Ablaufkontrolle von Prozess-Choreographien

Dissertation

zur Erlangung des Doktorgrades
an der Fakultät für Mathematik, Informatik und Naturwissenschaften,
Fachbereich Informatik
der Universität Hamburg

vorgelegt von

Michael von Riegen

Hamburg, 2012

Erster Gutachter:	Prof. Dr.-Ing. Norbert Ritter
Zweiter Gutachter:	Prof. Dr.-Ing. Stefan Deßloch
Tag der Disputation:	09.08.2012

Kurzfassung

Um organisationsübergreifende Prozesse mit häufig heterogenen und autonomen Teilnehmern durch IT-Infrastrukturen zu unterstützen, werden heute hauptsächlich dienstorientierte Architekturen (SOA) implementiert. Im Bereich der SOA hat sich dafür das Konzept der Choreographie etabliert, welches Interaktionen zwischen Diensten aus Sicht eines idealen Beobachters beschreibt. Dieser kann alle Interaktionen zwischen Diensten sowie deren Daten- und Kontrollflussabhängigkeiten sehen. Halten sich alle Teilnehmer an die Choreographie, ist kein zentraler Koordinator notwendig, denn jeder Teilnehmer weiß aufgrund des Modells, wie er auf welche Nachrichten reagieren muss. Trotzdem bleibt die Frage, wie dabei sichergestellt werden kann, dass sich Teilnehmer auch zur Laufzeit so wie vorher spezifiziert verhalten und damit, ob die Prozess-Integritätsbedingungen einer Choreographie zur Laufzeit eingehalten werden. Herausfordernd ist dabei die Autonomie der Teilnehmer und damit, dass nur ein nicht-intrusiver Überprüfungsmechanismus infrage kommt, bei dem keine Software bei einem Teilnehmer verändert oder neu installiert werden muss.

Diese Arbeit stellt dafür einen neuen Ansatz vor, um das beobachtbare Verhalten der Teilnehmer einer Choreographie zu überwachen und Verletzungen von Integritätsbedingungen einer Choreographie zur Laufzeit zu verifizieren. Zunächst werden die hierfür benötigten Sensoren und deren Infrastruktur eingeführt und ein grundlegender Mechanismus der Erkennung von Integritätsverletzungen entwickelt. Dies wird durch ein formales Modell untermauert, das auf dem Ereigniskalkül basiert. Das Kalkül ermöglicht es durch zeitliches Schließen, Abweichungen von einem bestehenden Choreographie-Modell und damit Verletzungen der Integritätsbedingungen zur Laufzeit zu erkennen. Der vorgestellte Ansatz ist dabei in der Lage, nicht nur Integritätsbedingungen für einzelne Nachrichten, sondern auch Bedingungen für Interaktionen wie Zeitschranken, Bedingungen für einzelne Instanzen wie den gesamten Kontrollfluss einer Choreographie und auch instanzübergreifende Bedingungen zu verifizieren, die für den erfolgreichen Ablauf eingehalten werden müssen.

Werden Integritätsverletzungen durch den im ersten Teil der Arbeit entwickelten Ansatz entdeckt, stellt sich die Frage, wie weiter verfahren wird. In dieser Arbeit wird dafür ein Recovery-Ansatz auf Basis von Transaktionsprotokollen vorgestellt, welcher die Verifikationsergebnisse direkt in die Ablaufkontrolle von Transaktionen einbinden kann. Das vorgestellte Konzept erweitert die bisherigen Ansätze der transaktionalen Kontrolle von Prozessen um einen Koordinator, der aufgrund von Regeln autonom entscheiden kann, ob eine Transaktion erfolgreich beendet werden kann oder nicht. In bisherigen Ansätzen ist der Initiator einer Transaktion immer derjenige Teilnehmer, welcher über den Transaktionsabschluss entscheidet, was in Choreographie-Umgebungen nicht immer angenommen werden kann. Der in dieser Arbeit vorgestellte Koordinationsansatz ist zudem in der Lage, die Ergebnisse der Verifikation direkt in der Ablaufkontrolle einer Transaktion zu nutzen und dabei Choreographien effektiv bei der Fehlerbehebung bei Integritätsverletzungen zur Laufzeit zu unterstützen.

Abstract

Today, service-oriented architectures (SOA) can be used for supporting the implementation of cross-organizational workflows with autonomous and heterogeneous participants. In this context, the concept of choreographies is well established, introducing a new viewpoint on interacting services. A choreography model describes interactions from the perspective of an ideal observer who is able to see all interactions and their flow and data dependencies. If all participants adhere to a choreography model, no centralized workflow coordinator is required because every participant knows how to react on messages due to the choreography model. Nevertheless, the question how to assure that participant abide the choreography model and behave like specified in beforehand remains and thus, if the process integrity constraints are met during run time. Challenging is the autonomy of participants in a cross-organizational setting und therefore that only non-intrusive monitoring techniques can be considered where no new software can be installed or changed at the participants side.

This work therefore provides a new approach in order to monitor the observable behavior of choreography participants and to verify choreography integrity constraints and their violation during run time. At first, this work will therefore introduce the required sensors and their corresponding infrastructure and will also introduce a basic mechanism for detecting integrity constraint violations during run time. This will be underpinned by a formal model based on the event calculus. The calculus allows for detection deviations from normal choreography conditions by temporal reasoning and therefore allows for detection of process integrity constraints violations at run time. The introduced approach is not only able to handle message based integrity constraints, but also able to handle interaction requirements like time constraints, requirements for single choreography instances like the control flow constraints and also cross choreography instance based constraints, which the participants have to abide in order to successfully run the choreography.

Are process integrity constraint violations detected by the approach developed within the first part of this work, the question how to solve integrity violations remains. Within this work, a recovery approach based on the transaction concept is introduced which is able to include the verification results directly into transactional activity control. The concept presented extends traditionally known protocols by a coordinator which is able to decide autonomously if a transaction is able to finish successfully by using rules. Traditionally, a transaction initiator is the participant deciding about the outcome of a transaction. In practical choreography situations there is not always an initiator who is able to decide about all participants. In addition, the presented approach is also able to use the verification results for transactional activity control and is therefore able to support choreography recovery in the case of process integrity constraint violations.

Danksagung

Diese Arbeit ist während meiner Tätigkeit am Arbeitsbereich Verteilte Systeme und Informationssysteme (VSIS) des Fachbereichs Informatik der Universität Hamburg und am Hamburger Informatik Technologie Center e.V. (HITeC e.V.) im Rahmen des EU-Forschungsprojektes R4eGov entstanden. Beim Verfassen der Arbeit haben mich in dieser Zeit viele Menschen unterstützt, denen ich hiermit noch einmal danken möchte.

Mein besonderer Dank geht an meinen Betreuer Prof. Norbert Ritter, der mich seit dem Studium auf meinem akademischen Weg begleitet hat. Du hast mir im Rahmen des EU-Projektes und bei der Dissertation die notwendigen Freiräume gelassen und durch zahlreiche Diskussionen und Gespräche die richtigen Denkanstöße gegeben. Es hat einfach Spaß gemacht, mit dir zusammenzuarbeiten. Mein Dank geht außerdem an Herrn Prof. Stefan Deßloch für seine Tätigkeit als Zweitgutachter.

Meinen Kollegen vom VSIS-Flur danke ich für die immer sehr entspannte Arbeitsatmosphäre und die zahllosen »Teechen«-Pausen. Besonders danke ich meinem Kollegen Marc Holze für die großartige Zeit in F522. Du hast mir immer die notwendige Ablenkung, Unterstützung sowie sachdienliche Hinweise (»Jetzt schreib das doch endlich mal auf«) gegeben. Weiterhin danke ich Anne Hansen-Awizen für die gute Laune und stetige Unterstützung sowie Martin Husemann und Sonja Zaplata für zahlreiche Gespräche über die Inhalte meiner Arbeit. Den (ehemaligen) Studenten vom VSIS-Flur Stefan Fink, Daniel Voigt, Alexander Fey, Matthias Schulz und Nils Meder danke ich für ihre engagierte Mitarbeit. Sie haben bei der Konzeptionierung und Realisierung der verschiedenen Prototypen mitgewirkt.

Zu guter Letzt danke ich meiner Frau Stephanie einfach für alles und vor allem für ihre Geduld (»Nein, ich kann jetzt nicht in den Urlaub, ich muss noch...«).

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Im Bereich des eGovernment lässt sich ein steigender Trend verzeichnen, organisationsübergreifende Prozesse durch IT-Infrastrukturen zu unterstützen. Ein Beispiel dafür sind Prozesse zwischen Europol, Eurojust und den Behörden der EU-Mitgliedsstaaten, die gemeinsam zur Bekämpfung grenzübergreifender Kriminalität zusammenarbeiten [Bou07]. Wie auch im EU-finanzierten Forschungsprojekt R4eGov festgestellt worden ist, sind für die IT-seitige Unterstützung solcher Prozesse die bekannten Modelle mit zentraler Steuerung aus dem Bereich des Workflow-Managements nicht direkt nutzbar, da alle Organisationen autonom bleiben und keine zentrale Steuerung akzeptieren [Bou07]. Um dieses Problem zu umgehen, kann für bestimmte Anwendungsfälle ein Nachrichtenprotokoll definiert werden, mit dessen Hilfe ein organisationsübergreifendes Ziel durch Nachrichtenaustausch erreicht werden kann. Wie ein Prozess innerhalb der Organisation zur Umsetzung des eigenen Verhaltens am Protokoll realisiert wird, soll dabei Außenstehenden verborgen bleiben.

Um solche organisationsübergreifenden Prozesse mit heterogenen Teilnehmern durchzuführen, bieten sich Service-orientierte Architekturen (SOA) an, welche ein bewährtes Architekturkonzept zur Realisierung von lose gekoppelten und verteilten Software-Systemen darstellen. Innerhalb einer SOA wird das Problem der Heterogenität von elektronischen *Diensten* durch Kommunikation über standardisierte und implementierungsunabhängige Schnittstellen überwunden. Die Anwendungslogik ist dabei charakteristischerweise nicht einem einzelnen zentralen System zugeordnet, sondern wird über Dienste innerhalb einer SOA verteilt. Web-Services dienen dabei typischerweise zur Implementierung des SOA-Paradigmas und werden mehr und mehr genutzt, um auch *Geschäftsprozesse* zu realisieren. Dabei werden im Bereich der SOA zunehmend Dienste betrachtet, welche auch lange anhaltende Konversationen mit anderen Diensten durchführen und weniger einfache und kurze Anfrage/Antwort-Interaktionen abhandeln.

Damit langlaufende Konversationen zwischen Diensten besser beschrieben werden können, wurde im Bereich der SOA eine neue Sichtweise auf Dienstinteraktionen eingeführt. Sie beschreibt die Interaktionen zwischen Diensten aus der Sicht eines idealen Beobachters, welcher alle Interaktionen zwischen Diensten sowie deren Daten- und Kontrollflussabhängigkeiten sehen kann [ZDH+06]. Das resultierende Interaktionsmodell wird auch *Choreographie* genannt [Pel03] und kann für Web-Services beispielsweise durch die *Web Service Choreography Description Language* (WS-CDL [BLF+05]) beschrieben werden. Das Konzept der Choreographie kann für die organisationsübergreifenden Prozesse beispielsweise im Fall von Europol und Eurojust direkt eingesetzt werden, da eine Choreographie das beobachtbare

Verhalten von Diensten sowie das Nachrichtenprotokoll beschreibt, ohne dabei das interne Verhalten der Dienste näher zu modellieren. Dabei wird zur Koordination der Prozesse zwischen den Organisationen keine zentrale Stelle benötigt, wenn sich alle Dienste an das spezifizierte Interaktionsmodell halten.

Choreographien sind natürlich nicht nur auf den Bereich des eGovernments beschränkt, denn es gibt Bemühungen, um Standards für verschiedene Bereiche von organisationsübergreifenden Choreographien zu etablieren. Weske beschreibt in [Wes07] als Beispiele für organisationsübergreifende Choreographien Projekte wie RosettaNet aus dem Bereich der Warenwirtschaft, SWIFTNet aus dem Bereich der Finanzen oder auch Health Level 7 (HL7) aus dem Bereich des Gesundheitswesens. Das Konzept der Choreographie haben sich eine Reihe von Ansätzen zunutze gemacht, um Protokolle zwischen Diensten von verschiedenen Organisationen zu definieren [ADO+06, BBMP06, RWR06] oder aber Dienstskelette für eine Implementierung zu generieren [DD04, ZDH+06, SVTD09], welche dem globalen Interaktionsmodell genügen. Aber auch bei der Modellierung von Choreographien gibt es eine Reihe von Ansätzen, welche die Korrektheit von Modellen in Bezug auf Kriterien wie beispielsweise Erfüllbarkeit, Verklemmungsfreiheit oder auch Ausführbarkeit sicherstellen sollen [FUMK03, FUMK04, BBM+05, ADO+06, VSC06, ADO+08]. Diese Ansätze sind geeignet, um Fehler in Choreographie-Modellen schon bei der Modellierung und damit vor der Laufzeit zu entdecken und auszuschließen.

Zusätzlich muss sichergestellt werden, dass sich Dienste auch zur Laufzeit so verhalten wie es das globale Interaktionsmodell vorschreibt. In organisationsübergreifenden Prozessen kann davon ausgegangen werden, dass Organisationen ihre Dienste selbstständig implementieren, um einen Dienst passend zum Interaktionsmodell zu realisieren. Da jede Organisation selbst für die Implementierung der Schnittstellen verantwortlich ist, können sich zum einen bei der Implementierung sowie zum anderen durch Evolution der Schnittstellen Fehler einschleichen, so dass sich Teilnehmer letztlich doch nicht mehr an das spezifizierte Choreographie-Modell halten. Genauso gibt es Integritätsbedingungen von Choreographien, die nur zur Laufzeit überprüfbar sind. Als Anwendungsbeispiel für Integritätsbedingungen kann eine Autovermietung betrachtet werden, die verschiedene Sensoren auf ihren Parkplätzen montiert hat [MS04]. Fährt ein Wagen vom Hof, registriert dies ein Sensor in der Ausfahrt und die Inventarliste wird entsprechend aktualisiert. Als Integritätsbedingung kann hier für den Prozess beispielsweise definiert werden, dass ein Wagen in einer weiteren Instanz des Vermietungsprozesses nicht noch einmal vom Hof fahren kann, wenn er nicht vorher zurückgekommen ist. Daher sind bisherige Verfahren, welche nur das Modell verifizieren aber nicht das Laufzeitverhalten betrachten, nicht ausreichend. Sollen Anforderungen wie Nachvollziehbarkeit und Integrität in Choreographien gewährleistet werden, ist eine Überwachung und Verifikation der Choreographie zur Laufzeit sowie eine geeignete Reaktion im Fehlerfall notwendig.

In dieser Arbeit soll untersucht werden, wie organisationsübergreifende Choreographien auf Basis von elektronischen Diensten angemessen und effizient zur Laufzeit überwacht,

überprüft und im Fehlerfall bei der Behebung der erkannten Abweichungen vom modellierten Verhalten unterstützt werden können. Solche organisationsübergreifenden Choreographien können auch als *Prozess-Choreographien* bezeichnet werden [Wes07] und stellen verteilte Geschäftsprozesse dar. Eine wichtige zu lösende Frage ist das Erkennen von Abweichungen einzelner Teilnehmer im Bezug auf das globale Interaktionsprotokoll und der spezifizierten Integritätsbedingungen zur Laufzeit. Herausfordernd ist, dass die Teilnehmer an einer Choreographie autonome Organisationen sein können und damit in den meisten Fällen nur ein Überwachungsmechanismus infrage kommt, bei dem keine Software innerhalb der Organisationen verändert oder neu installiert werden muss. Weiterhin ergibt sich aus der Autonomie der Organisationen, dass ein zentrales System zur Steuerung der Prozesse nicht infrage kommt. Die Verifikation einer Prozess-Choreographie kann daher nur auf das beobachtbare Verhalten der Teilnehmer zurückgreifen, weshalb die Entwicklung eines nicht-intrusiven Überwachungsmechanismus eine wesentliche Anforderung dieser Arbeit darstellt.

Wichtig zur Erkennung von Abweichungen zum modellierten Verhalten in Prozess-Choreographien ist ein Modell, anhand dessen ein Soll-Ist-Vergleich effizient zu Laufzeit durchgeführt werden kann. Beispielsweise basiert WS-CDL als Sprache zur Modellierung von Choreographien auf dem bekannten π-Kalkül [CYM+06]. Allerdings lässt sich mit dem Kalkül hauptsächlich sehr gut feststellen, was als Nächstes oder zu einem späteren Zeitpunkt passieren soll, weniger aber, was in der Vergangenheit hätte passieren sollen [MMCT08]. Diese Anforderung ist jedoch wesentlich zur Überwachung von Prozessen. Der π-Kalkül eignet sich also eher für die Spezifikation und Verifikation von Modellen, aber weniger für die Laufzeitüberwachung. Eine wichtige Anforderung für die Arbeit ist daher die Wahl eines geeigneten formalen Modells zur effizienten Überprüfung, ob das Laufzeitverhalten der Choreographie-Teilnehmer dem spezifizierten Verhalten im Modell entspricht.

Das Erkennen von Fehlern zur Laufzeit eröffnet für Organisationen die Möglichkeit einer direkten Reaktion und damit das sofortige Einleiten von Maßnahmen zur Wahrung der Konsistenz des organisationsübergreifenden Prozesses. Allerdings stellt sich hier eine weitere wichtige Frage, welche Strategien zur Behandlung von Abweichungen spezifizierter Integritätsbedingungen angewendet werden können. Zur Lösung dieses Problems bietet sich beispielsweise das aus den Datenbanken bekannte Paradigma der Transaktion an, welches sicherstellen kann, dass im Falle von Integritätsverletzungen Aktivitäten im Prozess zurückgesetzt werden.

1.1. Kurzüberblick über den Stand der Forschung

Zur Erkennung von Integritätsverletzungen und damit der Erkennung von Abweichungen des Laufzeitverhaltens zum vorher definierten Modell gibt es bereits eine Reihe von Ansätzen. Ein sehr allgemeiner Ansatz aus dem Bereich des Requirements-Monitoring wird

von Robinson in [Rob06] vorgeschlagen. Robinson definiert dabei ein Rahmenwerk namens *ReqMon*, welches eine Methode zur Definition von Anforderungen, dem Erkennen von Konflikten in Anforderungen und der Analyse von Überwachungsdaten definiert. Innerhalb von ReqMon werden Anforderungen mithilfe der zielorientierten Sprache KAOS (siehe [Lam01]) in Form von temporal-logischen Ausdrücken definiert. Wie Sensoren allerdings in verschiedenen Domänen instrumentiert werden, wird in diesem Ansatz nicht weiter ausgeführt.

Weiterhin gibt es Ansätze für zentral koordinierte Orchestrierungen, die durch zentrale Workflow-Engines gesteuert werden. Im Bereich der zentral koordinierten SOA-Ansätze ist beispielsweise der Ansatz von Mahbub und Spanoudakis zu nennen, welcher mithilfe des Ereigniskalküls zur Laufzeit entsprechende Abweichungen zum Geschäftsprozess-Modell detektiert [MS05]. Genauso haben Guinea und Baresi einen Ansatz entwickelt, um Geschäftsprozesse zur Laufzeit mit Hilfe von Integritätsbedingungen zu kontrollieren [BG05]. Weitere Ansätze, wie beispielsweise [LAP06], [PBB+04], [LJH06] oder auch [GCN+07] nutzen ebenso formale Methoden für einen Soll-Ist-Vergleich und können Sensoren zur Überwachung des Laufzeitverhaltens sehr gut in eine zentrale Workflow-Engine einbringen, was bei organisationsübergreifenden Prozessen aber nicht mehr der Fall ist.

Im Bereich der Choreographien gibt es bisher weit weniger Ansätze. Montali, Mello, Chesani und Torroni [MMCT08] etwa nutzen wie Mahbub und Spanoudakis das Ereigniskalkül zur Laufzeitverifikation von Choreographien. Allerdings beschränkt sich dieser Ansatz eher auf die Definition und das Erkennen von Fehlern, stellt aber keinen vollständigen Überwachungsansatz dar. In ihrem Ansatz fehlt beispielsweise das Vorgehen, wie die Abbildung von einer Menge von Nachrichten auf die Regeln des Ereignis-Kalküls erfolgen soll, oder auch, wie die Nachrichten in organisationsübergreifenden Choreographien überhaupt überwacht werden können. Ein weiteres Manko des Ansatzes ist die Grundlage der Choreographie-Spezifikation. Diese basiert nicht auf Standards, wie es bei organisationsübergreifenden Prozessen wünschenswert ist, sondern auf der grafischen Modellierungssprache DecSerFlow [AP06].

1.2. Problembeschreibung, Lösungsansatz und Gliederung dieser Arbeit

Wie in den vorhergehenden Abschnitten kurz vorgestellt, gibt es für Prozess-Choreographien bisher keine ausreichende Unterstützung bei der Laufzeitüberwachung und damit der Feststellung, ob sich autonome Teilnehmer an Choreographien auch zur Laufzeit so wie spezifiziert verhalten. Aus diesem Grunde wird in dieser Arbeit eine Weiterentwicklung hin zur Laufzeitüberwachung von Prozess-Choreographien angestrebt, welche bei organisationsübergreifenden Prozessen besser für eine Überwachung von Integritätsbedingungen der Prozesse genutzt werden kann. Wird eine Abweichung zu den formulierten Integritäts-

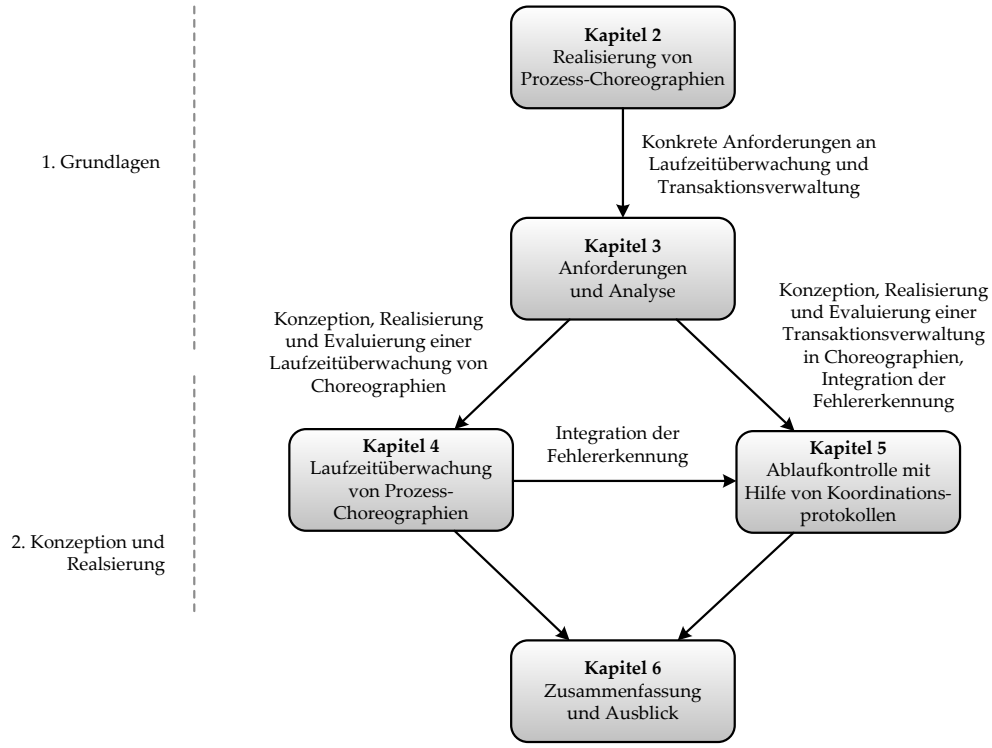


Abbildung 1.1.: Struktureller Aufbau der Arbeit

bedingungen erkannt, müssen Choreographien bei der Behebung der Integritätsprobleme nach Möglichkeit durch Software unterstützt werden.

Damit gliedert sich die vorliegende Arbeit in zwei Hauptbereiche, und zwar (1) den technischen Grundlagenabschnitten zur Analyse von bestehenden Techniken zur Kontrolle von Prozess-Choreographien sowie der Ermittlung der Anforderungen an die Laufzeitüberwachung und Behebung von Integritätsverletzungen, welche in (2) durch Konzepte und dazugehörige Realisierungen entsprechend umgesetzt werden (siehe auch Abbildung 1.1).

Für den ersten Bereich der technischen Grundladen wird in Abschnitt 2 der aktuelle Stand der Technik zur Realisierung von organisationsübergreifenden Prozessen auf Basis von dienstbasierten Architekturen vorgestellt. Eine Analyse der vorhandenen Ansätze in Bezug auf Überwachung und Behebung von Integritätsverletzungen schließt dabei mit der Entwicklung von generellen Anforderungen an die Arbeit ab, die in Abschnitt 3 noch weiter in Bezug auf konkrete zu verwendende Mechanismen verfeinert werden. Im Detail werden dafür in Abschnitt 2 serviceorientierte Architekturen eingeführt und zusätzlich die Grundlagen zur Prozessverarbeitung auf Basis von Web-Services erläutert. Eine wichtige Rolle spielen dabei die verschiedenen Sichten der Orchestrierung und der Choreographie sowie deren Modellierungsmöglichkeiten. Dafür werden verschiedene Ansätze zur Modellierung von Choreographien vorgestellt und verglichen, da diese Modellierungsansätze eine Möglichkeit zur Spezifikation von überwachbaren Anforderungen zur Laufzeit darstellen und damit als Grundlage eines späteren Monitoring-Ansatzes benutzt werden. Zusätzlich wer-

den in Abschnitt 2 automatisierte Fehlerbehebungsmechanismen auf Basis des Transaktionskonzeptes vorgestellt und in Bezug auf Choreographien weiter untersucht. Die Analyse der Grundlagen und der verwandten Arbeiten stellt die Grundlage für Abschnitt 3 dar, welches formale Beschreibungsmodelle zur Spezifikation von überwachbaren Anforderungen genauso wie technische Anforderungen an die Kontrollinfrastruktur weiter konkretisiert. Dies wird im zweiten Teil des dritten Abschnitts dieser Arbeit auch für den Bereich der Behebung von Integritätsverletzungen durch Transaktionen durchgeführt.

Der zweite Hauptbereich, welcher die Konzeption, Realisierung und Evaluierung der Überwachung von Choreographien und der späteren Fehlerbehebung von Integritätsverletzungen durch Transaktionen umfasst, wird in zwei umfangreicheren Abschnitten behandelt. Für die Überwachung von Choreograhie-Anforderungen werden dafür in Abschnitt 4 zum einen die Infrastruktur zur Überwachung und deren Realisierung sowie verschiedene Möglichkeiten für den Soll-Ist-Vergleich auf Basis von Regeln und auf Basis des Ereigniskalküls vorgestellt. Der Abschnitt schließt mit einer Evaluation der Implementierungen und einem Vergleich der Ansätze ab. Zur Fehlerbehebung auf Basis von Transaktionen werden entsprechend in Abschnitt 5.1 die Infrastruktur sowie die Implementierung der Transaktionskontrolle vorgestellt. Eine Evaluation der Implementierung schließt den Teilabschnitt ab. Abschließend wird in Abschnitt 5.2 eine Möglichkeit der Integration der Fehlererkennung in die Ablaufkontrolle von Transaktionen vorgestellt, damit für einen erfolgreichen Ausgang einer Transaktion nicht nur die Teilnehmer selbst definieren, dass sie ihre Arbeit erfolgreich abgeschlossen haben, sondern dies zusätzlich auch durch die Laufzeitverifikation belegt werden kann.

Im letzten Abschnitt werden die Ergebnisse der Arbeit zusammengefasst sowie die Zielerreichung der Arbeit diskutiert. Den Abschluss bildet eine Diskussion über weiterführende Arbeiten und Themen.

1.3. Ergebnisse und Beiträge

Diese Arbeit beschreibt auf Basis der geführten Untersuchungen und der Implementierung der Konzepte einen neuen Ansatz zur Laufzeitüberwachung und zur Kontrolle von Prozess-Choreographien. Im Einzelnen sind dabei für den Bereich der Laufzeitüberwachung folgende für Forschung und Praxis relevante Teilergebnisse erreicht worden, die hier vorweg schon genannt werden sollen:

- **Nicht-Intrusive Sensoren** (Abschnitt 4.1): Auf der Basis eines Enterprise-Service-Busses (ESB), welcher in der Lage ist, Nachrichten und Ereignisse an eine Monitoring-Komponente weiterzuleiten, kann das beobachtbare Verhalten von Choreographie-Teilnehmern über Stellvertreterdienste sehr pragmatisch ermittelt werden. Das Konzept ist in [RR09] veröffentlicht und führt bei der Überwachung nur einen geringen Mehraufwand ein, um das beobachtbare Verhalten der Teilnehmer zu ermitteln.

- **Ein formales Interaktionsmodell** (Abschnitt 4.2 sowie 4.3): Auf Basis von WS-CDL sind in dieser Arbeit zwei unterschiedliche Ansätze zur formalen Spezifikation von Interaktionen mit dem Ziel der Laufzeitüberwachung entwickelt worden. Zum einen ist dies ein regelbasiertes System, welches direkt zur einfachen Überwachung von Interaktionen eingesetzt werden kann. Dieses Modell wurde im Rahmen des EU-geförderten Projekts R4eGov[1] entwickelt [SIR+09] und im Rahmen von Prototypen auch bei Use-Case-Partnern vorgestellt [RLS+09]. Es besitzt allerdings einige Einschränkungen, die mithilfe des zweiten Ansatzes auf Basis des Ereigniskalküls behoben werden können. Mit Unterstützung des Models auf Basis des Ereigniskalküls lassen sich nicht nur funktionale Bedingungen einer Prozess-Choreographie überwachen, sondern auch non-funktionale Bedingungen wie beispielsweise Zeitschranken. Dabei ist nicht nur eine instanzbasierte Überwachung, sondern auch eine instanzübergreifende Überwachung von Prozess-Choreographien möglich. Beide formalen Beschreibungen auf Basis von Regelsystemen können direkt aus einer WS-CDL-Beschreibung abgeleitet werden und durch eigene Bedingungen erweitert werden.

- **Effizienter Soll/Ist-Vergleich** (Abschnitt 4.2.3 sowie 4.3.7): Im Rahmen der Evaluation der entwickelten Interaktionsmodelle zur Überwachung lässt sich zusammenfassend sagen, dass sich einfache Modelle auf Basis von einfachen Regeln sehr gut zur Interaktionskontrolle während der Laufzeit einsetzen lassen. Die Zeit, die ein heute üblicher Rechner zur Verifikation einer einzigen Nachricht benötigt, beläuft sich dabei auf wenige Millisekunden. Sollen aber auch weitere Integritätsbedingungen wie non-funktionale Bedingungen nicht nur instanzbasiert für einen Prozess, sondern auch instanzübergreifend für mehrere Prozesse geprüft werden, wird mit dem Ereigniskalkül zwar ein Mehraufwand eingeführt, der sich aber auf heute üblichen Rechnern im Bereich von 30-50ms pro Nachricht bewegt und damit gut zur Laufzeitüberwachung geeignet ist.

Für den Bereich der Fehlerbehebung im Falle von Integritätsverletzungen sind im Rahmen dieser Arbeit folgende Teilergebnisse erreicht worden:

- **Transaktionale Kontrolle für Choreographien** (Abschnitt 5): Im Rahmen von verschiedenen Veröffentlichungen ([RHR08, RHFR10]) sind zum einen die Herausforderungen von transaktionaler Kontrolle in Choreographien definiert sowie zum anderen ein neues Konzept entwickelt worden, um Transaktionen in Prozess-Choreographien zu steuern. Das Konzept erweitert die bisherigen Ansätze der transaktionalen Kontrolle von Prozessen um einen Koordinator, der aufgrund von Regeln und eines Aufrufbaumes autonom entscheiden kann, ob eine Transaktion erfolgreich beendet werden kann oder nicht. In bisherigen Ansätzen ist der Initiator einer Transaktion immer derjenige Teilnehmer, welcher über den Transaktionsabschluss entscheidet (ent-

[1] `http://www.r4egov.eu`

weder Abbruch oder Festschreiben der Ergebnisse). In vielen praktischen Szenarien kann der Initiator dieses aber nicht entscheiden, weshalb der Koordinator in diesem Ansatz diese Rolle übernimmt. Das Rahmenwerk zur Koordination ist auch als Erweiterung der Standards WS-Coordination sowie WS-AtomicTransaction und WS-BusinessActivities implementiert und als Open-Source-Projekt unter `http://tracg.sourceforge.net` veröffentlicht.

- **Anbindung an die Fehlererkennung** (Abschnitt 5.2): Die Anbindung der Fehlererkennung an die transaktionale Kontrolle ist eine Optimierung des Transaktionsabschlusses im Abbruchfall. Wird bereits während der Interaktionen ein Fehler erkannt, muss eine transaktionale Kontrolle nicht mehr auf den Abschluss der Aktivitäten warten, sondern kann direkt korrigierend eingreifen. Dies kann allerdings nur für jene Fehler geschehen, die keine manuelle Fehlerkorrektur benötigen.

Zusammenfassend tragen die Ergebnisse dieser Arbeit dazu bei, dass Choreographien robuster ausgeführt werden können und dass das jeweilige Verhalten von Teilnehmern in Bezug auf Integritätsbedingungen für alle Teilnehmer transparent ist.

2. Realisierung von Prozess-Choreographien

Dieses Kapitel beinhaltet den ersten Teil der technischen Grundlagen der vorliegenden Arbeit. Dabei sollen in diesem Kapitel die heutzutage zur Realisierung von organisationsübergreifenden Prozessen üblichen Techniken kurz eingeführt und in Bezug auf die Aufgabenstellung analysiert werden. Im ersten Teil des Kapitels werden dafür die Grundlagen der Dienstkomposition zur Realisierung von organisationsübergreifenden Prozessen vorgestellt. Im Detail sind dies serviceorientierte Architekturen, Web-Services und Dienstkompositionstechniken auf Basis von Web-Services. Im zweiten Teil des Kapitels werden aktuelle Ansätze zur Überwachung von Prozessanforderungen für Dienstkompositionen eingehender betrachtet und generelle Anforderungen für die Überwachung von organisationsübergreifenden Prozessen formuliert, welche im Rahmen der Arbeit in den späteren Kapiteln eingehender betrachtet werden. Im letzten Teil des Kapitels werden zudem Grundlagen der Transaktionskontrolle für Dienstkompositionen betrachtet und die Frage erörtert, welche Probleme bei der Fehlerbehebung in organisationsübergreifenden Prozessen auftreten und ob aktuelle Ansätze diese lösen können.

2.1. Dienstkomposition zur Realisierung organisationsübergreifender Prozesse

Zur Realisierung von organisationsübergreifenden Prozessen gibt es verschiedene Technologien, welche zum Einsatz kommen können. Dabei muss allerdings immer die *Autonomie* der am Prozess teilnehmenden Organisationen berücksichtigt werden. Die beteiligten Organisationen haben in den meisten Fällen eigene IT-Infrastrukturen, welche bei einer Zusammenarbeit entsprechend interoperabel gehalten werden müssen, ohne die internen Infrastrukturen maßgeblich zu verändern. Die *Überwindung von Heterogenität* ist daher eine der wesentlichen Anforderungen an organisationsübergreifende Prozesse. In den letzten zehn Jahren haben sich zur Realisierung von organisationsübergreifenden Prozessen serviceorientierte Architekturen herausgebildet, die zum einen die Interoperabilität zwischen Organisationen herstellen sollen und zum anderen den jeweiligen Dienstanbietern die Möglichkeit lassen, ihre eigenen Technologien zur Realisierung von Diensten auszuwählen.

2.1.1. Serviceorientierte Architekturen

Eine serviceorientierte Architektur (SOA) ist eine abstrakte Softwarearchitektur zur Realisierung von größeren verteilten Systemen [MET+10]. Die wesentlichen Kennzeichen großer verteilter Systeme innerhalb von Organisationen fasst Josuttis wie folgt zusammen [Jos08]:

- Große verteilte Systeme bestehen zu einem großen Teil aus *Altlasten* (legacy systems), die zu verschiedenen Zeiten von verschiedenen Teams mit unterschiedlichen Techniken implementiert und eingesetzt wurden.
- Große verteilte Systeme sind aufgrund von unterschiedlicher verwendeter Technologie sehr *heterogen*. Ein weiterer Grund für Heterogenität ist meistens die lange Lebensdauer von verschiedenen internen Systemen. Wenn diverse Teilsysteme miteinander interagieren, werden die entstehenden Systeme zudem sehr *komplex*.
- Einzelne Komponenten großer Systeme haben *verschiedene Eigentümer*, was das Maß an Heterogenität innerhalb einer Systemlandschaft noch erhöht.
- Große Systeme besitzen ein gewisses Maß an *Redundanz*, was für die Konsistenz von Daten problematisch ist. Genauso gibt es in den Systemen viele *Flaschenhälse*, welche die Skalierbarkeit der Systeme stark beeinflussen.

Diese Kennzeichen treffen natürlich auch auf die Realisierung von organisationsübergreifenden Prozessen zu, welche auf Basis der schon bestehenden Systeme von verschiedenen Organisationen arbeiten müssen. Eine SOA kann die Realisierung von organisationsübergreifenden Prozessen unterstützen, da sie für diese verschiedenen Kennzeichen Realisierungsmöglichkeiten anbietet. Ein wesentliches Merkmal einer SOA ist dabei der modulare Aufbau aus voneinander unabhängigen und wiederverwendbaren Komponenten, welche fachliche Funktionalität in Form von *Diensten* bereitstellt [RHS05]. Diese Dienste werden bei Verwendung durch standardisierte und implementierungsunabhängige Schnittstellen angesprochen. Richter, Haller und Schrey argumentieren weiterhin, dass durch das Verbergen von Implementierungsdetails hinter einheitlichen Schnittstellen das Geheimnisprinzip umgesetzt und damit die Implementierung von der Nutzung eines Dienstes getrennt wird [RHS05]. In Verbindung mit gängigen Transportprotokollen wie beispielsweise HTTP[1] führt dies zu einer hohen Plattform- und Implementierungsunabhängigkeit eines Dienstes, da jeder Dienstnutzer seine eigenen Technologien zum Ansprechen eines Dienstes nutzen kann. Die Kapselung von fachlicher Funktionalität ermöglicht zudem die Bereitstellung von Funktionalitäten der oben angesprochenen Altlasten und soll auch zur Reduktion der Komplexität der Systeme beitragen [RHS05].

Weiterhin gehört die *lose Kopplung* von Diensten zu den wichtigsten Eigenschaften des SOA-Konzepts. Sie wird durch dynamisches Suchen, Finden und Einbinden von Diensten

[1] Hypertext transfer protocol, siehe [FGM+99]

zur Laufzeit realisiert. Die Voraussetzungen für die lose Kopplung sind dabei maschinenlesbare und einheitlich beschriebene Schnittstellen sowie Verzeichnisdienste, welche das Auffinden von Diensten zur Laufzeit erst ermöglichen [MET+10]. Innerhalb einer SOA lassen sich somit drei unterschiedliche Rollen identifizieren [GP09], welche in Abbildung 2.1 dargestellt sind:

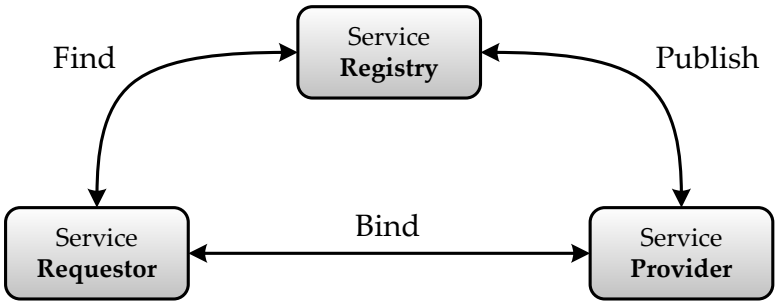


Abbildung 2.1.: Schema einer serviceorientierten Architektur (nach [GP09]).

- Der **Service-Provider:** Ein Dienst veröffentlicht seine Dienstbeschreibung in einem Verzeichnis (*Publish*). Die veröffentlichte Dienstbeschreibung dokumentiert die öffentliche Schnittstelle eines Service-Providers in maschinenlesbarer Form. Über diese öffentliche Schnittstelle können Dienstnutzer mit dem Dienst interagieren (*Bind*) und die angebotenen Dienstleistungen in Anspruch nehmen.
- Die **Service-Registry:** Ein Dienstverzeichnis soll das Veröffentlichen (*Publish*) und die gezielte Suche nach Diensten ermöglichen (*Find*).
- Der **Service-Requestor:** Ein Dienstnutzer, der über das Dienstverzeichnis einen passenden Dienst gefunden hat (*Find*), kann jetzt mit dem gefundenen Dienst aufgrund der publizierten Schnittstellenbeschreibung interagieren (*Bind*).

Zur Realisierung einer SOA können verschiedene dienstbasierte Techniken wie CORBA [Obj04] oder RPC [Thu09] herangezogen werden. Verschiedene Autoren argumentieren allerdings, dass Web-Services aufgrund von XML-basierten Spezifikationen besser zur Realisierung von SOA eingesetzt werden können, da XML schon von Haus aus eine plattform- und implementierungsunabhängige Spezifikation erlaubt [KL04, ACKM04, Pap08, GP09]. Damit leisten Web-Services einen wichtigen Beitrag zur Überwindung von Heterogenität zwischen einzelnen Diensten, da ausgetauschte Nachrichten und Schnittstellenbeschreibungen in XML codiert sind. Zusätzlich lassen sich Web-Services durch einfache Web-Adressen (URIs) identifizieren, was in Verbindung mit dem Transportprotokoll HTTP eine einfache Verwendung von unterschiedlichen Diensten auch über Firewall-Grenzen hinweg ermöglicht [ACKM04]. Die Dienste sind damit gegenüber ihren Dienstnutzern autonom, da jede Technologie zur Realisierung eines Web-Services genutzt werden kann, solange die spezifizierte Schnittstelle mithilfe von XML-Nachrichten angesprochen werden kann. Genauso

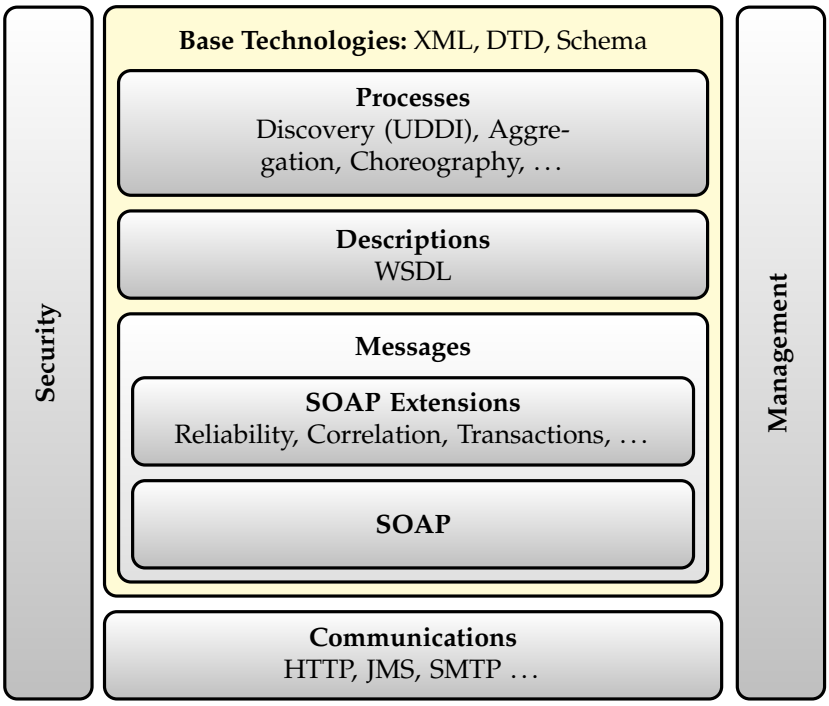


Abbildung 2.2.: Das WS-Architecture-Schichtenmodell [MBF+04].

bleibt der Dienstnutzer in der Wahl der Technologien zum Ansprechen der Dienste weitestgehend frei, solange die Technologien die Web-Services-Standards beherrschen.

Das *World-Wide-Web-Konsortium* (W3C) beschreibt in der *Web-Service-Architecture*-Spezifikation [MBF+04] eine Architektur, in der für die technische Umsetzung von Web-Services die W3C-Standards UDDI, WSDL und SOAP verwendet werden. Diese Standards beschreiben das Auffinden von Web-Services über Verzeichnisdienste (UDDI, [Bel02]), ein XML-basiertes Nachrichtenaustauschformat (SOAP, [ML07]) sowie eine XML-basierte Sprache zur Schnittstellenbeschreibung von Web-Services (WSDL, [CWMR07]). Die Abbildung 2.2 zeigt das Schichten-Modell der WS-Architecture in Kurzform, wobei aufgrund der Erweiterbarkeit von SOAP noch eine Reihe von Zusatzspezifikationen (*SOAP Extensions*) existieren, welche SOAP um Funktionalitäten wie beispielsweise Sicherheit, Zuverlässigkeit oder Transaktionssteuerung ergänzen. In der obersten Schicht ist die Prozessebene der WS-Architecture dargestellt, wo mithilfe von Dienstkomposition komplexe Abläufe realisiert werden können. Die Komposition von Diensten auf der Prozessebene des Web-Services-Stacks ist (etwas vereinfacht) auch der wesentliche Grund für den Einsatz von SOA in Organisationen. Wenn Standardsoftware oder Legacy-Systeme ihre Dienste in Form von Web-Services bereitstellen, können sie auf der einen Seite zu einem Geschäftsprozess komponiert werden und auf der anderen Seite lässt sich so über die Dienste die IT-Infrastruktur leicht an die Geschäftsprozesse einer Organisation anpassen. Zusätzlich gibt es vertikale Aspekte, die sich durch alle Schichten der Architektur ziehen. Zum einen sind dies Sicherheits- und zum anderen Management-Aspekte, die beide auf jeder Ebene eine Rolle spielen.

Im Folgenden wird kurz das Nachrichtenformat SOAP für Web-Services vorgestellt, welches im späteren Verlauf der Arbeit bei der Überwachung von Nachrichten eines Prozesses im Detail benötigt wird.

2.1.1.1. SOAP

Wie oben beschrieben, definiert SOAP [ML07] als W3C-Recommendation ein Nachrichtenaustauschformat in XML. Die Spezifikation legt dafür die Darstellung und Modelle für die Verarbeitung von Informationen, die Erweiterungsmöglichkeiten von SOAP und Regeln zur Bindung an verschiedene Transportprotokolle fest[2].

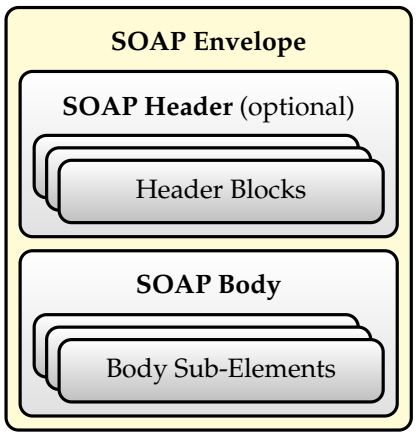


Abbildung 2.3.: Schema einer SOAP-Nachricht (nach [Cer02, GHM+07]).

Eine SOAP-Nachricht besteht dabei aus einem XML-Dokument, welches aus dem Wurzelelement `Envelope` sowie den darin enthaltenen Elementen `Header` und `Body` (vgl. Abb. 2.3) besteht. Der `Body` einer Nachricht ist dabei verpflichtend anzugeben, die Angabe eines Nachrichtenkopfes ist aber optional. Der `Header` enthält dabei einzelne Informationen (*Header-Blocks*) zur Verarbeitung der Nachricht, welches beispielsweise Sicherheitsinformationen, Nachrichten-Identifikatoren oder auch andere anwendungsspezifische Informationen sein können. Daher ist die konkrete Definition von `Header` auch nicht Teil der SOAP-Spezifikation, sondern wird für spezielle Anwendungsfälle mithilfe von SOAP-Erweiterungen in anderen Spezifikationen beschrieben. Allerdings definiert die SOAP-Spezifikation, dass ein Header-Block das `mustUnderstand`-Attribut beinhalten kann. Mit diesem Attribut wird definiert, dass ein Empfänger den spezifischen Header-Block verarbeiten können muss. Kann er dies nicht, wird eine Fehlermeldung zurückgeliefert. Genauso kann für jeden Header-Block ein `role`-Attribut definiert werden, was beschreibt, welche Empfängerrolle die jeweilige Information verarbeiten soll. Dies ist insbesondere wichtig, falls eine Nachricht über mehrere Zwischenstationen an einen Empfänger gesendet wird.

[2] Diese werden in den entsprechenden weiteren Teilen der SOAP-Spezifikation in [GHM+07] und [MGH+07] definiert.

Bei Auftreten von Fehlern bei der Nachrichtenverarbeitung innerhalb eines Dienstes definiert der SOAP-Standard ein bestimmtes Nachrichtenformat für die Fehlerreaktion. Dafür wird eine spezielle SOAP-Nachricht versendet, welche als Nutzdaten ein `Fault`-Element mit den Unterelementen `Code` und `Reason` enthält. Die Unterelemente enthalten jeweils einen Fehlercode und eine kurze Beschreibung des Fehlers.

Der Aufbau der SOAP-Nachrichten spielt beim späteren Überwachen von organisationsübergreifenden Prozessen eine wesentliche Rolle, da der Nachrichteninhalt zum Erkennen von Fehlern bei organisationsübergreifenden Prozessen analysiert werden muss. In diesem Zusammenhang spielt auch die SOAP-Erweiterung WS-Addressing[3] eine große Rolle, da diese Spezifikation diverse Standarderweiterungen für die vom Transportprotokoll unabhängige Lokalisierung von Dienstempfängern sowie diverse Sender- und Empfängerdaten definiert. Aus diesem Grund wird die Spezifikation im Folgenden näher betrachtet.

WS-Addressing: Wie im vorangegangenen Abschnitt beschrieben, definiert SOAP sowohl ein Transportprotokoll als auch ein Nachrichtenformat. Trotzdem sind die meisten relevanten Protokollinformationen zur Kommunikation über SOAP durch das darunterliegende Transportprotokoll wie HTTP beschrieben. Problematisch daran ist, dass ein Dienst innerhalb der WSDL-Beschreibung über seine URI[4] lokalisiert wird. Dies funktioniert in den meisten Fällen nur, solange die Web-Services zustandslos sind und über HTTP angesprochen werden. Sind für den Transport der Nachricht beispielsweise Instanzinformationen von Prozessen relevant oder kommen andere Transportprotokolle zum Einsatz, müssen diese Meta-Informationen über den Endpunkt eines Dienstes entsprechend vorgehalten und in der Nachricht transportiert werden, um den jeweiligen Dienst korrekt anzusprechen zu können [Pap08]. Die vom W3C vorgestellte Spezifikation *WS-Addressing* definiert hierfür eine SOAP-Erweiterung, welche genau diese Schwächen der SOAP-Spezifikation beseitigen soll.

Dazu definiert WS-Addressing zum einen das Konzept der Endpunktreferenz (Endpoint Reference, EPR) und zum anderen das Konzept der Nachrichtenadressierung (Message Addressing Properties, MAP) [MET+10]. Weiterhin wird die Abbildung der Konzepte auf entsprechende SOAP-Header-Blöcke durch die Spezifikation des SOAP-Bindings für WS-Addressing vom W3C detaillierter spezifiziert [HRG06]. Ein EPR als XML-Dokument besteht dabei wie oben beschrieben aus der URI eines Dienstes sowie aus optionalen Meta-Informationen, welche für die Interaktion mit dem jeweiligen Dienst notwendig sind. Diese Meta-Informationen werden jeweils in einem `ReferenceParameters`-Element vorgehalten. Für das zweite Konzept der Nachrichten-Adressierungsinformation definiert WS-Addressing verschiedene Angaben für eine SOAP-Interaktion, welche sich auf das Routing der Nachrichten und auf den Kommunikationspfad beziehen. Dies sind beispielsweise die Header-Blöcke über die Ziel-Adresse `TO`, den Absender `FROM`, den Nachrichten-Identifikator `MessageID` oder auch spezielle das Nachrichten-Routing betreffende Blöcke wie `ReplyTo`, `FaultTo` oder auch

[3] W3C-Spezifiaktion WS-Addressing, siehe [RHG06]

[4] Siehe dazu die RFC 3986: Uniform Resource Identifier (URI), Generic Syntax [BLFM05]

`RelatesTo`. Durch die letzten drei Blöcke lassen sich auch asynchrone oder Publish/Subscribe-Interaktionsmuster verwirklichen, die beispielsweise über mehrere HTTP-Verbindungen realisiert werden.

Für die Analyse von organisationsübergreifenden Interaktionen kann WS-Addressing daher sehr gut eingesetzt werden, da diverse Daten über Empfänger, Sender und auch Nachrichtensequenzen im Kopf einer Nachricht vorhanden sind. Durch den Einsatz von WS-Addressing lassen sich Nachrichten später leichter auf ein Kommunikationsmodell abbilden, als es ohne dessen Einsatz der Fall ist.

2.1.1.2. Enterprise-Service-Bus

Da Web-Services über verschiedene Transportprotokolle angesprochen oder genauso auch verschiedene Spezifikationen benutzen können, reicht der bloße Einsatz von Web-Services alleine nicht aus, um das Ziel der Interoperabilität zu erreichen. Zur Unterstützung der Interoperabilität wird zusätzlich eine Infrastruktur benötigt, welche beispielsweise zwischen verschiedenen Transportprotokollen vermittelt und die Kommunikation zwischen verschiedenen Web-Services sicherstellt. Weiterhin sieht WSDL zusammen mit SOAP und WS-Addressing nur direkte Punkt-zu-Punkt-Verbindungen vor, um konkrete Dienste direkt anzusprechen. Dies ist aber für größere verteilte Systeme problematisch, da es für diese Systeme wünschenswert ist, dass bei Dienstausfällen andere Dienste einspringen oder eine einfache Lastverteilung über verschiedene gleichartige Dienste erfolgt, um die in der Einleitung des Kapitels angesprochenen Flaschenhälse in größeren Systemen zu vermeiden. Als Infrastruktur für die Web-Service-Kommunikation und damit zur Unterstützung der Interoperabilität und der Flexibilität in größeren verteilten Systemen wird im Bereich der Web-Services der *Enterprise-Service-Bus* (ESB) vorgeschlagen [Cha04, Ley05].

Die Aufgaben eines Enterprise-Service-Busses als Nachrichteninfrastruktur einer SOA fasst Josuttis in [Jos08] daher als das *Herstellen von Interoperabilität* zwischen unterschiedlichen Diensten zusammen. Der Bus als Infrastruktur muss dafür sorgen, dass durch *Datentransformation*, entsprechendes *Routing* und Herstellen der *Konnektivität* bei unterschiedlichen Transportprotokollen die Interoperabilität zwischen Diensten sichergestellt wird. Als weitere wichtige Aspekte nennt Josuttis *Sicherheit* und *Zuverlässigkeit* des Nachrichtentransports, bei dem Garantien wie beispielsweise die Ordnungserhaltung bei der Übermittlung eingehalten werden müssen. Weitere Aufgaben eines Enterprise-Service-Busses gibt beispielsweise Papazoglou in [Pap08] an, wo er beschreibt, dass ein Service-Bus beispielsweise die Koordination von langlaufenden Prozessen oder Transaktionen unterstützen muss. Beides sind allerdings eher höherwertige Dienste, die durch entsprechende Dienste wie Transaktions- oder Prozesskoordinatoren auch außerhalb des Busses erbracht werden können. Im weiteren Verlauf der Arbeit wird deshalb ein ESB als die Nachrichteninfrastruktur angesehen, welche durch entsprechendes Routing, Datentransformation und Vermittlung zwischen unterschiedlichen Protokollen und Protokollversionen die Konnektivität der be-

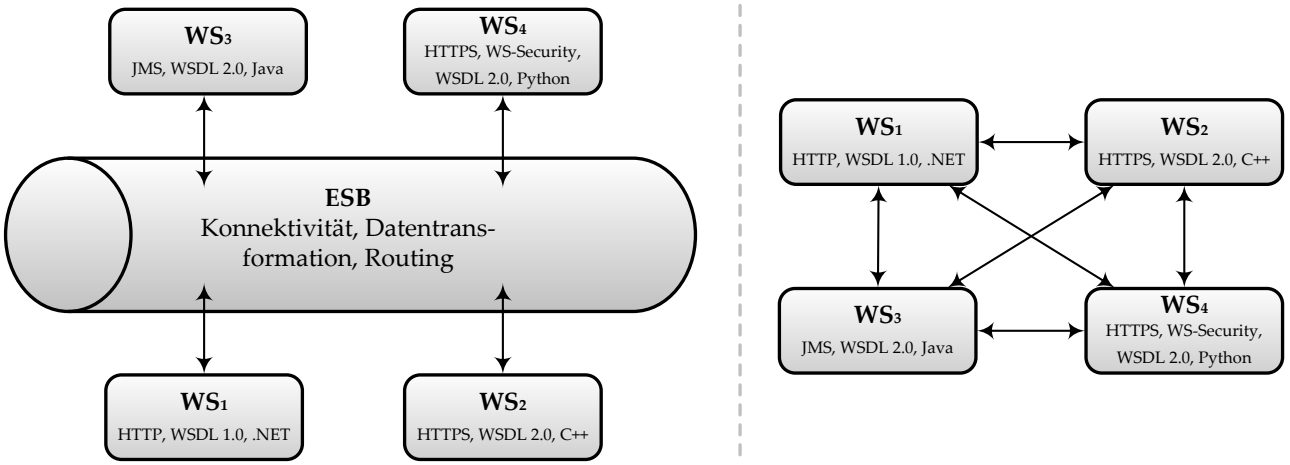


Abbildung 2.4.: Topologie mit und ohne Enterprise-Service-Bus

teiligten Diensten herstellt.

Im Vergleich zu schon bekannten Konzepten der verteilten Systeme, stellt ein Enterprise-Service-Bus im Kern den *Message Broker* einer Message-Oriented-Middleware (MOM) dar [Cha04] und ermöglicht im Vergleich zum Einsatz einer SOA ohne Enterprise-Service-Bus eine vereinfachte Topologie zur Verbindung von Diensten über einheitliche Kommunikationskanäle [MET+10]. Abbildung 2.4 macht den Topologieunterschied noch einmal deutlich. Wenn innerhalb einer Organisation jeder Dienst mit jedem kommunizieren muss, gibt es eine deutlich höhere Anzahl an Schnittstellen, als wenn jeder Dienst nur eine Schnittstelle zum Service-Bus implementieren muss und der Bus dann die Datentransformation und das Protokollrouting zwischen den Diensten übernimmt [MET+10]. Ein ESB stellt somit für größere verteilte Systeme eine Infrastruktur dar, um die Anzahl der Schnittstellen deutlich zu reduzieren und ist daher in heute implementierten serviceorientierten Systemen immer zu finden.

2.1.2. Workflows und Web-Services

Im letzten Abschnitt sind Techniken und Infrastrukturen vorgestellt worden, um einfache Web-Services-Interaktionen durchzuführen. Darauf aufbauend wird nun die *Koordination* von Folgen von Web-Service-Interaktionen betrachtet, bei denen mehrere Web-Services-Aufrufe zu Prozessen komponiert werden. Als Prozess wird an dieser Stelle ein automatisierter und von Diensten unterstützter fachlicher Ablauf verstanden, der von der Definition eines Betriebssystemprozesses zu unterscheiden ist.

Bei der Komposition von Dienstaufrufen zu Prozessen gibt es zwei elementare Sichtweisen, die zum einen als *Orchestrierung* und zum anderen als *Choreographie* beschrieben werden [Pel03]. Eine Orchestrierung ist dabei die klassische Workflow-Sicht, bei der es einen zentralen Koordinator gibt, der einen Prozess aufgrund von implementierter Geschäftslogik steuert und damit den Fluss der Web-Services-Interaktionen entsprechend dirigiert [MET+10].

Im Unterschied zur Orchestrierung mit einem zentralen Prozesskoordinator geht es bei der Choreographie darum, das Zusammenspiel von autonomen Teilnehmern an einem Prozess zu definieren. Eine Choreographie beschreibt daher die Kooperation der beteiligten Dienste, welche genau wissen, welche Rolle sie innerhalb eines Prozesses spielen. Im Vergleich wird klar, dass Orchestrierungen mit zentralen Prozesskoordinatoren eher innerhalb von Organisationen eingesetzt werden. In diesem Fall steuern die Orchestrierungskoordinatoren sogenannte *Geschäftsprozesse*, welche die internen und automatisierbaren Prozesse einer Organisation darstellen [Wes07]. Im Gegensatz dazu ist die Sicht der Choreographien eher bei organisationsübergreifenden Prozessen anzutreffen, wo beschrieben wird, wie sich einzelne Teilnehmer an einem solchen Prozess verhalten. Im Zusammenhang von organisationsübergreifenden Prozessen wird dabei auch von *kooperierenden Prozessen* [MET+10] oder auch *Prozess-Choreographien* [Wes07] gesprochen. In Abbildung 2.5 wird der Zusammenhang zwischen Choreographien und Orchestrierungen durch einen Bestellvorgang veranschaulicht.

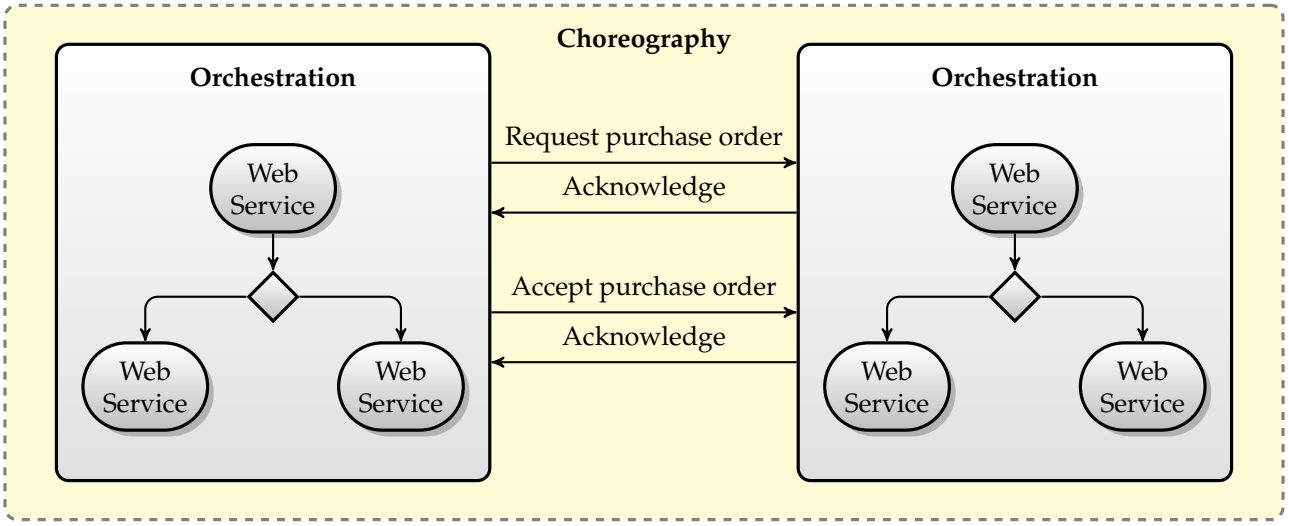


Abbildung 2.5.: Orchestrierung versus Choreographie [Pel03].

Zudem wird anhand der Abbildung 2.5 deutlich, dass das zentrale Merkmal einer Choreographie die *Konversation* zwischen den beteiligten Partnern ist. Eine Konversation beschreibt dabei die konkrete Kommunikation zwischen einzelnen, an der Choreographie beteiligten Diensten. Ihr liegt immer ein gemeinsamer Kontext der beteiligten Kommunikationspartner zugrunde, der den Zustand einer Konversation wiedergibt [ACKM04]. Weiterhin werden die Schnittstellen der Teilnehmer an einer Choreographie als *Behavioral Interfaces* [BDO05b] oder auch als *abstrakter Prozess* [Pel03] bezeichnet. Durch einen abstrakten Prozess wird die Reihenfolge festgelegt, in der die von WSDL beschriebenen Schnittstellen eines Web-Services aufgerufen werden sollen. Ein abstrakter Prozess definiert damit das *beobachtbare Verhalten* eines einzelnen Dienstes. Zaha, Dumas, Hofestede, Barros und Decker sprechen dabei auch von der *lokalen Sicht* auf einen einzelnen Teilnehmer, im Gegensatz zur

globalen Sicht einer Choreographie, welche die Konversationen und das beobachtbare Verhalten von allen Teilnehmern betrachtet [ZDH+06]. Im Gegensatz zur Orchestrierung wird bei den Sichten auf die Teilnehmer einer Choreographie nie das interne Verhalten und damit die Geschäftslogik der Teilnehmer beschrieben; von diesem wird bei Choreographien mit Absicht abstrahiert. Typische Sprachen zur Beschreibung einer Orchestrierung lassen sich deshalb auch direkt ausführen, weil sie das interne Verhalten des Dirigenten einer Orchestrierung mit beschreiben. Im Gegensatz dazu modellieren Sprachen und Modelle für Choreographien nur das beobachtbare Verhalten und sind damit nicht direkt durch eine Software ausführbar. Wie in Abbildung 2.5 zu sehen, kann aber eine Orchestrierung genutzt werden, um das lokale Verhalten eines Teilnehmers innerhalb einer Choreographie umzusetzen.

Im Folgenden werden die Begriffe *Orchestrierung* und *Choreographie* noch weiter vertieft und verschiedene Choreographie-Beschreibungssprachen vorgestellt und verglichen.

2.1.2.1. Orchestrierungen

Wie oben beschreiben, definiert eine Orchestrierung einen Prozess immer *aus der Sicht des Dirigenten* und damit aus der Sicht eines Kooperationspartners. Bezüglich einer Organisation kann dabei eine Orchestrierungs-Software sowohl mit internen Web-Services interagieren als auch mit organisationsfremden und damit externen Web-Services. Der Dirigent einer Orchestrierung kann dabei selbst wieder als Dienst bereitgestellt werden. Er wird damit zu einem komponierten Web-Service, welcher seine eigene Dienstleistung auf Basis von anderen Web-Services bereitstellt und damit die Dienstleistungen von anderen Diensten zu einer neuen Dienstleistung aggregiert. Der Entwicklungsprozess eines komponierten Web-Services wird dementsprechend auch *Dienstkomposition* genannt [ACKM04]. Eine Orchestrierung beschreibt daher alle notwendigen Merkmale der Aggregation von Dienstleistungen und damit den Kontrollfluss, die Datenstrukturen, alle Abhängigkeiten und auch die Reihenfolge der Konversationen von anderen Diensten mit dem Dirigenten.

Eine populäre Sprache zur Beschreibung von Dienstkompositionen im Bereich der Orchestrierungen ist beispielsweise WS-BPEL[5] [AAA+07]. BPEL bietet dabei verschiedene Merkmale, um das Modellieren und Ausführen von Geschäftsprozessen auf Basis von Web-Services zu ermöglichen. BPEL baut dafür auf den Schnittstellenbeschreibungen von WSDL auf und spezifiziert so einen Prozess auf Basis der jeweiligen WSDL-Port-Types der beteiligten Dienste.

2.1.2.2. Choreographien

Im Gegensatz zur Orchestrierung beschreibt eine Choreographie die Kommunikation zwischen autonomen, gleichberechtigten Kooperationspartnern. Dabei spezifiziert eine Choreo-

[5] Abkürzung für *Web Services Business Process Execution Language*, wird weiter verkürzt auch einfach als BPEL bezeichnet.

graphie das Nachrichtenprotokoll und damit die zulässigen Nachrichtenfolgen zwischen den einzelnen Teilnehmern der Choreographie. Wie oben beschrieben, wird dabei nur das beobachtbare Verhalten modelliert. Die entsprechende Geschäftslogik zur Entscheidung, wie auf welche Nachrichten reagiert werden soll und welche der möglicherweise verschiedenen Antwortalternativen auf eine Nachricht ausgewählt wird, ist durch ein Choreographie-Modell nicht definiert. Jeder Teilnehmer einer Choreographie reagiert daher vollständig autonom auf eintreffende Nachrichten. Choreographie-Modelle stellen daher einen Vertrag dar, den unterschiedliche Teilnehmer einer Choreographie einhalten müssen [Pap08], damit ein gemeinsames Prozessziel erreicht werden kann. Zentrales Modellierungsartefakt ist dabei der Nachrichtenaustausch zwischen zwei oder mehreren Web-Services sowie die dabei ausgetauschten Informationen.

Durch eine Choreographie-Beschreibung allein kann allerdings nicht sichergestellt werden, dass sich die Teilnehmer einer Choreographie so wie spezifiziert verhalten. Im Gegensatz dazu kann dies in Orchestrierungen teilweise durch den zentralen Dienst sichergestellt werden, der die jeweiligen Teilnehmer an der Orchestrierung entsprechend kontrolliert, überwacht und im Fehlerfall entsprechend reagiert.

Um Prozess-Choreographien auf Basis von Web-Services zu definieren, sind verschiedene Ansätze wie beispielsweise WS-CDL [BLF+05] oder BPEL4Chor [DKLW07] entwickelt worden[6], die aus verschiedenen Modellierungsansätzen heraus entstanden sind. Beispielsweise liegt der Fokus von WS-CDL auf der Spezifikation von *Interaktionsmodellen*, aus denen später Schnittstellen generiert werden können. Der Fokus von BPEL4Chor hingegen liegt auf der Definition von Modellen, die *bestehende Verhaltensschnittstellen verknüpfen*.

WS-CDL: Die *Web-Services-Choreography-Description-Language* (WS-CDL) ist eine XML-basierte Choreographie-Beschreibungssprache und liegt seit November 2005 als Candidate Recommendation der W3C vor [BLF+05]. WS-CDL modelliert die globale Sicht auf einen Prozess auf Basis von Web-Services und definiert dabei die Rollen der Teilnehmer, die ausgetauschten Informationen und vor allen Dingen die korrekten Interaktionen zwischen zwei oder mehreren Teilnehmern. WS-CDL modelliert über ein XML-Dokument also genau das, was dem oben beschriebenen Choreographie-Begriff entspricht.

Jeder Teilnehmer einer Choreographie erhält dadurch eine Beschreibung des von ihm erwarteten beobachtbaren Verhaltens, welches zur Generierung von Schnittstellen für einen Teilnehmer genutzt werden kann. Im Normalfall wird WS-CDL daher in einem Top-Down-Entwicklungsprozess genutzt, bei dem im ersten Schritt die Choreographie und damit das Interaktionsprotokoll zwischen den Teilnehmern spezifiziert wird [BLF+05]. Im zweiten Schritt wird dann der abgeleitete abstrakte Prozess eines Teilnehmers mit den entsprechenden internen IT-Diensten verknüpft. Es gibt aber auch Ansätze, die aus bestehenden Interaktionen Choreographien modellieren, um diese dann zu analysieren und zu erweitern,

[6] Die beiden Ansätze sind hier nur stellvertretend für diverse andere Modellierungsansätze genannt, die im späteren Verlauf dieses Abschnitts noch aufgeführt werden.

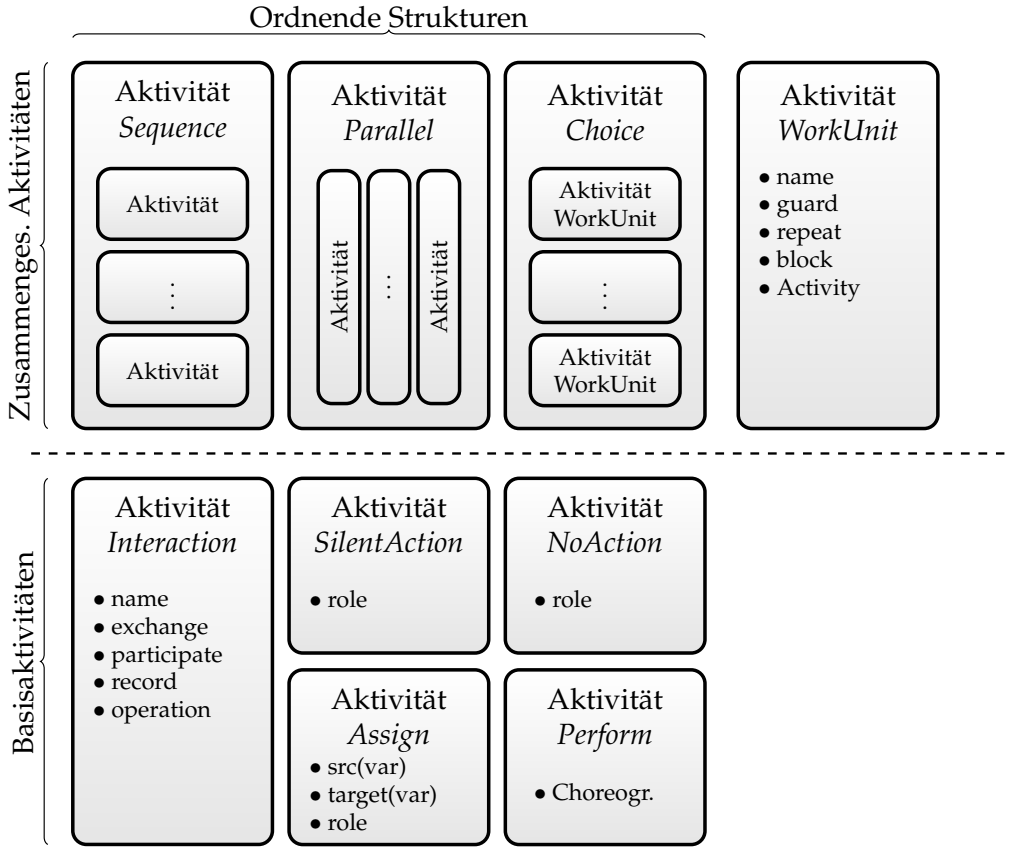


Abbildung 2.6.: WS-CDL-Aktivitäten; nach [BDO05a]

einer größeren Anzahl an Teilnehmern zugänglich zu machen oder sie einfach zu optimieren [DR07]. Eine grafische Notation zur Modellierung wird in WS-CDL nicht vorgesehen. Dies wird in [BDO05a] und [DOZ06] kritisiert, da so der Einsatz von WS-CDL in frühen Entwurfsphasen einer Choreographie-Modellierung erschwert wird. Allerdings existieren bereits Tools wie π4SOA[7], welche die Arbeit mit WS-CDL in den verschiedenen Entwicklungsphasen auch mit grafischen Werkzeugen unterstützen.

Mit Unterstützung von WS-CDL lassen sich somit der Kontrollfluss von Aktivitäten innerhalb einer Choreographie und damit die funktionalen Anforderungen einer Choreographie definieren. Non-funktionale Anforderungen werden dabei nur teilweise wie beispielsweise durch Zeitschrankendefinitionen unterstützt. Um eine Choreographie auf Basis von WS-CDL zu modellieren, wird ein XML-Dokument definiert, welches eine Menge von Aktivitäten enthält, die dann von einem oder mehreren Teilnehmern einer Choreographie ausgeführt werden. Diese Aktivitäten sind später wichtig für die Überprüfung, ob sich Teilnehmer einer Choreographie so verhalten wie sie sollen, da sie das Verhalten der Dienste untereinander definieren. WS-CDL unterscheidet dabei drei Aktivitätstypen, nämlich die in Abbildung 2.6 dargestellten *Basisaktivitäten*, die *ordnenden Strukturen* und die *WorkUnits*.

Die Basisaktivitäten werden weiterhin in zwei Klassen unterteilt, und zwar in die *beobacht-*

[7] `http://pi4soa.sourceforge.net`

baren Aktivitäten wie `Interaction` und die *nicht beobachtbaren* Aktivitäten wie `NoAction`, `SilentAction` und `Assign`. Die `Interaction`-Aktivität modelliert dabei die Nachrichten, die von einem Web-Service zu einem anderen gesendet werden. Die Nachrichten werden durch einen Namen, die Rollen der Teilnehmer, die aufgerufenen Web-Service-Schnittstellen und die ausgetauschten Informationen charakterisiert. Die nicht beobachtbaren Aktivitäten wie `NoAction` und `SilentAction` beschreiben, ob ein Teilnehmer entweder keine Aktivität ausführt oder die Aktivität still ohne Effekt auf den Rest der Choreographie ausgeführt wird. Die `Assign`-Aktivität wird benutzt, um Variablenzuweisungen innerhalb des Choreographie-Modells zu beschreiben. Diese Variablen werden beispielsweise genutzt, um Schleifen und deren Bedingungen zu definieren. Die letzte bisher noch nicht erwähnte Aktivität ist die `Perform`-Aktivität, welche zum Inkludieren von weiteren Choreographie-Beschreibungen genutzt wird und damit der Definition von Subprozessen dient.

Weiterhin gibt es drei ordnende Aktivitätstypen, und zwar sind dies `Sequence`, `Parallel` und `Choice`, welche eine oder mehrere Subaktivitäten enthalten. Eine Sequenz definiert dabei, dass eine oder mehrere Subaktivitäten seriell ausgeführt werden müssen, und zwar in der angegebenen Reihenfolge. Eine `Parallel`-Aktivität definiert Subaktivitäten, die in beliebiger Reihenfolge oder auch gleichzeitig ausgeführt werden. Eine `Choice` und damit eine Auswahl beschreibt Unteraktivitäten, von denen genau eine aus einer Menge von Alternativen ausgewählt und ausgeführt wird. Da ordnende Aktivitäten den Kontrollfluss ihrer Unteraktivitäten definieren, wird im weiteren Verlauf der Arbeit auch von *zusammengesetzten* oder *komplexen* Aktivitäten gesprochen.

Der letzte fehlende Aktivitätstyp aus Abbildung 2.6 ist die `WorkUnit`. Eine Work-Unit definiert hauptsächlich die Richtlinien und Bedingungen zur Ausführung der einzigen enthaltenen Teilaktivität. Beispielsweise lässt sich über eine Eintrittsbedingung einschränken, ob die Teilaktivität ausgeführt wird oder nicht. Im Zusammenspiel mit einer Blockbedingung lässt sich so zusätzlich definieren, ob die Choreographie wartet (blockiert) bis die Eintrittsbedingung erfüllt ist oder ob die Aktivität bei fehlerhafter Eintrittsbedingung übersprungen wird. Zusätzlich lassen sich mithilfe einer Schleifenbedingung auch Schleifen definieren. Ist diese Schleifenbedingung wahr, wird die Work-Unit nach Ausführung wiederholt und damit erneut betreten. Die Work-Unit wird daher durch Variablen beeinflusst, welche durch `Assign`-Aktivitäten gesetzt werden können.

Es gibt noch weitere Aktivitäten wie `Finalize` oder Fehlerbehandlungsaktivitäten, welche aufgerufen werden, wenn Fehler auftreten oder wenn die Aktivitäten der Choreographie beispielsweise durch Transaktionsmechanismen beendet werden sollen. Diese Aktivitäten teilen aber das gleiche Verhalten wie die Work-Unit wie beispielsweise Eintrittsbedingungen. Sie werden daher im Verlaufe der Arbeit synonym verwendet.

BPEL4Chor: BPEL4Chor ist von Decker, Kopp, Leymann und Weske 2007 als Alternative zu WS-CDL vorgestellt worden [DKLW07]. Die Autoren sehen bei WS-CDL das Problem, dass die Sprache eigene Kontrollflussartefakte definiert, die sich schwer auf die Artefakte

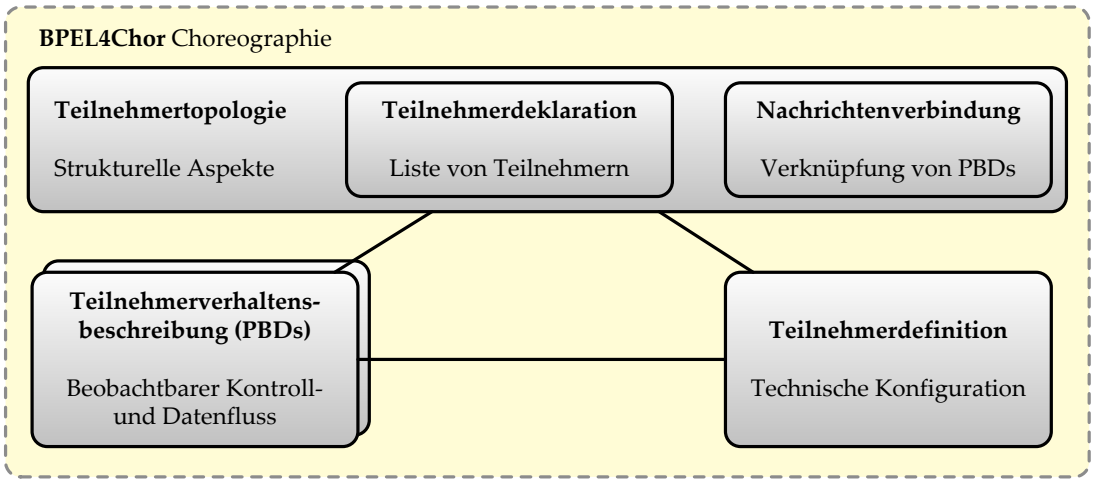


Abbildung 2.7.: BPEL4Chor-Artefakte ([DKLW07])

von WS-BPEL abbilden lassen, um daraus lokale BPEL-Prozesse zu generieren. Allerdings ist das oben bereits genannte Tool π4SOA eben genau dazu in der Lage und kann aus einer WS-CDL-Beschreibung größtenteils die jeweiligen BPEL-Prozess-Skelette der einzelnen Teilnehmer generieren.

BPEL4Chor ist eine Modellierungssprache, deren Fokus auf der Nutzung von (abstraktem) BPEL zur Beschreibung des Teilnehmerverhaltens liegt, wobei zusätzlich noch eine weitere Schicht definiert wird, in der die Verbindung zwischen den einzelnen Schnittstellen der Teilnehmer genauer beschrieben ist. Abbildung 2.7 zeigt die BPEL4Chor-Artefakte im Detail:

- Die *Beschreibung des Teilnehmerverhaltens* erfolgt mithilfe von abstraktem BPEL (siehe auch *WS-BPEL Abstract Processes* [AAA$^+$07]), welches den jeweiligen abstrakten Prozess an der Schnittstelle eines Teilnehmers beschreibt. Als Einschränkung wird dabei aber nur der Teil der WS-BPEL-Spezifikation genutzt, der den beobachtbaren Kontroll- und Datenfluss beschreibt. Genauso wird dabei von konkreten Implementierungen in WSDL und den entsprechenden Datentypen abstrahiert.

- Die technische Beschreibung eines Choreographie-Teilnehmers erfolgt über *konkrete Teilnehmerdefinitionen* in XML, welche den abstrakten Prozess mit den entsprechenden WSDL-Schnittstellen und Datentypen der konkreten Implementierung verknüpfen. Durch diese Kapselung der konkreten Implementierung lässt sich eine Choreographie-Beschreibung einfach wiederverwenden, wenn nur die konkrete Teilnehmerdefinition und damit die Abbildung auf konkrete WSDL-Schnittstellen geändert werden muss.

- Die für Choreographien wichtige globale Sicht wird durch eine Beschreibung der *Teilnehmertopologie* in XML erreicht. In dieser Topologiebeschreibung werden zum einen die Teilnehmerrollen definiert sowie zum anderen die Verknüpfungen der Schnittstellen und Rollen und damit die Nachrichtenverbindungen definiert.

Da für die Beschreibung der abstrakten Prozesse und damit dem Teilnehmerverhalten Abstract-BPEL genutzt wird, lässt sich leicht aus den Beschreibungen eine Abbildung in BPEL realisieren. Damit ist BPEL4Chor - wie der Name schon andeutet - recht eng mit der Orchestrierungs-Sprache WS-BPEL verzahnt. Es lassen sich aber auch gut andere Schnittstellen aus den abstrakten Prozessbeschreibungen generieren, die nicht nur auf BPEL beschränkt sind. Um eine Choreographie auf Basis von BPEL4Chor zu modellieren, müssen also für jeden Teilnehmer der abstrakte Prozess und die Abbildung des abstrakten Prozesses auf die konkreten WSDL-Schnittstellen beschrieben werden. Weiterhin muss die Topologie mit den Teilnehmerrollen und den Nachrichtenkanälen entsprechend formuliert werden.

Die Teilnehmerbeschreibung in Abstract-BPEL enthält dabei alle in BPEL gebräuchlichen Artefakte zur Beschreibung eines ausführbaren Prozesses und zusätzlich noch sogenannte verdeckte (opaque) Aktivitäten, welche von Aktivitäten abstrahieren, die nicht beobachtbar sind oder bei denen die Informationen über die interne Geschäftslogik verborgen werden sollen. Wie in WS-CDL auch, lassen sich diese Aktivitäten in verschiedene Aktivitätstypen wie Basisaktivitäten und ordnende Strukturen einteilen. Basisaktivitäten sind beispielsweise `invoke`, `receive` und `reply`, welche in WS-CDL durch `Interaction` zusammengefasst werden. Weitere Basisaktivitäten sind die verdeckten Aktivitäten oder die Aktivität `empty` und die Variablenzuweisung `assign`, welche zusammen in WS-CDL dann den nicht beobachtbaren Aktivitäten `NoAction`, `SilentAction` oder auch `Assign` entsprechen. In BPEL gibt es in Gegensatz zu WS-CDL noch spezielle Basisaktivitäten, um Fehler anzuzeigen (`throw`), den Prozessablauf zu terminieren (`terminate`) oder auf Aktivitäten zu warten (`wait`). Ordnende Strukturen sind in BPEL mit `sequence`, `switch`, `flow` und `while` angeben, die in WS-CDL über `Sequence`, `Choice`, `Parallel` oder `WorkUnit` ihre Entsprechungen haben.

BPEL4Chor beschreibt damit dieselben Sachverhalte wie WS-CDL, allerdings liegt das globale Modell nur implizit und nicht explizit in einem einzigen XML-Dokument wie in WS-CDL vor. Umgekehrt braucht bei BPEL4Chor nicht aus einem globalen Modell der jeweilige für den Teilnehmer relevante abstrakte Prozess extrahiert zu werden.

Andere Ansätze: Es gibt noch weitere Ansätze zur Modellierung von Choreographien wie beispielsweise das *Web-Service-Choreography-Interface* (WSCI), welches hauptsächlich eine WSDL-Erweiterung darstellt, um die Schnittstellenbeschreibung um strukturelle Informationen wie die Aufrufreihenfolge der Schnittstellen zu erweitern [AAF+02]. Damit lassen sich hauptsächlich abstrakte Prozesse definieren, wie es auch mit *Abstract BPEL* realisierbar ist. Zusätzlich kann in WSCI auch die globale Sicht einer Choreographie über ein *Global Model* definiert werden; allerdings liefert dieses globale Modell nur die Verknüpfung der einzelnen Schnittstellen, definiert aber nicht den Kontrollfluss der Schnittstellen. Damit WSCI benutzt werden kann, müssen die entsprechenden WSDL-Schnittstellenbeschreibungen schon existieren, damit diese um WSCI-Elemente erweitert werden können. WSCI geht damit den umgekehrten Weg wie beispielsweise WS-CDL, welches erst mit dem Protokoll und den Rollen beginnt und danach aus der Choreographie-Beschreibung die entsprechen-

den Schnittstellenbeschreibungen generiert. Wie WS-CDL ist WSCI sehr technisch motiviert und bietet keinerlei grafische Modellierungsmöglichkeiten. Trotzdem bietet WSCI im Gegensatz zu WS-CDL auch Spracherweiterungen, um auch Transaktionen und damit Kompensation im Fehlerfall zu definieren.

Ein weiterer Ansatz zur Modellierung von Choreographien ist das *ebXML Business Process Specification Schema* (BPSS) [DAM06]. BPSS ist ein OASIS-Standard und ein Teil von ebXML, welches eine Sammlung von Spezifikationen für konkreten organisationsübergreifenden Nachrichtenaustausch darstellt[8]. Ein wesentlicher Nachteil ist, dass BPSS ausschließlich bi-laterale Choreographien unterstützt [DAM06]. Decker argumentiert hier, dass damit die Anzahl der möglichen Kollaborationsszenarien limitiert ist [Dec06]. Damit unterstützt BPSS nur sehr wenige und recht einfache Dienstinteraktionsmuster, wie sie beispielsweise in [BDH05] spezifiziert sind. Um organisationsübergreifende Choreographien zu modellieren, ist BPSS also nur bedingt geeignet und wird daher auch nicht als Ausgangspunkt zur Überwachung von Prozess-Choreographien weitergehend betrachtet, da etwa WS-CDL oder BPEL4Chor nicht nur bilaterale, sondern explizit auch multilaterale Choreographien unterstützen.

Im Gegensatz zu WSCI, welches keine grafischen Modellierungsmöglichkeiten bietet, gibt es auch vollständig grafische Modellierungssprachen wie *Let's Dance* von Zaha, Barros, Dumas und Hofstede [ZBDH06] oder auch *DecSerFlow* [AP06] von Aalst und Pesic. Der Nachteil von grafischen Modellen ist aber die Tatsache, dass die ausgetauschten Daten und damit die Nachrichtenformate nicht direkt modelliert werden. WS-CDL oder auch BPEL4Chor modellieren die ausgetauschten Informationen und Daten aber explizit. Dieses Problem lässt sich wie bei der grafischen Spezifikationssprache *BPMN*[9] auch durch verschiedene Annotierungen lösen, aber als Ausgangspunkt für eine Choreographie-Überwachung sind explizite Datenmodelle und explizite globale Modelle in Form von XML einfacher zu handhaben. Auch BPMN lässt sich zur Modellierung von Choreographien einsetzen, wird aber an dieser Stelle genau wie DecSerFlow und Let's Dance als grafische Spezifikationssprache angesehen, die entsprechend weiter in technische Beschreibungen wie WS-CDL oder BPEL überführt werden muss, damit eine technische Überwachung der späteren Implementierung einer Choreographie erfolgen kann.

2.1.3. Implikationen für diese Arbeit

Nach Betrachtung der Ansätze zur Choreographie-Beschreibung ergibt sich zusammenfassend die in Tabelle 2.1 dargestellte Übersicht. Um später Choreographien auf einen korrekten Ablauf hin zu überwachen, eigenen sich also WS-CDL und BPEL4Chor als Grundlage für die Überwachung am besten, da die funktionalen Anforderungen und teilweise

[8] ebXML ist die Abkürzung für die *Electronic Business using eXtensible Markup Language* und ist als Projekt unter der Adresse `http://www.ebxml.org` zu finden.

[9] Kurzform für „Business Process Model and Notation", siehe auch [Obj09].

auch die non-funktionalen Anforderungen wie Zeitrestriktionen in den jeweiligen XML-Dokumenten spezifiziert sind. Bei BPEL4Chor ist der globale Kontroll- und Datenfluss aber nur implizit über die Teilnehmerverhaltensbeschreibungen und der Topologiebeschreibung gegeben. WS-CDL hingegen definiert die globale Sicht auf eine Choreographie explizit und ist damit als Ausgangspunkt für die Überprüfung des beobachtbaren Verhaltens von Prozess-Choreographien direkt einsetzbar. Im weiteren Verlauf der Arbeit werden daher Choreographie-Beschreibungen immer auf Basis von WS-CDL erfolgen. Weiterhin liegt WS-CDL der W3C zur Standardisierung vor, während dies bei BPEL4Chor nicht der Fall ist. Da es darüber hinaus bei der Mächtigkeit der beiden Beschreibungssprachen keine wesentlichen Unterschiede gibt, ist die Wahl zur Beschreibung von Choreographien in dieser Arbeit daher auf WS-CDL gefallen.

Tabelle 2.1.: Analyse der vorhandenen Choreographie-Sprachen

Sprache	**Globales Modell**	**Multilateral**	**Transaktionen**	**Datenmodell**
WS-CDL	✓	✓	-	✓
BPEL4Chor	-	✓	(✓)	✓
WSCI	-	✓	(✓)	✓
BPSS	-	-	-	✓
Let's Dance	✓	✓	-	-
DecSerFlow	✓	✓	-	-

2.2. Überwachung von Prozessanforderungen

Nachdem im vorigen Abschnitt aktuelle Ansätze zur Modellierung von organisationsübergreifenden Prozessen auf Basis von Web-Services-Choreographien vorgestellt wurden, werden jetzt Ansätze vorgestellt, die sich mit der Überwachung von Prozessanforderungen beschäftigen. Zum einen sind dies allgemeine Ansätze aus dem Bereich des Requirements-Engineering und dem verbundenen Monitoring, zum anderen Ansätze zur Überwachung von Orchestrierungen und Choreographien.

2.2.1. Requirements-Monitoring

Ein sehr allgemeiner Ansatz aus dem Bereich des Requirements-Monitorings wird von Robinson in [Rob06] vorgeschlagen. Robinson definiert dabei ein Rahmenwerk namens *ReqMon*, welches im Wesentlichen aus zwei Komponenten besteht: zum einen aus dem Requirements-Monitoring-Rahmenwerk, welches eine Methode zur Definition von Anforderungen, dem Erkennen von Konflikten in Anforderungen und der Analyse von Überwachungsdaten

definiert. Zum anderen schlägt ReqMon die zu verwendenden Werkzeuge vor, welche das Überwachen in Systemen erst ermöglichen. Innerhalb von ReqMon werden Anforderungen mithilfe der zielorientierten Sprache KAOS (siehe [Lam01]) in Form von temporal-logischen Ausdrücken definiert. Das Rahmenwerk definiert weiterhin eine Sprache zur Definition von Sensoren und der Abbildung der Daten von Sensoren auf die Anforderungen. Wie Sensoren in verschiedenen Domänen instrumentiert werden, wird in diesem Ansatz nicht weiter ausgeführt. Daher kann dieser Ansatz nur als genereller Leitfaden gesehen werden, um Überwachungswerkzeuge für Choreographien zu entwickeln.

2.2.2. Überwachung von Orchestrierungen

Sehr eng an den ReqMon-Ansatz angelehnt ist die Arbeit von Dingwall-Smith [DS06], welche ein Rahmenwerk zur Kontrolle von BPEL-basierten Prozessen vorschlägt. Die Anforderungen für BPEL-Prozesse sind auch in diesem Ansatz in zielorientierten KAOS-Ausdrücken definiert. Im Gegensatz zum ReqMon-Projekt definiert Dingwall-Smith die Instrumentierung von Sensoren sehr konkret und führt diese mithilfe von AspectJ[10] aus. AspectJ webt dafür neue Anwendungslogik in bestehende Java-Systeme ein, um darüber Ereignisse generieren und senden zu können; entsprechende Monitore fangen diese Ereignisnachrichten dann später auf. Eine Sprache zur Abbildung der Ereignisse auf die Anforderungen wird auch in diesem Ansatz vorgeschlagen und kann genutzt werden, um automatisch Sensoren für die Laufzeitumgebung zu erzeugen. Problematisch an diesem Ansatz ist, dass Programm-Code in bestehende Anwendungen eingewebt wird, um Ereignisse nach außen zu geben. Dieses Herausgeben von Informationen ist vor allem von Organisationen explizit nicht gewünscht, weshalb in diesen Umgebungen zumeist nur die beobachtbaren Daten herangezogen werden können.

Ein weiterer Ansatz zur Überwachung von BPEL-Prozessen wird von Mahbub und Spanoudakis vorgeschlagen [MS05]. Sie gehen dabei den umgekehrten Weg, indem ausgehend von der Sprache BPEL die Anforderungen extrahiert werden und diese dann in eine formale Repräsentation überführt werden. Die Anforderungen des Prozesses werden dabei in eine auf dem Ereigniskalkül [Sha99] basierende Sprache überführt. Mit Unterstützung der erzeugten Wissensbasis sind sie in der Lage, Nachrichten des Prozesses auf Ereignisse des Kalküls zu überführen, um Fehler bei der Ausführung zu erkennen. Die Idee, die Anforderungen für einen Prozess direkt aus der Prozessbeschreibung zu extrahieren ist zweckmäßig und kann beispielsweise auch zur Extraktion von funktionalen Anforderungen aus einer Choreographie-Beschreibungssprache genutzt werden. Der entwickelte Ansatz beschränkt sich aber nur auf die Sicht des Dirigenten und bezieht nicht die globale Sicht einer Choreographie mit ein.

Einen anderen Weg zur Überwachung von BPEL-Prozessen gehen Guinea und Baresi in

[10] AspectJ als Tool und weitere Beschreibungen dazu sind auch unter `http://www.eclipse.org/aspectj` zu finden.

[BG05]. Sie schlagen ein Rahmenwerk vor, welches Nachrichten in einem BPEL-Prozess anhand von funktionalen und nicht-funktionalen Eigenschaften überwachen kann. Der Ansatz ist nicht formal und basiert auf dem Kommentieren von bestehenden BPEL-Beschreibungen. Die Kommentare werden durch einen Parser ausgewertet und später durch ein Werkzeug als entsprechende Überwachungsaktivitäten zum Prozess hinzugefügt. Dieser erweiterte Prozess wird dann später von der Laufzeitumgebung ausgeführt. Auch dieser Ansatz ist nicht für organisationsübergreifende Prozesse geeignet, da das Einweben von Aktivitäten in die internen Prozesse von Organisationen deren Autonomie verringert und aufgrund der Heterogenität innerhalb der Organisationen auch nicht immer ohne Probleme möglich ist.

Weitere Ansätze, wie beispielsweise [LAP06], [PBB+04], [LJH06] oder [GCN+07] geben keine wesentlich neuen Erkenntnisse in Hinblick auf die bereits vorgestellten Ansätze. Es werden nur andere formale Methoden wie Automaten, temporal-logische Ansätze, Prozess-Algebren, Petri-Netze oder auch Prädikatenlogik für einen Soll-Ist-Vergleich genutzt. Weiterhin definieren sie Methoden zur Instrumentierung von Monitoren, welche zur Laufzeit einen Prozess überwachen. Leider beschränken sich die bisher vorgestellten Überwachungsansätze nur auf zentral koordinierte Prozesse auf Basis von BPEL und beschränken sich dabei auf die Dirigentensicht eines Prozesses.

