser aus der Liste der *Condition*s der entsprechenden Matcher entfernt werden, da der Async- bzw. SyncMatcher3.0 in der jeweiligen Liste des *CommunicationMatcher* als erstes Element eingefügt wird, um so beim Auffinden eines entsprechenden Musters die nicht konsolidierbaren Message Links direkt in *NMML* einzufügen.

5 Zusammenfassung und Ausblick

Das Ziel der vorliegenden Diplomarbeit war die Implementierung und Erweiterung der in [WKL11] vorgestellten Konzepte zur Konsolidierung einer BPEL4Chor-Choreographie in Form eines Eclipse-Plugins. Ausgangsbasis und Eingabe hierfür ist eine BPEL4Chor-Choreographie [DKLW07] mit den entsprechenden Fragmenten: Die PBDs, die den Kontroll- und Datenfluss modellieren, die Topology-Datei, die über die `ParticipantTypes` sowie die `Participants` die Verbindung zu den PBDs herstellt und die Kommunikation der beteiligten Teilnehmer in Form von Message Links darstellt sowie die Grounding-Datei, die technische Artefakte über die jeweiligen Message Links und die involvierten Aktivitäten mit den zusätzlichen Informationen aus den WSDL-Dateien assoziiert. Das Ergebnis der Konsolidierung ist ein ausführbarer BPEL-Prozess, der die gleiche Kontrollflusssemantik der ursprünglichen Choreographie beibehält, jedoch die choroegraphie-intern kommunizierenden Aktivitäten durch Kombinationen aus entsprechenden Kopier- und Synchronisationaktivitäten ersetzt, um so eine bessere Laufzeitperformance im Hinblick auf die Anzahl der verwendeten Prozessinstanzen sowie einen reduzierten Kommunikationsaufwand durch das Vermeiden des SOAP-Messaging zu gewährleisten.

In Kapitel 3 wurde hierfür das Vorgehen beim Konsolidieren genauer erklärt: Zunächst wird die BPEL4Chor-Choreographie und die zugehörigen WSDL-Dateien mit den technischen Artefakten eingelesen und in den entsprechenden Datenstrukturen gespeichert. Anschließend wird ein neuer leerer BPEL-Prozess angelegt und die PBDs inklusive aller enthaltener Datenstrukturen in neue `<scope>`-Aktivitäten in diesen Prozess kopiert (Abschnitt 3.2.1 Anlegen des konsolidierten BPEL-Prozesses). Im darauffolgenden Abschnitt wurde die Idee zur Generierung des Kontrollflusses im neuen konsolidierten Prozess aus den ursprünglichen PBDs und ihrer Kommunikationsbeziehungen erklärt (Abschnitt 3.2.2 Generierung des Kontrollflusses). Hierzu wurden für die asynchrone sowie die synchrone Kommunikation jeweils zwei Varianten vorgestellt und die jeweiligen Vor- und Nachteile aufgezeigt und nachgewiesen, dass die Kontrollflussrelationen nach der Konsolidierung erhalten bleiben. Zusätzlich wurden die Anpassungen aufgezeigt, die an den `<transitionCondition>`s sowie den `<joinCondition>`s der beteiligten Aktivitäten durchgeführt werden müssen.
In Abschnitt 3.2.3 Generierung des Datenflusses wurde die Idee der Ersetzung der choreographie-intern kommunizierenden Aktivitäten durch entsprechende Kombinationen aus `<assign>`- und synchronisierenden `<empty>`-Aktivitäten zur Emulierung des ursprünglichen Nachrichtenflusses vorgestellt sowie die Auswirkungen der Konsolidierung auf die involvierten Korrelationsmengen (Abschnitt 3.2.3.2 Auswirkungen der Konsolidierung auf die verwendeten CorrelationSets).
Im Abschnitt 3.3 Taxonomie der Konsolidierungsmuster („Merge-Patterns“) wurden die Erkennungs- und Konsolidierungmuster für asynchron sowie synchron kommunizierende Aktivitätenpaare vorgestellt. Hierzu wurden zunächst die asynchronen Merge-Patterns ausgehend von den Allgemeinen hin zu den Speziellen präsentiert (Abschnitt 3.3.1 Asynchrone Merge-Patterns AsyncPattern1.1-AsyncPattern3.0) und anschließend entsprechend die synchronen Merge-Patterns (Abschnitt 3.3.2 Synchrone Merge-Patterns SyncPattern1.1-SyncPattern3.0).
In Abschnitt 3.4 Vervollständigung der technischen Artefakte im neuen konsolidierten Prozess und Übernahme der WSDLs wurden die Schritte vorgestellt, die zum Konfigurieren des neuen ausführbaren Prozesses notwendig sind.

