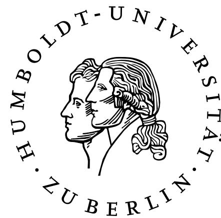
Task Force „Geschäftsprozesssprache BPEL4WS“

Business Process Execution Language for Web services

– Semantik, Analyse und Visualisierung –

Axel Martens, Christian Stahl, Daniela Weinberg,
Dirk Fahland, Thomas Heidinger



Humboldt-Universität zu Berlin
Institut für Informatik
Unter den Linden 6
D-10099 Berlin

Das Projekt TASK FORCE BPEL4WS am Lehrstuhl von Prof. Reisig beschäftigt sich in enger Kooperation mit IBM Deutschland Entwicklung GmbH mit der Geschäftsprozess-Modellierungssprache BPEL4WS (*Business Process Execution Language for Web services*).

Danksagung

Einen wesentlichen Anteil an der vorliegenden Arbeit haben die weiteren Mitarbeiter und Berater der TASK FORCE BPEL4WS: Dr. Karsten Schmidt, Dr. Michael Weber, Stephan Roch, Carsten Frenkler, Ralf Immig, Elke Salecker und Manja Wolf, sowie die Mitglieder der „Kaffeerunde".
Für vielfältige Anregungen und konstruktive Kritik bedanken wir uns bei Prof. Reisig, dem Initiator der TASK FORCE und bei Prof. Leymann, unserem Ansprechpartner auf Seiten von IBM Deutschland.

Zusammenfassung

Moderne Systeme der Informationstechnik bestehen zumeist aus einer Vielzahl von Komponenten, die in einem Netzwerk auf verteilten Knoten ausgeführt werden. Mit dem Web-Service-Ansatz können solche Systeme einfacher und flexibler entwickelt werden. Diese Arbeit befasst sich mit der Modellierung, Visualisierung und Analyse von Web Services.

Ein Web Service kapselt eine Anwendung und stellt diese über ein wohldefiniertes Interface der Außenwelt zur Verfügung. Im Gegensatz zu früheren Ansätzen dienen eine Reihe zusammenhängender Technologien zur Beschreibung eines Web Service. Diese Arbeit beschäftigt sich vor allem mit der internen Struktur eines Web Service, beschrieben mit Hilfe der *Business Process Execution Language for Web Services (BPEL4WS)*[ACD+02].

Der Web-Service-Ansatz bietet ein homogenes Konzept von Komponenten und ihrer Komposition über einem heterogenen Netzwerk. Damit ist die syntaktische Grundlage für die Entwicklung verteilter Systeme gelegt. Wesentlich für den Erfolg der Web Services ist jedoch die Beantwortung der semantischen Fragestellungen: Passen zwei gegebene Web Services inhaltlich zusammen? Kann in einem verteilten System ein gegebener Web Service durch einen anderen ersetzt werden? Entspricht ein konkreter Web Service einer gegebenen abstrakten Spezifikation?

Diese Arbeit befasst sich mit der Beantwortung dieser und weiterer Fragestellungen im Web-Service-Ansatz: In einem ersten Schritt entwickeln wir eine formale Semantik für die Sprache BPEL4WS. Darauf aufbauend werden Methoden zur Analyse verteilter Systeme auf die konkreten Anforderungen übertragen und neue Verfahren entwickelt. Für die Diskussion der Modelle und Eigenschaften entwickeln wir eine intuitive graphische Repräsentation der Sprache BPEL4WS. Das Ziel der Forschungen ist die Umsetzung der Methoden in einem integrierten Entwicklungswerkzeug für BPEL4WS. Die vorliegende Arbeit beschreibt die ersten Ergebnisse in einem laufenden Projekt.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen von Web Services **7**
2.1 Web Services 7
2.2 Business Process Execution Language 12
2.3 Perspektiven von Web Services 14
2.4 Szenarien der Anwendung 15

3 Analyse von Petrinetz-Modulen **17**
3.1 Modellierung 18
3.2 Verifikation 22
3.3 Zusammenfassung 26

4 Eine Petrinetz-Semantik für BPEL4WS **29**
4.1 Ansatz der Semantik 29
4.2 Übersetzung einer Aktivität 31
4.3 Übersetzung der link-Semantik 34
4.4 Ausblick 39

5 Reduktion von BPEL4WS-Petrinetzen **41**
5.1 Notwendigkeit der Reduktion 41
5.2 Reduktionsregeln 42
5.3 Slicing 44
5.4 Ausblick 46

6 Eine ASM-Semantik für BPEL4WS **47**
6.1 Ansatz für eine formale Semantik 47
6.2 Abstract State Machines 48
6.3 Übersetzung von Strukturen 54
6.4 Übersetzung des Verhaltens 60
6.5 Schlußfolgerungen und Ausblick 71

7 Eine intuitive Graphik für BPEL4WS **73**
7.1 Graphik als Chance 73
7.2 visualBPEL 73
7.3 Ausgewählte Elemente 82
7.4 Zusammenfassung und Ausblick 88

8 Der Prototyp WOMBAT4WS **89**
8.1 Entwicklungsumfeld 89
8.2 Architektur 91
8.3 Implementierung 95

9 Abschließende Bemerkungen **105**
9.1 Erreichte Ergebnisse 105
9.2 Weitere Forschungen 107

Literaturverzeichnis **111**

1 Einleitung

In zunehmendem Maße ist die Entwicklung wirtschaftlich bedeutsamer IT-Systeme eine komplexe Herausforderung, die von der ganzheitlichen Betrachtung der Geschäftsprozesse über die Integration bestehender Anwendungssysteme bis hin zur Implementierung auf einer verteilten, heterogenen Infrastruktur reicht. Solche Systeme lassen sich i. Allg. nicht aus einem Guss und von Grund auf neu entwickeln. Stattdessen ist die Entwicklung gekennzeichnet durch die Prinzipien *Dezentralisierung* und *Integration*, z. B. durch Auslagerung von Teilprozessen bei gleichzeitigem Management der organisationsübergreifenden Workflows [AH02].

Für dieses Anwendungsgebiet scheinen *Web Services* [ACKM02] die geeignete technologische Grundlage zu sein, denn sie bieten ein standardisiertes und plattform-unabhängiges Komponentenkonzept. Im Gegensatz zu vergleichbaren Ansätzen (z. B. CORBA) vereint der Web-Service-Ansatz eine ganze Reihe zusammenhängender Technologien. Somit wird einerseits die Komposition auf einer homogenen Schicht ermöglicht und andererseits die flexible Implementierung der einzelnen Komponenten – zugeschnitten auf die jeweilige Infrastruktur. Führende Unternehmen der Software-Industrie, darunter IBM, Microsoft und SUN Microsystems, arbeiten mit Hochdruck an der Standardisierung dieser Technologien und beginnen, darauf aufbauende Produkte am Markt zu platzieren.

Mit dem Web-Service-Ansatz entsteht in absehbarer Zeit eine gemeinsame *Syntax* zur Beschreibung der Struktur und Funktionalität verteilter Systeme. Weiterhin offen sind hingegen *semantische* Fragen: Passen zwei Web Services zusammen, so dass ihre Komposition ein fehlerfreies System ergibt? – die Frage nach *Kompatibilität*. Kann ein Web Service in einem komponierten System durch einen anderen ersetzt werden, ohne dass das System verändert werden muss? – die Frage nach *Äquivalenz*. Kann ein gegebener Web Service überhaupt von einer anderen Komponente fehlerfrei verwendet werden? – die Frage nach *Bedienbarkeit*.

Nach Einschätzung der Gartner Group [Hos02] hängt der Erfolg der Web Services maßgeblich an der methodischen Unterstützung nicht-trivialer Entwicklungsschritte. Das Forschungsprojekt TASK FORCE BPEL4WS am Lehrstuhl für Theorie der Programmierung beschäftigt sich daher mit der Anwendung formaler Methoden zur Beantwortung der o. g. Fragestellungen. Dabei konzentrieren sich die Arbeiten vor allem auf die interne Struktur eines Web Service, beschrieben mit Hilfe der *Business Process Execution Language for Web Services (BPEL4WS)*[ACD+02]. Die Forschungsaktivitäten lassen sich in die vier Bereiche *Formalisierung*, *Analyse*, *Visualisierung* und *Werkzeugentwicklung* unterteilen – Abbildung 1.1 verdeutlicht den Zusammenhang.

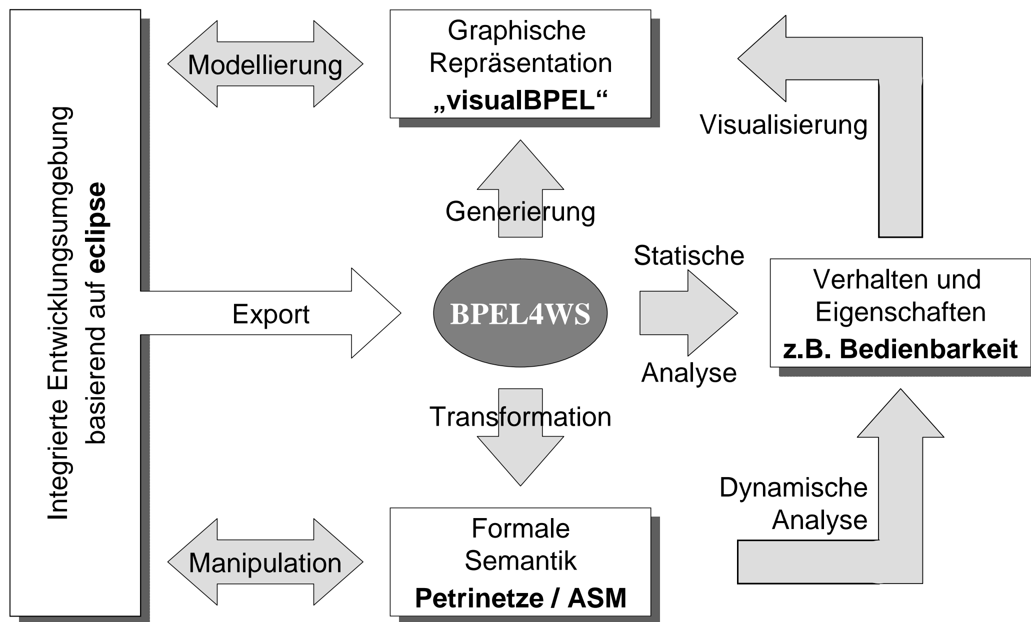


Abbildung 1.1: Roadmap der Task Force BPEL4WS

Formalisierung

Die Business Process Execution Language for Web Services (BPEL4WS) ist eine XML-basierte Sprache zur Beschreibung der Aggregation von Web Services. Sie entstand durch die Vereinigung der Sprachen WSFL von IBM [Ley01] und XLANG von Microsoft [Tha01]. Geschuldet ihrer Entstehungsgeschichte besitzt die Sprache BPEL4WS eine beachtliche Komplexität teilweise mit redundanten sowie mehrdeutig beschriebenen Konzepten. Außerdem wurden die Wechselwirkungen der verschiedenen Konzepte im Vorhinein nicht ausreichend untersucht. Aus diesem Grund ist die Definition einer *formalen Semantik* für BPEL4WS eines der grundlegenden Ziele in diesem Forschungsprojekt. Um die praktische Relevanz sicher zu stellen, formulieren wir eine Reihe von Anforderungen an die entstehenden Semantik:

- Als erstes soll die Semantik zum *Verständnis* der Sprache BPEL4WS dienen. Dazu haben wir eine modulare Herangehensweise gewählt, die jedem Konstrukt der Sprache eine überschaubare Definition zuordnet und den direkten Bezug zwischen der Syntax und der Semantik erhält. Durch eine geeignete Strukturierung der Definitionen lassen sich darüber hinaus Gemeinsamkeiten verschiedener Schrachkonstrukte aufdecken. Komplexe Elemente der Sprache werden schrittweise aufgebaut und die Wechselwirkungen der Bausteine durch Szenarien beschrieben.