2.2.3. Überwachung von Choreographien

Es gibt bisher kaum Veröffentlichungen, die sich in diesem Zusammenhang mit organisationsübergreifenden Choreographien beschäftigen. Montali, Mello, Chesani und Torroni [MMCT08] haben für Choreographien einen Ansatz vorgestellt, der ebenso wie der von Mahbub und Spanoudakis auf dem Ereignis-Kalkül basiert. Allerdings beschränkt sich dieser Ansatz eher auf die Definition und dem Erkennen von Fehlern, stellt aber keinen vollständigen Überwachungsansatz dar. Auch wenn der Ansatz in [Mon10] verfeinert worden ist, fehlt beispielsweise ein Vorgehen, wie die Abbildung von einer Menge von Nachrichten auf die Regeln des Ereignis-Kalküls erfolgen soll, oder auch, wie die Nachrichten in organisationsübergreifenden Choreographien überhaupt überwacht werden können. Ein weiteres Manko des Ansatzes ist die Grundlage der Choreographie-Spezifikation. Diese basiert nicht auf dem Standard WS-CDL, sondern auf der grafischen Modellierungssprache DecSerFlow [AP06]. Diese unterstützt beispielsweise nur rudimentär Schleifen, welche in den aktuellen Choreographie-Modellierungssprachen aber durchaus vorgesehen und wünschenswert sind.

Ein anderer Ansatz zum Monitoring von choreographierten Web-Services mithilfe von *Chronicle-Recognition* wird von Le Guillou, Cordier, Robin und Roze vorgeschlagen. Dieser Ansatz wird auch im WS-Diamond-Projekt der EU in leicht abgewandelter Form benutzt ([WD08], [AFG+06]) und basiert auf WS-Coordination [FJ09], um Teilnehmer zu überwachen. Dabei müssen sich zu überwachende Teilnehmer bei einem Koordinator beziehungsweise Monitor registrieren, der wiederum mithilfe eines Protokolls bestimmte Nachrichten

der Teilnehmer zugestellt bekommt. Dieses setzt aber voraus, dass Teilnehmer bestimmte Funktionalität zum Überwachen in ihre eigenen Prozesse implementieren müssen, was im organisationsübergreifenden Fall nicht unbedingt angenommen werden kann.

Ein weiterer wichtiger Bereich zur Überwachung von Prozessen ist das Business-Activity-Monitoring (BAM), welches im Normalfall nur im Bereich der intraorganisationalen Prozesse angewendet wird, um bestimmte Key-Performance-Indikatoren (KPI) von Geschäftsprozessen zu überwachen [McC02]. Dabei geht es allerdings weniger um die funktionalen Anforderungen eines Prozesses als um die Überwachung von Dienstgüte-Parametern wie beispielsweise die Verfügbarkeit, der Durchsatz oder die Reaktionszeiten, deren Grenzen in Service-Level-Agreements (SLA) definiert werden. Wetzstein, Karastoyanova, Kopp, Leymann und Zwink haben dafür einen Ansatz für Choreographien vorgestellt, welcher auf Basis eines Monitoring-Agreement-Dokuments und einer BPEL4Chor-Beschreibung einen organisationsübergreifenden Prozess überwacht [WKK+10]. Sie argumentieren, dass bei einem Top-Down-Entwicklungsprozess zum Entwickeln einer Choreographie auch gleich ein Vertrag über die Monitoring-Daten definiert werden kann. In diesem Monitoring-Agreement wird definiert, für welche Aktivitäten Monitoring-Ereignisse generiert werden müssen, um beispielsweise Aktivitätsveränderungen innerhalb eines abstrakten Prozesses anzuzeigen. In dem Agreement-Dokument wird zusätzlich angegeben, an welches Ziel die Ereignisse entsprechend gesendet werden müssen. Auch dieser Ansatz entspricht im Kern der Idee, dass die Teilnehmer selbst anzeigen, wenn sich entsprechende Änderungen an ihrem (abstrakten) Prozess ergeben. Auch dies setzt voraus, dass Teilnehmer Funktionalität zum Überwachen in ihre eigenen Prozesse implementieren müssen, was im organisationsübergreifenden Fall nicht angenommen werden kann.

2.2.4. Implikationen für diese Arbeit

Eine wesentliche Frage der Arbeit ist die Sicherstellung, dass sich an Choreographien beteiligte Dienste zur Laufzeit genau so verhalten wie es die Spezifikation vorsieht. Dafür muss eine *Laufzeitverifikation* durchgeführt werden, welche von Colin und Mariani in [CM04] wie folgt definiert wird (frei übersetzt):

> „Laufzeitverifikationstechniken werden benutzt, um dynamisch das beobachtete Verhalten eines Zielsystems in Bezug auf gegebene Anforderungen zu verifizieren. Diese Anforderungen sind meistens formal spezifiziert und die Verifikation wird dazu genutzt, um zum einen zu prüfen, ob das beobachtete Verhalten den Eigenschaften genügt, und zum anderen, um explizit Fehlverhalten im Zielsystem zu erkennen und Fehler zu benennen.“

Allerdings lässt sich auf Basis des aktuellen Standes der Technik erkennen, dass noch kein vollständiger Ansatz zur Laufzeitverifikation von Choreographien existiert. Um einen

Ansatz zu entwickeln, sind daher insbesondere Lösungen zu folgenden Herausforderungen zu entwickeln:

- **Ein formales Interaktionsmodell**
  Zur Überwachung von Prozess-Choreographien müssen die zu überwachenden *Anforderungen* vor der Ausführung spezifiziert werden. In der Regel werden dafür die funktionalen Anforderungen einer Choreographie durch ein Interaktionsmodell in Form von WS-CDL beschrieben, welches aber kein formales Model darstellt. Wie oben angedeutet, gibt es eine Vielzahl von Möglichkeiten, um ein formales Prozessmodell aus einer Beschreibung zu extrahieren. In den meisten der bisher genannten Ansätze liegt dabei der Fokus für die formalen Modelle einer Choreographie allerdings im Testen von bestimmten formalen Modelleigenschaften. Sie sind daher weniger für eine effiziente Laufzeitverifikation geeignet. Daher soll in dieser Arbeit ein *formales Modell von Choreographien entwickelt* werden, welches *effizient für die Laufzeitverifikation* von Choreographien eingesetzt werden kann.

- **Nicht-Intrusive Sensoren**
  Aufgrund der Autonomie und der Heterogenität von beteiligten Organisationen können für eine Verifikation von Prozess-Choreographien hauptsächlich nur die beobachtbaren Daten herangezogen werden. Zwar können interne Daten aus Organisationen für die Verifikation herangezogen werden, allerdings ist das Bereitstellen dieser Daten eher die Ausnahme. Die bisherigen Ansätze zur Überwachung von Prozessen respektieren diese Autonomie jedoch nicht, da sie davon ausgehen, dass bestehende interne Prozesse beispielsweise durch Überwachungsaktivitäten angereichert werden können. Eine wesentliche Herausforderung der Arbeit stellt also die *Entwicklung eines nicht-intrusiven Überwachungsmechanismus* dar, welcher die Daten für die Überwachung rein aus den beobachtbaren Daten liefern kann.

- **Effizienter Soll/Ist-Vergleich**
  Eine schnelle Erkennung von Abweichungen des Sollverhaltens ermöglicht es Organisationen, direkt auf Fehler zu reagieren, ohne dabei auf langwierige Analysen von Protokolldateien zurückzugreifen. Damit die Fehlererkennung realisiert werden kann, muss auf Basis des formalen Modells und der Überwachungsdaten ein *effizientes Verfahren* entwickelt werden, welches zur Laufzeit einen *Soll/Ist-Vergleich* durchführen kann und damit Abweichungen des Laufzeitverhaltens vom Modell erkennt.

Die Herausforderungen werden im späteren Verlauf der Arbeit noch weiter verfeinert und entsprechende Lösungen dazu erarbeitet.

2.3. Wiederherstellungsmechanismen für Prozesse

Auch bei Erreichung der eben beschriebenen Ziele zur effizienten Fehlererkennung für Prozess-Choreographien bleibt immer noch die Frage offen, wie ein organisationsübergreifender Prozess im Fehlerfall bei der Behebung von verschiedenen Fehlern unterstützt werden kann. Ein Prozess besteht dabei aus mehreren Aktivitäten, welche verschiedene Änderungen an Daten in verschiedenen Systemen vornehmen oder auch Aktionen in der realen Welt anstoßen können. Kann ein Prozess beispielsweise aufgrund eines Fehlers nicht weiter fortgeführt werden, sind aber teilweise schon Daten verändert und auch Prozesse in der realen Welt angestoßen worden. Eine Fehlerbehebung kann zwar auch durch Fehlersignalisierung von Hand gelöst werden, aber in der Literatur wird zur Automatisierung dafür das Paradigma der Transaktionen vorgeschlagen. Im Folgenden wird daher der Bereich der Transaktionskontrolle in serviceorientierten Architekturen und verteilten Systemen im Allgemeinen beleuchtet.

2.3.1. Das klassische Transaktionskonzept

Das klassische Transaktionskonzept wurde ursprünglich für den Bereich der Datenbanken entwickelt und umschreibt das Zusammenfassen von mehreren Datenbankaktivitäten zu einer logischen Einheit, welche beim Ausführen gewissen Eigenschaften genügen muss. In Datenbanken muss eine Transaktion dabei den ACID-Eigenschaften genügen, die für *Atomicity*, *Consistency*, *Isolation* und *Durability* stehen [Gra81]. ACID bedeutet in Kurzform, dass eine Transaktion entweder ganz oder gar nicht ablaufen muss (Atomicity), die Änderungen nach erfolgreichem Abschluss einer Transaktion dauerhaft in eine Datenbasis einfließen (Durability) und dass die Datenbasis vor und nach Ausführen einer Transaktionen einen konsistenten Zustand repräsentiert (Consistency). Eine weitere wichtige Eigenschaft ist die Isolation, welche vor allen Dingen beim nebenläufigen und verzahnten Ausführen von Transaktionen zur Verhinderung von Mehrbenutzeranomalien wirkt.

Innerhalb von Datenbanken sind die Aktivitäten von verschiedenen Transaktionen als Lese- und Schreiboperationen auf Datenelemente aufzufassen, welche auch automatisiert über Logging- und Recovery-Mechanismen im Fehlerfall koordiniert zurückgesetzt werden können [HR01]. Es gibt allerdings nicht nur Datenbankoperationen, sondern auch andere Aktivitäten eines Systems, die beispielsweise einen Effekt auf die reale Welt haben. Insgesamt lassen sich die Aktivitäten in einem System grob in drei Gruppen unterteilen [GR93]:

- *Ungeschützte Aktivitäten*: Diese Aktivitäten genügen keinerlei der ACID-Eigenschaften.
- *Geschützte Aktivitäten*: Geschützte Aktivitäten genügen den ACID-Eigenschaften und können bei Fehlern zurückgesetzt werden. Die Resultate geschützter Aktivitäten werden erst nach Beenden der Aktivität sichtbar.

- *Reale Aktivitäten*: Diese Aktivitäten haben eine Auswirkung auf die reale Welt und lassen sich unter Umständen schwer rückgängig machen. Dies bedeutet zwar, dass ausgeführte reale Aktivitäten dauerhaft sind, aber nicht unbedingt reversibel. Aus diesem Grunde ist es schwer, reale Aktivitäten durch einen Transaktionskontext mit der Atomaritätseigenschaft zu kapseln. Ist die reale Aktivität[11] ausgeführt und die zugehörige Transaktion schlägt fehl, ist ein automatisches Zurücksetzen unter Umständen nicht möglich. In diesem Fall müssen Kompensationstransaktionen die Effekte der Aktivitäten logisch ausgleichen.

Ein Informationssystem muss bei einer Transaktion also verschiedene Maßnahmen ergreifen, um unterschiedliche Aktivitäten zu einer Transaktion zusammenfassen zu können. Eine *Transaktionsverwaltung* übernimmt dabei zum einen die Garantie für die korrekte Ausführung von nebenläufigen Transaktionen und zum anderen Garantien bei einem Systemausfall oder Abbruch von Transaktionen. Bei Systemen mit Effekten auf die reale Welt wie bei einem Bankautomaten muss die Transaktionsverwaltung zusätzlich dafür sorgen, dass beispielsweise die Aktivität des Geldauszahlens (die Realweltaktion) so lange verzögert wird, bis das Konto belastet ist.

2.3.2. Verteilte Transaktionen

Aktivitäten einer Transaktion müssen wie im letzten Abschnitt beschrieben nicht nur auf ein System wie beispielsweise eine Datenbank beschränkt sein, sondern können auch auf größere verteilte Systeme verteilt werden. Sollen bei der Verteilung von Aktivitäten einer Transaktion auf mehrere Rechnerknoten auch die ACID-Eigenschaften eingehalten werden, müssen zur Sicherstellung der Atomarität Commit-Protokolle definiert werden, welche sicherstellen, dass alle Teilnehmer einer Transaktion ihre Änderungen entweder ganz oder gar nicht abschließen. Im Folgenden werden dafür jetzt kurz die bekanntesten Konzepte und Modelle für verteilte Transaktionen eingeführt, da diese auch bei den später betrachteten Web-Services-Transaktionen benutzt werden.

2.3.2.1. Das Zweiphasen-Commit-Protokoll

Zur Einhaltung der Atomarität in verteilten Systemen wird das bekannte Zweiphasen-Commit-Protokoll (2PC) genutzt. Das Hauptziel dieses Protokolls ist die Sicherstellung, dass alle Teilnehmer einer Transaktion einheitlich entweder erfolgreich beenden oder abgebrochen werden. Dazu sieht das Protokoll einen zentralen Koordinator und verschiedene Teilnehmer als Rollen vor. Der Protokollablauf erfolgt dabei in zwei Phasen und wird wie folgt durchgeführt [BN96]:

[11] Eine reale Aktivität ist beispielsweise der Abschuss einer Rakete, welcher nicht mehr reversibel ist.

1. **Start der Phase 1**: Um eine Transaktion abzuschließen, sendet der Koordinator eine `REQUEST-TO-PREPARE`-Nachricht an alle Teilnehmer und wartet danach auf die Wahl der Teilnehmer. Jeder Teilnehmer wählt eine Antwort aus und sendet sie dem Koordinator zurück. Dabei sendet ein Teilnehmer `PREPARED`, wenn er die eigene Aktivität erfolgreich beenden kann. Umgekehrt sendet er `NO`, wenn der Teilnehmer aufgrund eines Fehlers die eigene Aktivität nicht abschließen kann. Zusätzlich kann der Teilnehmer die Wahlphase auch zeitlich herauszögern, wenn er überlastet ist.

2. **Start der Phase 2**: Wenn der Koordinator von allen Teilnehmern eine `PREPARED`-Nachricht erhält, entscheidet er auf `COMMIT`. Erhält er eine `NO`-Nachricht beziehungsweise innerhalb eines Timeouts keine Nachricht von einem Teilnehmer, entscheidet der Koordinator auf `ABORT`. Die Entscheidung wird den Teilnehmern mitgeteilt, welche sich dann entsprechend der Entscheidung verhalten und den Erhalt der Entscheidung mit `DONE` quittieren. Nachdem der Koordinator `DONE` von allen Teilnehmern erhalten hat, ist die Transaktion beendet.

Das 2PC-Protokoll lässt sich wie in [BN96] beschrieben für Spezialfälle noch weiter optimieren. Beispielsweise kann ein Teilnehmer die Koordinatorrolle selbst übernehmen, wenn er der einzige Teilnehmer an einer Transaktion ist. Genauso brauchen Teilnehmer, die nur lesend an einer Transaktion teilnehmen, nicht in der zweiten Phase benachrichtigt zu werden, da sie unabhängig von dieser Phase keinen dauerhaften Einfluss auf eine Datenbasis haben.

Problematisch an dem 2PC-Protokoll ist der Übergang von Phase 1 in Phase 2. Wenn ein Teilnehmer nach dem `PREPARED` abstürzt, kann er eine mögliche `COMMIT`- oder `ABORT`-Nachricht nicht mehr entgegennehmen. Die Teilnehmer müssen daher verschiedene Protokollierungsmechanismen durchführen, damit nach einem Neustart des Teilnehmers die Transaktion normal weiterlaufen kann. Genauso problematisch ist der Absturz des Koordinators zwischen den beiden Phasen. Die Teilnehmer sind in dieser Phase darauf angewiesen, dass der Koordinator ihnen die Entscheidung mitteilt. Kann er dies nicht, sind die Teilnehmer blockiert, da sonst die Atomarität nicht mehr eingehalten werden kann. Als Problemlösung dafür wird beispielsweise das Dreiphasen-Commit-Protokoll vorgeschlagen, was durch eine weitere Phase den Effekt des Blockierens abschwächt [SS83].

Eine Anwendung und Spezifikation des 2PC-Protokolls findet sich beispielsweise in der Transaction-Internet-Protocol-Spezifikation (TIP) [LEK98], die als Vorläufer für die später noch vorgestellten Web-Services-Transaktionsprotokolle angesehen werden kann. Genauso findet es sich im Distributed-Transaction-Processing-Standard der X/Open-Group wieder, welcher später noch kurz vorgestellt wird.

2.3.2.2. Lang laufende Transaktionen

In den vorigen Abschnitten ist bisher implizit davon ausgegangen worden, dass Transaktionen im Normalfall sehr schnell ablaufen und damit beispielsweise für die Synchronisation notwendige Sperren auch schnell wieder freigeben. Problematisch wird dies aber, wenn Transaktionen wie in Geschäftsprozessen länger laufen und viele Aktivitäten enthalten. Sperren können dabei sehr lange gehalten werden, was die Gefahr von Verklemmungen bei nebenläufig ablaufenden Transaktionen erhöht [MET+10]. Zur Auflösung dieser Verklemmungen werden Transaktionen zurücksetzt oder auch neu gestartet. Wird dabei eine langlaufende Transaktion zurückgesetzt, spricht Gray vom Problem der *verlorenen Arbeit* [Gra81]. Für diesen Fall wird vorgeschlagen, die ACID-Eigenschaften von Datenbanktransaktionen aufzuweichen, um Änderungen beziehungsweise Sperren von Unteraktivitäten auch schon vor Beendigung einer Transaktion freigeben zu können. Wie bei Aktivitäten mit Bezug auf die reale Welt ist es auch bei langlaufenden Transaktionen vorteilhafter, das Prinzip der *Kompensationstransaktionen* einzusetzen, um die Effekte von vor dem Ende einer Transaktion freigegebenen Daten zumindest logisch wieder rückgängig zu machen [Elm92].

Weiterhin wird in den vorangegangen Abschnitten von einer flachen Struktur von Transaktionen ausgegangen, bei denen eine Transaktion nur aus den Aktivitäten besteht. Um das Problem der verloren gegangenen Arbeit zu minimieren, wird beispielsweise von Moss das Prinzip der geschachtelten Transaktionen vorgestellt, bei der eine Transaktion aus Aktivitäten und weiteren atomaren Subtransaktionen bestehen kann [Mos85]. Dabei gibt es als Variationen die *geschlossen geschachtelte Transaktion*, welche weiterhin für die Toplevel-Transaktion ACID garantiert. Durch die separate Rücksetzbarkeit von Untertransaktionen ist es jetzt beispielsweise möglich, nur bestimmte Teile einer Transaktion zurückzusetzen, ohne dass die gesamte Transaktion davon betroffen ist. Eine weitere Variante sind die *offen geschachtelten Transaktionen*, wo im Unterschied zu den Subtransaktionen der geschlossen geschachtelten Transaktionen diese ihre Ergebnisse bereits vor dem Ende der Vatertransaktion freigeben können. An dieser Stelle wird die Isolation aufgeweicht, welche durch entsprechende Kompensationstransaktionen für jede Untertransaktion aber gewährleistet werden kann. In Datenbanken kann eine Kompensationstransaktion im gewissen Rahmen sogar automatisiert hergeleitet werden. Bei Geschäftsprozessen ist dies nicht mehr der Fall, die entsprechenden Umkehroperationen müssen bei Definition des Geschäftsprozesses im Voraus definiert werden [Elm92], damit beispielsweise die Effekte in der realen Welt wie das Auszahlen von Geld am Bankautomaten kompensiert werden können.

Darüber hinaus gibt es noch weitere Modelle wie *Sagas* [GMS87] oder auch deren Weiterentwicklung *Contracts* [Elm92], allerdings sind die offen geschachtelten Transaktionen dennoch aufgrund des Kompensationskonzepts und durch die Gruppierung in Untertransaktionen am häufigsten anzutreffen [MET+10]. Eine gute Übersicht und ein Vergleich der erweiterten Transaktionsmodelle ist auch in [Sch99] zu finden.

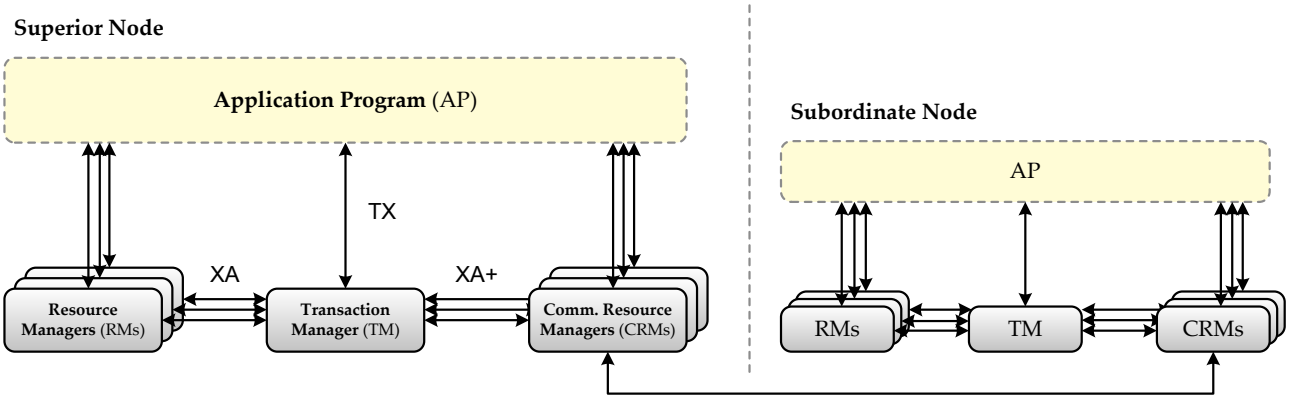


Abbildung 2.8.: Schnittstellen und Rollen für X/Open DTP; nach [Ope96]

2.3.2.3. X/Open Distributed-Transaction-Processing

Der *Distributed-Transaction-Processing*-Standard (DTP) der X/Open-Group definiert ein Modell zur Implementierung einer verteilten Transaktionsverarbeitung. DTP definiert dafür die notwendigen Schnittstellen und die Protokolle sowie vier grundlegende funktionale Rollen [Ope96]:

- das *Anwendungsprogramm*, welches eine Transaktion startet und die Demarkation der Transaktion veranlasst,
- den *Transaktionskoordinator*, der für die Koordination der Teilnehmer einer Transaktion über das 2PC-Protokoll verantwortlich ist,
- eine Menge von *Teilnehmern* an einer Transaktion, die sich beim Transaktionsmanager zur Koordination anmelden und ihre Arbeit gemeinsam beenden müssen,
- eine Gruppe aus Anwendungsprogramm, Transaktionskoordinator und Teilnehmern wird als *Instanz* bezeichnet. Sind mehr als ein Anwendungsprogramm an einer Transaktion beteiligt, so übernehmen *Kommunikationsmanager* die Interaktion zwischen den einzelnen Instanzen.

Der Ablauf einer Transaktion nach DTP startet beim Anwendungsprogramm, welches zum Start einen Transaktionskoordinator kontaktiert. Der Koordinator erstellt dabei einen Transaktionskontext, den er an das Anwendungsprogramm zurückliefert. Dieser Kontext kann jetzt durch das Anwendungsprogramm beim Aufrufen der Teilnehmer mitgesendet werden, damit diese sich darüber beim entsprechenden Koordinator zur Koordination anmelden können. Dieser Registrierungsvorgang wird vom Koordinator bestätigt und danach können die einzelnen Teilnehmer ihre Aufgaben erfüllen. Ist das Anwendungsprogramm mit der eigentlichen Aufgabe fertig, kann der erfolgreiche oder fehlerhafte Ausgang der

Aktivität dem Koordinator mitgeteilt werden. Der Transaktionsmanager benutzt zur Koordination das 2PC-Protokoll, um die Transaktion über alle Teilnehmer konsistent abzuschließen. Die Kommunikation während der Transaktion läuft über standardisierte Schnittstellen. Die Kommunikation zwischen Transaktionsmanager und Teilnehmer läuft über sogenannte *XA*-Schnittstellen, die Kommunikation zwischen Applikation und Transaktionsmanager über *TX*-Schnittstellen und die Kommunikation zwischen den Kommunikationsmanagern und einem Transaktionsmanager über *XA+* Schnittstellen. Das Verhalten und das Aussehen der jeweiligen Schnittstellen ist von der Open Group entsprechend in weiteren Spezifikationen definiert [Ope95, Ope94, Ope92].

Sind mehrere Instanzen an einer Transaktion beteiligt, werden Hierarchien von Koordinatoren aufgebaut. Jeder Koordinator ist für seine Instanz verantwortlich und kommuniziert über die Kommunikationsmanager mit den anderen Koordinatoren. Der Koordinator der initiierenden Instanz ist der Hauptkoordinator, dessen Entscheidungen sich alle anderen Koordinatoren unterordnen müssen. In diesem Falle wird das 2PC-Protokoll zu einem hierarchischen 2PC-Protokoll (siehe auch geschachtelte Transaktionen), bei dem ein weiterer Koordinator sich über die Kommunikationsmanager als Teilnehmer bei einer übergeordneten Transaktion registriert. Abbildung 2.8 zeigt die entsprechenden Schnittstellen und Rollen von DTP.

2.3.3. Verteilte Transaktionen über Web-Services

Auch für den Bereich der Web-Services und damit der serviceorientierten Architekturen gibt es Ansätze zur Transaktionskoordination. Für den Bereich der Web-Services kommt es allerdings ganz auf den Einsatz an, welche Modelle dabei implementiert werden. Kommen Web-Services nur als Nachrichten-Middleware zur Überwindung von Heterogenität zum Einsatz, können auch kurzlebige verteilte Transaktion mithilfe des 2PC-Protokolls durchgeführt werden, um beispielsweise die Atomarität über verschiedene Systeme hinweg zu gewährleisten. Sollen Web-Services im Bereich von SOA aber Geschäftsprozesse implementieren, so sind die entstehenden Abläufe eher langlebig, da zum Abschließen einer Transaktion viele Schritte mit menschlicher Interaktion notwendig sind [ACKM04]. Durch die Langlebigkeit der Prozesse kommt wie bei den offen geschachtelten Transaktionen das Konzept der Kompensation zum Einsatz. Für die beiden Bereiche der kurzlebigen und der langlebigen Transaktionen hat die OASIS[12] sogar drei verschiedene Spezifikationen auf Anregen verschiedener Industriepartner herausgebracht. Namentlich sind dies *WS-Coordination*, das *Web-Services-Composite-Application-Framework* (WS-CAF) und das *Business-Transaction-Protocol* (BTP). Die Letzteren beiden sind auf der OASIS-Web-Seite als abgeschlossen markiert, obwohl teilweise wie bei WS-CAF noch nicht alle vorläufigen Teil-Spezifikationen als Standard herausgebracht sind. WS-Coordination dagegen wird nach wie vor von der OASIS

[12] Die OASIS ist die *Organization for the Advancement of Structured Information Standards*, siehe auch `http://www.oasis-open.org`

weiterentwickelt. Die einzelnen Standards und Spezifikationen werden jetzt im Einzelnen kurz vorgestellt, verglichen und in Bezug auf den Einsatz in Choreographien hin untersucht[13].

2.3.3.1. WS-Coordination

WS-Coordination ist ein erweiterbares Rahmenwerk zur Verwaltung von Kontexten für gemeinsame Aktivitäten [FJ09] und wurde ursprünglich von Microsoft, IBM und Bea entwickelt und später an die OASIS zur Standardisierung übergeben. WS-Coordination bietet als zentrales Element die Spezifikation eines Koordinatordienstes an, welcher aus einem *Aktivierungs-* und einem *Registrierungsdienst* sowie einer Menge von Diensten für verschiedene Koordinationstypen besteht. Der Aktivierungsdienst ist für die Generierung eines neuen Kontextes verantwortlich. Erzeugt eine Anwendung einen neuen Kontext, können mithilfe des Kontextes und entsprechenden SOAP-Nachrichten weitere Ressourcen angesprochen werden, die sich dann beim Koordinator über den Registrierungsdienst zur Koordination anmelden. Die angebotenen Koordinationstypen für WS-Coordination sind dabei für kurzlebige Transaktionen *WS-Atomic-Transaction* (WS-AT) [LW09] und für langlebige Prozesse der Koordinationstyp *WS-Business-Activities* (WS-BA) [FL09]. Die einzelnen Koordinationstypen bestehen dabei jeweils aus einer Menge von Protokollen und entsprechenden Schnittstellenbeschreibungen für die jeweiligen Protokolldienste. Anzumerken ist, dass sich eine Ressource auch für mehr als ein Protokoll registrieren kann, indem sie mehrere Registrierungsnachrichten sendet. Abbildung 2.9 zeigt schematisch einen Koordinator mit den einzelnen Schnittstellen, die er beherrschen muss.

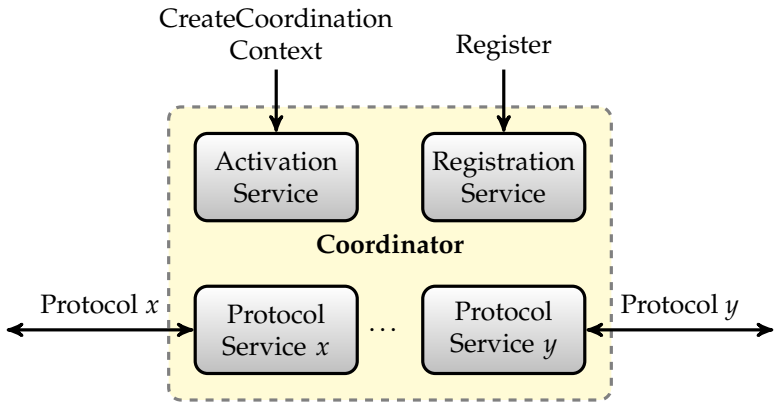


Abbildung 2.9.: Aufbau eines Koordinators nach WS-Coordination (aus [FJ09]).

WS-Coordination spezifiziert mit WS-AT und WS-BA zwar bereits zwei Koordinationstypen, allerdings können auch eigene Koordinationstypen definiert werden, wie es beispiels-

[13] Die einzelnen Standards sind von mir bereits in [Rie05] vorgestellt worden. Die Vorstellung ist an dieser Stelle aber deutlich gekürzt und in Bezug auf neue Versionen der Standards überarbeitet worden. Im Unterschied zum alten Text ist an dieser Stelle eine Analyse des Einsatzes in Choreographien hinzugekommen.

weise in [LP05] durch die Definition von Protokollen für Online-Auktionen gezeigt wird. Im Folgenden werden jetzt die einzelnen Koordinationstypen weiter vorgestellt. Ein weiterer wichtiger Punkt bei WS-Coordination ist die *Interposition* von Koordinatoren. Dabei können Koordinatoren sich auch als Teilnehmer für Aktivitäten anmelden. Mit Unterstützung dieses Prinzips lassen sich verschiedene Koordinationsprotokolle hierarchisch strukturieren.

WS-AtomicTransaction: WS-Atomic-Transaction ist einer der beiden von der OASIS spezifizierten Koordinationstypen für WS-Coordination und ermöglicht die Koordination von kurzlebigen verteilten Transaktionen auf Basis des 2PC-Protokolls [LW09]. Der Koordinationstyp enthält dafür zwei Protokolle, und zwar zum einen das *Completion*-Protokoll, welches zwischen Anwendung und Koordinator ausgeführt wird, und das 2PC-Protokoll, welches zwischen den Teilnehmern und dem Koordinator für die Demarkation einer Transaktion benutzt wird.

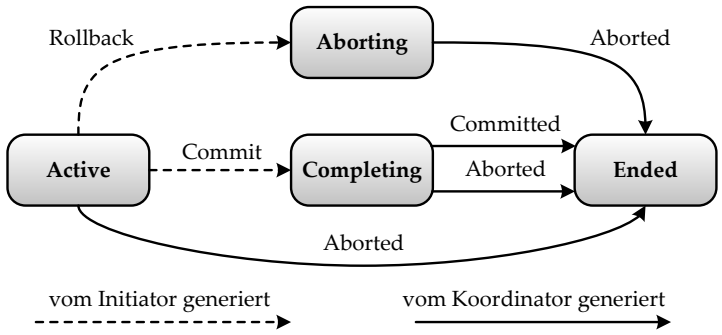


Abbildung 2.10.: WS-AtomicTransaction – Completion (aus [LW09]).

Das Completion-Protokoll wird von Anwendungen und Diensten genutzt, um dem Koordinator mitzuteilen, ob eine Transaktion koordiniert abschließen oder abbrechen soll. In der Spezifikation wird dabei allerdings nicht definiert, wer sich für das Completion-Protokoll registrieren darf. Dies ist stark von der Implementierung und vom Anwendungsfall abhängig. Allerdings wird im Normalfall der Initiator einer Transaktion sich für dieses Protokoll registrieren. Abbildung 2.10 zeigt die Zustände des Completion-Protokolls zwischen Koordinator und dem Teilnehmer, welcher die Demarkation veranlasst.

Beim zweiten Protokoll, welches im Wesentlichen dem 2PC-Protokoll entspricht, wird zwischen *Volatile*- und *Durable*-2PC als Protokollvarianten unterschieden. Die Unterscheidung in volatile und durable bezieht sich dabei auf das Persistenzverhalten der Ressourcen und ist eine der Optimierungen des Protokolls. Teilnehmer, die keine verändernden Effekte auf Datenbestände haben, brauchen später in der zweiten Phase des Protokolls nicht mehr angesprochen werden. Abbildung 2.11 zeigt die Zustände für das 2PC-Protokoll und die entsprechenden Zustandsübergänge. Anhand der Protokollzustände zwischen Koordinator und einer individuellen Ressource lassen sich die beiden Phasen des 2PC-Protokolls gut erkennen. In der ersten Phase sendet der Koordinator eine `Prepare`-Nachricht und die Res-

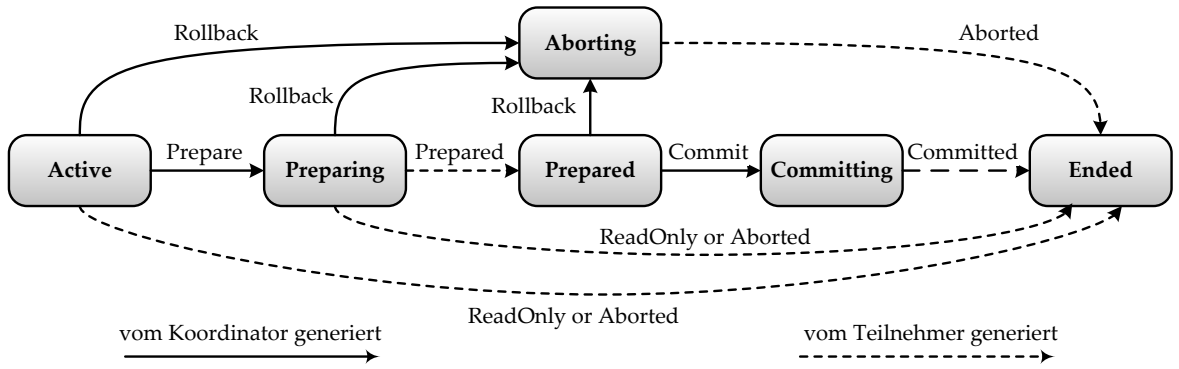


Abbildung 2.11.: WS-AtomicTransaction – 2PC (aus [LW09]).

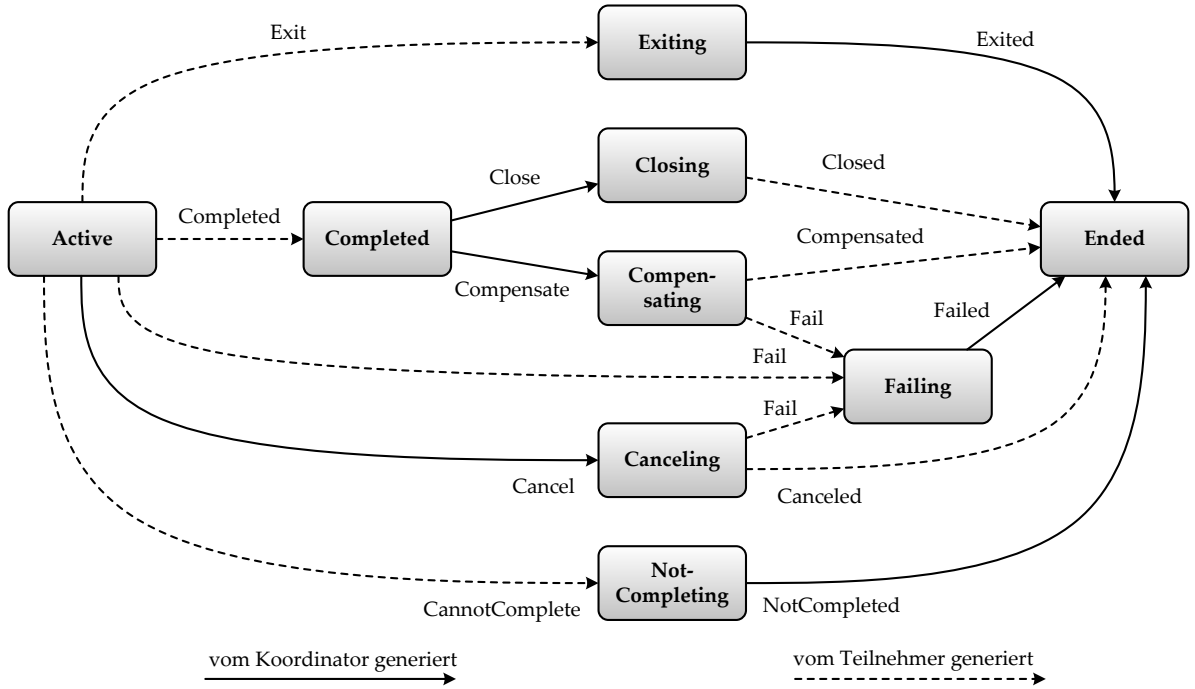
source antwortet mit `Prepared` für Durable-2PC oder `ReadOnly` für Volatile-2PC; im Fehlerfall wird entsprechend `Aborted` gesendet. Bis zum Prepared-Zustand kann der Koordinator die Transaktion auch mit einer `Rollback`-Nachricht noch abbrechen. Ist der Prepared-Zustand erreicht, beginnt Phase zwei des Protokolls, bei dem der Koordinator ausgehend von allen `Prepared`-Nachrichten der Ressourcen entscheidet, ob die Transaktion abgeschlossen wird oder nicht. Wird auf `Commit` entschieden, quittiert eine Ressource diese Nachricht mit `Commited`; bei `Rollback` entsprechend mit `Aborted`.

WS-BusinessActivities: Im Unterschied zu dem vorigen Koordinationstyp behandelt WS-BA langlebige Transaktionen und nutzt dabei ein offen geschachteltes Transaktionsmodell [FL09]. Im Kern wird dabei wieder das Prinzip der Kompensationstransaktionen unterstützt. Zusätzlich wird eine dynamische Teilnehmerliste erlaubt, da Teilnehmer selbstständig eine Aktivität verlassen können.

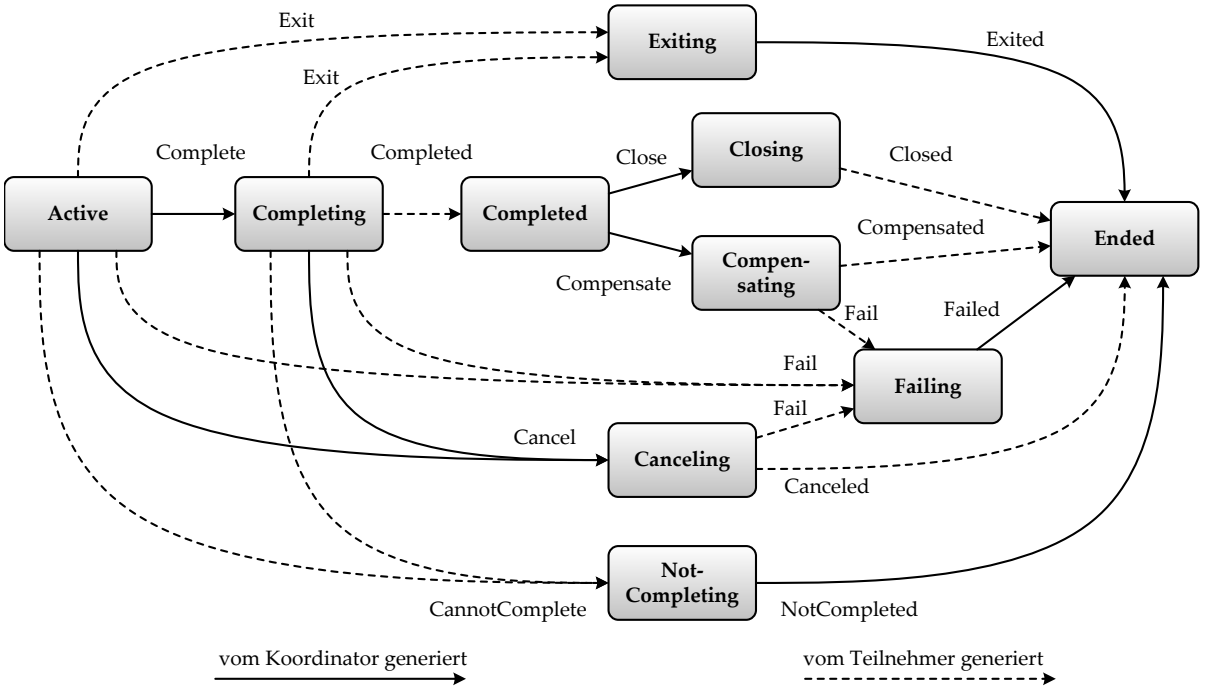
Auch für den Koordinationstyp WS-BA gibt es zwei Protokolle. Dies ist zum einen *BusinessAgreementWithParticipantCompletion* (BAPC) und zum anderen *BusinessAgreementWithCoordinatorCompletion* (BACC). Zusätzlich gibt es für die beiden Protokolle zwei interne Koordinationstypen, nämlich das *AtomicOutcome* und das *MixedOutcome*. AtomicOutcome bedeutet, dass die Eigenschaft der Atomarität für die Ressourcen eingehalten werden muss. Umgekehrt bedeutet ein MixedOutcome, dass der Koordinator die Atomarität nicht einhalten und damit nur ein Teil der Ressourcen erfolgreich abschließen muss, damit die BusinessActivity erfolgreich beenden kann. Beispielsweise kann bei einer Reisebuchung die Mietwagenreservierung optional sein und die Reise trotz fehlendem Mietwagen gebucht werden.

Die beiden angebotenen Protokolle von WS-BA bedienen unterschiedliche Szenarien. Bei BAPC kann eine Ressource selbstständig nach Beendigung einer Aufgabe in den Zustand *completed* gehen, ohne dass ein Koordinator sie dazu veranlasst. Dies ist in Abbildung 2.12(a) illustriert. Bei BACC muss eine Ressource auf Beendigung der eigenen Arbeit warten, bis eine `Complete`-Nachricht vom Koordinator gesendet wird. Dieses ist entsprechend in Abbildung 2.12(b) zu sehen.

Im Unterschied zu WS-AT definiert die WS-BA-Spezifikation aber nicht, wann die je-



(a) WS-BusinessActivity – BusinessAgreementWithParticipantCompletion



(b) WS-BusinessActivity – BusinessAgreementWithCoordinatorCompletion

Abbildung 2.12.: Zustandsdiagramme der beiden WS-BusinessActivity-Protokolle [FL09]

weiligen Protokolle eingeleitet werden. Bei BACC ist beispielsweise nicht definiert, wann der Koordinator die `Complete`-Nachricht an die Teilnehmer senden muss. Bei WS-AT wird dies beispielsweise durch das Completion-Protokoll eindeutig bestimmt. Genauso wenig ist beim MixedOutcome definiert, wie ein Koordinator intern funktioniert. Beim 2PC-Protokoll ist dies für WS-AT eindeutig durch die Atomarität gewährleistet. Beim MixedOutcome hingegen ist nicht spezifiziert, welche Teilnehmer kompensiert und welche erfolgreich abgeschlossen werden müssen. Hier muss also Anwendungslogik in einen Koordinator einfließen, um diese Szenarien anwendungsspezifisch zu entscheiden. Damit wird erstmal jede Implementierung eines Koordinators nach WS-BA anwendungsabhängig. Je nach Implementierung kann dies für organisationsübergreifende Prozesse bedeuten, dass mit dem Koordinator eine zentrale Koordinierungseinheit geschaffen wird, was der Choreographie-Idee von autonomen Teilnehmern widerspricht.

2.3.3.2. WS-CAF – Web-Services-Composite-Application-Framework

WS-CAF [BCH+03a] steht als Abkürzung für *Web-Service-Composite-Application-Framework* und wurde von Arjuna, Fujitsu, IONA, Oracle und Sun entwickelt, bevor es an die OASIS zur Standardisierung übergeben wurde. OASIS hat die Arbeit an der Spezifikation auf Ihrer Web-Seite als „abgeschlossen“ markiert, obwohl nur ein kleiner Teil des Rahmenwerkes als Standard freigegeben wurde. Ist also davon auszugehen, dass WS-CAF nicht mehr weiterentwickelt wird.

Das Rahmenwerk unterteilt sich insgesamt in drei Teile, und zwar sind dies die Spezifikationen zu *Web-Services-Context* (WS-CTX) [BCH+03b], *Web-Services-Coordination-Framework* (WS-CF) [BCH+03c] und *Web-Services-Transaction-Management* (WS-TXM) [BCH+03d]. WS-CTX bietet als Rahmenwerk einen einfachen Dienst an, um gemeinsame Daten und damit einen gemeinsamen Kontext zwischen den Teilnehmern einer Aktivität zu teilen. Etwas vereinfacht betrachtet entspricht WS-CTX dem Aktivierungsdienst eines Koordinators sowie den dazugehörigen Protokollen und Nachrichtenformaten bei WS-Coordination. Das Koordinationsrahmenwerk WS-CF bietet auf WS-CTX aufbauend weitergehende Dienste an, um Anwendungen mithilfe eines Koordinators zu steuern. Dafür gibt es in WS-CF drei zentrale Rollen:

- Der *Koordinator*, an dem sich Teilnehmer registrieren können. Er ist dafür verantwortlich, dass Ergebnis einer Aktivität zu speichern und registrierten Teilnehmern mitzuteilen.
- Der *Teilnehmer*, welcher am Koordinierungsprozess teilnimmt und sich am Koordinator registriert.
- Der *Koordinationsdienst*, welcher bestimmte Aktionen nach einem bestimmten Protokoll am Ende einer Aktivität durchführt. Ein Beispiel wäre ein Protokolldienst für 2PC. WS-CF spezifiziert aber nicht, wie diese Protokolle definiert werden.

WS-TXM bietet dann abschließend die Transaktionsprotokolle für die Koordinierung nach WS-CF an. Im Einzelnen sind dies die Protokolle und Nachrichtenformate für *Web-Services ACID* (WS-ACID), *Web-Services Long-Running Action* (WS-LRA) und *Web-Services Business-Process* (WS-BP), welche im Folgenden kurz weiter eingeführt werden.

WS-ACID: Die erste Protokollspezifikation für WS-ACID implementiert die Koordination von kurzlebigen Transaktionen mithilfe des 2PC-Protokolls. Wie bei WS-Coordination können als Optimierung Teilnehmer ohne dauerhaften Einfluss auf Daten bereits nach der ersten Phase des 2PC-Protokolls ihre Arbeit beenden. Weiterhin kann abweichend von traditionellen ACID-Transaktionen ein Teilnehmer auch selbstständig entscheiden, ob er nach einer erfolgreichen `Prepare`-Nachricht des Koordinators ein Commit oder Rollback seiner eigenen Arbeit veranlasst. Wenn er dieses durchführt, muss er die Entscheidung speichern, um später darauf reagieren zu können. Stimmt die Entscheidung mit der späteren Entscheidung des Koordinators nicht überein, ist das Resultat der gemeinsamen Aktivität nicht mehr atomar und entspricht dann einem in der Spezifikation beschriebenen heuristischen Ergebnis. Der Teilnehmer muss in diesem Fall die Entscheidung des Koordinators umsetzen und nachträglich seine eigene Entscheidung revidieren.

WS-LRA: Im Gegensatz zu WS-ACID spezifiziert WS-LRA ein Koordinationsprotokoll für langlebige Transaktionen. Als Modell für langlebige Transaktionen wird wieder ein offen geschachteltes Transaktionsmodell vorgesehen und dafür muss der Kompensationsvorgang vom Koordinator verwaltet werden. Für das Paradigma der Kompensation führt WS-LRA als Rolle den *Kompensierer* ein. Dafür muss ein kompensierbarer Dienst einen entsprechenden Dienst zur Kompensation registrieren, der im Falle der Kompensation angesprochen wird.

WS-BP: Als letztes Koordinationsprotokoll spezifiziert WS-BP ein Protokoll zur zentralen Koordination von Prozessen, die aus verschiedenen WS-ACID- und WS-LRA-Transaktionen bestehen. Alle Teilnehmer an einer Transaktion nach WS-BP werden dabei in entsprechende Geschäftsdomänen unterteilt. Ein Koordinator nach WS-BP ist dann dafür verantwortlich, ein gemeinsames Ergebnis über die Subtransaktionen zu koordinieren. Um Rücksetz- und Kompensierungsaufwände zu verringern, werden zusätzlich in WS-BP Sicherungspunkte eingeführt, in denen alle Teilnehmer ihren Status und die bisher erledigte Arbeit speichern. Eine Transaktion kann im Fehlerfall ab einem Sicherungspunkt wieder anlaufen. Protokollierung ist daher eine der wichtigsten Aufgaben bei den Teilnehmern. Eine weitere Rolle bei WS-BP nimmt der Teilnehmer ein, der über den Ausgang von WS-BP informiert wird. Diese sogenannten *Terminierungsteilnehmer* registrieren sich bei den jeweiligen Koordinatoren einer Domäne, deren Ergebnis beziehungsweise Beendigung sie mitgeteilt bekommen möchten. Wie bei WS-Coordination und WS-BA wird auch hier nicht definiert, wann die Transaktionskontrolle in einem Geschäftsprozess ausgelöst wird.

2.3.3.3. BTP – Business-Transaction-Protocol

BTP [FDF+04] ist die Abkürzung für *Business-Transaction-Protocol* und wurde ursprünglich von BEA und Choreology entwickelt. Wie für die anderen bisher vorgestellten Transaktionsansätze gibt es auch für BTP mittlerweile eine OASIS-Arbeitsgruppe, welche BTP als OASIS-Standard herausgebracht hat. Die Arbeit am Standard ist aber auf der Web-Seite als „abgeschlossen“ markiert. Genauso ist die Firma Choreology als Hauptinitiator des Standards mittlerweile nicht mehr existent, weshalb weitere Versionen des Standards fraglich bleiben.

BTP definiert dabei ein vom Transportkanal unabhängiges Protokoll, um verschiedene Transaktionstypen zu unterstützen und basiert auf einem koordinatorbasierten Ansatz. Das Ergebnis einer Transaktion wird dabei immer zweiphasig ermittelt. Zur Koordination werden vom Koordinator in der ersten Phase die Nachricht `PREPARE` und in der zweiten Phase `CONFIRM` und `CANCEL` an die jeweiligen Teilnehmer gesendet. Damit trotz eines einzigen Protokolls zwischen ACID-Transaktionen und langlebigen Transaktionen unterschieden werden kann, werden von BTP zusätzlich zwei Transaktionstypen definiert:

- *Atom*: Eine Transaktion nach diesem Typ hält die Atomarität über die Teilnehmer ein.
- *Cohesion*: Bei diesem Typ von Transaktionen ist es möglich, die Atomarität aufzuweichen und gemischte Ergebnisse bei Teilnehmern zuzulassen. Genauso kommt ein offen geschachteltes Transaktionsmodell zum Einsatz, welches wieder Kompensationstransaktionen voraussetzt.

Tabelle 2.2.: Implementierungsmuster von BTP-Teilnehmern; aus [FDF+04]

Implementierungsmuster	**PREPARE**	**CONFIRM**	**CANCEL**
Do-Compensate	do + log	forget	reverse
Validate-Do	validate + log	do	forget
Provisional-Final	do, mark pending	mark final	delete

Zusätzlich sieht BTP im Gegensatz zum klassischen 2PC verschiedene Implementierungsmöglichkeiten für einen Teilnehmer vor, welche in Tabelle 2.2 dargestellt sind. Sie zeigen, wie ein Teilnehmer auf eine der drei Nachrichten reagieren kann. Um beispielsweise *Provisional-Final* zu implementieren, muss ein Teilnehmer in der Lage sein, das Resultat seiner Aktivität in einem schwebenden (provisional) Zustand zu halten. Der Teilnehmer kann dann durch Nachrichten später dazu veranlasst werden, den schwebenden Zustand festzuschreiben (confirm) oder rückgängig zu machen (cancel). Durch die Implementierungsmuster bedingt, muss eine Transaktion mit dem Typ *Atom* nicht unbedingt einer verteilten

ACID-Transaktion entsprechen, weil beispielsweise durch das Do-Compensate-Muster bestimmte Ergebnisse von Aktivitäten bereits sichtbar sind, obwohl die Transaktion noch nicht beendet ist.

Wie bei WS-Coordination lassen sich bei BTP die Koordinatoren zu Hierarchien zusammenfügen und damit die Interposition von Koordinatoren umsetzen.

2.3.4. Analyse der Standards und verwandter Ansätze

Wie im vorangegangen Abschnitt beschrieben, gibt es bereits mehrere Standards zur Koordination von Web-Services-Transaktionen. Es gibt allerdings noch weitere Ansätze, die sich wie die bereits genannten Spezifikationen grob in drei Kategorien einteilen lassen, welche ähnlich zu denen von Schroth, Janner und Hoyer in [SJH08] vorgestellten Kategorien zur Realisierung von organisationsübergreifenden Prozessen zu sehen sind:

- Zum einen gibt es *zentral* gesteuerte Aktivitäten, bei denen eine zentrale Stelle für den Prozessfluss und auch für die Transaktionskoordination verantwortlich ist. Für diesen Bereich, bei dem Prozess- und Transaktionskoordination eng gekoppelt ist, gibt es eine Reihe von Ansätzen und Veröffentlichungen wie beispielsweise [ABDN07, LZ04]. Auch WS-BPEL [AAA+07] ordnet sich in diese Reihe ein.

- Zum anderen gibt es die komplett *dezentralen* Modelle, wie beispielsweise der Ansatz von Türker, Haller, Schuler und Schek, welcher eine optimistische Variante des „Serialization Graph Testings“ benutzt, um eine Transaktion ohne zentrale Steuerung zu realisieren [THSS05]. Sie gehen dabei sehr stark auf Grid-Spezifika ein, welche bei organisationsübergreifenden Prozessen nur indirekt eine Rolle spielen. Ein sehr ähnlicher Ansatz wird auch von Chiasson, McAllister und Slonim in [CMS02] vorgeschlagen, welcher aber nicht auf Web-Services eingeht.

- Eine dritte Kategorie der Transaktionskoordination ist ein Mittelweg aus beiden, in dem sich Koordinatoren durch *Interposition* zu einer Hierarchie zusammenfügen. Dabei gibt es zwar immer noch einen Hauptkoordinator, dieser braucht aber nicht mehr alle Teilnehmer zu kennen und delegiert Entscheidungen an entsprechende Sub-Koordinatoren. Diesen Weg gehen auch die bisher von OASIS vorgeschlagenen Spezifikationen WS-Coordination [FJ09], WS-CAF [LNP06] und BTP [CDF+02] für den Bereich der Web-Services.

Die Ansätze der ersten Kategorie scheiden für die transaktionale Koordination von Choreographien aus, da eine zentrale Stelle zwischen Organisationen, welche einen organisationsübergreifenden Prozess steuert, explizit nicht gewünscht ist. Die Ansätze der zweiten Kategorie sind bisher noch nicht ausgereift, um im Organisationskontext eingesetzt

zu werden. Hier zählt der Einsatz von Standards mehr, da diese die Interoperabilität steigern. Leider sind die in der dritten Kategorie aufgezählten Standards zwar durch Interposition der Koordinatoren flexibler als klassische Verfahren, allerdings gibt es hier für Choreographie-Umgebungen ein Problem. Die Ansätze gehen immer davon aus, dass der Initiator einer Transaktion weiterhin eine kontrollierende Stellung im Prozess besitzt und entscheiden kann, wann eine Transaktion entweder durch Festschreiben oder Zurücksetzen beendet wird. Dies muss in organisationsübergreifenden Prozessen nicht immer der Fall sein.

Wie in den vorangegangen Abschnitten vorgestellt, gibt es zahlreiche Spezifikationen, welche das Interpositionsprinzip implementieren. Bemerkenswert ist, dass OASIS drei Spezifikationen zur Koordination von Web-Services-Transaktionen herausgegeben hat. Die allgemeine Funktion dieser Standards wurde von vielen Autoren analysiert und verglichen sowie an einigen Stellen erweitert [HRR07, SM05, LP05, LF03, Vet06]. Diese Erweiterungen beziehen sich zumeist auf die Rolle des Koordinators selbst und auf Wahl eines Teilnehmers, welcher die Kontrolle über eine Transaktion innerhalb von komplexen Geschäftsprozessen innehat. Dafür werden beispielsweise für WS-BusinessActivities als Erweiterungen Protokolle vorgeschlagen, welche analog zum *Completion-Protocol* von WS-AtomicTransaction arbeiten. Diese Protokollerweiterungen lassen einen Transaktionskoordinator eher wie einen Stellvertreter dieses kontrollierenden Teilnehmers erscheinen und machen den gesamten Prozess von diesem einen Teilnehmer abhängig. Die Rolle des kontrollierenden Teilnehmers ist normalerweise implizit an den Teilnehmer vergeben, welcher den Prozess startet.

Leymann und Pottinger argumentieren, dass es nicht möglich ist, den Status einer Koordination von WS-Coordination abzufragen [LP05]. Sie beschreiben, dass fallspezifische Anwendungslogik zur Koordination notwendig ist, und schlagen deshalb vor, dass ein zusätzlicher *protokollspezifischer Dienst* diese Anwendungslogik enthält. Dieser Dienst bildet dann mithilfe des Koordinators zusammen einen anwendungsspezifischen Koordinator. Sie beschreiben außerdem die Regeln und die Parametrisierung von Koordinatoren am Beispiel von Auktionen, bringen die Koordination leider allerdings nicht in Verbindung mit Transaktionen.

Vogt et al. zielen darauf ab, die Logik der Transaktionskoordination von der Geschäftslogik für einen Prozess zu trennen, und zwar aufseiten des Initiators und des Koordinators. Sie schlagen dafür eine erweiterte Implementierung von WS-Coordination vor, welche einen Koordinator und einen neu eingeführten *Transactor* innerhalb einer transaktionalen Middleware-Komponente bündeln [VZG+05]. Obwohl der Ansatz durch den Transactor den jeweiligen Initiator einer Transaktion von der konkreten Implementierung der WS-Coordination-Protokolle entkoppelt, bleibt die Entscheidung, ob eine Transaktion abgeschlossen oder beendet werden soll, beim Initiator, welcher durch konkrete `Confirm`- und `Cancel`-Nachrichten den Transactor fernsteuert. Die Architektur stellt daher effektiv nur die technische Kommunikation zwischen Initiator und Koordinator bereit, macht aber keine weiteren Vorschläge über die Entkopplung des Initiators in Bezug auf die Entscheidung

über den Ausgang einer Transaktion.

Ein weiterer Ansatz in dieser Richtung wird von Erven et al. vorgestellt [EHHZ07]. Sie betonen die Notwendigkeit, den Koordinator für WS-BusinessActivitites von anwendungsspezifischer Logik zu befreien, um generelle Transaktionskoordinatoren bereitzustellen. Dieses ist vor allem für die Koordination von Transaktionen zwischen heterogenen Teilnehmern von Vorteil. Das vorgeschlagene *Web-Services-BusinessActivity-Initiator-Protocol* erlaubt Anwendungen, WS-Coordination-kompatible Koordinatoren zu kontrollieren. Der Ansatz adressiert damit den in der WS-BA-Spezifikation undefinierten Zeitpunkt, wann der Koordinator seine Protokolle starten soll. Obwohl viele Fortschritte im Einsatz von WS-Coordination im Zusammenhang mit WS-BusinessActivities erzielt werden, bleibt der Initiator weiterhin die kontrollierende Instanz innerhalb einer Transaktion und bietet daher keine neuen Funktionalitäten in Richtung Choreographien.

Es gibt noch weitere marginale Ansätze wie beispielsweise das *Tentative-Hold-Protokol* (THP), welches die Transaktionskoordination vor der eigentlichen Transaktionsverarbeitung unterstützen soll [LEK98]. Es geht dabei hauptsächlich um das Reservieren von Ressourcen im Laufe eines organisationsübergreifenden Geschäftsprozesses und damit der Unterstützung von nicht atomaren Abläufen. Diese Prinzipien sind aber teilweise schon in den aktuellen Standards entsprechend berücksichtigt.

2.3.5. Implikationen für diese Arbeit

Zusammenfassend lässt sich erkennen, dass existierende Ansätze zwar die Limitierungen der bisherigen Web-Service-Transaktionsspezifikationen analysieren und daher vorschlagen, allgemeingültige und nicht anwendungsspezifische Koordinatoren für Transaktionen zu implementieren. Allerdings gehen die bisherigen Ansätze kaum auf die Lösung des Problems ein, dass der Initiator eines Prozesses immer derjenige ist, der auch die Demarkation einer Transaktion veranlasst.

Um Choreographien mit Transaktionsmechanismen bei der Fehlerbehebung zu unterstützen, wird daher in dieser Arbeit ein Ansatz zu entwickelt, welcher insbesondere Lösungen zu folgenden Herausforderungen anbietet:

- **Transaktionale Kontrolle für Choreographien**
  Klassische Koordinationsverfahren für Transaktionen sind in der Regel immer zentral koordiniert. Dabei wird dem Initiator eines Prozesses die Entscheidung überlassen, ob eine Transaktion als Ganzes abgeschlossen oder zurückgesetzt werden soll; er entscheidet damit über die *Demarkation* der Transaktion. In vielen praktischen Szenarien für organisationsübergreifende Prozesse ist allerdings der Initiator eines Prozesses nicht derjenige Teilnehmer, welcher über die Demarkation entscheiden kann. Eine wichtige Zielsetzung für den Einsatz von Transaktionskoordination in Choreographien ist daher die Entwicklung von Verfahren, welche diese Szenarien unterstützen.

- **Anbindung an die Fehlererkennung**
  Im Rahmen der transaktionalen Kontrolle ist vorgesehen, dass Teilnehmer einer Transaktion selbst Fehler an entsprechende Transaktionskoordinatoren melden. Ist ein externer Verifikationsdienst beteiligt, so muss auch dieser in die transaktionale Kontrolle einbezogen werden. Ein weiteres Ziel der Arbeit stellt daher die Kombination der Ansätze aus transaktionaler Koordination auf der einen und der Fehlererkennung auf der anderen Seite dar. Damit kann im Falle eines Fehlers bei der Interaktion im Prozess schon direkt die transaktionale Koordination für den Abbruch der betreffenden Aktivitäten gestartet werden. Das Einbeziehen der Fehlererkennung in die Transaktionale Koordination stellt in diesem Fall eine Optimierung im Fall eines Abbruchs dar, der damit schneller koordiniert erreicht werden kann.

Die Herausforderungen werden im späteren Verlauf der Arbeit noch weiter verfeinert und mit entsprechenden Lösungen unterfüttert.

3. Anforderungen und Analyse

Nachdem im letzten Abschnitt zusätzlich zu den technischen Grundlagen verwandte und aktuelle Ansätze analysiert und grobe Anforderungen für die Arbeit definiert worden sind, werden die Anforderungen in diesem Kapitel weiter verfeinert und konkretisiert. Dies betrifft zum einen die Anforderungen an die Überwachung von Choreographien und insbesondere dem dafür benutzten formalen Model und der dafür notwendigen Infrastruktur sowie zum anderen die Anforderungen an die Behebung von Abweichungen des erwarteten Laufzeitverhaltens.

3.1. Überwachung von Prozess-Choreographien

Zur Überwachung von Prozess-Choreographien müssen verschiedene Aspekte betrachtet werden. Zum einen muss betrachtet werden, welche Techniken für einen Vergleich des Soll- und des Istzustandes genutzt werden können und damit auch, wie Anforderungen formal spezifiziert und zur Laufzeit überprüft werden sollen. In diesem Unterkapitel werden daher verschiedene formale Modelle weiter beschrieben und auf ihre Verwendungsmöglichkeiten zur Laufzeitverifikation hin untersucht. Zum anderen sind auch die Datenerhebung und die Ermittlung des beobachtbaren Verhaltens von Prozess-Choreographien wichtig, weshalb zusätzlich in diesem Teilabschnitt auch die Anforderungen für eine Infrastruktur zur Laufzeitüberwachung ermittelt werden.

3.1.1. Anforderungen an ein formales Interaktionsmodell

Um ein passendes formales Interaktionsmodell zur Überwachung von Choreographien zu spezifizieren, muss als Erstes definiert werden, was überwacht werden soll. Generell betrachtet gibt es funktionale und non-funktionale Bedingungen, welche zur Laufzeit einer Choreographie eingehalten werden müssen. Diese Integritätsbedingungen einer Choreographie sind daher das Ziel der Laufzeitüberprüfung. Die Aufgaben der Integritätskontrolle zur Laufzeit sind somit die Überwachung und die Einhaltung von Integritätsbedingungen, welche sich durch funktionale und non-funktionale Anforderungen an einen Prozess definieren lassen. Härder und Rahm beschreiben in [HR01] ein einfaches Klassifikationsschema für semantische Integritätsbedingungen von Datenbanken, welches sich auch gut zur Einordnung und Herleitung von Arten verschiedener Integritätsbedingungen für Choreographien nutzen lässt:

- **Modellinhärente vs. sonstige (modellunabhängige) Bedingungen**:
  Härder und Rahm beschreiben in [HR01], dass modellinhärente Bedingungen aus den Strukturbeschreibungen des jeweiligen Datenmodells folgen. Bezogen auf Prozess-Choreographien können dafür in verschiedenen Bereichen Integritätsbedingungen definiert werden. Beispielsweise können die *verhaltensbezogenen Aspekte* einer Choreographie durch WS-CDL beschrieben werden. Ein WS-CDL-Dokument spezifiziert dabei funktionale Bedingungen für den Kontrollfluss oder auch non-funktionale Bedingungen wie Zeitschranken einer Choreographie. Genauso lassen sich für die *informationsbezogenen Aspekte* einer Choreographie durch WS-CDL die jeweiligen Nachrichtenformate und Datentypen für den Informationsaustausch definieren. Sonstige und damit nicht modellinhärente Bedingungen lassen sich wie in [MS04] auch als *inhaltliche Integritätsbedingungen* bezeichnen. Diese werden aber durch WS-CDL nicht beschrieben.

- **Reichweite der Bedingungen**:
  Das zentrale Modellierungsartefakt einer Choreographie ist eine ausgetauschte Nachricht zwischen zwei Teilnehmern. Damit lässt sich die Reichweite von Integritätsbedingungen einer Choreographie einfach definieren:

  - **Nachricht**: Bezüglich einer einzelnen Nachricht lassen sich Integritätsbedingungen durch WS-CDL und XML-Schema definieren. Diese Integritätsbedingungen, welche benutzte Datentypen und Wertebereiche innerhalb einer Nachricht betreffen, müssen auch durch ein formales Modell entsprechend beschrieben und zur Laufzeit überprüfbar sein.
  - **Interaktion**: Bezüglich einer Interaktion zwischen zwei Teilnehmern lassen sich Integritätsbedingungen via WS-CDL über den Kontrollfluss definieren. Eine Integritätsbedingung kann beispielsweise definieren, welche Antworten innerhalb einer Interaktion zwischen zwei Teilnehmern erlaubt sind. Genauso lassen sich non-funktionale Bedingungen wie Zeitschranken für eine Interaktion in WS-CDL definieren.
  - **Instanzbasiert**: WS-CDL definiert bezüglich eines Gesamtprozesses den einzuhaltenden Kontroll- und Datenfluss zwischen den Teilnehmern. Sollen aber weitere non-funktionale Bedingungen wie Leistung und Effizienz für die Choreographie definiert werden, so müssen diese zusätzlich zur WS-CDL-Beschreibung definiert werden.
  - **Instanzübergreifend**: Für instanzübergreifende Bedingungen sieht WS-CDL keine Modellierungsartefakte vor. Instanzübergreifende Bedingungen sind aber im Bereich des Geschäftsprozess-Managements wichtig, da beispielsweise mithilfe von Business-Activity-Monitoring verschiedene non-funktionale Key-Performance-Indikatoren überwacht werden müssen. Ein Indikator kann beispielsweise der durchschnittliche Transaktionsdurchsatz sein, welcher instanzübergreifend ge-

prüft werden muss. Genauso lassen sich aber auch funktionale Bedingungen prüfen. Ein Beispiel hierfür ist eine Autovermietung, die verschiedene Sensoren auf ihrem Parkplatz montiert hat (siehe auch [MS04]). Fährt ein Wagen vom Hof, sieht dies ein Sensor in der Ausfahrt und die Inventarliste wird entsprechend aktualisiert. Als funktionale instanzübergreifende Integritätsbedingung kann beispielsweise definiert werden, dass ein Wagen in einer weiteren Instanz des Vermietungsprozesses nicht noch einmal vom Hof fahren kann, ohne vorher zurückgekommen zu sein.

- **Statische vs. dynamische Bedingungen**:
  Härder und Rahm beschreiben statische Integritätsbedingungen als Zustandsbedingungen, welche die zulässigen Zustände beschränken. WS-CDL beispielsweise spezifiziert für Choreographien genau dieses durch den Kontrollfluss und die ausgetauschten Datenformate, allerdings auch nicht mehr. Dynamische Bedingungen, welche Härder und Rahm als zulässige Zustandsübergänge beschreiben, können mit WS-CDL nicht definiert werden. Als Beispiel für dynamische Bedingungen wird in [HR01] ein Gehalt genannt, welches nicht kleiner werden darf, oder auch temporale Bedingungen, welche längerfristige Abläufe betreffen. WS-CDL definiert hierfür allerdings keine Artefakte; solche Integritätsbedingungen zu einer Choreographie-Beschreibung müssen zusätzlich definiert werden. Diese dynamischen Integritätsbedingungen können auch den oben beschriebenen inhaltlichen, nicht modell-inheränten Bedingungen zugeordnet werden.

- **Zeitpunkt der Überprüfbarkeit**:
  Dies können zum einen unverzögerte Bedingungen sein, welche sofort überprüft werden müssen; beispielsweise das ausgetauschte Nachrichtenformat oder der Kontrollfluss. Es können zum anderen aber auch verzögerte Bedingungen definiert werden, welche erst nach Ablauf einer Choreographie überprüft werden. Beispielsweise können bei einer Reisebuchung mehrere Sitze in verschiedenen Airlines reserviert sein, aber für den erfolgreichen Ausgang der Choreographie muss erst am Ende der Reisebuchung sichergestellt sein, dass genau ein Sitz für die Reise reserviert ist.

Die Übersicht zeigt, dass es eine Reihe von Integritätsbedingungen gibt, die nur durch Laufzeitüberwachung überprüft werden können. Zwar können verschiedene Eigenschaften wie die Konsistenz eines Prozessmodells auch vor der Laufzeit geprüft werden, aber dynamische und auch instanzbasierte sowie instanzübergreifende Bedingungen, welche den Kontext des aktuell laufenden Prozesses einbeziehen, können nur zur Laufzeit geprüft werden. Die Übersicht zeigt weiterhin, dass WS-CDL für Choreographien nicht alle Arten von Integritätsbedingungen definieren kann, sondern nur einen Auszug der für den Kontroll- und Datenfluss notwendigen Bedingungen. Sollen weitere Bedingungen für Choreographien definiert werden, so müssen diese zusätzlich zur Choreographie-Beschreibung erfol-

gen. In Analogie zur Konsistenz eines Datenbestandes, welcher durch Integritätsbedingungen und deren Überprüfung sichergestellt wird, lässt sich auch Prozesskonsistenz entsprechend durch Integritätsbedingungen realisieren, die zur Laufzeit überprüft werden müssen. Der Begriff der Prozesskonsistenz bezieht sich dabei auf den Prozess, aber nicht auf das Prozessmodell selbst. Im weiteren Verlauf der Arbeit wird davon ausgegangen, dass entsprechende Prozessmodelle von Choreographien konsistent sind, bevor sie in eine Laufzeitumgebung übertragen werden.

Eine Laufzeitüberwachung von Choreographien muss also in der Lage sein, *alle Arten von Integritätsbedingungen* zu überwachen und entsprechende Fehler zu melden. Ein formales Interaktionsmodell muss weiterhin in der Lage sein, alle Integritätsbedingungen einer Choreographie zu definieren, egal, ob sie modellinhärente oder inhaltliche Anforderungen darstellen oder welche Reichweiten sie haben. Zusätzlich zu den Anforderungen bezüglich der Arten von Integritätsbedingungen gibt es auch noch *technische Anforderungen*, welche sich auch auf die Art des Modells auswirken können. Um non-funktionale Eigenschaften wie beispielsweise Zeitschranken zu prüfen, muss ein Modell mit *quantifizierten Zeiten* umgehen können. Genauso darf beim Prüfen der Bedingungen zur Laufzeit kein Mehraufwand entstehen, um beispielsweise zu vermeiden, dass im schlimmsten Fall eine Überprüfung länger dauert als der eigentliche Choreographie-Ablauf. Eine weitere wichtige Bedingung für den Einsatz zur Laufzeitüberwachung ist auch die *Benutzbarkeit* des verwendeten Modells und damit die Einhaltung von Kriterien wie *Werkzeugunterstützung*, *Ausdrucksmächtigkeit* der Sprache bezüglich der Integritätsbedingungen und *Skalierbarkeit* bei der Laufzeitüberwachung.

3.1.2. Analyse vorhandener Methoden zur formalen Spezifikation

Aus der vorangegangenen Analyse der verwandten Arbeiten in Kapitel 2.2.3 werden jetzt die formalen Beschreibungsmöglichkeiten dieser Arbeiten näher betrachtet und bezüglich der im letzten Abschnitt definierten Anforderungen hin untersucht. Die in den bereits beschriebenen Ansätzen benutzten formalen Spezifikationsansätze sind:

- Der *π-Kalkül*, welcher aufgrund seiner Nähe zu WS-CDL weiter geprüft wird (siehe auch [DOZ06]). Der π-Kalkül kommt beispielsweise auch in [DPW06] zur formalen Beschreibung von Interaktionen zum Einsatz, um damit die Konsistenz von Interaktionsmodellen zu überprüfen.

- Die Methodologie *KAOS* zur Anforderungsanalyse in Softwareprojekten und den damit verbundenen Temporallogiken zur formalen Beschreibung und Verifikation der Anforderungen. In [Rob06] kommen KAOS und die damit verbundenen formalen Modelle in einem Monitoring-Ansatz zum Einsatz, um zur Laufzeit die Einhaltung von Anforderungen zu prüfen.

- Der *Ereigniskalkül*, welcher schon in anderen Orchestrierungsprojekten zur Überwachung von Anforderungen zum Einsatz gekommen ist.

Weitere Ansätze wie Petri-Netze oder Automaten werden zusätzlich am Ende dieses Abschnittes kurz diskutiert und bezüglich der Anforderungen geprüft.

3.1.2.1. Der π-Kalkül

Der π-Kalkül wurde von Milner, Parrow und Walker in [MPW92a, MPW92b] vorgestellt und in [Mil99] weiter vertieft. Er gehört zur Familie der Prozessalgebren, welche Ansätze zur formalen Modellierung von nebenläufigen Systemen bereitstellen. Gegenstand der Betrachtung sind kommunizierende Prozesse, welche miteinander über Kanäle interagieren. Wie auch in [DPW06] beschrieben, können diese Kanäle beziehungsweise deren Eigenschaften benannt werden und als Nachrichten an andere Prozesse versendet werden. Ziel dieser Kanalweiterleitung ist es, einen neu in einen Prozess eingebundenen Teilnehmer in die Lage zu versetzen mit einem bereits bestehenden Teilnehmer zu interagieren. Als Beispiel sei hier ein Onlineversand genannt, über welchen ein Nutzer Waren bestellen kann. Wird die Buchung abgeschlossen, wird über eine weitere Versandfirma das Paket versendet. Dieser Versandfirma muss allerdings vor dem Versand mitgeteilt werden, wie sie den Kunden erreichen kann. Im Sinne des π-Kalküls muss an dieser Stelle erst der Kanal weitergeleitet werden und damit die Information, wie ein Teilnehmer mit einem anderen interagieren kann, wenn die Kontaktmöglichkeiten nicht im voraus bekannt sind. An dieser Stelle ist auch die Nähe der Choreographie-Beschreibungssprache WS-CDL zum Kalkül zu sehen. Der Begriff des Kanals und der Kanalweiterleitung ist elementarer Bestandteil der WS-CDL-Spezifikation.

Tabelle 3.1.: Sprachkonstrukte des π-Kalküls; zusammengefasst aus [MPW92a] und [Par01]

Formel	**Bedeutung**
0	Der leere Agent 0 kann keine Aktivitäten ausführen.
$P_1 \mid P_2$	P_1 und P_2 sind nebenläufige Agenten.
$P_1 + P_2$	Kontravalenz: Entweder wird P_1 oder P_2 ausgeführt.
$!P$	Für jeden Aufruf von P wird eine Kopie von P erzeugt.
$\overline{y}\langle x\rangle .P$	Beschreibt die Ausgabe von x an Kanal $\overline{y}$ mit nachfolgendem P.
$y(x).P$	Beschreibt die Eingabe von x an Kanal y mit nachfolgendem P.
$\tau.P$	Führt eine stille Aktion τ aus und verhält sich danach wie P.
$[x = y]P$	Match: Wenn $x = y$ ist, dann folgt P ansonsten 0.
$(\nu z)P$	Stellt sicher, dass ein neuer privater Kanal z geöffnet wird.

Die Syntax des π-Kalküls ist auch in Tabelle 3.1 aufgelistet. Dabei wird, wie in [Par01] beschrieben, davon ausgegangen, dass es eine unendliche Menge von Namen N gibt, welche als Kommunikationskanäle, Variablen oder Werte angesehen werden. Weiterhin gibt es eine endliche Menge von Agenten A, deren Namen das interne Verhalten der Agenten charakterisieren. Diese Agenten können durch einen Bezeichner aus einer Indexmenge I identifiziert werden. Es gibt dabei Unterscheidungen zwischen Eingabe- und Ausgabeparametern sowie stille Aktionen, welche ohne Interaktion mit der Umwelt ausgeführt werden. Weiterhin lässt sich die parallele Ausführung von Agenten durch $P_1|P_2$ oder durch $\prod_{i\in I}(P_i)$ verallgemeinert darstellen, sowie die Kontravalenz zweier Agenten durch $P_1 + P_2$ oder verallgemeinert durch $\sum_{i\in I}(P_i)$ ausdrücken.

Um den Einsatz des π-Kalküls zu erläutern, folgt ein kleines Beispiel, welches sich wiederum auf das zuvor eingeführte Beispiel eines Onlineversandes bezieht. Für das Szenario gibt es vier Rollen: den Shop S, den Kreditkartenprüfer CC, den Versender V und den Kunden C. Der Kreditkartenprüfer ist in der Lage, eine gegebene Kreditkarte auf Bonität zu prüfen, der Versender versendet die Ware, der Shop nimmt Aufträge entgegen und der Kunde kann Aufträge an den Shop übergeben sowie bestellte Produkte annehmen. Diese einfachen Prozesse lassen sich jetzt leicht mit dem π-Kalkül formalisieren. Für den Kreditprüfer ergibt sich die Formel 3.1:

$$CC = !check(x, c).(\overline{x}\,\langle yes\rangle + \overline{x}\,\langle no\rangle) \tag{3.1}$$

Wird der Kreditprüfer mehrmals via *check* aufgerufen, wird jedes mal durch das ! eine neue Instanz bei einer erneuten Interaktion erzeugt. Zusätzlich akzeptiert der *check*-Kanal zwei Eingabeparameter: Zum einen den Kanal x, welcher als Rückkanal genutzt wird und zum anderen die zu prüfende Kreditkarte c. Nach dem Aufruf von *check* kann der Kreditprüfer eine von zwei möglichen Antworten über den Kanal x zurücksenden und zwar entweder *yes* oder *no*.

Auch der Prozess des Versenders lässt sich darüber formalisieren, wobei der Versender die Adresse für die Lieferung (den Kanal y) und das Produkt p übergeben bekommt, welches er an den Empfänger weiterleitet:

$$V = !send(y, p).\overline{y}\,\langle p\rangle \tag{3.2}$$

Der Shop S lässt sich ähnlich formalisieren:

$$S = !order(z, p, c).\left((\nu w)\,\overline{check}\,\langle w, c\rangle\,|w(r)\right).[r = yes]\,\overline{send}\,\langle z, p\rangle \tag{3.3}$$

Der Shop bekommt über den *order*-Kanal drei Parameter mitgegeben: Den Rückkanal z, das zu bestellende Produkt p und die zu belastende Kreditkarte c. Erfolgt ein Bestellaufruf, wird danach der Kreditprüfer gefragt, ob die Kreditkarte belastet werden darf. Dafür wird mit $(\nu w)\,\overline{check}\,\langle w, c\rangle$ zum einen die Kreditkarte an den Kreditprüfer gesendet, sowie mit (νw) ein neuer Kanal erzeugt, auf dem die Rückantwort erfolgen kann. Parallel dazu wird mit $w(r)$ auf dem neu erzeugten Kanal auf das Resultat r gewartet. Ist das Resultat r po-

sitiv, wird der Versender über seinen Kanal *send* angesprochen und die Versanddetails wie Adresse z und das zu versendende Produkt mitgeteilt.

Ein Kunde C des Shops lässt sich formalisieren, in dem er zum einen die Bestellung über den *order*-Kanal sendet und zum anderen auf seinem Antwortkanal auf das Produkt wartet:

$$C = (\nu z)\, \overline{order}\, \langle z, p, c \rangle\, .z(p) \tag{3.4}$$

Der sich insgesamt ergebende Prozess P ergibt sich damit zu:

$$P = C|S|CC|V \tag{3.5}$$

Decker und Puhlmann zeigen in [DPW06], dass sich der π-Kalkül zur Formalisierung von Interaktionen besser als Petri-Netze eignet, da beispielsweise durch Petri-Netze nicht alle bekannten Interaktionsmuster aus [BDH05] abbildbar sind. Im Gegensatz dazu lassen sich mithilfe des π-Kalküls alle Interaktionsmuster formalisieren. Sie argumentieren aber, dass die Formalisierung der Interaktionsmuster noch entsprechend in die Workflow-Muster (siehe [AHKB03]) integriert werden müssen, damit zur Modellierungszeit die Spezifikation einer Choreographie verifiziert werden kann. Der π-Kalkül ist daher gut zur Formalisierung des Kontrollflusses geeignet. Problematisch sind aber Integritätsbedingungen, die sich auf Nachrichtenparameter beschränken. Zwar können hier durch Erweiterungen auf Prädikatenlogik beliebige Match-Bedingungen (siehe Tabelle 3.1) formuliert werden, aber diese müssen entsprechend auch durch die Werkzeuge unterstützt werden. Genauso lassen sich schwer instanzübergreifende Integritätsbedingungen oder andere inhaltliche Anforderungen formulieren; auch hierfür müssten die Match-Bedingungen entsprechend erweitert werden. Technisch ist dies zwar machbar, schränkt die Benutzbarkeit zur Laufzeitverifikation an dieser Stelle aber etwas ein, wenn keine Werkzeugunterstüzung dafür vorhanden ist. Weiterhin lassen sich quantifizierte Zeiten zwar durch Simulation von Timern nutzen, allerdings ist die Verwaltung der Timer nicht besonders übersichtlich, wie auch in [DPW06] zu sehen. Die Ausdrücke zur Formalisierung von einzelnen Diensten werden dann entsprechend lang, was die Benutzbarkeit weiter einschränkt.

3.1.2.2. KAOS und Temporallogiken

KAOS wird zur zielorientierten Anforderungsanalyse von Softwaresystemen genutzt und steht als Akronym für *Knowledge Acquisition in automated Specification* [DLF93].

Die Methodologie beschreibt unter anderem, wie Systemanforderungen bei der Anforderungsanalyse durch ein Zielmodell definiert und formal spezifiziert werden. Die dabei verwendeten temporalen Operatoren der Spezifikationssprache sind in Tabelle 3.2 dargestellt und entsprechen Formeln linearer temporaler Logik (LTL) und damit einer Erweiterung der Aussagenlogik. LTL wird oft zur Spezifikation und Verifikation von Systemen genutzt und kann Aussagen über die Zukunft von Zustandspfaden machen, wie es auch in der Tabelle

3.2 durch den $\diamond$-Operator gemacht wird. Die Aussage $\diamond P$ sagt beispielsweise aus, dass auf einem Zustandspfad die Aussage P irgendwann wahr wird. Wird so eine Aussage als Ziel im Rahmen des KAOS-Zielmodells definiert, so muss es einen Zustandspfad geben, in dem diese Aussage gilt. Andernfalls ist das Ziel nicht erreicht.

Projekte wie ReqMon [Rob06, Rob10] nutzen diese in der Anforderungsanalyse formal spezifizierten Richtlinien eines Systems zur Überprüfung der Anforderungen zur Laufzeit. Ähnliches schlagen auch Colin und Mariani vor, die zur Laufzeitüberwachung LTL-Formeln und die dazu entsprechend passenden Tools nutzen [CM04].

Tabelle 3.2.: Temporale Operatoren im KAOS-Rahmenwerk; aus [LL02]

Formel	**Bedeutung**
$\diamond P$	P gilt in einem zukünftigen Zustand
$\Box P$	P gilt in allen Zuständen in der Zukunft
$A \Rightarrow C$	In allen zukünftigen Zuständen impliziert A C, also $\Box(A \rightarrow C)$
$A \Leftrightarrow C$	In allen zukünftigen Zuständen ist A äquivalent zu C, also $\Box(A \leftrightarrow C)$
$\bullet P$	P gilt im vorherigen Zustand
$\circ P$	P gilt im nächsten Zustand
$@P$	P ist gerade wahr geworden

Um mit einem KAOS-Zielmodell die Integritätsbedingungen eines Prozesses zu modellieren, wird an dieser Stelle wieder das Beispiel des Onlineversandes aus Kapitel 3.1.2.1 herangezogen und entsprechend nach [LL02] in ein Zielmodell überführt. Das Beispiel soll in diesem Fall weder eine komplette Anforderungsanalyse noch einen fehlerfreien Prozess erzeugen, sondern nur die verwendeten Sprachelemente zeigen und später in Bezug auf die Anforderungen an ein formales Modell analysieren helfen.

Das erste einzuhaltende Ziel des Onlineversenders ist die Kreditkartenprüfung nach Eingang einer Bestellung. Es muss also sichergestellt werden, dass direkt nach dem Eingang der Bestellung als Erstes die Kreditkarte überprüft wird und in der Zwischenzeit keine weiteren Aktivitäten getätigt werden:

Goal: Maintain(CreditCheckIffOrder)
InformalDef: *Nach Eingang einer Bestellung muss die Kreditkarte geprüft werden.*
FormalDef: $order \Rightarrow \circ check$

Die Kreditkartenprüfung kann dabei sowohl negativ als auch positiv sein. Nach dem Zustand der Kreditkartenprüfung muss also entweder der Zustand *yes* oder *no* gelten:

Goal: Maintain(YesOrNoIffCreditCheck)

InformalDef: *Die Kreditkartenprüfung muss ein Ergebnis „yes" oder „no" liefern.*
FormalDef: $check \Rightarrow \circ((yes \wedge \neg no) \vee (\neg yes \wedge no))$

Bei negativer Prüfung muss sichergestellt werden, dass der Versender weder einen Lieferauftrag bekommen noch eine Lieferung versenden darf:

Goal: Maintain(NoSendNoDeliveryIffNo)
InformalDef: *Der Versender darf bei negativer Prüfung weder einen Auftrag bekommen noch ein Paket versenden.*
FormalDef: $no \Rightarrow \Box(\neg send \wedge \neg deliver)$

Genauso muss bei positiver Prüfung als nächster Schritt im Prozess der Versender instruiert werden, an wen er das Paket zu versenden hat:

GOAL: Maintain(SendIffYes)
InformalDef: *Bei positiver Prüfung muss der Versand angestoßen werden.*
FormalDef: $yes \Rightarrow \circ send$

Erhält der Versender via *send* die Versandinformationen, muss er als nächsten Schritt im Prozess direkt das Paket versenden:

GOAL: Maintain(DeliverIffSend)
InformalDef: *Wird ein Versandauftrag angenommen, muss versendet werden*
FormalDef: $send \Rightarrow \circ deliver$

Zusätzlich müssen noch Ziele definiert werden, die ausschließen, dass vor einem *order* oder nach einem *deliver* Aktivitäten innerhalb einer Prozessinstanz auftreten können. Diese Ziele sind aus Gründen der Übersichtlichkeit hier nicht mehr mit angegeben. Entsprechende Tools können jetzt zum einen das Modell prüfen und zum anderen auch zur Laufzeit die Pfadformeln überprüfen, wenn die Korrelation zwischen versendeten Nachrichten und entsprechenden aussagenlogischen Temporalformeln sichergestellt ist. In diesem einfachen Beispiel ist schon erkennbar, dass nur aussagenlogische Formeln genutzt werden und damit zusätzlich Logik in den Überwachungswerkzeugen implementiert werden muss, um bestimmte Aussagen zu tätigen. Prädikatenlogische Ansätze sind an dieser Stelle in der Definition der Integritätsbedingungen deutlich flexibler. Trotzdem lässt sich der Kontrollfluss eines Prozesses gut durch temporale Logik überprüfen.

Obwohl Projekte wie ReqMon [Rob06, Rob10] KAOS als Methodik zur Anforderungsspezifikation und entsprechende Tools für Temporallogiken zur Laufzeit für die Überwachung benutzen, hat die formale Spezifikation mithilfe von Temporallogik einige Nachteile. Montali, Mello, Chesani und Torroni beschreiben, dass es bei temporalen Logiken weniger um die Beschreibung von zeitlichen Abläufen als um die Eigenschaften von Zuständen und deren Änderung in Systemabläufen geht [MMCT08]. Weiterhin argumentieren sie, dass für

die Überwachung von Prozessen aber genau die zeitlichen Abfolgen sowie eine quantifizierte Zeit wichtige Kriterien sind, um Interaktionsmodelle und damit Choreographien zu überwachen. Die Autoren diskutieren zusätzlich, dass Temporallogiken nur Schlussfolgerungen darüber anbieten, was als Nächstes oder irgendwann in der Zukunft passieren kann; Schlussfolgerungen darüber, was vor 60 Sekunden passiert ist, sind damit nur schwer möglich. Es gibt zwar Erweiterungen wie die metrische Temporallogik [AH90], diese sind aber aufgrund ihrer Berechnungskomplexität und dem Problem der Zustandsexplosion nur bedingt für die Laufzeitüberwachung geeignet [Wan04]. Die Benutzbarkeit von LTL in Bezug auf Prozessüberwachung ist damit eingeschränkt.

3.1.2.3. Der Ereigniskalkül

Der Ereigniskalkül ist ein logisches Schlussverfahren, welches von Kowalski und Sergot 1986 vorgestellt wurde [KS86], um Schlussfolgerungen über Ereignisse und Gültigkeitsperioden von dazugehörigen Wahrheitswerten zu ermöglichen. Er basiert auf Prädikatenlogik erster Ordnung und lässt sich dadurch vollständig auf Basis von Horn-Klauseln beschreiben, welche sich beispielsweise durch Prolog-Interpreter direkt ausführen lassen.

Abbildung 3.1.: Funktionsweise des Ereigniskalküls; aus [Sha99]

Die heute gebräuchlichen Varianten des ursprünglichen Ereigniskalküls stellt Shanahan in [Sha99] vor. Dabei beschreibt er, wie in Abbildung 3.1 dargestellt, die generelle Funktionsweise des Ereigniskalküls anhand von drei Bestandteilen: Ein Protokoll gibt die Historie der Ereignisse wieder (*Was passiert wann*), Fluents beziehungsweise Wahrheitswerte (*Was ist wann wahr*) dienen zur Repräsentation von Gültigkeitsperioden, und die Auswirkungen von Ereignissen auf Fluents (*Was bewirken Ereignisse*) werden durch entsprechende Regeln beschrieben. Der Ereigniskalkül verknüpft mit seinen Axiomen diese drei Bereiche und erlaubt dadurch Schlussfolgerungen zu Ereignissen und entsprechenden Gültigkeitsperioden (*Logischer Mechanismus*).

Die Historie der Ereignisse ist dabei wie in [KS86] beschrieben streng monoton wachsend. Ereignisse werden daher nicht aus einer vorhandenen Wissensbasis gelöscht, wenn Gültigkeitsperioden verändert werden sollen, sondern durch Hinzufügen von neuen Erkenntnissen entsprechend verändert. Der Ereigniskalkül nutzt für Schlussfolgerungen das Prinzip der *Negation als Fehlschlag* (negation as failure) [KS86, Cla87], welches bei Nichtvorhandensein von Fakten diese automatisch invalidiert. Damit sind die Reihenfolgen der Ereignisse

innerhalb einer Historie irrelevant, es zählt nur der Zeitpunkt der Ereignisse, welcher einen Einfluss auf die Gültigkeitsperioden von zugehörigen Fluents hat.

Shanahan beschreibt weiterhin, dass sich mithilfe des Ereigniskalküls auf drei verschiedene Arten Schlussfolgerungen ziehen lassen, welche sich auf die gegebenen und gesuchten Informationen beziehen [Sha99]:

- *Deduktion*: Sind Ereignisse und die Auswirkungen der Ereignisse auf Fluents gegeben, kann mithilfe der Ereigniskalkül-Axiome ermittelt werden, welche Fluents zu bestimmten Zeitpunkten gelten. Die Deduktion ist daher zur Überwachung von Integritätsbedingungen sehr gut geeignet, da ermittelt werden kann, ob eine Nachrichtenfolge eines Prozesses in einen gültigen Zielzustand führt. Bei der Deduktion sind in Bezug auf die Überwachung zwei verschiedene Arten von Wissensbasen zu unterscheiden:
  - Eine *konstante* Faktenbasis: Die Berechnung der Auswirkungen auf Fluents durch Ereignisse muss nur jeweils einmal erfolgen, da keine neuen Fakten mehr hinzukommen und alle Fakten zum Anfragezeitpunkt bekannt sind. Dies ist nur der Fall, wenn beispielsweise nachträglich festgestellt werden soll, ob eine bereits abgeschlossene Prozessinstanz so wie vereinbart abgelaufen ist.
  - Eine *dynamisch wachsende* Faktenbasis: Die Auswirkungen auf Fluents durch Ereignisse ändern sich ständig, da immer neue Ereignisse zur Faktenbasis hinzugefügt werden. Der Zustand der Fluents kann sich also bei jedem Hinzufügen eines Ereignisfakts verändern. Dies ist prinzipiell bei der Laufzeitüberwachung immer der Fall und somit müssen nach jedem Hinzufügen eines Ereignisses die Gültigkeitsperioden der Fluents neu berechnet werden.

- *Abduktion*: Sind Effekte von Ereignissen und Belegungen der Fluents zu den jeweiligen Zeiten bekannt, kann durch Abduktion ermittelt werden, welche Folge von Ereignissen zum Zielzustand führt. Voigt beschreibt in [Voi10], dass diese Form vor allem bei der Planung eingesetzt wird, in dem beispielsweise Software-Agenten mit bestimmten Zielwerten ermitteln, welche Aktionsfolgen zu einem definierten Zielzustand führen. Abduktion kann daher auch im Monitoring eingesetzt werden (wie beispielsweise in [MMCT08] durch das *SCIFF* -Rahmenwerk), wenn Zielzustände nach jeder Interaktion bekannt sind. Die berechneten zulässigen Interaktionsfolgen zum Zielzustand müssen dann entsprechend mit der Historie verglichen werden. In diesem Fall wird nach dem Planen jeweils noch ein Vergleich mit der aktuellen Historie getätigt, der bei der Deduktion nicht notwendig ist. Abduktion im Bereich des Monitorings ist daher immer ein zweischrittiges Verfahren, in dem erst geplant und dann mit dem aktuellen Zustand verglichen wird.

- *Induktion*: Sind Ereignisse und die Belegungen von Fluents gegeben, lässt sich der Effekt von Ereignissen auf Fluents ableiten. Shanahan beschreibt in [Sha99], dass die

Induktion daher vor allem im Bereich des Lernens oder der Theoriebildung genutzt werden kann. Im Rahmen der Choreographie-Überwachung kann beispielsweise aus vorhandenen Interaktionen und Effekten auf Fluents ein Prozessmodell ermittelt werden, welches mit dem vorher definierten Prozessmodell verglichen werden kann.

Aufgrund der Beschreibung der Schlussarten ist schnell erkennbar, dass für eine spätere Überwachung von Choreographien das Prinzip der Deduktion zum Tragen kommen kann, wobei eine dynamisch wachsende Faktenmenge zur Laufzeit bei der Deduktion eine wichtige Rolle spielt. Abduktion kann zwar prinzipiell genutzt werden, allerdings ist aufgrund des zweiphasigen Vorgehens mit Planen und Vergleich der Historien davon auszugehen, dass das Laufzeitverhalten schlechter als bei der Deduktion ist. Gleiches gilt für die Induktion, wo der Vergleich zweier Prozessmodelle sich deutlich komplexer gestaltet als der Vergleich von Nachrichtenhistorien wie bei der Abduktion.

Tabelle 3.3.: Sprachelemente des einfachen Ereigniskalküls; aus [Sha99]

Formel	**Bedeutung**
$Initiates(\alpha, \beta, \tau)$	Fluent β gilt nach Eintreten von Ereignis α zur Zeit τ.
$Terminates(\alpha, \beta, \tau)$	Fluent β gilt nicht nach Eintreten von Ereignis α zur Zeit τ.
$Initially_p(\beta)$	Fluent β gilt Initial, also ab dem Zeitpunkt 0.
$\tau_1 < \tau_2$	Zeitpunkt τ_1 liegt vor Zeitpunkt τ_2.
$Happens(\alpha, \tau)$	Ereignis α tritt zum Zeitpunkt τ auf.
$HoldsAt(\beta, \tau)$	Fluent β gilt zum Zeitpunkt τ.
$Clipped(\tau_1, \beta, \tau_2)$	Fluent β wird zwischen den Zeitpunkten τ_1 und τ_2 terminiert.