Im Kapitel 4 Implementierung wurden die in der Umsetzung verwendeten Technologien und Frameworks kurz beschrieben sowie ein Überblick über die Architektur des Eclipse-Plugins gegeben. Zusätzlich wurde der Aspekt der Erweiterbarkeit in Bezug auf die hier implementierte Lösung erläutert und Einstiegspunkte in der Architektur für mögliche Anpassungen der bereits vorhandenen sowie neuer Merge-Patterns aufgezeigt.

5.1 Ausblick

Die in dieser Arbeit nicht behandelten Konsolidierungmuster sind Ziel zukünftiger Arbeiten. Hierzu gehören die asynchronen Kommunikationsmuster zwischen choreographie-intern kommunizierenden Partnern, die durch das AsyncPattern3.0 abgefangen werden und nach der Konsolidierung als intra-prozess kommunizierende Aktivitätenpaare vorliegen:

`<onEvent>`-Zweige einer `<scope>`-Aktivität oder des Prozessscope: Wie in Abschnitt 3.3.1.12.1 beschrieben, kann auch die `wsu:id` eines `<onEvent>`-Zweigs als `receiveActivity` in einem Message Link *ml* aus *ML* auftauchen.

Asynchron kommunizierende `<invoke>`-Aktivität innerhalb von Handlern (EH, CH, FH, TH): Abschnitt 3.3.1.12.2 zeigt die nicht implementierten Fälle in denen sich die asynchron kommunizierende `<invoke>`-Aktivität *s* innerhalb von CHs oder EHs befindet. Auch das Auftreten einer solchen Konstellation von *s* als `sendActivity` in *ml* wird in dieser Arbeit nicht konsolidiert. Eine Ausnahme bilden hier die FHs sowie THs, da diese ausgehende Links enthalten dürfen, jedoch nur solche, die nicht in den zugehörigen `<scope>` zeigen.

Empfangende `<receive>`/`<pick>`-Aktivität innerhalb von Handlern (EH, CH, FH, TH): Abschnitt 3.3.1.12.3 zeigt die Fälle in denen sich die `receiveActivity` *r*, die entweder eine `<receive>`-Aktivität oder ein `<onMessage>`-Zweig einer `<pick>`-Aktivität sein kann, innerhalb eines Handlers befindet. Diese Fälle werden aus den gleichen Gründen wie im vorherigen Abschnitt nicht konsolidiert mit dem Zusatz, dass auch FHs und THs nicht behandelt werden, da diese keine eingehenden Links enthalten dürfen.

Sendende und/oder empfangende Aktivitäten innerhalb von Schleifen: Befinden sich die `receiveActivity` und/oder die `sendActivity` innerhalb einer Schleife, so werden diese Aktivitäten ebenfalls als intra-prozess kommunizierende in den neuen konsolidierten Prozess übernommen (vgl. Abschnitt 3.3.1.12.4)

Für die synchron kommunizierenden Partner einer Choreographie gelten dieselben Voraussetzungen für die Konsolidierung der entsprechenden Aktivitäten bzw. die Übernahme dieser als intra-prozess kommunizierende in den neuen konsolidierten Prozess. Die durch das SyncPattern3.0 abgefangenen Fälle beinhalten die in Abschnitt 3.3.2.10 erwähnten Konstellationen.

Literaturverzeichnis

[AL83] J.F. Allen. *Maintaining Knowledge about Temporal Intervals*. Communications of the ACM Volume 26 Issue 11. pp. 832-843, Nov. 1983.

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

[BDH05] A. Barros, M. Dumas, A. H. M. T. Hofstede. *Service Interaction Patterns*. In W.M.P. van der Aalst, editor, Proceedings of the 3rd International Conference on Business Process Management (BPM 2005), volume 3649 of Lecture Notes in Computer Science, pp. 302-318. Springer-Verlag.

[BFG05] M. Benedikt, W. Fan, and F. Geerts. *XPath satisfiability in the presence of DTDs*. In Proceedings of the twentyfourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems – PODS ’05, pages 25–36. ACM, 2005. doi:10.1145/1065167.1065172.

[BN09] P.A. Bernstein, E. Newcomer. *Principles of Transaction Processing for the Systems Professional*. 2nd ed. Morgan Kaufmann, 2009.