- Als zweites sollen mit der Semantik *Unklarheiten* in der Sprache BPEL4WS präzise lokalisiert werden und durch geeignete Festlegungen beseitigt werden. Dazu wurde großer Wert auf die vollständige und exakte Abbildung aller Konzepte der Sprache gelegt. Erst dadurch ließen sich die Probleme im Zusammenspiel von Fehler- und

Kompensationsbehandlung aufdecken. Alle dem gewählten Formalismus geschuldete Designentscheidungen wurden ausführlich diskutiert.

- Als drittes und wichtigstes Ziel soll die Semantik die *Analyse* von Modellen der Sprache BPEL4WS ermöglichen. Dazu wurde bei der Auswahl der Formalismen auf die Verfügbarkeit von etablierten Analysemethoden bzw. die Möglichkeit der Simulation der Modelle geachtet. Darüber hinaus wurden Verifikationsaufgaben aus der Spezifikation von BPEL4WS abgeleitet.

Wie aus der Abbildung 1.1 ersichtlich wird, wurden zwei unterschiedliche Formalismen verwendet: *Petrinetze* und *Abstract State Machines (ASM)*. Um die Möglichkeiten und Grenzen der Formalismen auszuloten, wurden in der ersten Phase zwei unabhängige Definitionen der Semantik erstellt (die jedoch inhaltlich abgeglichen wurden). Diese Arbeiten sind größtenteils abgeschlossen und die Ergebnisse finden sich in diesem Bericht und in weiteren Veröffentlichungen. In einer zweiten Phase sollen dann beide Formalismen vereinigt werden.

Analyse

Wie im Abschnitt zuvor dargelegt, ist die Analyse von Modellen der Sprache BPEL4WS, das heißt, der Nachweis relevanter Eigenschaften wie *Bedienbarkeit*, *Kompatibilität* und *Äquivalenz*, das wichtigste Ziel in diesem Forschungsprojekt. In diesem Bereich sind die Forschungen am weitesten fortgeschritten und dieser Bericht widmet sich ausführlich dieser Fragestellung.

Neben dem Nachweis solcher Standardeigenschaften, sollen aus der Spezifikation von BPEL4WS abgeleitete bzw. von Anwendern der Sprache individuell spezifizierte Anforderungen überprüft werden können. Auch in diesem Bereich verfolgt das Projekt einen zweigeteilten Ansatz. Einige Eigenschaften können direkt auf dem BPEL4WS-Quelltext verifiziert werden, z. B. Initialisierung von Variablen, Gültigkeitsbereiche von Daten und so genannte *race conditions*. Zu ihrem Nachweis kommen Techniken der *statischen Analyse* zum Einsatz. Andere Eigenschaften lassen sich nur mit *dynamischer Analyse* verifizieren, das heißt, mit der Betrachtung der erreichbaren Systemzustände. Hierfür ermöglicht die Petrinetz-basierte Semantik den Rückgriff auf ein ganzes Arsenal von Analysemethoden bis hin zu effizienten Model-Checking-Werkzeugen.

Abbildung 1.1 verdeutlicht beide Ansätze der Verifikation. Die Ergebnisse der Analyse sollen dem Anwender bei beiden Ansätzen in einer verständlichen Form präsentiert werden, das heißt, sie werden zurück in die BPEL4WS-Welt übersetzt und die verwendeten Methoden bleiben weitestgehend verborgen. Dafür benötigt man jedoch eine ansprechende graphische Repräsentation von BPEL4WS-Modellen.

Visualisierung

Die Sprache BPEL4WS bietet eine XML-basierte Syntax zur Beschreibung von Geschäftsprozessen. Selbst für geübte Menschen ist es sicherlich schwer, einen langen, derart notierten BPEL4WS-Prozess komplett zu verstehen und alle Strukturen zu überblicken. Oftmals werden die konkreten Strukturen innerhalb eines Prozesses erst durch ein Bild klar. Wir sehen

in der graphischen Darstellung eines BPEL4WS-Prozesses die Chance, einfacher über diesen Prozess reden zu können und zunächst unsichtbare Strukturen sichtbar zu machen. So können beispielsweise Kontroll- und Nachrichtenflüsse rasch nachvollzogen werden, was innerhalb eines XML-Baumes durchaus ein Problem darstellt.

Es gibt derzeit keine einheitliche graphische Darstellung für BPEL4WS. Ein kurzer Blick in das Internet zeigt eine Flut verschiedenster graphischer Darstellungen – jeder Autor scheint eine andere, ad-hoc entworfene Art der Veranschaulichung zu verwenden. In dieser Arbeit stellen wir mit *visualBPEL* eine systematisch entwickelte graphische Darstellung für BPEL4WS vor. Die Graphik ist einfach gehalten und orientiert sich an wenigen, intuitiven Stilmitteln. Damit soll es jedem Betrachter ermöglicht werden, Prozesse und Strukturen auch ohne tiefgreifende Kenntnisse von BPEL4WS kurzerhand zu erfassen. Jedes syntaktische Element der Sprache ist in die graphische Repräsentation einbezogen. Somit kann für jeden BPEL4WS-Prozess die Graphik automatisch erstellt werden. Um die übersichtliche Darstellung komplexer Prozesse zu erreichen, ist es möglich, auf jeder Ebene Detailinformation ein- bzw. auszublenden.

Wie aus Abbildung 1.1 ersichtlich wird, ist, neben der Darstellung der Struktur eines BPEL4WS-Prozesses, die Visualisierung des Verhaltens sowie wesentlicher Eigenschaften eine weitere Aufgabe in diesem Bereich. Mit einem auf *visualBPEL* basierenden Editor als Komponente einer umfassenden Entwicklungsumgebung für BPEL4WS soll es dem Nutzer ermöglicht werden, auf der gleichen intuitiven Ebene der Darstellung BPEL4WS-Prozesse zu modellieren, zu simulieren und individuelle Eigenschaften zu spezifizieren und zu analysieren. Die Grundlagen hierfür sind mit den vorliegenden Ergebnissen gelegt, die Entwicklung eines solchen Editors ist Gegenstand sich anschließender Forschungen.

Werkzeugentwicklung

Die Bereiche Formalisierung, Analyse und Visualisierung bilden das theoretische Fundament für die systematische Entwicklung strukturierter Web Services und Web-Service-basierter Prozesse. Das Ziel des Forschungsgebiets Werkzeugentwicklung ist es, den Anwendern diese Ergebnisse über ein einheitliches Frontend möglichst umfassend zur Verfügung zu stellen.

Am weitesten fortgeschritten sind die Arbeiten im Bereich der Analyse. Mit dem Werkzeug Wombat4ws [Mar03b] existiert bereits eine prototypische Implementierung der Verfahren zum Nachweis der Bedienbarkeit und weiterer Standardeigenschaften – basierend auf dem am Lehrstuhl entwickelten Petrinetz-Kern PNK [Micb]. Bei diesen Algorithmen stellt, wie bei allen dynamischen Verfahren, die Explosion des Zustandsraums ein gewichtiges Problem dar. Eine Herausforderung für aktuelle Forschungen ist es, diesem Problem durch geeignete Reduktionstechniken zu begegnen.

Neben der Entwicklung eigener Algorithmen bildet die Anbindung existierender Werkzeuge einen Schwerpunkt der Forschungen. Es handelt sich dabei sowohl um Petrinetzspezifische Werkzeuge (z. B. LoLA [Sch00b] und Woflan [VBA00]) als auch um Werkzeuge zur statischen Datenflussanalyse, zur Manipulation von XML-Dokumenten und zur graphischen Aufbereitung von Datenstrukturen.

Letztlich gilt es, die Handhabung der entwickelten Werkzeuge und den einfachen Austausch von Modellen und Informationen mit anderen Werkzeugen zu ermöglichen. Das

entwickelte Werkzeug ist auf die Funktionalität fokussiert, seine modulare Architektur ermöglicht jedoch sowohl die Erweiterung um zusätzliche Verfahren der Analyse als auch die Integration in ein umfassendes Entwicklungswerkzeug. In einer Kooperation mit IBM Deutschland soll daher die Anbindung an die universelle Werkzeug-Plattform *eclipse* [Gal02] realisiert werden.

Aufbau der Arbeit

Dieser Bericht umfasst neben der Einleitung und dem abschließenden Resumee und Ausblick sieben Kapitel, in denen ausgewählte Ergebnisse aus Studien- und Diplomarbeiten sowie einer Dissertation vorgestellt werden. Zu Beginn führt Kapitel 2 in die Begriffswelt der Web Services ein und verschafft einen Überblick der Sprache BPEL4WS sowie Szenarien ihrer Anwendung.

Die anschließenden vier Kapitel beschäftigen sich mit der formalen Methode der Petrinetze: Kapitel 3 präsentiert allgemein die Modellierung von Web Services mit Petrinetzen sowie die Analyse solcher Modelle. Kapitel 4 überführt BPEL4WS in Petrinetze und gibt somit eine formale Semantik für diese Sprache. Kapitel 5 schließlich geht auf die Besonderheiten der Analyse solcher, durch die Überführung entstandener Petrinetze ein und präsentiert Ansätze zur Reduktion des Aufwands.

Die Definition einer formalen Semantik für BPEL4WS mit Hilfe der Abstract State Machines wird in Kapitel 6 beschrieben. Kapitel 7 präsentiert die entwickelte intuitive Graphik für BPEL4WS anhand eines kleinen Beispiels. Mit Wombat4ws wird eine bereits implementierte Komponente aus dem Werkzeugkonzept in Kapitel 8 vorgestellt. Die Arbeit endet mit der Zusammenfassung der erreichten Ergebnisse und einem Ausblick auf aktuelle und zukünftige Forschungen in Kapitel 9.

2 Grundlagen von Web Services

Autor: Axel Martens

Das Internet hat sich in den letzten Jahren von einem reinen Kommunikationskanal hin zu einer Infrastruktur für verteilte Systeme aller Art entwickelt. Durch die weltweite Verfügbarkeit und die einfache und standardisierte Architektur ist es Organisationen prinzipiell möglich, ihre Geschäftsprozesse über das Internet miteinander zu verzahnen. Zwei Beobachtungen verdeutlichen die Probleme bei der praktischen Umsetzung dieser Idee.

- Die IT-Welt ist stark heterogen
  Es existiert eine Vielzahl verschiedener Systeme und Modelle im Umfeld des Workflow-Managements. Das Fraunhofer-Institut für Software- und Systemtechnik listet zur Zeit 51 kommerzielle Produkte [Fra01]. Zur Komposition derart heterogener Prozesse bedarf es einer standardisierten und umfassenden Vermittlungsschicht.

- Isolierte Lösungen helfen nicht
  Viele der existierenden verteilten Geschäftsprozesse basieren auf „handgestrickten" Lösungen, d. h. sie sind oft aufwendig, teuer und unflexibel. Teilweise wurden bereits verschiedene Standards für Komponenten und Protokolle innerhalb verteilter Systeme vorgeschlagen (Corba, RMI, DCOM usw.– siehe [Hee98]). Diese konnten sich aufgrund mangelnder Unterstützung jedoch nicht endgültig durchsetzen.

Führende Unternehmen der Softwareindustrie (darunter IBM, Microsoft, Sun Microsystems u. a.) entwickeln zur Zeit eine gemeinsame, standardisierte Architektur für die Entwicklung, Bereitstellung und Komposition verteilter Dienstleistungen – den *Web-Service-Ansatz*. Der Web-Service-Ansatz nutzt konsequent die Internet-Architektur und bietet so ein Hardware- und Betriebssystem-unabhängiges Konzept von Komponenten und Komposition.

2.1 Web Services

Web Services werden als die „nächste Revolution des Internets" gehandelt [Tid00]. Obwohl viele Software-Hersteller bereits mit dem Schlagwort „Web Service" werben, ist eine einheitliche, konsistente und standardisierte Terminologie zum jetzigen Zeitpunkt noch nicht verfügbar. Wir verwenden an dieser Stelle eine Definition aus [Moh02].