Shanahan führt in [Sha99] verschiedene Versionen des Ereigniskalküls ein, beginnend mit einer einfachen Version (Simple Event Calculus) und darauf aufbauend Erweiterungen, die zum vollen Ereigniskalkül (Full Event Calculus) führen. Der einfache Ereigniskalkül basiert auf den in Tabelle 3.3 dargestellten Sprachelementen. Das Auftreten eines Ereignisses α zum Zeitpunkt τ wird beispielsweise durch das $Happens(\alpha, \tau)$-Prädikat charakterisiert. Im Gegensatz dazu wird im vollen Ereigniskalkül ein Ereignis auch mit Start- und Endzeitpunkt erfasst und besitzt damit eine gewisse Dauer. Bezogen auf die Laufzeitüberwachung reicht die einfache Sicht auf Ereignisse ohne Dauer aber aus, da ausgetauschte Nachrichten immer zu einem gewissen Zeitpunkt durch einen Sensor beobachtet werden können. Die weiteren Prädikate *Initiates*, *Terminates* und $Initially_p$ beschreiben die Effekte von Ereignissen auf Fluents und werden damit auch als Effekt-Axiome bezeichnet [Sha99].

Mit Unterstützung des *HoldsAt*-Prädikats werden später Anfragen an eine Wissensbasis gestellt, in dem die Gültigkeit eines Fluents zu einem bestimmten Zeitpunkt durch die domänenunabhängigen Ereigniskalkül-Axiome abgeleitet wird. Diese dafür notwendigen

Axiome lauten für den einfachen Ereigniskalkül wie folgt [Sha99]:

$$HoldsAt(f,t) \leftarrow Initially_p(f) \wedge Clipped(0,f,t) \quad (3.6)$$

$$HoldsAt(f,t) \leftarrow Happens(a,t_1) \wedge Initiates(a,f,t_1) \wedge t_1 < t_2 \wedge \neg Clipped(t_1,f,_t 2) \quad (3.7)$$

$$Clipped(t_1,f,t_2) \leftarrow Happens(a,t) \wedge t_1 < t < t_2 \wedge Terminates(a,f,t) \quad (3.8)$$

Das Axiom 3.6 beschreibt, dass ein Fluent f genau dann zum Zeitpunkt t gilt, wenn es Initial gültig ist und in der Zwischenzeit nicht terminiert wurde. Das entsprechende *Clipped*-Prädikat in Axiom 3.8 beschreibt einen Zeitraum, währenddessen ein Fluent f durch ein Ereignis a terminiert wird. Durch t_1 und t_2 werden dabei Start- und Endzeitpunkt des Zeitraums festgelegt. Das Axiom 3.7 definiert, dass ein Fluent gültig ist, wenn es zu einem Zeitpunkt in der Vergangenheit durch ein Ereignis initiiert und zwischenzeitlich nicht durch ein weiteres Ereignis terminiert wurde. Shanahan beschreibt, dass die Gültigkeitsperioden von Fluents durch den Einsatz der Axiome erst direkt nach Eintreten eines Ereignisses beginnen. Umgekehrt reicht die Gültigkeitsperiode eines Fluents bis einschließlich zum Auftreten des Ereignisses, welches das Fluent terminiert. Ist kein solches Ereignis in der Wissensbasis vorhanden, dann gilt das entsprechende Fluent unbegrenzt.

Durch die explizite Beschreibung von Ereigniseffekten auf Fluents gibt es allerdings ein Problem, welches in [Sha97] und [Sha99] anhand eines Beispiels zum Laden und Abschießen einer Pistole diskutiert wird. Vereinfacht geht es um die Beschreibung der Auswirkungen von Ereignissen auf Fluents, ohne sämtliche Nicht-Auswirkungen zu beschreiben. Im beschriebenen Beispiel kann durch ein bestimmtes Ereignis nicht ausgeschlossen werden, dass eine Waffe nicht wieder entladen wurde. Dieses *Rahmenproblem* muss beim Ereigniskalkül entsprechend berücksichtigt werden und wird von Shanahan durch die Grundannahmen gelöst, dass es keine unerwarteten Ereignisse gibt und dass es durch Ereignisse keine unerwarteten Effekte gibt. Diese Grundannahmen müssen bei der Überwachung später entsprechend berücksichtigt werden. Beispielsweise kann der durch Regeln beschriebene Zustand vom realen Zustand abweichen, wenn nicht alle Ereignisse durch entsprechende domänenabhängige Effekt-Axiome abgebildet werden.

Da der Ereigniskalkül auf Basis von Prädikatenlogik funktioniert, unterstützt er von Haus aus alle Reichweiten von Integritätsbedingungen. Seien es nachrichtenbasierte Integritätsbedingungen, wo bestimmte Restriktionen der verwendeten Datentypen geprüft werden müssen, oder instanzbasierte Bedingungen, die den Kontrollfluss modellieren. Für jede Überprüfung müssen entsprechende Prädikate und Effekt-Axiome eingeführt werden. Darüber hinaus sind auch problemlos instanzübergreifende Integritätsbedingungen möglich, wenn domänenabhängige Prädikate entsprechende Instanzparameter enthalten.

Caching im Ereigniskalkül: Der Einsatz des klassischen Ereigniskalküls zeigt allerdings ein recht schwaches Laufzeitverhalten, da bei jeder Anfrage mit $HoldsAt$ an die Wissensbasis die Schlussfolgerungen und damit die Fluent-Belegungen erneut und bei verschiedenen

Fluents auch innerhalb einer Anfrage mehrfach berechnet werden. Um dieses zu umgehen, schlagen Chittaro und Montanari in [CM94] den *Cached-Event-Calculus* (CEC) vor, welche die Berechnung der Fluent-Belegungen vom Anfragezeitpunkt auf den Einfügezeitpunkt der Fakten verschieben. Dafür berechnet der CEC beim Einfügen eines neuen Ereignisfaktes die jeweiligen maximalen Gültigkeitsperioden (maximum validity intervals, MVIs) von Fluents neu und fügt die durch das Ereigniskalkül abgeleiteten Gültigkeitsperioden als neue Fakten zur Wissensbasis hinzu. Wie genau die Berechnungen des CEC funktionieren, ist in [CM96] ausführlich dargestellt.

Bei der Laufzeitüberwachung trägt ein weiterer Fakt zur Optimierung bei, da in einer Nachrichtenhistorie die Ereignisse schon chronologisch sortiert sind. Der Fall, dass neue Ereignisse zwischen bereits bekannten Ereignissen auftreten, kann damit ausgeschlossen werden. Chittaro und Montanari argumentieren daher, dass im chronologischen Fall die Revision älterer Gültigkeitsperioden komplett wegfällt [CM94]. Weiterhin haben Chittaro und Montanari die unterschiedlichen Aufwände der Einfüge- und Schlussfolgerungsoperationen wie in Tabelle 3.4 dargestellt abgeschätzt. Die dargestellten Komplexitäten stellen die Anzahl der Zugriffe auf Fakten in der Wissensbasis dar, wobei n die Anzahl der Ereignisse bezeichnet, die einen Einfluss auf ein Fluent haben. Der Faktor L_{bk} beschreibt die Schachtelungstiefe von Vorbedingungen und damit, in welchem Umfang eine Änderung eines Fluents vom Kontext abhängig ist. Gibt es keine Kontextabhängigkeiten ($L_{bk} = 0$), so sind CEC und EC im Laufzeitverhalten gleich. Aber sobald sich die Abhängigkeiten vom Kontext erhöhen, verhält sich der CEC deutlich besser beim Schlussfolgern als der klassische EC. Zusätzlich sind Anfragen an die Wissensbasis mit dem CEC in linearer Zeit und damit abhängig von der Größe der Wissensbasis zu lösen.

Tabelle 3.4.: Komplexitätsanalyse zwischen EC und CEC; aus [CM96]

	EC-Update	**EC-Anfrage**	**CEC-Update**	**CEC-Anfrage**
$L_{bk} = 0$	konstant	$O(n^3)$	$O(n^3)$	$O(n)$
$L_{bk} = 1$	konstant	$O(n^6)$	$O(n^4)$	$O(n)$
$L_{bk} = 2$	konstant	$O(n^9)$	$O(n^5)$	$O(n)$
$L_{bk} = k(> 0)$	konstant	$O(n^{(k+1)*3})$	$O(n^{k+3})$	$O(n)$

Der CEC löst damit die Probleme des Mehrfachberechnens von Gültigkeitsperioden in der gleichen Anfrage und beschleunigt sogar das Schlussfolgern bei höherer Kontextabhängigkeit. Dies ist vor allem bei der Laufzeitüberwachung von Choreographien wichtig, da viele Fluents vom Kontext abhängig sind. Beispielsweise kann ein Fluent, welches den fehlerfreien Start einer Sequenz beschreibt, nur dann gültig sein, wenn die Vorbedingungen erfüllt sind. Dies kann beispielsweise eine vorige Sequenz sein, welche erfolgreich beendet

sein muss, bevor die betrachtete Sequenz beginnen kann.

3.1.2.4. Weitere Ansätze zur formalen Beschreibung

Für die formale Repräsentation von Choreographien gibt es noch weitere Ansätze, wie beispielsweise Petri-Netze, welche auch im Bereich des Business-Process-Managements oft zur Modellierung eingesetzt werden. Allerdings wird von Decker und Puhlmann in [DPW06] argumentiert, dass Petri-Netze hinsichtlich der Formalisierung von Interaktionen und damit dem Protokoll einer Choreographie diverse Probleme mit sich bringen, um alle in [BDH05] definierten Interaktionsmuster abzubilden. Genauso argumentieren sie, dass beispielsweise dynamisch wachsende oder schrumpfende Teilnehmermengen in Petri-Netzen schwer zu modellieren sind, da alle potenziellen Interaktionspartner zur Modellierungszeit bekannt sein müssen und jede mögliche Verbindung modelliert werden muss. Genauso wird in [AHKB03] argumentiert, dass es mit Petri-Netzen nicht direkt möglich ist, alle in [AHKB03] definierten Workflow-Muster zu modellieren. Aalst und Hofstede haben daher beispielsweise in [AH05] die *Yet-another-Workflow-Language* (YAWL) entwickelt, um die in dem Papier angesprochenen Probleme bei Petri-Netzen zu überwinden. Aus den angeführten Gründen sind Petri-Netze daher auch bei der Laufzeitüberwachung nicht in die engere Wahl gekommen, da nicht alle modellinhärenten Integritätsbedingungen eines WS-CDL-Dokuments einfach abzubilden sind.

Weitere Ansätze zur formalen Spezifikation sind auch Automaten, wie sie beispielsweise auch im Ansatz von Simmons, Gan, Chechik, Nejati, O'Farrel zur Überwachung von Web-Services-Konversationen benutzt werden [GCN+07, SCN+08, SGC+09]. Problematisch an Automaten sind allerdings genau wie bei dem π-Kalkül, dass instanzübergreifende Bedingungen nicht direkt möglich sind. Genauso dienen Automaten wie beim π-Kalkül eher zur Definition von Systemübergängen und weniger zur Beschreibung von quantifizierten Zeitabläufen. Damit lässt sich also sehr schnell und einfach der Kontrollfluss einer Choreographie prüfen, weniger allerdings instanzübergreifende Integritätsbedingungen oder auch non-funktionale Bedingungen wie Zeitschranken. Daher werden automatenbasierte Techniken in dieser Arbeit zur Überwachung von Choreographien nicht weiter zur Realisierung herangezogen.

3.1.2.5. Implikationen für diese Arbeit

Durch die Analyse in den letzten Abschnitten ergibt sich als Ergebnis die Tabelle 3.5. Für das weitere Vorgehen in dieser Arbeit wird daher das Ereigniskalkül zur Formalisierung und mit entsprechendem Caching auch zur Laufzeitüberprüfung genutzt. Mit dem Ereigniskalkül lassen sich alle Integritätsbedingungen inklusive des Kontrollflusses modellieren sowie zur Laufzeit entsprechend mit Prolog ohne weitere Erweiterungen direkt einsetzen.

Die anderen Ansätze lassen sich zwar prinzipiell auch einsetzen, haben aber meist nur gewisse Vorteile in bestimmten Bereichen wie der Überprüfung des Kontrollflusses und

Tabelle 3.5.: Formale Modelle im Vergleich

	Modellinhärente Eigenschaften	**Inhaltliche Anforderungen**	**Reichweite: Nachricht**	**Reichweite: Interaktion**	**Reichweite: lokal**	**Reichweite: übergreifend**	**Quantifizierte Zeiten**	**Benutzbarkeit**
π-Kalkül	✓	(✓)	(✓)	(✓)	✓	(✓)	(✓)	(✓)
Temporallogiken	✓	(✓)	(✓)	(✓)	✓	(✓)	(✓)	-
Ereigniskalkül	✓	✓	✓	✓	✓	✓	✓	✓
Petrinetze	(✓)	(✓)	(✓)	(✓)	✓	(✓)	(✓)	(✓)
Automaten	✓	(✓)	(✓)	✓	✓	-	(✓)	(✓)

müssen an einigen Stellen erweitert werden, um bestimmte Arten von Integritätsbedingungen zu modellieren. Beispielsweise eignet sich das π-Kalkül sehr gut zur Formalisierung von Interaktionen, allerdings gibt es einige Nachteile bezüglich der Laufzeitüberprüfung bei nachrichtenbasierten Integritätsbedingungen, inhaltlichen Anforderungen oder auch instanzübergreifenden Integritätsbedingungen. Eine Erweiterung wäre zwar möglich, aber der Fokus des π-Kalküls liegt eindeutig in der Modellierung des Kontrollflusses. Weitere kleinere Probleme bei der Modellierung von Zeitschranken schränken den Einsatz des Kalküls weiter ein, weshalb es zur Laufzeitüberprüfung nicht weiter betrachtet wird.

Ähnlich verhält es sich mit Temporallogiken, welche auch entsprechend um prädikatenlogische Ausdrücke erweitert werden müssen, um etwa nachrichtenbasierte Integritätsbedingungen, inhaltlichen Anforderungen oder auch instanzübergreifenden Integritätsbedingungen zu verifizieren. Dadurch müssen Tools erweitert werden, was die direkte Benutzbarkeit des Kalküls entsprechend einschränkt. Weiterhin argumentieren beispielsweise Montali, Mello, Chesani und Torroni in [MMCT08], das Temporallogiken keine quantifizierten Zeiten unterstützen. Erweiterungen sind zwar möglich, diese sind aber aufgrund ihrer Berechnungskomplexität und dem Problem der Zustandsexplosion nicht für die Laufzeitüberwachung geeignet [Wan04].

Weitere Ansätze wie Petri-Netze haben Schwierigkeiten, alle modellinhärenten Eigenschaften einer Choreographie-Beschreibung in WS-CDL zu beschreiben. Wenn nicht alle Interaktionsmuster oder Prozessmuster ohne Probleme abbildbar sind, dann sind Petri-Netze nicht in der weiteren Auswahl der Methoden zu berücksichtigen. Genauso verhält es sich auch mit automatenbasierten Ansätzen, die zwar schnell sind, aber wie auch beim π-Kalkül eher den Fokus auf Systemübergänge haben und weniger die Beschreibung von zeitlichen Abläufen mit quantifizierten Zeiten im Blick haben. Genauso sind mit Automaten instanzübergreifende Integritätsbedingungen schwierig umzusetzen, weshalb auch Automaten in dieser Arbeit nicht weiter betrachtet werden.

3.1.3. Technische Anforderungen an das Monitoring

Im letzten Abschnitt sind die Anforderungen an die formale Beschreibung der Integritätsbedingungen sowie der Techniken zur Laufzeitverifikation vorgestellt und analysiert worden. In diesem Abschnitt werden jetzt die Anforderungen an eine Infrastruktur vorgestellt, mit deren Hilfe benötigte Daten für die Laufzeitverifikation bereitgestellt werden. Eine Choreographie spezifiziert vor allem das extern sichtbare Verhalten seiner Teilnehmer, welches sich zwischen den teilnehmenden Rollen durch den Nachrichtenaustausch ergibt. Dieser Nachrichtenaustausch muss durch einen Monitor für die spätere Verifikation protokolliert werden. Die Hauptaufgabe eines Monitors ist also die Überwachung des Nachrichtenaustausches zwischen den Kollaborationspartnern.

Im Folgenden werden dafür kurz bekannte Monitoring-Techniken vorgestellt und im Hinblick auf das Einsatzfeld der Choreographie-Überwachung diskutiert. Monitoring-Ansätze für verteilte Systeme analysiert beispielsweise Kurschl in [Kur00]. Er unterscheidet dabei zwischen *internen Ereignissen*, welche normalerweise nicht nach außen sichtbar sind, aber trotzdem häufig über Debugging- oder Management-Schnittstellen sichtbar gemacht werden und *externen Ereignissen*, die beobachtbar sind, da sie das Verhalten von Prozessen und Interaktionen darstellen. Kurschl argumentiert, dass externe Ereignisse das Prinzip der Datenkapselung von Objekten und Prozessen nicht verletzt. Choreographien beschreiben generell das beobachtbare Verhalten von Web-Services, daher sind diese externen Ereignisse auch das Ziel des Monitorings. Weiterhin unterscheidet Kurschl in [Kur00] auch zwischen *einfachen Ereignissen* und *zusammengesetzten Ereignissen*. Bezüglich einer Choreographie ist ein einfaches Ereignis das Übertragen einer Nachricht von einem Teilnehmer zum anderen, während ein zusammengesetztes Ereignis wie beispielsweise der Abschluss einer Sequenz aus mehreren Ereignissen entsteht. Der Fokus der Überwachung liegt daher auf einfachen Ereignissen und damit auf der Überwachung von übertragenen Web-Services-Nachrichten. Komplexe Ereignisse entstehen erst durch die Laufzeitverifikation, in dem beispielsweise eine abgeschlossene Sequenz aus mehreren Interaktionen als korrekt markiert wird.

Sensoren für Ereignisse können, wie in [Ger01] und [Kur00] beschrieben, auf verschiedene Arten implementiert werden: Entweder als *Software*-Systeme, als *Hardware*-Systeme oder auch als *hybride* Systeme. Beim Hardware-Monitoring wird spezielle Hardware zur Überwachung eingesetzt, welche passiv Informationen sammelt und die Beobachtung von hardwarenahen internen Ereignissen mit minimaler Beeinflussung des beobachteten Systems ermöglicht. Kurschl bezeichnet Hardware-Monitore daher auch als nicht-intrusiv. Da in Choreographien allerdings nur externe Ereignisse relevant sind, können Hardware-Monitore zur Beobachtung von internen Ereignissen in dieser Arbeit nicht eingesetzt werden. Bei Software-Monitoren werden umgekehrt entsprechende Sensoren als Software in das Zielsystem eingebettet, um bestimmte Ereignisse wie das Senden und Empfangen von Nachrichten zu beobachten. Wichtig an dieser Stelle ist die Autonomie von Teilnehmern in Choreographien, die zumeist kein Interesse an der Herausgabe von internen Ereignissen und an der

Veränderung ihrer internen Software-Systeme haben. Um externe Ereignisse zu beobachten, kann in Choreographien ein Sensor also nicht direkt in interne Systeme wie beispielsweise in [DS06] mithilfe von AspectJ eingebettet werden, sondern muss entweder zwischen den Teilnehmern oder an der Schnittstelle der Teilnehmer instrumentiert werden. Des weiteren gibt es auch hybride Monitore, welche eine Kombination aus Hard- und Software nutzen. Dies kann beispielsweise durch softwarebasierte Sensoren und entsprechende Hardware realisiert werden, welche Ereignisse weiterleiten können. Software- und hybride Monitore haben daher immer einen Einfluss auf zu überwachende Systeme und können das Laufzeitverhalten eines Systems im schlimmsten Fall verändern.

Sollen Software-Monitore zum Einsatz kommen, die entweder zwischen den Teilnehmern oder an der Schnittstelle instrumentiert werden, können verschiedene Verfahren angewendet werden. Zum einen sind dies *proxy-basierte* Verfahren, wie sie beispielsweise zur Überwachung von Orchestrierungen von Baresi, Ghezzi und Guinea in [BGG04] genutzt werden. Das Proxy-Pattern oder auch Stellvertreter-Muster ist ein Entwurfsmuster aus der Software-Entwicklung (siehe [GHJV96]), welches die Kontrolle auf den Zugriff auf bestimmte Objekte durch vorgeschobene Objekte realisiert. Durch einen Proxy lässt sich das Senden und Empfangen von Nachrichten bestimmter Schnittstellen einfach beobachten, da der Proxy direkt zwischen den beiden Interaktionspartnern sitzt. Zum anderen können bei der Beobachtung von Ereignissen auch *Sniffing*-Methoden zur Anwendung kommen, wenn Nachrichten über öffentliche Netze transportiert werden und der Nachrichtenkanal an verschiedenen Stellen abgehört werden kann. Letzteres ist in Choreographie-Umgebungen schwer realisierbar, da Organisationen ein Interesse daran haben, dass Nachrichten über sichere Kanäle transportiert werden, die entweder durch kryptografische Methoden gesichert sind oder dedizierte Punkt-zu-Punkt-Verbindungen implementieren. Proxy-basierte Verfahren sind an dieser Stelle deutlich einfacher durch die Organisationen zu kontrollieren und zu implementieren. Zusätzlich bleibt die Autonomie der Organisationen gewahrt, welche ihre eigenen internen Systeme nicht direkt anpassen müssen, sondern durch entsprechende Stellvertreterdienste eine Monitoring-Schnittstelle dem eigentlichen Dienst vorschalten, um das externe Verhalten beobachten zu lassen.

Allgemein lassen sich die Arten der Beobachtung von Ereignissen in *ereignisorientiert* und *zeitgesteuert* unterteilen, bei denen die Sensoren entsprechend Daten entweder zeitgesteuert oder ereignisorientiert liefern. Beide Arten lassen sich zur Überwachung von Choreographien nutzen, wenn sichergestellt werden kann, dass die Sensoren bei der zeitgesteuerten Abfrage entsprechend alle zwischenzeitlich detektierten Ereignisse zur Laufzeitverifikation liefern können. Prinzipiell kommt es auf das Anwendungsumfeld an, welche der Methode eingesetzt wird. Gibt es in einem System ständig viele Ereignisse, dann kann eine zeitgesteuerte Abfrage von Sensoren Kommunikationsaufwand sparen, wenn in einer Nachricht mehrere Ereignisse zusammengefasst übertragen werden. Werden aber kaum Ereignisse beobachtet, kann sich eine zeitgesteuerte Abfrage negativ auswirken, da sehr viele Anfragen an einen Sensor so gut wie keine Resultate liefern. Umgekehrt wird bei einer ereignisorien-

tierten Steuerung nur bei einem auftretenden Ereignis eine Nachricht an relevanten Stellen versendet. Zusätzlich lassen sich beide Methoden auch in der Komplexität der Implementierung unterscheiden. Bei einer zeitgesteuerten Beobachtung müssen entsprechende Schnittstellen bei den Teilnehmern implementiert werden, welche ein aktives Ansprechen von Sensoren ermöglichen. Die Sensoren selbst müssen zusätzlich in der Lage sein, bestimmte Ereignisse zu speichern und erst bei Bedarf herauszugeben. Eine Technik aus dem Web-Services-Umfeld zur Unterstützung solcher aktiver Sensoren ist beispielsweise die Spezifikation *Web Services Distributed Management* (WSDM) der OASIS [KBV06], bei der Web-Services über standardisierte Schnittstellen zur Herausgabe von Ereignissen angesprochen werden können. In Choreographie-Umgebungen kann der Einsatz dieser Techniken aber nicht als gegeben angenommen werden, da jeder Teilnehmer selbst für die Implementierung seiner Schnittstellen verantwortlich ist und zusätzlich typische Choreographie-Beschreibungen die jeweiligen Monitoring-Mechanismen nicht betrachten. Damit Choreographien aber generell unterstützt werden können, kann also nur ein ereignisorientierter Mechanismus genutzt werden. Auch bei ereignisorientierter Beobachtung gibt es unterschiedliche Mechanismen, wie beispielsweise die Spezifikationen der WS-Notification-Familie [GHM06, VGN06, CL06], welche Publish/Subscribe-Mechanismen für Web-Services anbieten. Werden diese Standards benutzt, muss auch hier jeder Choreographie-Teilnehmer selbst *aktiv* Sensordaten an registrierten Nutzer verteilen. Im Gegensatz dazu können auch *passive* Methoden zum Einsatz kommen, bei denen ein Teilnehmer nicht selbst Sensordaten verwalten und an Monitoring-Systeme versenden muss. Die oben angesprochenen Proxies sind ein Beispiel für den Einsatz solcher Systeme.

Weiterhin gibt es nicht nur Anforderungen an den Ort der Instrumentierung und Betriebsart der Sensoren, zusätzlich muss durch die Sensoren auch die *Konsistenz der Sensordaten* im verteilten System gewährleistet sein. Beim Beobachten von Interaktionen innerhalb von Choreographien ist die Bestimmung von Auftrittszeitpunkten der Ereignisse in einem verteilten System zu beachten. Besitzen Sensoren unterschiedliche interne Uhrzeiten, kann beim Zusammenführen von Sensordaten nicht mehr festgestellt werden, ob beispielsweise Zeitschranken eingehalten werden oder nicht. Abhilfe schaffen hier synchronisierte Uhren über das NTP-Protokoll [MDM+10] oder auch Techniken wie beispielsweise Vektor-Uhren, wie sie von Lamport in [Lam78] vorgestellt werden. Sollen Daten mehrerer Sensoren ausgewertet werden, müssen entsprechende Techniken zur Ordnung der Ereignisse oder zur Synchronisierung von Uhren genutzt werden. Zusätzlich impliziert der Einsatz von proxybasierten Sensoren die Sicherstellung, dass alle Nachrichten auch wirklich über die Stellvertreterdienste gesendet werden, da ansonsten bestimmte Ereignisse nicht detektiert werden können und eine Laufzeitverifikation auf inkonsistenten Daten arbeiten würde. Es muss also sichergestellt werden, dass alle möglichen Interaktionsmuster wie In-Only, In-Out, Out-Only oder Out-In aus [GHM+07] durch einen proxy-basierten Sensor unterstützt werden. Genauso muss durch die Stellvertreterdienste sichergestellt werden, dass zum einen die normale Nachrichtenübertragung an die Zieldienste zeitlich nicht deutlich länger als ohne Stell-

vertreterdienst dauert und zum anderen eingehende Nachrichten auch wirklich beim Empfänger ankommen. Ansonsten enthält ein entsprechendes Sensor-Protokoll eine Nachricht, die nicht vollständig übertragen worden ist. Umgekehrt darf natürlich auch keine Nachricht an einen Dienst weitergeleitet werden, welche nicht im Protokoll vorhanden ist. Die Protokollierung und Weiterleitung von Ereignissen muss im Sinne einer Transaktion also atomar erfolgen.

Eine weitere wichtige Anforderung an die Sensoren ist die Korrelation der Ereignisse zum formalen Modell. Es muss also sichergestellt werden, dass die Projektion aller für die Choreographie wichtigen Ereignisse eines Nachrichtenstroms auf das formale Modell vollständig ist und es damit keine Choreographie-Nachrichten gibt, die nicht auf das formale Modell abgebildet werden können. Dabei müssen entsprechende Nachrichten mindestens einer Choreographie-Instanz sowie einzelnen Teilnehmern oder Teilnehmerrollen (Sender und Empfänger) zugeordnet werden können. Ist dies nicht möglich, kann beispielsweise der Kontrollfluss einer Choreographie nicht verifiziert werden, da laut Modell immer noch auf eine bestimmte Nachricht gewartet wird, die aber schon längst erfolgt ist. Die Laufzeitverifikation muss also auf vollständigen Daten arbeiten.

3.1.3.1. Implikationen für die Arbeit

Zur Laufzeitverifikation von Choreographien müssen zusammenfassend also externe Ereignisse durch Sensoren beobachtet werden, die passiv ohne größeren Implementierungsaufwand seitens der Teilnehmer arbeiten und nicht in interne Systeme der Teilnehmer instrumentierbar sind. Dafür bieten sich Verfahren der passiven ereignisorientierten Beobachtung an, welche beispielsweise als Stellvertreterdienste zwischen den Teilnehmern platziert werden. Ziel der Instrumentierung von Sensoren ist daher die Nachrichteninfrastruktur zwischen den Teilnehmern.

Weiterhin muss die zeitliche Ordnung der verteilt detektierten Ereignisse konsistent sein und nicht durch unterschiedliche lokale Uhren verfälscht werden. Zusätzlich müssen alle für eine Choreographie relevanten Ereignisse durch die Sensoren erkannt werden und entsprechende Stellvertreterdienste zusätzlich sicherstellen, dass der normale Nachrichtenfluss zu den eigentlichen Teilnehmern nicht verlangsamt oder unterbrochen wird. Die Konsistenz der protokollierten Ereignisse eines Sensors muss zusätzlich gewahrt bleiben, was bedeutet, dass er nur Ereignisse protokollieren darf, die auch wirklich später beim Empfänger ankommen. Das Weiterleiten an den Empfänger und der Protokollierungsvorgang müssen dementsprechend atomar sein.

Als wichtigste Voraussetzung für die Laufzeitverifikation muss allerdings ein Ereignis zum formalen Modell korrelierbar sein, ansonsten kann die Laufzeitverifikation nicht korrekt arbeiten. Damit muss zum einen jede innerhalb einer Choreographie versendete Nachricht eine formale Entsprechung haben und eindeutig auf diese abbildbar sein.

Insgesamt ergeben sich die folgenden Anforderungen an die Implementierung einer In-

frastruktur zur Interaktionsprotokollierung, welche sich in drei Dimensionen einordnen lassen:

- Die **Geschäftsdimension**: Durch **Performanz** und **Skalierbarkeit** der Sensoren muss eine möglichst geringe Beeinflussung des Betriebes gewährleistet werden.
- Die **Modelldimension**: Die **Korrelation** zwischen formalem Modell und beobachteten Nachrichten muss möglich sein.
- Die **Architekturdimension**:
  - Einsatz von möglichst **nicht-instrusiven Sensoren** durch
    - **Implementierung von Stellvertreterdiensten** zwischen Teilnehmern, um interne Systeme möglichst wenig zu verändern,
    - **Einsatz von ereignisorientierten Sensoren**, damit Teilnehmer möglichst wenig Infrastruktur vorhalten und nicht selbst aktiv eigene externe Ereignisse kommunizieren müssen.
  - Sicherstellung der **Konsistenz der Sensordaten** durch
    - **Unterstützung aller Service-Interaktionsmuster** von Sensoren,
    - **Atomare Interaktionsprotokollierung**, welche sicherstellt, dass bei erfolgter Protokollierung eine Nachricht auch gesendet und angekommen sein muss,
    - **Zeitliche Einordnung der Ereignisreihenfolge** von verschiedenen Sensoren.

Um diese Anforderungen an die Implementierung umzusetzen, werden in Kapitel 4.1 entsprechende Lösungen erarbeitet.

3.2. Transaktionale Kontrolle von Prozess-Choreographien

In den vorangegangen Abschnitten sind entsprechende formale Mechanismen ausgewählt worden, um Laufzeitfehler und damit Abweichungen von definierten Integritätsbedingungen zu erkennen. In diesem Abschnitt geht es um die Anforderungen an die Behebung dieser Fehler und vor allem auch, welche Arten von Fehlern zur Laufzeit beispielsweise durch transaktionale Koordination automatisiert behoben werden können und welche Anforderungen dadurch entstehen.

3.2.1. Fehlermodell

Um transaktionale Fehlerbehebung zu unterstützen, muss dafür als Erstes definiert werden, welche Arten von Fehlern automatisiert behoben werden können. Chan, Bishop, Steyn, Baresi und Guinea haben beispielsweise in [CBS+07] eine Fehler-Taxonomie für Web-Services-

Kompositionen vorgestellt, die Gründe für Abweichungen vom erwarteten Verhalten klassifizieren. Sie klassifizieren dabei die auftretenden Fehler grob in die drei Kategorien *physische Fehler*, *Entwicklungsfehler* und *Interaktionsfehler*. Zwar kann eine Laufzeitverifikation alle Klassen dieser Fehler erkennen, allerdings lässt sich nur ein kleiner Teil davon automatisch zur Laufzeit beheben. Beispielsweise sind Fehler aus der Kategorie Entwicklungsfehler starke systematische Fehler, welche sich nicht automatisiert korrigieren lassen. Physische Fehler wie Netzwerkpartitionen und Systemausfälle sollten allerdings nicht dazu führen, dass auf teilnehmenden Systemen inkonsistente Systemzustände zurückgelassen werden, weshalb diese Fehler nach Möglichkeit koordiniert zu beheben sind. Fehler aus der Kategorie der Interaktionsfehler beziehen sich auf die bereits diskutierten Integritätsbedingungen. Sind diese verletzt, müssen Maßnahmen zur Behebung angestoßen werden, um diese Fehler beziehungsweise die betreffenden Aktivitäten zurückzusetzen.

Im Bereich der Datenbanken gibt es ein vergleichbares Fehlermodell, welches Härder und Rahm in [HR01] vorstellen. Die Autoren beschreiben *Transaktionsfehler*, *Systemfehler*, *Gerätefehler* sowie *Katastrophen*, für die ein Datenbanksystem entsprechende Vorkehrungen wie Logging und Recovery treffen muss, um auf diese Arten von Fehlern reagieren zu können. Diese Kategorisierung lässt sich auch auf eine verteilt ausgeführte Choreographie anwenden. In der Kategorie der Transaktionsfehler lassen sich Fehler einer einzelnen Choreographie-Instanz einordnen, welche durch Interaktionsfehler und einem Abbruch der Choreographie-Instanz entstehen können. Weiterhin gibt es Transaktionsfehler, die aus der Verletzung instanzübergreifender Integritätsbedingungen resultieren. In allen Fällen muss die dazugehörige globale Transaktion koordiniert zurückgesetzt werden. Die Systemfehler, Gerätefehler oder Katastrophen entsprechen den physischen Fehlern im vorangegangenen Abschnitt, wobei diese sich bezüglich einer Choreographie in *beobachtbare* und *nicht öffentlich sichtbare Fehler* einteilen lassen.

Um Gerätefehler in Datenbanken beheben zu können, werden hauptsächlich redundante Daten in entsprechenden Log-Dateien gesammelt, bevor Änderungen in eine permanente Datenbank eingebracht werden. Für ein einzelnes Datenbanksystem kann Logging vorausgesetzt werden, aber für autonome Choreographie-Teilnehmer muss dies nicht unbedingt gelten. Ein Transaktionsprotokoll kann daher nur die entsprechenden Nachrichten an Teilnehmer senden, um beispielsweise die Kompensation einer Aktivität zu veranlassen. Ob ein Teilnehmer dieses auch intern implementiert, kann nicht garantiert oder durch das beobachtbare Verhalten überprüft werden. Daher kann zusammenfassend gesagt werden, dass eine spätere Fehlerbehebung sich nur auf beobachtbare Interaktionsfehler sowie auf beobachtbare physische Fehler wie ausgefallene Dienste oder Netzwerkpartitionen beziehen kann. Beobachtbare physische Fehler sind in diesem Falle nur als Fehler erkennbar, wenn sie durch entsprechend formulierte Integritätsbedingungen wie Zeitschranken überprüfbar sind. Gibt es für bestimmte Interaktionen keine Zeitschranken, kann bei einer Interaktion auch endlos auf eine Antwort auf eine Anfrage gewartet werden. Ein ausgefallener Dienst oder eine Netzwerkpartition, bei der sich verschiedene Dienste untereinander nicht mehr er-

reichen können, sind also nicht zwangsläufig Fehler. Sind organisationsinterne Systeme wie Speichersysteme betroffen, sollten diese zwar mit entsprechenden Logging- und Recovery-Mechanismen versehen sein, aber garantiert werden kann dies für autonome Teilnehmer an einer Choreographie nicht. Entwicklungsfehler können zwar durch die Überwachung aufgedeckt werden, aber nicht zur Laufzeit behoben werden. Die Fehler dieser Kategorie können im schlimmsten Fall zu einem dauerhaften Zurücksetzen von bestimmten Fehlern führen, weshalb in dieser Arbeit die Grundannahme getroffen wird, dass die Software weitestgehend fehlerfrei läuft, da sonst ein Normalbetrieb nicht gewährleistet werden kann.

Weiterhin muss definiert werden, wie solche Arten von Fehlern bei der Behebung unterstützt werden können. Dabei können, wie in [HR01] beschrieben, zwei verschiedene Strategien genutzt werden: Forward- und Backward-Recovery. Bei der Forward-Recovery-Strategie soll jedoch durch verschiedene Maßnahmen und ohne das System zurückzusetzen ein neuer Zustand erreicht werden, ab dem ein System weiterlaufen kann. Umgekehrt werden bei Backward-Recovery verschiedene Aktivitäten zurückgesetzt, um dann beispielsweise ab einem bestimmten Sicherungspunkt erneut zu starten. Forward-Recovery-Strategien für integritätsverletzende Interaktionsfehler sind dabei einfach zu kategorisieren, wie es beispielsweise auch Sam Guinea in [Gui05] macht:

- **Erneutes Ausführen**: Wie im realen Leben kann im Fehlerfall eine Aktivität unter bestimmten Umständen einfach erneut ausgeführt werden, um dem Teilnehmer eine erneute Chance zur Erbringung seiner Dienstleistung zu geben. Die Möglichkeit eines erneuten Ausführens einer Interaktion ist allerdings stark anwendungsabhängig, weshalb solche Maßnahmen im Choreographie-Modell spezifiziert sein müssen. Als Beispiel kann dafür ein Auszahlen von Währung am Geldautomaten dienen, was ohne weitere Prüfungsaktivitäten nicht erneut ausgeführt werden sollte. Daher kann ein generischer Transaktionskoordinator ohne Anwendungswissen nicht entscheiden, ob eine Aktivität erneut ausgeführt werden kann.

- **Ersetzen eines Teilnehmern**: Für das Ersetzen von fehlerhaften Teilnehmern gibt es beispielsweise das Modell der *Flex*-Transaktionen [ELLR90], bei denen Abhängigkeiten zwischen verschiedenen transaktionalen Aktivitäten definiert werden, damit ein Transaktionskoordinator diese Art der Recovery durchführen kann. Dabei gibt es positive Abhängigkeiten, die eine sequenzielle Ausführung von Aktivitäten bedingen, aber auch negative Abhängigkeiten, welche die Kontravalenz der Teilnehmer beschreiben[1]. Dabei wird entweder der eine Teilnehmer ausgeführt oder der andere. Auch hier kann diskutiert werden, ob diese Fälle nicht eher in einem Choreographie-Modell spezifiziert werden müssen, da zum Starten einer Alternativtransaktion ein generischer Transaktionskoordinator in der Lage sein muss, diese Aktivitäten auch entsprechend zu starten. Genauso kann es auch mehrere Alternativen geben, die vom Anwendungs-

[1] Zu weiteren Beschreibungen von Abhängigkeiten transaktionaler Aktivitäten siehe auch [Elm92].

fall und dem jeweiligen Kontext abhängig sind. Ein generischer Transaktionskoordinator kann hier ohne weiteres Anwendungswissen kaum entscheiden, welche dieser Alternativen zu einem bestimmten Zeitpunkt ausgeführt werden soll.

- **Restrukturierung einer Choreographie**: Um Forward-Recovery durch Restrukturierung zu erreichen, müssen für fehlerhafte Teilnehmer entsprechende alternative Teilprozessabläufe gefunden werden, um das gleiche Ziel zu erreichen. Dies kann beispielsweise mithilfe entsprechender Prozess-Templates wie in [Mag07] realisiert werden. Ein Transaktionskoordinator ist auch hier ohne Anwendungswissen und Zielvorgaben nur schwer in der Lage, eine Restrukturierung automatisiert durchzuführen.

Anhand dieser einfachen Kategorisierung von Forward-Recovery-Maßnahmen ist ersichtlich, dass eine automatisierte Forward-Recovery für Fehler schwer durch generische Mechanismen unterstützbar ist. Backward-Recovery mit entsprechendem Zurücksetzen von Effekten kann hier deutlich einfacher automatisiert durch Transaktionsprotokolle unterstützt werden. Im weiteren Verlauf der Arbeit werden daher nur noch Backward-Recovery-Maßnahmen zur Behebung von Interaktionsfehlern und beobachtbaren physischen Fehlern betrachtet. Wichtig ist hierbei allerdings, dass im Gegensatz zu Datenbanktransaktionen in Choreographien auch reale Aktivitäten durchgeführt werden[2], welche nicht unbedingt reversibel sind. In diesem Fall kann eine Backward-Recovery nur entsprechende Kompensationstransaktionen anstoßen, welche den Effekt einer Aktivität rückgängig machen sollen. Für das WS-Business-Activities-Protokoll wird dafür beispielsweise eine `COMPENSATE`-Nachricht an die jeweiligen Teilnehmer gesendet.

3.2.2. Analyse von Beispielszenarien

Wie in Abschnitt 2.3.5 bereits diskutiert, gibt es viele praktische Szenarien für organisationsübergreifende Prozesse, bei denen der Initiator eines Prozesses nicht derjenige Teilnehmer ist, welcher über die Demarkation von Transaktionen entscheiden kann. Im Folgenden werden jetzt die Protokolle und Standards zur Koordination von Web-Services-Transaktionen in Bezug auf dieses Problem mithilfe von Anwendungsfällen hin untersucht. Dafür werden im nächsten Abschnitt ein klassisches Käufer-Anbieter-Szenario sowie ein reales Choreographie-Szenario auf Basis von Europol und Eurojust diskutiert.

3.2.2.1. Käufer-Anbieter-Szenarien

Die bisher am häufigsten in der Literatur genannten Beispielszenarien für Web-Services-Transaktionen sind Buchungen für Reisen, welche über ein Reisebüro durchgeführt werden. Ein Reisebüro stellt dabei Dienste zum Buchen von Hotels, Flügen und Mietwagen zur Verfügung. Eine Reise besteht dabei aus einer Hotelreservierung, einer Reservierung

[2] Zu den Arten von Aktivitäten siehe auch Kapitel 2.3.1 oder [Gra81].

für Hin- und Rückflug und einer Mietwagenreservierung. Der Reisende versucht dabei, eine Reise mit möglichst minimalen Kosten zu buchen und kombiniert dabei die günstigsten individuellen Angebote für die Reise und sortiert die teuren Angebote dabei aus und zieht die Reservierung der teuren Angebote zurück. Sind die individuellen Aktivitäten der Reise ausgewählt, werden das entsprechende Hotel, die Flüge und der Mietwagen gebucht und somit die Gesamttransaktion abgeschlossen.

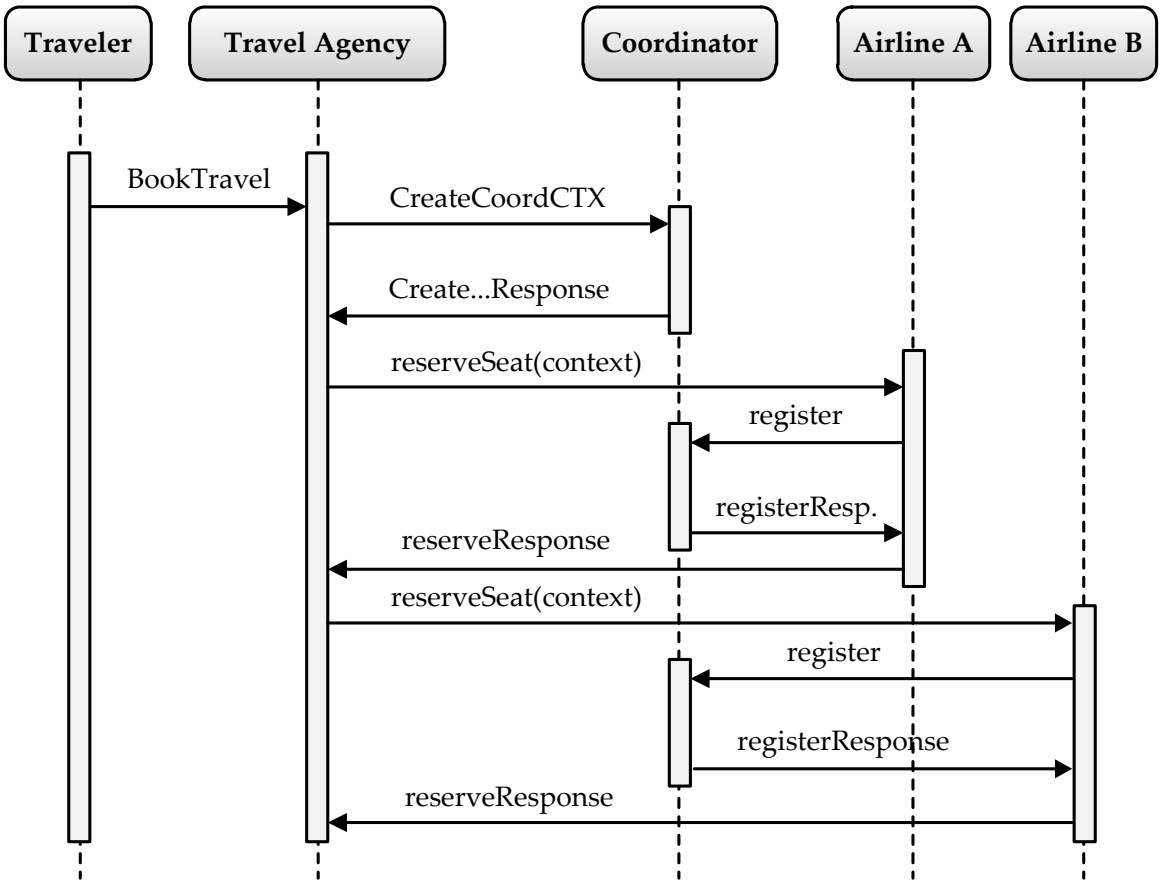


Abbildung 3.2.: Ein beispielhafter Nachrichtenaustausch für WS-Coordination

Abbildung 3.2 zeigt dabei den Nachrichtenfluss für WS-Coordination. In dem Beispiel ist zu sehen, dass dem Transaktionskoordinator am Ende der Reservierungsphase mitgeteilt werden muss, welche der Reservierungen bei Fluglinie *A* oder Fluglinie *B* bestätigt werden soll. Für WS-AtomicTransaction (WS-AT) wird dafür ein *Completion*-Protokoll definiert, welches es dem Reisebüro erlaubt, dem Koordinator mitzuteilen, dass die Transaktion abgeschlossen werden soll [LW09]. Im Falle von WS-AtomicTransaction werden beide Fluglinien atomar die Reservierung bestätigen, was im Anwendungsbeispiel nicht das gewünschte Verhalten darstellt. Für die Reisebuchung soll nur eine Reservierung für einen Flug erfolgreich abgeschlossen werden, die zweite Reservierung muss zurückgezogen werden. Solche Szenarien mit *gemischten Ergebnissen* (Mixed Outcome) werden zwar nicht durch WS-AT aber durch die WS-BusinessActivity-Spezifikation (WS-BA) unterstützt [FL09]. Allerdings unterstützt WS-BA kein entsprechendes Completion-Protokoll aus WS-AT, daher bleibt im

Standard die Art der Kommunikation zwischen Reisebüro und Koordinator unspezifiziert.

Dieses einfache Beispiel zeigt, dass die bekannten Spezifikationen zwar im ersten Schritt versuchen, die transaktionale Koordination von der anwendungsspezifischen Funktionalität zu trennen. Aber schon in diesem Beispiel wird anwendungsspezifisches Wissen benötigt, damit der Koordinator Entscheidungen über das erfolgreiche Abschließen der Transaktion treffen kann. Ein anwendungsneutraler Koordinator, welcher beispielsweise kein hartcodiertes Wissen über das Szenario enthält, kann damit nicht entscheiden, welche Teilnehmer der Transaktion erfolgreich abschließen sollen und welche nicht. Das notwendige Wissen dafür liegt in der entsprechenden Anwendung, welche normalerweise den *Initiator* einer Transaktion beziehungsweise eines verteilten Prozesses darstellt.

Weiterhin zeigt das Beispiel, dass im Prozess nicht jede Prozessaktivität erfolgreich beenden muss, damit der Prozess und damit die Transaktion erfolgreich ist. Im vorgestellten Szenario wird beispielsweise nur die Sitzplatzreservierung für einen Hin- und Rückflug benötigt, aber nicht zwei oder mehr. Daher ist es nicht fatal, wenn eine Fluglinie die Reservierung zurückzieht oder die Reservierung allgemein fehlschlägt, solange es eine erfolgreiche Reservierung gibt. Für diesen Zweck führt die Literatur die Eigenschaft der *Vitalität* (Vitality) von transaktionalen Teilnehmern ein [GMGK$^+$90]. Ein Teilnehmer ist für eine Transaktion vital, wenn er für den erfolgreichen Ausgang der Transaktion benötigt wird. Im Kontrast dazu führt ein Fehler eines nicht-vitalen Teilnehmers nicht sofort zum Fehlschlag der Gesamttransaktion. Die Atomarität ist für diese Prozesse also keine Bedingung mehr und wird damit aufgeweicht. WS-BA unterstützt das Aufweichen der Atomarität durch zwei verschiedene Techniken:

- Durch zwei verschiedene Koordinationstypen: Auf der einen Seite gibt es den Koordinationstyp *Atomic-Outcome*, durch welchen die bereits von Datenbanktransaktionen bekannte Alles-oder-Nichts-Eigenschaft durchgesetzt wird. Auf der anderen Seite gibt es den Koordinationstyp *Mixed-Outcome*, um genau die eben beschriebenen gemischten Ergebnisse in der Reisebuchung zu unterstützen. Dieser Koordinationstyp adressiert indirekt die Vitalität von Teilnehmern, weil nicht alle Teilnehmer für den erfolgreichen Abschluss der Transaktion notwendig sind. Allerdings wird die Vitalität durch die Spezifikationen nicht direkt modelliert, es existieren dafür weder Artefakte zum Annotieren dieser Eigenschaft noch wird dies in den Protokollen weiter genutzt.

- Die WS-BA-Spezifikation unterstützt das (ungesicherte) Zurückziehen von Teilnehmern aus der Transaktion. Für das Beispiel aus Abbildung 3.2 kann dies eine Fluglinie sein, welche sich mit einer `Exit`-Nachricht an den Koordinator von der Transaktion abmeldet. Der Koordinator besitzt allerdings keinerlei Wissen darüber, ob ein ausgetretener Teilnehmer vital ist oder nicht. Daher benötigt der Koordinator weiteres Wissen über die Vitalitäts-Eigenschaft von Teilnehmern, um Transaktionen koordinieren und damit entscheiden zu können, ob die Transaktion erfolgreich war oder nicht.

Offensichtlich ist die Eigenschaft der Vitalität eine gute Eigenschaft, um Transaktionen mit gemischten Ergebnissen robuster durchzuführen. Das ungeschützte Austreten von Teilnehmern aus einer Transaktion, wie sie durch WS-BA mit einer `Exit`-Nachricht unterstützt wird, oder die Entscheidung über gemischte Ergebnisse sind ohne Anwendungslogik nicht trivial. Trotzdem benötigt ein allgemeines Koordinationsrahmenwerk eine klare Trennung zwischen Anwendungs- und Koordinationslogik, da sonst ein allgemein einsetzbarer Koordinator als softwaretechnischer Dienst nicht möglich ist. Daher zeigt das einfache Reisebuchungsbeispiel schon verschiedene Schwächen der Spezifikation:

- Es gibt *keine saubere Trennung aus Anwendungs- und Koordinationslogik*, um gemischte Ergebnisse zu koordinieren.
- Die *Vitalität* als Eigenschaft von transaktionalen Teilnehmern ist bei WS-Coordination und seinen Protokollen nicht vorgesehen.

Um die Trennung zwischen Anwendung und Koordinator sauberer zu realisieren, lassen sich zwei Konzepte identifizieren:

- Das *Proxy*-Konzept: Ein Koordinator agiert als Proxy, beispielsweise als Hub zwischen Initiator und den anderen Teilnehmern der Transaktion. Entscheidungen eines Initiators werden über den Koordinator an die Teilnehmer weitergeleitet. Ein Koordinator stellt dabei sicher, dass die Entscheidung des Initiators von allen Teilnehmern eingehalten wird, und teilt das Ergebnis der Koordination dem Initiator mit. Der Koordinator wird explizit durch den Initiator durch ein Kontrollprotokoll gesteuert; das Completion-Protokoll von WS-AT ist eine Implementierung des Proxy-Konzeptes.
- Das *Rückfrage*-Konzept: Ein Koordinator entscheidet selbstständig über die Koordination. Im Falle von anwendungsspezifischen Entscheidungen kontaktiert der Koordinator den Initiator, um die entsprechende Entscheidung zu erfragen, die er dann bei den Teilnehmern durchsetzt. Dieser Ansatz erlaubt einen höheren Grad der Entkopplung zwischen Initiator und Koordinator als das Proxy-Konzept, da der Initiator nur noch bei anwendungsspezifischen Entscheidungen gefragt werden braucht.

Trotzdem bleibt in beiden Konzepten der Initiator die treibende Kraft für Entscheidungen über die Teilnehmer an einer Transaktion, das Rückfragekonzept bietet daher hauptsächlich nur technische Vorteile gegenüber dem Proxy-Konzept. Verschiedene Ansätze haben bereits das Proxy-Konzept für WS-BA implementiert (siehe Abschnitt 2.3.1). Bisher gibt es aber keine Implementierung des Rückfrage-Konzepts.

3.2.2.2. Europol/Eurojust-Szenario

Die Behörden Eurojust und Europol sind gegründet worden, um den EU-Mitgliedsstaaten bei der Kooperation zur Bekämpfung von grenzüberschreitender Kriminalität zu unter-

stützen. Eurojust ist dabei die europäische Behörde für justizielle Zusammenarbeit während Europol die europäische Polizeibehörde darstellt. Beide zusammen bieten verschiedene Dienstleistungen für die Zusammenarbeit von Behörden der EU-Mitglieder im Kontext von gemeinsamen Polizeiaktivitäten oder Justizvorgängen an.

Ein Beispielprozess zwischen diesen beiden Behörden sowie deren Unterorganisationen ist auch im EU-finanzierten Forschungsprojekt R4eGov vorgestellt worden [Bou07]: Ein spanischer Polizist findet bei einer Routineuntersuchung im Hafen von Malaga Kokain auf einem Schiff, welches Kaffee geladen hat. Der Container mit dem Kokain kommt aus Caracas in Venezuela und sollte nach Antwerpen in Belgien transportiert werden. Verschiedene Personen werden bei der Untersuchung auf dem Schiff in Gewahrsam genommen. Der Polizist erkennt, dass dieser Vorgang nicht alleine in Spanien geklärt werden kann und Europol sowie Eurojust müssen in diesem Fall mit allen beteiligten Mitgliedsstaaten zusammenarbeiten. Der spanische Polizist ist innerhalb eines solchen Prozesses nur der Initiator. Allerdings ist er nicht derjenige Teilnehmer am Prozess, welcher entscheiden kann, welche Aktivitäten als Nächstes durchgeführt oder im Sinne einer Transaktion später erfolgreich beendet oder wieder rückgängig gemacht werden müssen.

Weder das Completion-Protokoll von WS-AT noch deren Adaptionen für WS-BA spezifizieren in diesem Falle, wer sich für das Completion-Protokoll registrieren kann. Der Initiator ist in allen Protokollen derjenige Teilnehmer, welcher implizit der kontrollierende Teilnehmer im Prozess ist. Dabei wird die *Rolle* des Initiators mit dem *Privileg* der Demarkation einer Transaktion vermischt. Das Verhalten von WS-AT ist sogar unspezifiziert, wenn sich mehrere Teilnehmer für das Completion-Protokoll registrieren. Andere Protokolle wie BTP oder WS-CAF erlauben das mehrfache Registrieren für ein Completion-Protokoll gar nicht erst.

Um das beschriebene Europol/Eurojust-Szenario zu unterstützen, ist mindestens ein explizites Management des Demarkations-Privilegs notwendig, in dem der kontrollierende Teilnehmer für die Demarkation zu jeder Zeit definiert ist. Trotzdem ist es teilweise schwer, einen expliziten Teilnehmer mit dem notwendigen Wissen in einem Prozess zu identifizieren und ihm mit dem Privileg auszustatten. Eine weitere Möglichkeit ist das Verteilen der Anwendungslogik auf mehrere Teilnehmer anstatt auf den Initiator alleine. Beispielsweise braucht ein Teilnehmer nicht die gesamte Reisebuchung zu kennen, sondern nur die Flugreservierungen, aus der eine Reservierung ausgewählt werden soll. Ein Koordinator kann diese lokalen Sichten beispielsweise aggregieren und dann aufgrund der aggregierten Sichten entscheiden, wann der Transaktionsabschluss eingeleitet werden soll und welche Teilnehmer für den erfolgreichen Abschluss notwendig sind und welche nicht.

3.2.3. Implikationen für diese Arbeit

Zusammenfassend lassen sich die Anforderungen in dieser Arbeit an die transaktionale Kontrolle und die Fehlerbehebung wie folgt weiter verfeinern:

- **Unterstützung von Backward-Recovery zur Fehlerbehebung**:
  Die transaktionale Koordination soll bei durch die Laufzeitverifikation signalisierten Interaktionsfehlern und beobachtbaren physischen Fehlern Aktivitäten einer Choreographie koordiniert zurücksetzen. Dabei soll die Demarkation einer Transaktion nicht allein durch einen Initiator oder durch die Teilnehmer selbst erfolgen, sondern zusätzlich auch durch die Laufzeitverifikation initiiert werden können.

- **Einsetzbarkeit von Transaktionsprotokollen in Choreographien**:
  Die vorangegangene Diskussion von Beispielszenarien hat verschiedene Schwächen im Rahmen von Choreographien offenbart:

  – **Trennen von Anwendungs- und Koordinationslogik**:
    Sollen Transaktionsprotokolle zur Fehlerbehebung eingesetzt werden, muss Anwendungslogik nach Möglichkeit von der Koordinationslogik getrennt werden, um generische Koordinatoren implementieren zu können. Die bisherigen Standards unterstützen dieses nur teilweise.

  – **Einfluss von Fehlern kategorisieren**:
    In Choreographie-Szenarien ist zu sehen, dass Fehler einzelner Teilnehmer nicht unbedingt zu einem Abbruch einer Transaktion führen müssen. Dafür muss bei einer automatisierten Fehlerkorrektur aber definiert sein, wie sich ein Fehler auswirkt. Dafür muss die *Vitalität* von Teilnehmern genutzt werden, damit ein Transaktionskoordinator entsprechend reagieren kann.

  – **Reduzierung der Abhängigkeit vom Initiator**:
    In klassischen Transaktionsbeispielen entscheidet immer der Initiator über die Demarkation einer Transaktion. Dieses muss in Choreographien nicht der Fall sein, sodass klassische Transaktionsmodelle nicht alle Szenarien von Choreographien unterstützen. Sie müssen daher erweitert werden, um die Abhängigkeit vom Initiator zu verringern oder sogar ganz auszuschließen.

Um diese Anforderungen zur Anwendung von Transaktionen in Choreographien umzusetzen, werden in Kapitel 5.1 entsprechende Lösungen erarbeitet. Soll die Demarkation von Transaktionen zur Fehlerbehebung zusätzlich durch die Laufzeitverifikation angestoßen werden, werden Lösungen dazu später in Kapitel 5.2 vorgestellt.

4. Laufzeitüberwachung von Prozess-Choreographien

Damit die Laufzeitverifikation von Choreographien durchgeführt werden kann, muss technisch dafür zum einen die Infrastruktur bereitgestellt werden, um Nachrichten innerhalb einer Choreographie zu protokollieren. Zum anderen müssen Verifikationstechniken entwickelt werden, um Choreographien zu verifizieren. Dafür wird im ersten Teilabschnitt die entsprechende Monitoring-Infrastruktur vorgestellt, die den bisher in dieser Arbeit entwickelten Anforderungen genügt. Im zweiten Teil werden zwei verschiedene Verifikationsmechanismen für WS-CDL auf Basis von Prädikatenlogik und Ereigniskalkül entwickelt, evaluiert und verglichen.

4.1. Technische Infrastruktur zur Verifikation von Choreographien

In diesem Abschnitt wird die Infrastruktur vorgestellt, mit deren Hilfe die Sensordaten für die Laufzeitverifikation von Choreographien bereitgestellt sowie die Verifikation durchgeführt werden. Dafür muss zur Lokalisierung von Sensoren im ersten Schritt diskutiert werden, wie allgemeine Infrastrukturen zur Unterstützung von organisationsübergreifenden Prozessen und Choreographien heute aussehen. Abbildung 4.1 zeigt dafür ein abstraktes Szenario der Zusammenarbeit, in der jede Organisation ein internes Verhalten besitzt, welches durch interne Prozesse realisiert wird. Jede dieser Organisationen besitzt eigene Schnittstellen, um mit anderen Organisationen zu kommunizieren. Diese Schnittstellen sind als Verhaltensschnittstellen in Abbildung 4.1 bezeichnet.

Innerhalb von organisationsübergreifenden Prozessen sind die Teilnehmer dabei autonom und haben die Kontrolle über ihre eigenen IT-Landschaften, obwohl sie sich für einen Prozess mit einem gemeinsamen Ziel zusammenschließen. Weiterhin entscheiden die Organisationen bei der Wahl der verwendeten Technologien selbstständig, daher ist von einer vollständig heterogenen Systemlandschaft auszugehen. Um diese Heterogenität zu überwinden, werden dienstorientierte Architekturen (SOA) genutzt, welche üblicherweise über Web-Services implementiert sind. Zur Implementierung solcher Architekturen für organisationsübergreifende Prozesse kategorisieren Schroth, Janner und Hoyer drei verschiedene Arten [SJH08] von Architekturen:

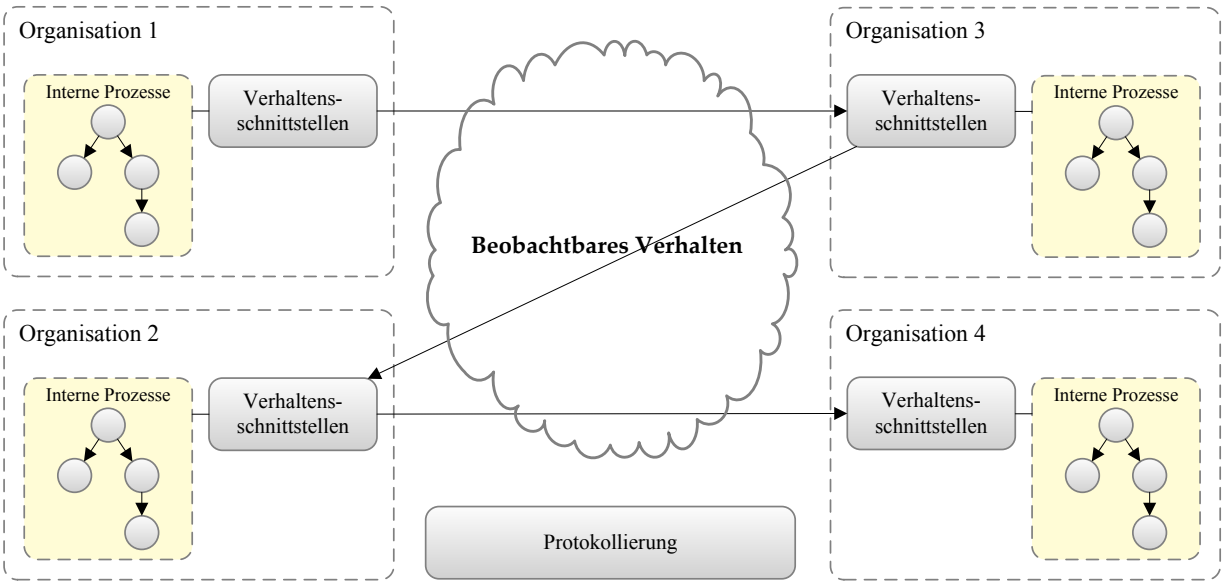


Abbildung 4.1.: Organisationsübergreifende Prozesse

- Die **zentralisierte Dienstorchestrierung**: Dieses Model der zentralen Kontrolle arbeitet mithilfe einer Workflow-Engine. Diese steuert den Kontrollfluss aller beteiligten Teilnehmer und realisiert damit das Konzept der Orchestrierung. Mit Unterstützung dieser Architektur lässt sich zwar ein organisationsübergreifender Prozess steuern, allerdings steuert ein Teilnehmer alle anderen, was der Autonomie der Organisationen widerspricht.

- Die **hub-basierte dezentrale Orchestrierung**: In diesem Model gibt es lokale Prozesse, die von den lokalen Teilnehmern selbst gesteuert werden. Diese lokalen Prozesse kommunizieren dabei über eine entsprechende Middleware, welche die Kommunikation zwischen den verschiedenen Teilnehmern der Choreographie realisiert.

- Die **dezentrale Orchestrierung ohne Hub**: Auch in diesem Model gibt es lokale Prozesse, welche über Nachrichtenaustausch mit anderen Teilnehmern interagieren. Allerdings gibt es hier keine zentralen Infrastrukturen mehr, jeder Teilnehmer muss mit jedem anderen Teilnehmer entsprechend bilateral aushandeln, wie die Kommunikation aussehen kann.

Die drei Architekturen stellen zum einen die beiden Extreme der vollständig zentralen Steuerung (Dienstorchestrierung) sowie der vollständig dezentralen und verteilten Steuerung dar und zeigen zum anderen eine hybride Möglichkeit auf, bei der nur Teile zentral vorgehalten werden, aber die Steuerung der lokalen Prozesse bei den Teilnehmern verbleibt. Zentral bleiben die Definition des Choreographie-Modells, welche alle Teilnehmer vor Implementierung der Choreographie abgestimmt haben, sowie eine gemeinsam genutzte Nachrichteninfrastruktur, welche als Vermittler zwischen verschiedenen Systemen

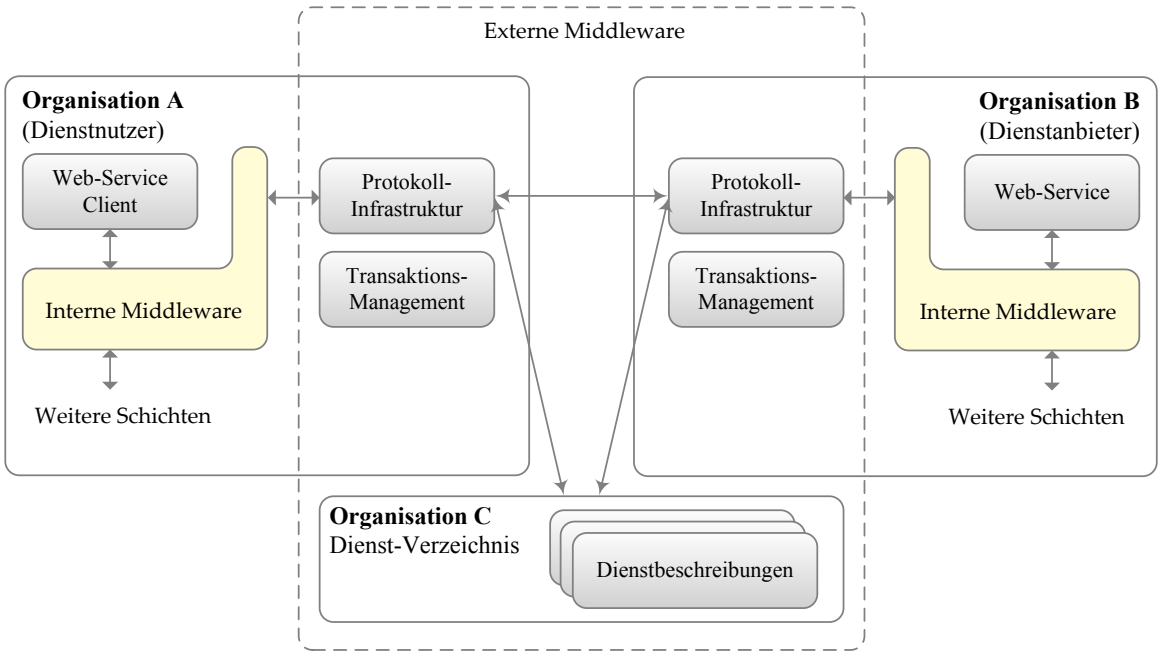


Abbildung 4.2.: Externe Architektur eines Web-Services; siehe [ACKM04].

genutzt wird. Alonso, Casati, Kuno und Machiraju beschreiben in [ACKM04] entsprechende Architekturen, in dem sie zwischen internen Web-Services-Architekturen mit klassischer Middleware und externen Web-Services-Architekturen unterscheiden, die als Middleware zwischen internen Web-Services-Architekturen implementiert werden, wie in Abbildung 4.2 dargestellt. Prinzipiell lassen sich damit hub-basierte dezentrale Orchestrierung und auch dezentrale Orchestrierung ohne Hub implementieren, allerdings kommt es dabei nur auf den Unterschied an, ob es eine geteilte Protokollinfrastruktur oder für jede Verbindung zwischen zwei Teilnehmern eigene Infrastrukturen gibt.

In der Literatur ist zu sehen, dass der hybride Ansatz der hub-basierten dezentralen Orchestrierung oft zum Einsatz kommt, da es bei Nutzung von verschiedenen Web-Services-Standards zu Inkompatibilitäten kommen kann. Dies spricht, wie in [SJH08] beschrieben, stark für eine Architektur auf Basis einer hub-basierten Orchestrierung, denn durch den Einsatz eines Hubs als vermittelnde Zwischenschicht lassen sich solche Inkompatibilitäten (zumindest teilweise) überwinden. Im Bereich der SOA wird für einen solchen Hub ein Enterprise-Service-Bus (ESB) genutzt, welcher als Mediator für Interaktionen zwischen verschiedenen Web-Services und Web-Service-Standards fungiert [Ley05]. Verglichen mit den Diensten in Abbildung 4.2 übernimmt ein Service-Bus die Aufgaben der Protokoll-Infrastruktur und die des Dienstverzeichnisses. Die Annahme, dass ein Enterprise-Service-Bus für die Implementierung von organisationsübergreifenden Prozessen auf Basis von SOA genutzt wird, wird auch durch andere Veröffentlichungen wie beispielsweise [SIM07] verstärkt, da durch den Service-Bus noch weitere Aufgaben wie Sicherheitsanforderungen oder Lastausgleich zwischen Diensten realisiert werden können. Liegt ein Service-Bus als Nachrichteninfrastruktur zwischen den beteiligten Organisationen vor, lassen sich entsprechende

Sensoren in diese Middleware einbetten, um die Interaktionen einer Choreographie zu protokollieren. Die verwendeten Sensoren sind dabei weitestgehend nicht intrusiv und verletzen damit die Autonomie der Organisationen nicht, da in diesem Falle nur das nach außen sichtbare Verhalten der Organisationen durch die Sensoren beobachtet werden kann.

4.1.1. Konsistente Sensordaten

Wie in den Architekturanforderungen in Abschnitt 3.1.3.1 beschrieben, wird eine verlässliche Infrastruktur benötigt, die alle Nachrichten einer Choreographie konsistent beobachten kann. Weiterhin wird, wie im letzten Abschnitt beschrieben, angenommen, dass bei organisationsübergreifenden Prozessen Web-Services zum Einsatz kommen und sämtliche Kommunikation durch einen Enterprise-Service-Bus als Middleware geleitet wird. Der Enterprise-Service-Bus spielt dabei die Rolle eines Mediators zwischen den Organisationen und ermöglicht damit das Beobachten sämtlicher öffentlich ausgetauschter Nachrichten einer Choreographie.

Es gibt mittlerweile eine Vielzahl von sowohl kommerziellen als auch Open-Source-Produkten, welche einen Enterprise-Service-Bus implementieren. Im Rahmen der Arbeit wird der Service-Bus *Synapse* der Apache Software-Foundation[1] genutzt. Zum einen liegen bei dieser Implementierung die Quellen vollständig frei, was eine einfache Erweiterung ermöglicht und zum anderen ist Synapse oft auch Basis kommerzieller Implementierungen, wie beispielsweise dem Enterprise-Service-Bus von WSO_2, welcher Synapse größtenteils nur um administrative Werkzeuge erweitert[2].

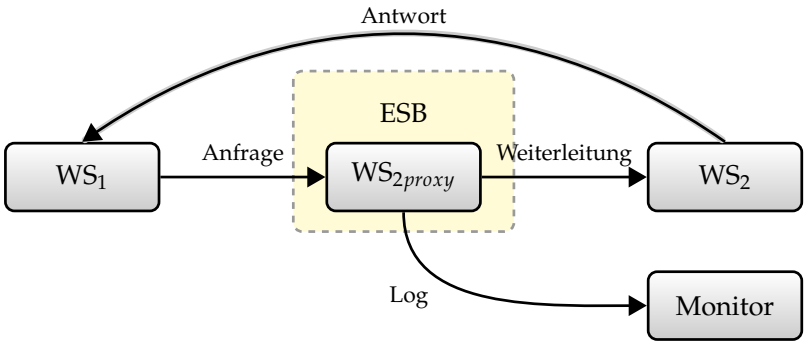


Abbildung 4.3.: Überwachung von Interaktionen via Proxy im Enterprise-Service-Bus

Die Überwachung zur Kontrolle von Interaktionen mithilfe des ESB erfolgt über Stellvertreterdienste, die für jeden Dienst, welcher an der organisationsübergreifenden Kommunikation teilnimmt, im ESB definiert wird. Diese Stellvertreterdienste haben zwei Aufgaben: Zum einen das Weiterleiten der Nachrichten an den (oder die) eigentlichen Empfänger

[1] Siehe auch `http://synapse.apache.org`.
[2] Siehe `http://wso2.com/products/enterprise-service-bus`.

und zum anderen das Vermitteln zwischen verschiedenen Protokollen wie beispielsweise zwischen verschiedenen Sicherheits- oder Transportprotokollen von teilnehmenden Organisationen. Abbildung 4.3 veranschaulicht dieses Prinzip sowie die Instrumentierung der Sensoren, welche Nachrichten direkt an einen Monitor weiterleiten. Das Einbetten von Sensoren in einem ESB hat zwei Vorteile. Zum einen wird die Autonomie der Teilnehmer bei der Wahl der Technologien zur Implementierung ihrer Web-Services-Infrastruktur nicht vermindert, weil ein Sensor zwischen den teilnehmenden Organisationen installiert wird und zum anderen wird der normale Geschäftsablauf nicht wesentlich beeinflusst, da ein Speichern und Weiterleiten der Nachrichten keinen wesentlichen Einfluss auf die Laufzeiten der Nachrichten besitzt, solange der Proxy zum Protokollieren nicht ausfällt. Weiterhin kann der Sensor ereignisorientiert betrieben werden, wie es in den Anforderungen in Abschnitt 3.1.3.1 definiert worden ist.

Durch Wahl dieser Sensorarchitektur entstehen zwei verschiedene Herausforderungen: Erstens muss sichergestellt werden, dass alle Nachrichten über Stellvertreterdienste im ESB gesendet werden, damit alle Nachrichten einer Choreographie beobachtet werden können. Dafür müssen die üblichen Service-Interaktionsmuster wie In-Only, In-Out, Out-Only oder Out-In (siehe auch [GHM+07]) überprüft werden, damit bei jeder Art von Interaktion die Nachrichten auch durch den Bus geleitet werden. In Abbildung 4.3 ist ein Problem mit einer asynchronen In-Out-Interaktion dargestellt, denn die Antwort einer asynchronen Web-Service-Interaktion wird nicht durch den Bus geleitet. Bei dieser Art von Interaktion wird mithilfe von WS-Addressing im Kopf einer Anfrage eine `replyTo`-Adresse definiert, wohin eine Antwort auf eine Anfrage gesendet werden soll. Wird an dieser Stelle kein Stellvertreterdienst als Antwortziel angegeben, wird eine entsprechende Antwort am Bus vorbei gesendet und der Sensor erhält über die Antwort keine Kenntnis. Dieses Problem der Infrastruktur muss behoben werden. Zweitens muss in dieser Architektur sichergestellt werden, dass jede Interaktion atomar geloggt wird. Atomarität bedeutet in diesem Falle, dass die zwei Aktionen des Weiterleitens an den eigentlichen Empfänger und an den Monitor als eine Einheit ausgeführt werden. Im Detail bedeutet dies, dass eine Nachricht, die den Empfänger erreicht hat, auch beim Monitor angekommen sein muss und umgekehrt. In beiden Fällen würde ohne entsprechende Maßnahmen das Laufzeitverhalten vom verifizierten Verhalten abweichen, was unbedingt zu vermeiden ist.

Bisher wurde angenommen, dass nur ein einziger Service-Bus zwischen den Organisationen vorhanden ist, was aber nicht immer zutreffend sein muss. Organisationen können die Kommunikation auch mit jedem Partner einzeln abstimmen und damit kann jede Organisation einen eigenen Service-Bus betreiben. Im Falle eines einzigen ESBs kann sichergestellt werden, dass für eine Verifikation später die Nachrichtenreihenfolge durch den Bus eingehalten wird oder zumindest die Zeitstempel der Nachrichten eine partielle Ordnung bilden. Bei nur einem Service-Bus ist diese Annahme einfach zu treffen, da die Zeitstempel der Nachrichten durch eine zentrale Instanz vergeben werden und damit die Zeitstempel konsistent sind. Sie lassen sich dann später durch einen Verifikationsdienst wieder in die

richtige Reihenfolge bringen, falls diese in leicht unterschiedlicher Reihenfolge beim Monitor eintreffen. Diese Annahme ist bei der Nutzung von mehreren ESBs hinfällig, in diesem Falle müssen die Uhren der ESBs zwingend synchronisiert sein, damit Zeitstempel für die Entscheidung der Reihenfolge von Ereignissen herangezogen werden können.

4.1.1.1. Überwachung von verschiedenen Interaktionsmustern

Um alle in einer Choreographie auftretenden Interaktionen zu überwachen, muss also jegliche Kommunikation über den Enterprise-Service-Bus geleitet werden. Der Bus fungiert somit als Mediator zwischen den Organisationen. Für die normalen Austauschmuster wie In-Only, In-Out, Out-Only oder Out-In gibt es dabei keine Probleme, wenn die Kommunikation synchron erfolgt. Die Kommunikation erfolgt dabei über Proxy-Dienste, welche die Nachrichten an den richtigen Empfänger im richtigen Format weiterleiten. Antworten werden dabei auch direkt über den Proxy zurück an den Empfänger geleitet. Anders sieht es aus, wenn der Austausch der Nachrichten asynchron erfolgt. Beispielsweise wird beim In-Out-Muster bei asynchroner Web-Service-Kommunikation WS-Addressing genutzt, um damit im Kopf einer Nachricht entsprechende Reply-To-Empfänger zu definieren. Produkte wie Apache Synapse sind dabei nicht mehr in der Lage, Antworten wieder zurück über den Proxy zu leiten. Die Nachricht wird damit am Bus vorbei direkt an dem durch den im Reply-To-Nachrichtenkopf definierten Empfänger gesendet. Dies trifft auch für das Out-In-Muster zu. Eine unidirektionale Kommunikation wird dagegen problemlos über den Bus geleitet.

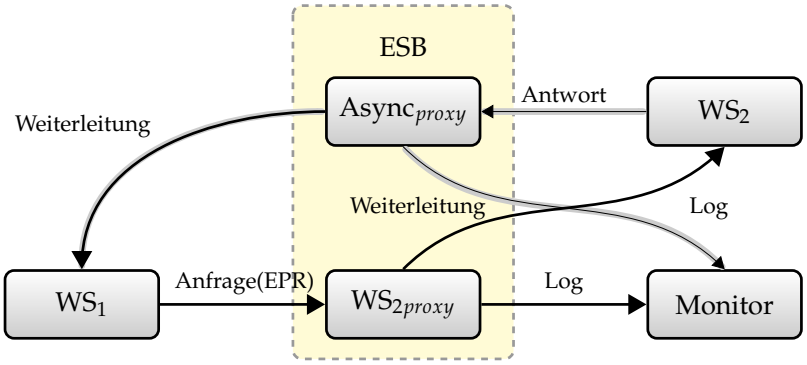


Abbildung 4.4.: Überwachung via Proxy und WS-Addressing-Mediator

In [DKS07] beschreiben Davies, Krishna und Schorow, dass ein Enterprise-Service-Bus bestimmte Aufgaben und Eigenschaften erfüllen muss, die sie als Ortstransparenz, lose Kopplung, Mediation, Schema-Transformation, Service-Aggregation, Lastverteilung, Sicherheit, Monitoring und Konfigurierbarkeit zusammenfassen. Durch die asynchrone Nachrichtenübertragung und das Versenden von Antworten am Bus vorbei verletzen ESBs aber genau diese *Ortstransparenz*, denn ein Dienstnutzer kommuniziert nicht mehr nur mit dem Bus, sondern im Antwortfall direkt mit den eigentlichen Diensten. Damit alle Interaktionsmus-

ter unterstützt werden, muss also die Ortstransparenz für das System hergestellt werden.

Zur Lösung des Problems wird für den in dieser Arbeit benutzten Enterprise-Service-Bus Apache Synapse ein Mediator implementiert, welcher bei asynchroner Kommunikation für die beiden Muster In-Out und Out-In entsprechend die Reply-To-Adressen austauscht und die Antworten damit auf einen generischen Reply-To-Proxy umleitet. Die Kommunikation kann jetzt wieder überwacht werden, wie auch in Abbildung 4.4 dargestellt. Durch Nutzung des Mediators kann garantiert werden, dass jegliche Kommunikation über den Service-Bus geleitet wird und damit überwacht werden kann.

4.1.1.2. Atomare Interaktionsprotokollierung

Die bisherige Architektur impliziert die grundlegende Herausforderung, jegliche Kommunikation zu erfassen. Zum einen muss dafür sichergestellt werden, dass jegliche erfolgreiche Kommunikation an einen Empfänger über den Bus erfasst und an den Monitor zur Verifikation gesendet wird. Zum anderen darf aber auch nichts an den eigentlichen Empfänger weitergeleitet werden, was nicht auch über den Monitor erfasst wird. Abbildung 4.5 illustriert das Problem: Ein Dienst WS_1 ruft den Dienst WS_2 über den Service-Bus auf. Innerhalb des Service-Busses leitet der Proxy WS_{2proxy} die Nachricht an WS_2 weiter und erfasst die Nachricht zusätzlich durch Weiterleiten an einen Monitor. Die Herausforderung hier ist, dass dieses atomar geschehen muss, damit eine Verifikation später auch korrekte Daten erhält.

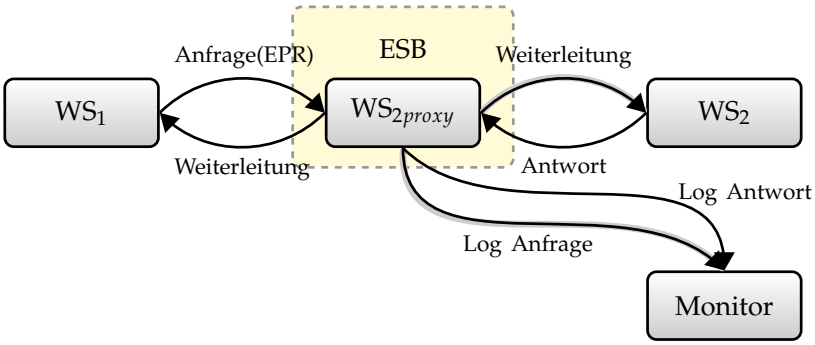


Abbildung 4.5.: Atomares Erfassen von Web-Service-Interaktionen

Als erste Lösungsmöglichkeit bietet sich hier das Paradigma der Transaktion an, wobei an dieser Stelle nicht die genaue Transaktionssemantik benötigt wird. Beispielsweise sollte eine Aktion in einem Prozess nicht rückgängig gemacht werden, wenn eine Log-Aktion fehlschlägt. Das Protokollieren und die Verifikation darf hier die normale Ausführung der Choreographie also nicht behindern. In diesem Falle muss erkannt werden, ob die Log-Nachricht an den Monitor gesendet worden ist oder nicht. Bei Bedarf kann die Log-Nachricht dann erneut gesendet werden. Umgekehrt muss aber eine Log-Nachricht wider-

rufen werden, falls festgestellt wird, dass der eigentliche Empfänger der Choreographie-Nachricht nicht erhalten beziehungsweise nicht verarbeitet hat.

Eine weitere Lösungsmöglichkeit für das Problem ist WS-ReliableMessaging (WS-RM), welches als Standard für Web-Services-Kommunikation Eigenschaften wie *exakt einmal* bei der Nachrichtenübertragung sicherstellen soll [FPD+09]. Diese Lösung kann beispielsweise für die Übertragung von Nachrichten vom Proxy zum Monitor genutzt werden. Bei der Übertragung an eine Organisation kann allerdings nicht immer voraussetzt werden, dass entsprechende Dienste einer Organisation diesen Standard unterstützen.

Falls eine Organisation WS-RM nicht unterstützt, wird also ein anderer Mechanismus zum Sicherstellen des atomaren Erfassens und Versendens benötigt. In dieser Arbeit wird dafür ein Bestätigungsmechanismus genutzt. Im ersten Schritt sendet der Proxy-Dienst dafür die Nachricht an den eigentlichen Empfänger und wartet auf eine Bestätigung. Trifft diese ein, wird die Nachricht zusätzlich mithilfe von WS-RM an den Monitor gesendet. Der Monitor weiß mit Erhalt der Nachricht, dass diese den Empfänger erreicht hat. Im Fehlerfall, wenn etwa keine Bestätigung vom eigentlichen Empfänger empfangen wurde, gibt es zwei Möglichkeiten der Interpretation. Auf der einen Seite kann eine Anfrage einen Empfänger erreicht haben, aber die Bestätigung der Anfrage ist verloren gegangen. Auf der anderen Seite kann die Nachricht den eigentlichen Empfänger nicht erreicht haben, sodass auch keine Bestätigung gesendet werden kann. In beiden Fällen ist es möglich, die Entscheidung in diesem Fehlerfall an die Choreographie zurückzugeben, welche beispielsweise dafür ein erneutes Senden der Nachrichten modellieren kann. Dieser Mechanismus muss dann allerdings von den Teilnehmern umgesetzt werden.

Durch Nutzen dieser Vorgehensweise kann zum einen die Nachrichtenübertragung zum Monitor und zum anderen die Übereinstimmung zwischen beobachtetem und realem Laufzeitverhalten garantiert werden. Die Vorgehensweise erhöht die Autonomie der beteiligten Organisationen, da sie keine weitere Software installieren müssen. Allerdings müssen für den entsprechenden Fehlerfall, dass keine Bestätigungsnachricht für den Empfang einer Anfrage gesendet wird, Maßnahmen wie das erneute Senden einer Anfrage im Prozess vorgesehen werden.

4.1.1.3. Implementierung

Synapse als Enterprise-Service-Bus realisiert seine Dienstleistungen auf Basis von Stellvertreterdiensten. Soll ein Dienst von einem Nutzer angesprochen werden, wird die Dienstinteraktion nur über diese Proxies ermöglicht. Ein Proxy innerhalb des Service-Busses nimmt durch sogenannte Mediatoren verschiedene Aufgaben wie die Transformation von Nachrichten, Routing, Load-Balancing oder auch das Vervielfältigen von Nachrichten wahr. Diese Mediatoren werden in einer Sequenz hintereinander ausgeführt und können entsprechend der Synapse-API auch selbst implementiert werden. Abbildung 4.6 macht dieses Vorgehen noch einmal deutlich. Dabei unterscheidet Synapse zwischen eingehenden und aus-

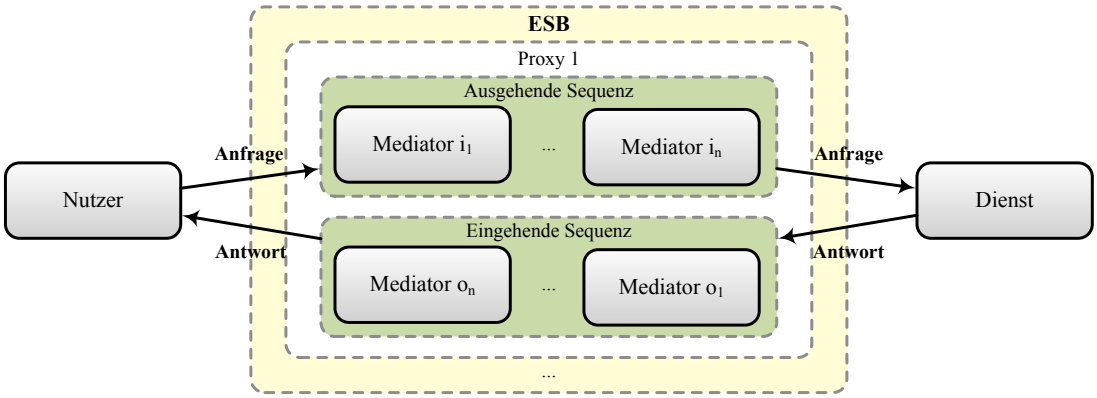


Abbildung 4.6.: Schematische Darstellung des Synapse-ESBs; nach [Sch09]

gehenden Sequenzen, die sich jeweils um die Verarbeitung einer Dienstanfrage oder einer Dienstantwort kümmern.

Die Konfiguration eines Stellvertreterdienstes, seiner Mediatoren und der Sequenzen lässt sich über einen eigenen für Synapse entwickelten XML-Dialekt definieren. Mit Unterstützung dieser Konfigurationssprache lassen sich die Endpunkte der Dienste, Stellvertreterdienste und die verwendeten Mediator-Klassen inklusive der Parameter für die jeweiligen Sequenzen beschreiben. Um einen Sensor zum Mitschneiden von beobachtbaren Nachrichten zu implementieren, muss in Synapse ein entsprechender Stellvertreter für den Dienst eingerichtet werden, bei dem ein *Log-Mediator* in die entsprechenden Sequenzen eingebettet wird.

Überwachung von verschiedenen Interaktionsmustern: Um die Ortstransparenz bei Synapse herzustellen, müssen die Proxies nur bei asynchroner Nachrichtenkommunikation angepasst werden. Bei asynchroner und nicht-blockierender Kommunikation sendet ein Dienstnutzer eine Nachricht und gibt für entsprechende Antworten eine Antwortadresse mit. Dafür wird über WS-Addressing der SOAP-Header `ReplyTo` als Antwortadresse gesetzt, die wie in Abbildung 4.4 dargestellt durch die Adresse eines Proxies im Service-Bus ersetzt wird.

Damit der Bus dieses umsetzen kann, werden zwei Stellvertreterdienste benötigt. Zum einen ist dies ein Proxy für die Anfrage, welcher die jeweiligen Routing-Informationen auf einen zweiten Proxy im Bus ersetzt und zum anderen ist das der Proxy, welcher die Antwort erhält und diese an die vorher gesetzte Adresse weiterleitet. Anfrage und Antwort müssen protokolliert werden, weshalb in beiden Eingangssequenzen der Proxies ein Log-Mediator seine Arbeit verrichtet. Davor werden die entsprechenden WS-Addressing-Header zwischengespeichert, ausgetauscht (je nachdem, ob Anfrage oder Antwort verarbeitet wird) und dann an die Empfänger versendet. Dieses Vorgehen ist in Abbildung 4.7 dargestellt, in dem für die Anfrage und Antwort jeweils ein eigener Proxy mit eigener Eingangsequenz definiert wird. Die notwendigen Mediator-Klassen sind ein Log-Mediator zum

Weiterleiten der Nachrichten an den Monitor, ein Mediator zum Austauschen und Zwischenspeichern der Nachricht und der Sende-Mediator, welcher die Nachrichten an den Empfänger versendet. Beim Synapse-ESB gibt es noch eine Besonderheit bei Eingangsequenzen, da alle Nachrichten am Ende einer Eingangsequenz an einen internen Sende-Mediator geleitet werden. Innerhalb dieses Sende-Mediators lassen sich WS-Addressing-Informationen einer Nachricht nicht mehr verändern. Daher ist in dieser Implementierung ein eigener Send-Mediator realisiert, welcher genau dieses erlaubt. Zusätzlich muss daher nach dem Senden die Weiterverarbeitung an den internen Send-Mediator verhindert werden, weshalb am Ende jeder Eingangssequenz ein Drop-Mediator vorgesehen ist [Sch09].

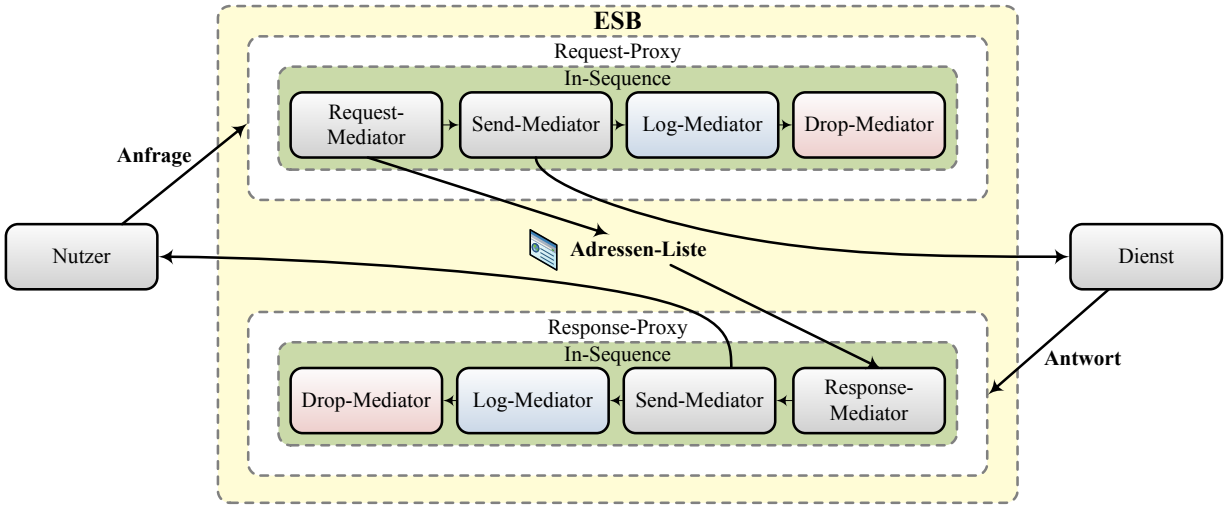


Abbildung 4.7.: Übersicht über die Anordnung der Mediatoren; nach [Sch09]

Weiterhin wird über eine entsprechende Klasse die Speicherung der ausgewechselten Adressdaten im Request-Mediator ermöglicht, welche dann im Response-Mediator wieder genutzt werden kann. Die jeweiligen Antwortadressen werden intern in Relation zur jeweiligen SOAP-`MessageID` gespeichert. Antwortet ein Dienst auf eine Nachricht an eine `ReplyTo`-Adresse, wird nach WS-Addressing ein `RelatesTo`-Header im Nachrichtenkopf der Antwort angegeben, welche die SOAP-`MessageID` der Anfrage enthält. Mit Unterstützung dieser Information ist der Response-Mediator in der Lage, die jeweils richtigen Zieladressen wieder einzusetzen und die Nachrichten an den eigentlichen Empfänger der Antwort zu versenden [Sch09].

Mit diesem Vorgehen kann die Ortstransparenz auch im Falle der asynchronen Kommunikation wieder hergestellt und zusätzlich auch jegliche Interaktion protokolliert werden, in dem ein entsprechender Mediator in die jeweiligen Eingangs- und Ausgangssequenzen des jeweiligen Proxies im Service-Bus platziert wird. Die Ortstransparenz wird aber nur für den Fall der Nutzung von WS-Addressing sichergestellt. Werden eigene Protokolle und damit Antwortadressen über eigene XML-Schemata definiert, müssen die entsprechenden Response- und Request-Mediatoren erweitert werden. Genauso können die Routing-Informationen bei vollständig verschlüsselten Nachrichten nicht angepasst werden. Die Me-

diatoren müssen also in der Lage sein, Nachrichten zu interpretieren und zu verarbeiten. Genauso müssen Sicherheitstechniken beachtet werden, welche durch digitale Signaturen implizit das Verändern von Nachrichten nicht erlauben, wenn die Signatur nicht mehr zu einer empfangenen SOAP-Nachricht passt. Trotzdem ist das Herstellen der Ortstransparenz wichtig, da ein ESB als Vermittler zwischen verschiedenen Architekturen und Standards auch sicherstellt, dass verschiedene Partner miteinander kommunizieren können. Beispielsweise kann im schlimmsten Fall trotz Angabe einer Antwortadresse ein Dienst eine Antwort nicht senden, da er nicht in der Lage ist, das Übertragungsprotokoll zu „sprechen“. Wird die Antwort durch den ESB geleitet, kann genau dieses sichergestellt werden.

Atomare Interaktionsprotokollierung: Wie im vorigen Abschnitt beschrieben, wird das Protokollieren von Nachrichten durch Weiterleiten der Nachrichten an einen Verifikationsdienst realisiert. Damit nichts protokolliert wird, was nicht den Empfänger erreicht hat, wird der Weiterleitungsmechanismus an den Monitor so lange verzögert, bis sichergestellt werden kann, dass die Nachricht beim eigentlichen Dienst angekommen ist. Die Verzögerung der Verifikation an dieser Stelle ist zudem notwendig, da im Synapse-ESB alle Nachrichten asynchron versendet werden. Wird zuerst die Nachricht an den Monitor gesendet, bevor die Nachricht beim eigentlichen Dienst angekommen ist, kann es zu Abweichungen zwischen realer Welt und dem Modell kommen, da der Monitor eine Nachricht als korrekt verifiziert, die noch nicht erfolgreich einem Empfänger zugestellt werden konnte. Für die Implementierung heißt dies, dass die Mediatoren im Unterschied zur normalen Arbeitsweise im ESB blockieren müssen, bevor bestimmte Aktivitäten abgeschlossen werden [Med10].