[BPLG12] bpel-g. *BPEL 1.1 und WS-BPEL 2.0 konforme OpenSource BPEL-Engine*. Online: http://code.google.com/p/bpel-g/ .

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

[DKLW07] G. Decker, O. Kopp, F. Leymann, M. Weske. *BPEL4Chor: Extending BPEL for Modeling Choreographies*. pp. 296–303, 2007. doi:10.1109/ICWS.2007.59.

[DKLW09] G. Decker, O. Kopp, F. Leymann, M. Weske. *Interacting services: From specification to execution*. volume 68, pp. 946–972. Elsevier Science Publishers, 2009. doi:10.1016/j.datak.2009.04.003.

[DKP07] G. Decker, O. Kopp, F. Puhlmann. *Service Referrals in BPEL-based Choreographies*. Proceedings of the 2nd European Young Researchers Workshop on Service Oriented Computing (YR-SOC 2007). pp. 25-30, 2007.

[DP02] D. Davis and M. P. Parashar. *Latency performance of SOAP Implementations*. CCGRID '02 Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid. pp. 407, IEEE, 2002.

[EBPELM] Eclipse BPEL Model. Online: http://www.eclipse.org/bpel/developers/model.php .

[EBPL12] Eclipse BPEL Designer. *Eclipse BPEL Designer Plug-In*, Version 1.0.2. 2012. Online: http://www.eclipse.org/bpel/ .

[ECL12] Eclipse IDE. *Eclipse Entwicklungsumgebung*, Version 4.2 (Juno) sowie Version 3.7 (Indigo). Online: http://www.eclipse.org .

[EEMF12] Eclipse EMF. *Eclipse Modeling Framework*. Online:

http://www.eclipse.org/modeling/emf/ .

[KGF+08] J. Küster, C. Gerth, A. Förster, G. Engels. *A tool for process merging in business-driven development*. In Proceedings of the Forum at the CAiSE, 2008. Online: http://ceur-ws.org/Vol-344/paper23.pdf .

[KHA08] R. Khalaf. *Supporting business process fragmentation while maintaining operational semantics: a BPEL perspective*. Doctoral thesis, University of Suttgart, Factulty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2008.

[KHK+11] O. Kopp, S. Henke, D. Karastoyanova, R. Khalaf, F. Leymann, M. Sonntag, T. Steinmetz, T. Unger, B. Wetzstein. *An Event Model for WS-BPEL 2.0.* Report 2011, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Technischer Bericht Informatik.

[KKL+05] M. Kloppmann, D. Koenig, F. Leymann, G. Pfau, A. Rickayzen, C. von Riegen, P. Schmidt, I. Trickovic. *WS-BPEL Extension for Sub-Processes – BPEL-SPE*. A Joint White Paper by IBM and SAP, September 2005. Online: http://xml.coverpages.org/BPEL-SPE-Subprocesses.pdf .

[KKL08] O. Kopp, R. Khalaf, F. Leymann. *Deriving Explicit Data Links in WS-BPEL Processes*. In IEEE International Conference on Services Computing. IEEE, 2008.

[KL06] R. Khalaf, F. Leymann. *Role-based Decomposition of Business Processes using BPEL.* In International Conference on Web Services (ICWS 2006), pp. 770–780. IEEE Computer Society, 2006. doi:10.1109/ICWS.2006.56.

[KNS92] G. Keller, M. Nüttgens, A.-W. Scheer. *Semantische Prozeßmodellierung auf der Grundlage Ereignisgesteuerter Prozeßketten (EPK).* Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi),Universität des Saarlandes, Heft 89. Januar 1992.

[KOP11a] O. Kopp. *Grounding Syntax*. Email, 2011.

[KOP11b] O. Kopp. *Topology Syntax*. Email, 2011.

[KRL09] R. Khalaf, D. Roller, F. Leymann. *Revisiting the Behavior of Fault and Compensation Handlers in WS-BPEL.* On the Move to Meaningful Internet Systems: OTM 2009. R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part I, LNCS 5870, pp. 286–303. Springer Berlin Heidelberg 2009. doi: 10.1007/978-3-642-05148-7_20.

[LEN11] J. Lenhard. *A Pattern-based Analysis of WS-BPEL and Windows Workflow*. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 88, Bamberg University, March 2011. ISSN 0937-3349.

[LEY01] F. Leymann. *Web Services Flow Language* (WSFL 1.0). 22 Mai 2001. Online: http://xml.coverpages.org/wsfl.html

[LEY10a] F. Leymann. *Workflow Management I* (Vorlesung). Universität Stuttgart: 2010.