Definition 2.1 (Web Service).
Ein Web Service ist eine abgeschlossene, selbsterklärende und modulare Software-Komponente, die über das Internet veröffentlicht, aufgefunden und benutzt werden kann. Ein Web Service stellt eine beliebig komplexe Funktionalität zur Verfügung. Ein veröffentlichter Web Service wird mit einer anderen Anwendung (möglicherweise ebenfalls ein Web Service) zu einem neuen System komponiert. Der Nachrichtenaustausch zwischen Web Service basiert zumeist auf dem XML-Format. ⋆

Ein Web Service ist somit eine offene Komponente mit einer wohl definierten Schnittstelle zur Außenwelt. Alle zur Benutzung notwendigen Informationen über einen Web Service sind in der öffentlich zugänglichen Beschreibung seiner Schnittstellen enthalten.

Diese Arbeit beschäftigt sich mit der Modellierung und Analyse verteilter Geschäftsprozesse auf der Basis von Web Services. Wir betrachten einen Web Service als Implementierung eines lokalen Geschäftsprozesses. Ein verteilter Geschäftsprozess wird durch die Komposition von mehreren Web Services realisiert.

2.1.1 Service Oriented Architecture

Bevor wir uns etwas detaillierter mit der Beschreibung eines Web Service befassen und den aus einer Vielzahl von eigenständigen Standards bestehende *Web Service Technology Stack* näher ansehen, wollen wir zuerst das Zusammenspiel der Akteure im Web-Service-Ansatz betrachten – beschrieben in der *Service Oriented Architecture*. Abbildung 2.1 (aus [Got00]) zeigt die Grundstruktur dieser Architektur.

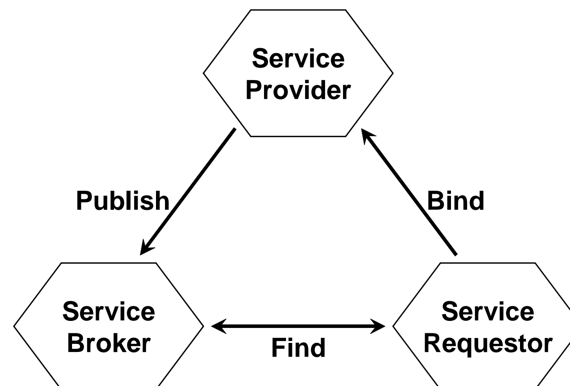


Abbildung 2.1: Rollen und Operationen der Service Oriented Architecture

Im Zusammenhang mit Web Services unterscheidet man drei Rollen. Jeder Teilnehmer ist bezogen auf einen Web Service entweder *Anbieter* oder *Vermittler* oder *Benutzer*.

- Ein Anbieter (Service Provider) stellt einen Web Service zur Verfügung und veröffentlicht dessen Beschreibung in einem (zum Teil öffentlichen) Verzeichnisdienst.
- Ein Vermittler (Service Broker) betreibt einen Verzeichnisdienst. Durch eine geeignete Katalogisierung vermittelt er den Kontakt zwischen Anbieter und Benutzer eines Web Service.
- Ein Benutzer (Service Requestor) kontaktiert den Vermittler, um einen für seine Zwecke passenden Web Service zu finden. Diesen Web Service komponiert der Benutzer mit seinen eigenen Komponenten. Dadurch entsteht ein verteilter Geschäftsprozess oder ein neuer (komponierter) Web Service.

Entsprechend der Rolle kann ein Teilnehmer mit einem Web Service Operationen ausführen. Dabei handelt es sich vor allem um die Operationen *Veröffentlichen*, *Suchen* und *Verbinden*. Abbildung 2.1 verdeutlicht die Zusammenhänge zwischen den Rollen und den Operationen.

- Damit ein Web Service von Dritten verwendet werden kann, muss er in einem Verzeichnis eingetragen oder anderweitig bekannt gemacht werden. Der Anbieter übergibt dazu die Beschreibung seines Web Service dem Vermittler (Operation publish). Diese Beschreibung muss alle zur Benutzung des Web Service notwendigen Informationen enthalten – eine im Allgemeinen schwierig zu entscheidende Eigenschaft. In dieser Arbeit unterstützen wir den Anbieter auf zweierlei Weisen: Er kann vor dem Veröffentlichen seinen Web Service auf *Bedienbarkeit* prüfen und er kann dessen Beschreibung in eine adäquate abstrakte Darstellung überführen – das so genannte *Public-View-Modell*.

- Ein Benutzer sucht beim Vermittler nach einem geeigneten Web Service (Operation find). Dazu muss der Benutzer spezifizieren, welche Eigenschaften der zu findende Web Service haben soll. Der Benutzer sollte keine neue Technik erlernen müssen, sondern mit einem abstrakten Web Service die Suche starten (ähnlich dem *Query-by-Example*-Prinzip). In dieser Arbeit unterstützen wir diesen Ansatz durch die Definition einer *Simulationsbeziehung* zwischen zwei Web Services.

- Hat ein Benutzer einen passenden Web Service gefunden, komponiert er diesen mit seiner eigenen Komponente (Operation bind). Auf diese Weise entsteht ein verteiltes System, das mit dem Anbieter und dem Benutzer mindestens zwei unabhängige Organisationen umfasst. In dieser Arbeit formulieren wir das für die Komposition zweier Web Services wichtige Kriterium der *Kompatibilität*.

Kapitel 3 beschäftigt sich eingehender mit den Eigenschaften der Bedienbarkeit und Kompatibilität sowie der Überführung in ein Public-View-Modell. Weitere Ausführungen zur Äquivalenz bzw. Simulation zwischen Web Services finden sich in [Mar04].

2.1.2 Web Service Technology Stack

Nun beschäftigen wir uns detaillierter mit der Architektur von Web Services. Im Gegensatz zum monolithischen Stil früherer Ansätze basieren Web Services auf einer Reihe zusammenhängender Technologien. In diesem Zusammenhang fällt häufig der Begriff des *Technology Stacks*. Abbildung 2.2 veranschaulicht dessen Aufbau in Anlehnung an [Sle01].

Den Kern (genannt Core Layers) bilden bereits etablierte Technologien. Durch den Rückgriff auf weitgehend akzeptierte Standards gelingt es mit dieser Schicht, zu fast allen existierenden Systemen kompatibel zu sein und eine Hard- und Software-unabhängige Basis für die aufbauenden Schichten zu schaffen. Zu den Kern-Schichten zählen:

Transport

Der Web-Service-Ansatz ist an kein bestimmtes Transport-Protokoll gebunden, um die Flexibilität nicht einzuschränken. Doch die Verwendung weit verbreiteter Protokolle wie TCP/IP und HTTP ist üblich, um Web Services in jedem Umfeld – vor allem auch über Firewalls hinweg – zugänglich zu machen. [Kre01]

Extensible Markup Language (XML)

XML ist ein allgemein anerkanntes Datenformat zum Austausch von Informationen

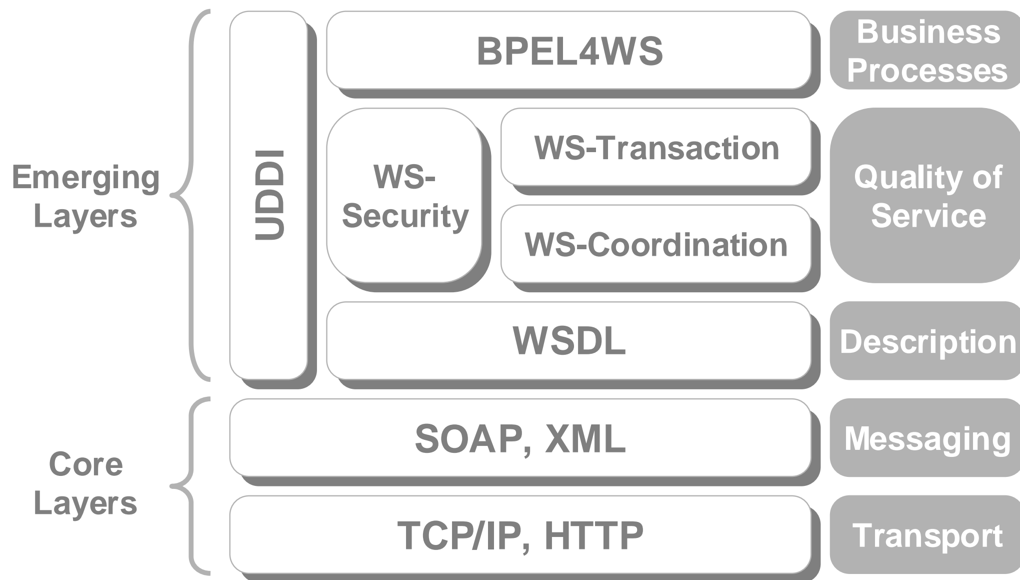


Abbildung 2.2: Web-Service-Technology-Stack

zwischen verschiedenen Anwendungen. Aufgrund seiner Flexibilität und Erweiterbarkeit wird XML als Grundstock der meisten Schichten im Web Service Technology Stack verwendet. [BPMM00]

Simple Object Access Protocol (SOAP)

SOAP unterstützt die Kodierung beliebiger Daten (normalerweise in XML) und deren Transfer (z. B. über HTTP). Im Zusammenhang mit Web Services ist SOAP u. a. eine Programmiersprachen-unabhängige Version von RPC (Remote Procedure Call). Parameter des Funktionsaufrufs und Antworten werden ebenfalls in der SOAP-Nachricht kodiert. [BEK+00]

Die höheren Schichten, hier als Emerging Layers bezeichnet, umfassen die neuen Technologien. Diese Technologien befinden sich teilweise noch im Entwicklungsstadium oder werden zur Zeit standardisiert. Als feste Bestandteile des *Technology Stacks* gelten:

Web Services Description Language (WSDL)

WSDL ist eine XML-basierte Sprache, mit der spezifiziert wird, welche Operationen der Web Service zur Verfügung stellt und wie die Verbindung mit dem Web Service ablaufen soll. Die Konzeption von WSDL sieht eine grobe Zweiteilung vor: Der abstrakte Teil definiert sprach- und plattformunabhängige Typen, Nachrichten, Operationen und Port-Typen. Der konkrete Teil – genannt *binding* – bildet die abstrakten Elemente auf konkrete Datenstrukturen, Protokolle und Adressen ab. [CCMW01]

Universal Description, Discovery, and Integration (UDDI)

UDDI ist die globale Verzeichnisstruktur für Web Services und gleichzeitig ein Protokoll, um Web Services zu suchen und zu veröffentlichen. Eine UDDI-Registrierung besteht aus den Weißen, Gelben und Grünen Seiten:

- Die *Weißen Seiten* sortieren die Kontaktinformationen der Anbieter von Web Services alphabetisch.
- Die *Gelben Seiten* sortieren die Web Services nach Branchen und verweisen auf die Weißen Seiten.
- Auf den *Grünen Seiten* finden sich technische Details zu den angebotenen Web Services und der Verweis auf bestehende Geschäftsprozesse.

WSDL dient als Grundstock zur Identifikation und Auffindung eines Services. [BCR02]

Mit WSDL lassen sich bereits einfache Web Services beschreiben, wie z. B. eine Fahrplanauskunft. Nach außen (zum Benutzer) spezifiziert WSDL die Schnittstelle, d. h. die Adresse des Web Service und das Nachrichtenformat. Nach innen (zur Ausführungskomponente) definiert WSDL die Anbindung von realer Software an die Schnittstelle.