Der Synapse-ESB bietet diese Option des blockierenden Versandes von Nachrichten nicht direkt an, daher muss Synapse an dieser Stelle erweitert werden. Für eine Erweiterung von Synapse gibt es drei verschiedene Möglichkeiten: Die *Implementierung einer eigenen Proxy-Klasse*, welche das Senden von Nachrichten blockierend ermöglicht, die *vorhandene Proxy-Klasse erweitern* oder einen *Mediator* entwickeln, der das Senden und Empfangen von Nachrichten übernimmt. Bei den ersten beiden Möglichkeiten wird tief in die Architektur von Synapse eingegriffen, um eigene Proxy-Klassen zu realisieren. Bei der mediatorbasierten Lösung ist dies nicht der Fall, weshalb diese Lösung in dieser Arbeit favorisiert wird. Wichtig ist dabei, dass Synapse selbst nicht in der Lage ist, Nachrichten blockierend zu versenden. Allerdings nutzt Synapse für die Nachrichtenübertragung das Rahmenwerk Apache Axis 2, welches verschiedene Schnittstellen für den Versand von Nachrichten anbietet[3]. Das Rahmenwerk bietet dafür die Klasse `OperationClient` an, welche den blockierenden Versand von Nachrichten an Web-Services unterstützt. Der `OperationClient` wartet dabei auf bestimmte Bestätigungsnachrichten der jeweiligen Übertragungsprotokolle wie TCP, bevor die Kontrolle nach dem Versand an die aufrufende Klasse zurückgegeben wird. Diese Klasse kann auch zur Implementierung eines Mediators genutzt werden.

[3]Siehe auch `http://axis.apache.org/axis2/java/core/`.

Die Implementierung ist wie in Abbildung 4.8 realisiert. Dem Log-Mediator wird dafür ein Mediator vorgeschaltet, welcher das blockierende Senden mithilfe des beschriebenen `OperationClients` übernimmt. Damit der interne Send-Mediator von Synapse die Nachricht nicht am Ende der `In-Sequence` erneut versendet, wird am Ende der Sequenz wieder der Drop-Mediator benutzt, um die Weiterverarbeitung der Nachricht durch Synapse zu beenden. Der `OperationClient` unterstützt synchrone Nachrichtenübertragung, bei der eine Antwort auf dem bereits geöffneten Kanal zurückgesendet wird. Geschieht dies, muss über den `OperationClient` die Antwort in die entsprechende `Out-Sequence` von Synapse injiziert werden, bevor sie dann wieder vom Log-Mediator protokolliert werden kann. In der `Out-Sequence` ist kein Drop-Mediator notwendig, da hier nicht wie bei der `In-Sequence` implizit ein interner Send-Mediator aufgerufen wird, sondern nur über den bereits offenen Kanal das Resultat der Mediatoren der `Out-Sequence` zurückgegeben wird. Bei asynchroner Verarbeitung wird natürlich nur die `In-Sequence` benötigt [Med10].

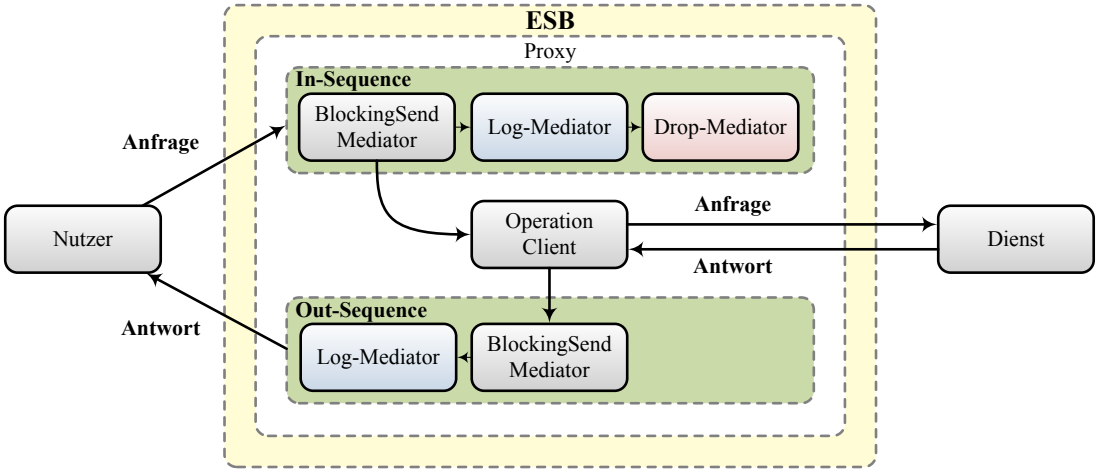


Abbildung 4.8.: Übersicht Mediatoren bei blockierendem Senden; nach [Med10]

Fehlerquellen bezüglich der Nachrichtenübertragung gibt es unabhängig von Entwicklungsfehlern zur Laufzeit nur zwischen Dienstnutzer und Bus sowie zwischen dem Bus und dem eigentlichen Dienst. Gibt es Übertragungsfehler zwischen Nutzer und Bus, so werden diese zwischen Nutzer und Bus durch das Übertragungsprotokoll über Mechanismen wie `Ack`-Nachrichten korrigiert. Kann diese Nachricht nicht über den Rückkanal gesendet werden, wird die Weiterverarbeitung im Bus abgebrochen. Gibt es einen Timeout beim Warten auf eine Bestätigung vom Bus, sendet der Nutzer die Nachricht erneut. An dieser Stelle ist bisher noch nichts über den Log-Mediator protokolliert worden und Modell und Laufzeitverhalten stimmen überein. Auch zwischen dem Bus und dem Dienstanbieter mit der Kommunikation über den `OperationClient` geschieht das Gleiche. Wird die TCP-Übertragung bestätigt, wird eine Nachricht als übertragen angesehen und kann protokolliert werden. Problematisch wird es allerdings, wenn der Übertragungskanal beispielsweise durch einen Absturz des Dienstes unterbrochen wird und bei synchroner Nachrichtenübertragung eine Antwort über dieselbe TCP-Verbindung nicht mehr vom Dienst an den Bus

zurückgesendet werden kann. Hier kommt es auf die Implementierung des Dienstnutzers und seiner Reaktion an. Nutzt er Forward-Recovery und sendet die Nachricht ein zweites Mal, kommt es zu einer Dopplung der Nachricht im Protokoll und je nach Choreographie-Modell wird bei der Verifikation jetzt ein Fehler erkannt. Nutzt der Dienstnutzer Backward-Recovery, wird der Nichterhalt einer Antwort je nach Choreographie-Modell möglicherweise zu einem Abbruch der Choreographie führen. Bei dieser Diskussion wird deutlich, dass das Choreographie-Modell entsprechende Recovery-Mechanismen wie erneutes Senden im Fehlerfall bei Forward-Recovery beschreiben muss, damit eine Nachrichtendopplung bei der Verifikation nicht zu einem Fehler führt.

4.1.2. Korrelation zwischen Interaktion und Model

Eine weitere wichtige Anforderung aus Kapitel 3.1.3.1 bezüglich der Modelldimension ist die Korrelation der Nachrichten auf das Choreographiemodell. Um die übertragenen Nachrichten zum Modell in Beziehung zu setzen, müssen als Erstes die Daten identifiziert werden, welche für die Korrelation notwendig sind. Für WS-CDL lassen sich die Daten wie folgt identifizieren:

- **Rollen**: Choreographie-Beschreibungssprachen wie WS-CDL beschreiben für jede Interaktion, welche Rollen (Sender und Empfänger) der Teilnehmer an einer Interaktion beteiligt sind. WS-CDL definiert dafür als Beschreibung des `Interaction`-Artefakts die zwei Elemente `fromRoleTypeRef` und `toRoleTypeRef`, welche genau diese beiden Rollen identifizieren. Um die Sender- und Empfängerrolle einer Nachricht zu identifizieren, lassen sich mehrere Techniken einsetzen. Beispielsweise kann die IP-Adresse des Senders oder die aufgerufene URL eines Web-Services Aufschluss darüber geben, von welchem Teilnehmer eine Nachricht an einen anderen Teilnehmer gesendet wurde. Diese Methode ist allerdings wenig zuverlässig, da beides nicht unbedingt eindeutig innerhalb einer Choreographie sein muss. Eine weitaus einfachere Möglichkeit zur Korrelation ist eine Kennzeichnung innerhalb des Nachrichtenkopfes einer ausgetauschten Nachricht.

- **Nachrichtentyp**: Um eine Nachricht zum Modell in Beziehung zu setzen, muss der Nachrichtentyp erkannt werden. Dafür wird auf der einen Seite in WS-CDL als Attribut des `Exchange`-Elements jeweils der `informationType` jedes Datenaustausches definiert. Der `informationType` in WS-CDL verweist auf eine XML-Schema-Definition einer WSDL-Beschreibung für den jeweiligen Zieldienst. Auf der anderen Seite gibt WS-CDL die Möglichkeit vor, für jeden Austausch einen Kanal (`channelType`) anzugeben, für den ein bestimmter `informationType` vorgegeben wird. Um zu erkennen, ob eine Nachricht von einem speziellen Typ ist, muss also der Aufbau der Nachricht mit dem XML-Schema der Nachrichtendefinition abgeglichen werden; die Korrelationsdaten sind also durch die Beschreibung der Nachrichten vorhanden.

- **Instanzkennung**: Zusätzlich zu den Rollen und Nachrichtentypen wird eine Instanzkennung benötigt, damit eine Nachricht einer Choreographie-Instanz zugeordnet und beispielsweise Fehler wie eine Dopplung von Nachrichten erkannt werden kann. Wie eine solche Instanzkennung generiert und benutzt werden kann, ist nicht als Teil von WS-CDL beschrieben. Zum Beispiel kann WS-Coordination (siehe [FJ09]) zur Generierung von gemeinsamen Aktivitäten und der Instanzkennung genutzt werden. Einen Standardweg gibt es hierfür nicht, allerdings bleibt als einfacher Weg wieder das Kennzeichnen der Instanz innerhalb eines Nachrichtenkopfes einer SOAP-Nachricht.
- **Choreographie-Typ**: Soll mehr als ein Choreographie-Typ zur Laufzeit verifiziert werden, muss eine Nachricht auch zu ihrem jeweiligen Choreographie-Typ in Beziehung gesetzt werden können. Zum einen kann dafür wieder der Nachrichtentyp herangezogen werden, allerdings ist auch hier die Eindeutigkeit nicht gewährleistet. Eine einfache Möglichkeit ist auch hier ein optionaler Nachrichtenkopf, welcher die Information über den Choreographie-Typen in jeder SOAP-Nachricht beschreibt.

Um diese Anforderungen für die Korrelation zu gewährleisten, müssen in jeder Nachricht spezielle SOAP-Kopfzeilen vorhanden sein. Der Quellcode 4.1 zeigt beispielhaft den Nachrichtenkopf einer SOAP-Nachricht, welcher die entsprechenden Daten enthält.

Quellcode 4.1: SOAP Header-Informationen

```
<envelope ...>
 <header ...> ...
  <vsis:chormon
    xmlns:vsis="http://vsis.de/monitoring">
   <fromRole>CNAME</fromRole>
   <toRole>CNAME</toRole>
   <chorIdentifier>String</chorIdentifier>
   <chorType>String</chorType>
  </vsis:chormon>
 </header> ...
</envelope>
```

Eine weitere Voraussetzung für die Korrelation ist auch die Eindeutigkeit der verwendeten Namen und Nachrichtentypen in der WS-CDL-Beschreibung. Wird via WS-CDL beispielsweise eine Bestätigungsnachricht mit dem gleichen Inhalt und dem gleichen Aufbau an verschiedenen Stellen verwendet, dann kann später nicht genau zugeordnet werden, in welchem Kontext diese Bestätigungsnachricht jetzt steht. Es kann anhand des Kontextes zwar vermutet werden, zu welchem Kontext eine Nachricht gehört, aber bei vielen parallelen Aktivitäten ist die Zuordnung nicht mehr eindeutig. Im Modell muss dafür jede Nachricht eindeutig ihrem Kontext zugeordnet werden können, weshalb jede in WS-CDL definierte Nachricht einen eindeutigen und nicht mehrfach verwendeten Typ haben muss.

4.1.3. Überwachung und Verifikation von Sensordaten

Sind Sensoren so implementiert, dass sie konsistente Daten für die Verifikation liefern, und sind entsprechende Daten in den Nachrichten vorhanden, die eine Korrelation der Nachrichten auf Choreographie-Instanzen ermöglichen, so fehlt noch ein Monitor, der die Sensordaten verarbeiten und verifizieren kann. Dafür sind verschiedene Schnittstellen eines Monitors zu definieren: Zum einen benötigt er die Sensordaten für die Verifikation und zum anderen benötigt er das entsprechende Choreographie-Modell und die Integritätsbedingungen, um ein internes formales Modell zur Verifikation einer Choreographie zu erstellen. Eine entsprechende Komponente im Monitor kann dann den Soll/Ist-Vergleich mithilfe des internen Modells und der Sensordaten durchführen.

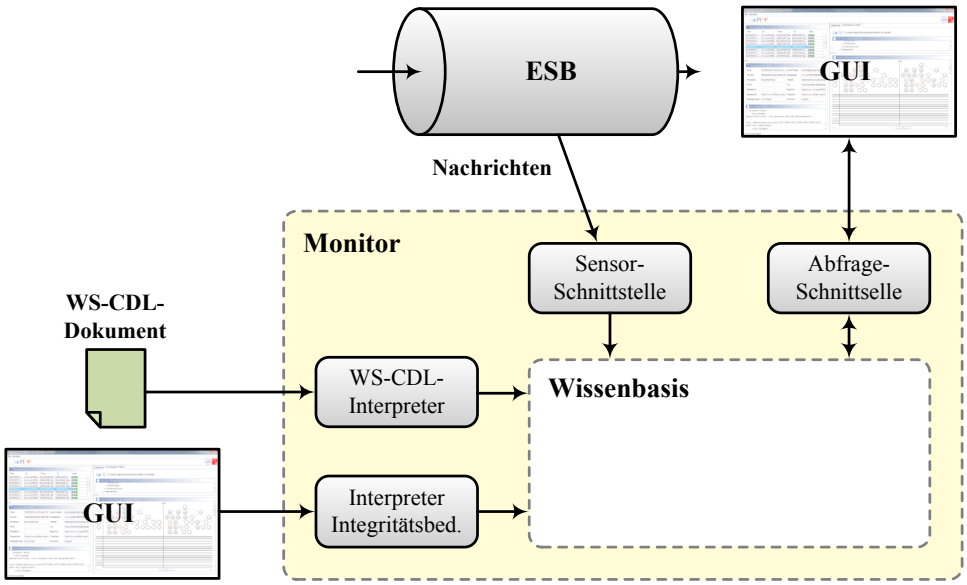


Abbildung 4.9.: Architekturübersicht für den Monitor (in Anlehnung an [Voi10])

Die Komponenten und deren Zusammenspiel für eine Monitor-Architektur ergeben sich daher direkt, wie in Abbildung 4.9 zu sehen. Die Sensordaten werden über eine Schnittstelle an den Monitor übergeben, welche dann in eine für den Soll/Ist-Vergleich interne Repräsentation überführt werden. Damit der Soll/Ist-Vergleich arbeiten kann, muss das Choreographie-Modell in Form von WS-CDL durch einen Interpreter in eine interne Form überführt werden, genauso wie die inhaltlichen Integritätsbedingungen, die zusätzlich zum WS-CDL-Dokument formuliert worden sind. Dies können beispielsweise instanzübergreifende Bedingungen oder auch andere non-funktionale Bedingungen sein, für die WS-CDL keine Modellierungsartefakte vorsieht. Das Resultat des Soll/Ist-Vergleichs muss über eine Abfrageschnittstelle anderen Werkzeugen zur Verfügung gestellt werden, um beispielsweise den Zustand der Choreographie zu visualisieren. Der Soll/Ist-Vergleich erfolgt wie in Abschnitt 3.1.2.5 beschrieben auf Basis des Ereigniskalküls und kann damit direkt durch einen Prolog-Interpreter durchgeführt werden. Im Zentrum der Architektur steht also eine Wissensbasis, welche mithilfe von Prolog verarbeitet wird.

Sensor-Schnittstelle: Für die Architektur sind in den letzten Abschnitten Sensoren realisiert worden, welche ereignisorientiert Nachrichten an einen Monitor weiterleiten. Die Sensorschnittstelle am Monitor ist für den Empfang der Daten und für die Überführung der Nachrichten in ein Format, welches in Prolog für Nachrichtenfakten genutzt werden kann. Zusätzlich muss die Übertragung zwischen Sensoren und Sensor-Schnittstelle zum einen stabil sein und zum anderen die Reihenfolge der Ereignisse erhalten.

Die Reihenfolge der Nachrichten spielt für die Arbeitsweise der Sensorschnittstelle eine wichtige Rolle. Werden Nachrichten chronologisch in die Faktenbasis eingefügt, brauchen beim Caching im Ereigniskalkül bereits abgeschlossene und berechnete Gültigkeitsperioden von Fluents nicht erneut berechnet werden. Da die Sensoren allerdings verteilt sind und Nachrichten im ESB nicht-blockierend versendet werden, kann es sein, dass Nachrichten nicht in der Reihenfolge ihrer Zeitstempel an der Sensorschnittstelle ankommen. Als einfache Möglichkeit wird hier ein Puffer implementiert, der die Nachrichten erst nach einer gewissen Zeit intern weiterleitet. Die Weiterleitung erfolgt sortiert, wenn angenommen werden kann, dass keine weiteren Nachrichten mehr von den Sensoren kommen, die zeitlich vor den ältesten Nachrichten im Puffer liegen. Ein Zeitintervall von etwa einer Sekunde hat sich dabei als praktikabel erwiesen, um die Neuberechnungen durch den Cached-EC (siehe Abschnitt 3.1.2.3) zu vermeiden.

Damit Nachrichten nach Zeitstempeln im Puffer sortiert werden können, müssen sie mit Zeitstempeln versehen werden, die ihren Auftrittszeitpunkt festhalten. Dies wird entsprechend von den Sensoren übernommen, welche ihre internen Uhren über das NTP-Protokoll synchronisieren. Zwar kann es bei der Synchronisierung trotzdem zu leichten Abweichungen der Zeitstempel kommen, allerdings sind Aktivitäten in Choreographien eher als langlaufend anzusehen, sodass sehr kleine Abweichungen der internen Uhren der Sensoren keine großen Auswirkungen auf die Reihenfolge der Nachrichten haben.

Weiterhin muss die Übertragung von den Sensoren zuverlässig sein und jede Nachricht nur einmal an die Sensor-Schnittstelle übertragen werden. Die Übertragung zwischen Sensoren und Monitor wird daher durch die WS-Reliable-Messaging-Spezifikation gesichert, welche bestimmte Übertragungsgarantien auf Anwendungsebene gibt.

Abfrageschnittstelle: Die Abfrageschnittstelle wird benutzt, um den Zustand einer Choreographie für andere Werkzeuge sichtbar zu machen. Die Abfrageschnittstelle kann dabei navigierend auf das Protokoll der Ereignisse zugreifen und die Einhaltung der Choreographie-Integritätsbedingungen nach jeder Nachricht ausgeben lassen. Die entsprechenden Ergebnisse lassen sich für Visualisierungen und auch für die Fehlerbehebung in Abschnitt 5.2 entsprechend nutzen.

Interpreter für Integritätsbedingungen: Der Interpreter für Integritätsbedingungen ist in zwei Schnittstellen unterteilt. Er besteht zum einen aus einem WS-CDL-Interpreter und zum

anderen einem Interpreter für Integritätsbedingungen, die sich durch WS-CDL nicht darstellen lassen. Die WS-CDL-Dokumente müssen dabei die Voraussetzung erfüllen, dass die Eindeutigkeit der verwendeten Namen und Nachrichtentypen in der WS-CDL-Beschreibung gewährleistet ist. Der Interpreter hat die Aufgabe, ein WS-CDL-Dokument in eine formale Repräsentation zu überführen. Er hat dabei nicht die Aufgabe das Choreographie-Modell statisch zu verifizieren. Daher wird in dieser Arbeit davon ausgegangen, dass ein Choreographie-Modell schon bei der Modellierung verifiziert worden ist und Eigenschaften, wie beispielsweise die Verklemmungsfreiheit oder auch die verteilte Ausführbarkeit des Modells, gewährleistet sind. Eine Laufzeitverifikation von Choreographien kann zwar auch mit fehlerhaften Prozessmodellen durchgeführt werden, führt aber nicht unbedingt zu nutzbaren Ergebnissen.

Die entsprechenden Interpreterschnittstellen sind *Konfigurationsschnittstellen*, welche vor der eigentlichen Laufzeit genutzt werden, um den Monitor mit Integritätsbedingungen zu füttern. Im Gegensatz dazu sind Sensor- und Abfrageschnittstellen reine *Laufzeitschnittstellen*, welche bei der Ausführung von Choreographien genutzt werden.

Wissensbasis: Die Wissensbasis und dazugehörige Anfragemechanismen sind der zentrale Baustein der Laufzeitverifikation. Sie liefert die Information darüber, ob bestimmte Integritätsbedingungen nach dem Einfügen neuer Ereignisfakten eingehalten sind oder nicht. Zentrales Element für die Implementierung ist ein Prolog-Interpreter, der entsprechende Regeln interpretieren kann. Die verwendete Prolog-Engine ist Eclipse-CLP[4], welche eine schnelle Anbindung an Java ermöglicht. Die Monitoring-Infrastruktur ist bis auf den Teil der Wissensbasis in Java implementiert. Es gibt zwar auch Ansätze wie etwa das Projekt Mandarax [Die05], um Wissensbasen direkt mithilfe von Java zu verwalten, allerdings hat sich Eclipse-CLP mit seiner Java-Anbindung im Gegensatz zu Mandarax im Laufzeitverhalten als deutlich schneller erwiesen. Eclipse-CLP wird daher in dieser Arbeit für die Laufzeitverifikation in Verbindung mit Java für die Web-Services-Infrastruktur eingesetzt.

Mit Unterstützung dieser Monitoring-Infrastruktur wird jetzt in den nächsten Abschnitten der Baustein der Wissensbasis weiter verfeinert und implementiert.

4.2. Regelbasierte Laufzeitverifikation

In Abschnitt 3.1.2.5 ist das Ereigniskalkül als Kandidat zur formalen Beschreibung von Choreographien und zur Laufzeitverifikation ausgewählt worden. Bevor aber ein Modell auf Basis des Ereigniskalküls eingeführt wird, wird an dieser Stelle eine einfache regelbasierte Verifikation definiert. Zum einen können mithilfe einer einfachen Lösung auf Basis von Prädikatenlogik schon Konzepte und Algorithmen für das Ereigniskalkül eingeführt werden und zum anderen dient die im Folgenden vorgestellte Lösung als Vergleichswert für die

[4]Siehe `http://www.eclipse-clp.org`.

spätere Evaluation des Ansatzes auf Basis des Ereigniskalküls. Darüber lässt sich beispielsweise der Mehraufwand beim Schlussfolgern durch Einführung der quantifizierten Zeiten im Ereigniskalkül abschätzen.

Die Laufzeitverifikation benutzt, wie in Abschnitt 2.1.3 diskutiert, WS-CDL als Basis für die Beschreibung von Choreographien. Damit WS-CDL als Basis genutzt werden kann, muss im ersten Schritt ein WS-CDL-Dokument in eine formale Beschreibung überführt werden. Die generellen Verfahrensweisen dafür werden im nächsten Abschnitt eingeführt, um eine einfache Verifikation des Kontrollflusses einer Choreographie zur Laufzeit durchzuführen.

4.2.1. Abbildung von WS-CDL in Regeln

Um Interaktionen zur Laufzeit zu verifizieren, kann die Baumstruktur von WS-CDL zur Verifikation ausgenutzt werden. Abbildung 4.10 zeigt eine einfache Sequenz, welche zwei Interaktionen enthält. Für die Verifikation werden dabei zwei Zustände für jeden Knoten vorgesehen: Entweder ist ein Knoten bei der Verifikation erfolgreich oder nicht. Jeder einzelne Knoten im Baum kann dabei aufgrund seiner direkt unter ihm liegenden Knoten entscheiden, ob eine Nachricht valide ist oder nicht. Bei einem Blattknoten muss dafür beispielsweise alles überprüft werden, was in Abschnitt 4.1.2 zur Korrelation zwischen Nachrichten und Modell definiert wurde, also `fromRole`, `toRole` und `messageType`. Stimmen alle Daten eines Datenaustausches mit den Daten des Modells überein, ist die Nachricht valide. Bei Knoten innerhalb des Baumes (also den ordnenden Strukturen wie Sequenzen) muss die Entscheidung zur Validität einer Nachricht an den jeweiligen aktiven Unterknoten delegiert werden. Um zu entscheiden, ob eine Nachricht bezüglich der gesamten Choreographie valide ist, muss also nur der Wurzelknoten des WS-CDL Baumes befragt werden, der die Frage an die entsprechenden Unterknoten delegiert. Wie das Delegieren der Frage an die Unterknoten geschieht, ist für jede Aktivitätsart (Sequenz, Choice, Parallel oder WorkUnit) verschieden.

Um die Idee zu illustrieren, wird der Baum aus Abbildung 4.10 mithilfe von validen Bei-

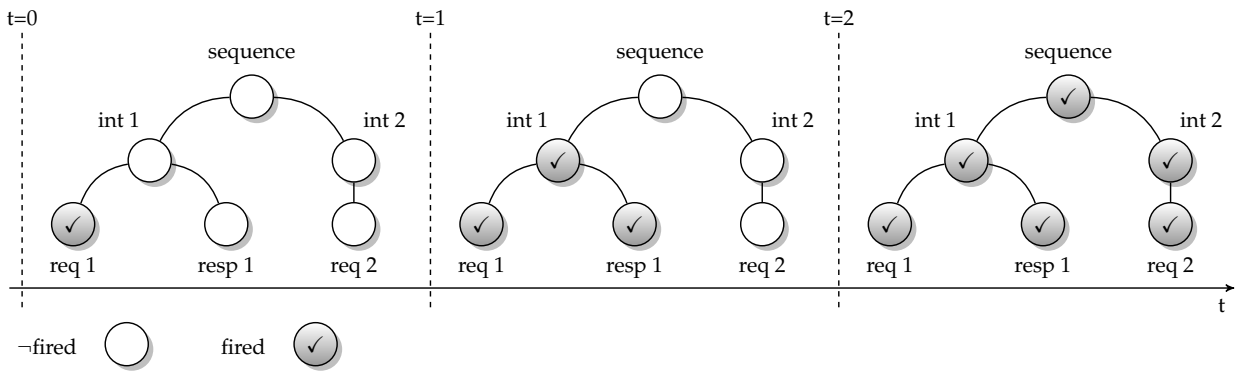


Abbildung 4.10.: Baumdarstellung einer Beispielsequenz

spielnachrichten durchlaufen. Wird die erste Nachricht beim Monitor aufgegriffen, wird der Wurzelknoten gefragt, ob die Nachricht valide ist oder nicht. Der Wurzelknoten ist in diesem Falle eine Sequenz und ist noch im Status 'nicht gefeuert'. Da eine Sequenz nicht selbst entscheiden kann, ob eine Nachricht valide ist, wird diese Frage an den ersten aktiven Unterknoten delegiert, in diesem Falle an eine Interaktion. Auch in diesem Falle kann der Interaktionsknoten im Baum nicht entscheiden, ob die Nachricht valide ist, und delegiert die Frage weiter an den ersten Unterknoten; in diesem Fall an einen `Request`-Knoten. Dieser Knoten ist allerdings ein Blattknoten und kann entscheiden, ob die Nachricht der Erwarteten entspricht. Dafür werden alle Korrelationsdaten der Nachricht verglichen und im Erfolgsfall wird der Knoten auf den Status *gefeuert* gesetzt und die erfolgreiche Antwort an den Vaterknoten weitergeleitet. Dieser entscheidet jetzt, ob alle Unterknoten erfolgreich beendet sind. Sind alle Unterknoten erfolgreich beendet, wird auch der Vaterknoten auch auf *gefeuert* gesetzt. Die erfolgreiche Antwort des Unterknotens wird jetzt weiter zum nächsten Vaterknoten weitergeleitet. Der Wurzelknoten hat jetzt die Antwort auf die Frage durch die Unterknoten beantwortet bekommen und die Nachricht ist valide. Am Ende ist eine Choreographie erfolgreich beendet, wenn der Wurzelknoten auf *gefeuert* gesetzt ist. Zur Verifikation wird also der Baum aus WS-CDL-Aktivitäten entsprechend traversiert.

Um diesen Algorithmus umzusetzen, lassen sich Regeln ausdrücken, die das Verhalten von WS-CDL modellieren. Diese Regeln lassen sich in zwei Teilbereiche kategorisieren, zum einen Regeln für Basisaktivitäten in WS-CDL (den Blattknoten) und zum anderen Regeln für zusammengesetzte komplexe Aktivitäten, welche die Knoten innerhalb des WS-CDL-Baumes beschreiben. Das Verhalten der Regeln für einen Knotentyp im WS-CDL-Baum ist dabei immer gleich; beispielsweise muss eine Sequenz immer in der angegebenen Reihenfolge durchlaufen werden, damit der Sequenzknoten als *gefeuert* angesehen werden kann. Daher lassen sich für WS-CDL entsprechende Axiome für jeden Knotentyp aufstellen. Allerdings ist die Struktur jeder auf WS-CDL basierenden Choreographie immer verschieden. Daher müssen die Axiome die Struktur einer Choreographie berücksichtigen, wenn sie die Frage der Validität einer Nachricht an entsprechende Unterknoten delegieren. Eine Wissensbasis zur Verifikation von WS-CDL-basierten Choreographien besteht daher aus zwei wesentlichen Komponenten: den Fakten über die Struktur der Choreographie und den Axiomen der einzelnen WS-CDL-Aktivitäten, wie in Abbildung 4.11 illustriert.

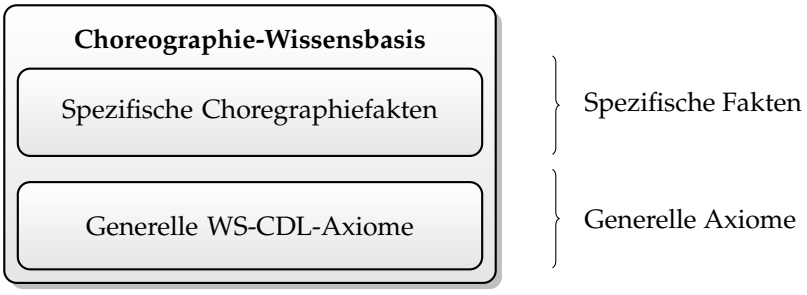


Abbildung 4.11.: Grundlegende Blöcke der WS-CDL-Regeln

4.2.1.1. Fakten über die Choreographie-Struktur

Der erste Teil zur Beschreibung von Regeln ist die Beschreibung der Struktur einer Choreographie. Eine Choreographie besteht dabei aus Basisaktivitäten und komplexen Aktivitäten. Laut WS-CDL-Spezifikation sind die Aktivitäten `Interaction`, `SilentAction`, `NoAction` und `Assign` als Basisaktivitäten benannt. `Interaction` kann allerdings als komplexe Aktivität angesehen werden, da sie aus einer Sequenz von Anfrage und Antwort bestehen kann. Die Interaktion wird an dieser Stelle daher noch weiter unterteilt. Die Basisaktivität der Interaktion ist ein einfacher Nachrichtenaustausch in eine Richtung, was einem `Exchange` in WS-CDL entspricht. Damit Regeln für die komplexen Aktivitäten später den Typ eines Knotens erkennen können, werden die Prädikate $IsExchange(a)$, $IsSilentAction(a)$, $IsNoAction(a)$ und $IsAssign(a)$ eingeführt. Zudem werden die einzelnen Basisaktivitäten weiter präzisiert. Eine einfache Nachricht, welche von einem Web-Service oder einer Anwendung an einen anderen Web-Service gesendet wird, ist hauptsächlich durch die Rollen der Teilnehmer und den Namen der Nachricht charakterisiert, wie in Prädikat 4.1 spezifiziert.

$$Exchange(name, fromRole, toRole) \tag{4.1}$$

Das Prädikat $NoAction$ oder $SilentAction$ definiert ein nicht beobachtbares Ereignis, welches durch Name und Rolle (Prädikat 4.3 und 4.2) charakterisiert wird.

$$NoAction(name, role) \tag{4.2}$$
$$SilentAction(name, role) \tag{4.3}$$

Die letzte fehlende Basisaktivität ist die `Assign`-Aktivität, welche die Zuweisung von Variablen innerhalb einer Choreographie definiert. Variablen in der WS-CDL-Beschreibung werden zur Definition von Bedingungen oder zur Definition von Schleifen genutzt. Die `Assign`-Aktivität ist dabei an eine Rolle in der Choreographie gebunden und besteht aus einem oder mehreren benannten `Copy`-Elementen. Diese Elemente definieren eine Variablenzuweisung, bestehend zum einen aus der Quelle (`Source`) des Variablenwertes und zum anderen aus dem Ziel (`Target`) der Zuweisung. Nur der Rolleninhaber aus der `Assign`-Aktivität ist in der Lage, die Werte der Variable zu verändern. Für jedes `Copy`-Element einer `Assign`-Aktivität wird ein Choreographie-Fakt zur Wissensbasis hinzugefügt, wie es in Prädikat 4.4 definiert ist. Dabei definieren die Variablen *cName* und *rName* den Namen des `Copy`-Elementes und die Rolle, welche Zugriff auf die Variable hat. Das Prädikat *getVar* ist äquivalent zur WS-CDL-Funktion `getVariable()` (siehe auch [BLF+05]) und definiert eine Variable *vname*. $Expression$ definiert einen beliebigen XPath-Ausdruck, der entweder einen Wert oder einen Ort definiert, wo der Wert der Variablen gefunden werden kann.

$$Assign(cName, rName, GetVar(vname, rName), Expression(xPath)) \tag{4.4}$$

Zur Beschreibung von komplexen Aktivitäten werden zwei Prädikatstypen benötigt:

- Der *Aktivitätstyp*: Die Prädikate *Sequence(name)*, *Parallel(name)* und *Choice(name)* beschreiben den Typ einer komplexen Aktivität, wobei der *name* der Variable für einen Choreographie-Typen eindeutig ist.
- Die *Teilaktivitäten*: Das Prädikat *MemberActivity(name,[member])* beschreibt die Teilaktivitäten einer komplexen zusammengesetzten Aktivität mit dem Namen *name*. Teilaktivitäten einer komplexen Aktivität können zum einen komplexe Aktivitäten oder auch Basisaktivitäten sein.

Das wichtigste Artefakt in der WS-CDL-Beschreibung ist die `Interaction`. In Bezug auf die Regeln wird eine Interaktion als komplexe Aktivität betrachtet, da eine Interaktion entweder aus einer Anfrage, einer Antwort oder beidem besteht, welche über `Exchange`-Elemente definiert werden. Bei der Definition in WS-CDL können bei einer Interaktion auch mehrere Antworten angegeben werden, wobei zur Laufzeit nur eine dieser Antworten ausgeführt werden darf. Eine Interaktion besteht daher also entweder aus einer einschrittigen Sequenz oder einer Sequenz aus zwei Schritten. Daher wird eine Interaktion genauso wie eine Sequenz abgebildet und die `Exchange`-Elemente als Teilaktivitäten über das Prädikat *MemberActivity* definiert. Ist allerdings bei einer Interaktion mehr als nur eine Antwort definiert, kann zur Laufzeit eine Antwort aus den gegebenen Möglichkeiten ausgewählt werden. In diesem Fall wird zusätzlich für die Antwort durch eine *Choice* definiert, welche `Exchange`-Elemente als Antworten möglich sind.

Eine weitere wichtige zusammengesetzte Aktivität ist die *WorkUnit*, welche mithilfe der Fakten $WorkUnit(name)$ und $MemberActivities(wuName, [member])$ beschrieben wird. Zusätzlich zu den bekannten komplexen Aktivitäten wie *Sequence* besitzt eine WorkUnit eine Ausführungsbedingung, eine Nebenbedingung in Bezug auf blockierendes Verhalten und die Möglichkeit, eine Schleifenbedingung zu definieren. Diese zusätzlichen Eigenschaften werden mit zusätzlichen Prädikaten beschrieben:

- Die *Blockbedingung* beschreibt, ob eine WorkUnit blockierend ist oder nicht. Blockierend bedeutet in diesem Fall, dass die Ausführung so lange blockiert wird, bis die Ausführungsbedingung wahr wird. Für dieses Verhalten wird der Fakt $Block(wuName)$ in die Wissensbasis eingefügt. Ist eine WorkUnit nicht blockierend, wird die WorkUnit bei Ausführung einfach übersprungen, wenn die Ausführungsbedingung nicht wahr ist.
- Die *Ausführungsbedingung* definiert, ob eine WorkUnit ausgeführt wird oder nicht. Dieses Verhalten wird durch den Fakt *GuardExpression(wuName,vName,op,value)* in der Wissensbasis definiert. Die drei Variablen $vName$, op und $value$ definieren dabei, welchen $value$ die Variable $vName$ haben muss (op ist hierbei ein einfacher binärer Operator wie beispielsweise $=$, $<$ oder $>$). Ist eine WorkUnit ohne Bedingung blockierend,

kann dies durch eine spezielle *GuardExpression* ausgedrückt werden, nämlich durch den Fakt *GuardExpression(wuName,_,_,true)*.

- Die *Schleifenbedingung* definiert, ob eine WorkUnit wiederholt werden kann oder nicht. Dieses wird durch das Prädikat *RepeatExpression* beschrieben, welches in der gleichen Wiese wie *GuardExpression* funktioniert.

Mit Unterstützung der in diesem Kapitel vorgestellten Prädikate lässt sich die Struktur einer Choreographie vollständig abbilden. Diese Struktur und die vorgestellten Prädikate werden jetzt im nächsten Schritt benutzt, um Regeln für das Verhalten der Aktivitäten abzuleiten.

4.2.1.2. WS-CDL-Axiome

Es ist klar, dass für jeden Knotentyp das Verhalten zur Delegation der Frage oder zur Beantwortung der Validitätsfrage verschieden ist. Im Falle einer Sequenz wird die Frage der Validität immer an den ersten Unterknoten delegiert, welcher nicht im Status *gefeuert* steht. Im Falle einer Auswahl müssen alle Unterknoten gefragt werden, ob genau einer der Unterknoten eine valide Antwort zurückliefert. Beim Parallelknoten muss einer der nicht gefeuerten Unterknoten erfolgreich sein, damit eine Nachricht als valide klassifiziert wird. Bei der WorkUnit ist der Algorithmus noch ein wenig komplizierter, da eine WorkUnit eine Schleife definieren kann. Im Falle einer Schleife müssen alle Unterknoten bei einem erneuten Schleifendurchlauf zurückgesetzt werden. Nur die Basisaktivitäten entscheiden direkt, ob eine Nachricht valide ist oder nicht. Sie stellen damit die Blattknoten im WS-CDL-Baum dar.

Basisaktivitäten: Ein wichtiger Bestandteil der regelbasierten Verifikation ist das Prädikat $Fired(id, node)$, welches definiert, ob ein Knoten *node* für eine Choreographie-Instanz *id* gefeuert hat oder nicht. Mit Unterstützung dieses Prädikats lassen sich schnell Regeln für die Basisaktivitäten definieren. Ein Nachrichtenaustausch ist für eine Choreographie-Instanz valide, wenn der Knoten in dieser Instanz vorher noch nicht gefeuert hat und die Nachricht zum Knoten passt. Ist das der Fall, dann wird der Knoten für die Instanz mithilfe von *Assert* auf *gefeuert* gesetzt, wie es auch in Axiom 4.5 definiert ist.

$$\begin{aligned} &IsValid(id, message, node) \leftarrow \\ &\quad IsExchange(node) \wedge \neg Fired(id, node) \wedge node = message \wedge \\ &\quad Assert(Fired(id, node)) \end{aligned} \tag{4.5}$$

Bei `SilentAction` und `NoAction` lassen sich die Regeln entsprechend formulieren. Allerdings gibt es bei diesen Aktivitäten eine Besonderheit, da sie zu den nicht beobachtbaren

Aktivitäten gehören. Wird beim Traversieren des Baumes eine dieser beiden Aktivitäten erreicht, dann feuert der Knoten sofort. Da allerdings kein Nachrichtenknoten erreicht wird, muss der Baum erneut traversiert werden, damit ein Nachrichtenknoten die Validität prüfen kann. Der Vorteil bei diesem Vorgehen ist einfach ersichtlich: Enthält beispielsweise eine Sequenz eine `SilentAction`, dann wird diese Aktivität beim Traversieren des Baumes erreicht und auf *gefeuert* [5] gesetzt. Die Sequenz kann dann mit der nächsten Aktivität fortfahren. An dieser Stelle wird also angenommen, dass wenn eine der nachfolgenden Aktivität ausgeführt wird, dann ist auch die `SilentAction` ausgeführt worden. Das erneute Traversieren des Baumes wird erreicht, in dem ein Fakt hinzugefügt wird, welcher signalisiert, dass eine `SilentAction` oder eine `NoAction` beim letzten Baumdurchlauf aktiviert wurde. Der Wurzelknoten der Choreographie entscheidet dann bei Vorliegen dieses Faktes, ob der Baum erneut traversiert werden muss. Die Axiome 4.6 und 4.7 drücken diesen Umstand aus.

$$
\begin{aligned}
&IsValid(id, \_, node) \leftarrow \\
&\quad IsSilentAction(node) \wedge \neg Fired(id, node) \wedge \\
&\quad Assert(Fired(id, node)) \wedge Assert(SAFired(id)) \qquad (4.6)\\
&IsValid(id, \_, node) \leftarrow \\
&\quad IsNoAction(node) \wedge \neg Fired(id, node) \wedge \\
&\quad Assert(Fired(id, node)) \wedge Assert(NAFired(id)) \qquad (4.7)
\end{aligned}
$$

Bei der `Assign`-Aktivität verhält es sich ähnlich zu den beiden eben genannten nicht beobachtbaren Aktivitäten, allerdings muss zusätzlich zu dem erneuten Traversieren des Baumes noch das Setzen der Variablen in der Wissensbasis erfolgen, damit beispielsweise eine `WorkUnit` später auf die Variablen zwecks Schleifenprüfung zugreifen kann.

$$
\begin{aligned}
&IsValid(id, \_, node) \leftarrow \\
&\quad IsAssign(node) \wedge \neg Fired(id, node) \wedge \\
&\quad Assign(\_, \_, var, Expression(value)) = node \wedge \\
&\quad ((ValueOf(id, var, \_), Retract(ValueOf(id, var, \_))) \vee \\
&\quad\ (\neg ValueOf(id, var, \_), true)) \wedge \\
&\quad Assert(ValueOf(id, var, value)) \wedge \\
&\quad Assert(Fired(id, node)) \wedge Assert(AFired(id)) \qquad (4.8)
\end{aligned}
$$

In Axiom 4.8 erfolgt das Setzen von Variablen mithilfe des Prädikates $ValueOf$, welches den Wert $value$ einer Variablen var innerhalb einer Prozessinstanz id definiert. Wird beim Durchlaufen des Baumes ein `Assign`-Knoten erreicht, wird geprüft, ob die Variable schon

[5] Dafür wird bei einer `SilentAction` der Fakt *SAFired* und bei einer `NoAction` der Fakt *NAFired* für die jeweilige Choreographie-Instanz zur Wissensbasis hinzugefügt.

gesetzt ist. Ist sie gesetzt, dann wird der alte Wert vor dem Setzen mithilfe von *Retract* aus der Wissensbasis gelöscht. Zusätzlich wird der Knoten auf *gefeuert* gesetzt und mit $Assert(AFired(id))$ gespeichert, dass der `Assign`-Knoten in dieser Instanz bereits durchlaufen worden ist. Der Baum wird nach Erkennen einer `Assign-Aktivität` wie bei `NoAction` und `SilentAction` erneut durchlaufen.

Komplexe zusammengesetzte Aktivitäten: Die Knoten für zusammengesetzte Aktivitäten delegieren die Prüfung der Validität an ihre Unterknoten. Die komplexen Aktivitäten sind daher ordnende Strukturen im Prozess, die ein bestimmtes Verhalten modellieren.

Das Verhalten einer Sequenz wird mithilfe von Axiom 4.9 beschrieben. Eine Sequenz ist valide bezüglich einer eingegangenen Nachricht, wenn sie selbst noch nicht gefeuert hat und der aktuelle Knoten der Sequenz valide ist. Der aktuelle Knoten wird mit dem Prädikat $GetCurrentNode(id, node, current)$ ermittelt: Ist noch kein Knoten ausgewählt, wird der erste Knoten der Sequenz als aktueller Knoten betrachtet. Ansonsten wird der letzte Knoten aus der Sequenz betrachtet, der bisher noch nicht gefeuert hat. Ist der aktuelle Knoten valide, wird geprüft, ob der Unterknoten gefeuert hat. Dabei müssen zwei Fälle unterschieden werden: zum einen, ob es einen weiteren Knoten in der Sequenz gibt (erfolgreich, wenn $SetNextNode(id, node, current)$ wahr ist) und zum anderen, ob der letzte Knoten in der Sequenz erreicht wurde. Im letzten Fall gilt die Sequenz als gefeuert, wenn der letzte Unterknoten der Sequenz gefeuert hat. Ist keiner dieser beiden Fälle wahr, muss nichts weiter getan werden.

$$
\begin{aligned}
&IsValid(id, message, node) \leftarrow \\
&\quad IsSequence(Node) \wedge \neg Fired(id, node) \wedge \\
&\quad GetCurrentNode(id, node, current) \wedge IsValid(id, message, current) \wedge \\
&\quad ((fired(id, current) \wedge SetNextNode(id, node, current)) \vee \\
&\quad\ (fired(id, current) \wedge GetLastNode(node, last) \wedge \\
&\quad\quad current = last \wedge Assert(Fired(id, node))) \vee (true))
\end{aligned}
\tag{4.9}
$$

Bei einer Auswahl (Choice) lässt sich das Verhalten mit Axiom 4.10 beschreiben. Eine Auswahl gilt als valide, wenn sie selbst noch nicht gefeuert hat und der aktuelle Knoten valide ist. Ist noch kein aktueller Knoten ausgewählt (beim ersten Betreten der Regel ist dies der Fall), so muss erst ein Knoten ausgesucht werden (`NoAction` und `SilentAction` werden dabei allerdings ausgenommen, weil diese Knoten direkt feuern würden). Dies wird durch einfaches Durchprobieren der Unterknoten gemacht, bis einer der Unterknoten valide ist. Ist in beiden Fällen der ausgesuchte Unterknoten valide, dann ist zu prüfen, ob der Knoten gefeuert hat. Ist er das, so muss auch die Auswahl gefeuert werden.

$$
IsValid(id, message, node) \leftarrow
$$

$$
\begin{aligned}
&IsChoice(node) \wedge \neg Fired(id, node)) \wedge \\
&((CurrentNode(id, node, current) \wedge IsValid(id, message, current)) \vee \\
&\ (\neg CurrentNode(id, node, \_) \wedge GetChild(node, current) \wedge \\
&\quad \neg IsSilentAction(current) \wedge \neg IsNoAction(current) \wedge \\
&\quad IsValid(id, message, current) \wedge Assert(CurrentNode(id, node, current)))) \wedge \\
&((Fired(id, current) \wedge Assert(Fired(id, node))) \vee (true))
\end{aligned} \tag{4.10}
$$

Parallele Aktivitäten in WS-CDL werden gleichzeitig gestartet. Allerdings besteht kein Zwang, dass diese Aktivitäten auch gleichzeitig abschließen. Für die Verifikation muss daher nur geprüft werden, ob jeweils einer der Unterknoten für eine Nachricht korrekt verifiziert, wie in Axiom 4.11 beschrieben. Eine parallele Aktivität ist also valide, wenn sie selbst noch nicht gefeuert hat und einer der Kinder der parallelen Aktivität valide ist. Ist das der Fall, muss noch geprüft werden, ob der valide Unterknoten gefeuert hat, um dann die Anzahl der gefeuerten parallelen Aktivitäten zu erhöhen (siehe $IncrementNodesFired$). Ist die Anzahl der gefeuerten Unterknoten gleich der Anzahl der Unteraktivitäten (siehe $LastNodeFired(id, node)$), dann muss auch die parallele Aktivität gefeuert werden.

$$
\begin{aligned}
&IsValid(id, message, node) \leftarrow \\
&\quad IsParallel(node) \wedge \neg Fired(id, node)) \wedge \\
&\quad GetChild(node, child) \wedge IsValid(id, message, child) \wedge \\
&\quad ((Fired(id, child) \wedge IncrementNodesFired(id, node, fired) \wedge \\
&\qquad (LastNodeFired(id, node) \wedge Assert(Fired(id, node))) \vee (true)) \vee \\
&\quad (true))
\end{aligned} \tag{4.11}
$$

Bei der Beschreibung von WS-CDL-Dokumenten sind Schleifen durch WorkUnits realisiert. Diese WorkUnits funktionieren im Kern genauso wie die drei anderen komplexen Aktivitäten. Eine WorkUnit ist daher bezüglich einer Nachricht valide, wenn sie bisher noch nicht gefeuert hat und ihr Unterknoten valide ist, wie in Axiom 4.12 spezifiziert. Falls der Unterknoten allerdings feuert, müssen zwei Fälle unterschieden werden. Zum einen, ob die Schleifenbedingung mit $Repeat(id, node)$ wahr ist. In diesem Fall müssen alle $CurrentNode$- und $ParallelNodesFired$- sowie die $Fired$-Fakten zu allen Unterknoten zurückgesetzt werden, damit die Schleife erneut durchlaufen werden kann. Dies wird mithilfe der Hilfsregel $ResetChildren(Node)$ erreicht. Wird die Schleife nicht noch einmal durchlaufen, muss auch die WorkUnit auf gefeuert gesetzt werden.

$$
\begin{aligned}
&IsValid(id, message, node) \leftarrow \\
&\quad IsWorkUnit(node) \wedge \neg Fired(id, node)) \wedge \\
&\quad GetChild(node, current) \wedge IsValid(id, message, xurrent) \wedge
\end{aligned}
$$

$$((Fired(id, current) \wedge \\ (Repeat(id, node) \wedge ResetChildren(node)) \vee \\ (\neg Repeat(id, node) \wedge Assert(Fired(id, node))) \vee (true)) \tag{4.12}$$

Damit sind alle vier komplexen Aktivitäten durch Axiome beschrieben und prinzipiell ist im Zusammenspiel mit den Fakten zur Choreographie-Struktur und den Basisaktivitäten die Laufzeitverifikation möglich. Allerdings fehlt noch die Behandlung von nicht beobachtbaren Ereignissen, welche ein erneutes Traversieren des WS-CDL-Baumes veranlassen. Für diesen Fall werden für `Assign`-, `NoAction`- und `SilentAction`-Aktivitäten beim Betreten der Regeln Fakten generiert, die anzeigen, dass diese Aktivitäten betreten wurden. In der obersten Ebene des WS-CDL-Baumes wird dann geprüft, ob eine dieser Aktivitäten betreten wurde und bei Bedarf wird dann die Verifikation neu durchlaufen.

$$IsValid(id, message, node) \leftarrow \\ Root(node) \wedge \neg Fired(id, node)) \wedge \\ GetChild(node, current) \wedge isValid(id, message, current) \wedge \\ ((AssignFired(id) \wedge Retract(AFired(id)) \wedge IsValid(id, message, current)) \vee (true)) \wedge \\ \ldots \\ ((Fired(id, current) \wedge Assert(Fired(id, node))) \vee (true)) \tag{4.13}$$

Wird der Kindknoten des Hauptknotens gefeuert, so wird auch der Hauptknoten gefeuert. Dies Verhalten des Neuanlaufens der Verifikation und des Feuerns von Wurzelknotens ist in Axiom 4.13 definiert. Im Axiom 4.13 ist das Ganze allerdings nur auszugshaft für eine `Assign`-Aktivität definiert. Andere nicht beobachtbaren Aktivitäten wie `NoAction` und `SilentAction` werden auf die gleiche Weise behandelt, wenn das entsprechende Fakt gesetzt wurde.

4.2.2. Implementierung

Die im letzten Abschnitt vorgestellten Regeln und Algorithmen sind die Basis der Implementierung für die regelbasierte Laufzeitverifikation von Choreographien. Die Verifikation basiert dabei auf zwei Komponenten: und zwar einer Komponente, welche WS-CDL-Choreographie-Beschreibungen einliest und in die in Abschnitt 4.2.1.1 vorgestellten Choreographie-Fakten übersetzt und einer weiteren Komponente, die aufgrund der erzeugten Regeln einen effizienten *Soll/Ist*-Vergleich zur Laufzeit durchführt. Intern werden die Axiome aus Abschnitt 4.2.1.2 und die Fakten aus Abschnitt 4.2.1.1 mithilfe einer Prolog-Implementierung verwaltet.

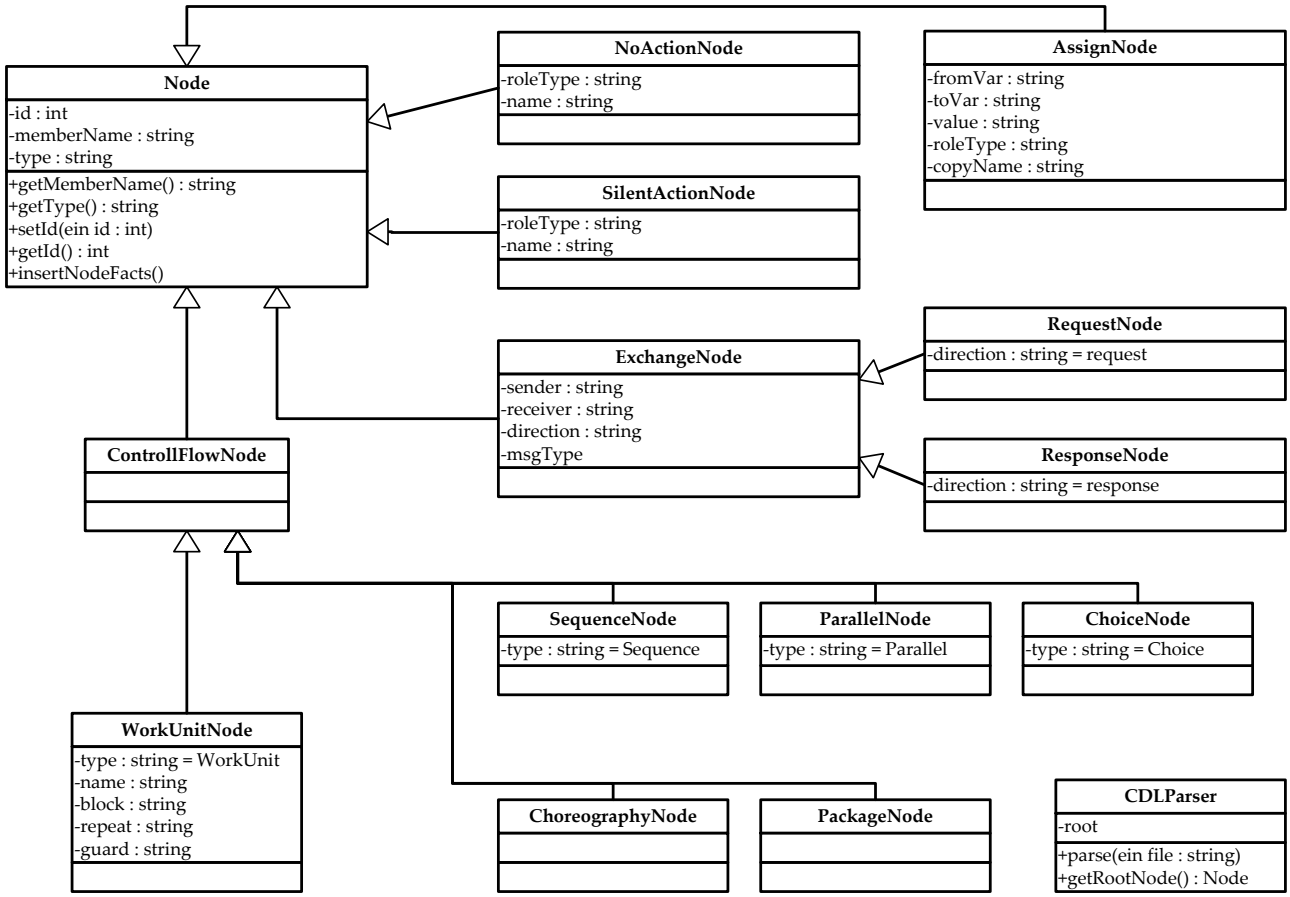


Abbildung 4.12.: Klassenhierarchie des WS-CDL-Parsers

4.2.2.1. Abbildungsprozess für WS-CDL

Der Interpreter basiert auf mehreren Klassen. Zum einen aus einem Parser, welcher ein WS-CDL-Dokument einlesen kann und zum anderen aus Klassen, welche die einzelnen Knoten im WS-CDL-Baum repräsentieren. Abbildung 4.12 zeigt die Klassenhierarchie für den Parser. Die in Abschnitt 4.2.1.1 vorgestellten Basisaktivitäten wie `NoAction`, `SilentAction`, `Assign` und `Exchange` erben alle von der Klasse `Node`, wobei `Request` und `Response` den Datenaustausch für `Exchange` noch weiter spezialisieren. Die Klasse `Node` stellt dabei alle Methoden zur Verfügung, um einen Hierarchie aus Aktivitäten zu erstellen und diese Hierarchie zu traversieren. Beispielsweise gibt es Methoden, um ein bestimmtes Kind eines Knotens oder alle Kinder eines Knotens auszugeben sowie neue Knoten hinzuzufügen. Die komplexen Aktivitäten erben alle von `ControlFlowNode`, welche auch von `Node` erbt. Mit Unterstützung der Klassenstruktur lässt sich ein Aktivitätsbaum aus WS-CDL direkt in eine Java-Struktur abbilden und benutzen. Der Parser liest dabei das XML-Dokument elementweise durch, erzeugt beispielsweise bei Erkennen einer Sequenz das entsprechende Sequenz-Objekt und fügt im weiteren Verlauf des Parsens die Kinder zu diesem Objekt hinzu.

Ist das Parsen abgeschlossen, kann über das Wurzelobjekt mithilfe der Methode `Insert-`

`NodeFacts()` die Struktur der Choreographie an die Wissensbasis übergeben werden. Dabei werden die Knoten nach Art der Breitensuche durchlaufen und die Fakten jedes einzelnen Knotens an die Wissensbasis weitergegeben. Jede einzelne Knotenklasse überschreibt dabei die Methode `InsertNodeFacts()` und erzeugt die entsprechend in Abschnitt 4.2.1.1 beschriebenen Fakten für den jeweiligen Knoten. Das Weitergeben der Fakten an Eclipse-CLP läuft dabei wie in Quellcode 4.2 über einen einfachen RPC-Aufruf innerhalb der Methode `InsertNodeFacts()`. Dabei wird in jeder Klasse zusätzlich die Methode `getMemberName()` genutzt, welche den Typ eines Knotens zurückgeben kann. Bei Basisaktivitäten entspricht der Name den Ereignisfakten, bei komplexen Aktivitäten liefert die Methode beispielsweise den eindeutigen Namen der Sequenz bestehend aus Typ-Bezeichnung und Sequenz-Identifikator.

Quellcode 4.2: Methoden zum Einfügen eines Exchange-Faktes in die Wissensbasis

```
1  @Override
2  public String getMemberName() {
3    // exchange(buyerrequestsquoterequest,buyerrole,sellerrole).
4    return "exchange(" + getMappedMessageType().toLowerCase()
5      + getDirection() + "," + getSender().toLowerCase() + ","
6      + getReceiver().toLowerCase() + ")";
7  }
8
9  @Override
10 public void insertNodeFacts() {
11   String fact = "basicActivity(" + getMemberName() + ")";
12   ...
13   eclipse.rpc("assert(" + fact + ")");
14 }
```

Die Wissensbasis ist nach dem Durchlaufen des Parsers mit den Fakten der Choreographie-Struktur gefüllt und der Verifizierer ist in der Lage, eingehende Nachrichten zu verifizieren.

4.2.2.2. Verifikation mithilfe von Eclipse-CLP

Der Verifizierer basiert auf zwei internen Komponenten. Zum einen der Sensorschnittstelle und zum anderen der Schnittstelle zu Eclipse-CLP.

Die Web-Service-Schnittstelle empfängt die Nachrichten vom Enterprise-Service-Bus und legt diese intern in einer Queue ab. Die in der Queue abgelegten Nachrichten müssen mindestens eine vorher definierte Zeitdauer (abhängig von ihrem Zeitstempel) innerhalb der Queue verbleiben. Der Empfang der Nachrichten muss nicht entsprechend in der Reihenfolge ihrer Zeitstempel an der Empfangsschnittstelle erfolgen; daher müssen die Nachrichten bei Eingang noch sortiert werden. Als praxistauglich hat sich dabei eine Zeitspanne

von einer Sekunde herausgestellt, in der die Nachrichten in der Queue verbleiben und erst dann zur Verifikation herausgenommen werden. Die Herausnahme von Nachrichten aus der Queue mit der Übergabe an den Verifizierer geschieht in einem separaten Thread.

Der Verifizierer selbst besteht aus der Verifikationskomponente, welche Eclipse-CLP ansprechen und damit Anfragen an die Wissensbasis stellen kann. Die in Abschnitt 4.2.1.2 entwickelten Axiome liegen als Prolog-Programm vor und werden von Eclipse-CLP bei Programmstart kompiliert. Da die Axiome schon als Horn-Klauseln formuliert sind, entspricht das implementierte Prolog-Programm nahezu identisch den vorgestellten Regeln. Die durch den Parser zusätzlich eingefügten Fakten über die Choreographie-Struktur ermöglichen dem Verifizierer, Nachrichten zur Laufzeit zu verifizieren, in dem direkt Anfragen an die Wissensbasis getätigt werden; siehe Quellcode 4.3.

Quellcode 4.3: Verifikation von Nachrichten

```
1 public boolean check(SoapMessage msg) {
2   String chorid = Util.getPrologChorId(msg.getChoreographyId());
3   String m = msg.getFormalEventString();
4   String query = "isValid(" + chorid + "," + m + ")";
5
6   boolean check = PrologEngine.getInstance().query(query);
7   return check;
8 }
```

Die eigentliche Abfrage geschieht, indem in Prolog die Regel $isValid(cid, event)$ ausgeführt wird. Diese Regel beginnt beim Wurzelknoten der Choreographie, welcher die Anfrage dann an die jeweiligen Kinderknoten delegiert. Es wird nur *true* oder $false$ als Resultat zurückgegeben. Eine genauere Analyse, welcher Pfad durch den Aktivitätsbaum durchschritten wurde, ist nur möglich, wenn nach jeder Nachricht noch die gefeuerten Knoten notiert werden. Das vorgestellte Analyseverfahren eignet sich daher auch nur zur schnellen Verifikation von Choreographien. Eine genaue Detailanalyse, welcher Knoten zu welcher Zeit den Fehler verursacht hat, ist mit den Regeln nicht gut möglich. Genauso lassen sich non-funktionale Eigenschaften wie Zeitschranken schwer mit diesen Verifikationsverfahren prüfen. Das Verfahren lässt sich aber später gut zum Vergleichen heranziehen, um zu definieren, welchen Mehraufwand das Ereigniskalkül durch die Behandlung von quantifizierten Zeiten einführt. Genauso lassen sich die Konzepte zum Einteilen der Wissensbasis in spezifische Choreographie-Fakten und generelle WS-CDL-Axiome aus diesem Ansatz auch für das Ereigniskalkül übernehmen. Die WS-CDL-Axiome und Fakten zur Struktur müssen dafür aber an das Kalkül angepasst werden.

4.2.3. Evaluation

Auf Basis der im letzten Abschnitt vorgestellten Implementierung werden jetzt verschiedene Evaluationsszenarien eingeführt. Die Szenarien sind darauf ausgelegt, verschiedene Aspekte der Implementierung zu zeigen. Die Szenarien geben die Möglichkeit, die Stärken und die Schwächen der Implementierung sowie des vorgeschlagenen Modells zu untersuchen. Das Hauptaugenmerk der Evaluation liegt dabei auf dem Testen der Eignung des Modells zur Laufzeitverifikation von Prozess-Choreographien und damit auf der Geschwindigkeit, in der Nachrichten verifiziert werden können. Ein Nebenaspekt der Evaluation liegt auf der Skalierbarkeit der vorgestellten Implementierung.

4.2.3.1. Szenarien

Um das vorgestellte Modell und die dazugehörige Implementierung zu evaluieren, werden zwei verschiedene Beispiel-Choreographien unterschiedlicher Komplexität vorgestellt. Die Komplexität einer Choreographie hängt von dem Einsatz verschiedener Modellierungsartefakte ab: Zum einen ergibt sich aus einer größeren Schachtelungstiefe innerhalb des WS-CDL-Baumes, dass in der Implementierung Regeln öfters durchlaufen werden. Zum anderen bedingt der Einsatz von nur einfachen Regeln wie beispielsweise einer Sequenz oder der Einsatz von nur wenigen nicht beobachtbaren Artefakten wie `SilentAction`, dass der WS-CDL-Baum so wenig wie möglich erneut traversiert werden muss, um zu entscheiden, ob eine Nachricht valide ist. Genauso verhält es sich mit der Nutzung von Schleifen. Daher wird für die Evaluation auf der einen Seite eine komplexe Choreographie vorgestellt, welche der Beispiel-Choreographie aus dem WS-CDL-Primer entspricht [FRT06]. Die Choreographie zeichnet sich durch den Einsatz von `Assign` und der Definition von bedingten Schleifen aus. Auf der anderen Seite wird eine einfache Choreographie-Definition für eine Beispielinteraktion zwischen Europol und Eurojust benutzt, die hauptsächlich aus Sequenzen besteht. Beide Szenarien nutzen zusammengenommen alle vorgestellten Basisaktivitäten und komplexen Aktivitäten und liefern damit ein gutes Bild der vorgestellten Implementierung.

Als Testumgebung wird ein Intel i5-750 mit 4 Kernen und 8GB RAM eingesetzt. Auf dieser Umgebung laufen der Service-Bus, der Anwendungsserver (Tomcat) für die Web-Services und der Verifikationsdienst gleichzeitig. Die Testumgebung ist leistungsfähig genug, alle Komponenten gleichzeitig auf der Maschine zu betreiben.

Szenario 1 - Kaufvorgang: Im ersten vorgestellten Szenario geht es um einen Kaufvorgang, bei dem ein Besteller die Möglichkeit hat, mit dem Verkäufer über den Preis zu verhandeln. Die Verhandlung läuft innerhalb einer Schleife ab, welche erst abgebrochen wird, wenn Einigkeit über den Preis besteht. Danach wird die Ware zum verhandelten Preis bestellt und der Verkäufer führt mithilfe eines externen Dienstleisters eine Kreditkartenprüfung durch. Wenn diese erfolgreich ist, weist der Verkäufer einen weiteren externen Dienstleister an, die Ware zum Käufer zu senden. Dieser externe Versender nimmt Kontakt mit dem Käufer auf

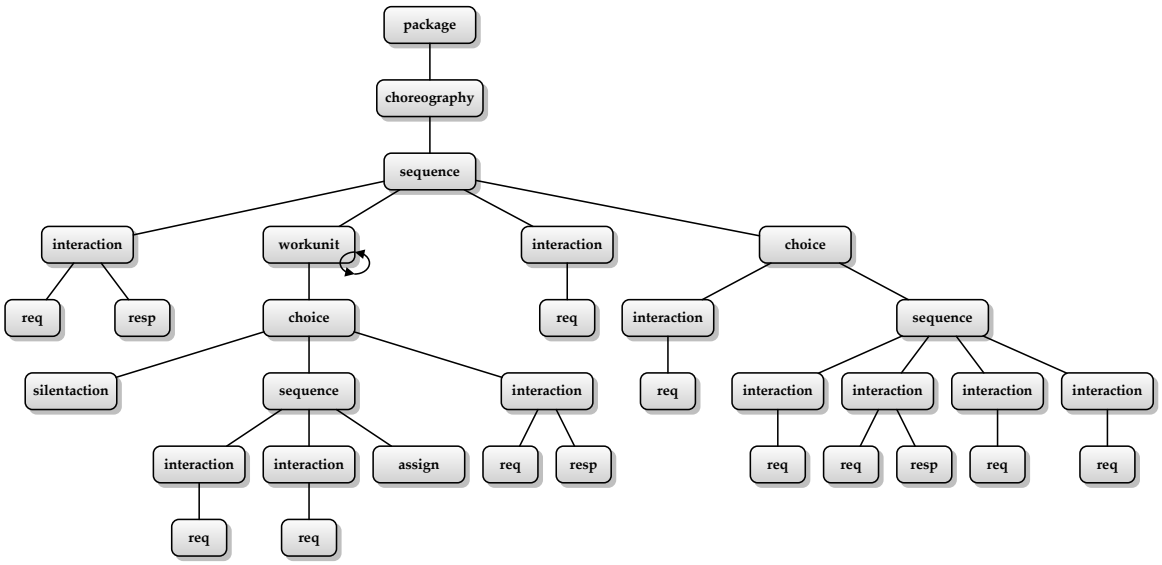
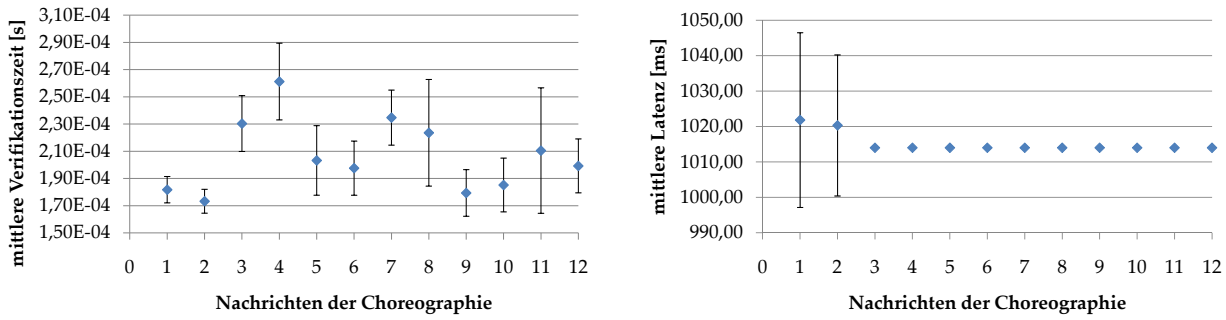


Abbildung 4.13.: WS-CDL-Struktur für einen Kaufvorgang

und teilt ihm die Details zum Versand mit. Der eigentliche Versand der Ware erfolgt dann nicht mehr über Web-Service-Kommunikation und kann daher nicht durch den Service-Bus beobachtet werden. Die Struktur des WS-CDL-Dokuments für das Szenario ist auch in Abbildung 4.13 dargestellt.

Für die Implementierung dieses Szenarios werden vier Web-Services benötigt, die dem Verhalten der einzelnen Rollen in der Choreographie-Definition entsprechen. Im Enterprise-Service-Bus werden zusätzlich die Proxies für die Dienste angelegt. Für die erste Evaluation wird die Choreographie zehn Mal in der gleichen Weise durchlaufen. Dabei werden jedes Mal nach einem Durchlauf der Service-Bus, der Verifikationsdienst und die Verhaltensdienste neu gestartet, damit für jeden Testlauf die gleichen Testbedingungen herrschen. Der Ablauf der Choreographie ist dabei immer gleich und damit lassen sich die Zeiten zur Verifikation in Abhängigkeit der versendeten Nachricht jeweils einzeln aufzeichnen. Aufgezeichnet werden dabei zwei verschiedene Zeiten: Zum einen die *reine Verifikationszeit* pro Nachricht und damit, wie viel Zeit Eclipse-CLP mit den vorgestellten Horn-Klauseln zur Verifikation benötigt, und zum anderen, wie viel Zeit zwischen dem Erkennen der Nachricht im Service-Bus bis zur erfolgten Verifikation vergeht (*Latenz*).

In Abbildung 4.14(a) ist die reine Verifikationszeit in Abhängigkeit von der versendeten Nachricht gezeigt. Der Verlauf der Punkte lässt sich recht gut anhand der Regeln und der Struktur aus Abbildung 4.13 nachvollziehen. Die ersten beiden Nachrichten stellen eine Preisanfrage dar und haben einen relativ kurzen Pfad zum Blatt, was sich in einer kurzen Verifikationszeit niederschlägt. Die erste Nachricht wird dabei langsamer verifiziert als die Zweite, weil für die Interaktion noch der Fakt für `CurrentNode` initialisiert werden muss. Bei der zweiten Nachricht kann die Verifikation direkt beim Response-Knoten weitermachen. Bei der dritten Nachricht wird die `WorkUnit` betreten und in der `Choice` erstmal der zutreffende Knoten im Unterbaum der Auswahl gesucht. Dabei werden für die Sequenzen jeweils



(a) Verifikationszeit (b) Latenz

Abbildung 4.14.: Einkäufer-Verkäufer-Szenario

die Fakten für `CurrentNode` initialisiert. Die nachfolgende Antwort wird daher schneller verifiziert, da in der Auswahl jetzt bekannt ist, welcher Unterbaum aktiv ist. Bei der fünften Nachricht wird durch die Schleifenbedingung die `WorkUnit` erneut ausgeführt und die `Sequence` mit der `Assign`-Aktivität am Ende aktiviert. Der Abschluss der `WorkUnit` erfolgt allerdings erst mit Eintreffen der siebten Nachricht, da `Assign` ein nicht beobachtbares Ereignis darstellt. Dabei läuft der Algorithmus so, dass der Baum bis zum `Assign`-Knoten traversiert wird, die Variablen gesetzt werden und beim Durchreichen der Antwort an die Vaterknoten ab dem `Choreographie`-Knoten das Traversieren erneut beginnt, um einen Nachrichtenknoten zu finden. Daher wird bei der siebten Nachricht auch der Baum zweimal bis zu einem Blatt durchlaufen, was sich in einer erhöhten Laufzeit niederschlägt. Danach wird mit Nachricht 8 - 12 der rechte Teilbaum durchlaufen. Die Verifikation von Nachricht 8 benötigt mehr Zeit als die nachfolgenden Nachrichten, da bei der Auswahl wieder erst geprüft werden muss, welcher Teilbaum durch die Nachricht aktiviert wird.

In Abbildung 4.14(b) ist die Latenz in Abhängigkeit von der versendeten Nachricht dargestellt. Dabei ist zu beachten, dass der Verifikationsdienst die Nachrichten erst puffert, damit Nachrichten, die in verschiedenen Threads nicht in der richtigen Reihenfolge empfangen wurden, noch in die richtige Reihenfolge gebracht werden können. Dabei wird beim Empfangen die Nachricht erstmal in eine Queue gelegt, in welchem die Nachrichten anhand ihres Zeitstempels sortiert vorliegen und erst nach Ablauf von einer Sekunde entnommen werden können. Damit wird gewährleistet, dass die Nachrichten in derselben Reihenfolge wie sie beim ESB auftreffen auch beim Verifikationsdienst eintreffen. Die Latenz ist daher immer größer als eine Sekunde und bewegt sich im Bereich von 1021,80 ms und 1014 ms. Die Schwankungen am Anfang der Choreographie lassen sich damit erklären, dass der Enterprise-Service-Bus nach erneutem Start erstmal seine Caches und entsprechende Sende-Queues initialisieren muss. Ist das erfolgt, bleibt die Latenz nahezu konstant, da die reine Verifikationszeit sich für diese Choreographie im Bereich von 0,2 ms bewegt.

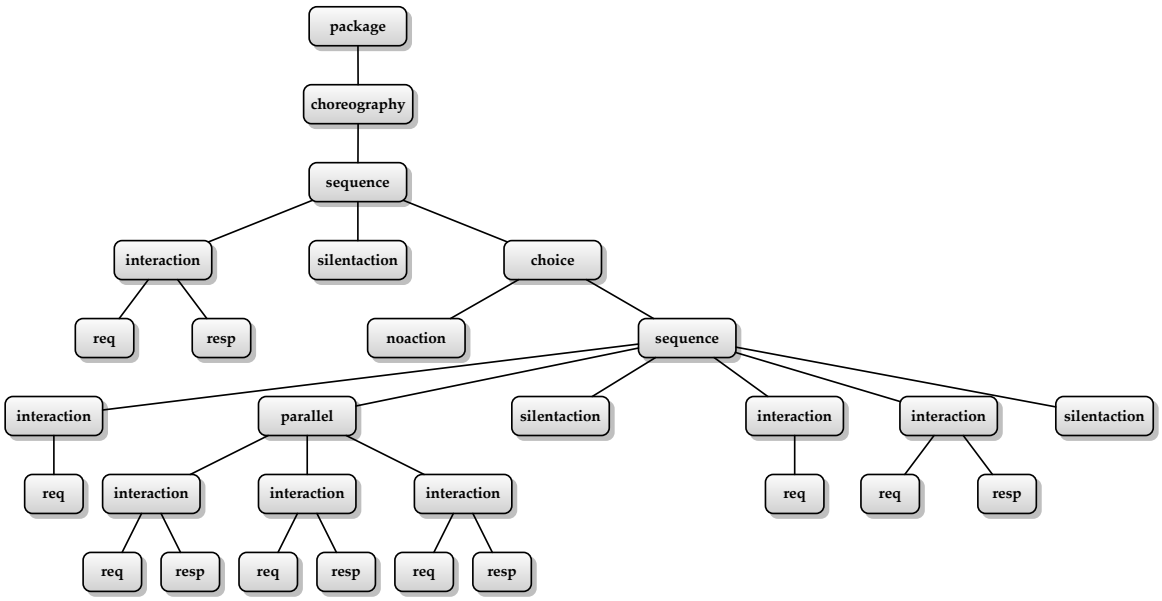
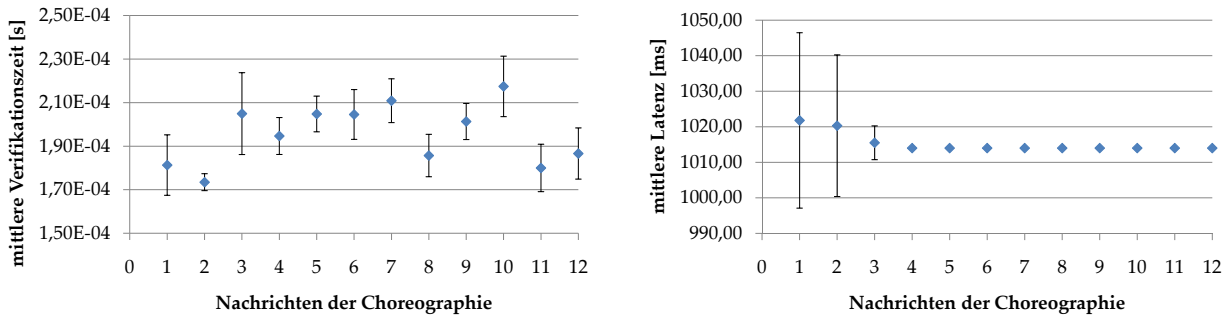


Abbildung 4.15.: WS-CDL-Struktur für das Europol-Eurojust-Szenario

Szenario 2 - Europol-Eurojust-Interaktion: Im zweiten Szenario geht es um eine einfache Anfrage von Eurojust an Europol, welche an weitere Teilnehmer delegiert wird, um ein Ergebnis für die Anfrage zu erhalten. Als Beispiel-Szenario gilt wieder der in Kapitel 3.2.2.2 beschriebene Kokain-Fund eines Polizisten bei einer Routineuntersuchung eines Schiffes in Malaga. Europol und Eurojust arbeiten in diesem Fall zusammen und brauchen dafür weitere Informationen, die untereinander ausgetauscht werden. Als Erstes wird für diesen Fall vom spanischen Eurojust-National-Member eine Akte im Case-Management-System (CMS) angelegt. Nach einem Meeting der National-Member, die vom Fall betroffen sind, werden weitere Informationen über Personen von Europol über den Europol-Liason-Officer (ELO) angefordert. Dieser kann die Anfrage nicht direkt beantworten und ermittelt die angeforderten Informationen über drei verschiedene interne Systeme und trägt sie dann zusammen. Dürfen die Daten weitergegeben werden, werden sie an den betreffenden Eurojust-National-Member weitergeleitet. Dieser aktualisiert die im CMS angelegte Akte und beruft ein weiteres Treffen der National-Member zum Fall ein, um das weitere Vorgehen zu besprechen. In diesem einfachen Fall werden Web-Service-Aufrufe für die Kommunikation mit dem CMS, mit dem ELO, den National-Member und den einzelnen Systemen auf Europol-Seite getätigt.

Für die Implementierung dieses Szenarios werden sechs Web-Services benötigt, die dem Verhalten der einzelnen Rollen in der Choreographie-Definition entsprechen. Im Enterprise-Service-Bus werden zudem die Proxy-Dienste für diese Dienste angelegt. Wie im ersten Szenario wird die Choreographie zehn Mal in der gleichen Weise durchlaufen. Dabei werden wie vorher jedes Mal der Service-Bus, der Verifikationsdienst und die Verhaltensdienste neu gestartet, damit für jeden Test die gleichen Testbedingungen herrschen.

In Abbildung 4.16(a) ist die reine Verifikationszeit für dieses Szenario in Abhängigkeit von

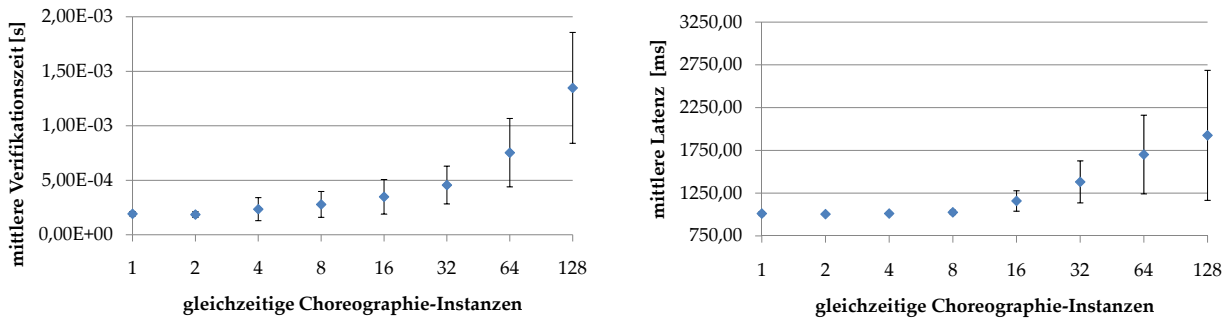


(a) Verifikationszeit (b) Latenz

Abbildung 4.16.: Europol-Eurojust-Szenario

der versendeten Nachricht gezeigt. Auch dieser Verlauf der Punkte lässt sich recht gut anhand der Regeln und der Struktur aus Abbildung 4.15 nachvollziehen. Mit der ersten Nachricht wird im CMS die Akte angelegt. Bei der Verifikation werden dabei im WS-CDL-Baum die Sequenz und die Interaktion mit *CurrentNode* initialisiert, daher ist die zweite Nachricht schneller in der Verifikation, da die Initialisierung der Knoten wegfällt. Zur Verifikation wird der Baum wieder zweimal traversiert, da die SilentAction besucht wird. Der Verifikationsalgorithmus beginnt nach Besuchen dieses Knotens wieder beim Wurzelknoten. Danach wird der ganze rechte Teilbaum besucht und dabei initialisiert. Die nachfolgenden sechs Nachrichten sind in dem Parallelzweig des Baumes enthalten und werden etwas langsamer als die ersten beiden Nachrichten verifiziert, da zur Verifikation jeweils die richtige Interaktion gesucht werden muss, zu der die Nachricht gehört. Am Ende wird die Verifikation der parallelen Aktivitäten schneller als die ersten Nachrichten, da bereits einige der Knoten der Parallel-Aktivität gefeuert haben und daher nicht erneut besucht werden müssen. Bei Nachricht 10 ist wieder zu erkennen, dass eine SilentAction besucht wird und damit der Baum wieder zweimal durchlaufen wird. Die Verifikationszeit nimmt damit etwas zu. Die Prüfung der letzten beiden Nachrichten wird dann wieder schneller durchlaufen, da eine Sequenz schneller durchlaufen wird als eine Parallel-Aktivität. Bei der letzten Nachricht muss allerdings die Sequenz darüber einen Knoten weiterschalten, was sich in einer leicht erhöhten Verifikationszeit im Gegensatz zu Nachricht 11 niederschlägt. Das ganze passiert intern über die `Assert`- und `Retract`-Anweisungen von Prolog, welche Zeit kosten. Die Choreographie ist bis Nachricht 12 erfolgreich verifiziert, ist aber noch nicht komplett abgeschlossen, da noch die letzte SilentAction fehlt. Diese ist allerdings für das System nicht beobachtbar, daher kann die Choreographie technisch als abgeschlossen gelten, da keine weitere Kommunikation über Web-Services mehr erfolgt.

In Abbildung 4.16(b) ist die Latenz in Abhängigkeit von der versendeten Nachricht gezeigt. Da die Verifikationszeit sich im Rahmen von 0,17 - 0,22 ms bewegt, sind auch hier keine Sprünge bei der Latenz zu erwarten. Auch am Anfang sind leichte Schwankungen zu beobachten, die auf den Enterprise-Service-Bus nach einem erneuten Start zurückzufüh-



(a) Verifikationszeit (b) Latenz

Abbildung 4.17.: Abhängigkeit von gleichzeitigen Prozessinstanzen

ren sind. Hat der Enterprise-Service-Bus seine Caches und seine Queue soweit initialisiert, bleibt auch in diesem Beispiel die Latenz nahezu konstant, da die reine Verifikationszeit im Gegensatz zur Weiterleitung der Nachricht an den Verifikationsdienst vernachlässigbar ist. Auch in diesem Szenario bewegt sich die Latenz zwischen 1022 und 1014 Millisekunden, da der Puffer jeweils eine Sekunde wartet, bis alle Nachrichten eingetroffen sind, damit die Reihenfolge der Nachrichten garantiert werden kann.

Szenario 3 - Mehrere Instanzen zur selben Zeit: Im dritten Szenario wird das komplexere der beiden bisherigen Szenarien auf Skalierbarkeit untersucht. Dafür werden für das Käufer-Verkäufer-Szenario in mehreren Schritten immer mehr gleichzeitige Instanzen des Prozesses gestartet und die Zeit zur reinen Verifikation sowie die Latenz gemessen. Nach jedem Schritt, bei dem die Anzahl der parallelen Prozessinstanzen gesteigert wird, wird der Enterprise-Service-Bus, die einzelnen Beispieldienste und der Verifikationsdienst neu gestartet, um gleichbleibende Testbedingungen zu gewährleisten.

In Abbildung 4.17(a) ist zu erkennen, dass bis zu 32 gleichzeitige Choreographie-Instanzen inkl. Service-Bus und Anwendungs-Server der Maschine keine Probleme bereiten. Die Verifikationszeit pro Nachricht steigt im Mittel kaum merklich an. Erkennbar ist, dass sich die Verifikation linear verhält[6]. Dies ist intuitiv einsehbar, da sich die Schritte zur Verifikation an sich nicht ändern, die Fakten-Menge aber linear vergrößert wird. Damit muss beispielsweise beim Testen des Prädikates $fired(Id, Node)$ bei einer größeren Anzahl von gleichzeitigen Prozessinstanzen auch eine größere Faktenmenge durchsucht werden, um zu erkennen, ob der Knoten in der speziellen Instanz gefeuert ist.

Etwas anders verhält es sich bei der Latenz, die sich nicht linear verhält. Zwar liegt die Verifikationszeit selbst bei 128 gleichzeitigen Prozessinstanzen bei etwa 1.4 Millisekunden pro Nachricht im Mittel, aber die Latenz pro Nachricht steigt im Mittel deutlich in den Sekundenbereich an. Ein Flaschenhals ist hier die Sensorschnittstelle mit dem Empfangspuffer, welcher die Nachrichten sortiert vorhält. Dies lässt sich zwar noch weiter optimieren, liegt

[6] Achtung: Die X-Achse ist logarithmisch aufgetragen.

aber etwas außerhalb der Betrachtung dieser Evaluation des Verifikationsansatzes. Beheben lässt sich dieses Problem beispielsweise durch Load-Balancing, in dem mehrere Dienste auf unterschiedlichen Rechnern den gleichen Choreographie-Typ verifizieren. Wichtig dabei ist, dass jeweils ein Verifikationsdienst pro Instanz zuständig ist und bei neuen Instanzen über ein Round-Robin-Verfahren jeweils ein anderer Verifikationsdienst für die neue Instanz ausgewählt wird. Die Implementierung der Verifikation ist daher gut skalierbar und kann auch in Cloud-Umgebungen für Prozessverifikation genutzt werden.

4.3. Laufzeitverifikation mithilfe des Ereigniskalküls

Um Choreographien mit dem Ereigniskalkül zur Laufzeit zu verifizieren, müssen die WS-CDL-Artefakte einer Choreographie in das Ereigniskalkül abgebildet werden. WS-CDL basiert auf XML und ist hierarchisch strukturiert. Dieser Fakt lässt sich bei der Abbildung in das Ereigniskalkül gut ausnutzen. Beispielsweise wird eine Sequenz durch ein Ereignis als valide angesehen, wenn die in der Sequenz aktuell abzuarbeitende Aktivität durch das gleiche Ereignis als korrekt verifiziert wird. Ein Fehler in der Sequenz tritt also auf, wenn der aktuelle abzuarbeitende Schritt der Sequenz nicht durch ein Ereignis als korrekt verifiziert wird. Das Prinzip ist wie eine Zuständigkeitskette zu verstehen, bei der Fehler in den unteren Ebenen jeweils Auswirkungen auf die oberen Ebenen haben. Zusätzlich kann in jeder Ebene entschieden werden, ob bestimmte Regeln wie die Reihenfolgen von Aktivitäten eingehalten werden. Die Effekte von Ereignissen auf WS-CDL-Aktivitäten werden entlang der Zuständigkeitskette bis an die Wurzelaktivität weitergegeben. Eine Choreographie ist daher solange als korrekt ausgeführt anzusehen, solange die Wurzelaktivität durch ein Ereignis über die Zuständigkeitskette als korrekt verifiziert wird.

Um diese hierarchische Struktur aufzubauen und die Effekte von Ereignissen auf die strukturierenden Elemente zu beschreiben, werden auf der einen Seite Fluents zur Beschreibung des Aktivitätsstatus und auf der anderen Seite Regeln zur Beschreibung der Effekte von Ereignissen auf die Fluents benötigt. Für Basisaktivitäten werden zwei Fluents genutzt, zum einen *Valid* und zum anderen *Complete*. Letzteres wird wahr, wenn eine Basisaktivität erfolgreich durch ein Ereignis abschließt. Basisaktivitäten sind zudem atomare Aktivitäten, welche sofort bei Auftreten des zugehörigen Ereignisses abschließen. Genauso sind Basisaktivitäten immer *Valid*, denn Fehler werden erst durch übergeordnete komplexe Aktivitäten entlang der Zuständigkeitskette erkannt. Für komplexe Aktivitäten werden die gleichen beiden Fluents genutzt, allerdings werden die beiden Fluents durch die jeweiligen Teilaktivitäten beeinflusst. Für die Beschreibung der Effekte auf die Fluents der komplexen Aktivitäten werden Regeln benötigt.

Abbildung 4.18 illustriert dieses Vorgehen anhand eines einfachen Beispiels: Zur Zeit $t = 0$ ist keine Aktivität *Complete* oder im Zustand *Valid*. Zur Zeit $t = 1$ wird ein Ereignis beobachtet und die zu diesem Ereignis gehörende Basisaktivität req_1 wird auf *Complete*

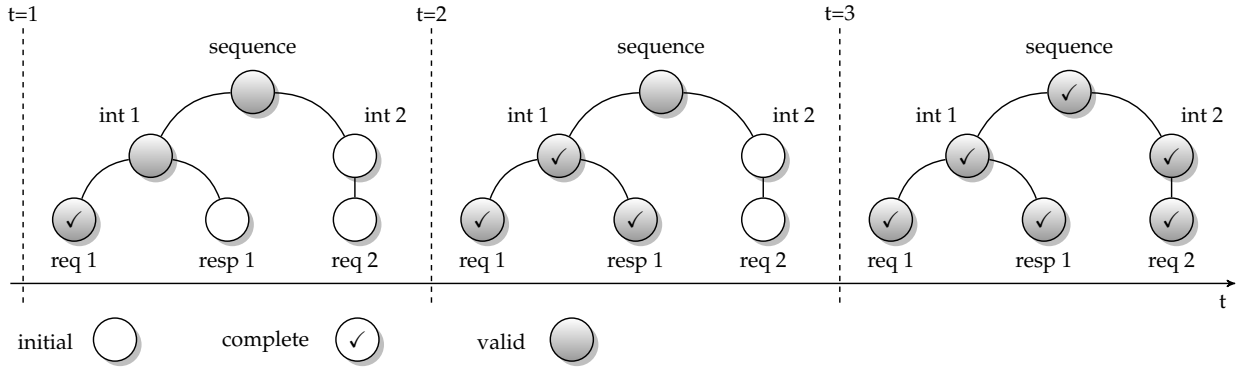


Abbildung 4.18.: Zuständigkeitskette

und *Valid* gesetzt. Diese Basisaktivität ist Teil einer Interaktion int_1, welche auf *Valid* gesetzt wird, da für diese Interaktion im ersten Schritt die Anfrage req_1 durchgeführt werden muss. Da die Anfrage durch das Ereignis valide wird, wird auch die Interaktion valide. Das Gleiche gilt für die *sequence*, da mit dem Ereignis der erste Schritt der Sequenz (die Interaktion) auf *Valid* gesetzt ist und somit auch die Sequenz valide wird. Mit Eintreffen der nächsten Nachricht zum Zeitpunkt $t = 2$ wird die Basisaktivität $resp_1$ auf *Complete* und *Valid* gesetzt. Die darüberliegende Interaktion int_1 prüft jetzt, ob der aktuelle Schritt valide ist. Der aktuelle Schritt kann in diesem Beispiel einfach ermittelt werden, in dem der erste Unterknoten der Aktivität betrachtet wird, welcher noch nicht erfolgreich beendet ist. In diesem Fall ist zu sehen, dass der aktuelle Schritt der Interaktion $reps_1$ ist. Da dieser Unterknoten gerade auf *Valid* gesetzt wurde, bleibt auch die Interaktion auf *Valid*, genauso wie die darüberliegende *sequence*. Zudem wird die Interaktion auf *Complete* gesetzt, da der letzte Schritt der Interaktion erfolgreich beendet wurde. Das gleiche Prinzip wird auch für das Ereignis zum Zeitpunkt $t = 3$ angewendet, welches die Beispiel-Choreographie erfolgreich verifiziert und beendet. Sollte beispielsweise zum Zeitpunkt $t = 2$ eine andere als die erwartete Nachricht eintreffen, wird int_1 nicht verifiziert werden und damit auch die darüberliegende Sequenz nicht. Das Ereignis würde zu diesem Zeitpunkt also nicht das Fluent *Valid* der Wurzelaktivität initiieren und damit wäre die Choreographie-Instanz fehlerhaft.

Zusätzlich zu den Fluents für Basisaktivitäten und komplexen Aktivitäten werden Regeln zur Beschreibung der Effekte von Ereignissen auf diese Fluents benötigt. Im Ereigniskalkül werden diese Regeln mithilfe der Prädikate *initiates* und *terminates* formuliert, welche die Aktivierung und das Abklingen der Fluents beschreiben. Es gibt zwei Wege zur Beschreibung der Regeln: Auf der einen Seite können Regeln für jedes einzelne Event und jede einzelne Aktivität definiert werden. Beispielsweise müssen dann für jede Basisaktivität und für jede komplexe Aktivität Regeln passend zu den möglichen Ereignissen definiert werden, und damit, wie sich die Fluents für jedes mögliche Event verhalten. Dies resultiert allerdings in einer steigenden Anzahl von Regeln, was Performanzprobleme nach sich zieht. Auf der anderen Seite kann für die Regeln auch die hierarchische Struktur ausgenutzt werden, in dem *indirekte Effekte* auf Fluents definiert werden. Ein indirekter Effekt wird durch Regeln

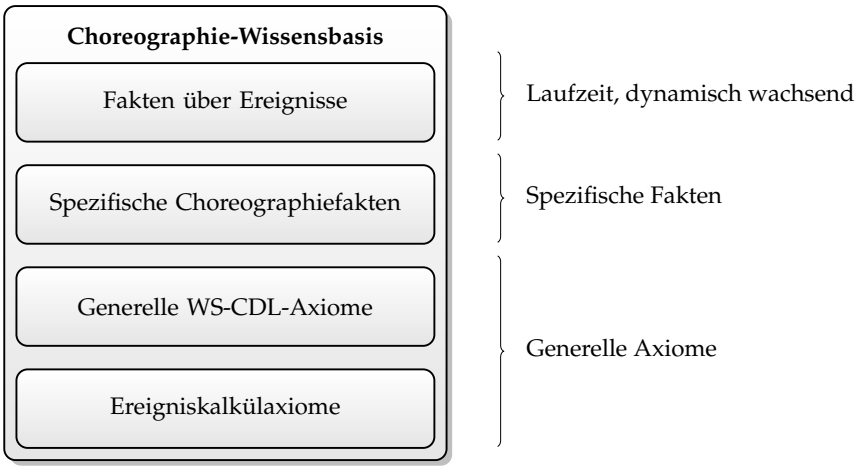


Abbildung 4.19.: Grundlegende Blöcke der WS-CDL-Formalisierung (nach [Voi10])

beschrieben, welche Fluents logisch miteinander verbinden. Beispielsweise kann eine Sequenz durch einen indirekten Effekt mit zusätzlichen Regeln valide werden, wenn eine in ihr enthaltene Basisaktivität *Valid* ist. Mit Unterstützung der indirekten Effekte lässt sich eine hierarchische Choreographie-Beschreibung sehr gut abbilden, in dem der zeitliche Fluentverlauf von komplexen Aktivitäten durch den zeitlichen Fluentverlauf der Teilaktivitäten beschrieben wird. Ein weiterer Vorteil dieses Vorgehens besteht darin, dass beispielsweise die Effekte von Teilaktivitäten auf eine Sequenz immer in der gleichen Art und Weise geschehen, da eine bestimmte Reihenfolge eingehalten werden muss. Der einzige für verschiedene Choreographien unterschiedliche Teil bei der Verifikation der Sequenz ist die Struktur der Sequenz an sich, also wie viele Teilaktivitäten in welcher Reihenfolge ausgeführt werden müssen. Es gibt für WS-CDL also generelle Axiome, die für alle Choreographien gelten, und auch domänenspezifische Axiome, welche sich auf die genaue Struktur der Choreographie für einen Anwendungsfall beziehen.