[LEY10b] F. Leymann. *Web Services I* (Vorlesung). Universität Stuttgart: 2010.

[MS06] J. Mendling, C. Simon. *Business process design by view integration*. J. Eder, S. Dustdar et al. (Eds.): BPM 2006 Workshops, LNCS 4103, pp. 55–64, 2006. Springer-Verlag Berlin Heidelberg. Online: http://mendling.com/publications/06-BPD.pdf .

[MT11] P. Mell, T. Grance. *The NIST Definition of Cloud Computing*. National Institute of Standards and Technology, vol. 53, no. 6, p. 50, September 2011. Online: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf .

[NCG04] A. Ng, S. Chen, and P. Greenfield. *An Evaluation of Contemporary Commercial SOAP Implementations*. In AWSA, 2004.

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

[OAS07b] OASIS. *Web Services Coordination (WS-Coordination) Version 1.1*, 16 April 2007. Online: http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.1-spec.pdf .

[OESB12] OpenESB. *OpenESB (vormals SUN OpenESB) OpenSource ESB*, Version 2.2. ESB mit BPEL-Engine (jedoch nicht vollständig WS-BPEL 2.0 konform, da keine Links unterstützt werden) sowie Netbeans 6.7.1 IDE mit BPEL-Designer. Online: http://logicoy.com/ESB.php .

[OMG10] OMG. *Unified Modeling Language (OMG UML), Superstructure, V2.3*. Object Management Group, May 2010. Online: http://www.omg.org/spec/UML/2.3/

[OMG11] OMG. *Business Process Model and Notation (BPMN) Version 2.0*. Januar 2011. Online: http://www.omg.org/spec/BPMN/2.0/PDF .

[PET62] Petri, C.A. *Kommunikation mit Automaten*. Dissertation. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. Online: http://www.informatik.uni-hamburg.de/TGI/mitarbeiter/profs/petri/doc/Petri-diss-de-d.pdf .

[SKY06] S. Sun, A. Kumar, J. Yen. *Merging workflows: A new perspective on connecting business processes*. Decision Support Systems, vol. 42, no. 2, pp. 844–858, 2006. doi: 10.1016/j.dss.2005.07.001.

[THA01] S. Thatte. *XLANG: Web Services for Business Process Design*. 6 Juni 2001. Online: http://xml.coverpages.org/xlang.html

[W3C01] W3C. *Web Services Description Language (WSDL) Version 1.1*, 15. März 2001. Online: http://www.w3.org/TR/wsdl .

[W3C05] W3C. *Web Services Choreography Description Language Version 1.0*, 9. November 2005. Online: http://www.w3.org/TR/ws-cdl-10/ .

[W3C07] W3C. *SOAP Version 1.2*. 27. April 2007. Online: http://www.w3.org/TR/soap12-part1/ .

[W3C12] W3C. *Extensible Markup Language (XML)*. Online: http://www.w3.org/XML/ .

[W3C99a] W3C. *XML Path Language (XPath) Version 1.0*. 16. November 1999. Online: http://www.w3.org/TR/xpath/ .

[W3C99b] W3C. *XSL Transformations (XSLT) Version 1.0*. 16 November 1999. Online: http://www.w3.org/TR/xslt .

[WADH03] P. Wohed, W.M.P. van der Alst, M. Dumas, A. H. M. T. Hofstede. *Analysis of Web Services Composition Languages: The Case of BPEL4WS*. In Proceedings of the 2003 International Conference on Conceptual Modeling (ER). pp. 200-215, 2003. doi: 10.1007/b13244.

[WCL+05] S,Weerawarana, F. Curbera, F. Leymann, T. Storey, D.F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005. pp 3–126 S. – ISBN 0131488740

[WDW07] M. Weidlich, G. Decker, M. Weske. *Efficient Analysis of BPEL 2.0 Processes Using π-Calculus*. apscc, pp.266-274, The 2nd IEEE Asia-Pacific Service Computing Conference (APSCC 2007), 2007. doi: 10.1109/APSCC.2007.36.

[WKL11] S. Wagner, O. Kopp, F. Leymann. *Towards Choreography-based Process Distribution In The Cloud*. Proceedings of the 2011 IEEE International Conference on Cloud Computing and Intelligence Systems. pp. 490-494, 2011. doi:10.1109/CCIS.2011.6045116.

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

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben. Wörtliche und sinngemäße Übernahmen aus anderen Quellen habe ich nach bestem Wissen und Gewissen als solche kenntlich gemacht.

Stuttgart, den 15. Februar 2013 ______________________