Aus der Definition 2.1 geht jedoch explizit die Ausrichtung des Web-Service-Ansatzes auf beliebig komplexe Funktionalität hervor. Aus diesem Grund bestreben allen voran IBM und Microsoft die Standardisierung weiterer wichtiger Technologien, mit denen sich qualitative Aussagen über Web Services formulieren lassen. Darüber hinaus soll es möglich sein, zwischen den Operationen eines Web Service Kausalität zu definieren. Erst auf diese Weise lassen sich Geschäftsprozesse beschreiben. Als Erweiterungen des Web Service Technology Stacks sind zu nennen:

Quality of Service (QoS)

Neben der *syntaktischen Kompatibilität* zweier Web Services – d. h. zueinander passende Nachrichtentypen und Operationen – gibt es beim Zusammenspiel verteilter Komponenten in verschiedenen Organisationen eine Vielzahl weiterer Aspekte, die zueinander passen müssen. Ein Teil davon wird als *Quality of Service* bezeichnet. Zur Beschreibung und zum Abgleich dieser Aspekte, befinden sich zur Zeit etliche *WS...*-Sprachen im Prozess der Standardisierung. Exemplarisch seien hier genannt:

WS-Security [ADLH+02] vereinigt die Mechanismen für Sicherheit und Authentifizierung und soll ermöglichen, Informationen verschlüsselt zwischen einzelnen Unternehmen auszutauschen. WS-Security definiert Erweiterungen für SOAP und stellt so die Integrität und den Schutz der Daten in verteilten Anwendungen sicher.

WS-Coordination [CCC+02a] dient zur Koordinierung von Web-Services unterschiedlicher Unternehmen. Über WS-Coordination können sich Web Services einander bekannt machen und ihre Kommunikation aufeinander abstimmen.

WS-Transaction [CCC+02b] setzt auf WS-Coordination auf und erlaubt es, einen Web-Service-basierten Geschäftsprozess im laufenden Betrieb nach einem Transaktionsprotokoll zu überprüfen. Damit kann rechtzeitig auf Störungen oder Fehler im Geschäftsprozess reagiert werden.

Business Process Execution Language

Die *Business Process Execution Language for Web Services* (BPEL4WS [ACD+02]) ist eine XML-basierte Sprache zur Beschreibung verteilter Geschäftsprozesse. Mit BPEL4WS kann sowohl der Geschäftsprozess innerhalb eines Web Service als auch

die Interaktion mit seinen Partnern spezifiziert werden. Der Geschäftsprozess in einem Web Service kann Web Services von Partnern einbinden. Im Gegenzug wiederum steht er als Baustein anderen zur Verfügung. BPEL4WS kombiniert die Sprachen WSFL von IBM [Ley01] und XLANG von Microsoft [Tha01], übernimmt deren Funktionen und ersetzt sie.

Der hier vorgestellte *Web Service Technology Stack* ist bei weitem nicht der einzige Ansatz, eine Architektur für Web Services zu definieren, jedoch der am weitesten verbreitete und anerkannte. Ein konkurrierender Ansatz ist *Electronic Business XML (ebXML)* von OASIS. In dieser Arbeit befassen wir uns vor allem mit BPEL4WS und verzichten auf einen Vergleich der alternativen Ansätze. Der generelle Charakter vieler Resultate in den folgenden Kapiteln lässt eine einfache Übertragung der Ergebnisse jedoch als möglich erscheinen.

2.2 Business Process Execution Language

In diesem Abschnitt stellen wir sehr grob die grundlegenden Konzepte der Sprache BPEL4WS vor. In der weiteren Arbeit werden die einzelnen Konzepte anhand von Beispielen genauer erläutert. Darüber hinaus bieten die Spezifikation der Sprache [ACD+02] und Tutorials auf der Web-Seite von IBM [Tid00] tiefer gehende Informationen.

Ein Modell der Sprache BPEL4WS beschreibt die interne Struktur eines Web Service und dessen Einbindung in die Umwelt, d. h. ein Modell besteht aus einer Menge von *Aktivitäten* die zu einem *Prozess* zusammengefasst werden und einer Menge von Nachrichtenkanälen (*Operationen* von *Porttypen*), die *Partnern* angeboten oder von diesen verlangt werden. Wir beginnen zuerst mit der internen Struktur:

Aktivitäten

Im Zentrum der Modellierung von Geschäftsprozessen stehen die Aktivitäten und ihre kausalen Zusammenhänge. In BPEL4WS gibt es zwei Sorten von Aktivitäten: *Basic Activities* und *Structured Activities*.

Die *einfachen Aktivitäten* repräsentieren atomare Einheiten und lassen sich erneut in zwei Gruppen teilen: Für die Kommunikation mit einem anderen Web Service sind die Aktivitäten *invoke*, *receive* und *reply* zuständig. Mit *invoke* kann ein anderer Web Service aufgerufen werden, d. h. es wird eine Nachricht an diesen gesendet und gleich oder später eine Antwort erwartet. Mit *receive* wird ein Aufruf (eine Nachricht) von einem anderen Web Service entgegen genommen und mit *reply* dieser beantwortet.

Die zweite Gruppe der einfachen Aktivitäten repräsentieren interne Schritte: *assign* ist eine Wertzuweisung, *wait* ist ein Timer und *empty* ist eine leere Aktivität. Teilweise steuern sie auch den weiteren Prozessverlauf: *terminate* bricht den Prozess ab, *throw* wirft einen Fehler und *compensate* veranlasst die Rücksetzung eines Teils des Prozesses.

Neben den einfachen Aktivitäten gibt es in BPEL4WS auch fünf Klassen *strukturierter Aktivitäten*. Mit diesen Aktivitäten wird der Kontrollfluss abgebildet: Die einfachste ist *sequence*, diese Aktivität definiert die sequentielle Ordnung einer Menge anderer Aktivitäten. Die alternative Auswahl zwischen Aktivitäten ist mit Hilfe von *pick* und *switch* möglich, bei *pick* entscheidet eine Nachricht von außen, bei *switch* wird durch die Auswertung von Daten

eine Entscheidung getroffen. Mit der Aktivität *while* ist es möglich, zyklisches Verhalten zu definieren. Letztlich dient die Aktivität *flow* dazu, eine unabhängige Menge von Aktivitäten zu spezifizieren. Innerhalb von *flow* können die Aktivitäten durch zusätzliche *links* untereinander synchronisiert werden. Jede der strukturierten Aktivitäten enthält ihrerseits mindestens eine Aktivität. Auf diese Weise können durch Schachtelung beliebig komplexe Kontrollfluss-Beziehungen gebildet werden.

Ein Sonderrolle spielt die Aktivität *scope*.

Überwachte Aktivitäten

In BPEL4WS ist es möglich, eine einzelne Aktivität unter besondere Beobachtung zu stellen, d. h. auftretende Fehler abzufangen, auf externe Ereignisse zu reagieren und ggf. die Aktivität nach erfolgreicher Ausführung zu kompensieren, wenn es die äußeren Umstände verlangen. Zu diesem Zweck gibt es das Konzept des *scopes* als Aggregation aus *event handler*, *fault handler*, *compensation handler* und einer überwachten Aktivität. Auf der einen Seite kann die überwachte Aktivität wiederum strukturiert sein, so dass der *scope* eine Menge von Aktivitäten überwachen kann. Auf anderen Seite kann der *scope* selbst als eine Aktivität aufgefasst werden und damit in einer strukturierten Aktivität vorkommen.

Prozesse

Aktivitäten können ineinander geschachtelt sein und bilden somit eine hierarchische Struktur – einen *Aktivitäten-Baum*. Die Wurzel dieses Baumes ist der *Prozess*: Ein Prozess besitzt genau eine Aktivität. Neben dieser Aktivität kann ein Prozess *event handler*, *fault handler* und *compensation handler* besitzen, d. h. ein Prozess ist auch gleichzeitig ein *scope*.

In BPEL4WS gehört zu einem Web Service genau ein Prozess. Es ist möglich, diesen Prozess präzise und vollständig zu modellieren. Auf diese Weise gelangt man zu einem direkt ausführbaren Prozessmodell (= *Executable Process*). Darüber hinaus gestattet BPEL4WS eine abstrakte Spezifikation des Verhaltens (= *Business Protocol*). Die abstrakte Spezifikation ist ein wesentlicher Teil der veröffentlichten Beschreibung des Web Service. Ein potentieller Anwender entscheidet aufgrund dieser Spezifikation, ob der vorliegende Web Service mit seiner Komponente *kompatibel* ist. Im Rahmen dieser Arbeit werden wir die Kompatibilität zweier Web Services definieren (siehe Abschnitt 3.1.4).

Die Definition der Kommunikation innerhalb des Prozesses stützt sich auf die Definition der Schnittstelle des Web Service mit WSDL ab.

Kommunikation

Ein Web Service ist ein offenes System, dass mit anderen Komponenten (o. B. d. A. ebenfalls Web Services) über Nachrichtenaustausch kommuniziert. Jede Nachricht ist ein XML-Dokument, wobei die Typ der Dokumente bereits im WSDL-Modell des Web Service definiert werden. Als konsequente Weiterführung verwendet BPEL4WS für interne Datenstrukturen (*variables*) ebenfalls XML-Dokumente.

Die Schnittstelle für die Kommunikation wird in WSDL definiert: Eine Nachrichten ist ein XML-Dokument, das Senden und empfangen einer Nachricht (mit oder ohne Feedback)

heißt *Operation*. Eine Menge von Operationen wird zu einem *Porttype* zusammengefasst. Jeder Web Service bietet eine Menge von Porttypen an.

Die Kommunikation über einen Porttypen kann man von zwei Seiten aus betrachten: Auf der einen Seite gibt es den Web Service, der diesen Porttypen anbietet und auf der anderen Seite gibt es einen Web Service, der diesen Porttypen von außen benutzt. Deshalb wird in BPEL4WS die Verbindung zwischen den Operationen eines Porttypen und den Aktivitäten im Prozess durch *PartnerLinks* abgebildet: Die Aktivität *receive* und *reply* implementieren eine Operation an der Schnittstelle dieses Web Service und stellen damit einem anderem Web Service Funktionalität zur Verfügung, die Aktivität *invoke* spezifiziert eine Operation an der Schnittstelle eines anderen Web Service und benutzt damit dessen Funktionalität. Da ein anderer Web Service (d. h. ein *Partner*) in einem BPEL4WS-Modell sowohl Funktionalität anbieten als auch benutzen kann, wird ein *Partner* durch eine Menge von *PartnerLinks* spezifiziert.

2.3 Perspektiven von Web Services

In der allgemeinen Euphorie, die stets mit dem Aufkommen einer neuen Technologie verbunden ist, werden Web Services als das Allheilmittel verteilter Informationssysteme angesehen. Auf der anderen Seite formiert sich eine Schar von Kritikern, die in Web Services nur eine neue Marketing-Kampagne für alt-hergebrachte Konzepte von Komponenten und Komposition sehen. Wie sooft liegt die Wahrheit irgendwo dazwischen.

Abbildung 2.3 zeigt eine Darstellung der Gartner-Group [Gar02]. Darin sind die zur Zeit diskutierten Technologien und ihr aktuelles Entwicklungsstadium verzeichnet. Web Services haben nach dieser Einschätzung den Punkt der höchsten Aufmerksamkeit bereits überschritten und befinden sich auf dem Weg der Konsolidierung. In zwei bis fünf Jahren wird diese Technologie ausgereift sein und von jedermann so selbstverständlich benutzt werden, wie heute das Internet als Informationsquelle.

Im Zuge dieser Weiterentwicklung verschieben sich die Anwendungsgebiete des Web-Service-Ansatzes. Nach Einschätzung der IBM fokussiert sich die weitere Entwicklung auf drei Aspekte:

Virtuelles Komponentenkonzept

Web Services eignen sich für nahezu alle Anwendungsgebiete moderner Informationstechnik: Von der Produktionssteuerung und -überwachung bis zum E-Commerce. Durch ihren modularen Aufbau mit austauschbaren Schichten können sich Web Services zu einem durchgängig akzeptierten Konzept entwickeln und damit heterogene Strukturen hinter einem homogenen Komponentenkonzept verbergen – so genannte *Virtuelle Komponenten.*

Inhouse-Lösungen

Schon heute wird ein großer Teil der existierenden Web Services nicht über das Internet und HTTP eingebunden, sondern innerhalb von Unternehmens-Netzwerken mit effizienter Middleware abgewickelt. Probleme langer Übertragungszeiten und asynchroner Protokolle entfallen bei diesem Vorgehen. Andere Probleme spielen dafür eine Rolle: Transaktionen und Fehlerbehandlung.