Zusammengefasst besteht eine auf diesem Ansatz basierende Wissensbasis also aus vier Teilen: Erstens aus den generellen Ereigniskalkül-Axiomen, welche den Cached-Event-Calculus implementieren. Zweitens aus generellen WS-CDL-Axiomen, welche die Effekte auf Basisaktivitäten und die indirekten Effekte auf komplexe Aktivitäten definieren. Drittens aus einer spezifisch auf die Choreographie abgestimmte Beschreibung der Choreographie-Struktur, welche durch die WS-CDL-Axiome benutzt werden kann. Der vierte Teil der Wissensbasis besteht aus dem Ereignisprotokoll, welches die Fakten zu jedem eingetretenen Ereignis enthält. Abbildung 4.19 zeigt die Struktur im Detail, welche in den folgenden Abschnitten weiter detailliert wird.

4.3.1. Fakten über Ereignisse und Choreographie-Struktur

Das Hauptereignis einer Choreographie ist eine Web-Services-Nachricht, welche von einem Teilnehmer einer Choreographie an einen anderen gesendet wird. Dies kann entweder eine

Anfrage oder eine Antwort auf eine Anfrage sein. Mit Unterstützung des Korrelationsmechanismus aus Kapitel 4.1.2 kann so ein Ereignis direkt der WS-CDL-Beschreibung zugeordnet werden. Damit ein solches Ereignis mit dem Ereigniskalkül verarbeitet werden kann, muss aber für jedes Ereignis ein Fakt in die Wissensbasis eingefügt werden.

Eine generelle Repräsentation eines Ereignisses innerhalb einer Choreographie lässt sich wie folgt ableiten:

$$Happens(Event(x, id), t) \tag{4.14}$$

Das Prädikat *Happens* ist aus dem Ereigniskalkül bekannt und beschreibt, dass ein Ereignis zur Zeit t eingetreten ist. Zusätzlich dazu wird das Prädikat *Event* benötigt, um ein Ereignis x einer Instanz id einer Choreographie zuzuordnen. Die möglichen Ereignistypen für die Variable x werden durch die WS-CDL-Spezifikation vorgegeben und sind eigentlich die in WS-CDL als Basisaktivitäten definierten Aktivitäten `Interaction`, `NoAction`, `SilentAction` und `Assign`. Allerdings besteht eine Interaktion aus mehreren `Exchange`-Elementen, weshalb in diesem Model das Prädikat *Exchange* einen Teil einer Interaktion als Basisaktivität modelliert. Interaktionen werden erst später wieder als komplexe Aktivitäten weiter betrachtet. Im weiteren Verlauf der Arbeit wird also abweichend von der WS-CDL-Spezifikation eine *Basisaktivität* immer eine atomare Aktivität bezeichnen.

Eine einfache Nachricht, welche von einem Web-Service oder einer Anwendung an einen anderen Web-Service gesendet wird, ist hauptsächlich durch die Rollen der Teilnehmer und den Namen der Nachricht charakterisiert:

$$Exchange(name, fromRole, toRole) \tag{4.15}$$

Mit Unterstützung der bisher vorgestellten Prädikate lässt sich ein einfacher Austausch von Nachrichten innerhalb der Wissensbasis als Fakten repräsentieren. Diese Fakten können dann später durch die Regeln und Axiome des Ereigniskalküls weiter benutzt werden. Die anderen Basisaktivitäten von WS-CDL wie `NoAction`, `SilentAction` und `Assign` sind nicht beobachtbare Ereignisse, welche daher nicht Teil der Ereignisfaktenbasis sind.

Zusätzlich zu den Ereignisfakten werden für die Verifikation die Fakten über die Choreographie-Struktur benötigt, damit die WS-CDL-Axiome auf Basis der Struktur Entscheidungen treffen können. Diese Strukturfakten werden genau wie in dem ersten bereits vorgestellten regelbasierten Ansatz abgebildet (siehe Abschnitt 4.2.1.1). Für die Basisaktivitäten werden daher die Prädikate $IsExchange(a)$, $IsSilentAction(a)$, $IsNoAction(a)$ und $IsAssign(a)$ zur Auszeichnung genutzt und die Prädikate *Exchange*, *NoAction*, *SilentAction* und *Assign* zur Beschreibung der einzelnen Basisaktivitäten eingesetzt. Die komplexen Aktivitäten werden genauso wie in Abschnitt 4.2.1.1 beschrieben abgebildet. Die gleiche Struktur wird auch hier eingesetzt, da sich mithilfe der Struktur die Zuständigkeitskette direkt abbilden und damit für das Ereigniskalkül nutzen lässt. Zusätzlich wird zur Vereinfachung bei den komplexen Aktivitäten ein $Root(x)$-Prädikat eingeführt, welches eine Aktivität x als das Wurzel-

element einer Choreographie auszeichnet.

Ein weiterer wichtiger Punkt bei der späteren Verifikation sind nicht beobachtbare Aktivitäten wie `NoAction`, `SilentAction` und `Assign`. Tritt etwa ein Ereignis auf, was innerhalb einer Sequenz nach dieser Aktivität auftreten soll, dann muss davon ausgegangen werden, dass auch die nicht beobachtbare Aktion ausgeführt wurde. Nachprüfbar ist dies allerdings mit den vorgestellten Mitteln der Überwachung der Interaktionen nicht. Um mit diesen Aktivitäten über das Ereigniskalkül umzugehen, wird eine neue Eigenschaft *überspringbar* für Aktivitäten eingeführt. Nicht beobachtbare Basisaktivitäten sind dabei *strukturell* bei der Verifikation auslassbar, da sie nicht direkt einem beobachtbaren Ereignis zugeordnet werden können. Für diesen Fall wird für die Choreographie-Struktur zusätzlich der Fakt $IsOmittable(a)$ zur Faktenbasis hinzugefügt, welcher beschreibt, dass eine Aktivitität a überspringbar ist.

Mit Unterstützung dieser Prädikate lässt sich die Struktur einer Choreographie vollständig abbilden. Diese Struktur und die vorgestellten Prädikate werden im nächsten Schritt benutzt, um Regeln für die Effekte von Ereignissen auf die Fluents der jeweiligen Aktivitäten abzuleiten.

4.3.2. WS-CDL Axiome

Bisher sind das Fluent-Model, die Repräsentation von Ereignissen und die Beschreibung der Choreographie-Struktur für die Wissensbasis eingeführt. Es fehlen noch die Regeln zur Beschreibung der direkten und indirekten Effekte auf Fluents einzelner Aktivitäten. Diese Regeln sind der Kern der Verifikation und werden getrennt für Basisaktivitäten und komplexe Aktivitäten eingeführt.

4.3.2.1. Basisaktivitäten

Für Basisaktivitäten lassen sich nun Effekt-Axiome definieren. Da beobachtbare Basisaktivitäten selbst nie fehlerhaft sind, wird das Fluent *Valid* sofort wahr, nachdem das entsprechende Ereignis für diese Basisaktivität beobachtet wird (Axiom 4.16). Basisaktivitäten sind atomar und daher wird das Fluent *Complete* sofort wahr, nachdem das Ereignis eingetreten ist (Axiom 4.17).

$$Initiates(Event(x,id),Valid(x,id),t) \leftarrow IsExchange(x) \tag{4.16}$$

$$Initiates(Event(x,id),Complete(x,id),t) \leftarrow IsExchange(x) \tag{4.17}$$

Bei überspringbaren Basisaktivitäten wie `SilentAction`, `NoAction` oder `Assign` gilt die Annahme, dass diese Aktivitäten unabhängig von Ereignissen immer abgeschlossen sind. Der Grund liegt in der Natur der Aktivitäten. Beispielsweise beschreibt eine `SilentAction` eine Aktivität, die zwar durchgeführt wird, aber nicht beobachtet werden kann. Es kann also davon ausgegangen werden, dass die Aktivität bei Eintreffen einer nachfolgenden Nach-

richt stattgefunden hat. Damit diese nachfolgende Nachricht allerdings richtig verifiziert werden kann, muss die überspringbare Aktivität ausgelassen werden. Die ist einfach zu erreichen, in dem die auslassbare Aktivität als beendet angesehen wird, wie später bei den komplexen Aktivitäten wie Sequenzen und deren Umgang mit Teilaktivitäten noch genauer zu sehen ist. Dieses Verhalten, dass überspringbare Basisaktivitäten immer als beendet betrachtet werden, beschreibt auch das Axiom 4.18.

$$HoldsAt(Complete(node, \_), \_) \leftarrow IsOmittable(node) \tag{4.18}$$

Zusätzlich zur Unterscheidung in normale und auslassbare Basisaktivitäten und deren Auswirkungen auf Fluents definiert die `Assign`-Aktivität das Setzen von Variablen. Die `Assign`-Aktivität verhält sich dabei in Bezug auf die Zuständigkeitskette genauso wie jede andere überspringbare Aktivität, allerdings muss zusätzlich noch mit dem Setzen der Variablen umgegangen werden. Die dafür notwendigen Axiome werden allerdings erst in Abschnitt 4.3.2.3 eingeführt, da hierfür das Zusammenspiel der `Assign`-Aktivität mit den komplexen Aktivitäten erst genauer erläutert werden muss. Die komplexen Aktivitäten werden jetzt im folgenden Abschnitt detaillierter beschrieben.

4.3.2.2. Zusammengesetzte komplexe Aktivitäten

Komplexe zusammengesetzte Aktivitäten sind ordnende Strukturen, welche das Verhalten ihrer Teilaktivitäten und den Kontrollfluss definieren. Das Verhalten der Teilaktivitäten und deren indirekte Effekte bestimmen den Zeitverlauf der Fluents einer komplexen Aktivität. Zur Vereinfachung mit dem Umgang von komplexen Aktivitäten wird die Eigenschaft *IsComplex* eingeführt, welche wahr wird, wenn eine Aktivität entweder eine Sequenz, eine Auswahl, eine Parallelaktivität oder eine WorkUnit ist.

Genau wie bei den Basisaktivitäten ist auch bei den komplexen Aktivitäten die Eigenschaft *überspringbar* wichtig. Zusätzlich zu den *strukturell überspringbaren* Aktivitäten lassen sich mithilfe der WorkUnit auch *zeitweise überspringbare* Aktivitäten definieren (siehe auch Abschnitt 4.3.2.4 zur WorkUnit), wenn der Guard zu einer bestimmten Zeit ungültig und die WorkUnit nicht blockierend ist. Für den Fall von zeitweise überspringbaren Aktivitäten reicht ein Choreographie-Fakt wie bei den Basisaktivitäten nicht mehr aus. In diesem Fall wird das Fluent $Omittable(node, id)$ eingeführt, welches anzeigt, ob die Aktivität *node* zur Zeit t überspringbar ist oder nicht. Für strukturell überspringbare Aktivitäten ist dieses Fluent wie in Axiom 4.19 natürlich in jeder Instanz und zu jeder Zeit wahr.

$$HoldsAt(Omittable(node, \_), \_) \leftarrow IsOmittable(node) \tag{4.19}$$

Nicht nur WorkUnits können übersprungen werden, sondern auch die drei komplexen Aktivitäten wie `Sequence`, `Parallel` und `Choice`, wenn sie zu einer bestimmten Zeit nur aus überspringbaren Aktivitäten bestehen. Axiom 4.20 beschreibt dieses, in dem geprüft

wird, ob für eine komplexe Aktivität keine Teilaktivität existiert, die nicht überspringbar ist.

$$\begin{aligned} &HoldsAt(Omittable(node, id), t) \leftarrow \\ &\quad IsComplex(node) \wedge \neg IsWorkUnit(node) \wedge \\ &\quad \neg(GetChild(node, child) \wedge \neg(HoldsAt(Omittable(child, id), t))) \end{aligned} \tag{4.20}$$

Ein Sonderfall bei überspringbaren komplexen Aktivitäten ergibt sich über Auswahlaktivitäten mit einer `NoAction` als Teilaktivität. Im Unterschied zu den anderen für die Verifikation auslassbaren Aktivitäten können diese *optionalen Aktivitäten* ausgeführt werden, müssen es aber nicht. Die Verifikation muss auch an dieser Stelle eine optionale Aktivität überspringen, falls ein der optionalen Aktivität nachfolgendes Ereignis eintrifft. Die Entscheidung eine optionale Aktivität zu überspringen, bleibt, wie später zu sehen, die gleiche wie bei den anderen überspringbaren Aktivitäten. Trotzdem ist die leicht andere Semantik in der Ausführung hervorzuheben und bei der Modellierung zu beachten.

Weiterhin ergibt sich durch die Zuständigkeitskette die generelle Fehlererkennung, in dem an der Wurzelaktivität geprüft wird, ob ein eingehendes Ereignis für diese Aktivität das Fluent *Valid* initiiert. Ist dies nicht der Fall, ist die Nachricht nicht valide und damit die Choreographie fehlerhaft. Dieser Fall kann wiederum durch ein Fluent $Error(Event(x, id))$ kenntlich gemacht werden, wie es in Axiom 4.21 gezeigt ist. Dieses Fluent macht allerdings nur beim Wurzelknoten Sinn, da die Teilknoten des Aktivitätsbaumes nicht den Gesamtüberblick haben und daher einen Fehler nicht direkt feststellen können. Trotzdem muss das Fluent *Valid* auch bei unbeendeten Teilaktivitäten im Fehlerfall terminiert werden, wenn ein unerwartetes Ereignis auftritt. Damit wird beispielsweise angezeigt, dass diese Aktivität erst beendet werden muss, damit ein nachfolgendes Ereignis eintreten kann. Das Fluent wird daher terminiert, wenn ein Ereignis auftritt, welches die Aktivität nicht validiert (siehe Axiom 4.22).

$$\begin{aligned} &Initiates(Event(x, id), Error(Event(x, id)), t) \leftarrow \\ &\quad Root(node) \wedge \neg Initiates(Event(x, id), Valid(node, id), t) \end{aligned} \tag{4.21}$$

$$\begin{aligned} &Terminates(Event(x, id), Valid(node, id), t) \leftarrow \\ &\quad IsComplex(node) \wedge \neg HoldsAt(Complete(node, id), t) \wedge \\ &\quad \neg Initiates(Event(x, id), Valid(node, id), t) \wedge \\ &\quad Initiates(Event(x, id), Valid(node_2, id), t) \wedge \neg InParallel(node, node_2) \end{aligned} \tag{4.22}$$

Im Folgenden werden die Axiome der komplexen Aktivitätstypen genauer vorgestellt. Insbesondere werden dabei überspringbare Aktivitäten eine größere Rolle spielen. Die Axiome lassen sich für alle Aktivitätstypen grob in zwei Kategorien einteilen: Zum einen in Axiome, die den normalen Verlauf ohne (zeitweise) überspringbare Aktivitäten beschreiben, und zum anderen in Axiome, welche die Sonderfälle bei überspringbaren Aktivitäten definieren.

Zusätzlich kommen später bei der WorkUnit noch die Axiome für den Schleifendurchlauf hinzu.

Sequence: Eine Sequenz enthält eine oder mehrere Teilaktivitäten. Dabei muss jede Teilaktivität sequenziell aktiviert und beendet werden, und zwar genau in ihrer definierten Reihenfolge. Eine Sequenz ist der Parallelaktivität sehr ähnlich, allerdings werden zusätzlich noch Überprüfungen benötigt, welche die Reihenfolge der Aktivierung von Teilaktivitäten überprüft. Generell gilt die Sequenz als valide, wenn für die aktuell aktive Teilaktivität der Sequenz ein Ereignis valide ist.

$$\begin{aligned} &Initiates(Event(x,id),Valid(node,id),t) \leftarrow \\ &\quad IsSequence(node) \wedge \neg HoldsAt(Complete(node,id),t) \wedge \\ &\quad SequenceStepActive(node,id,t,current) \wedge \\ &\quad Initiates(event(x,id),Valid(current,id),t) \end{aligned} \tag{4.23}$$

Um die Regeln für die Sequenz etwas zu vereinfachen, wird eine Regel für das Prädikat $SequenceStepActive(node,id,t,current)$ eingeführt, welches die aktuell aktive Teilaktivität der Sequenz für die laufende Instanz zu einer bestimmten Zeit ermittelt. Die Regel durchsucht dabei, wie in Abbildung 4.20 zu sehen, die Teilaktivitäten der Sequenz und liefert die erste Teilaktivität *current* zurück, für die das Fluent *Complete* oder *Omittable* zum Zeitpunkt t nicht gilt. Damit lässt sich einfach eine aktive Sequenz validieren, indem geprüft wird, ob die aktuelle Teilaktivität durch das gerade beobachtete Ereignis validiert wird. Mit diesem Vorgehen lässt sich auch die Reihenfolge der Sequenz prüfen. Wenn die Teilaktivitäten nicht in der vorgesehenen Reihenfolge abschließen und beispielsweise ein späterer Schritt beendet wird, dann wird der aktive Schritt nicht validiert und die Sequenz ist fehlerhaft. Axiom 4.23 zeigt das Verhalten einer noch nicht beendeten Sequenz bezüglich des Fluents *Valid*.

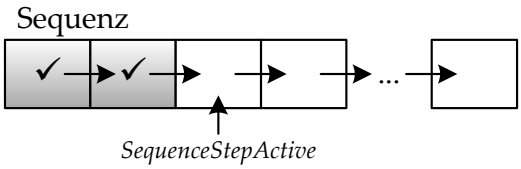


Abbildung 4.20.: Sequenz schematisch

Das Beenden einer Sequenz erscheint intuitiv, denn eine Sequenz ist im Normalfall beendet, wenn der letzte Teilschritt beendet wird. Dabei muss allerdings noch die Unterscheidung zwischen normalen und überspringbaren Teilaktivitäten getroffen werden. Überspringbare Teilaktivitäten bis zum aktiven Teilschritt der Sequenz werden durch die Regel *SequenceStepActive* direkt übersprungen und sind zu diesem Zeitpunkt unproblematisch. Problematisch sind diese Aktivitäten allerdings am Ende einer Sequenz, da zur Entscheidung, ob eine Sequenz abgeschlossen ist, die letzte Teilaktivität beendet sein muss. Hier

gibt es allerdings zwei spezielle Fälle zu unterscheiden. Zum einen den Fall, dass nach dem aktiven Schritt zu einem Zeitpunkt alle nachfolgenden Teilaktivitäten *Omittable* sind. In diesem Fall könnte die Sequenz direkt durch das Ereignis beendet werden. Zum anderen kann der zweite Fall eintreten, indem durch zeitweise überspringbare Aktivitäten bei Eintreffen eines weiteren Ereignisses alle übrig gebliebenen Teilaktivitäten einer noch nicht beendeten Sequenz überspringbar sind. Es gibt in diesem Fall kein Ereignis, welches direkt auf die Sequenz angewendet werden kann. Die in diesem Absatz vorgestellten Fälle zum Beenden einer Sequenz werden jetzt im Einzelnen weiter vertieft.

Den normalen Fall, dass die letzte Teilaktivität durch ein Ereignis beendet wird, beschreibt Axiom 4.24. Dieser Fall muss nur überprüft werden, wenn es keine auslassbaren (*containsOmittable*) Teilaktivitäten mehr in der Sequenz gibt und der aktive Teilschritt zum einen der letzte Schritt ist und zum anderen durch ein Ereignis valide beendet wird.

$$\begin{aligned} &Initiates(Event(x,id),Complete(node,id),t) \leftarrow \\ &\quad IsSequence(node) \wedge \neg ContainsOmittable(node,id,t)) \wedge \\ &\quad SequenceStepActive(node,id,t,current) \wedge \\ &\quad MemberActivities(node,members) \wedge Last(current,members) \wedge \\ &\quad Initiates(Event(x,id),Valid(current,id),t) \wedge \\ &\quad Initiates(Event(x,id),Complete(current,id),t) \end{aligned} \tag{4.24}$$

Die beiden Sonderfälle, in denen nur noch überspringbare Aktivitäten in der Sequenz vorkommen, lassen sich mit einer Regel entsprechend beenden. In beiden Fällen gilt, dass eine Sequenz beendet werden kann, wenn zum einen die Sequenz nur beendete und überspringbare Aktivitäten enthält und zum anderen eine weitere Basisaktivität durch ein *nachfolgendes* Ereignis außerhalb der Sequenz validiert und beendet wird. In diesem Fall muss allerdings auf Parallelaktivitäten geachtet werden, die über der Sequenz als Vateraktivität liegen. Ist die Sequenz im Geltungsbereich einer Parallelaktivität, dann ist ein Ereignis, welches innerhalb dieser Parallelaktivität validiert, kein nachfolgendes Ereignis, sondern ein Ereignis, welches parallel erfolgt. Die Sequenz darf also noch nicht beendet werden, da beispielsweise der letzte Schritt optional sein kann und eventuell zu einem späteren Zeitpunkt noch

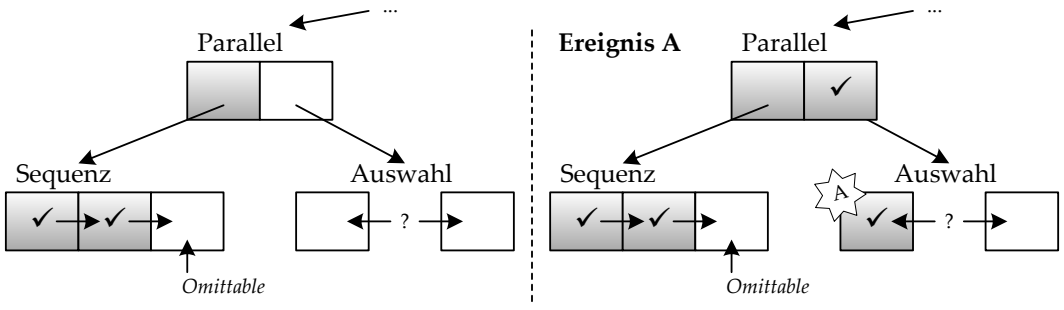


Abbildung 4.21.: Sequenz mit überspringbaren Aktivitäten als Teil von Parallel

ausgeführt werden kann oder muss. Abbildung 4.21 illustriert das Vorgehen, was durch Axiom 4.25 ausgedrückt wird. Das Axiom prüft, ob ein Ereignis eine Aktivität $node_2$ validiert, welche außerhalb des Unterbaumes der Sequenz ($\neg GetChildren(node, node_2)$) und außerhalb des Geltungsbereichs einer Parallelaktivität ($\neg InParallel(node, node_2)$) liegt. Zusätzlich müssen alle Aktivitäten der abzuschließenden Sequenz beendet oder überspringbar sein ($SequenceCompleted(node, id, t)$).

$$\begin{aligned} &Initiates(Event(x, id), Complete(node, id), t) \leftarrow \\ &\quad IsSequence(node) \wedge \neg HoldsAt(Complete(node, id), t) \wedge \\ &\quad ContainsOmittable(node, id, t) \wedge IsExchange(node_2) \wedge \neg(node_2 = node) \wedge \\ &\quad Initiates(Event(x, id), Valid(node_2, id), t) \wedge \neg InParallel(node, node_2) \wedge \\ &\quad \neg GetChildren(node, node_2) \wedge SequenceCompleted(node, id, t) \end{aligned} \tag{4.25}$$

Als problematisch erweist sich, wenn die so beendete Sequenz wieder Teil einer anderen komplexen Aktivität ist. Wenn etwa diese Sequenz Teil einer anderen Sequenz ist, dann muss die darüberliegende Sequenz die auf diese Weise beendete Sequenz überspringen, um die darauf nachfolgende Teilaktivität zu verifizieren. Damit die darüberliegende Aktivität dieses erkennt, wird temporär zu diesem Zeitpunkt das Fluent *Omittable* gesetzt und die Vateraktivität kann darauf reagieren, wie in Abbildung 4.22 illustriert. Dieses Vorgehen wird in Axiom 4.26 ausgedrückt, welches zu den gleichen Bedingungen wie zum Beenden der Sequenz wahr wird. Die Vateraktivität kann jetzt im Detail entscheiden, ob dies relevant ist oder nicht.

$$\begin{aligned} &HoldsAt(Omittable(node, id), t) \leftarrow \\ &\quad IsSequence(node) \wedge \neg HoldsAt(Complete(node, id), t) \wedge \\ &\quad ContainsOmittable(node, id, t) \wedge IsExchange(node_2) \wedge \neg(node_2 = node) \wedge \\ &\quad Initiates(Event(x, id), Valid(node_2, id), t) \wedge \neg InParallel(node, node_2) \wedge \\ &\quad \neg GetChildren(node, node_2) \wedge SequenceCompleted(node, id, t) \end{aligned} \tag{4.26}$$

Weiterhin stellen zeitweise überspringbare Teilaktivitäten, welche nicht beendet wurden, ein

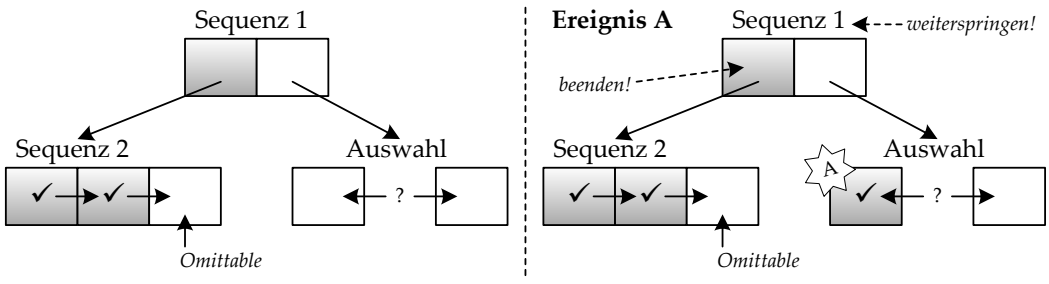


Abbildung 4.22.: Sequenz mit überspringbaren Aktivitäten

Problem dar. Ist die übersprungene Teilaktivität zu einem späteren Zeitpunkt nicht mehr als überspringbar (beispielsweise durch Axiom 4.20) markiert, kann sie durch *SequenceStepActive* als aktiver Knoten herangezogen werden, obwohl die Sequenz vielleicht schon längst einen Schritt weiter ist. In diesem Fall muss sichergestellt werden, dass übersprungene Aktivitäten innerhalb einer Sequenz immer beendet werden. Axiom 4.27 stellt dies sicher, in dem es prüft, ob bei einer noch nicht beendeten Sequenz eine validierte Teilaktivität nach einer überspringbaren Teilaktivität liegt ($Forerunner(node, current)$).

$$\begin{aligned} &Initiates(Event(x,id), Complete(node,id), t) \leftarrow \\ &\quad HoldsAt(Omittable(node,id), t) \wedge GetParent(node, parent) \wedge \\ &\quad IsSequence(parent) \wedge \neg HoldsAt(Complete(parent,id), t) \wedge \\ &\quad SequenceStepActive(parent, id, t, current) \wedge \\ &\quad Initiates(Event(x,id), Valid(current,id), t) \wedge \\ &\quad Forerunner(node, current) \end{aligned} \tag{4.27}$$

Damit ist zum Beenden einer Sequenz zum einen der normale Fall ohne überspringbare Teilaktivitäten abgedeckt (Axiom 4.24) und zum anderen alle Fälle mit überspringbaren Teilaktivitäten am Ende der Sequenz (Axiom 4.25) sowie als Zwischenaktivitäten (Axiom 4.27). Eine mögliche Vateraktivität kann auf eine durch überspringbare Teilaktivitäten beendete Sequenz reagieren, da ein spezielles Flag *Omittable* in diesem Fall gesetzt wird (Axiom 4.26). Eine Sequenz wird valide, wenn ein Ereignis den aktuellen Schritt der Sequenz validiert (Axiom 4.23). Fehler für eine Sequenz werden erkannt, wenn ein Ereignis das Fluent *Valid* für die aktive Teilaktivität nicht initiiert und damit auch das Fluent *Valid* für die Sequenz nicht initiiert wird. Der Fehler wird in diesem Fall über dieses Fluent weiter in der Zuständigkeitskette nach oben durchgereicht.

Parallel: Eine `Parallel`-Aktivität enthält eine oder mehrere Teilaktivitäten, die gleichzeitig gestartet aber nicht zur selben Zeit beendet werden müssen. Jede der Teilaktivität darf nur exakt einmal ausgeführt werden. Auf die Laufzeitumgebung übertragen bedeutet diese Feststellung, dass die Teilaktivitäten in beliebiger Reihenfolge ausgeführt werden dürfen. Generell gilt also für die Zuständigkeitskette, dass eine `Parallel`-Aktivität für ein Ereignis valide ist, wenn für eine noch nicht beendete Teilaktivität das Ereignis auch valide ist. Als Axiom ausgedrückt, muss eine Teilaktivität ($GetChild(node, child)$) existieren, welche zum einen noch nicht beendet ist und zum anderen durch das aktuell beobachtete Ereignis zur Zeit t validiert wird. Axiom 4.28 drückt dieses Verhalten bezüglich des Fluents *Valid* aus.

$$\begin{aligned} &Initiates(Event(x,id), Valid(node,id), t) \leftarrow \\ &\quad IsParallel(node) \wedge \neg HoldsAt(Complete(node,id), t) \wedge GetChild(node, child) \wedge \\ &\quad \neg HoldsAt(Complete(child,id), t) \wedge Initiates(Event(x,id), Valid(child,id), t) \end{aligned} \tag{4.28}$$

Eine `Parallel`-Aktivität ist beendet, wenn die letzte noch nicht beendete Teilaktivität beendet wird. Um festzustellen, ob bisher alle Teilaktivitäten bis auf eine erfolgreich beendet worden sind, wird auch hier eine Hilfsregel *ParallelNearCompletion* eingeführt. Diese Hilfsregel liefert „wahr“ zurück, wenn alle anderen Teilaktivitäten bis auf *child* für die Instanz *id* zur Zeit t erfolgreich beendet oder überspringbar sind. Ist die `Parallel`-Aktivität also kurz vor der Beendigung, kann geprüft werden, ob ein Ereignis zur Zeit t für die letzte Teilaktivität zum einen das Fluent *Valid* und zum anderen das Fluent *Complete* initiiert. Ist dies der Fall, dann ist auch die Parallelaktivität erfolgreich beendet, wie in Axiom 4.29 spezifiziert.

$$\begin{aligned}
&Initiates(Event(x,id),Complete(node,id),t) \leftarrow \\
&\quad IsParallel(node) \wedge GetChild(node,child) \wedge \neg holdsAt(Complete(child,id),t) \wedge \\
&\quad Initiates(Event(x,id),Valid(child,id),t) \wedge \\
&\quad Initiates(Event(x,id),Complete(child,id),t) \wedge \\
&\quad ParallelNearCompletion(node,id,child,t)
\end{aligned} \tag{4.29}$$

Wie auch bei der Sequenz existiert durch zeitweise überspringbare Aktivitäten das Problem, dass eine `Parallel`-Aktivität noch nicht beendet ist, aber dennoch zu einem Zeitpunkt nur noch aus überspringbaren Aktivitäten besteht. Auch hier wird das gleiche Vorgehen wie bei der Sequenz angewendet, in dem geprüft wird, ob ein der `Parallel`-Aktivität nachfolgendes Ereignis eintritt, wie in Axiom 4.30 beschrieben.

$$\begin{aligned}
&Initiates(Event(x,id),Complete(node,id),t) \leftarrow \\
&\quad isParallel(node) \wedge \neg HoldsAt(Complete(node,id),t) \wedge \\
&\quad ContainsOmittable(node,id,t) \wedge IsExchange(node_2) \wedge \neg(node_2 = node) \wedge \\
&\quad Initiates(Event(x,id),Valid(node_2,id),t) \wedge \neg InParallel(node,node_2) \wedge \\
&\quad \neg GetChildren(node,node_2) \wedge ParallelCompleted(node,id,t)
\end{aligned} \tag{4.30}$$

Genauso wie bei der Sequenz muss auch hier der Vateraktivität mitgeteilt werden, dass diese `Parallel`-Aktivität durch überspringbare Aktivitäten beendet wurde. Auch in diesem Fall gelten die gleichen Bedingungen wie bei der Sequenz, wie in Axiom 4.31 spezifiziert.

$$\begin{aligned}
&HoldsAt(Omittable(node,id),t) \leftarrow \\
&\quad isParallel(node) \wedge \neg HoldsAt(Complete(node,id),t) \wedge \\
&\quad ContainsOmittable(node,id,t) \wedge IsExchange(node_2) \wedge \neg(node_2 = node) \wedge \\
&\quad Initiates(Event(x,id),Valid(node_2,id),t) \wedge \neg InParallel(node,node_2) \wedge \\
&\quad \neg GetChildren(node,node_2) \wedge ParallelCompleted(node,id,t)
\end{aligned} \tag{4.31}$$

Damit sind zum Beenden dieser Art von Aktivität zum einen der normale Fall abgedeckt (Axiom 4.29) und zum anderen alle Fälle mit überspringbaren Teilaktivitäten beim Beenden

der Aktivität (Axiom 4.30). Ein Problem mit überspringbaren Zwischenaktivitäten wie bei der Sequenz gibt es in diesem Fall nicht, da die Reihenfolge der Teilaktivitäten hier keine Rolle spielt. Eine mögliche Vateraktivität kann auf eine durch überspringbare Teilaktivitäten beendete `Parallel`-Aktivität reagieren, da ein spezielles Flag *Omittable* auch in diesem Fall gesetzt wird (Axiom 4.31). Eine `Parallel`-Aktivität wird valide, wenn ein Ereignis eine Teilaktivität entsprechend validiert (Axiom 4.28). Fehler für eine `Parallel`-Aktivität werden erkannt, wenn ein Ereignis das Fluent *Valid* nicht für eine unbeendete Teilaktivität initiiert und damit auch das Fluent *Valid* für die `Parallel`-Aktivität nicht initiiert wird. Der Fehler wird in diesem Fall über dieses Fluent weiter in der Zuständigkeitskette nach oben durchgereicht.

Choice: Die `Choice`-Aktivität enthält eine oder mehrere Teilaktivitäten, von denen exakt eine ausgeführt wird. Im Detail bedeutet dies, dass Ereignisse im Kontext der Auswahl immer zur ausgewählten Teilaktivität gehören, damit diese erfolgreich beenden kann. Eine Teilaktivität wird ausgewählt, wenn ein Ereignis das Fluent *Valid* einer Teilaktivität wahr werden lässt.

Zur Vereinfachung der `Choice`-Aktivität wird die Regel *ChoiceStepActive* eingeführt, die ermittelt, ob eine beobachtbare Teilaktivität in einem Schritt vorher schon valide gewesen ist. Diese Teilaktivität wird dann zurückgeliefert. Ist bereits eine Teilaktivität ausgewählt, muss nur geprüft werden, ob diese mit dem neuen Ereignis validiert. Ist noch keine Teilaktivität ausgewählt, dann ist die Auswahl nur dann valide, wenn es eine beobachtbare Teilaktivität ($GetChild(node, child)$) gibt, die mit dem Ereignis zu diesem Zeitpunkt valide ist. Diese wird dann ausgewählt und weiter für die Validierung genutzt. Axiom 4.32 zeigt die Regel für das Fluent *Valid*.

$$\begin{aligned}
&Initiates(Event(x, id), Valid(node, id), t) \leftarrow \\
&\quad IsChoice(node) \wedge \neg HoldsAt(Complete(node, id), t) \wedge \\
&\quad ((ChoiceStepActive(node, id, t, current) \wedge \\
&\quad\ Initiates(Event(x, id), Valid(current, id), t) \\
&\quad) \vee (ChoiceStepActive(node, id, t, current) \wedge \\
&\quad\ GetChild(node, current) \wedge \neg IsOmittable(current) \wedge \\
&\quad\ Initiates(Event(x, id), Valid(current, id), t)))
\end{aligned} \tag{4.32}$$

Eine `Choice`-Aktivität ist beendet, wenn die ausgewählte Teilaktivität beendet ist. Falls kein Knoten ausgewählt ist, wird geprüft, ob eine Teilaktivität der Auswahl direkt mit einem Ereignis beendet wird. Das Verhalten entspricht fast dem Axiom 4.32, mit dem Unterschied,

dass ein Ereignis eine betreffende Teilaktivität auch abschließen muss; siehe Axiom 4.33.

$$\begin{aligned}
&Initiates(Event(x,id),Complete(node,id),t) \leftarrow \\
&\quad IsChoice(node) \wedge ((ChoiceStepActive(node,id,t,current) \wedge \\
&\quad\ Initiates(Event(x,id),Complete(current,id),t) \\
&\quad) \vee (\neg choiceStepActive(node,id,t,current) \wedge \\
&\quad\ GetChild(node,current) \wedge \neg IsOmittable(current) \wedge \\
&\quad\ Initiates(Event(x,id),Complete(current,id),t))) \qquad (4.33)
\end{aligned}$$

Ein sehr wichtiger Punkt bei Auswahlaktivitäten sind überspringbare Aktivitäten. Eine noch nicht beendete Auswahl wird dabei zu einem Zeitpunkt eines Ereignisses selbst überspringbar, wenn sie zum einen überspringbare Teilaktivitäten enthält, sie noch nicht betreten wurde und keine der Teilaktivitäten mit einem Ereignis validiert. Eine Vateraktivität der Auswahl kann dann entscheiden, ob es relevant ist, dass die Auswahl zu einem Zeitpunkt überspringbar ist. Axiom 4.34 drückt genau diesen Zusammenhang aus, welcher auch in Abbildung 4.23 skizziert ist.

$$\begin{aligned}
&HoldsAt(Omittable(node,id),t) \leftarrow \\
&\quad isChoice(node) \wedge ContainsOmittable(node,id,t) \wedge \\
&\quad \neg HoldsAt(Complete(node,id),t) \wedge \neg HoldsAt(Valid(node,id),t) \wedge \\
&\quad \neg Initiates(Event(X,id),Valid(node,id),t) \qquad (4.34)
\end{aligned}$$

Eine Auswahl kann indirekt auch durch die überspringbaren Teilaktivitäten beendet werden. Dies ist beispielsweise als Teil einer Sequenz der Fall, wenn die Auswahl durch Axiom 4.34 übersprungen wird und damit die Auswahl durch Axiom 4.27 auch beendet wird. Genauso wird eine noch nicht betretene Auswahl beendet (falls nicht schon geschehen), wenn die Vateraktivität einer Auswahl beendet wird. Dies ist vor allem relevant, wenn mögliche Assign-Teilaktivitäten der Auswahl noch bearbeitet werden sollen. Dieses Verhalten wird

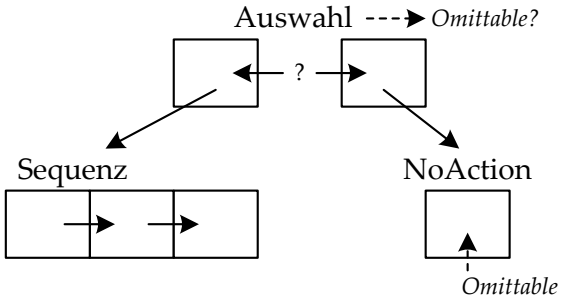


Abbildung 4.23.: Auswahl mit überspringbaren Aktivitäten

auch durch Axiom 4.35 definiert.

$$\begin{aligned}
&Initiates(Event(x,id),Complete(node,id),t) \leftarrow \\
&\quad isChoice(node) \wedge ContainsOmittable(node,id,t) \wedge \\
&\quad \neg HoldsAt(Complete(node,id),t) \wedge \neg HoldsAt(Valid(node,id),t) \wedge \\
&\quad HoldsAt(Omittable(node,id),t) \wedge GetParent(node,parent) \wedge \\
&\quad Initiates(Event(x,id),Complete(parent,id),t)
\end{aligned} \tag{4.35}$$

Weiterhin muss eine durch überspringbare Aktivitäten beendete Auswahl diesen Fakt auch an einen Vaterknoten weiterreichen, damit dieser mit dem Nachfolger der Auswahl weitermachen kann. Dies entspricht dem gleichen Vorgehen wie bei Parallel und Sequenz, in dem dieser Fakt dem Vaterknoten über das Fluent *Omittable* mitgeteilt wird. Axiom 4.36 setzt in diesem Fall das Fluent. Die Bedingungen entsprechen denen zum Beenden einer Auswahl durch eine überspringbare Aktivität.

$$\begin{aligned}
&HoldsAt(Omittable(node,id),t) \leftarrow \\
&\quad isChoice(node) \wedge ContainsOmittable(node,id,t) \wedge \\
&\quad \neg HoldsAt(Complete(node,id),t) \wedge ((ChoiceStepActive(node,id,t,current) \wedge \\
&\quad\ Initiates(Event(x,id),Complete(current,id),t) \wedge \\
&\quad\ HoldsAt(Omittable(current,id),t) \\
&\quad) \vee (\neg choiceStepActive(node,id,t,current) \wedge \\
&\quad\ GetChild(node,current) \wedge Initiates(Event(x,id),Complete(current,id),t) \wedge \\
&\quad\ HoldsAt(Omittable(current,id),t)))
\end{aligned} \tag{4.36}$$

Damit ist zum Beenden einer Auswahl zum einen der normale Fall abgedeckt (Axiom 4.33) und zum anderen alle Fälle mit überspringbaren Teilaktivitäten (Axiom 4.35 und 4.27). Eine mögliche Vateraktivität kann auf überspringbare Auswahlaktivitäten reagieren, wenn das Fluent *Omittable* gesetzt wird (Axiom 4.34). Eine Auswahl wird valide, wenn ein Ereignis eine ausgewählte Teilaktivität entsprechend validiert (Axiom 4.32). Fehler für eine Auswahl werden erkannt, wenn ein Ereignis das Fluent *Valid* nicht für die aktive oder eine der Teilaktivitäten initiiert (Falls die Auswahl noch nicht betreten wurde) und damit auch das Fluent *Valid* für die Auswahl nicht initiiert wird. Der Fehler wird in diesem Fall über dieses Fluent weiter in der Zuständigkeitskette nach oben durchgereicht.

4.3.2.3. Variablenzuweisung durch Assign

Die Axiome für Basis- und komplexe Aktivitäten (außer der WorkUnit) sind damit eingeführt. Allerdings fehlt für die Konstruktion von Schleifen oder Guards bei WorkUnits noch das Mittel der Variablenzuweisung durch `Assign`-Aktivitäten. Zur Erinnerung, eine

Assign-Aktivität wird, wie in Ausdruck 4.37 gezeigt, als Choreographie-Fakt in der Wissensbasis beschrieben. Dabei definieren die Variablen *cName* und *rName* den Namen des `Copy`-Elementes und die Rolle, welche Zugriff auf die Variable hat. Die Variable an sich wird dabei durch das Prädikat *GetVar* beschrieben, welches äquivalent zur WS-CDL-Funktion `getVariable()` ist (siehe auch [BLF+05]). Die Funktion definiert eine Variable *vName*, welche durch die Rolle *rName* zugreifbar ist. Zusätzlich wird in WS-CDL bei der `Assign`-Aktivität beim Zuweisen von Variablen ein Ausdruck (*Expression*) als Zuweisung definiert, welcher entweder einen Wert oder einen Ort definiert, wo der Wert von Variablen gefunden werden kann. Dies kann an dieser Stelle beliebig komplex werden, weshalb in der Implementierung später bisher nur einfache Wert-Zuweisungen vorgesehen sind. Sollen beliebige XPath-Ausdrücke in der Implementierung benutzt werden, kann dies beispielsweise über bestimmte XPath-Bibliotheken in Prolog gelöst werden, die an dieser Stelle eingebunden werden müssen[7].

$$Assign(cName, rName, GetVar(vname, rName), Expression(xPath)) \tag{4.37}$$

Damit die so beschriebenen Variablen im Ereigniskalkül benutzt werden können, werden Variablen als Fluents modelliert, die bestimmte Gültigkeitsperioden besitzen. Wird beispielsweise ein `Assign` ausgelöst, wird das Fluent für die Variable entsprechend wahr. Wichtig ist dabei, dass eine Variable zu jedem Zeitpunkt immer nur einen Wert besitzt. Dafür muss beim erneuten Setzen der Variable das Fluent mit dem alten Variablenwert entsprechend terminiert werden, wie in Axiom 4.38 zu sehen. Ein erneutes Setzen wird, wie im Ereignis-Kalkül üblich, mit einer *Initiates*-Regel durchgeführt. Dabei wird zum Setzen von Variablen das Fluent $ValueOf(var, value, id)$ genutzt. Ist dieses Fluent für eine Choreographie-Instanz *id* wahr, dann ist die Variable mit dem Wert *value* belegt.

$$\begin{aligned} &Terminates(Event(x, id), ValueOf(GetVar(vName,'','', rName), value, id), t) \leftarrow \\ &\quad Initiates(Event(x, id), ValueOf(GetVar(vName,'','', rName), nvalue, id), t) \wedge \\ &\quad \neg(value = nvalue). \end{aligned} \tag{4.38}$$

Die Fälligkeit einer `Assign`-Aktivität und damit die Notwendigkeit zum Setzen des *ValueOf*-Fluents muss dabei durch die Zuständigkeitskette eindeutig erkannt werden. Ist die `Assign`-Aktivität beispielsweise als Teil einer Sequenz definiert, gibt es zwei Möglichkeiten, wann eine Variablenzuweisung erfolgen muss. Auf der einen Seite kann eine Aktivität innerhalb einer Sequenz übersprungen werden. Wenn diese ausgelassene Aktivität eine Variablenzuweisung ist, dann muss in diesem Fall die Zuweisung erfolgen. Auf der anderen Seite kann eine Sequenz auch durch am Ende liegende, überspringbare Aktivitäten beendet werden (siehe Abschnitt 4.3.2.2). Für diesen Fall muss die Variablenzuweisung erfolgen, wenn die Vateraktivität durch eine überspringbare Aktivität beendet wird. Dieser Fall gilt

[7] Siehe `http://www.swi-prolog.org/pldoc/man?predicate=xpath/3`.

auch für `Parallel`-Aktivitäten, wenn die Parallelaktivität nur noch aus noch nicht beendeten, überspringbaren Aktivitäten besteht. Axiom 4.39 prüft entsprechend, ob eine Vateraktivität einer Variablenzuweisung durch eine überspringbare Aktivität beendet wurde.

$$\begin{aligned} &Initiates(Event(x,id), ValueOf(GetVar(vName,'','',rName),value,id),t) \leftarrow \\ &\quad Initiates(Event(X,id), Complete(node,id),T) \wedge HoldsAt(Omittable(node,id),T) \wedge \\ &\quad isComplex(node) \wedge NodeContainsAssign(node,assign) \wedge \\ &\quad Assign(\_,rName,GetVar(vName,'',''),Expression(value)) = assign \end{aligned} \tag{4.39}$$

Damit lassen sich Assign-Aktivitäten am Ende einer komplexen Aktivität erkennen. Wird die Variablenzuweisung allerdings als Teil einer Sequenz beim Verifizieren ausgelassen, dann muss beim Überspringen der Aktivität die Variablenzuweisung trotzdem erfolgen. Axiom 4.40 stellt dies sicher, indem geprüft wird, ob die Vateraktivität einer `Assign`-Aktivität eine Sequenz ist, welche nicht beendet wurde und für welche ein Ereignis einen nachfolgenden aktiven Schritt ($Forerunner(node,current)$) innerhalb der Sequenz validiert.

$$\begin{aligned} &Initiates(Event(x,id), ValueOf(GetVar(vName,'','',rName),value,id),t) \leftarrow \\ &\quad IsAssign(node) \wedge GetParent(node,parent) \wedge \\ &\quad IsSequence(parent) \wedge \neg HoldsAt(Complete(parent,id),t) \wedge \\ &\quad SequenceStepActive(parent,id,t,current) \wedge \\ &\quad Initiates(Event(x,id), Valid(current,id),t) \wedge \\ &\quad Forerunner(node,current) \wedge \\ &\quad Assign(\_,rName,GetVar(vName,'',''),Expression(value)) = node \end{aligned} \tag{4.40}$$

Variablenzuweisungen können allerdings nicht nur innerhalb von Sequenzen oder Parallel-Aktivitäten vorkommen, sondern auch innerhalb von `Choice`- oder `WorkUnit`-Aktivitäten definiert werden. Bei einer WorkUnit ist es relativ einfach zu erkennen, ob die `Assign`-Aktivität ausgeführt werden muss. Wird die `WorkUnit` durch die `Assign`-Aktivität beendet (siehe Abschnitt 4.3.2.4), dann muss auch das entsprechende Fluent für die Variable gesetzt werden. Bei einer Auswahl ist dies nicht mehr ganz so einfach zu erkennen, wenn diese mehrere überspringbare Aktivitäten enthält und die Auswahl durch eine der überspringbaren Teilaktivitäten beendet wird. Da die überspringbaren Aktivitäten nicht beobachtbar sind, ist dieser Fall für Assign nicht mehr entscheidbar. Daher können in Bezug auf Assign nur diejenigen Choreographien verifiziert werden, in denen Auswahlaktivitäten mit Assign so definiert werden, dass keine weiteren auslassbaren Aktivitäten vorkommen. Ist dies als Vorbedingung der Fall, verhält sich die Variablenzuweisung als Teil einer Auswahl genauso wie bei der WorkUnit, Sequenz und Parallel und wird bei Beenden der Vateraktivität durch die überspringbare Aktivität (*Omittable* gilt) ausgelöst, wie in Axiom 4.39 spezifiziert.

Wie genau Vateraktivitäten von Variablenzuweisungen bei einer `Choice` beendet werden,

wurde bereits im Abschnitt 4.3.2.2 beschrieben. Für die `WorkUnit` wird dies im folgenden Abschnitt diskutiert.

4.3.2.4. WorkUnits: Bedingte Aktivitäten und Schleifen

Eine WorkUnit besteht aus einer Teilaktivität und beschreibt die Anforderungen, welche erfüllt sein müssen, um eine Teilaktivität (erneut) auszuführen. Wie in 4.2.1.1 beschrieben, definiert WS-CDL drei Bedingungen für eine WorkUnit: Eine Ausführungsbedingung, eine Schleifenbedingung und eine Verhaltensbedingung bezüglich des Blockierungsverhaltens. Die Aktivierung einer WorkUnit ist nahezu identisch zur `Choice`; allerdings muss zusätzlich bei der Aktivierung die Ausführungsbedingung (Guard) geprüft werden.

Eine aktive WorkUnit ist also valide, wenn die einzige Teilaktivität der WorkUnit durch ein beobachtetes Ereignis validiert wird. Dabei sind zwei Fälle zu unterscheiden: Im ersten Fall muss bei Betreten der WorkUnit getestet werden, ob der Guard wahr ist. Im zweiten Fall muss genau dieser Guard nicht mehr geprüft werden, wenn die WorkUnit bereits betreten ist. Für den ersten Fall wird dafür das Prädikat $GuardTrue(node, id, t)$ eingeführt, welches in Regel 4.41 definiert ist. Diese Regel prüft, ob eine Variable zur Zeit t gesetzt ist und damit, ob das Fluent $ValueOf$ zu dieser Zeit wahr ist und der Wert dem Erwarteten entspricht.

$$
\begin{aligned}
&GuardTrue(node, id, t) \leftarrow \\
&\quad GuardExpression(node, GetVar(vName,'','), op, tValue) \wedge \\
&\quad HoldsAt(ValueOf(GetVar(vName,'',\_), value, id), t) \wedge \\
&\quad TestValue(value, op, tValue) \qquad (4.41) \\
&RepeatTrue(node, id, t) \leftarrow \\
&\quad RepeatExpression(node, GetVar(vName,'','), op, tValue) \wedge \\
&\quad HoldsAt(ValueOf(GetVar(vName,'',\_), value, id), t) \wedge \\
&\quad TestValue(value, op, tValue) \qquad (4.42)
\end{aligned}
$$

Ist der Guard wahr, dann kann die WorkUnit betreten und ausgeführt werden. Im zweiten Fall, bei der die WorkUnit bereits betreten (bereits *Valid*) ist, muss der Guard nicht mehr geprüft werden. Axiom 4.43 beschreibt das Setzen des *Valid*-Fluents für beide Fälle und überprüft, ob mit einem Ereignis die einzige Teilaktivität validiert wird.

$$
\begin{aligned}
&Initiates(Event(x, id), Valid(node, id), t) \leftarrow \\
&\quad IsWorkUnit(node) \wedge \neg HoldsAt(Complete(node, id), t) \wedge \\
&\quad ((\neg HoldsAt(Valid(node, id), t) \wedge GuardTrue(node, id, t)) \vee \\
&\quad\ (HoldsAt(Valid(node, id), t))) \wedge GetChild(node, child) \wedge \\
&\quad Initiates(Event(x, id), Valid(child, id), t) \qquad (4.43)
\end{aligned}
$$

Eine WorkUnit wird beendet, wenn die Teilaktivität der WorkUnit valide beendet wird und die Schleifenbedingung nicht mehr wahr ist. Auch hier müssen die beiden Fälle wie bei *Valid* betrachtet werden. Ist die WorkUnit noch nicht betreten, muss für einen direkten Abschluss der Guard überprüft werden. Ist sie bereits betreten, wird diese Prüfung nicht mehr benötigt. Um zu erkennen, ob die Schleifenbedingung nicht mehr gilt, wird entsprechend wie für den Guard ein $RepeatTrue(node, id, t)$ eingeführt. Schleifen in WS-CDL sind immer als $while(Bedingung)do$-Schleifen definiert, für die eine $RepeatExpression$ in der Faktenbasis über die Choreographie definiert ist. Daher muss nur geprüft werden, ob die Variable zur Zeit t auch gesetzt ist, also das Fluent $ValueOf$ zu dieser Zeit wahr ist und ob der Variablenwert der Schleifenbedingung entspricht. Das Vorgehen beim Schleifentest ist daher das gleiche wie für den Guard und wird wie in Axiom 4.42 überprüft. Das Prüfen auf Beendigung der WorkUnit ist durch Axiom 4.44 definiert.

$$\begin{aligned} &Initiates(Event(x,id), Complete(node,id), t) \leftarrow \\ &\quad IsWorkUnit(node) \wedge \neg RepeatTrue(node,id,t) \wedge \\ &\quad ((\neg HoldsAt(Valid(node,id),t) \wedge GuardTrue(node,id,t)) \vee \\ &\quad\ (HoldsAt(Valid(node,id),t))) \wedge GetChild(node,child) \wedge \\ &\quad Initiates(Event(x,id), Complete(child,id), t) \end{aligned} \tag{4.44}$$

Wichtig ist bei einer WorkUnit, ob sie als blockierend definiert ist. Wenn sie nicht blockierend und der Guard vor Betreten ungültig ist, dann muss laut Spezifikation die WorkUnit übersprungen werden. Damit eine Vateraktivität der WorkUnit damit umgehen kann, wird mit einem Effekt-Axiom der Status mit dem Fluent *Omittable* gesetzt, wie es auch in Axiom 4.45 beschrieben ist. Dabei wird geprüft, ob die WorkUnit noch nicht betreten ($\neg Valid$), nicht blockierend (Choreographie-Fakt $IsBlocking$) sowie der Guard ungültig ist.

$$\begin{aligned} &HoldsAt(Omittable(node,id),t) \leftarrow \\ &\quad IsWorkUnit(node) \wedge \neg HoldsAt(Valid(node,id),t) \wedge \\ &\quad \neg IsBlocking(node) \wedge \neg GuardTrue(node,id,t) \end{aligned} \tag{4.45}$$

Genauso wie bei den anderen komplexen Aktivitäten gibt es auch bei der WorkUnit Sonderfälle mit überspringbaren Aktivitäten. Beispielsweise kann eine WorkUnit selbst übersprungen werden, wenn die einzige Teilaktivität der WorkUnit bei Betreten überspringbar und der Guard wahr ist. Genauso muss der Sonderfall betrachtet werden, bei der eine Teilaktivität, wie beispielsweise eine Sequenz, durch überspringbare Aktivitäten beendet wurde. In diesem Fall muss das *Omittable* auch an eine Vateraktivität durchgereicht werden, damit diese mit einer möglichen nachfolgenden Aktivität weitermacht. Probleme mit blockierendem Verhalten gibt es an dieser Stelle nicht, da eine WorkUnit nur betreten werden kann, wenn der Guard wahr ist, und dies wird beim direkten Betreten im Axiom 4.45 schon ge-

prüft. Genauso geschieht dies indirekt durch $initiates(Valid)$ bei vorherigem Betreten der WorkUnit. Die Regel lässt also nicht zu, dass eine blockierende WorkUnit als überspringbar markiert wird, wenn sie noch nicht betreten wurde.

$$\begin{aligned} &HoldsAt(Omittable(node, ID), T) \leftarrow \\ &\quad IsWorkUnit(node) \wedge GetChild(node, child) \wedge HoldsAt(Omittable(child, id), t) \wedge \\ &\quad ((\neg HoldsAt(Valid(node, id), t) \wedge GuardTrue(node, id, t)) \vee \\ &\quad\ (HoldsAt(Valid(node, id), t))) \end{aligned} \tag{4.46}$$

Eine WorkUnit kann indirekt auch durch die überspringbaren Teilaktivitäten beendet werden. Dies ist beispielsweise bei einer Sequenz als Teilaktivität der Fall, wenn diese durch Axiom 4.27 beendet wird. Genauso wird eine noch nicht betretene WorkUnit beendet (falls nicht schon geschehen), wenn die Vateraktivität der WorkUnit beendet wird. Dies ist vor allem dann relevant, wenn eine `Assign`-Teilaktivität ihre Variable setzen soll. Dieses Verhalten wird durch Axiom 4.47 definiert. Probleme mit blockierendem Verhalten gibt es an dieser Stelle nicht, da eine WorkUnit nur unter den oben genannten Bedingungen als *Omittable* definiert ist.

$$\begin{aligned} &Initiates(Event(x, id), Complete(node, id), t) \leftarrow \\ &\quad IsWorkUnit(node) \wedge ContainsOmittable(node, id, t) \wedge \\ &\quad \neg HoldsAt(Complete(node, id), t) \wedge \neg HoldsAt(Valid(node, id), t) \wedge \\ &\quad HoldsAt(Omittable(node, id), t) \wedge GetParent(node, parent) \wedge \\ &\quad Initiates(Event(x, id), Complete(parent, id), t) \end{aligned} \tag{4.47}$$

Damit sind zum Beenden einer WorkUnit zum einen der normale Fall abgedeckt (Axiom 4.44) und zum anderen alle Fälle mit einer überspringbaren Teilaktivität (Axiom 4.47 und 4.27). Eine mögliche Vateraktivität kann auf eine überspringbare WorkUnit reagieren, wenn das Flag *Omittable* gesetzt wird (Axiom 4.46 und 4.45). Eine WorkUnit wird valide, wenn ein Ereignis die Teilaktivität unter bestimmten Bedingungen validiert (Axiom 4.43). Was fehlt, ist eine Behandlung der Schleifenbedingungen, die im folgenden Abschnitt weiter erläutert wird. Fehler für eine WorkUnit werden erkannt, wenn ein Ereignis das Fluent *Valid* nicht für die unbeendete Teilaktivität initiiert und damit auch das Fluent *Valid* für die WorkUnit nicht initiiert wird. Der Fehler wird in diesem Fall über dieses Fluent weiter in der Zuständigkeitskette nach oben durchgereicht.

Verifikation von Schleifen: Wenn die Schleifenbedingung einer WorkUnit nach Beenden der Teilaktivität wahr ist, muss die Teilaktivität wiederholt werden. Die meisten Unteraktivitäten im Unterbaum der WorkUnit haben aber vor erneutem Anlaufen der Schleife diverse Fluents wie *Complete* und *Valid* gesetzt und damit greifen die üblichen Regeln beim erneu-

ten Schleifendurchlauf nicht mehr. Die Fluents der Unteraktivitäten der WorkUnit müssen also vor erneutem Durchlaufen der Schleife terminiert werden, um die Verifikation erfolgreich durchführen zu können. Da allerdings mit einem Ereignis die Teilaktivität der WorkUnit beendet und damit das Fluent *Complete* gesetzt wird, kann mit dem gleichen Ereignis nicht das gleiche Fluent terminiert werden. Um dieses Problem zu umgehen, wird ein neues Ereignis $Repeat(node, id)$ eingeführt, was im Falle eines erneuten Schleifendurchlaufs von *node* in der Choreographie-Instanz *id* gefeuert wird. Tritt dieses Ereignis direkt nach Beenden der Teilaktivität der WorkUnit auf, werden alle *Valid*- und *Complete*-Fluents im Unterbaum der WorkUnit terminiert. Das Terminieren und damit das Zurücksetzen der Fluents ist in den Axiomen 4.48 und 4.49 definiert.

$$\begin{aligned} &Terminates(Repeat(node, id), Complete(any, id), t) \leftarrow \\ &\quad IsWorkUnit(node) \wedge GetChildren(node, any) \end{aligned} \tag{4.48}$$

$$\begin{aligned} &Terminates(Repeat(node, id), Valid(any, id), t) \leftarrow \\ &\quad IsWorkUnit(node) \wedge GetChildren(node, any) \end{aligned} \tag{4.49}$$

Ein erneuter Schleifendurchlauf ist nötig, wenn die Teilaktivität der WorkUnit beendet und die Schleifenbedingung wahr ist. Ist dies der Fall, wird über *Update(Repeat(node, id), NT)* ein virtuelles Ereignis zum Anzeigen der Wiederholung in die Wissensbasis eingefügt. Wichtig ist dabei, dass die WorkUnit bei einem erneuten Durchlauf auch erneut betreten werden kann. Dafür muss der Guard ein erneutes Betreten der WorkUnit auch zulassen; das Prüfen dieser Bedingung ist in Axiom 4.51 definiert. Während eine Schleifenwiederholung läuft, wird das Fluent *Repeating* wahr, welches beendet wird, wenn die WorkUnit beendet wird (siehe auch Axiom 4.50).

$$\begin{aligned} &Terminates(Event(x, id), Repeating(node, id), t) \leftarrow \\ &\quad IsWorkUnit(node) \wedge Initiates(Event(x, id), Complete(node, id), t) \wedge \\ &\quad \neg RepeatTrue(node, id, t) \end{aligned} \tag{4.50}$$

$$\begin{aligned} &Initiates(Event(X, id), Repeating(node, id), t) \leftarrow \\ &\quad isWorkUnit(node) \wedge GetChild(node, child) \wedge \\ &\quad Initiates(event(X, id), Complete(child, id), t) \wedge \\ &\quad GuardTrue(node, id, t) \wedge RepeatTrue(node, id, t) \wedge \\ &\quad Succ(t, t1) \wedge Update(Repeat(node, id), t1) \end{aligned} \tag{4.51}$$

Mit dem Trick des Einfügens eines neuen Ereignisses zum Zurücksetzen der Schleifenteilnehmer können Schleifen einfach behandelt werden. Problematisch ist allerdings, wenn ein nachfolgendes Ereignis den selben Zeitstempel besitzt, wie das *Repeat*-Ereignis. Da allerdings die Zeitstempel der Nachrichten in der Implementierung bei Durchlaufen des Service-Busses bis auf Nanosekunden genau bestimmt werden, ist das Eintreten dieses Sonderfalls

selbst bei parallel eintreffenden Nachrichten für eine Choreographie-Instanz sehr unwahrscheinlich. Die typischen Laufzeiten der Nachrichten liegen im Bereich von Millisekunden.

4.3.3. Verifikation von Zeitschranken

Bisher sind nur funktionale Anforderungen und damit der Kontrollfluss der Interaktionen von Choreographien betrachtet worden. Zusätzlich bietet WS-CDL aber nicht nur die Spezifikation der funktionalen Anforderungen, sondern auch das Spezifizieren von rudimentären non-funktionalen Anforderungen wie Zeitschranken bei Interaktionen an. Zeitschranken werden in WS-CDL anhand des `Timeout`-Elements für Interaktionen definiert, wobei mithilfe eines XPath-Ausdrucks entweder eine Maximaldauer oder eine obere Zeitschranke als Ausführungsfrist für die Interaktion festgelegt werden kann. Soll beispielsweise die maximale Ausführungszeit oder eine Zeitschranke dynamisch zur Laufzeit durch Variablen gesetzt werden, muss an dieser Stelle das Modell erweitert werden. Für das generelle Verständnis zum Testen von Zeitschrankenverletzungen gibt es dabei aber keinen neuen Erkenntnisgewinn, weshalb nur die beiden statisch definierten Typen an dieser Stelle stellvertretend vorgestellt werden.

Um die Verletzung von Zeitschranken zu erkennen und zu beschreiben, werden zwei weitere Prädikate $Duration(i, d_{max})$ und $Deadline(i, t_{max})$ eingeführt. Sie spezifizieren zum einen die maximale Ausführungszeit d_{max} sowie zum anderen eine absolute Zeitschranke t_{max} für eine Interaktion i. Die entsprechenden Fakten einer Interaktion werden bei der Abbildung der Choreographie-Struktur zusätzlich in die Wissensbasis eingefügt und können von den Axiomen zur Erkennung von Zeitschrankenverletzungen benutzt werden. Zusätzlich wird ein weiteres Fluent $Timeout(i, id)$ für eine Interaktion i eingeführt. Es wird wahr, wenn eine Zeitschranke in einer Choreographie-Instanz id verletzt wird.

$$
\begin{aligned}
&HoldsAt(Timeout(i, id), t) \leftarrow \\
&\quad Duration(i, d_{max}) \wedge (\\
&\quad (HoldsFor(Valid(i, id), [t_s, t_e]) \wedge \neg HoldsFor(Complete(i, id), \_)) \vee \\
&\quad (HoldsFor(Valid(i, id), [t_s, \_]) \wedge HoldsFor(Complete(i, id), [t_e, \_]))) \wedge \\
&\quad (t_e - t_s) > d_{max} \wedge (t_s + d_{max}) <= t
\end{aligned}
\tag{4.52}
$$

Neben der dargestellten Abbildung der Fakten sind für das Erkennen zusätzliche Effekt-Axiome notwendig. Diese nutzen statt des bereits bekannten $HoldsAt$-Prädikats das Prädikat $HoldsFor(Fluent, [t_s, t_e])$[8], über welches die maximale Gültigkeitsperiode eines Fluents ermittelt wird. Das Prädikat gibt mit t_s den Startzeitpunkt der Gültigkeitsperiode an und mit t_e den Endzeitpunkt. Im Falle der maximalen Ausführungsdauer einer Interaktion gibt es zwei unterschiedliche Fälle zu betrachten. Im ersten Fall wird eine Interaktion normal

[8] $HoldsFor(Fluent, [t_s, t_e])$ ist im Cached-Event-Calculus von Haus aus definiert und lässt sich an dieser Stelle sehr gut zum Erkennen von Zeitschrankenverletzungen einsetzen.

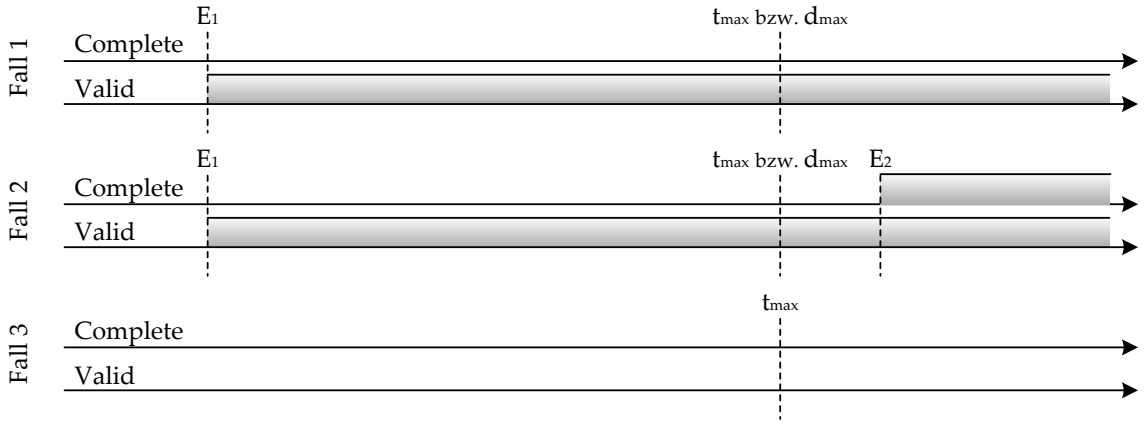


Abbildung 4.24.: Verletzung von Zeitschranken

gestartet und beendet, wobei dann jeweils die Startzeiten der Fluents *Valid* und *Complete* betrachtet und mit d_{max} vergleichen werden müssen. Im zweiten Fall wurde die Interaktion zwar gestartet, aber noch nicht beendet. In diesem Fall muss nur die Gültigkeitsdauer des Fluents *Valid* betrachtet werden. Ist zu einem Zeitpunkt t die maximale Ausführungszeit überschritten, wird die Zeitschrankenverletzung erkannt, wie in Axiom 4.52 spezifiziert. Das Vorgehen ist auch in Abbildung 4.24 illustriert.

Im Falle einer Ausführungsfrist gibt es, wie in Abbildung 4.24 zu sehen, sogar drei unterschiedliche Fälle. Neben den beiden Fällen, in denen eine Interaktion nach dem Start entweder schon beendet worden ist oder nicht, gibt es auch den Fall, dass eine Interaktion noch nicht gestartet wurde, aber die Ausführungsfrist schon überschritten ist. In diesem Fall wird eine dritte Unterscheidung bei der Erkennung von Fristen benötigt, wie in Axiom 4.53 spezifiziert.

$$
\begin{aligned}
&HoldsAt(Timeout(i,id),t) \leftarrow \\
&\quad Timeout(i,t_{max}) \wedge t >= t_{max} \wedge (\\
&\quad (HoldsFor(Valid(i,id),[t_s,t_e]) \wedge \neg HoldsFor(Complete(i,id),\_) \wedge t_e > t_{max}) \vee \\
&\quad (HoldsFor(Valid(i,id),[t_s,\_]) \wedge HoldsFor(Complete(i,id),[t_e,\_]) \wedge t_e > t_{max}) \vee \\
&\quad \neg(HoldsFor(Valid(i,id),\_)))
\end{aligned}
\tag{4.53}
$$

Prinzipiell lassen sich mit den beschriebenen Effekt-Axiomen Zeitschrankenverletzungen für beliebige Aktivitäten innerhalb einer Choreographie erkennen. Dazu sind zusätzliche, manuell zu erstellende *HoldsAt*-Prädikate notwendig, die anhand der Prädikate *Duration* oder *Deadline* Zeitschranken für eine Aktivität festlegen. Die Darstellung dieser Prädikate für andere Aktivitäten muss analog zur Abbildung der Interaktionszeitschranken erfolgen, damit sie anschließend durch die Effekt-Axiome berücksichtigt werden können.

4.3.4. Weitere Integritätsbedingungen

Damit weitere Integritätsbedingungen wie nachrichtenbasierte Integritätsbedingungen, instanzbasierte oder instanzübergreifende Integritätsbedingungen verifiziert werden können, muss das Modell noch erweitert werden. Bisher können der Kontrollfluss von WS-CDL-Dokumenten sowie darüber spezifizierte non-funktionale Eigenschaften wie einfache Zeitschranken von Interaktionen verifiziert werden.

Zur Unterstützung von inhaltlichen Bedingungen werden im Rahmenwerk die Axiome 4.16 und 4.17 der Basisaktivitäten um eine Bedingung erweitert, welche mithilfe von selbst definierten Regeln spezifiziert werden kann:

$$\begin{aligned} &Initiates(Event(x,id),Valid(x,id),t) \leftarrow \\ &\quad IsExchange(x) \wedge ExchangeConstraint(Event(x,id),t) \qquad (4.54) \\ &Initiates(Event(x,id),Complete(x,id),t) \leftarrow \\ &\quad IsExchange(x) \wedge ExchangeConstraint(Event(x,id),t) \qquad (4.55) \end{aligned}$$

Das Prädikat `ExchangeConstraint` muss dabei als eigene Regel definiert werden. Wenn allerdings keine weiteren Bedingungen außer den üblichen Formatbestimmungen durch WS-CDL definiert sind, muss zumindest ein `ExchangeConstraint`-Fakt zur Wissensbasis hinzugefügt werden, damit bei Auftreten einer Nachricht diese auch verifiziert werden kann. Mit Unterstützung der Interpreter-Schnittstelle aus Abschnitt 4.1.3 kann eine Regel definiert werden, welche weitere Bedingungen für spezielle Nachrichtentypen definiert. Das Prädikat besitzt dafür zwei Argumente: zum einen das Ereignis x und zum anderen einen Zeitpunkt t, um auch zeitliche Bedingungen definieren zu können. Um das Vorgehen zu illustrieren, wird in einem Beispiel eine einfache Zeitschrankenbedingung für eine Interaktion i definiert, welche eine zum Ereignis passende Basisaktivität x enthält:

$$\begin{aligned} &ExchangeConstraint(Event(x,id),t) \leftarrow \\ &\quad \neg HoldsAt(Timeout(i,id),t) \wedge GetChild(i,x) \qquad (4.56) \end{aligned}$$

Die Regel muss entsprechend über die Interpreter-Schnittstelle zur Wissensbasis hinzugefügt werden. Wird dann zur Laufzeit das Ereignis x detektiert, wird das *Valid*- beziehungsweise *Complete*-Fluent für die Basisaktivität nur dann initiiert, wenn die Zeitschrankenrestriktion eingehalten wird. Über diesen Mechanismus lassen sich zu beliebigen Fakten in der Wissensbasis Bedingungen definieren, die bei Erhalt einer Nachricht geprüft werden. Da mithilfe der Zuständigkeitskette Fehler in den Basisaktivitäten an komplexe Vateraktivitäten hochgereicht werden, reicht an dieser Stelle die Erweiterungsmöglichkeit über die Basisaktivitäten aus, um ereignisorientierte inhaltliche Integritätsbedingungen zu definieren.

Als schwierig erweisen sich nachrichtenbasierte Integritätsbedingungen, die sich auf den

Inhalt einer Nachricht beziehen. Bisher lässt die formale Repräsentation eines Ereignisses dieses nicht zu, da ein Ereignis wie eine Nachrichtenübertragung nur durch das Prädikat $Happens(Event(Exchange(name, fromRole, toRole), id), t)$ definiert wird. Eine einfache Möglichkeit ist an dieser Stelle die Erweiterung der internen Repräsentation einer protokollieren Nachricht von $Exchange(name, fromRole, toRole)$ um ein viertes Attribut, welches den übertragenen Inhalt der Nachricht übergibt. Hier gibt, es wie im vorigen Abschnitt über Zeitschranken bereits diskutiert, entsprechende XML- und XPath-Erweiterungen, die auf XML-Strukturen zugreifen und Bedingungen überprüfen können.

Bisher ist zur Definition von Regeln als Erweiterung der Integritätsbedingungen nur Prolog als Sprache vorgesehen, bei der ein Modellierer sehr gute Kenntnisse über die interne Struktur der Regeln besitzen muss. Hier sind später sicherlich auch benutzerfreundlichere Möglichkeiten vorzuziehen, die in dieser Arbeit allerdings nicht mehr weiter entwickelt werden, da sie außerhalb des Fokus der Arbeit liegen. Die Definition weiterer Integritätsbedingungen ist aber prinzipiell möglich und wird vom Rahmenwerk durch die eben vorgestellten Erweiterungspunkte und durch entsprechende Interpreter unterstützt.

4.3.5. Bestimmung des Zustandes einer Choreographie

Zur Laufzeit werden durch das Monitoring-Framework Anfragen gegen die Wissensbasis gestellt, um den Zustand der überwachten Ablaufinstanzen zu ermitteln und um gegebenenfalls auf Fehler reagieren zu können. Außerdem kann es für die Fehlerbehebung hilfreich sein, rückblickend den Zustand einer Ablaufinstanz für verschiedene Zeitpunkte oder die Zustandsverläufe der Teilaktivitäten über den gesamten Ausführungszeitraum zu betrachten.

Zur Bestimmung des Gesamtzustands einer bestimmten Ausführungsinstanz eignet sich das dafür vorgesehene Fluent *Valid*. Aufgrund der Zuständigkeitskette lässt sich der Gesamtzustand auf den Zustand des Wurzelknotens zurückführen. Wenn der Wurzelknoten valide ist, dann ist auch die Ausführungsinstanz valide. Damit der Gesamtzustand einfach zu bestimmen ist, wird eine Hilfsregel 4.57 mit dem Prädikat $IsValid(id, t)$ eingeführt, welches für eine Ausführungsinstanz *id* bestimmt, ob sie zur Zeit t valide ist. Dabei muss zum einen der Wurzelknoten zu dieser Zeit valide sein und zusätzlich darf keine Zeitschrankenverletzung bei der Ausführung zu diesem Zeitpunkt erkennbar sein. Zur Laufzeitverifikation kann in der Implementierung wie in Abschnitt 4.2.2 beschrieben diese Regel aufgerufen werden.

$$
\begin{aligned}
&IsValid(id, t) \leftarrow \\
&\quad Root(node) \wedge \neg HoldsAt(Error(Event(x, id)), t) \wedge \\
&\quad \neg HoldsAt(Timeout(\_, id), t) \wedge Initiates(Event(x, id), Valid(node, id), t)
\end{aligned} \tag{4.57}
$$

$$IsComplete(id, t) \leftarrow$$
$$Root(node) \wedge HoldsAt(Complete(node, id), t) \tag{4.58}$$

Weiterhin lassen sich mit den Fluents auch weitere Rückschlüsse auf den Ablauf einer Instanz ziehen. Beispielsweise kann mit Regel 4.58 bestimmt werden, ob eine Ablaufinstanz bereits beendet ist. Eine Ablaufinstanz ist beendet, wenn der Wurzelknoten beendet ist. Hierbei sei aber anzumerken, dass die sichere Bestimmung auf Beendigung der Instanz nicht ganz trivial ist. Wenn beispielsweise die letzte Aktivität optional ist und keine weiteren Zeitschranken definiert sind, dann kann aus dem Status Quo nicht ermittelt werden, ob die optionale Aktivität noch ausgeführt werden wird oder nicht. Die Choreographie-Instanz darf in diesem Falle nicht beendet werden. Es lässt sich aber mit einfachen Mitteln feststellen, welche Aktivitäten noch nicht beendet sind, um Rückschlüsse darauf zu ziehen, auf welche Aktivitäten noch gewartet wird. Allerdings ist es möglich, dass eine Choreographie im schlimmsten Fall nie als *Complete* markiert werden kann. Dies stellt aber für die Laufzeitverifikation kein Problem dar, da bisher alle beobachteten Ereignisse verifiziert worden sind.

Zusätzlich kann durch Regel 4.59 der gesamte Zeitraum ermittelt werden, über den eine mittlerweile abgeschlossene Instanz aktiv war. Auf dieselbe Weise können auch Zeitschrankenverletzungen einer Choreographie-Instanz einfach mit einer Anfrage auf des *Timeout*-Fluent mithilfe von *HoldsAt* ermittelt werden.

$$Duration(id, t_s, t_e) \leftarrow$$
$$Root(r) \wedge HoldsFor(Valid(r, id), [t_s, \_]) \wedge HoldsFor(Complete(r, id), [t_e, \_]) \tag{4.59}$$

Bisher wurde nur der Gesamtzustand der Choreographie und damit der Wurzelknoten betrachtet. Mit den hier gezeigten Anfragen und Hilfsregeln lässt sich natürlich auch detailliert zu jedem Zeitpunkt der Zustand jeder einzelnen Aktivität in der Choreographie bestimmen. Dafür muss lediglich auf die Konkatenation mit dem Prädikat $Root(node)$ in den jeweiligen Regeln verzichtet werden.

$$Error(event, id, t) \leftarrow$$
$$Happens(Event(event, id), t) \wedge t > t_e \wedge \neg IsValid(id, t_e) \tag{4.60}$$

Ein wichtiger Punkt ist die Erkennung, welche Aktivität nicht erfolgreich mit einem Event validieren kann. Ein Fehler zur Laufzeit lässt sich dabei zum einen schnell ermitteln, wenn nach Eingang einer Nachricht am Monitor das Prädikat *IsValid* ungültig wird und zum anderen gut mit den vorgestellten Mitteln von Hand analysieren. Mit Unterstützung von Regel 4.60 lassen sich fehlerhafte Ereignisse direkt zurückgeben. Das erste und damit das zeitlich frühste Ereignis ist der Verursacher des Fehlers, alle nachfolgenden sind entsprechend Folgefehler. Bei der Nutzung dieser Regel werden die entsprechenden Ereignisse an

die Variable *event* gebunden, durch welche die Instanz nicht als valide initiiert wurde. In der Regel ergibt sich dabei die Bezeichnung eines *Exchange*-Elements, welches weiter analysiert werden kann. Durch dieses Element kann sowohl der Nachrichtentyp festgestellt werden, der den Fehler verursacht hat, als auch die an dem Nachrichtenaustausch beteiligten Rollen. Zusätzlich liefert die Anfrage über die Zeitvariable t den Zeitpunkt des fehlerhaften Abbruchs.

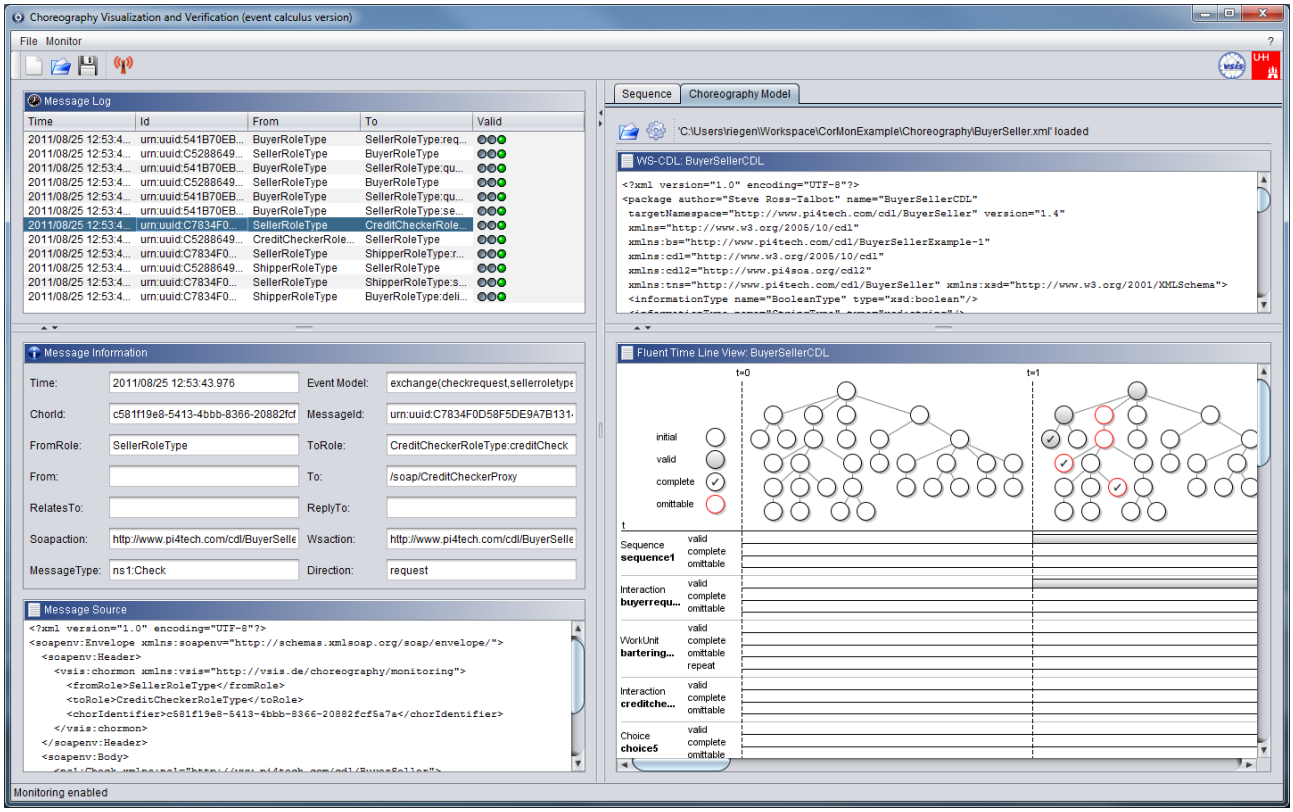


Abbildung 4.25.: Screenshot vom Prototypen der EC-Laufzeitverifikation

Mit Unterstützung der vorgestellten Hilfsregeln lassen sich der Zustand einer Choreographie, die Laufzeit und auch die fehlerverursachenden Ereignisse schnell ermitteln. Die Hauptanfrage für die Laufzeitverifikation bezieht sich allerdings auf die Regel *IsValid*, mit deren Hilfe jede einzelne Nachricht einfach überprüft werden kann. Wird ein Fehler erkannt, kann die Ablaufinstanz mit den vorgestellten Mitteln zusätzlich von Hand analysiert werden. Im Rahmen dieser Arbeit ist auch ein Prototyp eines Verifikationswerkzeugs entstanden, welches diese manuelle Analyse unterstützt. Wie in Abbildung 4.25 zu sehen, zeichnet das Werkzeug zu jedem Zeitpunkt eines Ereignisses den jeweiligen Status der Fluents einer Choreographie-Instanz auf und stellt diese grafisch dar.

4.3.6. Abbildung und Verifikation von WS-CDL am Beispiel

Um die bisher vorgestellte Abbildung von WS-CDL in das Ereigniskalkül sowie die dazugehörige Laufzeit-Verifikation zu illustrieren, wird der Vorgang anhand eines kleinen Bestell-

prozesses skizziert (Beispiel aus [Voi10]). Im Bestellprozess stellt ein Nutzer eine Preisanfrage, die von einer Bestellung gefolgt wird. Diese Interaktionen werden durch das Sequence-Artefakt aus WS-CDL verkettet, was auch in Quelltext 4.4 dargestellt ist. Jede Interaktion besteht dabei aus einer Anfrage und einer Antwort.

Quellcode 4.4: Beispiel-Choreographie

```
<choreography ...>
  <sequence>
    <interaction ... name="getQuote">
      <participate fromRoleTypeRef="Buyer" toRoleTypeRef="Seller" ... />
      <exchange action="request" name="qReq">
        ...
      </exchange>
      <exchange action="respond" name="qResp">
        ...
      </exchange>
    </interaction>
    <interaction ... name="order">
      <participate fromRoleTypeRef="Buyer" toRoleTypeRef="Seller" ... />
      <exchange action="request" name="oReq">
        ...
      </exchange>
      <exchange action="respond" name="oResp">
        ...
      </exchange>
    </interaction>
  </sequence>
</choreography>
```

Im ersten Schritt wird bei der Konfiguration die Wissensbasis mit den Fakten der Choreographie-Struktur gefüllt. Dabei werden beim Parsen eines WS-CDL-Dokuments die Basisaktivitäten identifiziert, was im Beispiel aus Quellcode 4.4 vier exchange-Artefakte bedeutet, für welche die folgenden Fakten zur Wissensbasis hinzugefügt werden:

- $BasicActivity(Exchange(qReq, buyer, seller))$,
- $BasicActivity(Exchange(qResp, seller, buyer))$,
- $BasicActivity(Exchange(oReq, buyer, seller))$,
- $BasicActivity(Exchange(oResp, seller, buyer))$

Nachdem die Basisaktivitäten eingefügt sind, werden die komplexen Aktivitäten identifiziert. Auf der einen Seite sind dies im Beispiel eine Sequenz und zwei Interaktionen mit jeweils einer Anfrage und einer Antwort. Die Interaktionen werden daher als Sequenzen $Sequence(getQuote)$ und $Sequence(order)$ in die Wissensbasis eingefügt. Die Teilaktivitäten der Interaktionen werden via $MemberActivities(getQuote, [qReq, qResp])$ und $MemberActivities(order, [oReq, oResp])$ hinzugefügt. Dem entsprechend wird die Sequenz als $Sequence(sequence0)$ mit den Teilaktivitäten $MemberActivities(sequence0, [getQuote, order])$ in der Wissensbasis bekannt gemacht. Da die Sequenz keinen definierten Namen hat, muss hier ein eindeutiger Name zur Identifikation vergeben werden. Die Sequenz ist gleichzeitig auch die Wurzelaktivität der Choreographie, was durch den Fakt $Root(sequence0)$ definiert wird. Die Struktur der Choreographie ist nun bekannt und kann später zur Laufzeit von den WS-CDL-Axiomen genutzt werden.

Im zweiten Schritt der Laufzeitverifikation empfängt der Monitor Nachrichten über die Sensor-Schnittstelle und fügt die entsprechenden Fakten über die Ereignisse in die Wissensbasis ein. In unserem Beispiel sind dies drei Ereignisse mit folgenden Fakten:

- $Happens(Event(Exchange(qReq, buyer, seller), a), 2)$,
- $Happens(Event(Exchange(oReq, buyer, seller), a), 4)$,
- $Happens(Event(Exchange(qResp, seller, buyer), a), 6)$,

Um jetzt zur Laufzeit den Status der Choreographie zu ermitteln, kann mit den im Abschnitt 4.3.5 vorgestellten Anfragen gearbeitet werden. Der resultierende Zeitverlauf der Fluents für das Beispiel ist auch in Abbildung 4.26 dargestellt. Dabei wird deutlich, dass für die erste Nachricht zum einen die Basisaktivität *qReq* valide beendet wird (durch die Axiome 4.16 und 4.17) und zum anderen, dass die jeweiligen Vateraktivitäten verifiziert werden. In diesem Fall sind es die Interaktion *getQuote*, welche durch das Sequenzaxiom 4.23 valide wird, und die Sequenz *sequence*, welche durch das gleiche Axiom valide wird. Zum Zeitpunkt $t = 4$ wird die Nachricht zur Basisaktivität *oReq* empfangen. Die Basisaktivität wird dabei durch die entsprechenden Axiome valide beendet. Genauso wird die Vateraktivität valide. Allerdings ist hierbei zu erkennen, dass das Ereignis für die Sequenz *sequence* das Fluent *Valid* nicht korrekt initiiert, da in Axiom 4.23 durch die Regel *SequenceStepActive* eigentlich das Ereignis des Vorgängers erwartet wird. Beim Vorgänger *getQuote* wird im gleichen Moment durch Axiom 4.22 das Fluent *Valid* deaktiviert, genauso wie bei der Sequenz. Da das Ereignis zum Zeitpunkt $t = 4$ für die Wurzelaktivität nicht das Fluent *Valid* initiiert, wird an dieser Stelle der Fehler erkannt und durch Axiom 4.21 das Fehlerfluent gesetzt. Erreicht jetzt nach diesem Zeitpunkt das eigentlich erwartete Ereignis *qResp* den Monitor, werden zwar die jeweiligen Teilaktivitäten wieder korrekt auf *Valid* und *Complete* gesetzt, allerdings ist die Choreographie immer noch als fehlerhaft markiert, da ein voriges Ereignis den Wurzelknoten nicht korrekt verifiziert hat.

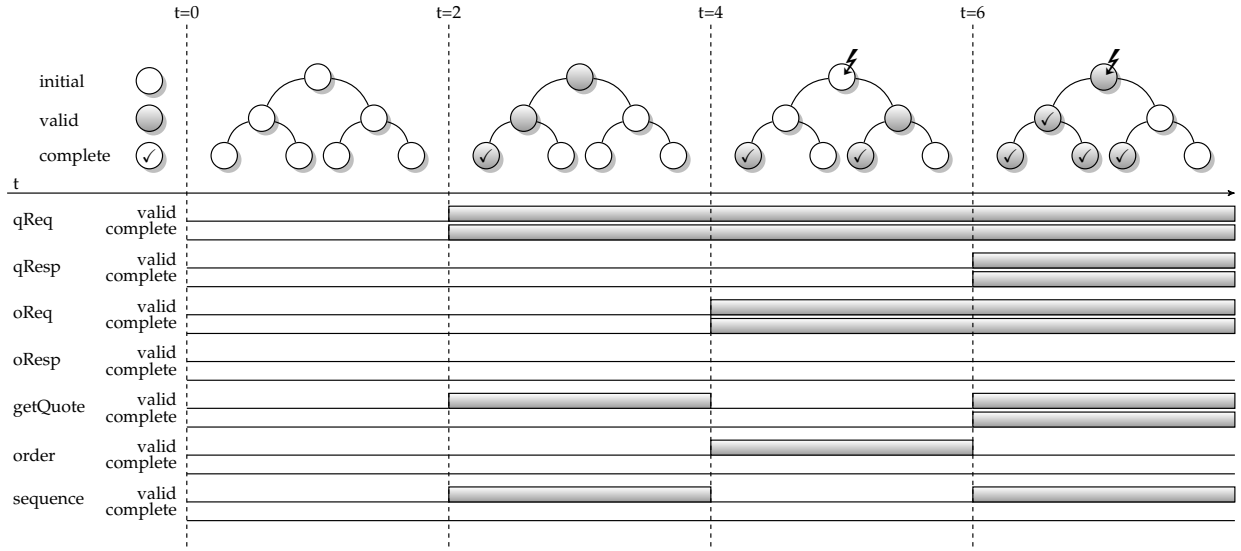


Abbildung 4.26.: Fluent-Zeitverlauf für die Beispiel-Choreographie

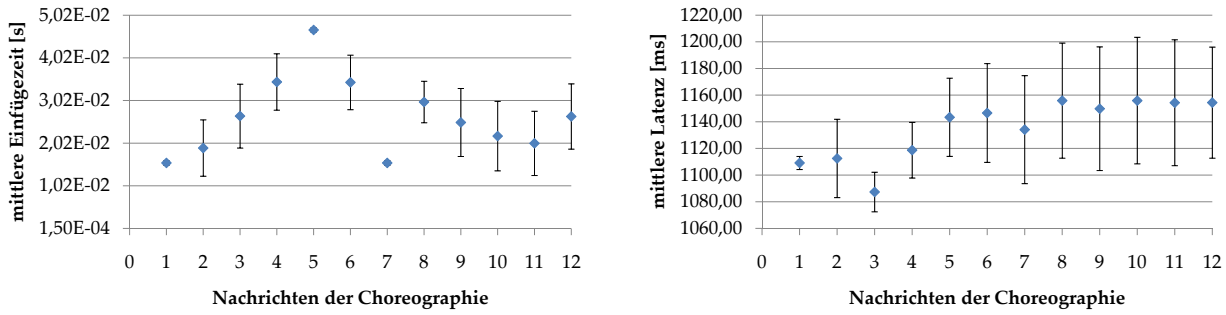
4.3.7. Evaluation

Die für die Implementierung vorgenommene Evaluation entspricht der Evaluation für den regelbasierten Ansatz aus Abschnitt 4.2.3. Auch für diese Evaluation werden die zwei Basisszenarien eines Einkaufs und einer Interaktion zwischen Europol und Eurojust für drei Testfälle genutzt. Die Testfälle untersuchen zum einen die Evaluation in Abhängigkeit von der Komplexität der Choreographie und zum anderen die Skalierbarkeitsaspekte der Implementierung. Auch bei dieser Evaluation liegt das Hauptaugenmerk auf dem Testen der Eignung des Modells zur Laufzeitverifikation von Prozess-Choreographien und damit auf der Geschwindigkeit, in der Nachrichten validiert werden können. Ein Nebenaspekt der Evaluation liegt auf der Skalierbarkeit der vorgestellten Implementierung.

4.3.7.1. Szenarien

Die Choreographie-Szenarien entsprechen den Szenarien aus Abschnitt 4.2.3 und werden daher hier nicht noch einmal ausgeführt, sondern nur kurz umrissen. Als Testumgebung für die Szenarien kommt auch hier ein Intel i5-750 mit 4 Kernen und 8GB RAM zum Einsatz. Auf dieser Umgebung laufen der Service-Bus, der Anwendungs-Server (Tomcat) für die Web-Services und der Monitor gleichzeitig. Die Testumgebung ist leistungsfähig genug, alle Komponenten gleichzeitig auf einer Maschine zu betreiben.

Szenario 1 - Kaufvorgang: Im ersten Szenario geht es um einen Kaufvorgang mit Preisverhandlung. Die Verhandlung läuft innerhalb einer Schleife ab, welche erst unterbrochen wird, wenn Einigkeit über den Preis besteht. Danach wird die Ware zum verhandelten Preis bestellt und der Verkäufer führt mithilfe eines externen Dienstleisters eine Kreditkartenprüfung durch. Wenn diese erfolgreich ist, weist der Verkäufer einem weiteren externen Dienst-



(a) Aktualisierungszeit der Wissensbasis (b) Latenz

Abbildung 4.27.: Einkäufer-Verkäufer-Szenario

leister an, die Ware zum Käufer zu senden. Dieser externe Versender nimmt Kontakt mit dem Käufer auf und teilt ihm die Details zum Versand mit.

Für die Implementierung dieses Szenarios sind die gleichen vier Web-Services wie für die Evaluation des regelbasierten Ansatzes benutzt worden. Für den Enterprise-Service-Bus sind die entsprechenden Proxy-Dienste angelegt und der Monitor ist aktiviert. Für die erste Evaluation wird die Choreographie zehn Mal in der gleichen Weise durchlaufen. Dabei werden jedes Mal nach einem Durchlauf der Service-Bus, der Monitor und die Verhaltensdienste neu gestartet, damit für jeden Test die gleichen Testbedingungen herrschen. Der Ablauf der Choreographie ist dabei immer gleich und damit lassen sich die Zeiten zur Verifikation in Abhängigkeit der versendeten Nachricht jeweils einzeln aufzeichnen. Aufgezeichnet werden dabei zwei verschiedene Zeiten: Zum einen die *reine Verifikationszeit* pro Nachricht und damit, wie viel Zeit Eclipse-CLP mit den vorgestellten Horn-Klauseln zur Validierung benötigt, und zum anderen, wie viel Zeit zwischen dem Erkennen der Nachricht im Service-Bus bis zur erfolgten Verifikation vergeht (*Latenz*). Wichtig ist hierbei anzumerken, dass die Verifikation Arbeit an zwei Stellen aufwirft. Zum einen wird bei jedem Einfügen eines neuen Ereignisses im Cached-Ereigniskalkül jeweils die Gültigkeitsperiode für jedes durch das Ereignis betroffene Fluent berechnet. Zum anderen werden bei der Abfrage, ob die Nachricht die Wurzelaktivität valide initiiert, die gerade berechneten Gültigkeitsperioden der geänderten Fluents abgefragt. Letzteres geschieht sehr schnell und ist damit für die Evaluation nicht direkt von Bedeutung. Der Hauptteil der Verifikation wird beim Einfügen eines Ereignisses in die Faktenbasis erbracht und wird daher im weiteren Verlauf der Evaluation hauptsächlich betrachtet.

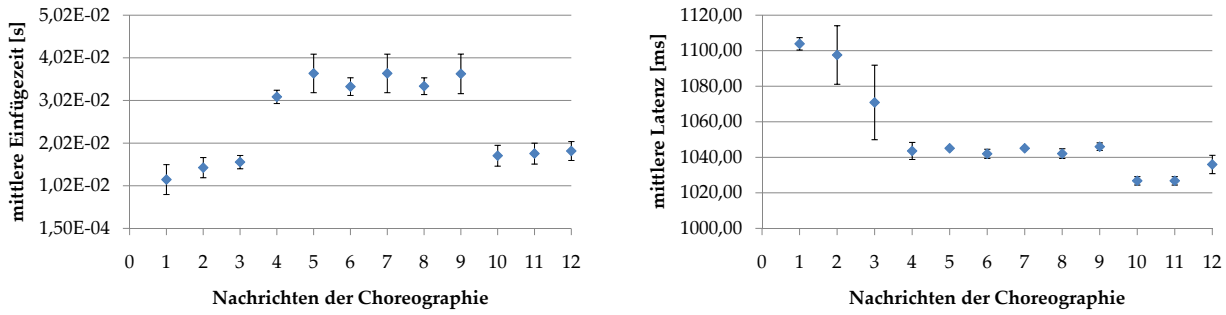
In Abbildung 4.27(a) ist die reine Einfügezeit in Abhängigkeit von der versendeten Nachricht für das erste Szenario gezeigt. Dabei ist zu sehen, dass die beiden ersten Nachrichten recht schnell verifiziert sind, da noch keine Fluents gesetzt sind. Das Einfügen der Antwort etwas dauert länger als die erste Anfrage, da in diesem Fall zusätzlich zu den *Valid*-Fluents schon das *Complete*-Fluent für die erste Interaktion gesetzt wird. Die Länge des Pfades vom Wurzelknoten zu der ersten Interaktion ist kurz und enthält daher kaum Aktivitäten. Es

feuern dabei wenige Regeln, weshalb die Einfügezeit kurz bleibt. Für die dritte Nachricht ist dies nicht mehr der Fall, da an dieser Stelle zur betreffenden Basisaktivität eine WorkUnit und eine Auswahl passiert werden müssen. Für diese müssen die entsprechenden Regeln ebenso geprüft werden. Mit Eintreffen der fünften Nachricht wird die WorkUnit durch die Schleifenbedingung erneut betreten, daher müssen alle Teilaktivitäten der WorkUnit entsprechend zurückgesetzt werden, was mehr Aufwand nach sich zieht. Die nachfolgenden beiden Aktivitäten als Teil der WorkUnit können dann wieder schneller verifiziert werden. Sie werden allerdings nicht beendet, da bei Betreten der Sequenz am Ende eine Assign-Aktivität zum Deaktivieren der Schleifenbedingung gesetzt wird. Die Sequenz mit dem Assign wird entsprechend durch Nachricht 7 beendet, was recht schnell passiert. Nach der WorkUnit ist nur noch eine einfache Auswahl mit einer größeren Sequenz von Interaktionen vorhanden, die zeitlich für die Einfügezeit auf recht gleichem Niveau liegen.

In Abbildung 4.27(b) ist die Latenz in Abhängigkeit von der versendeten Nachricht gezeigt. Dabei ist zu beachten, dass der Monitor die Nachrichten erst puffert. Wie in Abschnitt 4.2.3 wird die Latenz daher immer größer als eine Sekunde sein und bewegt sich bei den Messungen im Bereich von 1087,30 bis 1155,90 Millisekunden. Dadurch, dass die Nachrichtenübertragung schneller als die Verifikation ist, sind meistens mehrere Nachrichten in der Queue. Werden zwei Nachrichten mit nahe beieinanderliegenden Zeitstempeln nacheinander verifiziert, dann ist Latenz der zweiten verifizierten Nachricht in der Nähe der Summe der beiden Verifikationszeiten. Zusätzlich gilt, wie in Abschnitt 4.2.3 beschrieben, dass der Enterprise-Service-Bus nach erneutem Start seine Caches und entsprechende Sende-Queues initialisieren muss und damit am Start eine etwas höhere Latenz als erwartet besitzt. Sie geht nach dem Start also erst leicht nach unten und steigt dann wieder an, wenn mehrere Nachrichten in der Queue vorhanden sind.

Szenario 2 - Europol-Eurojust-Interaktion: Im zweiten Szenario geht es um eine einfache Anfrage von Eurojust an Europol, welche an weitere Teilnehmer delegiert wird. Als Erstes wird für diesen Fall vom einem Eurojust-National-Member eine Akte im Case-Management-System (CMS) angelegt. Nach einem Meeting der einen Fall betreffenden National-Member werden weitere Informationen über Personen von Europol über den Europol-Liason-Officer (ELO) angefordert. Dieser kann die Anfrage nicht direkt beantworten und ermittelt die angeforderten Informationen über drei verschiedene interne Systeme und trägt sie dann zusammen. Dürfen die Daten weitergegeben werden, werden sie an den betreffenden Eurojust-National-Member weitergeleitet. Dieser aktualisiert die im CMS angelegte Akte und beruft ein weiteres Treffen der National-Member zum Fall ein, um das weitere Vorgehen zu besprechen. In diesem einfachen Fall werden für die Kommunikation mit dem CMS, mit dem ELO, den National-Member und den einzelnen Systemen von Europol Web-Services-Aufrufe getätigt.

Für die Implementierung dieses Szenarios werden die gleichen sechs Web-Services wie für die Evaluation des regelbasierten Ansatzes benutzt. Wie im ersten Szenario wird die



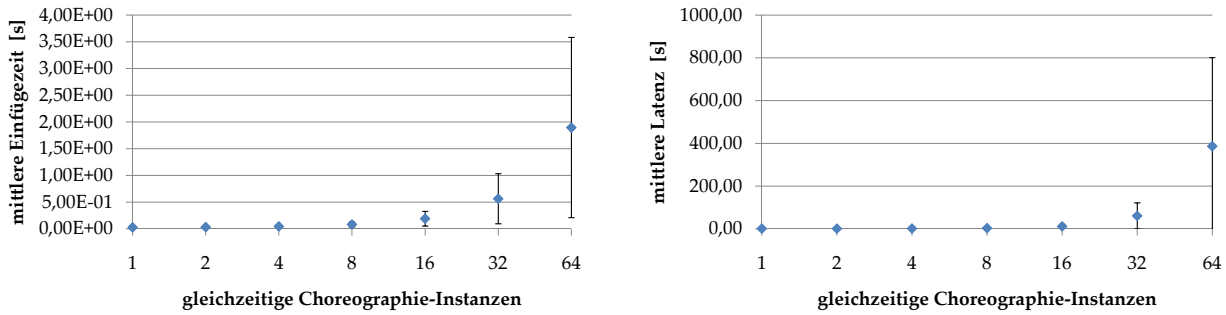
(a) Aktualisierungszeit der Wissensbasis (b) Latenz

Abbildung 4.28.: Europol-Eurojust-Szenario

Choreographie zehn Mal in der gleichen Weise durchlaufen. Dabei werden wie vorher jedes Mal der Service-Bus, der Monitor und die Verhaltensdienste neu gestartet, damit für jeden Test die gleichen Testbedingungen herrschen.

In Abbildung 4.28(a) ist jeweils die reine Einfügezeit in die Wissensbasis in Abhängigkeit von der versendeten Nachricht gezeigt. In der Abbildung ist zu sehen, dass die drei ersten und die drei letzten Nachrichten schnell verifiziert sind, die sechs mittleren als Teil der Parallelaktivität nicht. Die jeweiligen Endnachrichten von Interaktionen werden immer ein wenig langsamer verifiziert als die Einstiegsnachrichten, weil das *Complete*-Fluent zusätzlich noch gesetzt werden muss. Zusätzlich muss für die Parallelaktivität immer jeweils getestet werden, ob es eine Teilaktivität gibt, welche mit dem Ereignis validiert. Für eine Sequenz ist dies schneller zu machen, da einfach die Sequenz in Axiom 4.23 sequenziell durchlaufen wird. Für die Parallelaktivität wird durch Backtracking in 4.28 jeweils probiert, ob es eine Teilaktivität gibt, für die die eingegangene Nachricht verifiziert. Daher ist die Verifikationszeit der Parallelaktivität leicht höher als die von Sequenzen, wie in der Abbildung auch zu sehen. Da die Choreographie ohne Schleifen und WorkUnits auskommt und nur eine optionale Aktivität enthält, greifen die meisten Regeln für die Sonderfälle nicht so oft wie beim ersten Szenario und daher sind im Schnitt die meisten Nachrichten schneller verifiziert.

In Abbildung 4.28(b) ist die Latenz in Abhängigkeit von der versendeten Nachricht in der Choreographie gezeigt. Für dieses Szenario folgen die Nachrichten nicht so schnell aufeinander, wie es beim ersten Szenario der Fall ist. Daher sind selten mehr als eine Nachricht in der Queue und die Verifikationszeiten der Nachrichten beeinflussen in diesem Fall nicht die Latenzen der anderen Nachrichten. Allerdings gilt auch hier, dass der Enterprise-Service-Bus nach erneutem Start erstmal seine Caches und entsprechende Sende-Queues initialisieren muss und damit am Start eine etwas höhere Latenz als erwartet besitzt. Wie in 4.28(b) zu sehen, sinkt die Latenz nach dem Start erstmal ab und pendelt sich dann bei etwas mehr als der Verifikationszeit der Nachricht ein.



(a) Validierungszeit (b) Latenz

Abbildung 4.29.: Abhängigkeit von gleichzeitigen Prozessinstanzen

Szenario 3 - Mehrere Instanzen zur selben Zeit: Im dritten Szenario wird das komplexere der beiden bisherigen Szenarien auf Skalierbarkeit untersucht. Dafür werden für das Käufer-Verkäufer-Szenario in mehreren Schritten immer mehr gleichzeitige Instanzen des Prozesses gestartet und die Zeit zur reinen Verifikation sowie die Latenz gemessen. Nach jedem Schritt, bei dem die Anzahl der parallelen Prozessinstanzen gesteigert wird, wird der Enterprise-Service-Bus, die einzelnen Beispieldienste und der Monitor neu gestartet, um gleichbleibende Testbedingungen zu gewährleisten.

Zu sehen ist, dass sich die Latenz bei bis zu 8 gleichzeitigen Instanzen wenig verändert. Ab einer Anzahl von 16 bis 32 gleichzeitigen Instanzen verändert sich die Latenz allerdings drastisch in den Minutenbereich. Auch die Verifikationszeit pro Nachricht geht in den Sekundenbereich. Die Latenz hängt also bei beiden Szenarien sehr stark davon ab, wie viele Nachrichten gleichzeitig in der Queue zur Verifikation sind. Eclipse-CLP selbst ist keine parallele Prolog-Umgebung; daher wird diese Umgebung im Schnitt wenig mehr als eine Anfrage zur Zeit pro Choreographie-Instanz bearbeiten können. Dieser Umstand macht sich auch in der Latenz bemerkbar, wenn die Übertragungszeiten von Nachrichten die Queue schneller füllen lassen, als die Prolog-Umgebung diese abarbeiten kann. Für die Messungen wird allerdings nur ein einziger Monitor betrieben, welcher alle Prozessinstanzen gleichzeitig verifiziert. Auf dem jeweiligen Testrechner ist sehr gut zu sehen, dass nur ein CPU-Kern mit der jeweiligen Verifikation beschäftigt ist. Das Ganze lässt sich noch deutlich optimieren, wenn ein Lastausgleich auf die anderen CPU-Kerne des Rechners ausgeführt wird, in dem beispielsweise pro CPU-Kern ein bis zwei Monitore gestartet werden. Jede Choreographie-Instanz lässt sich an einen dieser Dienste binden und damit die Auslastung erhöhen. Sind die CPU-Grenzen nicht mehr ausreichend, kann aufgrund der Web-Service-Eigenschaften der Monitor auch auf andere Rechner verteilt werden und damit die Skalierbarkeit gewährleisten. Anzupassen ist dafür nur der Enterprise-Service-Bus, der die jeweiligen Nachrichten an die dafür vorgesehene Instanz weiterleiten muss.

4.4. Diskussion und Zusammenfassung

In diesem Abschnitt ist zum einen die Infrastruktur zur Protokollierung von Ereignissen in Choreographien und zum anderen die Verifikation vorgestellt worden, um festzustellen, ob sich Teilnehmern innerhalb einer Choreographie so wie spezifiziert verhalten.

Die Anforderungen aus Abschnitt 3.1.3.1 an die technische Sensorinfrastruktur ist dabei mit kleinen Einschränkungen auf Basis eines Enterprise-Service-Busses fast vollständig umgesetzt worden. Zum einen wird die *Geschäftsdimension* einer Choreographie kaum beeinflusst, da durch Nutzung der Service-Bus-Technologie entsprechende Mediatoren zur Protokollierung zwischen den Teilnehmern eingebettet werden können und das Weiterleiten von Nachrichten im Bus an entsprechende Monitore so gut wie keine Verzögerung bei der normalen Nachrichtenübertragung zwischen den Teilnehmern verursacht. Zum anderen kann durch spezielle Korrelationsdaten in den Nachrichten für die Verifikation eine Beziehung zwischen Nachrichten und einer Choreographie-Instanz hergestellt werden. Damit sind auch die Anforderungen an die *Modelldimension* erfüllt. Problematisch daran ist nur, dass alle Teilnehmer diese Daten in den jeweiligen SOAP-Headern auch nutzen müssen. Ist dies nicht der Fall, kann keine Verifikation stattfinden. Weiterhin sind durch die Implementierung auch verschiedene Anforderungen an die *Architekturdimension* erfüllt. Die Konsistenz der Sensordaten wird durch Implementierung von Mediatoren sichergestellt, damit zum einen Protokoll und reale Ereignisse übereinstimmen (atomare Interaktionsprotokollierung) und zum anderen, dass alle Interaktionsmuster durch die Sensoren unterstützt werden und alle Ereignisse beobachtbar sind.

Tabelle 4.1.: Implementierung der Verifikationsansätze im Vergleich

	Modellinhärente Eigenschaften	**Inhaltliche Anforderungen**	**Reichweite: Nachricht**	**Reichweite: Interaktion**	**Reichweite: lokal**	**Reichweite: übergreifend**	**Quantifizierte Zeiten**	**Benutzbarkeit**
Regelbasierte Verifikation	(✓)	-	(✓)	(✓)	(✓)	(✓)	-	(✓)
Verifikation mit Ereigniskalkül	✓	✓	✓	✓	✓	✓	✓	(✓)

Auf Basis dieser Infrastruktur sind in diesem Abschnitt zusätzlich zwei unterschiedliche Verifikationsmechanismen vorgestellt worden, welche die Anforderungen aus Abschnitt 3.1.2.5 teilweise unvollständig oder ganz unterstützen. Der Vergleich der Implementierungen für die beide Ansätze in Bezug auf die Anforderungen an das formale Modell ist auch in Tabelle 4.1 dargestellt.

Der erste auf Prädikatenlogik basierende Ansatz kann den Kontrollfluss einer Choreographie sehr schnell zu überprüfen. Der Ansatz ist damit in der Lage, nicht zeitbezogene

Integritätsbedingungen des Kontrollflusses einer Choreographie zu verifizieren und unterstützt damit nicht alle modellinhärenten Eigenschaften von WS-CDL. Weiterhin unterstützt er nicht alle Reichweiten von Integritätsbedingungen, welche sich auf instanzbasierte und interaktionsbasierte Integritätsbedingungen beschränken. Technisch sind auch nachrichtenbasierte Integritätsbedingungen möglich. Inhaltliche Anforderungen und damit selbst definierte Integritätsbedingungen sind nicht vorgesehen, da keine zeitbezogenen Integritätsbedingungen möglich sind. Der Ansatz ist als erste Annäherung an das Ereigniskalkül implementiert, um auf der einen Seite einen Vergleichswert für eine Evaluation zu haben und um auf der anderen Seite erste Algorithmen für einen WS-CDL-Interpreter sowie das Verifizieren von hierarchisch strukturierten Prozessbeschreibungen zu realisieren. Die Evaluation ergibt, dass der Ansatz den Kontrollfluss einer Choreographie sehr schnell verifizieren kann und mit den genannten Einschränkungen für die Laufzeitverifikation geeignet ist.

Der zweite Ansatz basiert auf dem Ereigniskalkül und unterstützt damit quantifizierte Zeiten. Der Ansatz kann damit auch non-funktionale Integritätsbedingungen prüfen und unterstützt über bestimmte diskutierte Erweiterungspunkts auch inhaltliche, selbst definierte Anforderungen. Trotzdem gibt es bei der Abbildung von WS-CDL und damit den modellinhärenten Integritätsbedingungen einige leichte Einschränkungen für den Einsatz des Ereigniskalküls. Beispielsweise müssen Nachrichten in WS-CDL so modelliert sein, dass sie eindeutig zu ihrem Kontext zugeordnet werden können, denn sonst werden bei Auftreten einer Nachricht mehrere Basisaktivitäten auf valide gesetzt. Beispielsweise kann ein Nachrichtentyp *Bestätigung* mehrfach benutzt werden, aber trotzdem muss aus der Nachricht ersichtlich sein, ob eine Kauf- oder Versandbestätigung gemeint ist. Ist dies bei einem Choreographie-Modell nicht der Fall, lässt sich dies aber durch einige leichte Anpassungen der Nachrichten einfach beheben, sodass sich die Einschränkung im praktischen Betrieb nicht auswirkt. Zusätzlich gibt es im Prototypen bei der Abbildung von WS-CDL-Artefakten wie `Assign` oder `WorkUnit` Einschränkungen bei der Ausdrucksmächtigkeit in WS-CDL. Der Standard sieht beispielsweise bei der Definition von Variablen beliebige XPath-Ausdrücke vor, die schwer vollständig in das formale Modell überführbar sind. Hier werden im Prototyp nur einfache Konstrukte unterstützt, welche die Definition von Variablen für Schleifenbedingungen ermöglichen. Für weitere Möglichkeiten müssen das formale Modell und der WS-CDL-Interpreter erweitert und angepasst werden. Weiterhin sind bei der Implementierung des WS-CDL-Interpreters einige Konstrukte nicht abgebildet worden, da sich für diese Konstrukte keine neuen Aktivitäten oder Verifikationsschritte ergeben. Dies betrifft `Perform` sowie `Finalizer`- und `Exception`-Blöcke. Die letzten beiden verhalten sich technisch wie `WorkUnits`, welche in bestimmten Ausnahmesituationen aufgerufen werden[9]. Genauso verhält es sich mit der `Perform`-Aktivität, welche ein einfaches Inkludieren einer anderen Choreographie-Beschreibung in die aktuelle Beschreibung darstellt. Diese Unter-Choreographien müssen beim Interpreter unterstützt werden, welcher die Sub-Prozesse in

[9]Der Aufruf erfolgt, wenn der `Finalizer`- oder `Exception`-Guard wahr wird

die Choreographie einbettet. Die Verifikation läuft an dieser Stelle aber wie bereits beschrieben, hier sind keine neuen Aktivitäten für die Verifikation mit Regeln zu unterstützen. Abseits von diesen kleineren Einschränkungen in der technischen Benutzbarkeit des Prototypen unterstützt der Ansatz aber alle Anforderungen, die bisher ans formale Modell gestellt sind. Abschließend ist durch die Evaluation zu sehen, dass durch die Unterstützung von quantifizierten Zeiten und allen Arten von Reichweiten bei Integritätsbedingungen im Gegensatz zum einfachen Ansatz ein größerer Mehraufwand bei der Verifikation anfällt. Der einfache regelbasierte Ansatz liefert Verifikationszeiten von unter einer Millisekunde, während das Ereigniskalkül je nach Komplexität der Choreographie-Struktur im Bereich von 30-50 Millisekunden auf der Testmaschine liegt. Da bei organisationsübergreifenden Prozessen oft langlaufende Interaktionen stattfinden, ist der Bereich der Laufzeit vom Ereigniskalkül für die Laufzeitverifikation ausreichend.

Etwas schwierig ist das Erstellen von inhaltlichen Anforderungen zusätzlich zu den Anforderungen aus WS-CDL. Hier muss ein Modellierer genau wissen, wie die Abbildung in das interne Modell erfolgt, um eigene Integritätsbedingungen formulieren zu können. Daher ist die Benutzbarkeit des Ansatzes an dieser Stelle nicht optimal, kann aber durchaus durch eine Sprache zur Definition der Integritätsbedingungen und weitere Werkzeugunterstützung verbessert werden, um beispielsweise instanzübergreifende Integritätsbedingungen definieren zu können. Diese Werkzeuge müssen dann eine semantische Integritätsprüfung beinhalten, damit ein Modellierer sinnvolle und korrekte Modelle ausdrücken kann. Um einen vernünftigen Einsatz zu gewährleisten, sollten dafür auch entsprechende Visualisierungskonzepte umgesetzt werden, welche die Darstellung, Prüfung und später auch Fehlerdarstellung besser benutzbar machen. Die Verbesserung der Modellierbarkeit liegt allerdings eindeutig nicht im Fokus dieser Arbeit.

Die Implementierung des Verifikationsdienstes ist in soweit generisch, dass je nach Anforderungen an die Choreographie im Monitor die Verifikationsmechanismen einfach ausgetauscht werden können. Dafür kann je nach Grad der Integritätsbedingungen einer Choreographie ein Ansatz passend zur Choreographie ausgewählt werden. Sind keine Zeitschranken oder inhaltliche Bedingungen sowie instanzübergreifende Integritätsbedingungen für eine Choreographie angegeben, kann die einfache regelbasierte Verifikation den Kontrollfluss einer Choreographie entsprechend schnell verifizieren und die fehlerverursachenden Teilnehmer schnell identifizieren. Sind aber Zeitschranken oder komplexere Integritätsbedingungen wie instanzübergreifende Bedingungen für die Choreographie definiert, kann der Ansatz auf Basis des Ereigniskalküls diese Choreographien zur Laufzeit verifizieren.

Wie erkannte Abweichungen von den Integritätsbedingungen in Choreographie-Umgebungen behoben werden können, wird in den nächsten Abschnitt beschrieben. Dafür werden zum einen Lösungen zur Transaktionskontrolle in Choreographien vorgestellt sowie auf der anderen Seite Verknüpfungsmöglichkeiten zwischen der Transaktionskontrolle und Verifikation, damit eine koordinierte Backward-Recovery für bestimmte Integritätsverletzungen durchgeführt werden kann.

5. Ablaufkontrolle mit Hilfe von Koordinationsprotokollen

Damit eine Wiederherstellung bei Integritätsfehlern in Choreographien ermöglicht werden kann, muss dafür zum einen die Infrastruktur bereitgestellt werden, um Transaktionen zu koordinieren. Für diese Infrastruktur müssen nach Möglichkeit generische Koordinatoren bereitgestellt werden, welche zur Koordination von Transaktionen genutzt werden können. Generische Koordinatoren haben den Vorteil, dass keine anwendungsspezifischen Koordinatoren implementiert werden müssen. Dafür muss, wie in Abschnitt 3.2.2 diskutiert, die Anwendungslogik von der Koordinationslogik einer Transaktion getrennt sein. Generische Koordinatoren können dann beispielsweise über einen Enterprise-Service-Bus als Middleware bereitgestellt werden, um globale Transaktionen über verschiedene Choreographie-Teilnehmer zu koordinieren. Der Enterprise-Service-Bus wird damit zu einem transaktionalen Service-Bus. Zum anderen muss die Transaktionskontrolle, wie in der Anforderungsanalyse in Abschnitt 3.2.3 beschrieben, in der Lage sein, übliche Szenarien in Choreographien zu unterstützen. In diesen Szenarien ist der Initiator einer Transaktion nicht derjenige Teilnehmer, der über die Demarkation entscheidet. Für beide Teilbereiche werden in diesem Abschnitt der Arbeit Lösungen vorgestellt, in dem ein Rahmenwerk zur transaktionalen Kontrolle und zur Implementierung generischer Koordinatoren eingeführt und durch eine Implementierung realisiert wird.

Sind die Voraussetzungen zur transaktionalen Kontrolle durch die Infrastruktur gegeben, kann die Fehlererkennung aus dem Monitoring-Abschnitt genutzt werden, um im Falle der Abweichung von Integritätsbedingungen entsprechende Gegenmaßnahmen durch Recovery-Mechanismen der Transaktionskontrolle einzuleiten. Dafür werden im zweiten Teil dieses Abschnitts verschiedene Möglichkeiten der Integration der Fehlerkontrolle in die transaktionale Ablaufkontrolle diskutiert und realisiert. Eine Evaluation der integrierten Lösung schließt den Abschnitt zusammen mit einer Diskussion und Zusammenfassung ab.

5.1. Ein Rahmenwerk zur Kontrolle von transaktionalen Aktivitäten

Hinsichtlich der transaktionalen Koordination verteilter Prozesse und damit der Ablaufkontrolle verteilter Aktivitäten sind zwei fundamentale Aspekte zu beachten. Erstens ist die Frage zu klären, *wann ein Prozess beendet* werden soll. Nach Beenden der eigentlichen Arbeit in einem Prozess wird bei einer Transaktion die Abschlussphase eingeleitet, welche zum konsistenten Festschreiben der bisher geleisteten Arbeit führt (der „Wann"-Aspekt). Die zweite zu klärende Frage ist, *welche Teilnehmer festschreiben* und welche Teilnehmer ihre Arbeit rückgängig machen sollen (der „Wer"-Aspekt). In der Literatur wird diese Abschlussphase eines Prozesses mit dem Festlegen der Transaktionsgrenzen auch als Demarkation bezeichnet [AU02].

Werden WS-Coordination und die zugehörigen Koordinationstypen in Bezug auf diese beiden Aspekte hin untersucht, ist leicht erkennbar, dass für WS-AtomicTransaction (WS-AT) beide Aspekte klar durch das Completion- und das 2PC-Protokoll definiert sind[1]. Wenn der Abschluss einer Transaktion durch das Completion-Protokoll ausgelöst wird, führt der Koordinator ein 2PC-Protokoll mit den Teilnehmern der Transaktion durch, sodass für alle Aktivitäten der Teilnehmer Atomarität gewährleistet wird. Der Wer-Aspekt ist damit durch die Atomarität eindeutig definiert. Das Completion-Protokoll wird in der Regel durch einen Initiator einer Transaktion ausgeführt, auch wenn die Spezifikation nicht genau definiert, wer sich für dieses Protokoll bei einem Transaktionskoordinator anmelden kann. Der Initiator kann das Completion-Protokoll genau dann anstoßen, wenn er von allen aufgerufenen Teilnehmern die Resultate ihrer Arbeit bekommen hat. Damit ist der Wann-Aspekt eindeutig definiert, da der Zeitpunkt der Demarkation vom Initiator definiert wird.

Für langlebige und komplexe Prozesse ist bei WS-Coordination der Koordinationstyp WS-BusinessActivitites (WS-BA) vorgesehen. Für diesen Koordinationstyp gibt es zwei verschiedene Varianten. Zum einen kann ein Teilnehmer beim Protokoll BAPC[2] selbsttätig das Beenden seiner lokalen Arbeit dem Koordinator mitteilen. Er wartet danach auf die Entscheidung des Koordinators, ob er das Resultat der eigenen Arbeit festschreiben oder rückgängig machen soll. Zum anderen gibt es die Möglichkeit, dass der Koordinator im Protokoll BACC[3] den Teilnehmern mitteilt, dass sie ihre lokale Arbeit beenden sollen. Der Sinn der Unterscheidung liegt darin begründet, dass ein Teilnehmer entweder weiß, ob er noch weitere Arbeit im Prozess verrichten soll, oder nicht. Weiß er dies nicht, kann ihm der Koordinator mitteilen, dass nach Abschluss der aktuellen Arbeit keine weitere Arbeit im Rahmen der Transaktion verrichtet werden muss. Die Teilnehmer können nach der entsprechenden Nachricht ihre Arbeit beenden und warten danach auf den Koordinator, bis die Entscheidung zum Festschreiben der Arbeit mitgeteilt wird. Beide Protokollvarianten finden zwi-

[1] Siehe dazu auch Abschnitt 2.3.3.1.
[2] WS-BusinessActivity - BusinessAgreementWithParticipantCompletion (BAPC).
[3] WS-BusinessActivity - BusinessAgreementWithCoordinatorCompletion (BACC).

schen einem Teilnehmer und dem Koordinator statt. Dabei ist in beiden kein expliziter Zeitpunkt definiert, wann die Demarkation eingeleitet werden soll. Die WS-BA-Spezifikation lässt damit nicht nur die vorher genannten Aspekte über das Wer und das Wann undefiniert, sondern sie vermischt sie sogar: Beispielsweise müssen bei einem gemischten Ergebnis nicht alle Teilnehmer erfolgreich abschließen, damit eine Gesamttransaktion erfolgreich ist. Gemischte Ergebnisse kommen durch die Auswahl von Teilnehmern zustande, in dem etwa zwei Sitzplätze in verschiedenen Flügen reserviert werden, aber abschließend nur eine Reservierung final gebucht wird. Die anderen Reservierungen müssen dann entsprechend abgebrochen werden. Die Demarkation kann also dann eingeleitet werden, wenn alle *notwendigen* Teilnehmer einen *abgeschlossenen* Zustand erreicht haben. Damit ist der Wann-Aspekt an den Wer-Aspekt gekoppelt. Normales WS-BA definiert dabei aber weder, wie der Koordinator entscheiden kann, wann Teilnehmer beendet werden sollen, noch wird definiert, welche Teilnehmer für einen erfolgreichen Ausgang der Transaktion notwendig sind.

Den Umstand, dass die Wer- und Wann-Aspekte für WS-BusinessActivities nicht klar im Standard geregelt sind, versuchen verschiedene Ansätze mithilfe von Protokollen zu lösen, welche ähnlich zu dem Completion-Protokoll aus WS-AT funktionieren. Beispielsweise definieren Erven et al. in [EHHZ07] ein Web-Services-BusinessActivity-Initiator-Protocol zur Klärung der beiden Aspekte. Durch diese Protokolle wird dem Koordinator direkt mitgeteilt, welche Teilnehmer ihre Arbeit beenden oder rückgängig machen sollen und wann diese Anweisungen durch den Koordinator an die Teilnehmer gesendet werden. Problematisch bleibt dabei immer noch, dass nur der Initiator eines Prozesses die Demarkation einleiten kann. Wie in Abschnitt 3.2.2.2 diskutiert, gilt diese Annahme, dass ein kontrollierender Initiator in Choreographie-Umgebungen existiert, nicht in jedem Szenario. Im Kontrast zu bestehenden Ansätzen wird daher jetzt ein Rahmenwerk vorgestellt, in dem der Koordinator selbst entscheiden kann, wann die transaktionale Koordination zur erfolgreichen Beendigung eines Prozesses gestartet werden soll. Dafür muss es für den Koordinator eine genaue Definition der Wer- und Wann-Aspekte geben, welche im Folgenden genauer vorgestellt wird.

5.1.1. Der „Wann“-Aspekt

Damit der Koordinator entscheiden kann, wann die transaktionale Koordination zur Beendigung eines Prozesses eingeleitet werden soll, benötigt der Koordinator einen Überblick darüber, wie weit der aktuelle Prozess bereits fortgeschritten ist. Wenn alle Interaktionen im Prozess zwischen den Teilnehmern erfolgt sind, keine weiteren Teilnehmer mehr angesprochen werden, um der Transaktion beizutreten und jede Auswahl für gemischte Ergebnisse getroffen wurden, dann kann der Koordinator die Demarkation einleiten. Teilnehmer, die in dieser Phase ihre Arbeit noch nicht beendet haben, können nach WS-BA ihre lokale Arbeit noch innerhalb ihres *completing*-Zustands fertigstellen.

Um diese Gesamtübersicht über einen Prozess im Koordinator zu bilden, wird ein Auf-

rufbaum genutzt. Damit die Aufrufhierarchie gebildet werden kann, teilen einer Transaktion neu beigetretene Teilnehmer dem Koordinator jeweils den Identifikator ihres Aufrufers mit und wie viele Teilnehmer sie selbst noch aufrufen werden. Damit Teilnehmer im Aufrufbaum überhaupt identifiziert werden können, wird ein Identitätsrahmenwerk für WS-Coordination benötigt, welches diese Informationen in den bisherigen WS-Coordination-Nachrichten mit überträgt. Der Koordinator vergibt dabei den Identifikator für einen Teilnehmer bei Registrierung via `RegisterResponse`-Nachricht. Jede weitere durch Teilnehmer der Transaktion versendete Nachricht enthält dann in einem erweiterten `CoordinationContext` diesen Identifikator. Wenn ein Teilnehmer jetzt durch einen weiteren Teilnehmer mit dem Transaktionskontext aufgerufen wird, enthält der Kontext den Identifikator des Aufrufers, welcher dann in der `Register`-Nachricht an den Koordinator mit angegeben wird. Enthält eine `Register`-Nachricht an den Koordinator keinen Identifikator, wird der Teilnehmer als Wurzelknoten des Aufrufbaumes angesehen, was der Normalfall für den Initiator eines Prozesses ist. Die `Register`-Nachricht enthält zusätzlich noch die Anzahl der weiterhin vom beitretenden Dienst aufgerufenen Dienste. Werden keine weiteren Teilnehmer aufgerufen, ist der Dienst ein *stabiler Knoten* im Aufrufbaum. Im Kontrast dazu ist ein Dienst, der weitere Dienste aufruft, ein *Zwischenknoten* innerhalb der Aufrufhierarchie. Ausgehend vom Initiator resultieren die Web-Services-Interaktionen in einer Hierarchie von Dienstaufrufen, welche auch beispielhaft in Abbildung 5.1 zu sehen ist.

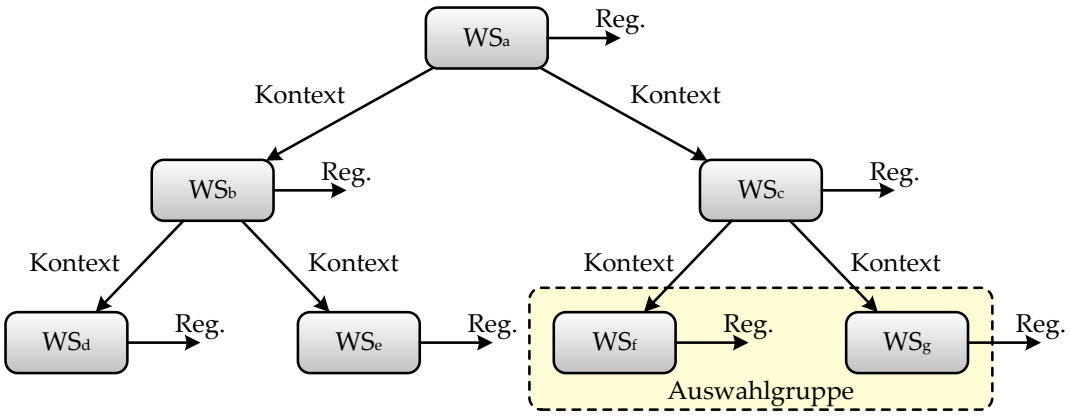


Abbildung 5.1.: Beispielhafter Aufrufbaum für Web-Services-Interaktionen

Um über die Demarkation bei einem Mixed-Outcome mithilfe des Aufrufbaumes zu entscheiden, werden zusätzlich noch *Auswahlgruppen* modelliert, um anzuzeigen, dass Dienste einer Auswahlgruppe parallel angesprochen werden können, aber nur eine Untermenge der Dienste erfolgreich beenden muss. Ein typisches Beispiel dafür sind die Käufer-Anbieter-Szenarien aus Kapitel 3.2.2. Dabei wird eine Menge von Diensten angesprochen, aber nur der günstigste Dienst für eine Bestellung ausgewählt. Allgemein sind Auswahlgruppen dabei nicht nur auf einfache „1 aus n“-Auswahlgruppen begrenzt, sondern können auch „m aus n“-Gruppen repräsentieren. Da das Auswahlkriterium stark anwendungsspe-

zifisch ist und von den ausgetauschten Nachrichten abhängt, kann der Koordinator ohne Anwendungswissen eine Auswahl nicht selbstständig entscheiden. Damit ein Koordinator Auswahlgruppen entscheiden kann, benötigt er also weitere Informationen. Eine einfache Möglichkeit zur Erlangung dieser Informationen sind die Initiatoren von Auswahlgruppen, welche dem Koordinator mitteilen, welche Teilnehmer in der Auswahlgruppe enthalten sind. Diese Details können huckepack via WS-Coordination-Nachrichten übertragen werden: Aufgerufene Dienste werden über den `CoordinationContext` informiert, dass sie Teil einer Auswahlgruppe sind, und können diese Information bei der Registrierung dem Koordinator mitteilen. Der Initiator einer Auswahl entscheidet dann, welche Teilnehmer der Auswahl zum erfolgreichen Ausgang der Transaktion erfolgreich beendet und welche abgebrochen werden sollen. Dieses Wissen kann der Initiator einer Auswahl über eine zusätzliche Nachricht dem Koordinator dann entsprechend mitteilen. Können die ausgewählten Teilnehmer ihre Arbeit nicht erfolgreich beenden, kann auch die Auswahl nicht erfolgreich beendet werden und die Transaktion wird abgebrochen.

Durch die interne Modellierung der Aufrufhierarchien im Koordinator und der Modellierung von Auswahlgruppen kann ein Koordinator über die Größe des Aufrufbaumes und darüber, welche Dienstaufrufe noch fehlen, entsprechend Buch führen. Wie oben beschrieben, müssen Teilnehmer zur Erzeugung der Aufrufhierarchie dem Koordinator mitteilen, wie viele weitere Teilnehmer sie noch einbinden werden. Ein Koordinator kann dann recht einfach bestimmen, wie viele Teilnehmer noch fehlen, in dem er nach jeder Registrierung eines Teilnehmer die Anzahl der bereits registrierten Teilnehmer N_{reg} und der noch erwarteten Teilnehmer N_{exp} bestimmt und entsprechend vergleicht. Ruft ein sich registrierender Teilnehmer keine weiteren Teilnehmer mehr auf und entspricht die Anzahl der erwarteten Teilnehmer der Anzahl der Registrierten, dann ist die maximale Ausdehnung der Aufrufhierarchie erreicht. Damit sind alle Blattknoten des Aufrufbaumes stabile Knoten und es können keine weiteren Teilnehmer mehr der Transaktion beitreten. Der Koordinator braucht ab diesem Zeitpunkt nur noch auf die Ergebnisse der Auswahlgruppen zu warten, bevor er autonom die Demarkation einleiten kann. Möglicherweise noch laufende Dienstaufrufe können innerhalb des normalen *Completing*-Zustands eines Teilnehmers später abgeschlossen werden.

Für die Entscheidung, wann die Demarkation eingeleitet wird, ist nun kein einzelner Choreographie-Teilnehmer mehr nötig, welcher eine globale Sicht auf den Prozess besitzt. Die jeweiligen individuellen lokalen Interaktionen und Auswahlgruppen werden im Koordinator zu einer globalen Sicht im Aufrufbaum aggregiert und tragen damit zur Beantwortung des Wann-Aspektes bei.

5.1.2. Der „Wer“-Aspekt

Um Teilnehmerzustände in WS-BusinessActivities zu behandeln, muss der Koordinator in mindestens zwei Phasen über den Teilnehmer entscheiden. Dafür muss der Koordinator

beispielsweise bei BACC über den „Wer“-Aspekt in einer ersten Phase entscheiden, ob ein Teilnehmer seine Arbeit erfolgreich beenden oder abbrechen muss. Bei BAPC teilen ihm die Teilnehmer das erfolgreiche Beenden ihrer Arbeit selbstständig mit. Die beiden Protokolle unterscheiden sich hier darin, ob der Koordinator den Teilnehmern mitteilt, was mit ihrer aktuellen Aktivität geschehen soll oder die Teilnehmer dies dem Koordinator mitteilen.

In einer zweiten Phase muss der Koordinator zusätzlich die Antworten der Teilnehmer aus der ersten Phase auswerten, ob sie ihre Arbeit erfolgreich beendet haben oder nicht. Abhängig vom Koordinationstyp (atomar oder gemischt) muss der Koordinator entscheiden, welche Nachrichten er den jeweiligen Teilnehmern danach sendet, damit sie ihre Arbeit entweder festschreiben oder kompensieren können. In dieser Phase benötigt der Koordinator explizit das Wissen über die Vitalität der Teilnehmer, um erkennen zu können, ob eine Transaktion erfolgreich beenden kann oder nicht. Die Vitalität eines aufgerufenen Web-Services beschreibt die Notwendigkeit, dass ein Teilnehmer erfolgreich seine Arbeit beendet, um die gesamte Transaktion erfolgreich beenden zu können. Standardmäßig ist jeder in eine Transaktion eingebundene Dienst vital, wenn er sich nicht explizit bei der Registrierung anders anmeldet. Diese Kennzeichnung eines Teilnehmers als „nicht vital“ kann durch einen aufrufenden Dienst erfolgen, der entscheiden kann, ob ein aufgerufener Web-Service für seine eigene Arbeit vital ist oder nicht. Ein Teilnehmer bekommt also die Eigenschaft der Vitalität durch seinen Aufrufer mitgeteilt und gibt den Status der Vitalität über seine eigene Registrierung an den Koordinator weiter.

Die beiden Phasen lassen sich also anhand von zwei Entscheidungen charakterisieren. Am Ende der ersten Phase entscheidet der Koordinator bei BACC über den Einfluss der Teilnehmer und damit, welcher Teilnehmer eine `Complete`- oder eine `Cancel`-Nachricht bekommt oder wartet auf das Beenden aller vitalen Teilnehmer, bevor Phase zwei eingeleitet wird. In der zweiten Phase wird die Entscheidung über den Erfolg der Transaktion anhand der beendeten Teilnehmer getroffen. Die beiden Entscheidungsphasen im Koordinator werden im folgenden Abschnitt noch weiter im Detail erläutert.

5.1.2.1. Phase 1: Entscheidung über den Einfluss eines Teilnehmers

Ist die Demarkation eingeleitet und sind damit alle erwarteten Teilnehmer registriert und alle Auswahlgruppen entschieden, kann der Koordinator in der ersten Phase von WS-BA die Gesamtmenge der Teilnehmer in eine *Complete-Menge* und eine *Cancel-Menge* einteilen. Die Semantik dieser Mengen entspricht denen der im Protokoll angegebenen Zustände: Teilnehmer aus der Complete-Menge sollen ihre Arbeit beenden und Teilnehmer aus der Cancel-Menge brechen Ihre Arbeit ab. Diese Mengen entsprechen also dem *möglichen Abschlussverhalten* der Teilnehmer am Ende einer Transaktion, basierend auf dem aktuellen Verhalten in der Aufrufhierarchie. Sie werden anhand des Fortschritts des Prozesses vom Koordinator eingeteilt, damit die Mengenzugehörigkeit jedes Teilnehmers jederzeit bekannt ist. Durch dieses Vorgehen über den Aufrufbaum sind die beiden „Wann“- und „Wer“-Aspekte der

transaktionalen Koordination effektiv voneinander getrennt. Die Beendigung eines Prozesses und damit das Einleiten der transaktionalen Koordination kann zu jeder Zeit erfolgen, da der Koordinator durch die Aufrufhierarchie alle notwendigen Informationen besitzt, um alle Teilnehmer zu einem erfolgreichen Ende oder zum Abbruch ihrer Arbeit zu bewegen.

Der Koordinator teilt dabei autonom und anwendungsunabhängig die Teilnehmer auf Basis der Interaktionen zwischen den Diensten in die Complete- und Cancel-Mengen ein. Die Interaktionen werden mithilfe des Aufrufbaumes im Koordinator abgebildet. Trotzdem werden, wie vorher angedeutet, anwendungsspezifische Entscheidungen für die Auswahlgruppen von Teilnehmern benötigt, die dem Koordinator mithilfe von Nachrichten mitgeteilt werden müssen. Nur auf diese Weise bleibt der Koordinator anwendungsunabhängig: Durch das Auslagern von Anwendungsentscheidungen in die Teilnehmer wird es ermöglicht, auch komplexe Anwendungslogik für die transaktionale Koordination zu benutzen, während die eigentliche Koordinationsarbeit vom Koordinator erledigt wird.

Um den Aufrufbaum zu modellieren und die Teilnehmer in die entsprechenden Mengen einzuteilen, lassen sich entsprechende Regeln definieren. Zur Definition dieser Regeln wird eine Menge von Prädikaten benötigt, um das Verhalten des Koordinators in Bezug auf die Complete- und Cancel-Mengen zu modellieren:

1. **Prädikate für Teilnehmer:** Um einen Teilnehmer zu modellieren, wird das Prädikat $participant(TX, A)$ genutzt, um die Zugehörigkeit von Teilnehmer A zur Transaktion TX anzuzeigen. Weiterhin sind Fakten notwendig, die den Zustand eines Teilnehmers im Rahmen des Protokolls kennzeichnen. Gekennzeichnet werden Teilnehmer bei Austritt mit $exit(TX, A)$. Bei Unfähigkeit, die eigene Arbeit erfolgreich zu beenden, werden sie mit $cannotComplete(TX, A)$ annotiert. Genauso werden Teilnehmer, die sich selbst als nicht-vital registriert haben, mit $invital(TX, A)$ ausgezeichnet.

2. **Prädikate für den Aufrufbaum:** Um den Aufrufbaum zu modellieren, müssen Web-Services-Aufrufe modelliert werden. Das Prädikat $invocation(TX, A, B)$ zeigt dabei an, dass ein Web-Service A einen anderen Web-Service B im Rahmen der Transaktion TX aufgerufen hat.

3. **Prädikate für Auswahlgruppen:** Um Auswahlgruppen zu beschreiben, wird ein $partOfChoice(TX, A)$-Prädikat genutzt. Dieses Prädikat definiert als Fakt, dass ein Teilnehmer A Teil einer Auswahl im Rahmen der Transaktion TX ist. Zusätzlich wird ein Prädikat zur Auszeichnung von Teilnehmern benötigt, um anzuzeigen, dass ein Teilnehmer A ausgewählt wurde. Dafür wird das Prädikat $chose(TX, A)$ genutzt.

4. **Prädikate für Complete- und Cancel-Mengen:** Die Gruppenzugehörigkeit zu den Complete- und Cancel-Mengen wird durch die Prädikate *cancel(TX,A)* und *complete(TX,A)* als entsprechende Fakten in einer Wissensbasis beschrieben.

Mit Unterstützung dieser Prädikate lassen sich nun einfach Regeln definieren, welche die Teilnehmer in die korrespondierenden Mengen einteilen. Ein Teilnehmer A wird dabei der

Cancel-Menge zugewiesen, wenn er von einem Teilnehmer V aufgerufen wird, der schon in der Cancel-Menge enthalten ist (5.1) oder wenn der Teilnehmer A als Teil einer Auswahl nicht ausgewählt wird (5.2). Zusätzlich muss bei beiden Regeln noch geprüft werden, ob der Vaterknoten nicht schon „ausgestiegen" ist oder seine Arbeiten nicht beenden kann. Genauso darf der zu prüfende Knoten nicht selbst schon ausgestiegen sein oder nicht unfähig sein, seine Arbeit zu beenden.

$$\begin{aligned} cancel(TX, A) \leftarrow\ & invocation(TX, V, A) \wedge \\ & \neg exit(TX, A) \wedge \neg cannotComplete(TX, A) \wedge \\ & (cancel(TX, V) \vee exit(TX, V) \vee cannotComplete(TX, V)) \end{aligned} \tag{5.1}$$

$$\begin{aligned} cancel(TX, A) \leftarrow\ & partOfChoice(TX, A) \wedge \neg chose(TX, A) \wedge \\ & \neg exit(TX, A) \wedge \neg cannotComplete(TX, A) \end{aligned} \tag{5.2}$$

Teilnehmer, die nicht Elemente der Cancel-Menge sind, sind damit automatisch Elemente der Complete-Menge, wenn sie nicht selbst ausgestiegen sind oder ihre Arbeit nicht beenden können.

$$\begin{aligned} complete(TX, A) \leftarrow\ & participant(TX, A) \wedge \neg exit(TX, A) \wedge \\ & \neg cannotComplete(TX, A) \wedge \neg cancel(TX, A) \end{aligned} \tag{5.3}$$

Für normale Dienstaufrufe gilt also, dass ein Dienst in die Complete-Menge eingeteilt wird, wenn der Vaterknoten nicht durch einen bestimmten Umstand in die Cancel-Menge eingeteilt wird. Dies ist auch der häufigste Fall, denn der Initiator eines Prozesses ruft weitere Dienste mit einem neu erzeugten Koordinationskontext auf. Wenn diese aufgerufenen Dienste selbstständig weitere Dienste aufrufen, wird normalerweise der Koordinationskontext weitergegeben, damit sich die neu aufgerufenen Dienste beim Koordinator registrieren können. Konsequenterweise wird dabei der Aufrufbaum durch die Registrierung der Teilnehmer aufgebaut, wie etwa in Abbildung 5.1 zu sehen. Sind keine Auswahlgruppen vorhanden und steigt kein Teilnehmer aus der Transaktion aus, sind alle Teilnehmer in der Complete-Menge enthalten. Im Gegensatz dazu ist ein Teilnehmer aus der Cancel-Menge ein passivierter Teilnehmer, welcher nur im Prozess verbleibt, um ein geordnetes Ergebnis sicherzustellen. Das Aufrufen weiterer Teilnehmer macht für einen solchen Teilnehmer wenig Sinn. Geschieht dies aber dennoch, werden diese aufgerufenen Teilnehmer entsprechend über Regel 5.1 auch der Cancel-Menge hinzugefügt. Zusammenfassend lässt sich festhalten, dass die Mitgliedschaft in einer der beiden Mengen jeweils von den Aufrufern an den Aufgerufenen propagiert werden. Dieses Verhalten ist vor allem wichtig, wenn Teilnehmer durch Anwendungsentscheidungen bei einer Auswahl in die Cancel-Menge sortiert werden. In diesem Fall werden alle Teilnehmer des darunterliegenden Teilbaumes auch in die Cancel-Menge verschoben.

Entsprechend zum $partOfChoice()$ Prädikat in Regel 5.2 können auch Auswahlgruppen

durch den Koordinator entschieden werden. Wenn der kontrollierende Teilnehmer einer Auswahlgruppe die Entscheidung trifft, welche Teilnehmer festschreiben und welche beenden sollen, so teilt er diese Entscheidung mithilfe einer neu eingeführten `Chose`-Nachricht dem Koordinator mit. Der Koordinator teilt dann die unausgewählten Teilnehmer (5.2) und deren Teilaufrufbäume (5.1) entsprechend der Cancel-Menge zu.

Um das Verfahren zu verdeutlichen, kann Abbildung 5.1 betrachtet werden. Die Aufrufhierarchie enthält eine Auswahlgruppe, welche von Web-Service c erzeugt wird und aus den Diensten f und g besteht. Wenn der Aufrufer c den Teilnehmer f auswählt, dann wird automatisch Teilnehmer g mithilfe der Regeln der Cancel-Menge zugeordnet. Der Koordinator ist also in der Lage, die Teilnehmer anhand von entsprechend mitgeteilten Anwendungsentscheidungen in die jeweiligen Mengen für den weiteren Protokollverlauf einzusortieren. Mitglieder der Cancel-Menge erhalten dann eine `Cancel`-Nachricht, Mitglieder der Complete-Menge entsprechend eine `Complete`-Nachricht, wenn das Protokoll BACC genutzt wird.

5.1.2.2. Phase 2: Entscheidung über den Erfolg der Transaktion

Innerhalb der ersten Phase entscheidet der Koordinator, welche Teilnehmer ihre Arbeit beenden und welche abbrechen sollen. Wenn alle abzubrechenden Teilnehmer ihre Arbeit zurückgesetzt und alle zu beendenden Teilnehmer ihre Arbeit fertiggestellt haben, dann kann der Koordinator entscheiden, ob die Transaktion erfolgreich beendet werden kann oder nicht. Der Koordinator sendet dafür entweder `Close`- oder `Compensate`-Nachrichten an die entsprechenden Teilnehmer, je nachdem welche Entscheidung getroffen wurde.

Die Entscheidung, ob eine Transaktion erfolgreich abgeschlossen werden kann oder nicht, wird mithilfe der Aufrufhierarchie aus dem letzten Abschnitt entschieden. Der Koordinator muss dabei entscheiden, ob ein fehlerhafter Teilnehmer in einer kaskadierenden Kompensation jeder Aktivität einer Transaktion resultiert. Zur Entscheidung, ob eine Transaktion erfolgreich beendet werden kann, benötigt der Koordinator das Wissen über die Vitalität der Teilnehmer. Kann er die Vitalität der Teilnehmer bestimmen, lässt sich der Erfolg einer Transaktion einfach bestimmen, denn wenn alle vitalen Teilnehmer in der Complete-Menge enthalten sind, lässt sich die Transaktion erfolgreich beenden, wie es auch in Regel 5.4 dargestellt ist.

$$finishing(TX) \leftarrow (\forall X)(vital(TX, X) \rightarrow complete(TX, X)) \tag{5.4}$$

Abbildung 5.2 zeigt das Prinzip der Einteilung in die jeweiligen Mengen. Die besonderen Vorkommnisse im Aufrufbaum sind das Aussteigen von Web-Service WS_e über eine `Exit`-Nachricht sowie das Auswählen des Web-Services WS_f im Rahmen einer Auswahlgruppe. Die Regel 5.2 teilt zum Entscheidungszeitpunkt WS_g in die Cancel-Menge ein. Die restlichen Dienste werden bis auf WS_e nach Regel 5.3 in die Complete-Menge eingeteilt. Die Menge

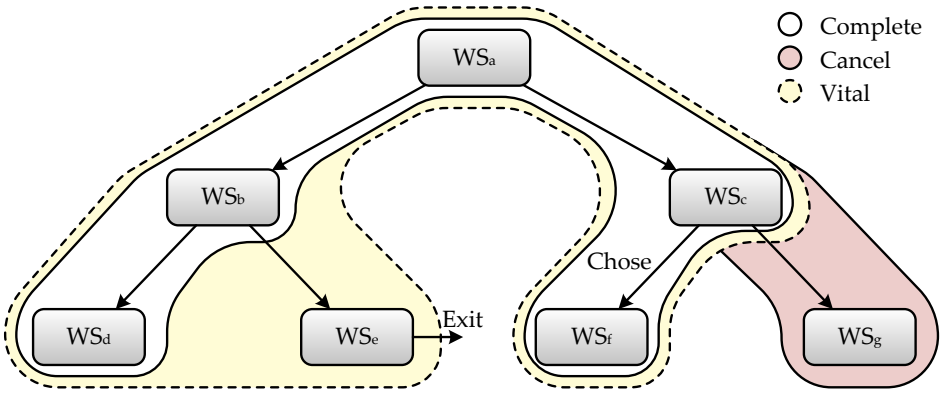


Abbildung 5.2.: Complete-, Cancel- und Vital-Mengen eines Aufrufbaumes

der zum erfolgreichen Abschluss der Transaktion benötigten Web-Services ist als Menge der vitalen Teilnehmer gekennzeichnet. Sind nicht alle vitalen Teilnehmer in der Complete-Menge, kann die Transaktion nach Regel 5.4 nicht erfolgreich abschließen. Der Koordinator stellt damit sicher, dass bei Erfolg alle vitalen Teilnehmer erfolgreich abschließen, und garantiert in Anlehnung an die Atomarität beim 2PC-Protokoll eine *Vitalitätsatomarität* der Aktivitäten beteiligter Web-Services.

Damit diese Regel arbeiten kann, benötigt der Koordinator Wissen über die Vitalität der Teilnehmer. Auch dies kann mithilfe der Aufrufhierarchie bestimmt werden. Als Grundannahme gilt, dass der Wurzelknoten im Aufrufbaum immer vital ist. Aus Sicht eines Geschäftsprozesses muss der Wurzelknoten aber nicht notwendigerweise dem Initiator des Prozesses entsprechen. Dieser kann beispielsweise einen anderen Teilnehmer aufrufen, der dann eine Transaktion startet; die so gestartete Transaktion ist *unabhängig* vom Aufrufer. Schlägt die Arbeit des Initiators fehl, kann die unabhängige Transaktion trotzdem erfolgreich beenden. Der Transaktionsstarter bildet den Wurzelknoten des Aufrufbaumes aus Sicht des Koordinators und ist vital, da hier die Grundannahme gilt, dass die aufgerufenen Teilnehmer notwendig für die Arbeit des Wurzelknotens sind. Steigt dieser aber mit einer `CannotComplete`- oder `Exit`-Nachricht aus der Transaktion aus, ist die von ihm angestoßene Arbeit bei den anderen Diensten nicht mehr relevant. Sicherlich können an dieser Stelle für die WS-BA-Protokolle wie beim WS-AT-Protokoll noch Optimierungen im Protokoll eingeführt werden, in dem Teilnehmer ausgezeichnet werden, die nicht am Ausgang der Transaktion interessiert sind. Dies kann mithilfe einer speziellen `CompleteExit`-Nachricht kenntlich gemacht werden. Allerdings ist auch in diesem Fall der Wurzelknoten vital für den Ausgang der Transaktion. Regel 5.5 definiert genau diesen Umstand, dass der Wurzelknoten immer vital ist. Genauso sind Zwischenknoten vital, wenn sie von einem vitalen Teilnehmer aufgerufen werden und nicht Teil einer Auswahl sind (Regel 5.6) und sich nicht selbst als invital beim Koordinator registriert haben. Teilnehmer als Teil einer Auswahl sind nur vital, wenn sie von einem vitalen Teilnehmer ausgewählt wurden (Regel 5.7).

$$vital(TX, A) \leftarrow participant(TX, A) \wedge \neg invocation(TX, \_, A). \tag{5.5}$$

$$vital(TX, A) \leftarrow invocation(TX, V, A), \neg partOfChoice(TX, A) \wedge \neg invital(TX, A) \wedge vital(TX, V) \tag{5.6}$$

$$vital(TX, A) \leftarrow invocation(TX, V, A) \wedge partOfChoice(TX, A) \wedge chose(TX, A) \wedge vital(TX, V) \tag{5.7}$$

Ist die Vitalität der Teilnehmer durch die Regeln bestimmt, kann der Koordinator mithilfe der Complete- und Cancel-Menge und Regel 5.4 bestimmen, ob die Transaktion erfolgreich ist oder nicht. Der Zeitpunkt der Entscheidung hängt wie in der ersten Phase von den Antworten der Teilnehmer ab. Haben alle Teilnehmer auf eine `Complete`-Nachricht entweder mit `Completed`, `Exit` oder `CannotComplete` geantwortet, kann der Koordinator über $finishing(TX)$ den Ausgang der Transaktion bestimmen und auf Basis dieser Entscheidung den entsprechenden Teilnehmern nun `Close`- oder `Compensate`-Nachrichten senden. Im Erfolgsfall bekommen diejenigen Teilnehmer eine `Close`-Nachricht, welche in der Complete-Menge enthalten sind und dem Koordinator mitgeteilt haben, dass sie ihre Arbeit beendet haben (Regel 5.8). Dementsprechend müssen alle Teilnehmer ihre Arbeit kompensieren, die nicht Teil der Complete-Menge sind und dem Koordinator bereits mitgeteilt haben, dass ihre Arbeit abgeschlossen ist (Regel 5.9). Dies kann beispielsweise im Protokoll BAPC geschehen, in dem Teilnehmer den Abschluss ihrer Arbeit selbstständig mitteilen. Falls die Transaktion nicht erfolgreich ist, müssen alle Teilnehmer, die ihre Arbeit als erfolgreich beendet gemeldet haben, diese Arbeit kompensieren (Regel 5.10). Dies gilt für beide Protokolle BACC und BAPC in gleicher Weise.

$$close(TX, A) \leftarrow finishing(TX) \wedge completed(TX, A) \wedge complete(TX, A) \tag{5.8}$$

$$compensate(TX, A) \leftarrow finishing(TX) \wedge completed(TX, A) \wedge \neg complete(TX, A) \tag{5.9}$$

$$compensate(TX, A) \leftarrow \neg finishing(TX) \wedge completed(TX, A) \tag{5.10}$$

Durch die Regeln, den Aufrufbaum und die Auswahlgruppen ist der Koordinator jetzt in der Lage, sehr viel besser durch die Einbeziehung der Vitalität über die Teilnehmer einer Transaktion zu entscheiden. Dabei ist auch die Entkopplung von Anwendungslogik und Koordinationslogik durch die Annotierung der Vitalität von Teilnehmern und durch die eindeutige Trennung der „Wer"- und „Wann"-Aspekte einer Transaktion besser geregelt. Mit Unterstützung der Regeln und des Aufrufbaumes lassen sich jetzt generische Koordinatoren realisieren, die über Teilnehmer auf Basis von mitgeteilten Anwendungsentscheidungen für Auswahlgruppen autonom entscheiden können.

In diesem Abschnitt sind nur die Arten der Entscheidungen und wie diese Entscheidungen unterstützt werden definiert. Die Zeitpunkte der Entscheidungen sind nur vage definiert, sodass eine präzise Definition fehlt, wann ein Koordinator und damit genauer in welchem Zustand er die Entscheidungen treffen muss. Die Zustände eines Koordinators ergeben sich dabei aus den Protokollzuständen und -abläufen der jeweiligen Spezifikationen.

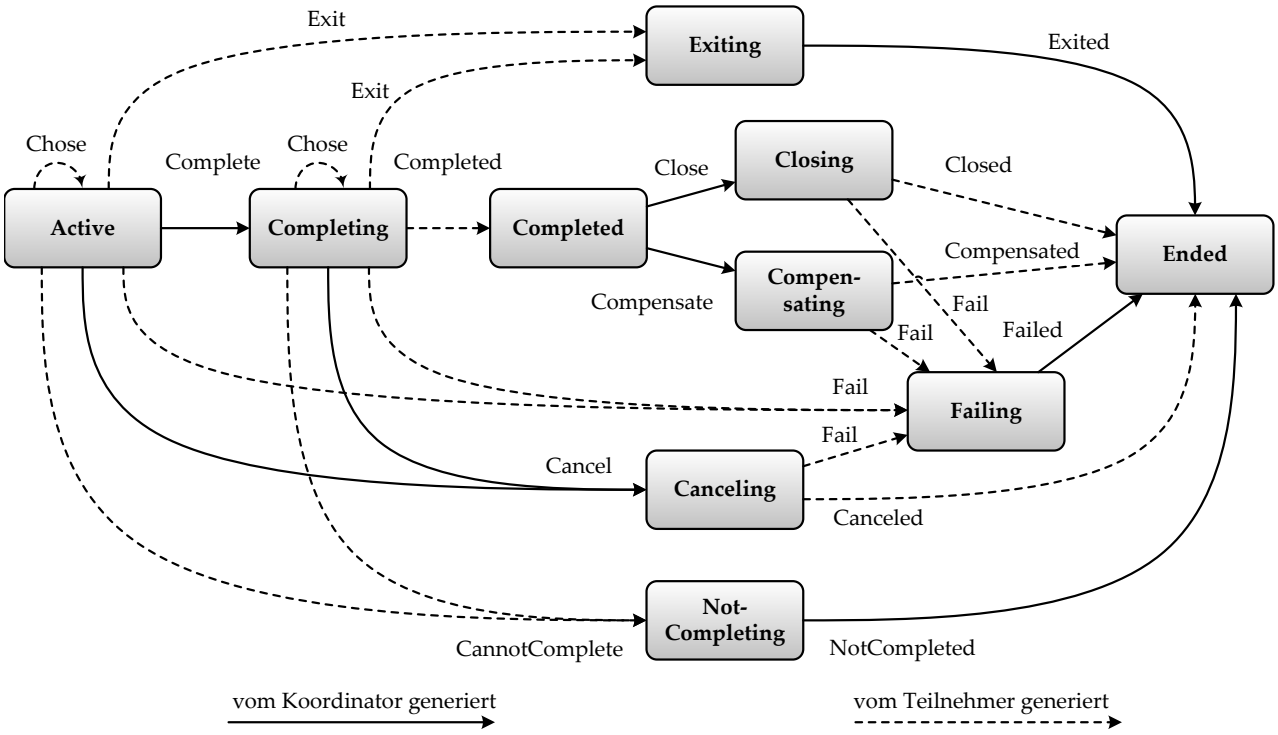


Abbildung 5.3.: BusinessAgreementWithCoordinatorCompletion [FL09], erweitert um `Chose`- und `Fail`-Nachrichten

Allerdings sind diese nur jeweils zwischen dem Koordinator und einem einzigen Teilnehmer definiert, und nicht als globale Zustände eines Koordinators. Globale Zustände sind aber wesentlich für eine Implementierung eines Koordinators, weshalb die Protokollzustände im folgenden Abschnitt noch genauer diskutiert werden.

5.1.3. Die Verwaltung des Transaktionsfortschritts

Das in WS-BusinessActivity (WS-BA) definierte Koordinationsprotokoll repräsentiert den Austausch von Nachrichten und damit die *Konversation* zwischen Koordinator und einem einzelnen Teilnehmer. Die Abbildung 5.3 zeigt die *Konversationszustände* des BusinessAgreementWithCoordinatorCompletion-Protokolls, erweitert um die `Chose`-Nachricht aus Kapitel 5.1.2.1 und einer zusätzlichen `Fail`-Nachricht, welche in Kapitel 5.1.3.2 noch eingeführt wird. Der Koordinator muss bei Ausführung des Protokolls die Zustände vieler Konversationszustände von verschiedenen Teilnehmern behandeln. WS-BA spezifiziert zwar die Zustandsübergänge einzelner Teilnehmer, aber nicht direkt, wie der Prozess im Ganzen behandelt wird, und damit fehlt die genaue Spezifikation, wie der globale Zustand des Koordinators sich in Abhängigkeit der individuellen Konversationen verhält.

Innerhalb von WS-AtomicTransaction ist die Beziehung aus Zuständen von einzelnen Konversationen und dem globalen Zustand offensichtlich. Der globale Koordinatorzustand, der sich aus den einzelnen Konversationszuständen zusammensetzt, ist durch das Comple-

tion-Protokoll explizit definiert. Wenn der Initiator der Transaktion den Koordinator mit einer `Commit`-Nachricht in den *Completing*-Zustand versetzt, dann führt der Koordinator ein Zwei-Phasen-Commit-Protokoll (2PC) mit den Teilnehmern der Transaktion aus und teilt das Ergebnis der Koordination dem Initiator mit. Innerhalb des 2PC-Protokolls durchlaufen die Teilnehmer dabei einzeln verschiedene Zustände, bis letztendlich die Zustände der Teilnehmer effektiv gleich sind und entsprechend zum globalen Zustand des Koordinators passen.

Im Gegensatz dazu ist die Betrachtung des globalen Zustandes eines Koordinators in WS-BA deutlich komplexer, da die Teilnehmer individuell durch die Zustände des Koordinationsprotokolls laufen und dabei auch heterogene Endzustände erreichen. Bei der Protokollausführung müssen sie trotzdem gemeinsam zu einem konsistenten Prozessausgang koordiniert werden. Ein erfolgreicher Prozess mit einem konsistenten Ergebnis heißt in diesem Fall nicht, dass alle Teilnehmer ihre Arbeit erfolgreich abschließen müssen. Innerhalb von Szenarien mit einem Mixed-Outcome müssen beispielsweise nur eine bestimmte Menge von Teilnehmern erfolgreich sein, damit der Gesamtprozess erfolgreich ist.

5.1.3.1. Globale Koordinatorzustände

Der *globale Transaktionszustand* definiert sich als Aggregation der individuellen Konversationszustände aufseiten des Koordinators. Der globale Zustand ergibt sich dabei aus den Zuständen der individuellen Konversationen und den ausstehenden Ereignissen. Er kann dabei als Zustand des Koordinators in Bezug auf alle Konversationen interpretiert werden, welchen der Koordinator gegenüber den individuellen Teilnehmern einnimmt. Das Verhalten des Koordinators und seine Reaktionen auf Nachrichten der Teilnehmer ergeben sich damit aus diesem globalen Zustand. Im Gegensatz dazu kennen die Teilnehmer nur ihren eigenen individuellen Konversationszustand, erlangen aber kein Wissen über den globalen Zustand. Die genaue Definition eines globalen Koordinatorzustandes ist aber wesentlich zur Implementierung eines Koordinators, der anhand der einzelnen Konversationszustände Entscheidungen treffen muss. Beispielsweise definieren die globalen Zustände eines Koordinators sowie deren Zustandsübergänge auch die Zeitpunkte, wann der Koordinator welche Entscheidungen treffen muss.

Bei Betrachtung der Konversationszustände in Abbildung 5.3 lassen sich fünf verschiedene globale Zustände identifizieren, wie in Abbildung 5.4 und 5.5 dargestellt. Die Knoten in dieser Abbildung repräsentieren die globalen Zustände *PREPARING*, *FINISHING*, *ABORTING* und *ENDED*, welche zusammen mit ihren möglichen Konversationszuständen zwischen Koordinator und Teilnehmern gezeigt sind. Zusätzlich gibt es noch den *FAULTED*-Zustand, welcher bei kritischen Fehlern eingenommen wird. Die Kanten im Graphen repräsentieren die durch die WS-BusinessActivities-Spezifikation definierten Nachrichten, welche vom Koordinator oder den Teilnehmern gesendet werden. Dabei ist in Abbildung 5.5 die Variante eines gemischten Ergebnisses gezeigt, bei dem Teilnehmer beispielswei-

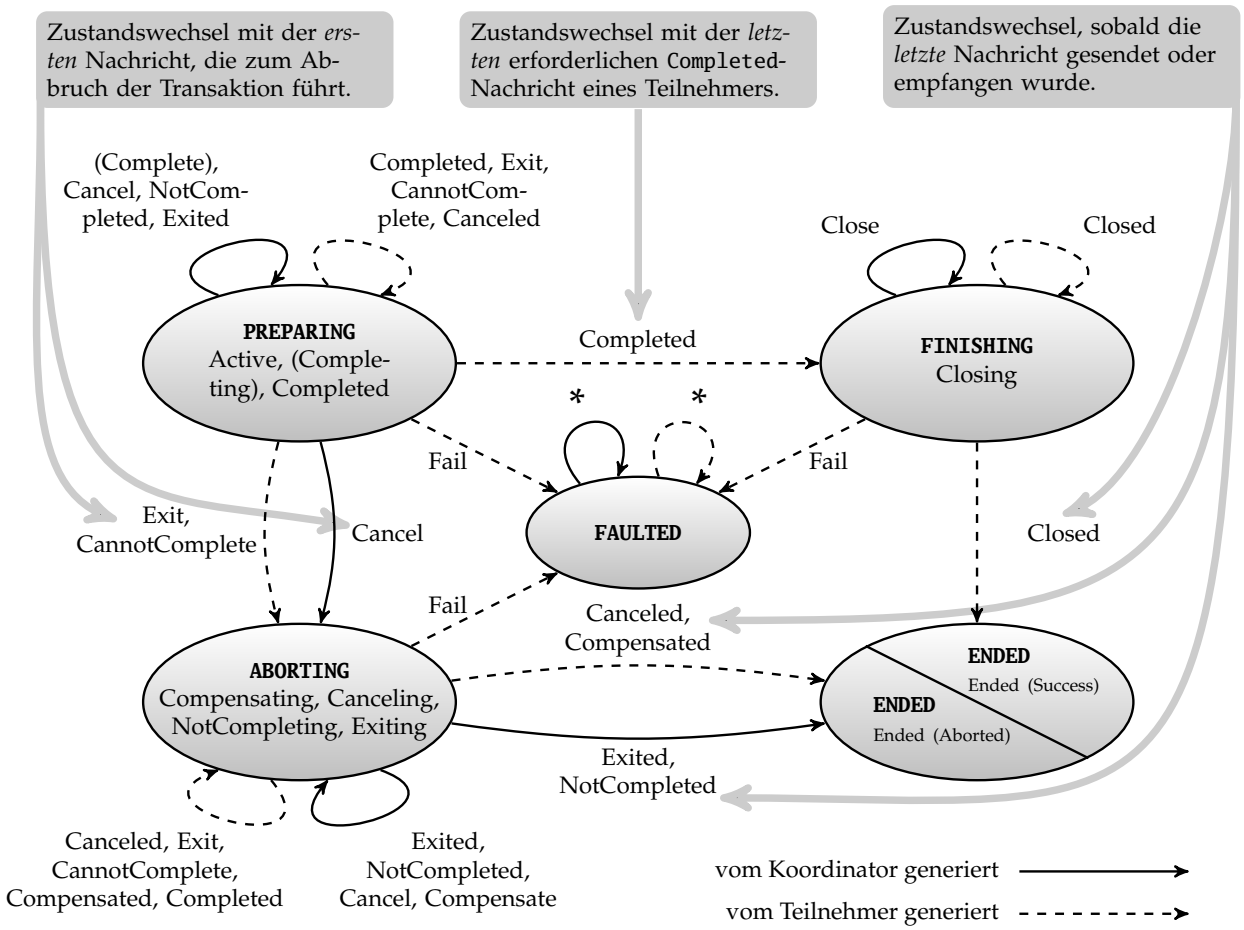


Abbildung 5.4.: Koordinatorzustände für BAPC und (BACC) mit Atomic-Outcome

se auch im Koordinatorzustand *FINISHING* noch eine `Canceled`-Nachricht senden können und als Teilnehmer selbst im Zustand *Canceling* sein dürfen. Im atomaren Fall ist dies natürlich nicht möglich, wie in Abbildung 5.4 zu sehen. Im atomaren Fall gibt es keine Auswahlgruppen und daher auch keine `Chose`-Nachrichten. Ist der Koordinator also im Zustand *FINISHING*, so kann er nur noch `Closed`-Nachrichten oder im Falle eines fatalen Fehlers `Fail`-Nachrichten der Teilnehmer erhalten. Einige der Nachrichten initiieren eine globale Zustandsänderung, welche in den jeweiligen Boxen in Abbildung 5.5 über dem Graphen genau beschrieben ist.

Der PREPARING-Zustand enthält bei einem Mixed-Outcome (Abbildung 5.5) hauptsächlich die vorherrschenden Konversationszustände *Active*, *Completing*, *Completed* und *Canceling*. Er enthält zum einen die aktive Phase des Prozesses, in der die Teilnehmer ihre Arbeit verrichten, und zum anderen die erste Phase der transaktionalen Koordination, welche eingeleitet wird, wenn der Koordinator, wie in Abschnitt 5.1.2.1 beschrieben, `Complete`- und `Cancel`-Nachrichten nach der Entscheidung über die Complete- und Cancel-Mengen versendet. Diejenigen Teilnehmer, welche Auswahlgruppen erzeugt haben, müssen das Ergebnis einer Auswahl via `Chose`-Nachricht an den Koordinator senden, bevor sie dem Koordi-

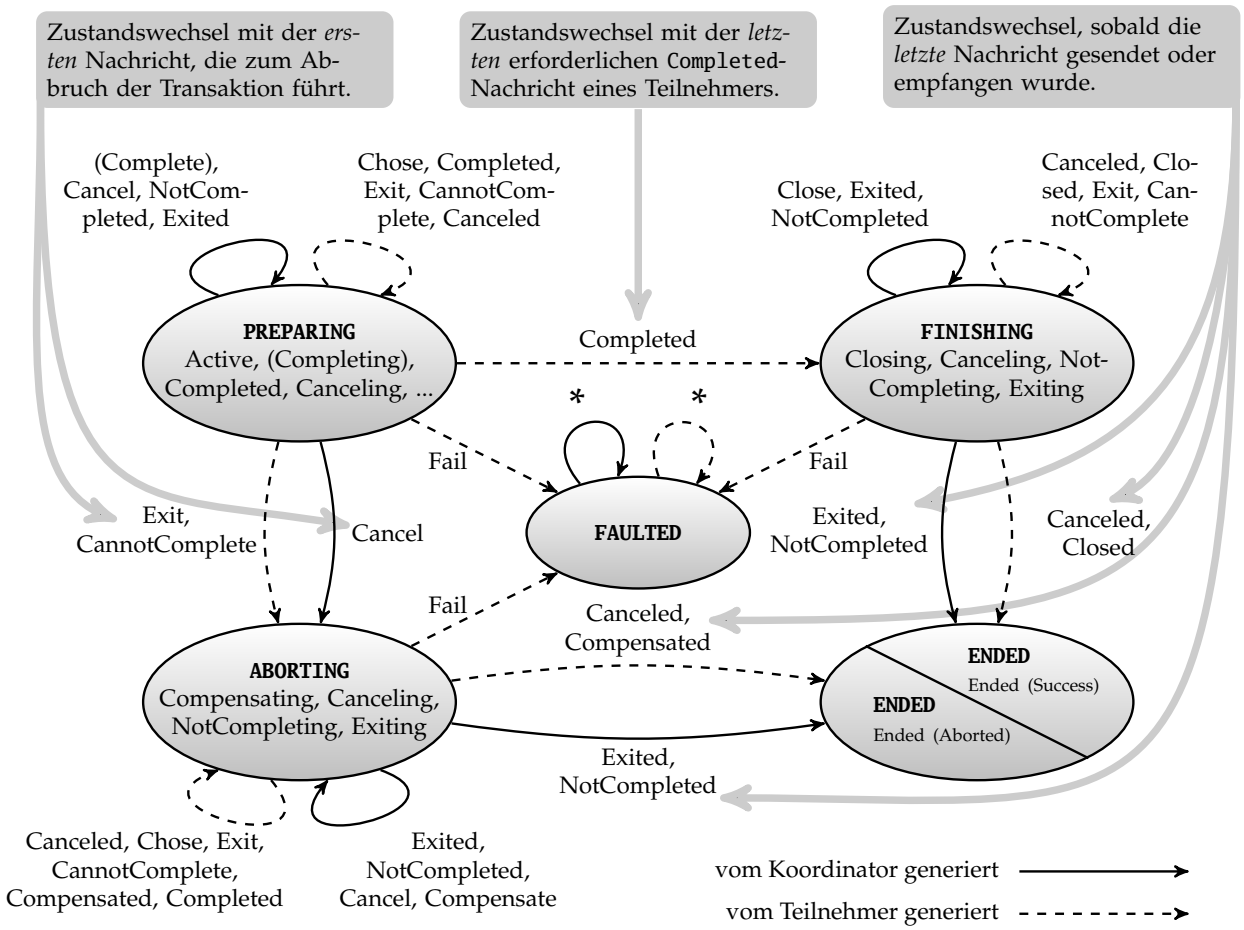


Abbildung 5.5.: Koordinatorzustände für BAPC und (BACC) mit Mixed-Outcome

nator selbst eine `Completed`-Nachricht senden können, denn ihre eigene Arbeit kann nicht abgeschlossen sein, wenn es noch unentschiedene Auswahlmengen gibt. Der PREPARING-Zustand bei einem Atomic-Outcome unterscheidet sich dazu nicht wesentlich (siehe Abbildung 5.4). Im atomaren Fall existieren keine Auswahlgruppen, wodurch es keine Konversationszustände von nicht ausgewählten Teilnehmern im PREPARING-Zustand des Koordinators geben kann. Daher führt der Austritt eines Teilnehmers oder die Unfähigkeit eines Teilnehmers, die Arbeit zu beenden, sofort in den Koordinatorzustand ABORTING.

Der globale Zustand ändert sich zu FINISHING, wenn alle Mitglieder der Complete-Menge eine `Completed`-Nachricht an den Koordinator gesendet haben. Dies geschieht beispielsweise, wenn die letzte erwartete `Completed`-Nachricht eines Teilnehmers den Koordinator erreicht hat. Der FINISHING-Zustand entspricht also der ersten Variante der zweiten Phase des transaktionalen Abschlusses, in der die Transaktion erfolgreich beendet werden kann. Die vorherrschenden Konversationszustände bei einem Mixed-Outcome sind daher *Closing* und *Canceling*. Mitglieder der Complete-Menge bekommen eine `Close`-Nachricht, während die Mitglieder der Cancel-Menge immer noch im *Canceling*-Zustand sein können. Wenn die letzte noch fehlende Bestätigung eines Teilnehmers via `Canceled`-Nachricht oder

`Closed`-Nachricht den Koordinator erreicht, dann wird der globale Zustand ENDED betreten. Im Unterschied dazu dürfen bei einem Atomic-Outcome natürlich keine Teilnehmer im Canceling-Zustand sein. Da alle Teilnehmer im Konversationszustand *Completed* sind, bevor der Zustand FINISHING betreten wird, gibt es auch keine weiteren vorherrschenden Konversationszustände als Closing in diesem Zustand. Wenn der letzte Teilnehmer im atomaren Fall seine `Closed`-Nachricht gesendet hat, geht der Koordinator in den Endzustand.

Im Falle eines Mixed-Outcome (siehe Abbildung 5.5) können im PREPARING- oder FINISHING-Zustand Teilnehmer die Transaktion noch mithilfe von `Exit`-Nachrichten verlassen oder ihre Unfähigkeit, die lokale Arbeit erfolgreich zu beenden, mithilfe einer `CannotComplete`-Nachricht zum Ausdruck bringen. Im Falle eines Atomic-Outcomes ist dies natürlich nur im PREPARING-Zustand möglich. Das Verlassen von Teilnehmern ist allerdings in WS-BusinessActivitities nicht reglementiert, was leicht in abgebrochenen Transaktionen enden kann. Mitglieder der Cancel-Menge können den Prozess dabei ohne weitere Auswirkung verlassen; ihr Ausstieg wird durch den Koordinator entsprechend behandelt. Wenn allerdings Mitglieder aus der Complete-Menge den Prozess verlassen, dann kann die Transaktion nicht mehr erfolgreich beendet werden und der Zustand ABORTING wird erreicht. Um das Verlassen von Teilnehmern aus einer Transaktion in diesem Fall robuster zu machen, kann beispielsweise, wie in [HRR07] vorgeschlagen, ein Protokoll für einen Exit-Handshake benutzt werden oder aber die Vitalität der Teilnehmer, wie in Abschnitt 5.1.2 beschrieben, genutzt werden. Außer durch verlassende Teilnehmer der Complete-Menge kann der Zustand ABORTING auch durch das Erkennen von Zeitschrankenverletzungen betreten werden. Zeitschranken sind in WS-Coordination durch das `Expires`-Element eines transaktionalen Kontextes unterstützt [FJ09]. In diesem Fall werden alle Teilnehmer der Cancel-Menge zugeordnet und der Koordinator sendet entsprechende Nachrichten an die Teilnehmer.

Wenn der ABORTING-Zustand betreten wird, ist es sicher, dass die Transaktion nicht mehr erfolgreich zum Abschluss gebracht werden kann. Daher entspricht dieser Zustand der zweiten Variante der zweiten Phase eines Transaktionsabschlusses. In diesem Zustand sendet der Koordinator die `Cancel`-Nachricht an die Mitglieder der Cancel-Menge und `Compensate`-Nachrichten an schon vorhandene Mitglieder der Complete-Menge. Die vorherrschenden Konversationszustände sind daher *Canceling* und *Compensating*. Genauso wie im PREPARING- und FINISHING-Zustand können Teilnehmer die Transaktion via `Exit`-Nachrichten verlassen oder mit `CannotComplete`-Nachrichten anzeigen, dass sie Ihre Arbeit nicht beenden können. Wird die letzte erwartete `Canceled`-, `Compensated`-, `Exited`- oder `NotCompleted`-Nachricht vom Koordinator empfangen, ändert sich der globale Zustand in ENDED. Der Zustand ABORTING unterscheidet sich bei Mixed- und Atomic-Outcome kaum. Der einzige Unterschied besteht in den Auswahlgruppen und der Möglichkeit, dass Teilnehmer trotzdem noch eine Chose-Nachricht senden können, die dann entsprechend ignoriert werden kann.

Der ENDED-Zustand kann über die Zustände FINISHING und ABORTING erreicht werden, wenn die letzte erwartete Nachricht gesendet bzw. empfangen wurde. In diesem Zu-

stand sind alle Konversationen im Zustand *Ended*; es werden jetzt keine weiteren Nachrichten vom Koordinator oder von Teilnehmern gesendet. Je nachdem, über welche Zustände der ENDED-Zustand erreicht wurde, ist die Transaktion erfolgreich abgelaufen oder abgebrochen worden.

In den beiden Abbildungen 5.4 und 5.5 ist zusätzlich noch der FAULTED-Zustand vorgesehen, der bei fatalen Fehlern von Teilnehmern betreten wird. Die Auswirkung von Fehlern und dafür notwendigen die Maßnahmen werden im folgenden Abschnitt im Detail diskutiert.

5.1.3.2. Fehler bei Teilnehmern und deren Auswirkungen

Die WS-BusinessActivity-Spezifikation erlaubt es den Teilnehmern, aus den Konversationszuständen *Active*, *Completing*, *Canceling* und *Compensating* heraus, mithilfe von `Fail`-Nachrichten Fehler an den Koordinator zu melden (siehe auch Abbildung 5.3). Im Kontrast zu den `CannotComplete`-, `Exit`-, und `Canceled`-Nachrichten zeigt eine `Fail`-Nachricht an, dass der Teilnehmer bei der Ausführung seiner Aktivität, wie beispielsweise dem Kompensieren, nicht erfolgreich gewesen ist und der Zustand seiner Arbeit daher *undefiniert* ist.

Das bisher vorgestellte erweiterte BACC-Protokoll beinhaltet zusätzlich zu der `Close`-Nachricht eine weitere `Fail`-Kante, die es Teilnehmern erlaubt, auch aus dem *Closing*-Zustand heraus Fehler zu melden. Diese Erweiterung basiert auf einem Vorschlag von Furniss and Green [FG04]. Der weitere Zustandsübergang erlaubt verschiedene Implementierungen wie *provisional-final* oder *validate-do* für die Teilnehmer, welche ihre Arbeit nach dem Bestätigen der `Close`-Nachricht vom Koordinator erst durchführen. Ohne diese Erweiterung können nur *do-compensate*-Muster für Teilnehmer implementiert werden, welche ihre Arbeit vor dem Senden der `Completed`-Nachricht verrichten und dann bei Bedarf ihre Arbeit rückgängig machen können[4].

In Bezug auf ein wohldefiniertes Transaktionsverhalten führt allerdings ein fehlerhafter Teilnehmer an dieser Stelle den Prozess in eine Sackgasse, da weder das Kompensieren noch das Festschreiben der bisher geleisteten Arbeit möglich ist. Da dadurch die Atomarität und die Konsistenz der Arbeit der Teilnehmer gefährdet ist, gibt es keinen direkten Weg zu einem erfolgreichen Abarbeiten der Transaktion mehr. `Fail`-Nachrichten sind also konsequenterweise nicht für die normale Transaktionskoordination ausgelegt, sondern definieren *fatale Fehler*, die nicht automatisch gelöst werden können. Eine Reaktion auf einen solchen Fehler kann zum einen das manuelle Ändern von Prozesszielen sein, damit das bestmögliche Ziel noch erreicht werden kann. Als Beispiel kann bei einer Reisebuchung bei nicht erfolgreicher Automietung die Reise trotzdem ohne Mietwagen angetreten werden; ein Teilziel des Prozesses wird dabei gestrichen. Zum anderen kann auch entschieden werden, eine manuelle Recovery durchzuführen, welche die bereits erfolgreich durchgeführten Aktivitäten wieder

[4] Die Implementierungsmuster von transaktionalen Teilnehmern sind auch in Abschnitt 2.3.3.3 für die BTP-Spezifikation diskutiert.

kompensiert. Beides erfolgt allerdings nicht automatisch, sondern erfordert das manuelle Eingreifen in den Prozess.

WS-BusinessActivity spezifiziert weiterhin, dass eine `Fail`-Nachricht vom Koordinator bestätigt werden muss. Der Koordinator soll dann den fehlerhaften Teilnehmer vergessen, genauso wie der Teilnehmer alle Daten über die transaktionale Aktivität vergessen soll [FL09]. Die konkrete Behandlung zum Lösen der Fehler ist den Implementierungen überlassen. Das globale Zustandsmodell liefert aber auch in diesem Fall eine wohldefinierte und erweiterbare Basis zur Behandlung von Teilnehmerfehlern. Wenn eine `Fail`-Nachricht vom Koordinator empfangen wird, dann wird der globale Zustand auf FAULTED gesetzt. Da der Koordinator die Transaktion durch einen fehlerhaften Teilnehmer nicht mehr zu einem geordneten Abschluss bringen kann, kann der entsprechende Prozess nur angehalten oder unter Berücksichtigung des Teilnehmerfehlers manuell fortgesetzt werden.

Eine Transaktion anzuhalten, bedeutet im Fehlerfall, dass alle Teilnehmer in ihrem Konversationszustand verbleiben und eine manuelle Intervention eines Administrators oder Domänenexperten erwarten. Dieses Konzept steht eindeutig im Konflikt eines autonomen und generischen Koordinators, offeriert aber ein stabiles und generisches Vorgehen bei fatalen Fehlern. Es ist die letzte Möglichkeit, wenn keine fallspezifischen Regeln für die Wiederaufnahme der Transaktion spezifiziert werden können. Das Wiederanlaufen einer Transaktion bedeutet das Koordinieren in Richtung eines modifizierten Abschlusses oder Abbruchs unter Einbeziehung des Teilnehmerfehlers. Das exakte Vorgehen ist dabei stark fallabhängig und entspricht typischerweise einem *geordneten Teilausfall* und damit dem Koordinieren der Transaktion in Richtung eines Ergebnisses, welches in Anbetracht eines fatalen Fehlers noch angemessen ist. Eine weitere offensichtliche Strategie ist das Abbrechen und Kompensieren der Arbeit aller anderen Teilnehmer.

Wegen der Natur des FAULTED-Zustands gibt es für diesen Zustand in den Abbildungen 5.4 und 5.5 keine genau spezifizierten vorherrschenden Konversationszustände, da alle in BACC definierten Nachrichten in diesem Zustand auftreten können. In beiden Abbildungen sind daher die Zustandsübergänge im FAULTED-Zustand mit einem * markiert. Wenn eine Transaktion wieder anläuft, dann ist der Nachrichtenaustausch ähnlich zu denen im PREPARING-, FINISHING- oder ABORTING-Zustand, je nachdem, welche Wiederanlaufstrategie verwendet wird. Der FAULTED-Zustand enthält daher auch den *Ended*-Zustand der individuellen Konversationszustände. Wenn eine Transaktion fehlerhaft ist, verbleibt sie also im FAULTED-Zustand bis zum Ende ihres Lebenszyklus, um genau den fatalen Fehler anzuzeigen.

Wie in diesem Abschnitt diskutiert, definieren `Fail`-Nachrichten die letzte Möglichkeit eines Teilnehmers, einen fatalen Fehler anzuzeigen. Problematisch daran ist, dass die WS-BA-Spezifikation den Zustand des Teilnehmers damit als *undefiniert* spezifiziert und damit nicht genau definiert werden kann, wie der Zustand des Teilnehmers ist. So bleibt auch der Zustand der Gesamttransaktion unspezifisch, was durch den *Faulted*-Zustand charakterisiert wird. Nach Möglichkeit sind diese Zustände in den Teilnehmern bei der Implementierung

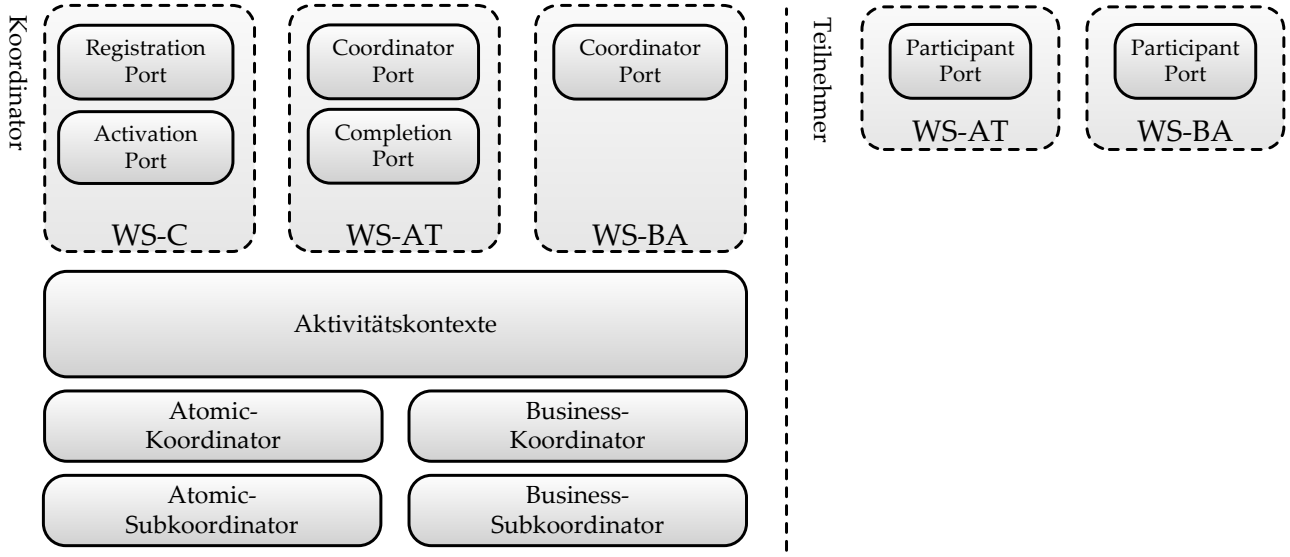


Abbildung 5.6.: Architekturübersicht über das Transaktionsrahmenwerk

unbedingt zu vermeiden, weshalb in dieser Arbeit auch davon ausgegangen wird, dass der Zustand nur in den seltensten Fällen wirklich benötigt wird. Zudem wird deutlich, dass in diesem Ausnahmefall zumeist eine manuelle Recovery notwendig ist, deren Notwendigkeit aber aufgrund des globalen Koordinatorzustandes signalisiert werden kann.

5.1.4. Implementierung eines generischen Transaktionskoordinators

Zur Implementierung eines Koordinators mit den eben diskutierten Erweiterungen sind verschiedene Projekte und Standards im Voraus betrachtet worden, um die Implementierung auf Basis von bereits bestehenden Projekten zu realisieren. Allerdings ist die Implementierung im Rahmen dieser Arbeit letztendlich nicht auf Basis von existierenden Projekten wie Apache Kandula2 (siehe [Apa06]) oder der WS-BusinessActivity-Implementierung von JBoss (siehe [Red06]) erfolgt, da beim Start der Implementierung Ersteres keine Unterstützung für WS-BusinessActivity angeboten hat und Letzteres auf Basis von bereits veralteten Versionen der WS-Coordination-Standards von Microsoft und IBM implementiert wurde. Die Implementierung eines generischen Koordinators ist im Rahmen dieser Arbeit daher komplett neu vorgenommen worden. Zur Implementierung bietet die Standardisierungsgruppe OASIS für die aktuellen Versionen der Standards diverse XML-Schemata, WSDL-Beschreibungen und Spezifikationspapiere an, welche als Basis der Implementierung dienen.

Die Hauptarchitektur des Prototyps ist in Abbildung 5.6 zu sehen. Um die WS-Coordination-Spezifikation zu implementieren, werden mindestens die entsprechenden Web-Services-Port-Types wie der *Registration*- oder *Activation*-Port benötigt. Entsprechendes gilt für die Koordinationstypen für kurzlebige Transaktionen mit WS-AtomicTransaction (*CoordinatorPort*, *CompletionPort*, *ParticipantPort*) und für langlebige Transaktionen mit WS-Busi-

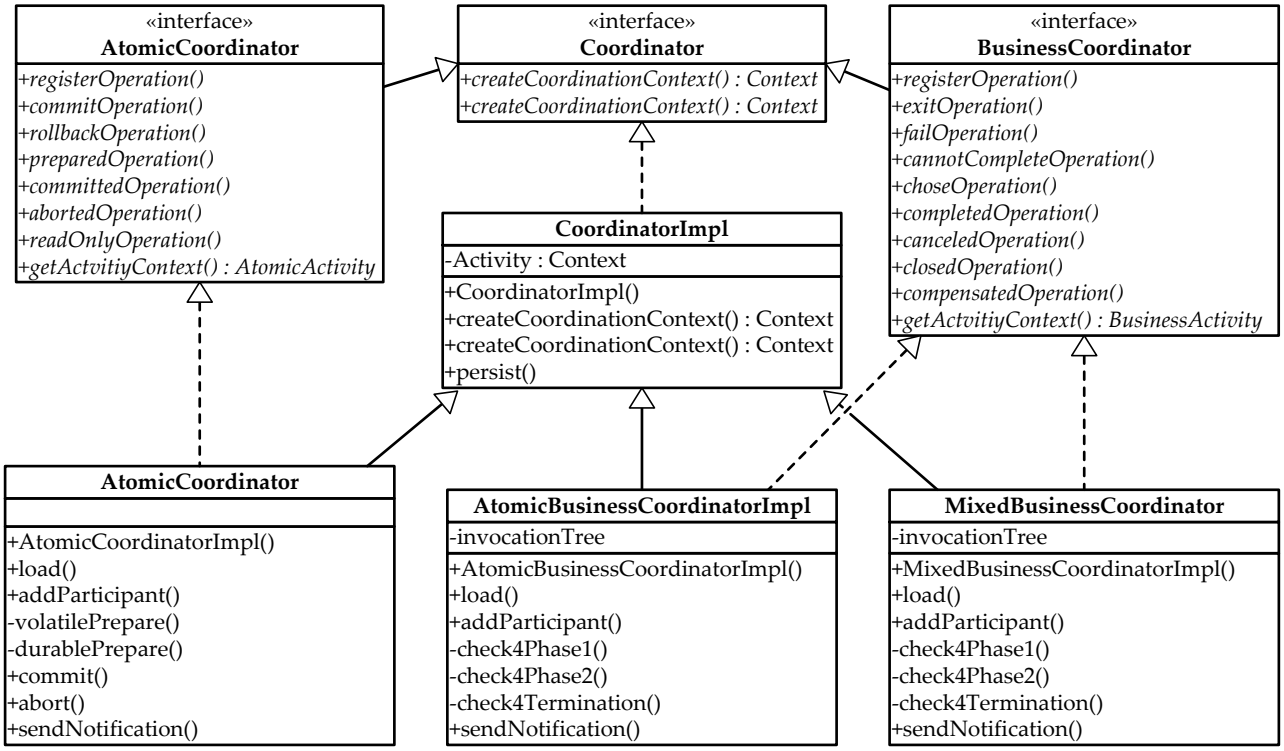


Abbildung 5.7.: Klassenhierarchie für Koordinatoren

nessActivity (*CoordinatorPort*, *ParticipantPort*). Diese Port-Types entsprechen in der Implementierung Web-Service-Schnittstellen, welche mit anderen transaktionsfähigen Teilnehmern kommunizieren. Zur Erstellung dieser Schnittstellen werden die von OASIS bereitgestellten WSDL-Dokumente genutzt, um darüber Schnittstellen mithilfe des Apache-Axis2-Toolkits (siehe [Apa09]) zu generieren. Die dabei generierten Klassenskelette werden dann mithilfe von Java um ihr erwartetes Verhalten erweitert. Ein implementierter Port initialisiert oder lädt dabei einen Koordinator, der zum Kontext einer empfangenen Nachricht passt. Dabei wird ein Atomic-Koordinator bei den WS-AT-Protokollen und ein Business-Koordinator bei den WS-BA-Protokollen genutzt. Der Koordinator nutzt weitere Klassen, welche den Kontext einer Transaktion repräsentieren. Das Rahmenwerk implementiert die komplette Transaktionsinfrastruktur in Bezug auf zwei Aspekte: Auf der einen Seite wird die gesamte Infrastruktur implementiert, um einen Koordinator für WS-AtomicTransaction und WS-BusinessActivity mit den vorgestellten Erweiterungen zu realisieren. Auf der anderen Seite werden im Rahmenwerk auch die Web-Services-Stubs implementiert, um Teilnehmer einer Transaktion anzusprechen. Ein Entwickler wird also auch bei der Entwicklung von Teilnehmern an einer Transaktion unterstützt. Das komplett implementierte Rahmenwerk ist auch auf `http://tracg.sourceforge.net` veröffentlicht und kann genutzt werden, um eigene Web-Services-Transaktionen zu realisieren.