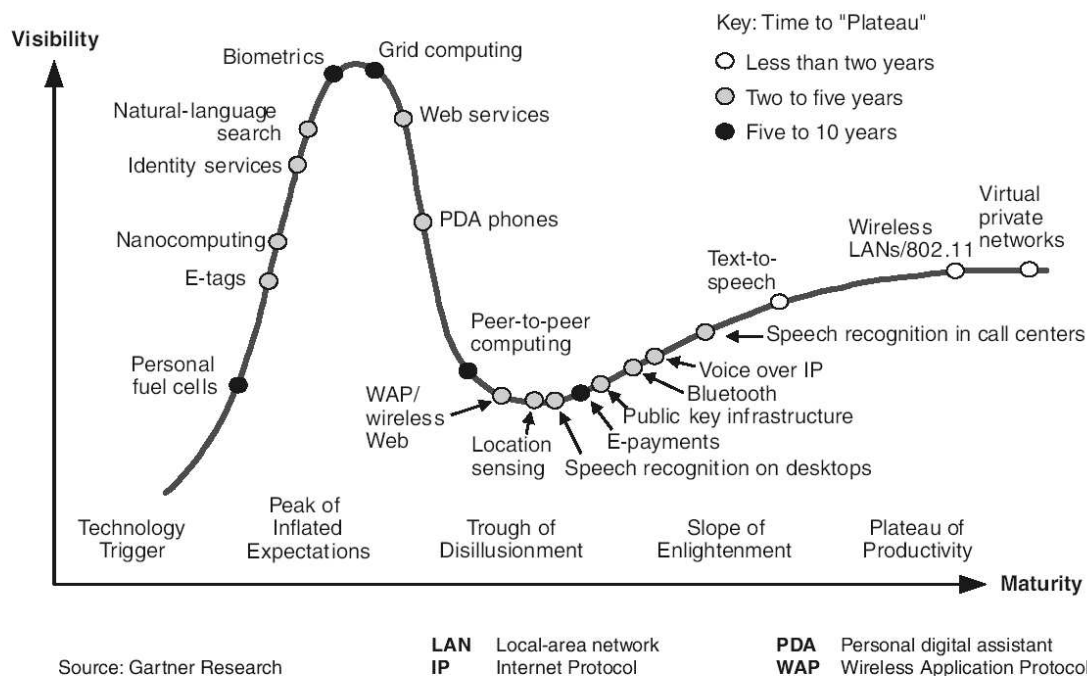


Abbildung 2.3: The 2002 Hype Cycle of Emerging Technologies

Business Grid

Die konsequente Weiterführung der Philosophie von Web Services führt zur Herausbildung so genannter *Business Grids*: Ein Anwender sucht nicht mehr einen speziellen Partner sondern fragt eine Funktionalität nach. Aus einem Pool verfügbarer Services wird ihm dann dynamisch ein Partner zugeordnet, der die spezifizierte Funktionalität bietet. Eine wesentliche Aufgabe dabei ist der Abgleich von gesuchter und angebotener Funktionalität.

Die Erfahrung lehrt, dass in den nächsten zwei bis fünf Jahren der Weiterentwicklung viele Technologien des Web Service Technology Stack geändert, verfeinert oder ausgetauscht werden. Aus diesem Grund löst sich die vorliegende Arbeit von der konkreten Spezifikation und betrachtet die zugrunde liegenden Konzepte. Damit wird eine größere Robustheit gegen syntaktische Änderungen bei gleichzeitig enger semantischer Bindung an die Thematik verteilter Geschäftsprozesse angestrebt.

2.4 Szenarien der Anwendung

Das Ziel dieser Arbeit ist eine durchgängige Entwicklungsmethode für verteilte Geschäftsprozesse auf Basis von Web Services. Dazu gehört die Unterstützung einer schrittweisen Modellierung und einer effektiven Analyse. Sowohl die Modellierungs- als auch die Analysemethoden müssen auf die konkreten Erfordernisse zugeschnitten sein. Wir skizzieren daher drei typische Anwendungsszenarien und leiten daraus die Anforderungen ab:

Szenario 1 - Anwenden von Web Services

Ein Anwender wählt einen verfügbaren Web Service, um ihn mit seiner Komponente (möglicherweise auch ein Web Service) zu komponieren. Passen die Schnittstellen beider Komponenten zusammen wie von der Web-Service-Architektur verlangt, dann entsteht ein syntaktisch korrektes System. Es ist jedoch nicht garantiert, dass das komponierte System ein „vernünftiges“ Verhalten hat – bereits in der Einführung haben wir dieses Problem anhand eines Versandhauses und eines Kunden skizziert. Der Anwender erwartet daher von der Entwicklungsmethode Aussagen über die Kompatibilität von Web Services.

In dieser Arbeit wird ein geeignetes Kriterium für „vernünftige“ verteilte Geschäftsprozesse entwickelt, das den Bedürfnissen der Praxis standhält. Mit diesem Kriterium ist es möglich, *semantische Kompatibilität* von Web Services als Voraussetzung für ihre Komposition zu charakterisieren.

Szenario 2 - Anbieten von Web Services

Ein Anbieter möchte seinen Web Service anderen Anwendern zur Verfügung stellen. Er weiß nicht, mit welchen anderen Komponenten sein Web Service später komponiert wird. Dennoch möchte der Anbieter wissen, ob sein Web Service überhaupt mit anderen zu einem „vernünftigen“ Gesamtsystem komponierbar ist oder ob ein gravierender Fehler in der Modellierung diesen Web Service unbrauchbar macht. Wenn es wenigstens eine Umgebung (d. h. einen weiteren Web Service) gibt mit der man diesen Web Service zu einem „vernünftigen“ Gesamtsystem komponieren kann, dann nennen wir den Web Service „bedienbar“.

In dieser Arbeit wird ein geeignetes Kriterium für *Bedienbarkeit* eines Web Service entwickelt, das nur von dem Web Service selbst abhängt. Darüber hinaus werden syntax-basierte Methoden zum Finden, Beseitigen und ggf. Vermeiden von Modellierungsfehlern entwickelt.

Szenario 3 - Austauschen von Web Services

Ein Anbieter möchte einen existierenden Web Service durch einen neuen Web Service ersetzen (z. B. aus Effizienzgründen). Dabei stellt sich die Frage, ob der neue Web Service alles kann, was der alte auch konnte, d. h. der neue Web Service simuliert den alten. Möglicherweise sind der neue und der alte Web Service nach außen hin nicht unterscheidbar, d. h.äquivalent. In dieser Arbeit werden Kriterien für die *Simulation* und die *Äquivalenz* von Web-Services entwickelt sowie Methoden, um diese Eigenschaften effizient zu entscheiden.

3 Analyse von Petrinetz-Modulen

Autor: Axel Martens

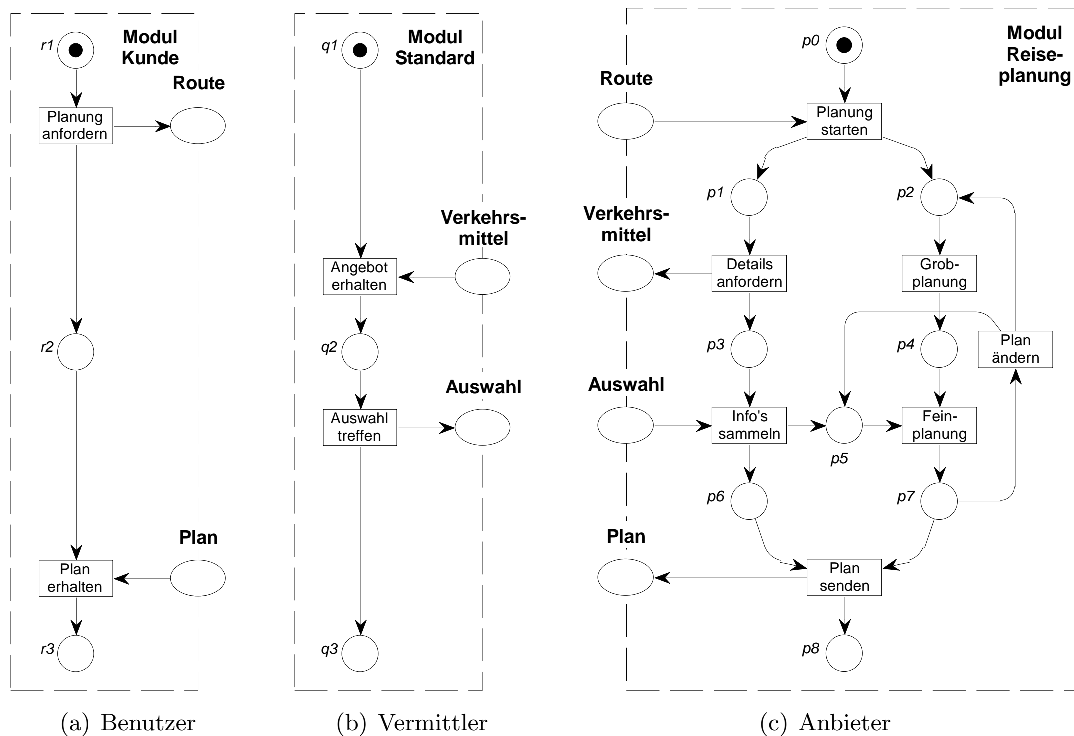
Aus der Notwendigkeit, Geschäftsprozesse über Unternehmensgrenzen hinweg zu organisieren entstand der Wunsch, jeden einzelnen Teilprozess durch eine lokal abgeschlossene Komponente zu realisieren und einen verteilten Geschäftsprozess durch deren Komposition über standardisierte Kommunikationsprotokolle zu etablieren. Für dieses Anwendungsgebiet scheinen *Web Services* die geeignete technologische Grundlage zu sein, denn sie bieten ein standardisiertes und plattform-unabhängiges Komponentenkonzept. Jeder lokale Geschäftsprozess kann durch einen oder mehrere Web Services implementiert werden, die Komposition von Web Services realisiert den verteilten Prozess.

Mit dem Web-Service-Ansatz entsteht in absehbarer Zeit eine gemeinsame *Syntax* zur Beschreibung der Struktur und Funktionalität verteilter Systeme. Weiterhin offen sind hingegen *semantische* Fragen: Passen zwei Web Services zusammen, so dass ihre Komposition ein fehlerfreies System ergibt? – die Frage nach *Kompatibilität.* Kann ein Web Service in einem komponierten System durch einen anderen ersetzt werden, ohne dass das System verändert werden muss? – die Frage nach *Äquivalenz.* Kann ein gegebener Web Service überhaupt von einer anderen Komponente fehlerfrei verwendet werden? – die Frage nach *Bedienbarkeit.* Nach Einschätzung der Gartner Group [Hos02] hängt der Erfolg der Web Services maßgeblich an der methodischen Unterstützung nicht-trivialer Entwicklungsschritte. Das Ziel der Arbeit ist es, formale Methoden für die Beantwortung der o. g. Fragestellungen und damit für den systematischen Entwurf verteilter Geschäftsprozesse zu entwickeln und in ein durchgängiges Vorgehensmodell zu integrieren.

Die Erfahrung lehrt, dass in den nächsten zwei bis fünf Jahren der Weiterentwicklung viele Konzepte in der Web-Service-Architektur geändert, verfeinert oder ausgetauscht werden. Aus diesem Grund löst sich die vorliegende Arbeit von der konkreten Spezifikation und betrachtet die zugrunde liegenden Konzepte. In diesem Kapitel werden Web Services mit Hilfe von Petrinetzen modelliert. Petrinetze sind eine etablierte Methode zur Modellierung in sich geschlossener Geschäftsprozesse. Sie gestatten es, die syntaktischen Strukturen existierender Technologien (z. B. BPEL4WS, WSCI) präzise und anschaulich zu unterlegen. Damit wird eine größere Robustheit gegen syntaktische Änderungen bei gleichzeitig enger semantischer Bindung an die Thematik verteilter Geschäftsprozesse erreicht.

Im Kern der Betrachtungen steht ein Kriterium für vernünftige Modelle von Web Services – die *Bedienbarkeit.* Mit diesem Kriterium ist es möglich, globale Eigenschaften verteilter Geschäftsprozesse durch lokale Verifikation zu garantieren. Die Relevanz der Bedienbarkeit zeigt sich in den vielfältigen Anwendungen dieses Kriteriums, und die Handhabbarkeit ist durch die prototypische Implementierung der Analyse belegt.

Das Kapitel präsentiert eine Kurzfassung der Arbeit [Mar04] in Form eines intuitiven Überblicks der Methode: Abschnitt 3.1 zeigt an einem kleinen Beispiel die Modellierung mit Petrinetzen, die Komposition von Workflow-Modulen und diskutiert die wesentlichen



(a) Benutzer (b) Vermittler (c) Anbieter

Abbildung 3.1: Modellierung mit Workflow-Modulen

Eigenschaften. Der Nachweis der Bedienbarkeit wird im Abschnitt 3.2 behandelt. Dazu wird anhand eines etwas komplexeren Beispiels die algorithmische Idee skizziert. Abschnitt 3.3 belegt die Bedeutung der Bedienbarkeit mit dem Verweis auf weitere praktische Anwendungen der Methode und fasst die Ergebnisse zusammen.

3.1 Modellierung

Dieser Abschnitt beschreibt die Modellierung eines Web Service und die Komposition von mehreren Web-Service-Modellen auf Basis von Petrinetzen. Dazu betrachten wir das Zusammenspiel von drei Akteuren: einem *Anbieter*, einem *Vermittler* und einem *Kunden*. Abbildung 3.1 zeigt die Modelle ihrer Web Services.

3.1.1 Modellierung mit Workflow-Modulen

Der Web Service des Anbieters erstellt eine Reiseplanung und kommuniziert zu diesem Zweck mit seiner Umgebung über vier Kanäle: zwei eingehende (Route, Auswahl) und zwei ausgehende (Verkehrsmittel, Plan). Abbildung 3.1(c) zeigt ein *Petrinetz* als Modell dieses Web Service. Dabei repräsentieren ein Rechteck eine Aktivität (genannt *Transition*) und ein Kreis bzw. eine Ellipse einen Kanal zwischen den Aktivitäten (genannt *Stellen*). Eine Kante stellt die Beziehung zwischen einer Stelle und einer Transition dar. Eine Nachricht in einem Kanal wird durch eine *Marke* auf der Stelle repräsentiert. Der gewählte Ansatz abstrahiert

vom Inhalt der Nachricht, das heißt, zwei Marken auf einer Stelle können nicht voneinander unterschieden werden. Man nennt ein solches Modell ein *einfaches (low-level) Petrinetz* – eine grundlegende Einführung findet sich in [Rei85]. In einem aktuellen Sammelband [GV02] wird der Einsatz von Petrinetzen in der ingenieurmäßigen Entwicklung komplexer Systeme umfassend beleuchtet.

Ein Petrinetz beschreibt neben der Struktur auch das *Verhalten* eines Systems. Im Beispiel aus Abbildung 3.1(c) kann die Transition Planung starten erst *schalten* (d. h. die Aktivität stattfinden), wenn die Anfrage eines Kunden über den Kanal Route eintrifft. Beim Schalten der Transition wird diese Marke sowie die Marke von p0 entfernt und jeweils eine Marke auf p1 und p2 erzeugt. Damit stößt die Transition Planung starten zwei parallele Handlungsstränge an: Einerseits wird dem Kunden eine Liste der möglichen Verkehrsmittel gesendet und seine Auswahl erwartet. Andererseits findet bereits eine Grobplanung statt. Die Feinplanung kann jedoch erst stattfinden, wenn die Auswahl des Kunden verarbeitet wurde, d. h. eine Marke auf p5 liegt. Nach der Feinplanung besteht die Möglichkeit, den Prozess durch die Transition Plan senden zu beenden oder die Arbeitsschritte Grob- und Feinplanung erneut durchzuführen. Das Petrinetz beschreibt hier ein *nichtdeterministisches* Verhalten, das heißt, die Entscheidung für eine der Möglichkeiten ist offen gelassen.

Ein Web Service realisiert einen lokal abgeschlossenen Geschäftsprozess, der mit seiner Umwelt über Nachrichtenaustausch kommuniziert. Aus diesem Grund besteht ein *Workflow-Modul* – das Petrinetzmodell eines Web Service – aus einem *Workflow-Netz* [Aal98], welches den internen Prozess des Web Service modelliert und einem *Interface* – einer Menge von (Schnitt-)Stellen: Eine *Input-Stelle* repräsentiert dabei einen Nachrichtenkanal von der Umgebung zum Web Service, analog ist eine *Output-Stelle* ein Kanal vom Modul zu seiner Umgebung. Aus Gründen der Einfachheit fordern wir, dass jede Transition mit höchstens einer Sorte Schnittstellen verbunden ist.

Definition 3.1 (Workflow-Modul).
Ein endliches Petrinetz $M = (P, T, F)$ heißt *Workflow-Modul* (kurz: *Modul*), wenn folgende Bedingungen gelten:

(i) Die Mengen der Stellen $P = P^N \cup P^I \cup P^O$ ist partitioniert in *interne Stellen* P^N, *Input-Stellen* P^I und *Ouput-Stellen* P^O.

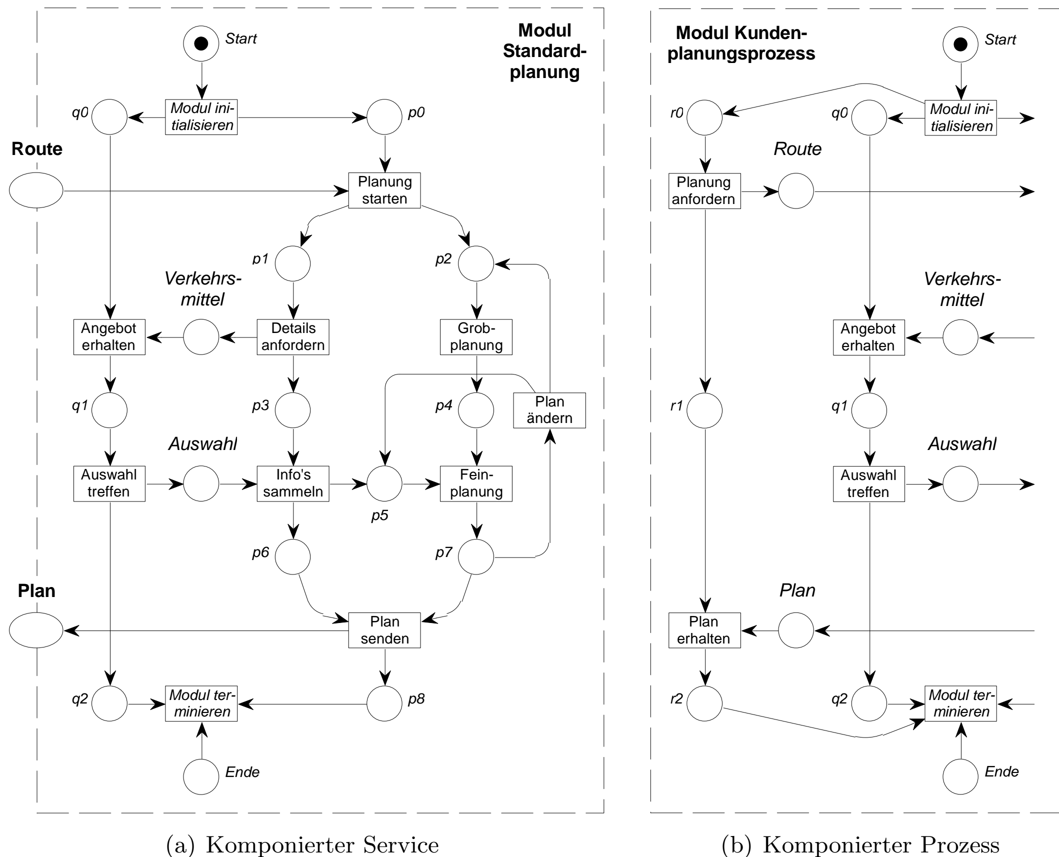
(ii) Die Flussrelation $F = F^N \cup F^C$ ist partitioniert in *interne Kanten* $F^N \subseteq (P \times T) \cup (T \times P)$ und *Kommunikationskanten* $F^C \subseteq (P^I \times T) \cup (T \times P^O)$.

(iii) Das Netz $\mathcal{N}(M) = (P^N, T, F^N)$ ist ein Workflow-Netz.

(iv) Keine Transition ist sowohl mit einer Input-Stelle als auch mit einer Ouput-Stelle verbunden, d. h. für alle $t \in T$ gilt $\left({}^\bullet t \cap P^I = \emptyset\right) \vee \left(t^\bullet \cap P^O = \emptyset\right)$. ⋆

3.1.2 Komposition von Workflow-Modulen

Der Zweck eines Web Services ist die Komposition mit anderen Komponenten. Deshalb müssen auch Workflow-Module miteinander komponiert werden können. Dies geschieht über das Zusammenstecken (d. h. *Verschmelzen*) der Schnittstellen. Wir betrachten die paarweise



(a) Komponierter Service (b) Komponierter Prozess

Abbildung 3.2: Komposition von Workflow-Modulen

Komposition, denn der allgemeine Fall der Komposition beliebig vieler Module kann auf diesen zurückgeführt werden.

Notwendige Voraussetzung für die Komposition zweier Module ist die *syntaktische Kompatibilität* der Schnittstellen. Das bedeutet, dass jede *gemeinsame* Schnittstelle beider Module in einem Modul eine Input-Stelle und in dem anderen eine Output-Stelle sein muss. Betrachten wir die Module aus Abbildung 3.1. Das Modul Standard ist kompatibel zum Modul Reiseplanung, ebenso das Modul Kunde. Wie gesehen, müssen syntaktisch kompatible Module keinesfalls vollständig überdeckende Interfaces besitzen. Die Schnittstellen des einen Moduls müssen auch keine Teilmenge der Schnittstellen des anderen sein, im Extremfall haben zwei kompatible Module überhaupt keine gemeinsame Schnittstelle (z. B. Modul Kunde und Modul Standard). Dann ist jedoch zumindest der Sinn der Komposition fraglich.