Um die dargestellten Komponenten zu realisieren, müssen Klassen für Koordinatoren, verwaltete Aktivitäten und Web-Services-Schnittstellen implementiert werden. Die Klassenhierarchie für Koordinatoren ist dabei in Abbildung 5.7 abgebildet. Alle Koordinatoren er-

ben von der Basisklasse `CoordinatorImpl`, welche in der Lage ist, Koordinationskontexte zu erzeugen und den Zustand eines Koordinators zu persistieren. Der Koordinationskontext beinhaltet alle Daten, die zur Koordination für WS-AT oder WS-BA notwendig sind, beispielsweise die Teilnehmer und deren Zustände sowie den globalen Koordinatorzustand eines Koordinators. Aus Abbildung 5.7 ist ersichtlich, dass für jede Nachricht im Protokoll eine entsprechende Methode des jeweiligen Koordinators vorgesehen ist. Zusätzlich gibt es für die WS-BA-Koordinatoren der Koordinationstypen Mixed-Outcome und Atomic-Outcome Methoden, welche bei Zufügen von Teilnehmern und bei Empfangen von Antworten der Teilnehmer prüfen können, ob die erste beziehungsweise zweite Phase der Transaktionskoordination eingeleitet werden kann (`check4Phase1()` und `check4Phase2()`). In der ersten Phase wird abgewartet, bis die maximale Ausdehnung des Aufrufbaumes erreicht ist und alle Auswahlgruppen entschieden sind. Danach können entsprechende `Complete`- und `Cancel`-Nachrichten versendet werden. In der zweiten Phase wird so lange gewartet, bis alle notwendigen Teilnehmer für den Transaktionserfolg auf `Complete`-Nachrichten geantwortet haben. Haben alle geantwortet, können der Erfolg des Transaktionsausgangs bestimmt und `Close`- beziehungsweise `Compensate`-Nachrichten versendet werden. Die Methode `check4Termination` prüft nach jeder erhaltenen Nachricht, ob alle Teilnehmer ihre Anfragen bestätigt haben. Wenn ja, ist die Transaktion komplett abgeschlossen und der Koordinator kann den Kontext der Transaktion vergessen.

5.1.4.1. Umsetzung der Regeln

Eine weitere wichtige Umsetzung im Koordinator betrifft die in Abschnitt 5.1.2 definierten Regeln zur Entscheidung der beiden „Wann“- und „Wer“-Fragen. Dafür gibt es zum einen eine Anbindung an den bereits in Abschnitt 4.2.2 beschriebenen Prolog-Interpreter Eclipse-CLP, welche über die Java-Klasse `PrologEngine` realisiert ist. Die Klasse ist in der Lage, neue Fakten zur Wissensbasis in Prolog hinzuzufügen und Anfragen an diese zu stellen. Die Regeln selbst sind in Prolog implementiert (siehe Anhang A.2.1) und entsprechen den im letzten Abschnitt hergeleiteten Regeln zur Koordination.

Zum anderen erfolgt die Repräsentation der Aufrufhierarchie mithilfe der Klasse `InvocationTreeImpl`. Diese Klasse wird von einem Koordinator zur Steuerung genutzt und bietet zusätzlich Methoden an, um weitere Daten über den Aufrufbaum zu erhalten. Beispielsweise bietet die Klasse zur Beantwortung der „Wann“-Frage - wie in Abbildung 5.8 zu sehen - Methoden zur Bestimmung an, ob der Aufrufbaum die maximale Ausdehnung erreicht hat und damit die Demarkation eingeleitet werden kann (`isPhase1()`). Dafür liefert die Klasse auch die Anzahl der noch erwarteten Registrierungen sowie für die spätere zweite Phase der Koordination die Anzahl der erwarteten Antworten. Damit der Aufrufbaum mit Daten gefüttert werden kann, sind entsprechende Methoden wie `addParticipant()`, `exit()`, `cannotComplete()`, `chose()` und auch `completed()` implementiert. Weiterhin kann ein Koordinator zur Bestimmung, welcher Teilnehmer welche Nachrichten bekommt, die

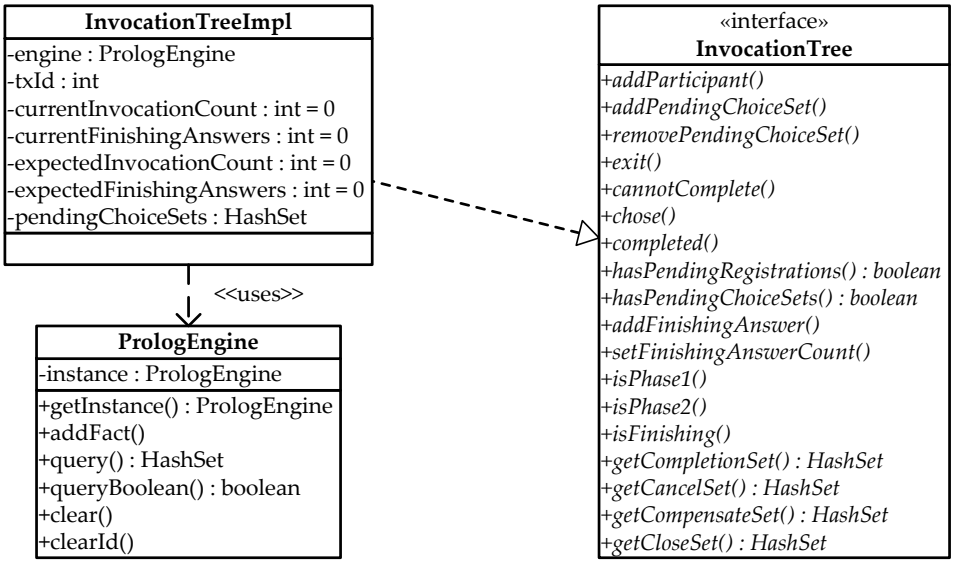


Abbildung 5.8.: Klassendiagramm zur Verwaltung von Aufrufhierarchien

jeweiligen Complete- und Cancel- oder auch die Close- und Compensate-Mengen ausgeben und auch die Entscheidung über den erfolgreichen Ausgang der Transaktion über die Regel `isFinishing()` ermitteln.

Durch das Formalisieren und dem deklarativen Beschreiben des Koordinatorverhaltens über den Aufrufbaum ist es also möglich, einen autonomen und weitestgehend anwendungsunabhängigen Koordinator zu spezifizieren, welcher kein explizites Completion-Protokoll wie bei WS-AtomicTransaction zur Steuerung mehr benötigt.

5.1.4.2. Erweiterungen von WS-Coordination

Bei der Implementierung der Transaktionsspezifikationen sind verschiedene Probleme der Standards zu erkennen, welche die Interoperabilität von verschiedenen WS-Coordination-Spezifikationen reduzieren, da verschiedene Entwurfsentscheidungen bei der Implementierung des Standards möglich sind. Im Folgenden werden jetzt die Protokoll-Erweiterungen für WS-Coordination genauer vorgestellt und dabei die erkannten Interoperabilitätsprobleme diskutiert:

1. **Verteilung des Kontexts:** WS-Coordination definiert nicht explizit, wie ein Transaktionskontext an spätere Teilnehmer einer Transaktion verteilt wird. Die Spezifikation des Standards schlägt bei Aufruf von Teilnehmern, den Kontext im Kopf einer SOAP-Nachricht mitzusenden. In der vorgestellten Implementierung ist dies zwar der Fall, allerdings kann eine andere Implementierung den Kontextweitergabemechanismus anders implementieren und damit nicht mehr interoperabel sein.

2. **Identifikation von Teilnehmern:** WS-Coordination definiert nicht, wie Teilnehmer in einer Transaktion identifiziert werden können. In der vorgestellten Implementierung

generiert der Koordinator während der Registrierungsphase einen Identifikator für den jeweiligen Teilnehmer. Damit einem Teilnehmer ein Identifikator zugeteilt werden kann, muss die `RegisterResponse`-Nachricht erweitert werden, welche den Identifikator des Teilnehmers enthält. Wann immer dieser Teilnehmer dann mit dem Koordinator kommuniziert, muss der Identifikator an die jeweilige Nachricht angehängt werden. Die `RegisterResponse`-Nachricht der Implementierung ist dafür wie in Listing 5.1 zu sehen um ein `YourId`-Element erweitert worden, um dem Teilnehmer in der Antwort nach der Registrierung seinen Identifikator mitzuteilen.

Quellcode 5.1: Zuteilung der Teilnehmer-Id

```
1 <coord:RegisterResponse>
2   ...
3   <vsis:YourId>
4     16103ECC-3AA1-42B2-8218-DF716763E94C
5   </vsis:YourId>
6 </coord:RegisterResponse>
```

Diese Erweiterung der `RegisterResponse`-Nachricht ist zwar nach wie vor standardkonform, da WS-Coordination bestimmte Erweiterungspunkte innerhalb von Nachrichten definiert. Trotzdem kann jede Implementierung von WS-Coordination eine andere Entwurfsentscheidung zur Identifizierung von Teilnehmern treffen und sich damit inkompatibel zur vorliegenden Implementierung machen.

Quellcode 5.2: Koordinationskontext mit Teilnehmer-Id

```
1 <coord:CoordinationContext>
2   ...
3   <vsis:participantId>
4     16103ECC-3AA1-42B2-8218-DF716763E94C
5   </vsis:participantId>
6 </coord:CoordinationContext>
```

Die Identifikatoren der Teilnehmer werden jetzt bei jeglicher Kommunikation mit dem Koordinator verwendet. Dabei wird der eigene Identifikator an den Transaktionskontext gehängt, wie es auch in Listing 5.2 zu sehen ist. Damit weiß der aufgerufene Koordinator, in welchem Kontext dieser Aufruf erfolgt ist und kann die Kontextdaten wie beispielsweise den Interaktionszustand eines Teilnehmers aktualisieren.

3. **Demarkation:** Die WS-BusinessActivity-Spezifikation definiert nicht, wie die Demarkation einer Transaktion eingeleitet werden soll. Speziell für dieses Problem ist im letzten Abschnitt ein generischer Koordinator vorgeschlagen worden, welcher eigenständig mithilfe von Regeln über einen Aufrufbaum einer Transaktion über die Demarkation entscheidet. Dafür benötigt der Koordinator allerdings verschiedene Informationen

der Teilnehmer zur Zeit der Registrierung und zur Laufzeit, wenn die Auswahlgruppen entschieden werden:

- **Den Identifikator des Aufrufers:** Wenn ein Teilnehmer von einem anderen Teilnehmer mithilfe eines Transaktionskontextes aufgerufen wird, enthält der Kontext den Identifikator des Aufrufers. Dieser Aufrufer-Identifikator wird bei der Registrierung des Teilnehmers dem Koordinator mitgeteilt, wie in Listing 5.3 mithilfe des `CallerId`-Elements dargestellt. Der Koordinator kann mithilfe der Angabe des Vaterknotens die Aufrufhierarchie intern aufbauen.

Quellcode 5.3: Identifikator des Aufrufers

```
<coord:CoordinationContext>
  ...
  <vsis:CallerId>
    2BAE044F-E034-4B25-B4DA-4ADC4E3C2CF1
  </vsis:CallerId>
</coord:CoordinationContext>
```

- **Die Anzahl der Sub-Teilnehmer:** Wenn sich ein Teilnehmer bei einem Koordinator registriert, teilt er dem Koordinator zusätzlich mit, wie viele Teilnehmer er zur Erbringung seiner eigenen Arbeit noch in die Transaktion einbeziehen muss. Dafür wird im Koordinationskontext die entsprechende Anzahl der Sub-Teilnehmer wie in Listing 5.4 dargestellt bei der Registrierung an den Koordinator übergeben.

Quellcode 5.4: Anzahl der Sub-Teilnehmer

```
<coord:CoordinationContext>
  ...
  <vsis:subsequentCalls>
    2
  </vsis:subsequentCalls>
</coord:CoordinationContext>
```

- **Teilnehmer einer Auswahl:** Wenn ein Web-Service über einen transaktionalen Kontext aufgerufen wird, dann teilt der Aufrufer dem aufgerufenen Dienst mit, ob er Teil einer Auswahlgruppe ist. Die Auswahlgruppe wird dann bei der Registrierung des Teilnehmers benannt. Das Resultat der Auswahl wird, wie in Abschnitt 5.1 beschrieben, mithilfe einer `Chose`-Nachricht an den Koordinator übermittelt. Die Nachricht wird wie in Listing 5.5 formatiert und teilt dem Koordinator die Menge der ausgewählten Teilnehmer mit. Teilnehmer, die nicht in dieser Menge enthalten sind, sind entsprechend nicht ausgewählt und können später zurückgesetzt werden, ohne den Erfolg der Gesamttransaktion zu beeinflussen.

Quellcode 5.5: Ausgewählte Teilnehmer einer Auswahlgruppe

```
<soapenv:Body>
  <coord:Chose ChoiceSet="516D8452...">
    <coord:ParticipantId>
      8A895863-CE72-4F2C-BBE8-96BD215801AA
    </coord:ParticipantId>
  </coord:Chose>
</soapenv:Body>
```

Mit Unterstützung dieser Erweiterungen von WS-Coordination und seinen Protokollen ist die Implementierung eines autonomen Koordinators möglich. Problematisch an einigen Entwurfsentscheidungen ist allerdings, dass die Implementierung trotz großer Standardkonformität (bis auf die Chose-Nachricht) verschiedene Interoperabilitätsprobleme mit anderen WS-Coordination-Implementierungen erwarten lässt. Dies liegt darin begründet, dass durch die Spezifikation von WS-Coordination selbst, wie eben diskutiert, diverse Interoperabilitätsprobleme aufgrund verschiedener aber immer noch standardkonformer Entwurfsentscheidungen entstehen. An dieser Stelle sollte der Standard nachgebessert werden, um solche Unsicherheiten zu vermeiden.

5.1.4.3. Evaluation der Implementierung

Auf Basis der im letzten Abschnitt vorgestellten Implementierung werden jetzt die realisierten Koordinatoren in Bezug auf Skalierbarkeit evaluiert. Dies betrifft den Koordinator für WS-AtomicTransaction und die Koordinatoren für die verschiedenen WS-BusinessActivities-Protokolle und Koordinationstypen wie Mixed- und Atomic-Outcome.

Als Testszenarien für die jeweiligen Protokolle werden flache Transaktionshierarchien mit einem zentralen Koordinator und einer wachsenden Anzahl von Teilnehmern betrachtet. Für WS-AT ist das Szenario einer einfachen Umbuchung über verschiedene Banken implementiert, bei der ein Betrag von der einen Bank abgebucht und der bei der anderen Bank wieder eingezahlt wird. Die Teilnehmer der Transaktion sind dann realisierte Bankdienste. Für WS-BA ist ein klassisches Flugbuchungsbeispiel implementiert, bei der je nach Koordinationstyp (Atomic oder Mixed) entweder nur ein Sitz oder alle Sitze gebucht werden sollen. Dabei wird pro Airline-Dienst jeweils nur ein Sitz reserviert. Die Implementierungen der jeweiligen Airline-Dienste gibt es in zwei Varianten für die Protokolle BAPC und BACC, bei denen entweder der Koordinator den Diensten mitteilt, dass sie ihre Arbeit abschließen sollen oder die Teilnehmer dies selbst dem Koordinator melden.

Als Testumgebung wird ein Intel i5-750 mit 4 Kernen und 8GB RAM mit installiertem Apache Tomcat und Axis2 eingesetzt. Gemessen wird jeweils die Zeit vom Start einer Transaktion und dem jeweiligen Ende, wenn der Initiator das Ende je nach Protokoll der Transaktion mitgeteilt bekommt. Der Koordinator und die teilnehmenden Dienste an den Transaktionen sowie der Initiator einer Transaktion werden auf demselben Testrechner gestartet.

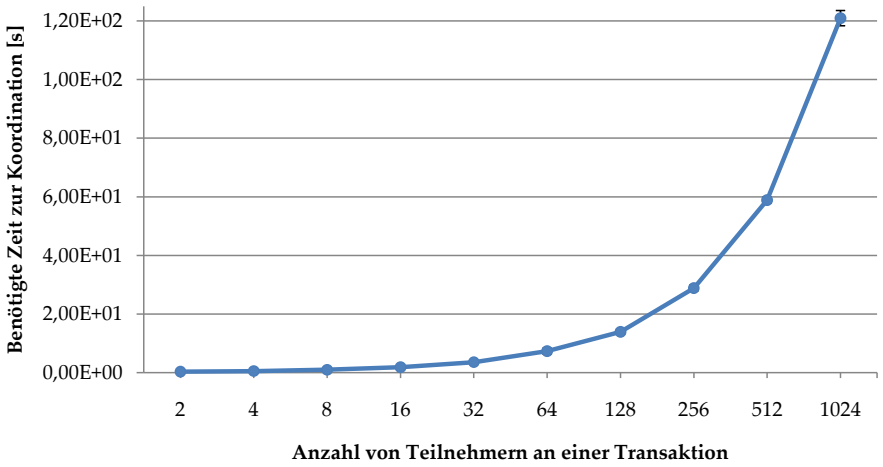


Abbildung 5.9.: Implementierung von WS-AT, 2PC-Verhalten

Szenario 1 - Umbuchungen (WS-AT): Abbildung 5.9 zeigt, dass die Implementierung von WS-AtomicTransaction mit WS-Coordination auf Basis von Apache Axis2 fast linear mit steigender Anzahl von Teilnehmern innerhalb einer Transaktion skaliert. Trotz einer Zahl von 1024 Teilnehmern in einer Transaktion benötigt der Testrechner knapp zwei Minuten zum Abarbeiten des 2PC-Protokolls. Dabei beinhaltet die gemessene Zeit die benötige Arbeitszeit, welche die Teilnehmer lokal zum Abarbeiten ihrer eigenen Arbeit benötigen. Die Beispieltransaktion ist eine einfache flache Transaktion, die aus einem Initiator besteht, welche Teilnehmer (dargestellt auf der x-Achse) aufruft. Das lineare Verhalten ist recht gut zu erklären, da zur Bestimmung des aktuellen Zustandes im Koordinator jeweils nur die Teilnehmer iteriert werden müssen. Genauso ist die Anzahl der versendeten Nachrichten von der Anzahl der Teilnehmer direkt abhängig, sodass bei wachsender Anzahl der Teilnehmer auch das entsprechende Verhalten zu erwarten ist. WS-AT ist nur zum Vergleich für die anderen Protokolle ohne Rule-Engine implementiert und das Resultat ist die Referenz für die anderen implementierten Koordinatoren. Gemessen wird in diesem Szenario die Zeit vom Erzeugen des Transaktionskontextes bis zum Empfang der Nachricht durch den Initiator, dass die Transaktion erfolgreich abgeschlossen worden ist.

Szenario 2 - Atomare Flugbuchungen (WS-BA): Im zweiten Evaluationsszenario werden für die beiden Koordinationsprotokolle BAPC und BACC verschiedene Flugreservierungen getätigt, die zum Erfolg der Transaktion alle erfolgreich bestätigt werden müssen. Der Initiator der Transaktion ist auch Teilnehmer der Transaktion und teilt über die `createCoordinationContext`-Nachricht dem Koordinator mit, wie viele Teilnehmer er in die Transaktion einbinden wird. Der Koordinator kann über die Anzahl der noch einzubindenden Teilnehmer die maximale Ausdehnung des Aufrufbaumes berechnen und im Falle von BACC selbstständig über das Auslösen der Demarkation entscheiden. Im Falle von BAPC wartet der Koordinator auf die `Complete`-Nachrichten der Teilnehmer. Gemessen wird die Zeit,

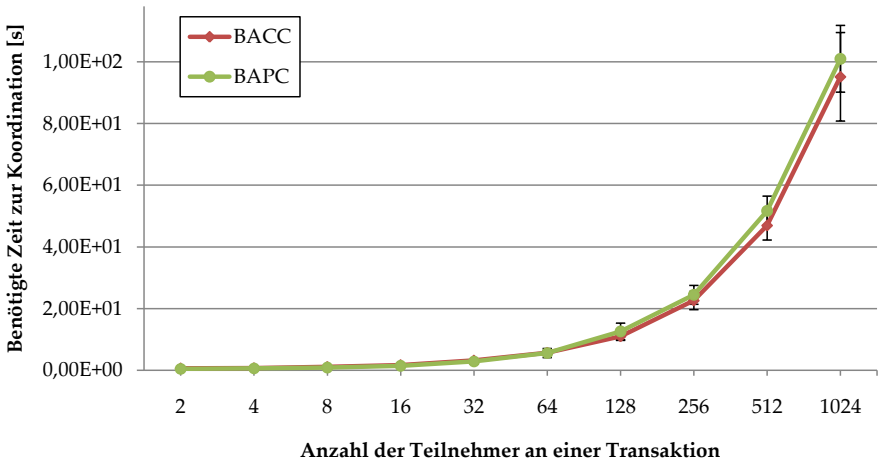


Abbildung 5.10.: WS-BA, Implementierung von BACC (Raute) und BAPC (Kreis), Atomic-Outcome

die vom Erzeugen des Kontextes bis zum Erhalt der `Close`-Nachricht durch den Initiator vergeht. In Abbildung 5.10 ist das Zeitverhalten für beide Protokolle aufgezeichnet. Da prinzipiell die gleichen Koordinationsentscheidungen getroffen werden müssen, ist im atomaren Fall das Zeitverhalten der beiden Kurven gleich. Allerdings kann das Einleiten der ersten Phase der Transaktionskontrolle bei BACC früher geschehen, wenn alle Teilnehmer registriert sind. Bei BAPC muss gewartet werden, bis sich alle Teilnehmer zusätzlich auch mit einer `Complete`-Nachricht zurückgemeldet haben. Auch in diesem Szenario verhält sich die Implementierung nahezu linear, da beim Atomic-Outcome das Einordnen in die Mengen über den Aufrufbaum nur über die einfachen Regeln ohne Auswahlmengen erfolgen. Zwar wird bei der Entscheidung, die Teilnehmer entweder zu kompensieren oder abzuschließen, der Aufrufbaum durch die Regeln komplett traversiert, allerdings hat dies durch die flache Hierarchie keinen großen Einfluss und entspricht einem Iterieren über alle Teilnehmer.

Szenario 3 - Gemischte Flugreservierungen (WS-BA): Das dritte Szenario entspricht dem Zweiten, nur mit dem Unterschied, dass der Initiator eine Auswahl startet, und darüber entscheidet, welche der reservierten Plätze einer Airline gebucht werden sollen und welche Reservierungen wieder zurückgezogen werden. Auch in diesem Fall teilt der Initiator der Auswahlgruppe über die Registrierung dem Koordinator mit, wie viele Teilnehmer er in die Transaktion einbinden wird. Zusätzlich entscheidet er bei Erhalt von allen Resultaten der Airline-Dienste, welche Reservierungen er festschreiben will und teilt dies dem Koordinator über die `Chose`-Nachricht durch die die mitgesendeten Teilnehmer-Identifikatoren der festzuschreibenden Dienste mit. In Abbildung 5.11 ist das Zeitverhalten des Koordinators entsprechend in Abhängigkeit von den teilnehmenden Diensten aufgezeichnet. Hier ist das Zeitverhalten von BAPC etwas schlechter als bei BACC, was darin begründet liegt, dass bestimmte Teilnehmer bereits in der Auswahl-Phase mithilfe von `Cancel`-Nachrichten

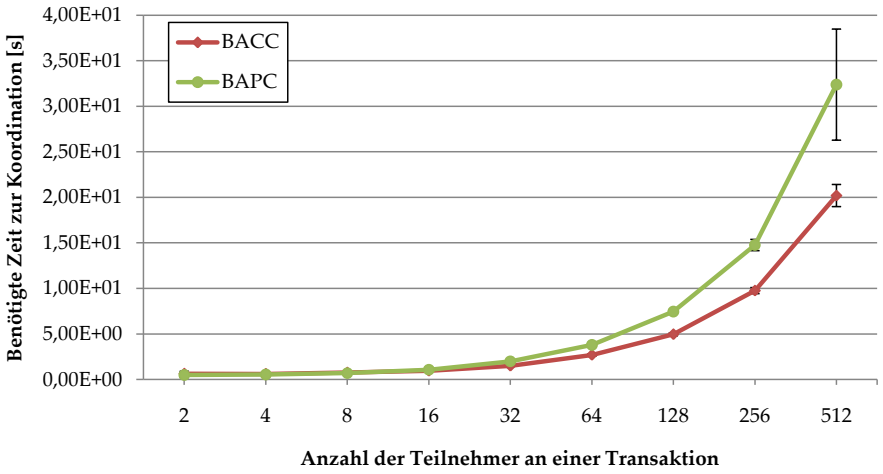


Abbildung 5.11.: WS-BA, Implementierung von BACC (Raute) und BAPC (Kreis), Mixed-Outcome

verdrängt werden und damit nicht weiter betrachtet werden müssen. Bei BAPC wartet der Koordinator entsprechend, bis alle Teilnehmer sich mit einer `Completed`-Nachricht gemeldet haben, und muss dann eine größere Anzahl an Teilnehmern für die Close- und Compensate-Regeln berücksichtigen. Damit ist auch zu sehen, dass das Laufzeitverhalten von BACC im gemischten Fall etwas besser als beim atomaren Fall ist, da auch hier durch die Auswahl verschiedene Teilnehmer nicht mehr bei den Close- und Compensate-Regeln betrachtet werden müssen. Auch hier verhält sich der Zeitverlauf im Testbereich so gut wie linear, was sich durch die letzte Evaluation auch gut zeigen lässt.

Szenario 4 - WS-BA, nur Prolog-Engine: Im letzten Szenario werden nur die Regeln ohne die Implementierung der Koordinatoren und das Versenden der Nachrichten getestet. Dabei werden im atomaren sowie im gemischten Szenario der Flugbuchung nur die jeweiligen Aufrufe an der Klasse `InvocationTreeImpl` simuliert, welche in entsprechende Prolog-Aufrufe resultieren. Die Nachrichtenlaufzeiten fallen damit komplett weg und zeigen die Skalierbarkeit der Implementierung auf Basis von Eclipse-CLP. Genauso fällt die Unterscheidung in BAPC und BACC weg, da sich die beiden Protokolle nur in der Bestimmung der Zeitpunkte für die Demarkation leicht unterscheiden. Die Wissensbasis wird daher bei beiden Protokollen auf die gleiche Art und Weise angefragt. In Abbildung 5.12 ist das Zeitverhalten der Klasse `InvocationTreeImpl` entsprechend in Abhängigkeit von den teilnehmenden Diensten aufgezeichnet.

Dabei ist zu sehen, dass selbst bei 1024 Teilnehmern an einer Transaktion die Prolog-Implementierung im Zeitbereich von einer halben Sekunde liegt, um Teilnehmer in die jeweiligen Mengen einzuteilen und zu entscheiden, ob die Transaktion erfolgreich ist oder nicht. Im atomaren Fall geht dies sogar schneller als im gemischten Fall, da hier die Regeln um Auswahlgruppen beginnend mit *partOfChoice* schnell invalidiert werden können. Im

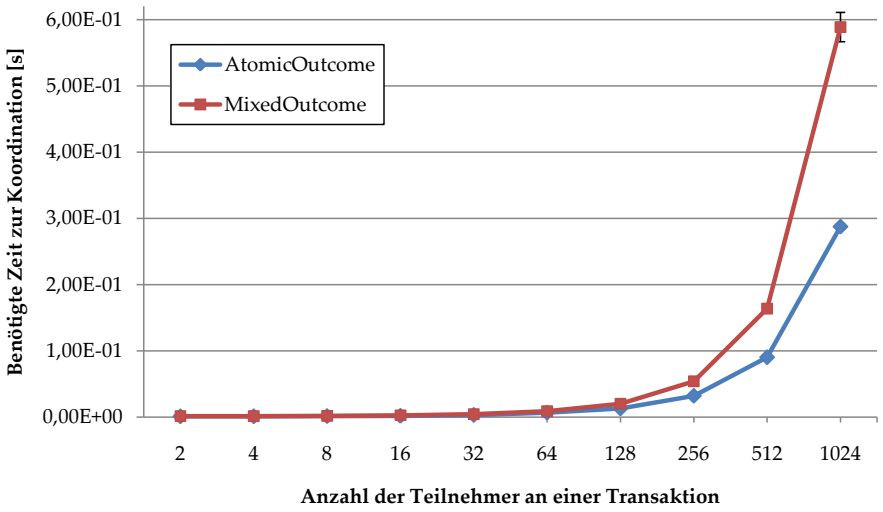


Abbildung 5.12.: Implementierung der Regeln, Atomic-Outcome (Raute) und Mixed-Outcome (Quadrat)

gemischten Fall müssen also deutlich mehr Regeln verarbeitet werden als im atomaren Fall, was sich mit steigender Teilnehmeranzahl auch in dem Leistungsverhalten niederschlägt. Da die benötigte Zeit für Koordinationsentscheidungen im Szenario gegenüber der Nachrichtenlaufzeit gering ist, wirkt sich nur die Übertragungszeit der versendeten Nachrichten auf das Zeitverhalten des Koordinators aus. Da im Protokoll die Anzahl der Nachrichten mit den Teilnehmern linear steigt, steigt auch die benötigte Zeit, bei der Teilnehmer auf Entscheidungen des Koordinators warten müssen, linear an. Die maximal getestete Anzahl von 1024 Teilnehmern an einer Transaktion wird in realen Szenarien auch kaum erreicht werden; daher ist das Laufzeitverhalten der Implementierung mehr als ausreichend. Im direkten Vergleich zwischen WS-AT ohne und WS-BA mit Prolog-Implementierung ist auch zu sehen, dass der regelbasierte Ansatz mit der Verwaltung von Teilnehmerlisten in Prolog schneller als eine Verwaltung von Teilnehmerlisten in Java ist.

5.2. Integration der Fehlererkennung in die Ablaufkontrolle

Im letzten Abschnitt ist ein Ansatz zur autonomen Transaktionskontrolle vorgestellt worden, der es ermöglicht, Szenarien mithilfe von WS-BusinessActivities zu koordinieren, bei denen der Initiator einer Transaktion nicht derjenige Teilnehmer ist, welcher über den Ausgang einer Transaktion entscheidet. Damit werden auch typische Choreographie-Szenarien unterstützt, die im organisationsübergreifenden Umfeld anzutreffen sind. Zusätzlich ist in Abschnitt 4 ein Verifikationsdienst vorgestellt worden, welcher Fehler bei Integritätsbedingungen ermitteln kann. Werden Fehler direkt zur Laufzeit erkannt, können Teilnehmer an

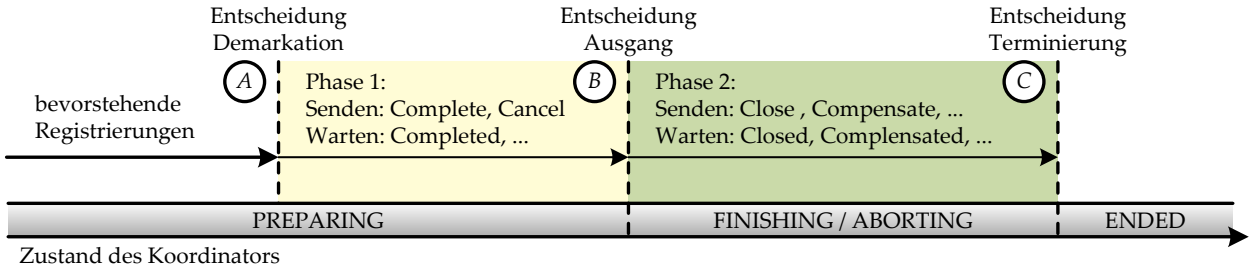


Abbildung 5.13.: Mögliche Anzeigepunkte A,B und C von Fehlern durch die Verifikation

einer Transaktion versuchen, Fehler bereits zur Laufzeit zu beheben. Dazu wird in diesem Abschnitt der Arbeit die Fehlererkennung in die Ablaufkontrolle integriert, damit für einen erfolgreichen Ausgang einer Transaktion nicht nur die Teilnehmer selbst definieren, dass sie ihre Arbeit erfolgreich abgeschlossen haben, sondern dies zusätzlich durch die Laufzeitverifikation auch belegt werden kann.

Eine wichtige Frage bei der Integration der Fehlererkennung in die transaktionale Koordination ist der Zeitpunkt, wann Fehler durch die Verifikation gemeldet werden können. Abbildung 5.13 zeigt die möglichen Zeitpunkte von Fehlermeldungen der Verifikation in Abhängigkeit von Zeitpunkten der Koordinatorentscheidungen und den jeweiligen Zuständen eines Koordinators. Zum Zeitpunkt *C* ist ein Koordinator beispielsweise bereits im Zustand FINISHING oder ABORTING. In dieser Phase haben für den Zustand FINISHING bereits alle notwendigen Teilnehmer eine `Completed`-Nachricht gesendet und der Koordinator hat vor Eintritt in diese Phase die Entscheidung auf Erfolg oder Misserfolg der Transaktion getroffen. Allerdings muss der Koordinator vor Eintritt in den Zustand FINISHING noch den Verifikationsdienst abwarten, weil ihn sonst Fehlermeldungen aus der Phase *B* bei Eintritt in Phase *C* erreichen können. Tut er dies, kann es keine Fehlermeldungen ab diesem Zeitpunkt mehr vom Verifikationsdienst geben, da alle Teilnehmer ihre Arbeit abgeschlossen haben und damit keine weiteren Interaktionen unter den Teilnehmern zu erwarten sind. Zum Zeitpunkt *B* sind zwar alle Teilnehmer registriert, allerdings kann selbst für einen gemeldeten Fehler durch den Verifikationsdienst noch keine Recovery eingeleitet werden, da die Entscheidung über den Erfolg oder Misserfolg auf Basis der Vitalität erst getroffen werden kann, wenn sich alle Teilnehmer zurückgemeldet haben. Das gleiche gilt auch für die Phase *A*, in der bis zum Einleiten der Demarkation gewartet wird. In beiden Phasen *A* und *B* muss ein gemeldeter fehlerhafter Teilnehmer annotiert werden, damit der Fehler bei den Koordinatorentscheidungen entsprechend berücksichtigt werden kann.

Diese Art der Einbeziehung von Fehlern durch die Verifikation geht allerdings mit der Annahme einher, dass das Transaktionsprotokoll selbst von der Verifikation ausgeschlossen ist und sich die Teilnehmer entsprechend den Transaktionsprotokollen richtig verhalten. Tun sie dies nicht, können durch den Verifikationsdienst auch Fehler in Phase *C* gemeldet werden, die dann allerdings nicht mehr automatisch behoben werden, da die Vitalitätsatomarität nicht mehr eingehalten werden kann. Beispielsweise darf ein Teilnehmer auf eine

`Close`-Nachricht die Transaktion nicht mit einer `Exit`-Nachricht verlassen, da sonst die bisher abgegebenen Garantien nicht eingehalten werden können. Passiert dies dennoch, muss eine manuelle Forward-Recovery durchgeführt werden. Eine weitere wichtige Annahme bei der automatischen Fehlerbehebung ist die Tatsache, dass Teilnehmer selbst am besten wissen, wie sie bisher geleistete Arbeit entweder abbrechen oder rückgängig machen können. Dafür sieht die WS-BA-Protokolle beispielsweise Nachrichten wie `Compensate` oder `Cancel` vor, auf die ein Teilnehmer entsprechend reagieren muss. Ein Teilnehmer muss dafür die Implementierungsmuster Do-Compensate, Validate-Do oder Provisional-Final implementieren, damit er Arbeit rückgängig machen kann[5]. Auch bei verletzten Integritätsbedingungen im Prozess muss also ein Teilnehmer in der Lage sein, seine bisher getätigte Arbeit zurückzusetzen. Kann er dies nicht, so muss er eine `Fail`-Nachricht senden.

Weiterhin gilt für die Integration der Fehlererkennung in die Ablaufkontrolle, dass sich nach Möglichkeit alle Teilnehmer einer Choreographie zur Koordination im Rahmen einer Transaktion registrieren müssen. Ansonsten kann die Verifikation möglicherweise Fehler zu Teilnehmern melden, die nicht Teil einer Transaktion sind. Semantisch bedeutet dies, dass solche Fehler sich auf nicht vitale Teilnehmer beziehen und ignoriert werden können. Umgekehrt bedeutet dies aber auch, dass sich vitale Teilnehmer an einer Transaktion zwangsläufig registrieren müssen, da sonst die Integration der Verifikation in die Ablaufkontrolle der Transaktionen keinen Sinn macht.

Um die Verifikation in die Ablaufkontrolle für Transaktionen zu integrieren, gibt es zwei Varianten. Zum einen kann der Verifikationsdienst selbst als Teilnehmer in eine Transaktion eingebunden werden. Eine solche Transaktion kann natürlich nur dann abschließen, wenn die Verifikation erfolgreich abgelaufen ist. Dies kann beispielsweise in der `PREPARING`-Phase durch eine `Completed`-Nachricht durch den Verifikationsdienst angezeigt werden. Der Vorteil dieser Lösung ist, dass beide Bestandteile der Transaktionskontrolle und der Verifikation eigene Bestandteile bleiben und auch unabhängig voneinander betrieben werden können. Der Nachteil der Lösung ist, dass der Verifikationsdienst keine Informationen über die Vitalität der Teilnehmer besitzt und somit die Transaktion durch einen nicht vitalen Teilnehmer abgebrochen werden könnte. Damit diese Lösung realisiert werden kann, benötigt der Verifikationsdienst als Teilnehmer also das Wissen über die Vitalität der anderen Teilnehmer. Zum anderen kann die Verifikation im Rahmen einer zweiten Lösung als Teil innerhalb des Koordinators betrieben werden, in dem bestimmte Regeln erweitert werden, welche die Ergebnisse der Verifikation direkt in die Ablaufkontrolle einbeziehen. Damit ist ein Problem der ersten Variante gelöst, denn der Koordinator kann das Resultat der Verifikation direkt mit in den Aufrufbaum einfließen lassen. Zwar müssen dafür die Regeln erweitert werden, damit der Koordinator die Verifikation einbeziehen kann, aber der Verifikationsdienst kann trotzdem unabhängig von der Transaktionskontrolle seinen Dienst verrichten.

Eine weitere wichtige zu lösende Frage zur Integration der Verifikation in die Ablaufkon-

[5]Siehe dafür Kapitel 2.3.3.3.

trolle ist die Korrelation der Teilnehmer-Identifikatoren zu den Rollen der Choreographie-Teilnehmer. Ohne die Korrelation kann ein Koordinator nicht entscheiden, wie er mit einem gemeldeten Fehler umgehen soll. Genauso kann umgekehrt der Verifikationsdienst fehlerhafte Teilnehmer nicht mit den Transaktionsteilnehmern in Beziehung setzen. Die beiden Ansätze zur Integration werden im folgenden Abschnitt weiter im Detail erläutert und jeweils durch eine Implementierung realisiert.

5.2.1. Integration durch Einbettung in den Koordinator

In diesem Abschnitt wird die Variante der Einbettung der Verifikation in den Regelsatz des Koordinators vorgestellt. Damit dies realisiert werden kann, müssen zum einen die Choreographie-Instanzen und die jeweiligen transaktionalen Aktivitäten in Beziehung gesetzt werden, um eine Korrelation zwischen beiden Welten zu ermöglichen. Weiterhin muss die Regelbasis der Ablaufkontrolle so erweitert werden, dass Fehler von Teilnehmern in entsprechende Maßnahmen der Transaktionskontrolle resultieren.

5.2.1.1. Korrelation zwischen Choreographie-Instanz und Transaktion

Um die gestartete Transaktion und die sich später registrierenden Teilnehmer an der Transaktion mit der jeweiligen Choreographie in Beziehung zu setzen, müssen als Erstes die Daten identifiziert werden, welche für diese Korrelation notwendig sind. Für die Choreographie lassen sich dabei folgende Daten identifizieren:

- **Rollen:** Choreographie-Beschreibungssprachen wie WS-CDL definieren für jede Interaktion, welche Rollen (Sender und Empfänger) der Teilnehmer an einer einzelnen Interaktion beteiligt sind. In der formalen Beschreibung sind dafür jeweils die Sender und Empfängerrollen in einem *exchange*-Prädikat definiert. Im Gegenzug wird registrierten Transaktionsteilnehmern ein Teilnehmer-Identifikator für ihre Teilnahme an der Transaktion zugewiesen. An dieser Stelle gibt es allerdings zwei Fälle in Bezug auf die Rollen zu unterscheiden:
  - **Mehrmalige Registrierung:** Eine Rolle mit mehreren Aktivitäten kann sich dabei für jede dieser Aktivitäten einzeln beim Koordinator zur Koordination anmelden. Problematisch ist an dieser Stelle, dass bei einem durch die Verifikation erkannten Fehler nicht mehr genau bestimmt werden kann, welcher Transaktionsteilnehmer zur Rolle genau zurückgesetzt werden soll. In diesem Fall können nur noch alle zu einer Rolle gehörenden Teilnehmer zurückgesetzt werden, um den Fehler zu korrigieren.
  - **Einmalige Registrierung:** Wenn sich jede Rolle genau einmal zur Koordination anmeldet, besteht die 1:n-Beziehung zwischen Teilnehmern und Rollen nicht mehr und ein Fehler einer lässt sich über eine 1:1-Beziehung direkt auf einen Transaktionsteilnehmer abbilden.

Damit die Abbildung der Rolle auf Transaktionsteilnehmer gelingt, muss bei der Registrierung eines Teilnehmer für eine Transaktion die jeweilige Rolle innerhalb der Choreographie mitgeteilt werden. Dabei wird bei der `Register`-Nachricht die eigene Rolle, wie in Listing 5.6 zu sehen, mitgesendet.

Quellcode 5.6: Korrelationsdaten für Rollen

```
<coord:Register>
  ...
  <vsis:chorRole>
    Seller
  </vsis:chorRole>
</coord:Register>
```

- **Choreographie-Instanzen:** Eine Choreographie-Instanzkennung ist für die Verifikation, wie in Abschnitt 4.1.2 beschrieben, notwendig. Diese Instanzkennung kann dem Koordinator beim Erzeugen eines Kontextes mitgeteilt werden, damit er weiß, dass er als Teil einer zu verifizierenden Choreographie läuft und die entsprechenden Regelsätze laden kann. Dabei kann die Instanzkennung zum einen direkt als Transaktions-Identifikator übernommen werden. Allerdings hat dieses Vorgehen Nachteile bei der Interposition von Koordinatoren, da Sub-Koordinatoren in diesem Falle die gleichen Transaktions-Identifikatoren nutzen. Zum anderen kann der Koordinator sich die Zuordnung zu einer Choreographie-Instanz merken und eigene Transaktions-Identifikatoren vergeben. In der Implementierung ist dieses Mapping von Transaktions-Identifikatoren auf Choreographie-Instanzkennungen mithilfe von Listen realisiert, die der Koordinator verwaltet.

Quellcode 5.7: Korrelationsdaten für Instanzkennungen

```
<coord:CoordinationContext>
  ...
  <vsis:chorVerificationService>
    <addressing:Address>
        http://localhost:8080/axis2/services/VerificationService
    </addressing:Address>
    <addressing:ReferenceProperties>
      <vsis:chorId>
        8A895863-CE72-4F2C-BBE8-96BD215801AA
      </vsis:chorId>
      <vsis:chorType>
        BuyerSellerCreditCheckerV1.1
      </vsis:chorType>
```

```
14      </addressing:ReferenceProperties>
15    </vsis:chorVerificationService>
16  </coord:CoordinationContext>
```

Um die Choreographie-Instanzkennung zu übergeben, muss eine `CreateCoordinationContext`-Nachricht um ein weiteres Element erweitert werden, wie in Listing 5.7 zu sehen. Weiterhin muss der Koordinator auch den Verifikationsdienst kennen, damit er die Verifikationsdaten in die Koordination der Transaktion einbeziehen kann. Dafür muss, wie in Listing 5.7 zu sehen, zusätzlich noch die Endpunktreferenz (EPR) des Verifikationsdienstes mit übergeben werden. Optional kann der Choreographie-Typ mit angegeben werden, falls der Verifikationsdienst für mehr als eine Choreographie-Art zuständig ist. Insgesamt werden also beim Erzeugen eines Kontextes zusätzlich die Choreographieinstanz, der Typ der Choreographie und die Adresse des Verifikationsdienstes angegeben, um die Fehlerdaten in die Ablaufkontrolle mit einbeziehen zu können.

Werden diese Daten benutzt, kann der Koordinator den Verifikationsdienst in die Ablaufkontrolle einbeziehen und Fehler eines Teilnehmer im Aufrufbaum vermerken. Wie genau dies passiert, wird im nächsten Abschnitt mit der Erweiterung der Regelbasis für den Aufrufbaum vorgestellt.

5.2.1.2. Erweiterung der Regelsätze für fehlerhaftes Verhalten von Teilnehmern

Für die Erweiterung der Regelbasis muss beachtet werden, dass Fehler, wie vorher diskutiert, nur innerhalb der PREPARING-Phase eines Teilnehmers gemeldet werden können, und der Übergang eines Koordinators in den Zustand FINISHING nur stattfinden kann, wenn die Verifikation für die Choreographie-Instanz abgeschlossen ist. Dafür muss sich der Koordinator beim Erzeugen des Transaktionskontextes beim Verifikationsdienst für die Notifikation von Fehlerereignissen anmelden, damit er die entsprechenden Ereignisse über Zustandsänderungen der Choreographie-Instanz mitgeteilt bekommt. In der Implementierung wird dafür die Spezifikation *WS-Eventing* [MWDC11] über Apache Savan genutzt[6], bei der sich der Koordinator einer Transaktion für Ereignisse über die Änderung des Choreographie-Zustandes beim Verifikationsdienst registriert. WS-Eventing ist eine Spezifikation, die den Nachrichtenaustausch zur Implementierung des Beobachter-Musters für Web-Services definiert.

Weiterhin gibt es bei der Nutzung der Verifikation bei der transaktionalen Ablaufkontrolle zwei unterschiedliche Fälle zu beachten. Zum einen kann die Verifikation Integritätsprobleme zu Teilnehmern melden, welche zu dem Teilnehmerverhalten im Laufe der Transaktion

[6]Für Apache Savan siehe `http://axis.apache.org/axis2/java/savan/`.

konform ist. Beispielsweise kann der fehlerhafte Teilnehmer bereits mit einer `Exit`- oder einer `CannotComplete`-Nachricht aus der Transaktion ausgestiegen sein oder auch durch den Koordinator bereits über eine `Cancel`-Nachricht informiert worden sein, dass er seine Arbeit abbrechen soll. Ein Fehler des Teilnehmers ist dem Koordinator also bereits bekannt. Zum anderen ist es auch möglich, dass das Teilnehmerverhalten nicht mit dem von der Verifikation gemeldeten Verhalten übereinstimmt. Dabei meldet der Teilnehmer beispielsweise über eine `Completed`-Nachricht, dass er seine Arbeit aus seiner Sicht erfolgreich beendet hat. Da der Choreographie-Monitor aber die globale Sicht auf alle Interaktionen hat und über die Integritätsbedingungen bestimmen kann, ob die erbrachte Arbeit im Rahmen der Choreographie wirklich erfolgreich war, muss die Ablaufkontrolle die Sicht des Verifikationsdienstes übernehmen und den gemeldeten Teilnehmerstatus überschreiben. Dabei kann ein Überschreiben des Teilnehmerstatus durch die vom Verifikationsdienst gemeldeten Fehler entweder zu einem Abbruch der Transaktion oder zu einem Weiterlaufen führen, je nach dem, ob der Teilnehmer vital für die Transaktion ist oder nicht.

Damit der Status eines Teilnehmers durch den angegebenen Status des Verifikationsdienstes überschrieben werden kann, wird ein gemeldeter Integritätsfehler in der Choreographie für einen Teilnehmer mithilfe des Prädikates $invalid(TX, A)$ beschrieben, welches anzeigt, dass der Teilnehmer A sich im Rahmen der Transaktion TX fehlerhaft verhalten hat. Dabei müssen, wie in Abbildung 5.13 illustriert, die beiden Zeitpunkte vor und nach Einleiten der Demarkation beachtet werden.

Wird ein Fehler *vor* Einleiten der Demarkation gemeldet, wartet der Koordinator bis zum Einleiten der Demarkation, wenn sich alle erwarteten Teilnehmer beim Koordinator registriert haben. Bei Entscheidung zur Demarkation und damit dem Einteilen der Teilnehmer in die Complete- und Cancel-Mengen braucht der Koordinator beispielsweise für Complete-Mengen nur noch zu prüfen, ob für einen Teilnehmer kein Fehler durch den Verifikationsdienst gemeldet worden ist. Dies ist auch in Regel 5.11 entsprechend dargestellt, in dem die vorherige Regel 5.3 um eine *invalid*-Bedingung erweitert wurde. Zusätzlich muss der Koordinator prüfen, ob der Teilnehmer eine `Cancel`-Nachricht erhalten soll. Dafür muss eine neue Regel eingeführt werden, die das fehlerhafte Verhalten respektiert. Nach der Entscheidung der Demarkation bekommt ein Teilnehmer also eine Cancel-Nachricht, wenn er als *invalid* gemeldet wurde. Dies reicht beim Protokoll BACC bereits als Bedingung aus. Allerdings kann ein Teilnehmer beim Protokoll BAPC seinen Status auch selbst melden, weshalb er in diesem Falle keine `Cancel`-Nachricht, sondern später eine `Compensate`-Nachricht bekommen muss. Dass ein Teilnehmer in diesem Fall nicht in der Cancel-Menge enthalten sein darf, wird über Regel 5.12 definiert, in dem als Bedingung geprüft wird, ob ein Teilnehmer seine eigene Arbeit bereits als *completed* gemeldet hat.

$$\begin{aligned} complete(TX, A) \leftarrow\ & participant(TX, A) \wedge \neg exit(TX, A) \wedge \neg cancel(TX, A) \wedge \\ & \neg cannotComplete(TX, A) \wedge \neg invalid(TX, A) \end{aligned} \tag{5.11}$$

$$cancel(TX, A) \leftarrow participant(TX, A) \wedge invalid(TX, A) \wedge \neg completed(TX, A) \tag{5.12}$$

Wird ein Fehler *nach* Einleiten der Demarkation gemeldet, wartet der Koordinator bis zur Entscheidung über den Erfolg der Transaktion, bei der er die Teilnehmer entsprechend über Regel 5.4 neu in Complete- und Vital-Mengen einteilt. Die Zugehörigkeit zur Complete-Menge wird an dieser Stelle neu berechnet und ein fehlerhafter Teilnehmer fällt damit über Regel 5.11 aus der Complete-Menge heraus, auch wenn er vorher darin enthalten gewesen sein kann. Die Bestimmung der Vitalität eines Teilnehmers bleibt allerdings unverändert, da die Eigenschaft der Vitalität nur durch den Aufrufbaum bestimmt wird und nicht von einem Fehler abhängig ist. Wenn die Bedingung $(\forall X)(vital(TX, X) \rightarrow complete(TX, X))$ für einen erfolgreichen Abschluss der Transaktion nicht gilt, dann wird diese abgebrochen und die beendeten Teilnehmer werden kompensiert.

5.2.1.3. Technische Infrastruktur und Implementierung

Die Implementierung der Erweiterungen zur Integration lässt sich leicht zusammenfassen. Zum einen wird die Regelbasis um die Regeln aus dem letzten Abschnitt erweitert und zum anderen müssen Verifikationsdienst sowie Koordinator so erweitert werden, dass sie miteinander kommunizieren können. Die Erweiterung der Regelbasis ist wieder in Prolog realisiert und auch im Anhang A.2.1 dargestellt.

Damit die Kopplung zwischen Verifikations- und Koordinationsdienst möglichst gering bleibt, ist für den Verifikationsdienst eine Ereignisschnittstelle implementiert, bei der sich andere Teilnehmer für bestimmte Ereignisse anmelden können. Dies ist mithilfe des Rahmenwerkes Savan der Apache Foundation und der WS-Eventing-Spezifikation realisiert. Bei Start einer Transaktion werden dafür, wie im letzten Abschnitt beschrieben, dem Koordinator die Endpunktreferenzen des Verifikationsdienstes mitgeteilt sowie weitere Daten wie Choreographie-Typ und Instanzkennung übermittelt. Mit den Daten kann sich der Koordinator am Verifikationsdienst für Ereignisse registrieren und erhält bei Fehlern zur Laufzeit die fehlerverursachenden Rollen.

Zusätzlich benötigt der Koordinator für die Entscheidung über den Erfolg der Transaktion noch die Mitteilung, dass die Choreographie insgesamt vollständig abgelaufen ist, damit kein Fehler mehr nach der Entscheidung über Erfolg oder Misserfolg der Transaktion gemeldet werden kann. Werden noch Fehler nach Einleiten des Festschreibvorgangs gemeldet, kann der Koordinator nur noch den Zustand FAILING betreten, da er seine Entscheidung zu früh getroffen hat und die Vitalitätsatomarität nicht mehr gewährleistet werden kann. Problematisch ist an dieser Stelle allerdings das Feststellen des Endes einer Choreographie. Denn in dem Fall, dass eine Choreographie mit einem nicht beobachtbaren oder einer optionalen Aktivität abschließt, kann der Verifikationsdienst das Ende der Transaktion nicht genau bestimmen. Zwar kann bei einem nicht beobachtbaren Ereignis davon aus-

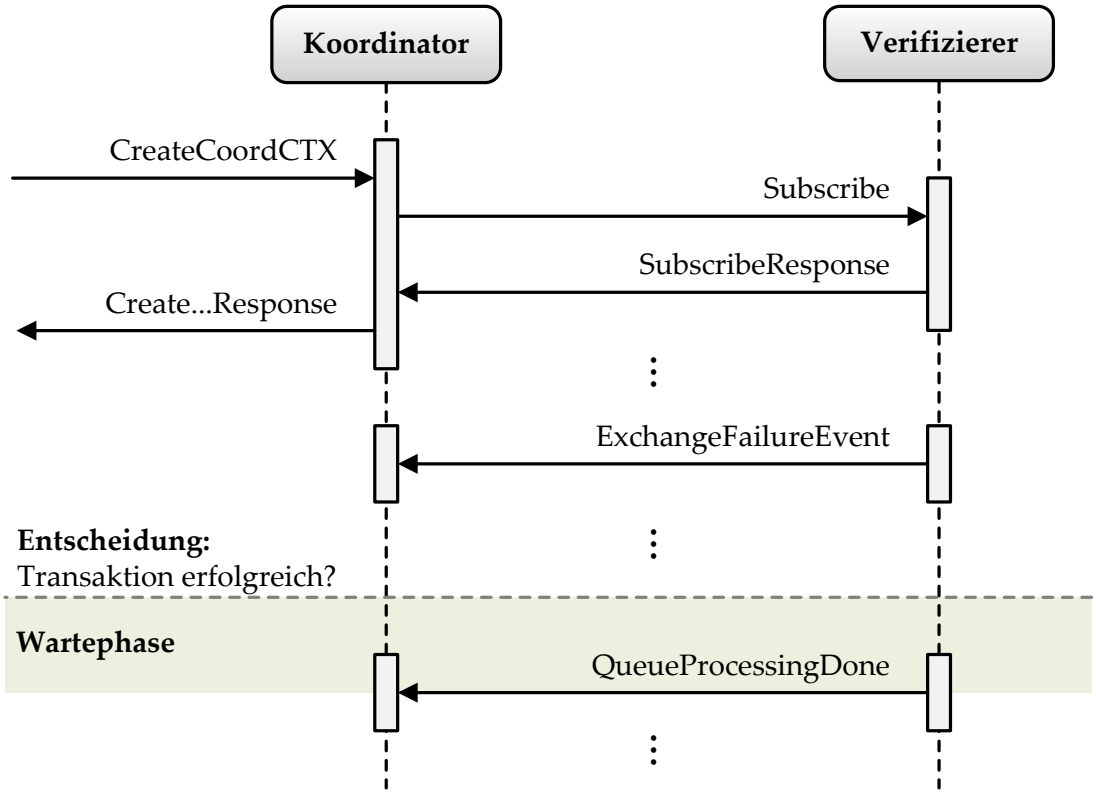


Abbildung 5.14.: Nachrichtenaustausch zwischen Verifikationsdienst und Koordinator

gegangen werden, dass die Choreographie in Bezug auf das beobachtbare Verhalten vollständig durchgeführt worden ist. Allerdings ist dies bei einer optionalen Aktivität nicht mehr so leicht bestimmbar, da der Verifikationsdienst nur vermuten kann, ob die Aktivität noch ausgeführt wird oder nicht. Der Verifikationsdienst besitzt an dieser Stelle nicht das Wissen des Koordinators über die Anzahl der sich registrierenden Teilnehmer. Sind keine Zeitschranken bei einer optionalen Aktivität am Ende einer Choreographie vorgesehen, kann der Verifikationsdienst im schlimmsten Fall eine Choreographie nie als beendet ansehen. Umgekehrt weiß der Koordinator aber über die Aufrufhierarchie, ob sich noch weitere Teilnehmer registrieren werden oder noch Teilnehmer ausstehen, die den Abschluss ihrer Arbeit noch nicht mitgeteilt haben. Haben aber alle Teilnehmer nach Einleiten der Demarkation dem Koordinator den Status ihrer Arbeit gemeldet und der Verifikationsdienst besitzt für die Instanz keine weiteren Nachrichten mehr in der Verifikations-Queue, dann kann aus dem Zustand beider Dienste geschlossen werden, dass die Choreographie nun beendet ist. Dafür benötigt der Koordinator zusätzlich nur den aktuellen Status des Verifikationsdienstes und den aktuellen Queue-Inhalt. Auch darüber gibt die WS-Eventing-Schnittstelle des Verifikationsdienstes entsprechend Auskunft. Ist die aktuelle Verifikations-Queue für die Choreographie-Instanz leer, wird ein Ereignis generiert, welches diese Information an den Koordinator sendet.

Die Abbildung 5.14 zeigt einen exemplarischen Nachrichtenaustausch zwischen einem Koordinator und einem Verifikationsdienst. Bei Erzeugen des Koordinationskontextes regis-

triert sich der Koordinator für Ereignisse beim angegebenen Verifikationsdienst und kann bei Bedarf die Ereignisse `ExchangeFailure` und `QueueProcessingDone` sowie `QueueProcessing` vom Verifikationsdienst erhalten. Das Ereignis `ExchangeFailure` enthält alle Daten der fehlerhaften Interaktionen zwischen zwei Web-Diensten und damit alle Daten die mittels des Prädikats $Happens(Event(Exchange(name, fromRole, toRole), id), t)$ zum Ereignis in der Wissensbasis gespeichert sind. Der Koordinator kann daraus den Sender der Nachricht ausmachen und darüber den fehlerhaften Transaktionsteilnehmer identifizieren. Mittels der Ereignisse `QueueProcessingDone` und `QueueProcessing` kann der Koordinator zusätzlich das Ende der Choreographie bestimmen, wenn alle Teilnehmer entweder mit `Exit` und `CannotComplete` ausgestiegen sind oder ihre Arbeit als erfolgreich abgeschlossen gemeldet haben. Wenn nach einem `QueueProcessing`-Ereignis der Teilnehmerstatus aller Teilnehmer (bis auf *invalid* vom Verifikationsdienst) bekannt ist, dann kann der Koordinator nach dem nächsten `QueueProcessingDone` annehmen, dass die Choreographie abgeschlossen ist, und damit die Entscheidung über den Erfolg der Transaktion treffen. Er sendet dann `Close` und `Compensate` an die jeweiligen Teilnehmer und schließt die Ablaufkontrolle ab.

5.2.2. Verifikationsdienst als Teilnehmer an einer Transaktion

In diesem Abschnitt wird die zweite Variante der in der Einleitung von Abschnitt 5.2 angesprochenen Möglichkeiten vorgestellt, welche den Verifikationsdienst als Teilnehmer in eine Transaktion einbezieht. Damit dies realisiert werden kann, muss zum einen genau definiert sein, wann und wie der verantwortliche Verifikationsdienst einer Transaktion beitreten kann. Zum anderen muss eine Möglichkeit gefunden werden, wie der teilnehmende Verifikationsdienst sich im Fehlerfall verhält und auch die Vitalität in seine Entscheidungen mit einbezieht. Bezieht er die Vitalität der Teilnehmer nicht in sein eigenes Verhalten ein, wird jeder Fehler eines Teilnehmers in einem Abbruch der Transaktion resultieren und damit zu unnötiger Recovery führen.

5.2.2.1. Einbinden eines Verifikationsdienstes als Teilnehmer

Die Einbindung des Verifikationsdienstes muss so erfolgen, dass die Struktur der Aufrufhierarchie nicht verändert wird und trotzdem Fehler von vitalen Teilnehmern in einem Abbruch der Transaktion resultieren. Die beste Stelle für die Integration des Verifikationsdienstes ist daher der Wurzelknoten einer Aufrufhierarchie, in dem der Verifikationsdienst vor den eigentlichen Initiator in die Hierarchie eingebunden wird. Der eigentliche Initiator wird laut Aufrufhierarchie virtuell vom Verifikationsdienst aufgerufen und ist daher ein vitaler Teilnehmer, da keine Auswahlgruppe durch den Verifikationsdienst gestartet wird. Die Regeln des Koordinators brauchen an dieser Stelle nicht verändert werden.

Ist der Verifikationsdienst als Teilnehmer registriert, ergibt sich das Verhalten des Verifikationsdienstes anhand der Fehlerzeitpunkte bei der Verifikation. Auch hier lässt sich anhand

Algorithmus 5.1: Entscheidung über den Fehlereinfluss von Teilnehmern

Input : VerificationQueue Q, ErroneousParticipants P_e, Transaktion T_{id}, RegisteredParticipants N_{reg}, FinishedParticipants N_{done}

```
1  decision ← false;
2  while ¬decision do
3      if N_reg − N_done = 1 ∧ Q = ∅ then
4          V = getVitalSet (T_id);
5          if (∃x)(x ∈ P_e → x ∈ V) then
6              sendNotification (T_id, CannotComplete);
7          else
8              sendNotification (T_id, Completed);
9          end
10         decision ← true;
11     end
12 end
```

von Abbildung 5.13 diskutieren, zu welchen Zeitpunkten Fehler der Teilnehmer auftreten können. Fehler können im Rahmen der Transaktion, wie vorher diskutiert, nur *vor* der Entscheidung über den Transaktionserfolg gemeldet werden. Die Möglichkeiten der Fehlersignalisierung des Verifikationsdienstes sind Nachrichten wie `CannotComplete`, `Fail` oder `Exit`. Da der Verifikationsdienst aber weder aus der Transaktion aussteigen kann noch aufgrund eines Teilnehmerfehlers einen undefinierten Zustand betritt, bleibt nur die `CannotComplete`-Nachricht, die der Dienst bei Integritätsverletzungen versenden kann. Dabei ist es unerheblich, ob diese Nachricht vor oder nach Einleiten der Demarkation gesendet wird, denn laut Protokoll ist die Nachricht zu beiden Zeitpunkten erlaubt. Wird nach Ablauf aller Interaktionen innerhalb einer Choreographie kein Fehler gefunden, so muss der Verifikationsdienst die Nachricht `Completed` an den Koordinator senden.

Problematisch ist dabei die Bestimmung des Zeitpunktes zum Versenden einer der beiden Nachrichten. Im Unterschied zum Ansatz des Einbettens der Entscheidungen vom Verifikationsdienst weiß der Verifikationsdienst nichts über die Ausdehnung des Aufrufbaumes und ob sich bereits alle Teilnehmer registriert haben. Genauso ist es schwierig, den Endzeitpunkt einer Choreographie zu bestimmen, wenn optionale Aktivitäten am Ende einer Choreographie modelliert sind. Wie in der eingebetteten Lösung kann hier anstatt des Verifikationsdienstes der Koordinator Ereignisse über WS-Eventing anbieten, die der Verifikationsdienst zur Entscheidung über das Ende der Choreographie nutzen kann. Notwendige Informationen für die Entscheidung sind die Anzahl der insgesamt registrierten Teilnehmer N_{reg} und die Anzahl der Teilnehmer N_{done}, auf die der Koordinator noch wartet, um die Entscheidung über Erfolg oder Misserfolg der Transaktion zu treffen. Weiterhin benötigt der Verifikationsdienst die Information, wie viele Nachrichten noch in seiner Queue zur Verifi-

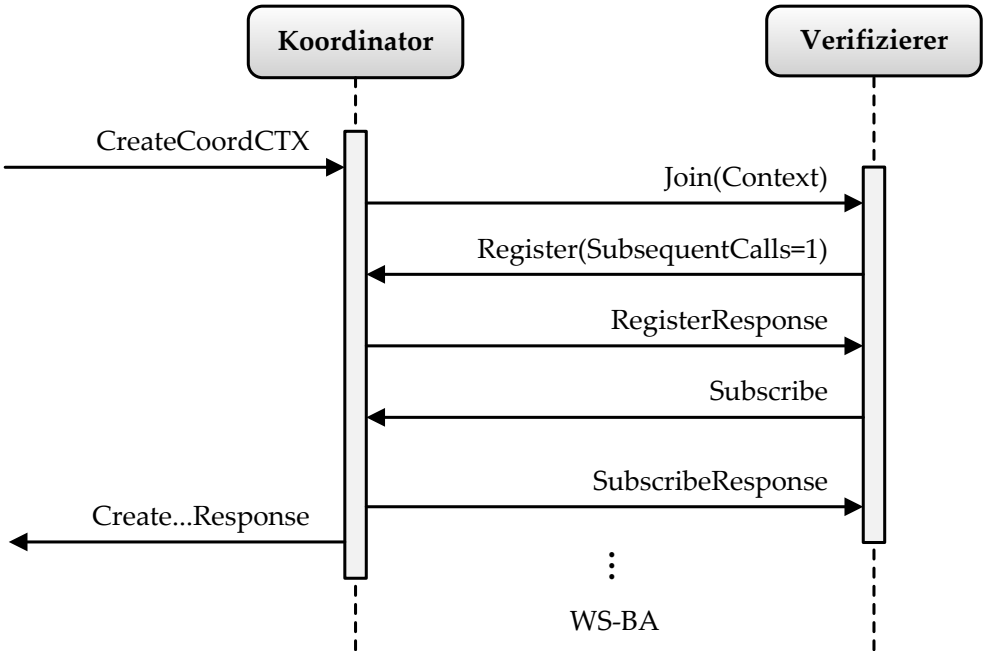


Abbildung 5.15.: Nachrichtenaustausch zwischen Verifikationsdienst und Koordinator bei Einbinden als Teilnehmer

kation anstehen, sowie die Teilnehmer, die vital für die Transaktion sind. Sind diese Daten bekannt, kann mithilfe von Algorithmus 5.1 festgestellt werden, ob ein Fehler eines Teilnehmers Einfluss auf die Transaktion hat. Dabei kann aufgrund der Tatsache, dass alle Teilnehmer bis auf den Verifikationsdienst den Status ihrer Arbeit mitgeteilt haben und einer leeren Verifikations-Queue darauf geschlossen werden, dass keine weiteren Interaktionen mehr in der Choreographie auftreten. Mit Unterstützung der Menge der fehlerhaften Rollen und der Menge der vitalen Teilnehmer in der Transaktion kann entschieden werden, ob der Verifikationsdienst eine `CannotComplete`- oder `Completed`-Nachricht an den Koordinator sendet. Je nach Protokoll geschieht Letzteres entweder auf Antwort einer `Complete`-Nachricht vom Koordinator (BACC) oder unaufgefordert (BAPC).

5.2.2.2. Technische Infrastruktur und Implementierung

Die Implementierung eines Verifikationsdienstes als Teilnehmer an einer Transaktion lässt sich schnell zusammenfassen. Zum einen muss dafür der Verifikationsdienst so erweitert werden, dass er als Teilnehmer innerhalb einer Transaktion agieren kann. Zum anderen muss auch der Koordinator so erweitert werden, dass er für die Arbeit des Verifikationsdienstes benötigte Daten wie beispielsweise die Liste der vitalen Teilnehmer herausgeben kann. Er benötigt daher bestimmte Management-Schnittstellen, damit diese Daten entsprechend herausgegeben werden können sowie eine WS-Eventing Schnittstelle, bei der sich der Verifikationsdienst für bestimmte Ereignisse wie die Anzahl der insgesamt registrierten Teilnehmer sowie die Antworten der Teilnehmer registrieren kann.

Damit das Einbinden des Verifikationsdienstes vor dem Einbinden der eigentlichen Teilnehmer gelingen kann, kann der Verifikationsdienst durch den Koordinator bei Erzeugung des Koordinationskontextes zur Registrierung aufgefordert werden. Dieses Vorgehen ist auch in Abbildung 5.15 gezeigt, in dem der Initiator der Transaktion seinen Kontext erst dann mitgeteilt bekommt, wenn der Verifikationsdienst der Transaktion beigetreten ist und sich für bestimmte Ereignisse registriert hat. Damit der Koordinator den richtigen Verifikationsdienst einbinden kann, müssen innerhalb der `CreateCoordinationContext`-Nachricht der entsprechende Dienst und die zu verifizierende Choreographie-Instanz übergeben werden[7]. Geschieht die Einbindung auf diese Weise, bleibt die grundlegende Struktur im Aufrufbaum erhalten und die Regeln im Koordinator müssen nicht angepasst werden. Die Transaktion ist genau dann erfolgreich, wenn die Verifikation erfolgreich ist.

Damit der Verifikationsdienst seine Entscheidungen treffen kann, benötigt er die Anzahl der insgesamt registrierten Teilnehmer. Dies teilt der Koordinator mithilfe eines `AllRegistered(Count)`-Ereignisses mit, welches zum einen anzeigt, dass sich alle erwarteten Teilnehmer an der Transaktion registriert haben, und zum anderen angibt, wie viele dies insgesamt sind. Zusätzlich sendet der Koordinator Ereignisse darüber, ob ein Teilnehmer den Status seiner Arbeit mithilfe einer `Completed`-, `Exit`-, `CannotComplete` oder auch `Fail`-Nachricht beendet hat. Die Information wird mithilfe eines `Done(Reason)`-Ereignisses übertragen. Der Verifikationsdienst kann dann mithilfe der Anzahl der beendeten Teilnehmer und der Anzahl der registrierten Teilnehmer bestimmen, ob nur noch auf seine eigene Entscheidung gewartet wird.

Wird nur noch auf die Entscheidung des Verifikationsdienstes gewartet, benötigt der Verifikationsdienst die Liste der vitalen Teilnehmer für seine Entscheidung. Dabei kann er mit der Liste der Teilnehmer-Identifikatoren nichts weiter anfangen, da er ja nur die Rollen der Teilnehmer kennt. Auch hier kann das gleiche Verfahren zur Korrelation wie im ersten Ansatz genutzt werden, dass registrierende Teilnehmer dem Koordinator auch die Rolle innerhalb der Choreographie mitteilen. Der Koordinator kann dann aufgrund dieser Informationen die Beziehung zwischen den Rollen und den Transaktionsteilnehmern herstellen und auf Anfrage des Verifikationsdienstes eine Liste der für die Transaktion vitalen Rollen mitteilen.

5.2.3. Vergleich der Ansätze und Evaluation

Im Vergleich der beiden Ansätze zur Integration kann festgestellt werden, dass prinzipiell mit beiden Ansätzen das Gleiche erreicht wird. Allerdings ist bei der eingebetteten Lösung die Trennung von Verifikationslogik und Ablaufkontrolle deutlich besser gelöst. Die gesamte Logik des Transaktionskoordinators bleibt damit bis auf die Einbeziehung von Fehlern in den Regeln unabhängig von der Implementierung des Verifikationsdienstes. Bei der zweiten Lösung wird dagegen ein Teil der Entscheidungen an den Verifikationsdienst abgetre-

[7] Siehe Abschnitt 5.2.1.1, über die Angabe des Verifikationsdienstes.

ten, denn der Verifikationsdienst benötigt für die Entscheidung, ob vitale Teilnehmer von Fehlern betroffen sind, explizit das Wissen über den Aufrufbaum, welches er sich erst vom Koordinator holen muss. Damit ist der Kopplungsgrad bei der zweiten Lösung deutlich stärker als bei der eingebetteten Lösung, bei der der Verifikationsdienst die Fehler nur über einfache Ereignisse meldet, aber der Koordinator die Entscheidung über den Erfolg einer Transaktion unabhängig trifft. Im zweiten Fall übernimmt der Verifikationsdienst zudem die Entscheidung über den Fehlereinfluss auf die Transaktion. Die beiden Lösungen lassen sich damit auch als *Einbettungs-* und als *Delegationslösung* charakterisieren.

Um den Aufwand der beiden Lösungen zur Laufzeit zu vergleichen, kann das Nachrichtenaufkommen in beiden Ansätzen verglichen werden. Bei der ersten Lösung gibt es zum einen zwei Queue-Ereignisse, die den Start und das Ende der Verifikationsverarbeitung für eine oder mehrere Nachrichten kennzeichnen. Im schlimmsten Fall resultiert eine Verifikation einer Nachricht in zwei Ereignissen, im besten Fall werden nur zwei Ereignisse für alle Nachrichten erzeugt. Im zweiten Ansatz bleibt die Anzahl der Ereignisse linear von der Anzahl der registrierten Teilnehmer abhängig, da für jeden gemeldeten Abschluss der Arbeit ein Ereignis generiert wird. In beiden gibt es zusätzlich noch Ereignisse, welche entweder die Fehler oder das Einleiten der Demarkation anzeigen. Im Grenzfall von sehr vielen gemeldeten Fehlern und sehr schnell erledigter Verifikation ist das Nachrichtenaufkommen der ersten Lösung deutlich höher als bei der zweiten Lösung und wird daher in höheren Zeitaufwand bei der Verarbeitung resultieren. Dies trifft aber nur im schlimmsten Fall zu; im normalen Betrieb verhalten sich beide Lösungen ähnlich, je nachdem wie viele Nachrichten gleichzeitig verifiziert werden müssen.

Weiterhin ist der Implementierungsaufwand der beiden Lösungen unterschiedlich. Während bei der Einbettungslösung der Verifikationsdienst nur Ereignisse versenden muss, die während der Verifikation so auch auftreten, muss in der Delegationslösung der Verifikationsdienst um entsprechende Logik erweitert werden, damit der Algorithmus 5.1 realisiert werden kann. Genauso müssen Web-Service-Schnittstellen im Koordinator sowie Schnittstellen zur Ermittlung der vitalen Teilnehmer realisiert werden. In der Einbettungslösung beschränkt sich die Erweiterung um bestimmte WS-Eventing-Schnittstellen und einer kurzen Erweiterung der Regeln.

Beide vorgestellten Realisierungen der Integration sind in Bezug auf WS-Coordination und WS-BusinessActivities standardkonform, auch wenn zur Realisierung der Korrelation der beiden Welten bestimmte Erweiterungen bei der `CreateCoordinationContext`-Nachricht vorgesehen sind. Über die WS-Eventing-Schnittstelle ist zudem die Kommunikation zwischen den beiden Diensten standardisiert, auch wenn die Kommunikation zwischen beiden je nach Lösung etwas unterschiedlich ist.

Trotzdem der Nachrichtenaufwand bei der Einbettungslösung im schlimmsten Fall höher sein kann, ist im direkten Vergleich die Einbettungslösung der Delegationslösung vorzuziehen, weil zum einen der Implementierungsaufwand kleiner ist und zum anderen auch die logische Trennung der beiden Dienste besser gelöst ist.

5.3. Diskussion und Zusammenfassung

In diesem Abschnitt ist zum einen die Infrastruktur zur Transaktionskontrolle mithilfe eines autonomen Koordinators und zum anderen die Integration der Verifikation in die Ablaufkontrolle vorgestellt worden, um bei erkannten Integritätsfehlern von Teilnehmern entsprechende Backward-Recovery durchzuführen.

Die in Abschnitt 3.2.3 entwickelten Anforderungen an die transaktionale Koordination sind dabei auf Basis eines autonomen Koordinators vollständig umgesetzt worden. Die *Backward-Recovery* in Choreographie-Umgebungen wird durch einen autonomen Koordinator unterstützt, welcher die Ergebnisse der Verifikation in die Ablaufkontrolle einfließen lassen kann. Teilnehmern an einer Choreographie wird aufgrund der *Integration der Fehlererkennung* in den Koordinator die Möglichkeit gegeben, direkt zur Laufzeit auf Fehler zu reagieren. Eine wichtige Annahme der Integration ist allerdings, dass für eine automatisierte Recovery die Nachrichten des Transaktionsprotokolls von der Verifikation ausgeschlossen sind und sich Teilnehmer auch genauso verhalten, wie das Transaktionsprotokoll es vorsieht. Beispielsweise darf ein Teilnehmer auf eine `Close`-Nachricht die Transaktion nicht mit einer `Exit`-Nachricht verlassen, da sonst die Vitalitätsatomarität nicht mehr eingehalten werden kann. Diese Einschränkung gilt allerdings für alle Transaktionsprotokolle, denn wenn sich die Teilnehmer einer Transaktion nicht an ein Transaktionsprotokoll halten, können Garantien wie Atomarität nicht eingehalten werden. Hält sich ein Teilnehmer nicht an ein Transaktionsprotokoll, kann dies auch leicht durch den Koordinator festgestellt werden. Der Verifikationsdienst braucht dies also nicht zusätzlich zu überprüfen und kann die Entscheidung über den Ausgang der Transaktion dem Koordinator überlassen. Die Integration der Verifikation in die Ablaufkontrolle zeigt zusätzlich ein laufendes Beispiel, wie ermittelte Integritätsfehler zur automatisierten Fehlerbehebung genutzt werden können. Dabei wird allerdings kein Fehler aktiv durch den Koordinator behoben, sondern die Teilnehmer müssen selbst wissen, wie sie ihre Arbeit rückgängig machen können. Die Transaktionsprotokolle stoßen über bestimmte Nachrichten nur die Aktionen zum Festschreiben, Kompensieren und Abbrechen von Aktivitäten an. Eine weitere Kontrolle, ob die Dienste dieses intern auch wirklich umsetzen, kann an dieser Stelle nicht erfolgen.

Zusätzlich werden durch den autonomen Koordinator *Transaktionsprotokolle auch in Choreographie-Umgebungen einsetzbar*. Durch den Koordinator werden jetzt auch Szenarien bei der transaktionalen Koordination unterstützt, bei denen nicht mehr der Initiator derjenige Teilnehmer ist, welcher die Demarkation einer Transaktion anstößt. Die *Entscheidung über den Ausgang einer Transaktion durch einen Koordinator* ist vom Initiator *entkoppelt*, da der Koordinator jetzt aufgrund von *mitgeteilten Anwendungsentscheidungen der Initiatoren von Auswahlgruppen* selbstständig entscheiden kann, ob die Demarkation eingeleitet werden muss. Er kann jetzt selbst ableiten, welche Teilnehmer ihre Arbeit abschließen sollen und welche nicht. Durch deklaratives Beschreiben des Koordinatorverhaltens kann die Koordinationslogik sehr leicht erweitert werden, um eine Integration der Fehlererkennung in den Ko-

ordinator zu ermöglichen. Weiterhin bleibt aufgrund der Struktur des Aufrufbaumes und der Weitergabe von Anwendungsentscheidungen über die Initiatoren von Auswahlgruppen die *Anwendungslogik strikt von der Koordinationslogik von Transaktion getrennt*, auch wenn verschiedene Anwendungsentscheidungen in die Ablaufkontrolle einfließen. Integritätsfehler in Choreographien lassen sich außerdem aufgrund der Kategorisierung der Teilnehmer in *vital* und *nicht vital* besser handhaben, denn ein Fehler muss nicht mehr in einen Abbruch von Transaktionen resultieren.

Die Evaluation zeigt, dass der implementierte Koordinator sehr gut skaliert und gut als generischer Koordinator beispielsweise als Teil eines Enterprise-Service-Busses zur Koordination von Transaktion eingesetzt werden kann. Zwar ist die Implementierung vollständig standardkonform, allerdings haben sich verschiedene Entwurfsentscheidungen als problematisch erwiesen, welche die Interoperabilität mit anderen Implementierungen des WS-Coordination-Standards einschränken. Leider sind damit Interoperabilitätsprobleme mit anderen Transaktionsrahmenwerken auf Basis von WS-Coordination zu erwarten. Aber es kann gezeigt werden, dass in den Standards verschiedene Interoperabilitätsprobleme wie beispielsweise die Indentifikation von Teilnehmern, die Art der Kontextverteilung und das Festlegen der Demarkation inhärent vorhanden sind, die nicht allein aufgrund der getroffenen Erweiterungen dieses Ansatzes entstanden sind. An dieser Stelle sollte der Standard eindeutig nachgebessert werden (siehe auch Abschnitt 5.1.4.2). Bei der Implementierung hat sich weiterhin gezeigt, dass die Standards zwar die individuellen Konversationszustände zwischen einem Teilnehmer und dem Koordinator definieren, aber globale Zustände für den Koordinator nur schwer ersichtlich sind. In dieser Arbeit sind daher auch die globalen Zustände des Koordinators definiert, in welchen die Entscheidungszeitpunkte des Koordinators besser herausgearbeitet sind. Zusätzlich ist damit auch das Verhalten eines Koordinators im Rahmen von `Fail`-Nachrichten definiert, welche einen undefinierten Zustand eines Teilnehmers kennzeichnen. Da in undefinierten Zuständen die Vitalitätsatomarität nicht gewährleistet werden kann, bleibt der Koordinator selbst im Zustand FAIL, was im Normalfall ein manuelles Eingreifen eines Administrators erfordert.

Der vorgestellte Ansatz über die Regelsätze unterstützt auch die im WS-Coordination-Standard vorgesehenen Konzepte wie Interposition von Koordinatoren und Zeitschranken von Aktivitäten. Zum einen ist für die Interposition keine Erweiterung notwendig, denn will ein Teilnehmer sich nicht am angegebenen Koordinator registrieren, sondern lieber seinen eigenen Koordinator nutzen, wird die Anzahl der aufgerufenen Teilnehmer nicht verändert. Der erwartete Teilnehmer registriert sich zwar nicht, aber dafür registriert sich der neue Koordinator als Teilnehmer innerhalb der Transaktion. Ruft dieser Web-Service mit dem eigenen Koordinator jetzt weitere Dienste auf, werden diese vom neu eingefügten Koordinator verwaltet. Der Wurzelkoordinator bekommt davon allerdings nichts mit. Ist der neue Koordinator beim Wurzelkoordinator registriert, ändert sich nichts an der Koordinationslogik im Aufrufbaum, da dieser Koordinator wie ein normaler Teilnehmer geführt wird, der das Verhalten seiner untergebenen Teilnehmer kapselt. Zum anderen werden durch die

Implementierung auch die für Koordinatoren vorgesehenen Zeitschranken von Aktivitäten unterstützt. Die Zeitschranken geben die Zeit in Millisekunden an, ab wann eine Aktivität als abgelaufen deklariert wird. Der Koordinator kann dann unabhängig die Entscheidung treffen, eine Transaktion zurückzusetzen. Dies kann er allerdings nur so lange machen, wie noch keine Entscheidung über den Erfolg der Transaktion getroffen wurde. Ist die Entscheidung über den Erfolg der Transaktion getroffen, muss diese auch unabhängig von Zeitschrankenverletzungen durchgesetzt werden. Insofern bleibt die Koordinationslogik auch bei Zeitschranken gleich, außer dass zu den Entscheidungszeitpunkten des Koordinators die Zeitschranken zusätzlich geprüft werden, bevor der Koordinator den Aufrufbaum für seine Entscheidungen benutzt.

Das Rahmenwerk zur Koordination ist auch als Erweiterung der Standards WS-Coordination sowie WS-AtomicTransaction und WS-BusinessActivities implementiert und als Open-Source-Projekt unter `http://tracg.sourceforge.net` veröffentlicht.

6. Zusammenfassung und Ausblick

Diese Arbeit hat Konzepte zur Ablaufkontrolle von Prozess-Choreographien erarbeitet und in Prototypen umgesetzt. Die dabei erarbeiteten Beträge werden in Abschnitt 6.1 zusammengefasst und bewertet. Abschnitt 6.2 gibt abschließend einen Ausblick auf weitere mögliche Arbeiten und in dieser Arbeit nur am Rande behandelte Themen.

6.1. Zusammenfassung der Ergebnisse

Die vorliegende Arbeit zeigt zunächst, dass es zum einen für Choreographien keinen vollständigen Ansatz zur Laufzeitverifikation gibt und zum anderen, dass bestehende Standards zur transaktionalen Kontrolle und damit zur Fehlerbehebung in Prozessen Defizite bei der Anwendung in Choreographien aufweisen. Diese Arbeit hat für beide Teilbereiche Lösungen erarbeitet, die zum einen die Laufzeitverifikation von Choreographien ermöglichen sowie eine automatisierte Fehlerbehebung bei Integritätsverletzungen mithilfe von Transaktionsprotokollen unterstützen. Die Beiträge dieser Arbeit in Bezug auf diese Teilbereiche lassen sich in Kurzform wie folgt zusammenfassen:

- **Nicht-Intrusive Sensoren:** (Abschnitt 4.1): Auf Basis von in einem Enterprise-Service-Bus eingebetteten Sensoren können Nachrichten und Ereignisse an eine Monitoring-Komponente weitergeleitet werden, sodass das beobachtbare Verhalten von Choreographie-Teilnehmern über Stellvertreterdienste pragmatisch ermittelt werden kann. Bei der Implementierung der Sensoren müssen Anforderungen wie die Sensordatenkonsistenz sowie das Sicherstellen der Ortstransparenz des ESBs erfüllt werden. Dafür muss zum einen die Ortstransparenz des Service-Busses sichergestellt werden, sodass jegliche Kommunikation wie beispielsweise auch asynchrone Kommunikation durch den Bus geleitet wird. Dies wird durch die realisierten Mediatoren erreicht. Zum anderen muss sichergestellt werden, dass der Monitor das beobachtete Verhalten der beteiligten Dienste sieht und damit der Vorgang des Weiterleitens von Nachrichten zum Monitor und zum eigentlichen Empfänger als atomare Einheit ausgeführt wird. Auch dies wird durch realisierte Mediatoren im Bus erreicht. Beide Konzepte zusammen führen bei der Überwachung nur einen geringen Mehraufwand ein, um das beobachtbare Verhalten der Teilnehmer zu ermitteln. Voraussetzung für die Sensoren ist allerdings das Kennzeichnen von bestimmten Korrelationsdaten innerhalb von Nachrichten, damit diese später auch auf ein formales Modell für Integritätsbedingungen

von Choreographien abgebildet werden können. Sie sind weiterhin nur in der Lage, das beobachtbare Verhalten von Teilnehmern zu protokollieren und schränken damit die Autonomie der Organisationen bei der Implementierung sowie Wartung ihrer eigenen Systeme nicht weiter ein, da sie nur das beobachtbare Verhalten der Dienste überwachen.

- **Ein formales Modell für Integritätsbedingungen von Choreographien** (Abschnitt 4.2 sowie 4.3): Auf Basis von WS-CDL sind in dieser Arbeit zwei unterschiedliche Ansätze zur formalen Spezifikation von Choreographien mit dem Ziel der Laufzeitverifikation entwickelt worden. Zum einen ist dies ein regelbasiertes System, welches direkt zur einfachen Überwachung von Interaktionen einer Choreographie eingesetzt werden kann. Es besitzt allerdings einige Einschränkungen, die mithilfe des zweiten Ansatzes auf Basis des Ereigniskalküls behoben werden können. Der erste Ansatz ist nicht in der Lage, zeitbezogene Integritätsbedingungen einer Choreographie zu verifizieren. Er unterstützt damit nicht alle modellinhärenten Eigenschaften von WS-CDL. Weiterhin unterstützt er nicht alle Reichweiten von Integritätsbedingungen, welche sich in diesem Ansatz auf nachrichtenbasierte, instanzbasierte und interaktionsbasierte Integritätsbedingungen beschränken. Inhaltliche Anforderungen und damit selbst definierte Integritätsbedingungen sind nicht vorgesehen, da keine zeitbezogenen Integritätsbedingungen möglich sind. Diese Beschränkungen werden mit dem zweiten Ansatz gelöst, da sich mit Unterstützung des Models auf Basis des Ereigniskalküls nicht nur funktionale Bedingungen einer Prozess-Choreographie, sondern auch Integritätsbedingungen für Choreographien im Allgemeinen überwachen lassen. Dies können Integritätsbedingungen für Nachrichten (beispielsweise Datentypen einer Nachricht), Bedingungen für Interaktionen (beispielsweise Zeitschranken), Bedingungen für einzelne Instanzen (beispielsweise der Kontrollfluss) und auch instanzübergreifende Bedingungen[1] sein, die für den Erfolg eines organisationsübergreifenden Prozesses eingehalten werden müssen. Beide formalen Beschreibungen auf Basis von Regelsystemen können direkt aus einer WS-CDL-Beschreibung abgeleitet werden und durch eigene Bedingungen manuell erweitert werden.

- **Effizienter Soll/Ist-Vergleich** (Abschnitt 4.2.3 sowie 4.3.7): Im Rahmen der Evaluation der entwickelten formalen Modelle zur Verifikation lässt sich zusammenfassend sagen, dass sich einfache Modelle auf Basis von einfachen prädikatenlogischen Regeln sehr gut zur Interaktionskontrolle während der Laufzeit einsetzen lassen. Die Zeit, die ein heute üblicher Rechner zur Verifikation einer einzigen Nachricht benötigt, beläuft sich dabei auf wenige Millisekunden. Sollen aber auch weitere Integritätsbedingungen wie non-funktionale Bedingungen nicht nur instanzbasiert für eine Choreographie, sondern auch instanzübergreifend für mehrere Prozessinstanzen geprüft werden, wird

[1] Siehe auch Abschnitt 3 zu instanzübergreifenden Integritätsbedingungen: „Ein Auto aus einer Autovermietung darf nicht zweimal vom Hof gefahren sein, ohne zwischendurch zurückgekommen zu sein“.

mit dem Cached-Ereigniskalkül zwar ein Mehraufwand eingeführt, der sich aber auf heute üblichen Rechnern im Bereich von 30-50ms pro Nachricht bewegt und damit gut zur Laufzeitüberwachung geeignet ist.

Die in dieser Arbeit entwickelten Anforderungen an die Laufzeitverifikation sind damit erfüllt. Eine weitere wichtige in dieser Arbeit zu beantwortende Frage war allerdings die Reaktion auf Integritätsverletzungen, die durch die Laufzeitverifikation erkannt werden. Hierfür ist ein Recovery-Ansatz auf Basis von Transaktionsprotokollen vorgestellt worden, welcher die Verifikationsergebnisse direkt in die Ablaufkontrolle von Transaktionen einbinden kann. Die Idee, einen Recovery-Mechanismus über Transaktionen zu nutzen, kommt direkt aus dem Datenbankbereich. Sind Integritätsbedingungen bei Datenmanipulation in einer Datenbank verletzt, werden die verursachenden Transaktionen innerhalb einer Datenbank zurückgesetzt und die Effekte ihrer Aktionen rückgängig gemacht. Transaktionsmechanismen bei Integritätsverletzungen auch in Prozessen zu nutzen ist daher naheliegend. Eine wichtige Voraussetzung dafür ist allerdings die generelle Anwendbarkeit von gängigen Transaktionsprotokollen in Choreographie-Umgebungen. In dieser Arbeit ist gezeigt worden, dass genau dieses nicht generell angenommen werden kann, da die Transaktionskontrolle in den klassischen Protokollen immer vom Initiator einer Transaktion abhängt. Die vorhandenen Transaktionsprotokolle wie das WS-BusinessActivities-Protokoll müssen daher, wie im nachfolgenden Abschnitt beschrieben, erweitert werden. Für den Bereich der Fehlerbehebung im Falle von Integritätsverletzungen sind im Rahmen dieser Arbeit also folgende Teilergebnisse erreicht worden:

- **Transaktionale Kontrolle für Choreographien** (Abschnitt 5.1): Zum einen sind die Herausforderungen von transaktionaler Kontrolle in Choreographien definiert sowie zum anderen ein neues Konzept entwickelt worden, um Transaktionen in Prozess-Choreographien zu steuern. Das Konzept erweitert die bisherigen Ansätze der transaktionalen Kontrolle von Prozessen um einen Koordinator, der aufgrund von Regeln und einem Aufrufbaum autonom über die Demarkation einer Transaktion entscheiden kann. In bisherigen Ansätzen ist der Initiator einer Transaktion immer derjenige Teilnehmer, welcher die Demarkation der Transaktion festlegt und das Ergebnis dem Koordinator mitteilt. In vielen praktischen Szenarien kann der Initiator dieses aber nicht entscheiden, weshalb der Koordinator in diesem Ansatz diese Rolle übernimmt. Weiterhin ist in dieser Arbeit bei der Realisierung eines autonomen Koordinators zum einen die *saubere Trennung von Anwendungs- und Koordinationslogik* im Koordinator realisiert. Diese Trennung erfolgt mithilfe eines Aufrufbaumes sowie der Modellierung von Auswahlgruppen, bei der Initiatoren von Auswahlgruppen dem Koordinator das Ergebnis einer Auswahl übermitteln. Die Entscheidungen, wann welche Teilnehmer gemäß Protokoll welche Nachrichten bekommen, lassen sich auf Basis des Aufrufbaumes und der Auswahlgruppen mithilfe einfacher Regeln fällen. Dadurch, dass für WS-BusinessActivities die Anwendungs- von der Koordinationslogik

getrennt ist, ist der Koordinator generisch nutzbar. Er kann also für verschiedene Choreographie-Typen zum Einsatz kommen, ohne ihn auf die spezielle Anwendungslogik eines Choreographie-Typs anzupassen. Zum anderen ist über den Aufrufbaum auch die *Abhängigkeit vom Initiator* deutlich reduziert, welcher nicht mehr den Gesamtüberblick über eine Choreographie haben muss, um die Entscheidung der Demarkation zu treffen. Der Gesamtüberblick über registrierte Teilnehmer und die Ergebnisse von Auswahlgruppen ergibt sich durch den Aufrufbaum und über die Nachrichten der lokalen Entscheidungen für die jeweiligen Auswahlgruppen.

- **Integration der Fehlererkennung in die Ablaufkontrolle** (Abschnitt 5.2): Die Anbindung der Fehlererkennung an die transaktionale Kontrolle ist eine Optimierung des Transaktionsabschlusses im Abbruchfall. Wird bereits während der Interaktionen ein Fehler erkannt, muss eine transaktionale Kontrolle nicht mehr auf den Abschluss der Aktivitäten warten, sondern kann direkt korrigierend eingreifen. Dabei ist in Choreographie-Szenarien zu sehen, dass Fehler einzelner Teilnehmer nicht unbedingt zu einem Abbruch einer Transaktion führen müssen. Für diese Fälle werden Teilnehmer in diesem Ansatz vom Koordinator entsprechend ihrer Aufrufhierarchie in vitale und nicht-vitale Teilnehmer eingeteilt. Der Koordinator kann jetzt aufgrund der *Vitalität* der Teilnehmer entscheiden, ob eine Transaktion erfolgreich abgeschlossen werden kann oder nicht. Dabei muss der Koordinator zusätzlich die Ergebnisse der Laufzeitverifikation mit einbeziehen, wofür in dieser Arbeit zwei Ansätze vorgestellt worden sind. Zum einen ist eine Einbettungslösung vorgeschlagen worden, bei der der Koordinator und ein Verifikationsdienst möglichst lose gekoppelt sind. Weiterhin ist eine Delegationslösung vorgestellt worden, bei der der Verifikationsdienst entscheidet, ob ein vitaler Teilnehmer einen Fehler verursacht. Im Vergleich ist die Einbettungslösung praktikabler als die Delegationslösung, da der Verifikationsdienst noch um weitere Anwendungslogik zur Entscheidung über die Vitalität der Teilnehmer erweitert werden muss. In der Einbettungsvariante ist daher die Koordinationslogik im Koordinator besser von der Verifikationslogik des Verifikationsdienstes getrennt. In beiden Ansätzen ist aber die Integration der Erkennung von Integritätsverletzungen und der Nutzung dieser Informationen in den Transaktionsablauf möglich. Die Integration ermöglicht dadurch eine direkte Fehlerreaktion, falls sich Teilnehmer einer Choreographie nicht so wie erwartet verhalten.

Zusammenfassend lässt sich festhalten, dass eine Laufzeitverifikation mithilfe des Ereigniskalküls für Choreographien gut realisierbar und auch eine automatische Fehlerbehebung durch Transaktionskontrolle möglich ist. Im Vergleich zu den bekannten Ansätzen kann in dieser Arbeit WS-CDL als W3C-Standard zur Laufzeit verifiziert werden und der dafür entwickelte Ansatz kann in Analogie zu wissensbasierten Systemen auch als *wissensbasiertes Monitoring* bezeichnet werden. Im zweiten Teil der Arbeit trägt der entwickelte generische Transaktionskoordinator dazu bei, dass Transaktionsprotokolle zur Integritätssicherung in

Choreographien genutzt werden und dass zusätzlich die Verifikationsergebnisse zur automatisierten Fehlerbehebung genutzt werden können. Diese Arbeit trägt also dazu dabei, dass Choreographien robuster ausgeführt werden können und dass das jeweilige Verhalten in Bezug auf Integritätsbedingungen für alle Choreographie-Teilnehmer transparent ist. Eine wichtige Anforderung in organisationsübergreifenden eGovernment-Umgebungen ist beispielsweise die Nachvollziehbarkeit von Prozessen und damit die Sicherstellung, dass sich Teilnehmer an Choreographien genauso wie erwartet verhalten. Das erwartete Verhalten lässt sich mit Integritätsbedingungen gut modellieren und mithilfe der Ansätze dieser Arbeit überprüfen.