Betrachten wir nun die Komposition der Module Standard und Reiseplanung. Der Zweck des Moduls Standard ist es, eine Vorauswahl abzubilden. Dazu erhält dieses Modul die Liste der verfügbaren Verkehrsmittel und trifft eine geeignete Auswahl anstelle des Kunden. Im Ergebnis der Komposition soll daher wiederum ein Workflow-Modul entstehen, das dem Kunden ein einfacheres Interface bietet. Aus diesem Grund wird das Netz nach dem Verschmelzen der Schnittstellen um eine *Initialisierungs-* und eine *Terminierungskomponente* ergänzt. Abbildung 3.2(a) zeigt das Modul Standardplanung – das Ergebnis der Komposition

Standard $\oplus$ Reiseplanung. Das neue Interface besteht aus den offen gebliebenen Schnittstellen beider Module. Ein solches Modul nennen wir Modell eines *verteilten Web Service.*

Definition 3.2 (Komponiertes System).
Seien $A = (P_a, T_a, F_a)$ und $B = (P_b, T_b, F_b)$ zwei syntaktisch kompatible Workflow-Module. Seien $\alpha_s, \omega_s \notin (P_a \cup P_b)$ zwei neue Stellen und $t_\alpha, t_\omega \notin (T_a \cup T_b)$ zwei neue Transitionen. Das *komponierte System* $\Pi = A \oplus B$ ist das Petrinetz (P_s, T_s, F_s), welches gegeben ist durch:

- $P_s = P_a \cup P_b \cup \{\alpha_s, \omega_s\}$
- $T_s = T_a \cup T_b \cup \{t_\alpha, t_\omega\}$
- $F_s = F_a \cup F_b \cup \{(\alpha_s, t_\alpha), (t_\alpha, \alpha_a), (t_\alpha, \alpha_b), (\omega_a, t_\omega), (\omega_b, t_\omega), (t_\omega, \omega_s)\}$ ⋆

Das Modul Kunde aus Abbildung 3.1(a) ist syntaktisch kompatibel zu diesem neu entstandenen Modul. Darüber hinaus überdecken sich die Schnittstellen beider Module vollständig. Wir nennen deshalb das Modul Kunde eine *Umgebung* des Moduls Standardplanung (und umgekehrt). Das Ergebnis der Komposition ist ein Workflow-Modul mit einem leeren Interface – angedeutet in Abbildung 3.2(b). Ein solches Modul nennen wir Modell eines *verteilten Geschäftsprozesses.* Dabei ist anzumerken, dass das Modul Standardplanung bereits aus syntaktischen Gründen um eine *Initialisierungs-* und *Terminierungskomponente* ergänzt wurde. Daher werden bei der weiteren Komposition mit dem Modul Kunde lediglich neue Kanten und keine neuen Transitionen eingefügt. Auf diese wird die Assoziativität der Komposition erreicht (d. h. $(A \oplus B) \oplus C = A \oplus (B \oplus C)$).

3.1.3 Bedienbarkeit von Workflow-Modulen

Ein Schwerpunkt der Arbeit ist die Modellierung verteilter Geschäftsprozesse. Ein Grund für den Erfolg von Petrinetzen im Bereich Geschäftsprozessmodellierung liegt in dem einfachen und zweckmäßigen Kriterium für „vernünftige“ Geschäftsprozesse (siehe *Soundness*-Kriterium [Aal98]):

1. Jeder begonnene Prozess muss auch terminieren können,
2. keine Nachricht darf in einem Kanal vergessen werden und
3. jede Aufgabe ist relevant, d. h. kann durchgeführt werden.

Mit dem Begriff der Umgebung ist es nun möglich, die Qualität eines einzelnen Workflow-Moduls zu untersuchen und den zentralen Begriff der *Bedienbarkeit* eines Workflow-Moduls herzuleiten: Ein Workflow-Modul heißt *bedienbar*, wenn dieses Modul Bestandteil eines „vernünftigen“ verteilten Geschäftsprozesses sein kann. Da es sich bei einem Workflow-Modul um ein Modell eines Web Service handelt, der i. Allg. mehr Verhalten anbieten kann als ein konkreter Kunde nachfragt, sind in diesem Kontext nur die ersten beiden Forderungen relevant (d. h. *Weak-Soundness*):

Definition 3.3 (Bedienbarkeit).
Sei M ein Workflow-Modul. Eine Umgebung U *bedient* das Modul M, wenn das komponierte System $M \oplus U$ weak-sound ist. Das Modul M heißt *bedienbar*, wenn es mindestens eine Umgebung U gibt, die M bedient. ⋆

Es lässt sich leicht zeigen, dass das Modul aus Abbildung 3.2(b) ein weak-sound Workflow-Netz ist. Damit sind die Module Standardplanung und Kunde jeweils bedienende Umgebungen für einander und selbst bedienbar. Wir nennen diese beiden Module daher zueinander *semantisch kompatibel*. Eine ausführliche Diskussion dieser und weiterer Eigenschaften findet sich in [Mar04]. Der folgende Abschnitt beschäftigt sich mit dem Nachweis der Bedienbarkeit für beliebige Workflow-Module.

3.1.4 Kompatibilität von Workflow-Modulen

Als Voraussetzung für die Komposition zweier Workflow-Module haben wir im Abschnitt zuvor die *syntaktische Kompatibilität* der Module gefordert. Betrachten wir ein Versandhaus, dass erst nach Erhalt der Zahlung die Ware versendet und einen Kunden, der ebenfalls mit der Zahlung wartet, bis er die Ware erhalten hat, so haben beide Systeme eine zueinander *syntaktisch kompatible* Schnittstelle (Ware und Zahlung), ihre lokalen Geschäftsprozesse passe jedoch nicht zueinander. Wie dieses Beispiel zeigt, ist diese Eigenschaft nicht ausreichend für ein vernünftiges komponiertes System ist. Deshalb benötigen wir den Begriff der *semantischen Kompatibilität* zweier Module.

Definition 3.4 (Kompatibilität, semantische).
Seien A und B zwei syntaktisch kompatible Workflow-Module. A und B heißen *semantisch kompatibel*, wenn das komponierte System $A \oplus B$ bedienbar ist. $\star$

Gemäß dieser Definition sind die Module Standard und Reiseplanung aus Abbildung 3.1 semantisch kompatibel, denn das komponierte Modul Standardplanung wird von dem Modul Kunde bedient. Die beiden Module Standardplanung und Kunde sind ebenfalls semantisch kompatibel. Das Ergebnis ihrer Komposition liefert zwar das Modell eines verteilten Geschäftsprozesses, doch auch für ein solchen Modul mit einem leeren Interface kann die Bedienbarkeit nachgewiesen werden. Eine ausführliche Diskussion der Kompatibilität findet sich in [Mar03a].

3.2 Verifikation

Dieser Abschnitt betrachtet ein Workflow-Modul mittlerer Größe: zum einen groß genug, um die interessanten Phänomene (ohne triviale Lösung) zu umfassen, und zum anderen hinreichend klein, um nach kurzen Erläuterungen überschaubar zu bleiben. Abbildung 3.3 zeigt das Modul Online Tickets.

Oftmals wird das Modell eines Web Service nicht explizit neu entworfen, sondern aus einem bestehenden Geschäftsprozessmodell abgeleitet. Wir nehmen also an, dieses Modell ist aus einem ehemals umfangreicheren Geschäftsprozess ausgeschnitten und mit einem Interface versehen worden. Daraus resultiert die unerwartet komplexe Struktur. Zum Verständnis des Modells betrachten wir eine Strukturierung nach zwei Gesichtspunkten:

Die *Geschäftsstrategie* sieht die Unterteilung der Kunden vor. Nach der Anmeldung eines Kunden mit seinem Namen schaltet entweder die Transition Standard Kunde oder die Transition Premium Kunde. Diese Entscheidung liegt im Ermessen des Web Service. Die alternativen Zweige laufen mit der Transition Reiseunterlagen senden wieder zusammen. Die *Organisation* des Online Ticket Service besteht aus drei Abteilungen (Auftragsannahme,

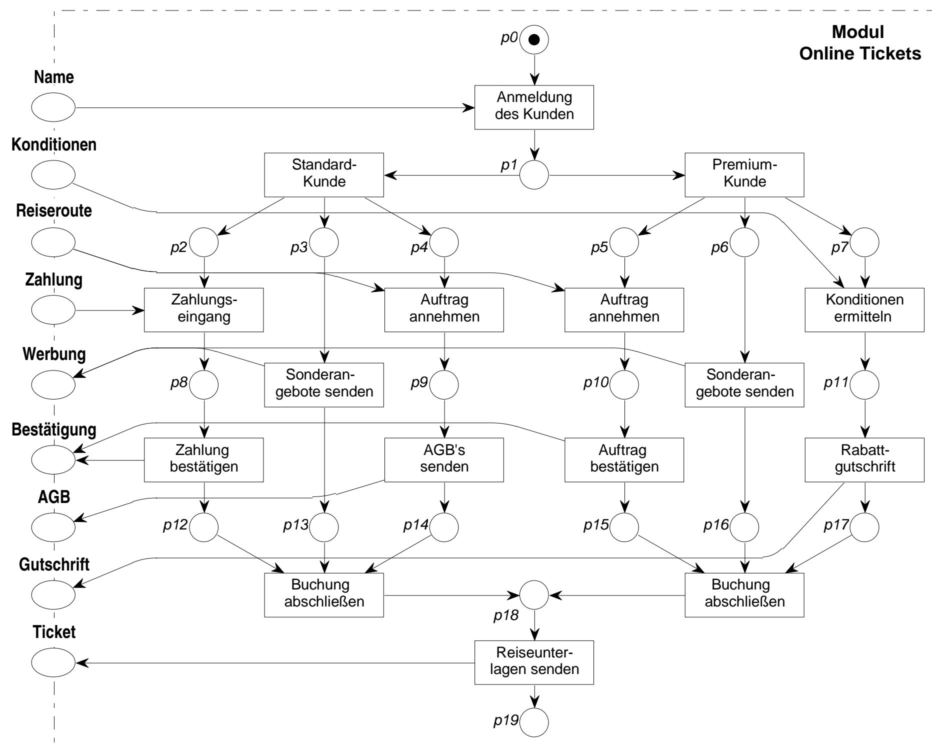


Abbildung 3.3: Komplexes Modul eines Online Ticket Service

Marketing und Buchhaltung) deren Aktivitäten weitgehend voneinander unabhängig sind. Daher hat jeder der beiden alternativen Bereiche drei nebenläufige Handlungsstränge.

Nachdem die Struktur des Moduls klar ist, stellt sich die Frage nach der Bedienbarkeit. Dazu betrachten wir folgende Umgebung: Ein Kunde sendet seinen Namen und gleichzeitig seine Konditionen, denn er wähnt sich als Premium Kunde. Aus der Struktur des Netzes geht hervor, dass beide Nachrichten unmittelbar nacheinander verarbeitet werden können, d. h. ohne weitere Kommunikation. Anschließend wartet der Kunde auf die Gutschrift. Im Modul schaltet jedoch nach der Anmeldung die Transition Standard Kunde, denn der Kunde hat lange nichts mehr bestellt und seinen Status verloren. Das Modul schickt die Werbung und wartet seinerseits auf die Reiseroute und die Zahlung. Beide Teile warten nun gegenseitig – ein typischer Deadlock. Eine für Menschen plausible Auflösung dieser Verklemmung wird an dieser Stelle nicht diskutiert, denn das gegebene Beispiel steht stellvertretend für vollautomatische Prozesse, denen kreative Strategien zur Fehlerbehandlung fehlen.

3.2.1 Nachweis der Bedienbarkeit

Es hat sich gezeigt, dass diese „naive“ Umgebung das gegebene Modul nicht bedient. Heißt das aber, dass das Modul nicht bedienbar ist oder wurde die Umgebung ungeschickt gewählt. Um die Bedienbarkeit eines Workflow-Moduls zu entscheiden, verwendet die Methode eine Datenstruktur, die dass *externe Verhalten* des Moduls, d. h. die Kommunikation mit der Umgebung explizit darstellt – den *Kommunikationsgraphen* des Moduls (*kurz:* K-Graphen).