6.2. Ausblick

Die in dieser Arbeit entwickelten Konzepte zur Integritätsüberwachung und zur Korrektur von Integritätsverletzungen für Prozess-Choreographien zeigen einen vollständigen Ansatz zur automatisierten Erkennung von Integritätsverletzungen und deren Behebung. Die dafür entwickelten Prototypen stellen allerdings reine Forschungsprototypen dar. Bevor die Konzepte in kommerziellen Systemen Anwendung finden können, sind noch weitere Schritte notwendig:

- **Modellierung von Integritätsbedingungen:** In Abschnitt 4.1.3 ist die generelle Architektur für den Verifikationsdienst mit seinen Schnittstellen vorgestellt worden. Mit Unterstützung von WS-CDL definierte Choreographien können dabei über einen Interpreter direkt in das formale interne Modell überführt werden. Sollen aber weitere inhaltliche Integritätsbedingungen definiert werden, kann dies auch über eine weitere Schnittstelle und über bestimmte Erweiterungspunkte (siehe Abschnitt 4.3.4) geschehen. Allerdings benötigt der Modellierer dieser Integritätsbedingungen ein sehr gutes Verständnis über Aufbau und Struktur der WS-CDL-Axiome und des internen Modells. An dieser Stelle fehlt beispielsweise eine einfache domänenspezifische Sprache zur Definition der Integritätsbedingungen und weitere Werkzeugunterstützung, um beispielsweise instanzübergreifende Integritätsbedingungen definieren zu können. Diese Werkzeuge unterstützen den Modellierer dabei, sinnvolle und korrekte Integritätsbedingungen auszudrücken. Um einen vernünftigen Einsatz zu gewährleisten, müssen dafür auch entsprechende Visualisierungskonzepte umgesetzt werden, welche die Darstellung, Prüfung und später auch Fehlerdarstellung besser benutzbar machen.

- **Service-Level-Agreements:** Gibt es eine Sprache und Werkzeugunterstützung zur Definition von WS-CDL-unabhängigen Integritätsbedingungen, dann können auch weitere non-funktionale Bedingungen einer Choreographie wie beispielsweise Service-Level-Agreements (SLAs) überwacht werden. Ansätze wie [FSTC04] beschreiben, wie

SLAs mithilfe des Ereigniskalküls überwacht werden können. Die Konzepte können auch hier angewendet werden, allerdings müssen sie auf die Zuständigkeitskette der hierarchischen Choreographie-Modelle angepasst werden. Wie dies allerdings genau im Rahmen der in dieser Arbeit entwickelten Konzepte umgesetzt werden kann, muss in weiteren Arbeiten untersucht werden. Weiterhin sind auch die in dieser Arbeit entwickelten Fehlerbehebungsmaßnahmen nur auf Backward-Recovery ausgelegt. Falls bestimmte SLAs nicht eingehalten werden können, darf natürlich ein fehlerfrei laufender Prozess nicht abgebrochen werden, nur weil im Schnitt zu wenige Prozessinstanzen innerhalb einer Zeitspanne erfolgreich beendet werden. Sollen also weitere non-funktionale Bedingungen im Rahmen von SLAs überwacht und eingehalten werden, müssen dafür andere Maßnahmen zur Behebung der Probleme eingesetzt werden, die im Bereich der Forward-Recovery und beispielsweise der autonomen Systeme zu suchen sind. Beispielsweise können SLA-Verletzungen in einer Neukonfiguration der beteiligten Dienste resultieren. Wie genau autonome Choreographie-Teilnehmer auf solche SLA-Verletzungen reagieren, ist aber intern den Organisationen überlassen. Sollen SLAs unterstützt werden, kann dies mit dem Überwachungsansatz der Integritätsbedingungen zwar umgesetzt werden, allerdings muss bei der Definition der Integritätsbedingungen und der SLAs festgelegt werden, welche harten Bedingungen in einem Rücksetzen von Aktivitäten resultieren und welche (weichen) SLA-Verletzungen beispielsweise in einer Rekonfiguration von Diensten resultiert. An dieser Stelle wird in der Arbeit bisher angenommen, dass alle Integritätsbedingungen harte Verletzungen darstellen, wenn ein Teilnehmer vital ist. Hier muss untersucht werden, wie sich der Ansatz auch um weichere in SLAs definierten Bedingungen erweitern lässt.

- **Interoperabilitätsprobleme in Transaktionsspezifikationen:** Bei der Realisierung des Transaktionsrahmenwerkes sind, wie im folgenden beschrieben, diverse Interoperabilitätsprobleme und Ungenauigkeiten bei der Definition von WS-Coordination und seinen Transaktionsspezifikationen identifiziert worden. Dies betrifft für WS-Coordination und WS-AtomicTransaction genau drei Dinge. Zum einen wird bei WS-AtomicTransaction nicht genau definiert, wer sich für das Completion-Protokoll registrieren kann. Damit ist unklar, welche Teilnehmer genau den Transaktionsabschluß auslösen können und was im Falle von Konflikten getan wird, wenn sich mehrere Teilnehmer für das Completion-Protokoll registrieren und sich wiedersprechende Anweisungen wie Commit und Rollback geben. Zum anderen betrifft dies WS-Coordination, wo das gesamte Identitätsmanagement der einzelnen Teilnehmer nicht definiert ist, und damit, wie ein Koordinator Teilnehmer bei den Interaktionen genau identifizieren kann. Genauso fehlt in WS-Coordination die genaue Definition, wie Transaktionskontexte an Teilnehmer weitergereicht werden (SOAP-Header oder SOAP-Body). Verschiedene standardkonforme Implementierungen sind damit mit hoher Wahrscheinlichkeit nicht kompatibel zueinander. Unzulänglichkeiten finden sich aber auch in WS-

BusinessActivities, wo in der Spezifikation nicht definiert ist, wie die Demarkation einer Transaktion eingeleitet werden soll. Diese Arbeit nutzt zur Entscheidung über die Demarkation den Aufrufbaum und Erweiterungen für das Protokoll, um Auswahlgruppen zu entscheiden. Trotzdem sind generische Transaktionskoordinatoren auf Basis des Standards aufgrund der Interoperabilitätsprobleme und Ungenauigkeiten schwer zu implementieren. Werden Koordinatoren implementiert, sind auch hier Teilnehmer mit hoher Wahrscheinlichkeit inkompatibel zu anderen WS-BA-Koordinatoren, auch wenn die jeweiligen Implementierungen standardkonform sind. Hier sollten die WS-Coordination- und WS-AtomicTransaction-Standards in Bezug auf das Identitätsmanagement, der Verteilung der Transaktionskontexte sowie Ungenauigkeiten des Completion-Protokolls beispielsweise mit den in dieser Arbeit vorgeschlagenen Erweiterungen nachgebessert werden. Genauso muss für WS-Business-Activities auf jeden Fall die Demarkation definiert werden, und damit die Frage, wann diese genau eingeleitet werden soll und welche Teilnehmer für den erfolgreichen Ausgang der Transaktion notwendig sind und welche nicht. Auch hier helfen die in dieser Arbeit vorgeschlagenen Erweiterungen.

Insgesamt bieten sich vor allem bei den Integritätsbedingungen und der Modellierung und Verifikation von Service-Level-Agreements noch weitere Entwicklungsmöglichkeiten an, die auch in weiteren Projekten über Forschungskooperationen mit der Industrie bereits vorgesehen sind.

A. Anhang

A.1. WS-CDL-Axiome für das Ereigniskalkül: Prolog-Implementierung

In diesem Anhang werden die Prolog-Implementierungen für die Axiome aus Abschnitt 4.3 gelistet sowie die Implementierungen der Hilsprädikate aufgeführt. Die Dokumentation aus dem Orginalquellcode ist teilweise enthalten, ansonsten ist der Quellcode hier im Anhang so wie besehen ohne weiteren Kommentar abgedruckt.

A.1.1. Hilfsregeln

Quellcode A.1: Tools

```
1  % liefert den Vater eines Knotens
2  getParent(Parent, Child) :-
3    ground(Child),
4    memberActivities(Parent, Members),
5    member(Child, Members).
6
7  % liefert ein Kind eines Knotens
8  getChild(Parent, Child) :-
9    ground(Parent),
10   memberActivities(Parent, Members),
11   member(Child, Members).
12
13 % liefert Kinder eines Knotens
14 getChildren(Parent, Child) :-
15   ground(Parent),
16   getChild(Parent, X),
17   getChildren(X, Child).
18 getChildren(Parent, Child) :-
19   ground(Parent),
20   getChild(Parent, Child).
21
22 % Liefert Vaterknoten
```

```
getParents(Child, Parent):-
  ground(Child),
  getParent(Child, X),
  getParents(X, Parent).
getParents(Child, Parent) :-
  ground(Child),
  getParent(Child, Parent).

% Hilfsmethoden für Vergleiche
testValue(VALUE,"!=",TESTVALUE) :-
  VALUE \= TESTVALUE.
testValue(VALUE,"=",TESTVALUE) :-
  VALUE = TESTVALUE.
testValue(VALUE,"<",TESTVALUE) :-
  VALUE < TESTVALUE.
testValue(VALUE,">",TESTVALUE) :-
  VALUE > TESTVALUE.
```

Quellcode A.2: Tools: Komplexe Aktivitäten generell

```
% Vereinfachung
isComplex(NODE) :-
  (isSequence(NODE);isChoice(NODE);isParallel(NODE);isWorkUnit(NODE)).

% Liefert, ob Aktivität eine auslassbare Aktivität enthält
containsOmittable(NODE, ID, T) :-
  isComplex(NODE),
  getChild(NODE, CHILD),
  holdsAt(omittable(CHILD, ID), T).

% liefert, ob ein Assignknoten vorhanden ist.
nodeContainsAssign(NODE, ASSIGN) :-
  isComplex(NODE),
  getChild(NODE, ASSIGN),
  isAssign(ASSIGN).

% Zählt beendete und überspringbare Teilaktivitäten
getChildCompleted(_, [], _, _, 0).
getChildCompleted(NODE, MEMBERS, ID, T, COUNT) :-
  [STEP | NEXT] = MEMBERS,
  (
    (
      (
```

```
        holdsAt(complete(STEP, ID), T)
      ;
        holdsAt(omittable(STEP, ID), T)
      ),
      CUR is 1
    )
  ;
    (
      not(holdsAt(complete(STEP, ID), T)),
      CUR is 0
    )
  ),
  getChildCompleted(NODE, NEXT, ID, T, NEXTVAL),
  COUNT is CUR + NEXTVAL.

% Liefert, ob zwei Aktivitäten 'inParallel' sind
inParallel(NAME1, NAME2) :-
  not(NAME1 = NAME2),
  getParents(NAME1, PAR),
  getParents(NAME2, PAR),
  isParallel(PAR),
  memberActivities(PAR, Members),
  member(Mem1, Members),
  (
    getParents(NAME1, Mem1);
    NAME1 = Mem1
  ),
  member(Mem2,Members),
  (
    getParents(NAME2, Mem2);
    NAME2 = Mem2
  ),
  not(Mem1 = Mem2).
```

Quellcode A.3: Tools: Sequenz

```
% Liefert den aktuell aktiven Knoten einer Sequenz (der erste Knoten der
% memberActivities, welcher nicht beendet oder überspringbar ist).
% Sequentielles Durchlaufen der Liste, da wir ja in einer Sequenz sind.
getSequenceStepActive(NODE, MEMBERS, ID, T, STEP) :-
  (
    (
      [STEP | _] = MEMBERS,
```

```
8       not(holdsAt(omittable(STEP, ID), T)),
9       not(holdsAt(complete(STEP, ID), T)),
10      !
11    );
12    (
13      [_ | NEXT] = MEMBERS,
14      getSequenceStepActive(NODE, NEXT, ID, T, STEP)
15    )
16  ).
17
18 % Wrapper für getSequenceStepActive
19 sequenceStepActive(NODE, ID, T, STEP) :-
20   isSequence(NODE),
21   memberActivities(NODE, MEMBERS),
22   getSequenceStepActive(NODE, MEMBERS, ID, T, STEP).
23
24 % Wrapper für getChildCompleted
25 sequenceCompleted(NODE, ID, T) :-
26   isSequence(NODE),
27   memberActivities(NODE, MEMBERS),
28   getChildCompleted(NODE, MEMBERS, ID, T, COUNT),
29   length(MEMBERS,L),
30   L=COUNT.
```

Quellcode A.4: Tools: Parallel

```
1  % Liefert, ob für eine Parallelaktivität alle Teilaktivitäten (ohne Current)
2  % entweder beendet oder überspringbar sind.
3  parallelNearCompletion(NODE, ID, CURRENT, T) :-
4    isParallel(NODE),
5    memberActivities(NODE, MEMBERS),
6    member(CURRENT, MEMBERS),
7    getChildCompleted(NODE, MEMBERS, ID, T, COUNT),
8    (
9      % Wenn Current auch die Bedinungen erfüllt, dann einen abziehen...
10     (
11       (
12         holdsAt(complete(STEP, ID), T)
13       ;
14         holdsAt(omittable(STEP, ID), T)
15       )
16     ),
17     succ(C, COUNT)
```

```
  ;
    % ... ansonsten alles beim alten
    C = COUNT
  ),
  succ(COUNT,Next),
  length(MEMBERS,L),
  L=Next.

% Liefert, ob für eine Parallelaktivität alle Teilaktivitäten entweder beendet
% oder überspringbar sind.
parallelCompleted(NODE, ID, T) :-
  isParallel(NODE),
  memberActivities(NODE, MEMBERS),
  getChildCompleted(NODE, MEMBERS, ID, T, COUNT),
  length(MEMBERS,L),
  L=COUNT.
```

Quellcode A.5: Tools: Choice

```
% Liefert einen aktiven Knoten der Choice
choiceStepActive(NODE, ID, T, STEP) :-
  isChoice(NODE),
  getChild(NODE, STEP),
  holdsAt(valid(STEP, ID), T).
```

Quellcode A.6: Tools: WorkUnit

```
% Schleifenbedingung (Guard ist entsprechend)
repeatTrue(NODE, ID, T) :-
  repeatExpression(NODE, getVariable(VARNAME, '', ''), OP, TESTVALUE),
  (
    (
      % Falls Variable gesetzt ist, dann testen
      holdsAt(valueOf(getVariable(VARNAME, '', '', _), VALUE, ID), T),
      testValue(VALUE, OP, TESTVALUE)
    );
    (
      % Ansonsten ist Schleifenbedingung wahr (Siehe WS-CDL-Doku)
      not(holdsAt(valueOf(getVariable(VARNAME, '', '', _), _, ID), T)),
      true
    )
  ).

```

```
% Guard
guardTrue(NODE, ID, T) :-
  guardExpression(NODE, getVariable(VARNAME, '', ''), OP, TESTVALUE),
  (
    (
      % Falls Variable gesetzt ist, dann testen
      holdsAt(valueOf(getVariable(VARNAME, '', '', _), VALUE, ID), T),
      testValue(VALUE, OP, TESTVALUE)
    );
    (
      % Ansonsten ist Guard wahr (Siehe WS-CDL-Doku)
      not(holdsAt(valueOf(getVariable(VARNAME, '', '', _), _, ID), T)),
      true
    )
  ).
```

A.1.2. Axiome

Quellcode A.7: Basisaktivitäten und Assign

```
% Exchange
initiates(event(X, ID), valid(X, ID), T) :-
  basicActivity(X),
  happens(event(X, ID), T).

initiates(event(X, ID), complete(NODE, ID), T) :-
  basicActivity(NODE),
  happens(event(X, ID), T),
  X = NODE.

% Setzen von Variablen via Assign beim Betreten einer Sequenz/Parallel
% (Vereinfachtes Vorgehen! Vorraussetzung: Nur ein Assign pro Aktivität)
initiates(event(X, ID), valueOf(getVariable(VARNAME, '', '', ROLENAME),
    VALUE, ID), T) :-
  happens(event(X, ID), T),
  initiates(event(X, ID), valid(NODE, ID), T),
  ( isSequence(NODE); isParallel(NODE) ),
  not(holdsAt(valid(NODE, ID), T)),
  nodeContainsAssign(NODE, ASSIGN),
  assign(_,ROLENAME,getVariable(VARNAME,'',''), expression(VALUE)) = ASSIGN.

% Setzen von Variablen via Assign bei Beenden von Choice und WorkUnit
```

```
23 % durch omittable (vereinfachtes Vorgehen bei Choice! Vorraussetzung:
24 % Nur ein omittable/assign zu dem Zeitpunkt innerhalb Choice, sonst
25 % nicht entscheidbar!)
26 initiates(event(X, ID), valueOf(getVariable(VARNAME, '', '', ROLENAME),
27     VALUE, ID), T) :-
28   happens(event(X, ID), T),
29   initiates(event(X, ID), complete(NODE, ID), T),
30   (isChoice(NODE); isWorkUnit(NODE)),
31   holdsAt(omittable(NODE, ID), T),
32   nodeContainsAssign(NODE, ASSIGN),
33   assign(_,ROLENAME,getVariable(VARNAME,'','',), expression(VALUE)) = ASSIGN.
34
35 % Das Fluent mit dem Variablenwert wird terminiert, wenn ein neuer Wert
36 % gesetzt wird.
37 terminates(event(X, ID), valueOf(getVariable(VARNAME, '', '', ROLENAME),
38     VALUE, ID), T) :-
39   initiates(event(X, ID), valueOf(getVariable(VARNAME, '', '', ROLENAME),
40     NEWVALUE, ID), T), not(VALUE = NEWVALUE).
```

Quellcode A.8: Komplexe Aktivitäten: generell

```
1  initiates(event(X, ID), error(NODE, ID), T) :-
2    root(NODE),
3    happens(event(E, ID), T),
4    not(initiates(event(X, ID), valid(NODE, ID), T)).
5
6  terminates(event(X, ID), valid(NODE, ID), T) :-
7    isComplex(NODE),
8    happens(event(X, ID), T),
9    not(holdsAt(complete(NODE, ID), T)),
10   not(holdsAt(omittable(NODE, ID), T)),
11   not(initiates(event(X, ID), valid(NODE, ID), T)),
12   initiates(event(X, ID), valid(NODE2, ID), T),
13   not(inParallel(NODE,NODE2)).
```

Quellcode A.9: Komplexe Aktivitäten: überspringbar

```
1 % Wenn eine Aktivität strukturell zu jeder Zeit auslassbar ist,
2 % dann gilt auch das Fluent immer
3 holdsAt(omittable(NODE, _), _) :-
4   isOmittable(NODE).
5
6 % Wenn eine Aktivität strukturell auslassbar ist, dann ist sie
```

```
7  % immer beendet
8  holdsAt(complete(NODE, _), _) :-
9    isOmittable(NODE).
10
11 % Wenn zu einer bestimmten Zeit eine Aktivität nur auslassbare
12 % Aktivitäten besitzt, dann ist sie selbst auslasbar.
13 % Sonderfall WorkUnit: Siehe unten
14 holdsAt(omittable(NODE, ID), T) :-
15   not(isOmittable(NODE)),
16   isComplex(NODE),
17   not(isWorkUnit(NODE)),
18   not((
19     getChild(NODE, CHILD),
20     not(holdsAt(omittable(CHILD, ID), T))
21   )).
```

Quellcode A.10: Komplexe Aktivitäten: Sequenz

```
1  % Wenn für den aktuellen Knoten der Sequenz die Nachricht valide
2  % ist, dann ist die Sequenz auch valide.
3  initiates(event(X, ID), valid(NODE, ID), T) :-
4    isSequence(NODE),
5    happens(event(X, ID), T),
6    not(holdsAt(complete(NODE, ID), T)),
7    sequenceStepActive(NODE, ID, T, CURRENT),
8    initiates(event(X, ID), valid(CURRENT, ID), T).
9
10 % Wenn die Nachricht valide ist und nach dem Valideren der letzte
11 % Knoten auf completed steht, dann ist auch die Sequenz fertig
12 % (braucht nur geprüft werden, wenn keine auslassbaren Aktivitäten
13 % da sind --> Wenn auslassbar: Sonderfälle)
14 initiates(event(X, ID), complete(NODE, ID), T) :-
15   isSequence(NODE),
16   happens(event(X, ID), T),
17   not(containsOmittable(NODE, ID, T)),
18   sequenceStepActive(NODE, ID, T, CURRENT),
19   initiates(event(X, ID), complete(CURRENT, ID), T),
20   memberActivities(NODE, MEMBERS),
21   last(CURRENT, MEMBERS),
22   LAST = CURRENT.
23
24 % ----------------------------------------
25 % Sonderfälle:
```

```
26 % ----------------------------------------
27
28 % Wenn die letzten Aktivitäten der Sequenz auslassbar sind (also kein aktiver
29 % Schritt existiert), aber die Sequenz noch nicht beendet ist (passiert durch
30 % zeitweise auslassbare Aktivitäten), dann muss die Sequenz beendet werden.
31 % Zusätzlich wird die so beendete Sequenz für diesen Zeitpunkt als "auslassbar"
32 % markiert, damit eine darüberliegende Sequenz mit dem nächsten Schritt weiter-
33 % macht. Nebenbedingung: Es kann nur beendet werden, wenn das Event nicht zu
34 % einer "parallelen" Aktvität gehört und nich im Unterbaum vorkommt
35 initiates(event(X, ID), complete(NODE, ID), T) :-
36   isSequence(NODE),
37   happens(event(X, ID), T),
38   not(holdsAt(complete(NODE, ID), T)),
39   % passiert nur bei auslassbaren Aktivitäten
40   containsOmittable(NODE, ID, T),
41   % Event passiert und validiert irgendeinen anderen Basisknoten
42   basicActivity(NODE2),
43   initiates(event(X, ID), valid(NODE2, ID), T),
44   not(NODE2 = NODE),
45   % Der andere validierte Knoten darf nicht parallel zu dem Aktuellen sein
46   % (und nicht im Teilbaum)
47   not(inParallel(NODE, NODE2)),
48   not(getChildren(NODE, NODE2)),
49   % ... und die Sequenz darf keine weiteren beendbaren Knoten enthalten.
50   sequenceCompleted(NODE, ID, T).
51
52 holdsAt(omittable(NODE, ID), T) :-
53   isSequence(NODE),
54   happens(event(X, ID), T),
55   not(holdsAt(complete(NODE, ID), T)),
56   % passiert nur bei auslassbaren Aktivitäten
57   containsOmittable(NODE, ID, T),
58   % Event passiert und validiert irgendeinen anderen Basisknoten
59   basicActivity(NODE2),
60   initiates(event(X, ID), valid(NODE2, ID), T),
61   not(NODE2 = NODE),
62   % Der andere validierte Knoten darf nicht parallel zu dem Aktuellen sein
63   % (und nicht im Teilbaum)
64   not(inParallel(NODE, NODE2)),
65   not(getChildren(NODE, NODE2)),
66   % ... und die Sequenz darf keine weiteren beendbaren Knoten enthalten.
67   sequenceCompleted(NODE, ID, T).
```

```
68
69 % Wenn ein Schritt der Sequenz auslassbar ist, muss dieser beendet werden, wenn
70 % ein nachfolgender Schritt in der Sequenz nach dem Ausgelassenen validiert bzw.
71 % beendet wird. Er muss beendet werden, da er sonst durch "sequenceStepActive"
72 % bei zeitweise auslassbaren Aktivitäten diese Aktvitäte erneut zur Validierung
73 % herangezogen wird...
74 initiates(event(X, ID), complete(NODE, ID), T) :-
75   happens(event(X, ID), T),
76   not(isOmittable(NODE)), % Strukturell sind eh beendet.
77   holdsAt(omittable(NODE, ID), T),
78   getParent(NODE, PARENT),
79   isSequence(PARENT),
80   % Wenn ein Schritt der Sequenz validiert wird...
81   sequenceStepActive(NODE, ID, T, CURRENT),
82   initiates(event(X, ID), valid(CURRENT, ID), T),
83   % ... und der Schritt "später" als die überspringbare Aktivität ist
84   memberActivities(PARENT, LIST),
85   nth0(I, LIST, NODE),
86   nth0(IC, LIST, CURRENT),
87   I < IC.
```

Quellcode A.11: Komplexe Aktivitäten: Choice

```
1  % Die Choice ist valide, wenn der aktuell ausgewählte Knoten valide ist. Ist kein
2  % Knoten ausgewählt, wird ein valider Knoten gesucht (wobei NoAction und
3  % SilentAction bei der Suche übersprungen werden, siehe Sonderfall)
4  initiates(event(X, ID), valid(NODE, ID), T) :-
5    isChoice(NODE),
6    not(holdsAt(complete(NODE, ID), T)),
7    happens(event(X, ID), T),
8    (
9      (
10       choiceStepActive(NODE, ID, T, CURRENT),
11       initiates(event(X, ID), valid(CURRENT, ID), T)
12     );
13     (
14       not(choiceStepActive(NODE, ID, T, CURRENT)),
15       getChild(NODE, CURRENT),
16       not(holdsAt(omittable(CURRENT, ID), T)),
17       initiates(event(X, ID), valid(CURRENT, ID), T)
18     )
19   ).
20
```

```
21 % Wenn der aktive Knoten mit einer Nachricht beendet wird, dann ist auch die
22 % Choice fertig. Falls kein Knoten ausgewählt ist, wird geprüft, ob ein
23 % Knoten direkt beendet
24 initiates(event(X, ID), complete(NODE, ID), T) :-
25   isChoice(NODE),
26   happens(event(X, ID), T),
27   (
28     (
29       choiceStepActive(NODE, ID, T, CURRENT),
30       initiates(event(X, ID), complete(CURRENT, ID), T)
31     );
32     (
33       not(choiceStepActive(NODE, ID, T, CURRENT)),
34       getChild(NODE, CURRENT),
35       initiates(event(X, ID), complete(CURRENT, ID), T)
36     )
37   ).
38
39 % ----------------------------------------
40 % Sonderfälle:
41 % ----------------------------------------
42
43 % Wenn die Teilaktivität der Choice durch Omittable beendet wird,
44 % muss das omittable auch noch oben durchgereicht werden.
45 holdsAt(omittable(NODE, ID), T) :-
46   isChoice(NODE),
47   happens(event(X, ID), T),
48   not(holdsAt(complete(NODE, ID), T)),
49   containsOmittable(NODE, ID, T),
50   (
51     (
52       choiceStepActive(NODE, ID, T, CURRENT),
53       initiates(event(X, ID), complete(CURRENT, ID), T),
54       holdsAt(omittable(CURRENT, ID), T)
55     )
56   ;
57     (
58       not(choiceStepActive(NODE, ID, T, CURRENT)),
59       getChild(NODE, CURRENT),
60       initiates(event(X, ID), complete(CURRENT, ID), T),
61       holdsAt(omittable(CURRENT, ID), T)
62     )
```

```
63    ).
64
65  % Wenn keiner der Knoten validiert und eine Teilaktivität auslassbar ist, dann wird
66  % Choice als auslassbar markiert: Die Vateraktivität entscheidet dann, welchen
67  % Effekt das Auslassen der Aktvität hat. Nebenbedingungen: Die Aktvitität darf noch
68  % nicht betreten (valid) oder beendet (complete) sein.
69  holdsAt(omittable(NODE, ID), T) :-
70    isChoice(NODE),
71    happens(event(X, ID), T),
72    % Noch nicht beendete und betreten
73    not(holdsAt(complete(NODE, ID), T)),
74    not(holdsAt(valid(NODE, ID), T)),
75    % Enthält übersprinbare
76    containsOmittable(NODE, ID, T),
77    % ... und die Choice darf nicht mit dem Event validieren...
78    not(initiates(event(X, ID), valid(NODE, ID), T)).
79
80  % Choice beenden, wenn Vateraktivität einer Choice beendet wird
81  initiates(event(X, ID), complete(NODE, ID), T) :-
82    isChoice(NODE),
83    happens(event(X, ID), T),
84    containsOmittable(NODE, ID, T),
85    not(holdsAt(complete(NODE, ID), T)),
86    not(holdsAt(valid(NODE, ID), T)),
87    holdsAt(omittable(NODE, ID), T),
88    getParent(NODE, PARENT),
89    initiates(event(X, ID), complete(PARENT, ID), T).
```

Quellcode A.12: Komplexe Aktivitäten: Parallel

```
1   % Die Parallel-Aktivität ist valide, wenn ein Knoten validiert wird.
2   initiates(event(X, ID), valid(NODE, ID), T) :-
3     isParallel(NODE),
4     not(holdsAt(complete(NODE, ID), T)),
5     happens(event(X, ID), T),
6     getChild(NODE, Child),
7     not(holdsAt(complete(Child, ID), T)),
8     initiates(event(X, ID), valid(Child, ID), T).
9
10  % Die Parallelaktivität is beendet, wenn das letzte Kind erfolgreich beendet
11  initiates(event(X, ID), complete(NODE, ID), T) :-
12    isParallel(NODE),
13    happens(event(X, ID), T),
```

```
14    getChild(NODE, Child),
15    not(holdsAt(complete(Child, ID), T)),
16    initiates(event(X, ID), complete(Child, ID), T),
17    parallelNearCompletion(NODE, ID, Child, T).
18
19  % ----------------------------------------
20  % Sonderfall:
21  % ----------------------------------------
22
23  % Wenn die letzten Aktivitäten auslassbar sind, aber Parallel noch nicht beendet
24  % ist, dann muss Parallel beendet werden. Zusätzlich wird die so beendete Parallel-
25  % Aktivität für diesen Zeitpunkt als "auslassbar" markiert, damit eine darüber-
26  % liegende Aktivität mit dem "nächsten" Schritt weitermacht.
27  initiates(event(X, ID), complete(NODE, ID), T) :-
28    isParallel(NODE),
29    happens(event(X, ID), T),
30    not(holdsAt(complete(NODE, ID), T)),
31    % Event passiert und validiert irgendeinen anderen Basisknoten
32    basicActivity(NODE2),
33    initiates(event(X, ID), valid(NODE2, ID), T),
34    not(NODE2 = NODE),
35    % Der andere validierte Knoten darf nicht parallel zu dem Aktuellen sein
36    % (und nicht im Teilbaum)
37    not(inParallel(NODE, NODE2)),
38    not(getChildren(NODE, NODE2)),
39    % ... und die Parallelaktvitiät darf keine weiteren beendbaren Knoten enthalten.
40    parallelCompleted(NODE, ID, T).
41
42  holdsAt(omittable(NODE, ID), T) :-
43    isParallel(NODE),
44    happens(event(X, ID), T),
45    not(holdsAt(complete(NODE, ID), T)),
46    % Event passiert und validiert irgendeinen anderen Basisknoten
47    basicActivity(NODE2),
48    initiates(event(X, ID), valid(NODE2, ID), T),
49    not(NODE2 = NODE),
50    % Der andere validierte Knoten darf nicht parallel zu dem Aktuellen sein
51    % (und nicht im Teilbaum)
52    not(inParallel(NODE, NODE2)),
53    not(getChildren(NODE, NODE2)),
54    % ... und die Parallelaktvitiät darf keine weiteren beendbaren Knoten enthalten.
55    parallelCompleted(NODE, ID, T).
```

Quellcode A.13: Komplexe Aktivitäten: WorkUnit

```
% Die WorkUnit ist valide, wenn die Teilaktivität validiert wird.
% Fall 1: WorkUnit noch nicht valide, dann Guard prüfen
% Fall 2: WorkUnit ist bereits valide (und betreten), dann Guard nicht mehr prüfen
initiates(event(X, ID), valid(NODE, ID), T) :-
  isWorkUnit(NODE),
  not(holdsAt(complete(NODE, ID), T)),
  happens(event(X, ID), T),
  (
    (
      % Fall 1
      not(holdsAt(valid(NODE, ID), T)),
      guardTrue(NODE, ID, T)
    );
    (
      % Fall 2
      holdsAt(valid(NODE, ID), T)
    )
  ),
  getChild(NODE, CHILD),
  initiates(event(X, ID), valid(CHILD, ID), T).

% WorkUnit beenden
% Fall 1: WorkUnit noch nicht valide, dann Guard prüfen
% Fall 2: WorkUnit ist bereits valide (und betreten), dann Guard nicht mehr prüfen
initiates(event(X, ID), complete(NODE, ID), T) :-
  isWorkUnit(NODE),
  happens(event(X, ID), T),
  not(repeatTrue(NODE, ID, T)),
  (
    (
      % Fall 1
      not(holdsAt(valid(NODE, ID), T)),
      guardTrue(NODE, ID, T)
    );
    (
      % Fall2
      holdsAt(valid(NODE, ID), T)
    )
  ),
  getChild(NODE, CHILD),
  initiates(event(X, ID), complete(CHILD, ID), T).
```

```
42
43 % ----------------------------------------
44 % Schleifen
45 % ----------------------------------------
46
47 % Repeat-Event feuern, wenn WorkUnit-Teilaktivität beendet wird, aber ein
48 % erneuter Schleifendurchlauf getätigt werden soll. Das Event setzt zum
49 % nächsten Zeitpunkt alle Kinder der WorkUnit zurück
50 initiates(event(X, ID), repeat(NODE, ID), T) :-
51   isWorkUnit(NODE),
52   happens(event(X, ID), T),
53   getChild(NODE, CHILD),
54   initiates(event(X, ID), complete(CHILD, ID), T),
55   guardTrue(NODE, ID, T),
56   repeatTrue(NODE, ID, T),
57   succ(T, NT),
58   update(repeat(NODE, ID), NT).
59
60 % Repeat terminieren, wenn Schleifenende erreicht ist
61 terminates(event(X, ID), repeat(NODE, ID), T) :-
62   isWorkUnit(NODE),
63   happens(event(X, ID), T),
64   getChild(NODE, CHILD),
65   initiates(event(X, ID), complete(CHILD, ID), T),
66   not(repeatTrue(NODE, ID, T)).
67
68 % Alle Unterknoten bei erneutem Schleifendurchlauf zurücksetzen
69 terminates(repeat(NODE, ID), complete(ANY, ID), T) :-
70   isWorkUnit(NODE),
71   happens(repeat(NODE, ID), T),
72   getChildren(NODE, ANY).
73
74 terminates(repeat(NODE, ID), valid(ANY, ID), T) :-
75   isWorkUnit(NODE),
76   happens(repeat(NODE, ID), T),
77   getChildren(NODE, ANY).
78
79 % ----------------------------------------
80 % Sonderfälle:
81 % ----------------------------------------
82
83 % Wenn WorkUnit nicht blockierend und Guard false ist, dann kann WorkUnit
```

```
84  % übersprungen werden (allerdings nur, wenn die Aktvität nicht schon läuft!).
85  holdsAt(omittable(NODE, ID), T) :-
86    isWorkUnit(NODE),
87    not(isOmittable(NODE)),
88    not(holdsAt(valid(NODE, ID), T)),
89    not(isBlocking(NODE)),
90    not(guardTrue(NODE, ID, T)).
91
92  % Wenn bei WorkUnit der Guard wahr ist, kann die Workunit übersprungen werden,
93  % wenn das Kind überspringbar ist
94  holdsAt(omittable(NODE, ID), T) :-
95    isWorkUnit(NODE),
96    not(isOmittable(NODE)),
97    happens(event(X, ID), T),
98    (
99      (
100       not(holdsAt(valid(NODE, ID), T)),
101       guardTrue(NODE, ID, T)
102     );
103     (
104       holdsAt(valid(NODE, ID), T)
105     )
106   ),
107   getChild(NODE, CHILD),
108   holdsAt(omittable(CHILD, ID), T).
109
110 % WorkUnit beenden, wenn Vateraktivität beendet wird
111 % Nebenbedingung: Noch nicht betreten, selbst überspringbar
112 initiates(event(X, ID), complete(NODE, ID), T) :-
113   isWorkUnit(NODE),
114   happens(event(X, ID), T),
115   containsOmittable(NODE, ID, T),
116   not(holdsAt(complete(NODE, ID), T)),
117   not(holdsAt(valid(NODE, ID), T)),
118   holdsAt(omittable(NODE, ID), T),
119   getParent(NODE, PARENT),
120   initiates(event(X, ID), complete(PARENT, ID), T).
```

A.2. Koordinationsregeln für Transaktionskoordinatoren: Prolog-Implementierung

In diesem Anhang ist die Prolog-Implementierung für den Aufrufbaum aus Abschnitt 5.1 dargestellt. Die Dokumentation aus dem Orginalcode ist teilweise enthalten, ansonsten ist der Quellcode hier wie besehen ohne weitere Kommentare vorgestellt.

A.2.1. Regeln

Quellcode A.14: Vitalität von Teilnehmern

```
% Die Wurzel ist immer vital
vital(TX,A) :- participant(TX,A), not(invocation(TX,_,A)).

% Wenn ein Teilnehmer ausgerufen wird, der sich selbst nicht als nonvital gemeldet
% hat und nicht Teil einer Auswahl ist, dann ist auch dieser Teilnehmer vital, wenn
% der Vater vital ist.
vital(TX,A) :-
  invocation(TX,V,A), not(partOfChoice(TX,A)), not(invital(TX,A)), vital(TX,V).

% Wenn ein Teilnehmer in einer Auswahl ist und ausgewählt wird, ist er vital, wenn
% der Vaterknoten vital ist.
vital(TX,A) :- invocation(TX,V,A), partOfChoice(TX,A), chose(TX,A), vital(TX,V).
```

Quellcode A.15: Complete- und Cancel-Mengen

```
% Cancel:
% ---------------------------------------
% Wenn ein Teilnehmer aufgerufen wurde, dessen Vater bereits ausgetreten ist, nicht
% beenden kann oder auch bereits in der Cancel-Menge ist, dann muss der Knoten ein
% Cancel bekommen, wenn er nicht selbst usgetreten ist oder nicht beenden kann.
cancel(TX,B) :-
  invocation(TX,A,B), (cancel(TX,A);exit(TX,A);cannotComplete(TX,A)),
  not(exit(TX,B)), not(cannotComplete(TX,B)).

% Wenn ein Teilnehmer Teil einer Auswahlmenge ist und nicht ausgewählt wurde, muss
% er eine Cancel-Nachricht bekommen, es sei denn er ist schon selbst ausgetreten
% und kann nicht beenden.
cancel(TX,B) :-
  partOfChoice(TX,B), not(chose(TX,B)),
  not(exit(TX,B)), not(cannotComplete(TX,B)).

```

```
17 % Teilnehmer, die Verifikationsdienst als fehlerhaft gemeldet hat, werden aussor-
18 % tiert (aber nur dann, wenn sie nicht wie in BAPC schon Completed gemeldet haben)
19 cancel(TX,A) :-
20   participant(TX,A), invalid(TX,A),
21   not(completed(TX,A)).
22
23 % Completion:
24 % ----------------------------------------
25 % Wenn ein Teilnehmer nicht in der Cancel-Menge ist, nicht ausgetreten ist und
26 % seine Arbeit beenden kann, dann soll er eine Complete-Nachricht bekommen, wenn
27 % die globale Entscheidung auf Completion getroffen wird.
28 completion(TX,A) :-
29   participant(TX,A), not(cancel(TX,A)),
30   not(exit(TX,A)), not(cannotComplete(TX,A)),
31   not(invalid(TX,A)).
```

Quellcode A.16: Entscheidung über den Ausgang einer Transaktion

```
1 % Wenn alle vitalen Teilnehmer in der Menge der Completion-Teilnehmer enthalten
2 % sind, dann kann die Transaktion abschliessen.
3 finishing(TX) :-
4   findall(X,vital(TX,X),V), list_to_ord_set(V, VS),
5   findall(P,completion(TX,P),C), list_to_ord_set(C, CS),
6   subset(VS,CS).
```

Quellcode A.17: Close- und Compensate-Mengen

```
1 % Close:
2 % ----------------------------------------
3 % Ein Knoten darf seine Arbeit nur beenden, wenn
4 %  a) die Transaktion erfolgreich beenden kann und
5 %  b) wenn er im Completion-Set enthalten ist.
6 % Der finish(TX)-Fakt wird gesetzt, wenn finishing(TX) wahr
7 % wird, um den Aufwand der Neuberechnung zu minimieren.
8 close(TX,A) :- finish(TX), completed(TX,A), completion(TX,A).
9
10 % Compensate:
11 % ----------------------------------------
12 % Ein Knoten muss seine Arbeit kompensieren, wenn er nicht Completion-Set ist und
13 % seine Arbeit bereits beendet hat. Dies kann bei erfolgreich beendeten Trans-
14 % aktionen bei BAPC im Rahmen eines Choice-Sets passieren oder bei Kompensationen.
15 compensate(TX,A) :- finish(TX), completed(TX,A), not(completion(TX,A)).
16 compensate(TX,A) :- not(finish(TX)), completed(TX,A).
```

Veröffentlichungen

Folgende Veröffentlichungen sind aus den Ergebnissen im Umfeld dieses Dissertationsprojektes hervorgegangen:

VON RIEGEN, MICHAEL; HUSEMANN, MARTIN; FINK, STEFAN; RITTER, NORBERT: *Rule-based Coordination of Distributed Web Service Transactions.* In: IEEE Transactions on Services Computing Nr. 1, Jg. 3, S. 60-72, 2010

VON RIEGEN, MICHAEL; RITTER, NORBERT: *Reliable Monitoring for Runtime Validation of Choreographies.* In: MARK PERRY und HIDEYASU SASAKI und MATTHIAS EHRMANN und GUADALUPE ORTIZ BELLOT und OANA DINI (Hrsg.), *Proceedings of the Fourth International Conference on Internet and Web Applications and Services (ICIW 2009)*, IEEE Computer Society Press, ISBN: 978-0-7695-3613-2, S. 310-315, Mai 2009

VON RIEGEN, MICHAEL; HUSEMANN, MARTIN; RITTER, NORBERT: *Providing Decision Capabilities to Coordinators in Distributed Processes.* In: ABDELHAMID MELLOUK und JUN BI und GUADALUPE ORTIZ und DICKSON K. W. CHIU und MANUELA POPESCU (Hrsg.), *Proceedings of the Third International Conference on Internet and Web Applications and Services (ICIW 2008)*, IEEE Computer Society Press, ISBN: 978-0-7695-3163-2, S. 500-505, Juni 2008

HUSEMANN, MARTIN; VON RIEGEN, MICHAEL; RITTER, NORBERT: *Transactional Coordination of Dynamic Processes in Service-Oriented Environments.* In: *2007 IEEE International Conference on Web Services (ICWS 2007)*, IEEE Computer Society Press, ISBN: 0-7695-2924-0, S. 1024-1031, Juli 2007

DECKER, GERO; VON RIEGEN, MICHAEL: *Scenarios and Techniques for Choreography Design.* In: WITOLD ABRAMOWICZ (Hrsg.), *Proceedings of the 10th International Conference on Business Information Systems (BIS 2007)*, Lecture Notes in Computer Science Nr. 4439, Springer-Verlag, ISBN: 978-3-540-72034-8, S. 121-132, April 2007

VON RIEGEN, MICHAEL; ZAPLATA, SONJA: *Supervising Remote Task Execution in Collaborative Workflow Environments.* In: TORSTEN BRAUN und GEORG CARLE und BURKHARD STILLER (Hrsg.), *Konferenzband zur KiVS 2007 für Industrie-, Kurz- und Workshopbeiträge*, VDE Verlag, ISBN: 978-3-8007-2980-7, S. 337-358, März 2007

HUSEMANN, MARTIN, VON RIEGEN, MICHAEL; RITTER, NORBERT: *Transaktionale Kontrolle dynamischer Prozesse in serviceorientierten Umgebungen.* In: Datenbank-Spektrum Nr. 20, S. 6-14, ISSN: 1618-2162, dpunkt.verlag, Februar 2007

Literaturverzeichnis

[AAA+07] ALVES, Alexandre ; ARKIN, Assaf ; ASKARY, Sid ; BARRETO, Charlton ; BLOCH, Ben ; CURBERA, Francisco ; FORD, Mark ; GOLAND, Yaron ; GUÍZAR, Alejandro ; KARTHA, Neelakantan ; LIU, Canyang K. ; KHALAF, Rania ; KÖNIG, Dieter ; MARIN, Mike ; MEHTA, Vinkesh ; THATTE, Satish ; RIJN, Danny van d. ; YENDLURI, Prasad ; YIU, Alex: Web Services Business Process Execution Language Version 2.0 / Organization for the Advancement of Structured Information Systems. 2007 (April). – OASIS Standard. – `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`

[AACP04] AIELLO, Marco (Hrsg.) ; AOYAMA, Mikio (Hrsg.) ; CURBERA, Francisco (Hrsg.) ; PAPAZOGLOU, Mike P. (Hrsg.): *Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, NY, USA, November 15-19, 2004, Proceedings*. ACM Press, New York, NY, USA, 2004 . – ISBN 1–58113–871–7

[AAF+02] ARKIN, Assaf ; ASKARY, Sid ; FORDIN, Scott ; JEKELI, Wolfgang ; KAWAGUCHI, Kohsuke ; ORCHARD, David ; POGLIANI, Stefano ; RIEMER, Karsten ; STRUBLE, Susan ; TAKACSI-NAGY, Pal ; TRICKOVIC, Ivana ; ZIMEK, Sinisa: Web Service Choreography Interface (WSCI) 1.0 / W3C. 2002 (August). – Note. – `http://www.w3.org/TR/2002/NOTE-wsci-20020808`

[ABCC05] AALST, Wil M. P. d. (Hrsg.) ; BENATALLAH, Boualem (Hrsg.) ; CASATI, Fabio (Hrsg.) ; CURBERA, Francisco (Hrsg.): *Business Process Management, 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005, Proceedings*. Bd. *3649*. Springer-Verlag, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–28238–6

[ABDN07] ALRIFAI, Mohammad ; BALKE, Wolf-Tilo ; DOLOG, Peter ; NEJDL, Wolfgang: Nonblocking Scheduling for Web Service Transactions. In: ZIMMERMANN, Wolf (Hrsg.) ; KÖNIG-RIES, Birgitta (Hrsg.) ; PAHL, Claus (Hrsg.): *Fifth IEEE European Conference on Web Services (ECOWS 2007), 26-28 November, Halle (Saale), Germany*, IEEE Computer Society Press, 2007. – ISBN 0–7695–3044–3, S. 213–222

[ACKM04] ALONSO, Gustavo ; CASATI, Fabio ; KUNO, Harumi ; MACHIRAJU, Vijay: *Web*

Services: Concepts, Architecture and Applications. Springer-Verlag, 2004. – ISBN 3–540–44008–9

[ADO+06] AALST, Wil van d. ; DUMAS, Marlon ; OUYANG, C. ; ROZINAT, Anne ; VERBEEK, H. M. W.: Choreography Conformance Checking: An Approach based on BPEL and Petri Nets. In: LEYMANN, Frank (Hrsg.) ; REISIG, Wolfgang (Hrsg.) ; THATTE, Satish R. (Hrsg.) ; AALST, Wil van d. (Hrsg.): *The Role of Business Processes in Service Oriented Architectures* Bd. 06291, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006 (Dagstuhl Seminar Proceedings). – ISSN 1862–4405, S. 1–71

[ADO+08] AALST, Wil M. P. d. ; DUMAS, Marlon ; OUYANG, Chun ; ROZINAT, Anne ; VERBEEK, Eric: Conformance checking of service behavior. In: *ACM Transactions on Internet Technology* 8 (2008), Nr. 3

[AFG+06] ARDISSONO, Liliana ; FURNARI, Roberto ; GOY, Anna ; PETRONE, Giovanna ; SEGNAN, Marino: Fault Tolerant Web Service Orchestration by Means of Diagnosis. In: GRUHN, Volker (Hrsg.) ; OQUENDO, Flávio (Hrsg.): *Software Architecture, Third European Workshop, EWSA 2006, Nantes, France, September 4-5, Revised Selected Papers* Bd. 4344, Springer-Verlag, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–69271–1, S. 2–16

[AH90] ALUR, Rajeev ; HENZINGER, Thomas A.: Real-time logics: complexity and expressiveness. In: *Proceedings of the fifth annual IEEE Symposium on Logic in Computer Science, Philadelphia, PA , USA, 4-7 Juni 1990*, IEEE Computer Society Press, Juni 1990, S. 390–401

[AH05] AALST, Wil M. P. d. ; HOFSTEDE, Arthur H. M.: YAWL: yet another workflow language. In: *Information Systems* 30 (2005), Nr. 4, S. 245–275

[AHKB03] AALST, Wil M. P. d. ; HOFSTEDE, Arthur H. M. ; KIEPUSZEWSKI, Bartek ; BARROS, Alistair P.: Workflow Patterns. In: *Distributed and Parallel Databases* 14 (2003), Nr. 1, S. 5–51. – ISSN 0926–8782

[AP06] AALST, Wil M. P. d. ; PESIC, Maja: DecSerFlow: Towards a Truly Declarative Service Flow Language. In: BRAVETTI, Mario (Hrsg.) ; NÚÑEZ, Manuel (Hrsg.) ; ZAVATTARO, Gianluigi (Hrsg.): *Web Services and Formal Methods, Third International Workshop, WS-FM 2006, Vienna, Austria, September 8-9, Proceedings* Bd. 4184, Springer-Verlag, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–38862–1, S. 1–23

[Apa06] APACHE SOFTWARE FOUNDATION (Hrsg.): *Apache Kandula2*. 2. 1901 Munsey Drive, Forest Hill, MD 21050-2747, USA: Apache Software Foundation, 2006. `http://ws.apache.org/kandula/2`

[Apa09] APACHE SOFTWARE FOUNDATION (Hrsg.): *Apache Axis2 Version 1.5.1*. 1.5.1. 1901 Munsey Drive, Forest Hill, MD 21050-2747, USA: Apache Software Foundation, 2009. http://ws.apache.org/axis2/1_5_1/userguide.html

[AU02] AHMED, Khawar Z. ; UMRYSH, Cary E.: *Developing enterprise Java applications with J2EE and UML*. Addison-Wesley Professional, 2002. – ISBN 978–0–201–73829–2

[BBM+05] BALDONI, Matteo ; BAROGLIO, Cristina ; MARTELLI, Alberto ; PATTI, Viviana ; SCHIFANELLA, Claudio: Verifying the Conformance of Web Services to Global Interaction Protocols: A First Step. In: BRAVETTI, Mario (Hrsg.) ; KLOUL, Leïla (Hrsg.) ; ZAVATTARO, Gianluigi (Hrsg.): *Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop, EPEW 2005 and International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, September 1-3, Proceedings* Bd. 3670, Springer-Verlag, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–28701–9, S. 257–271

[BBMP06] BALDONI, Matteo ; BAROGLIO, Cristina ; MARTELLI, Alberto ; PATTI, Viviana: A Priori Conformance Verification for Guaranteeing Interoperability in Open Environments. In: DAN, Asit (Hrsg.) ; LAMERSDORF, Winfried (Hrsg.): *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, Proceedings* Bd. 4294, Springer-Verlag, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–68147–7, S. 339–351

[BCH+03a] BUNTING, Doug ; CHAPMAN, Martin ; HURLEY, Oisin u. a. ; SUN MICROSYSTEMS (Hrsg.): *Web Services Composite Application Framework (WS-CAF)*. Version 1.0. 901 San Antonio Road, Palo Alto, CA 94303: Sun Microsystems, Juli 2003. http://developers.sun.com/techtopics/webservices/wscaf/primer.pdf

[BCH+03b] BUNTING, Doug ; CHAPMAN, Martin ; HURLEY, Oisin ; LITTLE, Mark ; MISCHKINSKY, Jeff ; NEWCOMER, Eric ; WEBBER, Jim ; SWENSON, Keith ; SUN MICROSYSTEMS (Hrsg.): *Web Services Context (WS-Context)*. Version 1.0. 901 San Antonio Road, Palo Alto, CA 94303: Sun Microsystems, Juli 2003. http://developers.sun.com/techtopics/webservices/wscaf/wsctx.pdf

[BCH+03c] BUNTING, Doug ; CHAPMAN, Martin ; HURLEY, Oisin ; LITTLE, Mark ; MISCHKINSKY, Jeff ; NEWCOMER, Eric ; WEBBER, Jim ; SWENSON, Keith ; SUN MICROSYSTEMS (Hrsg.): *Web Services Coordination Framework (WS-CF)*. Version 1.0. 901 San Antonio Road, Palo Alto, CA 94303: Sun Microsystems, Juli 2003. http://developers.sun.com/techtopics/webservices/wscaf/wscf.pdf

[BCH+03d] BUNTING, Doug ; CHAPMAN, Martin ; HURLEY, Oisin ; LITTLE, Mark ; MISCHKINSKY, Jeff ; NEWCOMER, Eric ; WEBBER, Jim ; SWENSON, Keith ; SUN MI-

CROSYSTEMS (Hrsg.): *Web Services Transaction Management (WS-TXM)*. Version 1.0. 901 San Antonio Road, Palo Alto, CA 94303: Sun Microsystems, Juli 2003. http://developers.sun.com/techtopics/webservices/wscaf/wstxm.pdf

[BCT05] BENATALLAH, Boualem (Hrsg.) ; CASATI, Fabio (Hrsg.) ; TRAVERSO, Paolo (Hrsg.): *Service-Oriented Computing - ICSOC 2005, Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005, Proceedings*. Bd. *3826*. Springer-Verlag, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–30817–2

[BDH05] BARROS, Alistair P. ; DUMAS, Marlon ; HOFSTEDE, Arthur H. M.: Service Interaction Patterns. In: [ABCC05], S. 302–318

[BDO05a] BARROS, Alistair ; DUMAS, Marlon ; OAKS, Phillipa: A Critical Overview of the Web Services Choreography Description Language (WS-CDL) / bptrends.com. 2005 (März). – White Paper. – http://www.bptrends.com/publicationfiles/03-05%20WP%20WS-CDL%20Barros%20et%20al.pdf

[BDO05b] BARROS, Alistair P. ; DUMAS, Marlon ; OAKS, Phillipa: Standards for Web Service Choreography and Orchestration: Status and Perspectives. In: [ABCC05], S. 61–74

[Bel02] BELLWOOD, Tom (Hrsg.). ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION SYSTEMS: UDDI Version 2.04 API Specification / Organization for the Advancement of Structured Information Systems. 2002 (Juli). – OASIS Standard. – http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.pdf

[BG05] BARESI, Luciano ; GUINEA, Sam: Dynamo: Dynamic Monitoring of WS-BPEL Processes. In: [BCT05], S. 478–483

[BGG04] BARESI, Luciano ; GHEZZI, Carlo ; GUINEA, Sam: Smart monitors for composed services. In: [AACP04], S. 193–202

[BLF+05] BURDETT, David ; LAFON, Yves ; FLETCHER, Tony ; RITZINGER, Greg ; BARRETO, Charlton ; KAVANTZAS, Nickolas: Web Services Choreography Description Language Version 1.0 / W3C. 2005 (November). – Candidate Recommendation. – http://www.w3.org/TR/ws-cdl-10/

[BLFM05] BERNERS-LEE, T. ; FIELDING, R. ; MASINTER, L.: RFC 3986: Uniform Resource Identifier (URI): Generic Syntax / Network Working Group. 2005 (Januar). – Standards Track. – http://www.ietf.org/rfc/rfc3986.txt

[BN96] BERNSTEIN, Philip A. ; NEWCOMER, Eric: *Principles of Transaction Processing for Systems Professionals.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. – ISBN 1–55860–415–4

[Bou07] BOUJRAF, Abdelkrim: *R4eGov Deliverable WP3 - D2: Case Study 2: The European Judical Network / Europol (EU FP6, IST-2004-026650).* online, 2007. – `http://www.r4egov.eu/resources/details.php?Id_taxonomy=6`

[CBS+07] CHAN, K. S. M. ; BISHOP, Judith ; STEYN, Johan ; BARESI, Luciano ; GUINEA, Sam: A Fault Taxonomy for Web Service Composition. In: NITTO, Elisabetta D. (Hrsg.) ; RIPEANU, Matei (Hrsg.): *Service-Oriented Computing - ICSOC 2007 Workshops, International Workshops, Vienna, Austria, September 17, Revised Selected Papers* Bd. 4907, Springer-Verlag, 2007 (Lecture Notes in Computer Science). – ISBN 978–3–540–93850–7, S. 363–375

[CDF+02] CEPONKUS, Alex ; DALAL, Sanjay ; FLETCHER, Tony ; FURNISS, Peter ; GREEN, Alastair ; POPE, Bill: Business Transaction Protocol (BTP) Version 1.0 / Organization for the Advancement of Structured Information Systems. 2002 (Juni). – Committee Specification. – `http://www.oasis-open.org/committees/download.php/1184/2002-06-03.BTP_cttee_spec_1.0.pdf`

[Cer02] CERAMI, Ethan: *Web Services Essentials*. O'Reilly & Associates, Sebastopol, CA, USA, 2002. – ISBN 0–596–00224–6

[Cha04] CHAPPELL, David A.: *Enterprise Service Bus*. O'Reilly & Associates, Sebastopol, CA, USA, 2004. – ISBN 978–0–596–00675–4

[CL06] CHAPPELL, Dave ; LIU, Lily: Web Services Brokered Notification 1.3 (WS-BrokeredNotification) / Organization for the Advancement of Structured Information Systems. 2006 (Oktober). – OASIS Standard. – `http://docs.oasis-open.org/wsn/wsn-ws_brokered_notification-1.3-spec-os.pdf`

[Cla87] CLARK, Keith L.: Negation as failure. In: GINSBERG, Matthew L. (Hrsg.): *Readings in nonmonotonic reasoning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987, S. 311–325

[CM94] CHITTARO, Luca ; MONTANARI, Angelo: Efficient Handling of Context Dependency in the Cached Event Calculus. In: *Proc. of TIME'94 - International Workshop on Temporal Representation and Reasoning, Pensacola Beach, Florida, USA, 4. Mai*, 1994, S. 103–112

[CM96] CHITTARO, Luca ; MONTANARI, Angelo: Efficient Temporal Reasoning in the Cached Event Calculus. In: *Computational Intelligence* 12 (1996), S. 359–382

[CM04] COLIN, Séverine ; MARIANI, Leonardo: Run-Time Verification. In: BROY, Manfred (Hrsg.) ; JONSSON, Bengt (Hrsg.) ; KATOEN, Joost-Pieter (Hrsg.) ; LEUCKER, Martin (Hrsg.) ; PRETSCHNER, Alexander (Hrsg.): *Model-Based Testing of Reactive Systems* Bd. 3472, Springer-Verlag, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–26278–4, S. 525–555

[CMS02] CHIASSON, Theodore ; MCALLISTER, Michael ; SLONIM, Jacob: Distributed Transaction Management in a Peer-to-Peer Process-Oriented Environment. In: PLAICE, John (Hrsg.) ; KROPF, Peter G. (Hrsg.) ; SCHULTHESS, Peter (Hrsg.) ; SLONIM, Jacob (Hrsg.): *Distributed Communities on the Web, 4th International Workshop, DCW 2002, Sydney, Australia, April 3-5, Revised Papers* Bd. 2468, Springer-Verlag, 2002 (Lecture Notes in Computer Science). – ISBN 3–540–00301–0, S. 182–192

[CWMR07] CHINNICI, Roberto ; WEERAWARANA, Sanjiva ; MOREAU, Jean-Jacques ; RYMAN, Arthur: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language / W3C. 2007 (Juni). – W3C Recommendation. – `http://www.w3.org/TR/2007/REC-wsdl20-20070626`

[CYM+06] CARBONE, Marco ; YOSHIDA, Kohei Honda N. ; MILNER, Robin ; BROWN, Gary ; ROSS-TALBOT, Steve: *A Theoretical Basis of Communication-Centred Concurrent Programming*. online, 2006. – To be published by W3C. Available at `http://www.dcs.qmul.ac.uk/~carbonem/cdlpaper/workingnote.pdf`

[DAM06] DUBRAY, Jean-Jacques ; AMAND, Sally S. ; MARTIN, Monica J.: ebXML Business Process Specification Schema Technical Specification v2.0.4 / Organization for the Advancement of Structured Information Systems. 2006 (Dezember). – OASIS Standard. – `http://docs.oasis-open.org/ebxml-bp/2.0.4/OS/spec/ebxmlbp-v2.0.4-Spec-os-en.pdf`

[DD04] DIJKMAN, Remco M. ; DUMAS, Marlon: Service-Oriented Design: A Multi-Viewpoint Approach. In: *International Journal of Cooperative Information Systems* 13 (2004), Nr. 4, S. 337–368

[Dec06] DECKER, Gero: *Process Choreographies in Service-oriented Environments*, Hasso Plattner Institut - Universität Potsdam, Diplomarbeit, 2006

[Die05] DIETRICH, Jens: *The Mandarax Manual*. 3.0. Institute of Information Sciences and Technology, Massey University, Palmerston North, New Zealand, 2005. `http://mandarax.sourceforge.net/docs/mandarax.pdf`

[DKLW07] DECKER, Gero ; KOPP, Oliver ; LEYMANN, Frank ; WESKE, Mathias: BPEL4Chor: Extending BPEL for Modeling Choreographies. In: [ICWS007], S. 296–303

[DKS07] DAVIES, Jeff ; KRISHNA, Ashish ; SCHOROW, David: *The Definitive Guide to SOA: BEA AquaLogic Service Bus*. Apress, Inc., 233 Spring Street, New York, NY 10013, USA, 2007. – ISBN 978–1590597972

[DLF93] DARDENNE, Anne ; LAMSWEERDE, Axel van ; FICKAS, Stephen: Goal-Directed Requirements Acquisition. In: *Science of Computer Programming* 20 (1993), Nr. 1-2, S. 3–50

[DOZ06] DECKER, Gero ; OVERDICK, Hagen ; ZAHA, Johannes M.: On the Suitability of WS-CDL for Choreography Modeling. In: WESKE, Mathias (Hrsg.) ; NÜTTGENS, Markus (Hrsg.): *EMISA* Bd. 95, Gesellschaft für Informatik, 2006 (Lecture Notes in Informatics). – ISBN 978–3–88579–189–8, S. 21–33

[DPW06] DECKER, Gero ; PUHLMANN, Frank ; WESKE, Mathias: Formalizing Service Interactions. In: DUSTDAR, Schahram (Hrsg.) ; FIADEIRO, José Luiz (Hrsg.) ; SHETH, Amit P. (Hrsg.): *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, Proceedings* Bd. 4102, Springer-Verlag, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–38901–6, S. 414–419

[DR07] DECKER, Gero ; RIEGEN, Michael V.: Scenarios and Techniques for Choreography Design. In: ABRAMOWICZ, Witold (Hrsg.): *Business Information Systems, 10th International Conference, BIS 2007, Poznan, Poland, April 25-27, Proceedings* Bd. 4439, Springer-Verlag, 2007 (Lecture Notes in Computer Science). – ISBN 978–3–540–72034–8, S. 121–132

[DS06] DINGWALL-SMITH, Andrew R.: *Run-Time Monitoring of Goal-Oriented Requirements Specifications*, Department of Computer Science, University College London, University of London, Diss., 2006

[EHHZ07] ERVEN, Hannes ; HICKER, Georg ; HUEMER, Christian ; ZAPLETAL, Marco: The Web Services-BusinessActivity-Initiator (WS-BA-I) Protocol: an Extension to the Web Services-BusinessActivity Specification. In: [ICWS007], S. 216–224

[ELLR90] ELMAGARMID, Ahmed K. ; LEU, Yungho ; LITWIN, Witold ; RUSINKIEWICZ, Marek: A Multidatabase Transaction Model for InterBase. In: MCLEOD, Dennis (Hrsg.) ; SACKS-DAVIS, Ron (Hrsg.) ; SCHEK, Hans-Jörg (Hrsg.): *16th International Conference on Very Large Data Bases, August 13-16, Brisbane, Queensland, Australia, Proceedings*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990. – ISBN 1–55860–149–X, S. 507–518

[Elm92] ELMAGARMID, Ahmed K. (Hrsg.): *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992. – ISBN 1–55860–214–3

[FDF+04] FURNISS, Peter ; DALAL, Sanjay ; FLETCHER, Tony ; GREEN, Alastair ; HAUGEN, Bob ; CEPONKUS, Alex ; POPE, Bill: Business Transaction Protocol Version 1.1.0 / Organization for the Advancement of Structured Information Systems. 2004 (April). – OASIS Committee Draft. – http://www.oasis-open.org/committees/download.php/12415/business_transaction-btp-1.1-cd-01.zip

[FG04] FURNISS, Peter ; GREEN, Alastair: Choreology Ltd. Feedback to the authors of WS-Coordination, WS-AtomicTransaction and WS-BusinessActivity / Choreology Ltd. 2004 (April). – Forschungsbericht. – www.oasis-open.org/committees/download.php/15808

[FGM+99] FIELDING, R. ; GETTYS, J. ; MOGUL, J. ; FRYSTYK, H. ; MASINTER, L. ; LEACH, P. ; BERNERS-LEE, T.: RFC 2616: Hypertext transfer protocol – HTTP/1.1 / Network Working Group. 1999 (Juni). – Standards Track. – http://www.ietf.org/rfc/rfc2616.txt

[FJ09] FEINGOLD, Max ; JEYARAMAN, Ram: Web Services Coordination (WS-Coordination) Version 1.2 / Organization for the Advancement of Structured Information Systems. 2009 (Februar). – OASIS Standard. – http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os.pdf

[FL09] FREUND, Tom ; LITTLE, Mark: Web Services Business Activity (WS-BusinessActivity) Version 1.2 / Organization for the Advancement of Structured Information Systems. 2009 (Februar). – OASIS Standard. – http://docs.oasis-open.org/ws-tx/wstx-wsba-1.2-spec-os.pdf

[FPD+09] FREMANTLE, Paul ; PATIL, Sanjay ; DAVIS, Doug ; KARMARKAR, Anish ; PILZ, Gilbert ; WINKLER, Steve ; YALÇINALP Ümit: Web Services Reliable Messaging (WS-ReliableMessaging) Version 1.2 / Organization for the Advancement of Structured Information Systems. 2009 (Februar). – OASIS Standard. – http://docs.oasis-open.org/ws-rx/wsrm/v1.2/wsrm.pdf

[FRT06] FLETCHER, Tony ; ROSS-TALBOT, Steve: Web Services Choreography Description Language: Primer / W3C. 2006 (Juni). – W3C Working Draft. – http://www.w3.org/TR/2006/WD-ws-cdl-10-primer-20060619/

[FSTC04] FARRELL, Andrew D H. ; SERGOT, Marek J. ; TRASTOUR, David ; CHRISTODOULOU, Athena: Performance Monitoring of Service-Level Agreements for Utility Computing Using the Event Calculus. In: *First IEEE International Workshop on Electronic Contracting (WEC'04), July 06, San Diego, California, USA*, IEEE Computer Society Press, 2004. – ISBN 0–7695–2184–3, S. 17–24

[FUMK03] FOSTER, Howard ; UCHITEL, Sebastián ; MAGEE, Jeff ; KRAMER, Jeff: Model-based Verification of Web Service Compositions. In: *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October, Montreal, Canada*, IEEE Computer Society Press, 2003. – ISBN 0–7695–2035–9, S. 152–163

[FUMK04] FOSTER, Howard ; UCHITEL, Sebastián ; MAGEE, Jeff ; KRAMER, Jeff: Compatibility Verification for Web Service Choreography. In: *Proceedings of the IEEE International Conference on Web Services (ICWS'04), June 6-9, San Diego, California, USA*, IEEE Computer Society Press, 2004. – ISBN 0–7695–2167–3, S. 738–741

[GCN+07] GAN, Yuan ; CHECHIK, Marsha ; NEJATI, Shiva ; BENNETT, Jon ; O'FARRELL, Bill ; WATERHOUSE, Julie: Runtime monitoring of web service conversations. In: LYONS, Kelly A. (Hrsg.) ; COUTURIER, Christian (Hrsg.): *Proceedings of the 2007 conference of the Centre for Advanced Studies on Collaborative Research, October 22-25, Richmond Hill, Ontario, Canada*, IBM, 2007, S. 42–57

[Ger01] GERGELEIT, Martin: *A Monitoring-based Approach to Object-Oriented Real-Time Computing*, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, Diss., 2001

[GHJV96] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 1996. – ISBN 3–8273–1862–9

[GHM06] GRAHAM, Steve ; HULL, David ; MURRAY, Bryan: Web Services Base Notification 1.3 (WS-BaseNotification) / Organization for the Advancement of Structured Information Systems. 2006 (Oktober). – OASIS Standard. – `http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf`

[GHM+07] GUDGIN, Martin ; HADLEY, Marc ; MENDELSOHN, Noah ; LAFON, Yves ; MOREAU, Jean-Jacques ; KARMARKAR, Anish ; NIELSEN, Henrik F.: SOAP Version 1.2 Part 1: Messaging Framework (Second Edition) / W3C. 2007 (April). – W3C Recommendation. – `http://www.w3.org/TR/2007/REC-soap12-part1-20070427/`

[GMGK+90] GARCIA-MOLINA, Hector ; GAWLICK, Dieter ; KLEIN, Johannes ; KLEISSNER, Karl ; SALEM, Kenneth: Coordinating Multi-Transaction Activities / Department of Computer Science, Princeton University. 1990 (CS-TR-297-90). – Forschungsbericht. – `http://ilpubs.stanford.edu:8090/1/1/1990-1.pdf`

[GMS87] GARCIA-MOLINA, Hector ; SALEM, Kenneth: Sagas. In: DAYAL, Umeshwar (Hrsg.) ; TRAIGER, Irving L. (Hrsg.): *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference,*

San Francisco, California, May 27-29, ACM Press, New York, NY, USA, 1987, S. 249–259

[GP09] GEORGAKOPOULOS, Dimitrios ; PAPAZOGLOU, Michael: *Service-oriented computing*. MIT Press, 2009. – ISBN 0–262–07296–3

[GR93] GRAY, Jim ; REUTER, Andreas: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. – ISBN 1–55860–190–2

[Gra81] GRAY, Jim: The Transaction Concept: Virtues and Limitations. In: *Very Large Data Bases, 7th International Conference, September 9-11, Cannes, France, Proceedings*, IEEE Computer Society Press, 1981, S. 144–154

[Gui05] GUINEA, Sam: Self-healing web service compositions. In: ROMAN, Gruia-Catalin (Hrsg.) ; GRISWOLD, William G. (Hrsg.) ; NUSEIBEH, Bashar (Hrsg.): *27th International Conference on Software Engineering (ICSE 2005), 15-21 May, St. Louis, Missouri, USA*, ACM Press, New York, NY, USA, 2005, S. 655

[HR01] HÄRDER, Theo ; RAHM, Erhard: *Datenbanksysteme: Konzepte und Techniken der Implementierung, 2. Auflage*. Springer-Verlag, 2001. – ISBN 3–540–42133–5

[HRG06] HADLEY, Marc ; ROGERS, Tony ; GUDGIN, Martin: Web Services Addressing 1.0 - SOAP Binding / W3C. 2006 (Mai). – W3C Recommendation. – `http://www.w3.org/TR/2006/REC-ws-addr-soap-20060509`

[HRR07] HUSEMANN, Martin ; RIEGEN, Michael von ; RITTER, Norbert: Transactional Coordination of Dynamic Processes in Service-Oriented Environments. In: [ICWS007], S. 1024–1031

[ICWS007] IEEE Computer Society Press (Veranst.): *2007 IEEE International Conference on Web Services (ICWS 2007), 9-13 July, 2007, Salt Lake City, Utah, USA*. IEEE Computer Society Press, 2007 . – ISBN 0–7695–2924–0

[Jos08] JOSUTTIS, Nicolai: *SOA in der Praxis - Systemdesign für verteilte Geschäftsprozesse*. Heidelberg, Germany : dpunkt.verlag, 2008. – ISBN 978–3–89864–476–1

[KBV06] KREGER, Heather ; BULLARD, Vaughn ; VAMBENEPE, William: WSDM MOWS - Web Services Distributed Management: Management Using Web Services (MUWS 1.1) / Organization for the Advancement of Structured Information Systems. 2006 (August). – OASIS Standard. – `http://docs.oasis-open.org/wsdm/wsdm-muws1-1.1-spec-os-01.pdf`

[KL04] KOSSMANN, Donald ; LEYMANN, Frank: Web Services. In: *Informatik Spektrum* 27 (2004), Nr. 2, S. 117–128

[KS86] KOWALSKI, Robert ; SERGOT, Marek: A Logic-based Calculus of Events. In: *New Generation Computing* 4 (1986), Nr. 1, S. 67–95. – ISSN 0288–3635

[Kur00] KURSCHL, Werner: *Monitoring von verteilten Systemen*, Institut für Wirtschaftsinformatik (Software Engineering) der Sozial- und Wirtschaftswissenschaftlichen Fakultät der Johannes Kepler Universität, Diss., 2000

[Lam78] LAMPORT, Leslie: Time, Clocks, and the Ordering of Events in a Distributed System. In: *Communications of the ACM* 21 (1978), Nr. 7, S. 558–565

[Lam01] LAMSWEERDE, Axel van: Goal-Oriented Requirements Engineering: A Guided Tour. In: *5th IEEE International Symposium on Requirements Engineering (RE 2001), 27-31 August, Toronto, Canada*, IEEE Computer Society Press, 2001. – ISBN 0–7695–1125–2, S. 249–262

[LAP06] LAZOVIK, Alexander ; AIELLO, Marco ; PAPAZOGLOU, Mike: Planning and monitoring the execution of web service requests. In: *International Journal on Digital Libraries* 6 (2006), Nr. 3, S. 235–246. – ISSN 1432–5012

[LEK98] LYON, J. ; EVANS, K. ; KLEIN, J.: RFC 2371: Transaction Internet Protocol Version 3.0 / Network Working Group. 1998 (Juli). – Standards Track. – `http://www.ietf.org/rfc/rfc2371.txt`

[Ley05] LEYMANN, Frank: The (Service) Bus: Services Penetrate Everyday Life. In: [BCT05], S. 12–20

[LF03] LITTLE, Mark ; FREUND, Thomas: A comparison of Web services transaction protocols (A comparative analysis of WS-C/WS-Tx and OASIS BTP) / IBM. 2003 (Oktober). – White Paper. – `http://www.ibm.com/developerworks/webservices/library/ws-comproto`

[LJH06] LI, Zheng ; JIN, Yan ; HAN, Jun: A Runtime Monitoring and Validation Framework for Web Service Interactions. In: *17th Australian Software Engineering Conference (ASWEC 2006), 18-21 April, Sydney, Australia*, IEEE Computer Society Press, 2006. – ISBN 0–7695–2551–2, S. 70–79

[LL02] LAMSWEERDE, Axel van ; LETIER, Emmanuel: From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering. In: WIRSING, Martin (Hrsg.) ; KNAPP, Alexander (Hrsg.) ; BALSAMO, Simonetta (Hrsg.): *9th International Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF 2002), 7-11 Ocktober 2002, Venice, Italy* Bd. 2941, Springer-Verlag, 2002 (Lecture Notes in Computer Science). – ISBN 3–540–21179–9, S. 153–166

[LNP06] LITTLE, Mark ; NEWCOMER, Eric ; PAVLIK, Greg: Web Services Business Process Specification (WS-BP) Version 1.0 / Organization for the Advancement of Structured Information Systems. 2006 (August). – Committee Draft. – http://www.oasis-open.org/committees/download.php/19475/WS-BP.zip

[LP05] LEYMANN, Frank ; POTTINGER, Stefan: Rethinking the Coordination Models of WS-Coordination and WS-CF. In: *Third European Conference on Web Services (ECOWS 2005), 14-16 November, Växjö, Sweden*, IEEE Computer Society Press, 2005. – ISBN 0–7695–2484–2, S. 160–169

[LW09] LITTLE, Mark ; WILKINSON, Andrew: Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.2 / Organization for the Advancement of Structured Information Systems. 2009 (Februar). – OASIS Standard. – http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-os.pdf

[LZ04] LIMTHANMAPHON, Benchaphon ; ZHANG, Yanchun: Web Service Composition Transaction Management. In: SCHEWE, Klaus-Dieter (Hrsg.) ; WILLIAMS, Hugh E. (Hrsg.): *Database Technologies 2004, Proceedings of the Fifteenth Australasian Database Conference, ADC 2004, Dunedin, New Zealand, 18-22 January 2004* Bd. 27, Australian Computer Society, 2004 (CRPIT). – ISBN 1–920682–06–6, S. 171–179

[Mag07] MAGDIC, Tihomir: *Dienstkomposition mit Hilfe semantischer Service-Templates*, Universität Hamburg, Fachbereich Informatik, Verteilte Systeme und Informationssysteme, Diplomarbeit, 2007

[MBF+04] MCCABE, Francis ; BOOTH, David ; FERRIS, Christopher ; ORCHARD, David ; CHAMPION, Mike ; NEWCOMER, Eric ; HAAS, Hugo: Web Services Architecture / W3C. 2004 (Februar). – W3C Note. – http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

[McC02] MCCOY, David W.: Business Activity Monitoring: Calm Before the Storm / Gartner. 2002 (LE-15-9727). – Research Report. – http://www.gartner.com/resources/105500/105562/105562.pdf

[MDM+10] MILLS, D. ; DELAWARE, U. ; MARTIN, J. ; BURBANK, J. ; KASCH, W.: RFC 5905: Network Time Protocol Version 4: Protocol and Algorithms Specification / Network Working Group. 2010 (Juni). – Standards Track. – http://www.ietf.org/rfc/rfc5905.txt

[Med10] MEDER, Nils: *Atomares Interaktionslogging über einen Enterprise-Service-Bus*, Universität Hamburg, Fachbereich Informatik, Verteilte Systeme und Informationssysteme, Bachelorarbeit, 2010

[MET+10] MELZER, Ingo ; EBERHARD, Sebstian ; THILE, Alexander H. ; FELMMING, Marcus ; TRÖGER, Peter ; RUDOLPH, Barabara ; STUMM, Boris ; LIPP, Matthias ; SAUTER, Patrick ; VAJDA, Jochen ; DOSTAL, Wolfgang ; JECKLE, Mario: *Service-orientierte Architekturen mit Web Services. Konzepte - Standards - Praxis*. Heidelberg, Germany : Spektrum-Akademischer Verlag, 2010. – ISBN 978–3–8274–2549–2

[MGH+07] MOREAU, Jean-Jacques ; GUDGIN, Martin ; HADLEY, Marc ; MENDELSOHN, Noah ; LAFON, Yves ; KARMARKAR, Anish ; NIELSEN, Henrik F.: SOAP Version 1.2 Part 2: Adjuncts (Second Edition) / W3C. 2007 (April). – W3C Recommendation. – http://www.w3.org/TR/2007/REC-soap12-part2-20070427/

[Mil99] MILNER, Robin: *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999. – ISBN 978–0521658690

[ML07] MITRA, Nilo ; LAFON, Yves: SOAP Version 1.2 Part 0: Primer (Second Edition) / W3C. 2007 (April). – W3C Note. – http://www.w3.org/TR/2007/REC-soap12-part0-20070427/

[MMCT08] MONTALI, Marco ; MELLO, Paola ; CHESANI, Federico ; TORRONI, Paolo: Verification of choreographies during execution using the Reactive Event Calculus. In: BRUNI, Roberto (Hrsg.) ; WOLF, Karsten (Hrsg.): *Web Services and Formal Methods, 5th International Workshop, WS-FM 2008, Milan, Italy, September 4-5, Proceedings* Bd. 5387, Springer-Verlag, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–642–01363–8, S. 55–72

[Mon10] MONTALLI, Marco: *Specification and Verification of Declarative Open Interaction Models (A Logic-Based Approach)*. Springer-Verlag, 2010. – ISBN 978–3–642–14537–7

[Mos85] MOSS, J. Eliot B. (Hrsg.): *Nested Transactions: An Approach to Reliable Distributed Computing* . The MIT Press, 1985. – ISBN 978–0262132008

[MPW92a] MILNER, Robin ; PARROW, Joachim ; WALKER, David: A Calculus of Mobile Processes, I. In: *Information and Computation* 100 (1992), Nr. 1, S. 1–40

[MPW92b] MILNER, Robin ; PARROW, Joachim ; WALKER, David: A Calculus of Mobile Processes, II. In: *Information and Computation* 100 (1992), Nr. 1, S. 41–77

[MS04] MAHBUB, Khaled ; SPANOUDAKIS, George: A framework for requirents monitoring of service based systems. In: [AACP04], S. 84–93

[MS05] MAHBUB, Khaled ; SPANOUDAKIS, George: Run-time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and

Evaluation Experience. In: *IEEE International Conference on Web Services (ICWS 2005), 11-15 July, Orlando, FL, USA*, IEEE Computer Society Press, 2005. – ISBN 0–7695–2409–5, S. 257–265

[MWDC11] MALHOTRA, Ashok ; WARR, Katy ; DAVIS, Doug ; CHOU, Wu: Web Services Eventing (WS-Eventing) / W3C. 2011 (April). – Candidate Recommendation. – http://www.w3.org/TR/2011/CR-ws-eventing-20110428

[Obj04] OBJECT MANAGEMENT GROUP (Hrsg.): *Common Object Request Broker Architecture: Core Specification*. 3.0.3. 140 Kendrick Street, Building A, Suite 300 Needham, MA 02494 USA: Object Management Group, März 2004. http://www.omg.org/cgi-bin/doc?formal/04-03-12.pdf

[Obj09] OBJECT MANAGEMENT GROUP (Hrsg.): *Business Process Model and Notation (BPMN)*. 1.2. 140 Kendrick Street, Building A, Suite 300 Needham, MA 02494 USA: Object Management Group, Januar 2009. http://www.omg.org/cgi-bin/doc?formal/09-01-03.pdf

[Ope92] THE OPEN GROUP (Hrsg.): *Distributed TP: The XA Specification*. Catalog number S423. 44 Montgomery St., Suite 960, San Francisco CA 94104-4704, USA: The Open Group, November 1992

[Ope94] THE OPEN GROUP (Hrsg.): *Distributed TP: The XA+ Specification, Version 2*. Catalog number C193. 44 Montgomery St., Suite 960, San Francisco CA 94104-4704, USA: The Open Group, Juli 1994

[Ope95] THE OPEN GROUP (Hrsg.): *Distributed TP: The TX (Transaction Demarcation) Specification*. Catalog number U011. 44 Montgomery St., Suite 960, San Francisco CA 94104-4704, USA: The Open Group, November 1995

[Ope96] THE OPEN GROUP (Hrsg.): *Distributed Transaction Processing: Reference Model, Version 3*. Catalog number G504. 44 Montgomery St., Suite 960, San Francisco CA 94104-4704, USA: The Open Group, Februar 1996

[Pap08] PAPAZOGLOU, Michael P. (Hrsg.): *Web Services: Principles and Technology*. Pearson Education Limited, Edinburgh Gate, Harlow, Essex CM20 2JE, England, 2008. – ISBN 978–0–321–15555–9

[Par01] *Kapitel* An Introduction to the π-Calculus. In: PARROW, Joachim: *Handbook of Process Algebra*. Elsevier Science Publishers, Amsterdam, Netherlands, 2001. – ISBN 0–444–82830–3, S. 479–543

[PBB+04] PISTORE, Marco ; BARBON, Fabio ; BERTOLI, Piergiorgio ; SHAPARAU, Dmitry ; TRAVERSO, Paolo: Planning and Monitoring Web Service Composition. In:

BUSSLER, Christoph (Hrsg.) ; FENSEL, Dieter (Hrsg.): *Artificial Intelligence: Methodology, Systems, and Applications, 11th International Conference, AIMSA 2004, Varna, Bulgaria, September 2-4, Proceedings* Bd. 3192, Springer-Verlag, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–22959–0, S. 106–115

[Pel03] PELTZ, Chris: Web Services Orchestration and Choreography. In: *IEEE Computer* 36 (2003), Nr. 10, S. 46–52. – ISSN 0018–9162

[PSE+09] PERRY, Mark (Hrsg.) ; SASAKI, Hideyasu (Hrsg.) ; EHRMANN, Matthias (Hrsg.) ; BELLOT, Guadalupe O. (Hrsg.) ; DINI, Oana (Hrsg.): *The Fourth International Conference on Internet and Web Applications and Services, ICIW 2009, 24-18 May 2009, Venice/Mestre, Italy*. IEEE Computer Society Press, 2009 . – ISBN 978–0–7695–3613–2

[Red06] RED HAT, INC. (Hrsg.): *JBoss Transactions Resources*. 1.0. 1801 Varsity Drive, Raleigh, North Carolina 27606, USA: Red Hat, Inc., 2006. `http://www.jboss.org/jbosstm/resources.html`

[RHFR10] RIEGEN, Michael von ; HUSEMANN, Martin ; FINK, Stefan ; RITTER, Norbert: Rule-Based Coordination of Distributed Web Service Transactions. In: *IEEE Transactions on Services Computing* 3 (2010), Nr. 1, S. 60–72. – ISSN 1939–1374

[RHG06] ROGERS, Tony ; HADLEY, Marc ; GUDGIN, Martin: Web Services Addressing 1.0 - Core / W3C. 2006 (Mai). – W3C Recommendation. – `http://www.w3.org/TR/2006/REC-ws-addr-core-20060509`

[RHR08] RIEGEN, Michael von ; HUSEMANN, Martin ; RITTER, Norbert: Providing Decision Capabilities to Coordinators in Distributed Processes. In: MELLOUK, Abdelhamid (Hrsg.) ; BI, Jun (Hrsg.) ; ORTIZ, Guadalupe (Hrsg.) ; CHIU, Dickson K. W. (Hrsg.) ; POPESCU, Manuela (Hrsg.): *Third International Conference on Internet and Web Applications and Services, ICIW 2008, 8-13 June, Athens, Greece*, IEEE Computer Society Press, 2008. – ISBN 978–0–7695–3163–2, S. 500–505

[RHS05] RICHTER, Jan-Peter ; HALLER, Harald ; SCHREY, Peter: Serviceorientierte Architektur. In: *Informatik Spektrum* 28 (2005), Nr. 5, S. 413–416

[Rie05] RIEGEN, Michael von: *Transaktionale Koordination dynamischer Prozesse in Grid-Umgebungen*, Universität Hamburg, Fachbereich Informatik, Verteilte Systeme und Informationssysteme, Diplomarbeit, 2005

[RLS+09] RIEGEN, Michael von ; LEE, Hannah ; SVIRSKAS, Adomas ; CADOR, Kevin ; KYLAU, Uwe ; GAALOUL, Khaled: *R4eGov Deliverable WP6 - D8: Prototype of advanced security and privacy mechanisms (EU FP6, IST-2004-026650)*. 2009

[Rob06] ROBINSON, William N.: A requirements monitoring framework for enterprise systems. In: *Requirements Engineering* 11 (2006), Nr. 1, S. 17–41. – ISSN 0947–3602

[Rob10] ROBINSON, William: A Roadmap for Comprehensive Requirements Modeling. In: *IEEE Computer* 43 (2010), Nr. 5, S. 64–72. – ISSN 0018–9162

[RR09] RIEGEN, Michael von ; RITTER, Norbert: Reliable Monitoring for Runtime Validation of Choreographies. In: [PSE+09], S. 310–315

[RWR06] RINDERLE, Stefanie ; WOMBACHER, Andreas ; REICHERT, Manfred: On the Controlled Evolution of Process Choreographies. In: LIU, Ling (Hrsg.) ; REUTER, Andreas (Hrsg.) ; WHANG, Kyu-Young (Hrsg.) ; ZHANG, Jianjun (Hrsg.): *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April, Atlanta, GA, USA*, IEEE Computer Society Press, 2006, S. 124–126

[Sch99] SCHWARZ, Kerstin: *Das Konzept der Transaktionshülle zur konsistenten Spezifikation von Abhänigigkeiten in komplexen Anwendungen.*, Universität Magdeburg, Diss., 1999

[Sch09] SCHULZ, Matthias: *Überwachung asynchroner Web-Service-Kommunikation über einen Enterprise Service Bus*, Universität Hamburg, Fachbereich Informatik, Verteilte Systeme und Informationssysteme, Bachelorarbeit, 2009

[SCN+08] SIMMONDS, Jocelyn ; CHECHIK, Marsha ; NEJATI, Shiva ; LITANI, Elena ; O'FARRELL, Bill: Property Patterns for Runtime Monitoring of Web Service Conversations. In: LEUCKER, Martin (Hrsg.): *Runtime Verification, 8th International Workshop, RV 2008, Budapest, Hungary, March 30, Selected Papers* Bd. 5289, 2008. – ISBN 978–3–540–89246–5, S. 137–157

[SGC+09] SIMMONDS, Jocelyn ; GAN, Yuan ; CHECHIK, Marsha ; NEJATI, Shiva ; O'FARRELL, Bill ; LITANI, Elena ; WATERHOUSE, Julie: Runtime Monitoring of Web Service Conversations. In: *IEEE Transactions on Services Computing* 2 (2009), Nr. 3, S. 223–244. – ISSN 1939–1374

[Sha97] SHANAHAN, Murray: *Solving the Frame Problem - A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, Cambridge, MA, USA, 1997. – ISBN 978–0–262–19384–9

[Sha99] SHANAHAN, Murray: The Event Calculus Explained. In: CARBONELL, J. G. (Hrsg.) ; SIEKMANN, J. (Hrsg.): *Artificial Intelligence Today: Recent Trends and Developments*. Springer-Verlag, 1999, S. 409–430

[SIM07] SVIRSKAS, Adomas ; ISACHENKOVA, Jelena ; MOLVA, Refik: Towards secure and trusted collaboration environment for European public sector. In: *3rd International Conference on Collaborative Computing: Networking, Applications and Worksharing, November 12-15, White Plains, USA*, IEEE Computer Society Press, 2007. – ISBN 1–4244–1317–6, S. 49–56

[SIR+09] SVIRSKAS, Adomas ; ISACENKOVA, Jelena ; RIEGEN, Michael von ; LEE, Hannah ; KYLAU, Uwe ; GAALOUL, Khaled: *R4eGov Deliverable WP6 - D7: Architecture of advanced security and privacy mechanisms (EU FP6, IST-2004-026650)*. 2009

[SJH08] SCHROTH, Christoph ; JANNER, Till ; HOYER, Volker: Strategies for Cross-Organizational Service Composition. In: *International MCETECH Conference on e-Technologies (MCETECH 2008), 23-25 January, Montreal, Quebec, Canada*, IEEE Computer Society Press, 2008, S. 93–103

[SM05] SAUTER, Patrick ; MELZER, Ingo: A Comparison of WS-BusinessActivity and BPEL4WS Long-Running Transaction. In: MÜLLER, Paul (Hrsg.) ; GOTZHEIN, Reinhard (Hrsg.) ; SCHMITT, Jens B. (Hrsg.): *Kommunikation in Verteilten Systemen (KiVS), 14. ITG/GI-Fachtagung Kommunikation in Verteilten Systemen (KiVS 2005) Kaiserslautern, 28. Februar - 3. März*, Springer-Verlag, 2005 (Informatik Aktuell). – ISBN 3–540–24473–5, S. 115–125

[SS83] SKEEN, Dale ; STONEBRAKER, Michael: A Formal Model of Crash Recovery in a Distributed System. In: *IEEE Transactions on Software Engineering* 9 (1983), Nr. 3, S. 219–228

[SVTD09] SEGHBROECK, Gregory V. ; VOLCKAERT, Bruno ; TURCK, Filip D. ; DHOEDT, Bart: Automated Instantiation and Extraction of Web Service Choreographies. In: [PSE+09], S. 455–461

[THSS05] TÜRKER, Can ; HALLER, Klaus ; SCHULER, Christoph ; SCHEK, Hans-Jörg: How can we support Grid Transactions? Towards Peer-to-Peer Transaction Processing. In: STONEBRAKER, Michael (Hrsg.) ; WEIKUM, Gerhard (Hrsg.) ; DEWITT, David (Hrsg.): *Second biennial Conference on Innovative Data Systems Research (CIDR 2005), Asilomar, Canada, January 4-7, Proceedings*, Very Large Database Endowment Inc., 2005, S. 174–185

[Thu09] THURLOW, R.: RFC 5531: RPC: Remote Procedure Call Protocol Specification Version 2 / Network Working Group. 2009 (Mai). – Standards Track. – `http://www.ietf.org/rfc/rfc5531.txt`

[Vet06] VETTER, Thorsten: *Anpassung und Implementierung verschiedener Transaktionsprotokolle auf WS-Coordination*, Universität Stuttgart, Diplomarbeit, 2006

[VGN06] VAMBENEPE, William ; GRAHAM, Steve ; NIBLETT, Peter: Web Services Topics 1.3 (WS-Topics) / Organization for the Advancement of Structured Information Systems. 2006 (Oktober). – OASIS Standard. – http://docs.oasis-open.org/wsn/wsn-ws_topics-1.3-spec-os.pdf

[Voi10] VOIGT, Daniel: *Run-Time-Monitoring von Web-Service-Choreographien anhand des Ereigniskalküls*, Universität Hamburg, Fachbereich Informatik, Verteilte Systeme und Informationssysteme, Diplomarbeit, 2010

[VSC06] VILLARREAL, Pablo D. ; SALOMONE, Enrique ; CHIOTTI, Omar: Transforming Collaborative Business Process Models into Web Services Choreography Specifications. In: LEE, Juhnyoung (Hrsg.) ; SHIM, Junho (Hrsg.) ; LEE, Sang goo (Hrsg.) ; BUSSLER, Christoph (Hrsg.) ; SHIM, Simon S. Y. (Hrsg.): *Data Engineering Issues in E-Commerce and Services, Second International Workshop, DEECS 2006, San Francisco, CA, USA, June 26, Proceedings* Bd. 4055, Springer-Verlag, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–35440–9, S. 50–65

[VZG+05] VOGT, Friedrich H. ; ZAMBROVSKI, Simon ; GRUSCHKO, Boris ; FURNISS, Peter ; GREEN, Alastair: Implementing Web Service Protocols in SOA: WS-Coordination and WS-BusinessActivity. In: *7th IEEE International Conference on E-Commerce Technology Workshops (CEC 2005 Workshops), 19 July, München, Germany*, IEEE Computer Society Press, 2005. – ISBN 0–7695–2384–6, S. 21–28

[Wan04] WANG, Farn: Formal verification of timed systems: a survey and perspective. In: *Proceedings of the IEEE* 92 (2004), Nr. 8, S. 1283–1305. – ISSN 0018–9219

[WD08] WS-DIAMOND: *Deliverable D3.1 - Specification of execution mechanisms and composition strategies for self-healing Web services - Phase 1 (EU FP6, IST-516933)*. online, 2008. – http://wsdiamond.di.unito.it/Reports/d3_1.pdf

[Wes07] WESKE, Mathias: *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag, 2007. – ISBN 978–3–540–73521–2

[WKK+10] WETZSTEIN, Branimir ; KARASTOYANOVA, Dimka ; KOPP, Oliver ; LEYMANN, Frank ; ZWINK, Daniel: Cross-organizational process monitoring based on service choreographies. In: SHIN, Sung Y. (Hrsg.) ; OSSOWSKI, Sascha (Hrsg.) ; SCHUMACHER, Michael (Hrsg.) ; PALAKAL, Mathew J. (Hrsg.) ; HUNG, Chih-Cheng (Hrsg.): *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, ACM Press, New York, NY, USA, 2010. – ISBN 978–1–60558–639–7, S. 2485–2490

[ZBDH06] ZAHA, Johannes M. ; BARROS, Alistair P. ; DUMAS, Marlon ; HOFSTEDE, Arthur H. M.: Let's Dance: A Language for Service Behavior Modeling. In:

MEERSMAN, Robert (Hrsg.) ; TARI, Zahir (Hrsg.): *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, OTM Confederated International Conferences, CoopIS, DOA, GADA, and ODBASE 2006, Montpellier, France, October 29 - November 3, Proceedings, Part I* Bd. 4275, Springer-Verlag, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–48287–3, S. 145–162

[ZDH+06] ZAHA, Johannes M. ; DUMAS, Marlon ; HOFSTEDE, Arthur H. M. ; BARROS, Alistair P. ; DECKER, Gero: Service Interaction Modeling: Bridging Global and Local Views. In: *Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006), 16-20 October, Hong Kong, China*, IEEE Computer Society Press, 2006. – ISBN 0–7695–2558–X, S. 45–55

Abkürzungsverzeichnis

2PC	**Zweip**hasen-Commit-Protokoll (engl. **two p**hase **c**ommit procotol)
ACID	**A**tomicity, **C**onsistency, **I**solation, **D**urability
BACC	WS-BusinessActivity - **B**usiness**A**greementWith**C**oordinator**C**ompletion
BAM	**B**usiness **A**ctivity **M**onitoring
BAPC	WS-BusinessActivity - **B**usiness**A**greementWith**P**articipant**C**ompletion
BPEL	Web Services **B**usiness **P**rocess **E**xecution **L**anguage
BPMN	**B**usiness **P**rocess **M**odel and **N**otation
BPSS	ebXML **B**usiness **P**rocess **S**pecification **S**chema
BTP	**B**usiness **T**ransaction **P**rotocol
CDL	Web Services **C**horeography **D**escription **L**anguage
CORBA	**C**ommon **O**bject **R**equest **B**roker **A**rchitecture
ebXML	**E**lectronic **B**usiness using e**X**tensible **M**arkup **L**anguage
EPR	**E**nd**p**oint **R**eference
ESB	**E**nterprise **S**ervice **B**us
HTTP	**H**yper**t**ext **t**ransfer **p**rotocol
KPI	**K**ey **P**erformance **I**ndicator
LTL	**L**ineare **T**emporale **L**ogik
MAP	**M**essage **A**ddressing **P**roperties
MOM	**M**essage **o**riented **M**iddleware
OASIS	**O**rganization for the **A**dvancement of **S**tructured **I**nformation **S**tandards
QoS	**Q**uality **o**f **S**ervice
RPC	**R**emote **P**rocedure **C**all
SLA	**S**ervice **L**evel **A**greement
SOA	**S**ervice **o**riented **A**rchitecture
UDDI	**U**niversal **D**escription, **D**iscovery and **I**ntegration
URI	**U**niform **R**esource **I**dentifier
W3C	**W**orld **W**ide **W**eb **C**onsortium
WS	**W**eb **S**ervice
WS-AT	**WS**-**A**tomic**T**ransaction
WS-BA	**WS**-**B**usiness**A**ctivitites
WS-BPEL	**W**eb **S**ervices **B**usiness **P**rocess **E**xecution **L**anguage
WS-C	**WS**-**C**oordination
WS-CAF	**W**eb **S**ervices **C**omposite **A**pplication **F**ramework

WS-CDL ..... **W**eb **S**ervices **C**horeography **D**escription **L**anguage
WSCI ..... **W**eb **S**ervice **C**horeography **I**nterface
WSDL ..... **W**eb **S**ervices **D**escription **L**anguage
WSDM ..... **W**eb **S**ervices **D**istributed **M**anagement
XML ..... e**x**tensible **m**arkup **l**anguage