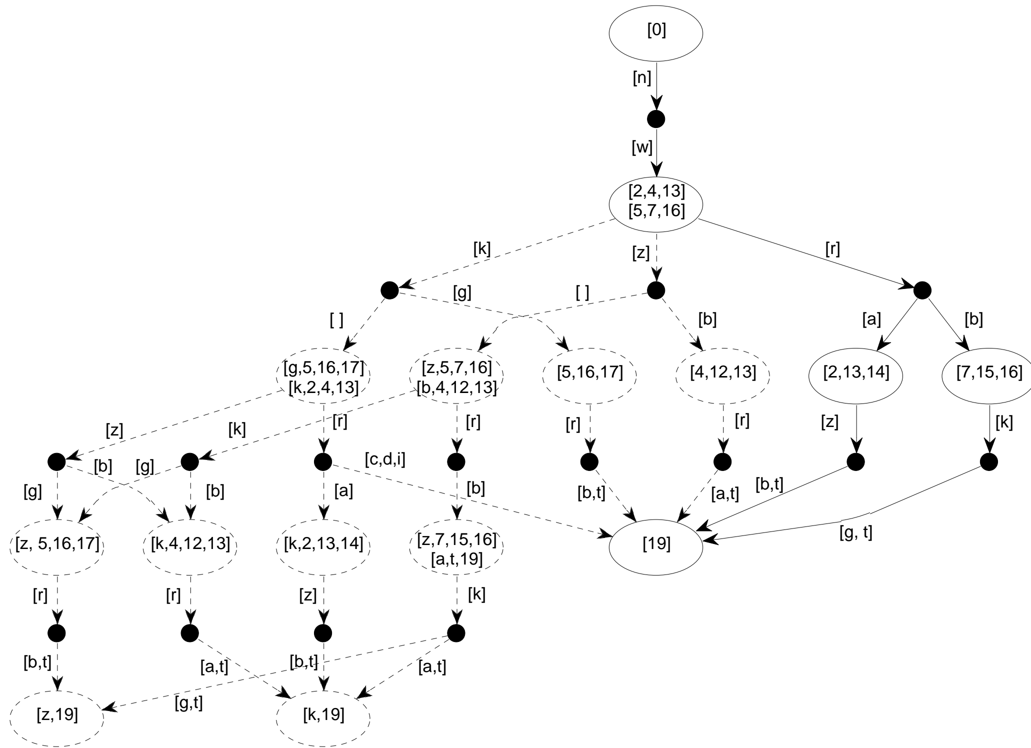


Abbildung 3.4: Kommunikationsgraph des Moduls Online Ticket

Abbildung 3.4 zeigt den K-Graphen des Online Ticket Service. Die unterschiedlichen Linien in der Darstellung werden später erläutert.

Der K-Graph besteht aus weißen und schwarzen Knoten und beschrifteten Kanten. Jeder weiße Knoten umfasst eine Menge von Zuständen des Moduls. Zu Beginn liegt im Modul Online Tickets nur eine Marke auf der Stelle p0, das heißt, das Modul befindet sich im Zustand [0] – dargestellt im Wurzelknoten des K-Graphen. Eine von einem weißen Knoten ausgehende Kante stellt eine *Eingabe* dar – eine (Multi-)Menge von Nachrichten *an* das Modul, die einerseits möglichst klein ist und andererseits ausreichend dafür ist, dass das Modul antworten kann. Im Ausgangszustand genügt der Name des Kunden ([n]). Ausgehend vom einem schwarzen Knoten sind alle daraufhin möglichen *Ausgaben vom* Modul als Kanten dargestellt. Als Antwort auf seinen Namen erhält der Kunde Werbung ([w]). Das Modul befindet sich nach diesem *Kommunikationsschritt* in einem von zwei *Folgezuständen* ([2,4,13] oder [5,7,16]). Die Konstruktion des Graphen endet, wenn das Moduls sich in einem *Endzustand* befindet.

Jeder Pfad vom Wurzelknoten zu einem Blattknoten im K-Graphen beschreibt eine Kommunikationssequenz zwischen dem Modul und einer potenziellen Umgebung. In einigen Blattknoten befinden sich Zustände, bei denen nicht alle Nachrichten aus den Kanälen entfernt wurden (z. B. [z,19]). Ein solcher Endzustände ist fehlerhaft und es gilt, die dorthin führenden Sequenzen zu vermeiden, d. h. die Umgebung muss ihre Kommunikation einschränken. In unserem Beispiel bleibt auf diese Weise nur der mit einer durchgehenden

Linie dargestellte Teilgraph übrig. Diesen nennen wir den *Bediengraphen* des Moduls (*kurz:* B-Graph). Nur wenn es so einen Teilgraphen gibt, dann ist das Modul überhaupt bedienbar. Diese Aussage ist das zentrale Theorem der Arbeit.

Theorem 3.1 (Bedienbarkeit).
Ein Workflow-Modul M ist genau dann bedienbar, wenn es einen endlichen Bediengraphen C des Moduls gibt und das komponierte System $M \oplus \Gamma(C)$ aus Modul und konstruierter Umgebung stets terminieren kann. ⋆

Alle Einzelheiten und der Beweis finden sich in [Mar04]. Neben dem Nachweis der Bedienbarkeit an sich, ist der B-Graph die Bedienungsanleitung eines Moduls. Bezogen auf unser Beispiel sollte der Kunde zuerst seinen Namen ([n]) senden, die Werbung ([w]) empfangen und anschließend die Reiseroute ([r]) übermitteln. Ist er ein Premium Kunde, dann erhält er als Antwort eine Bestätigung ([b]). Anderenfalls erhält er die AGB's ([a]) des Online Ticket Service. Mit diesem Wissen kann er die weitere Kommunikation ohne Probleme fortsetzen.

3.2.2 Abstraktion

Mit der Bedienbarkeit eines Workflow-Moduls und der Kompatibilität zweier Workflow-Modulen haben wir Begriffe und Verfahren entwickelt, um die Ergebnisse der Modellierung von Web Services zu analysieren. In diesem Abschnitt wird ein Verfahren vorgestellt, das direkt den Prozess der Modellierung unterstützt und damit den Modellierer von aufwändigen und fehleranfälligen Aufgaben entlastet – die Generierung eines abstrakten Prozessmodells oder auch *Public View Generation*.

Ausgangspunkt für die Überlegungen waren zwei gegensätzliche Beobachtungen. Zum einen ist Modellierung teuer, das heißt, die Abbildung eines realen Geschäftsprozesses in ein Prozessmodell ist eine zeitaufwändige Tätigkeit, die geeignete Wahl des Abstraktionsniveaus erfordert ein hohes Maß an Erfahrung, der Abgleich zwischen Modell und Realität ist schwierig, und fehlende Informationen, unklare Strukturen und implizite Zusammenhänge verursachen häufig schwerwiegende Fehler. In vielen Fällen liegt aufgrund vorhandener Workflow-Management-Systeme in den Unternehmen bereits ein Modell des Geschäftsprozesses vor. Aufgrund der maschinellen Unterstützung und Überwachung der Abläufe hat sich dieses Modell als ein adäquates Abbild der Realität erwiesen. Es ist daher nahe liegend, dieses Modell gemeinsam mit dem Web Service zu veröffentlichen.

Zum anderen werden an das Prozessmodell eines Web Service gewisse Anforderungen gestellt. Gemäß der Definition 2.1 muss ein Web-Service selbsterklärend sein, d. h. ohne eine zusätzliche Bedienungsanleitung verstanden und benutzt werden können. Ein automatisch überführtes Modell ist jedoch in den meisten Fällen überfrachtet mit unnötigen Details und damit zu umständlich (siehe Abbildung 3.3). Darüber hinaus gelten die konkreten Prozessmodelle in vielen Unternehmen als vertraulich und dürfen damit nicht in dieser Form publiziert werden.

Das Ziel ist eine weitgehend automatische Überführung einer gegebenen detaillierten Darstellung des Prozesses – bezeichnet als *Private-View-Modell* – in eine geeignete abstrakte Darstellung des extern sichtbaren Verhaltens – bezeichnet als *Public-View-Modell*. Abbildung 3.5 verdeutlicht das entwickelte Verfahren zur Überführung:

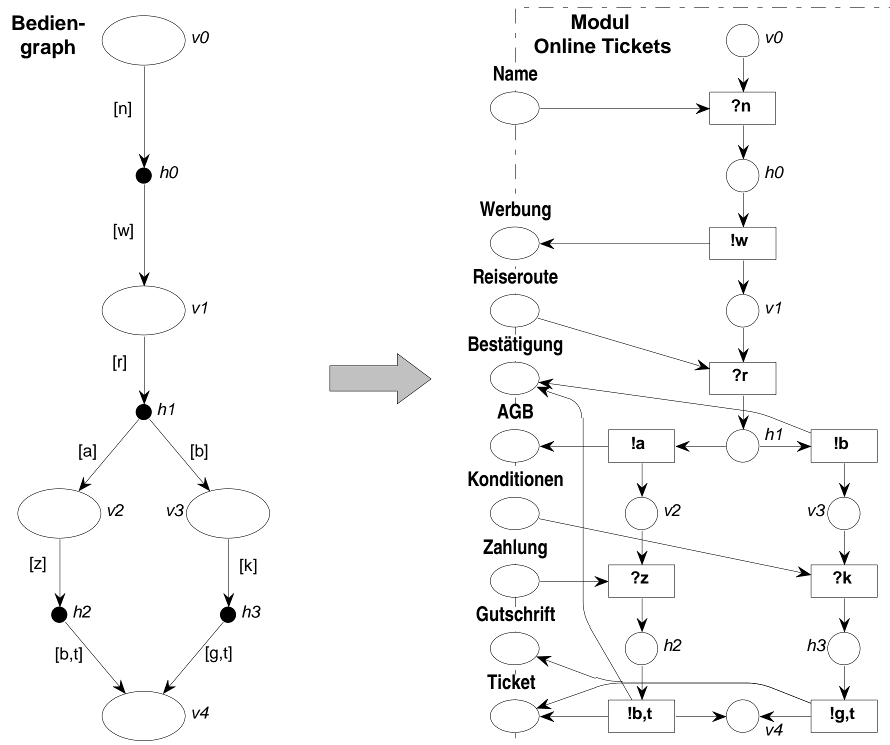


Abbildung 3.5: Vereinfachung des Moduls Online Ticket

Ein Modellierer analysieren vor der Veröffentlichung seines Web Service die Bedienbarkeit, das heißt, er konstruiert den Bediengraphen. Es ist daher nahe liegend, dass der Modellierer diese Information nutzt, um daraus ein Public-View-Modell zu erzeugen. Das auf diese Weise entstandene abstrakte Modul des Online Ticket Service hat die Struktur des Bediengraphen des Originals. Es ist leicht zu zeigen, dass sich das durch Transformation entstandene Modul nach außen hin genau so verhält wie das Original. Durch die Veröffentlichung des abstraktes Moduls anstelle des Originals bleiben dem potenziellen Benutzer jedoch missverständliche Strukturen erspart.

3.3 Zusammenfassung

Web Services sind ein geeigneter Ansatz für verteilte Geschäftsprozesse. Die Technologien im Umfeld von Web Services befinden sich jedoch zum großen Teil noch im Prozess der Standardisierung und sind starken Änderungen unterworfen. Petrinetze sind ein geeigneter Formalismus, um sich konzeptionell mit diesen Anforderungen und Problemen zu befassen. Neben ihrer anschaulichen graphischen Repräsentation, gründen Petrinetze auf eine reichhaltige Theorie der verteilten Systeme, insbesondere der *verteilten Geschäftsprozesse.*

Petrinetze fokussieren auf die Konzepte des Anwendungsgebietes. Ein *Workflow-Modul* bildet einen komplexen Web Service ab. Dabei besteht ein Workflow-Modul analog zum Web Service aus einem Workflow-Netz als Prozessmodell und einer Menge von Schnittstellen

als Interface. Zwei Workflow-Module werden durch Fusion der Schnittstellen miteinander komponiert. Voraussetzung dafür ist die *syntaktische Kompatibilität.*

Mit der entwickelten Methode ist es möglich, die *Bedienbarkeit* – eine essenzielle Qualitätseigenschaft von Web Services – effektiv und konstruktiv nachzuweisen. Dieser Artikel versteht sich als ein informaler Überblick der Methode. Alle notwendigen Begriffe sind in [Mar04] ausführlich hergeleitet und die Algorithmen präzise definiert sowie ihre Korrektheit formal bewiesen. Darüber hinaus finden sich dort Betrachtungen zur Laufzeitkomplexität und eine Klassifikation von Spezialfällen, die eine vereinfachte Analyse ermöglichen, darunter syntaktische Richtlinien, die Bedienbarkeit per Konstruktion garantierten. Die Handhabbarkeit der Methode ist durch die prototypische Implementierung der Analyse belegt [Mar03b].

Neben der Analyse eines einzelnen Web Service stehen Beziehungen zwischen verschiedenen derartigen Komponenten im Blickpunkt: Als Voraussetzung für die Komposition von Web Services wird in dieser Arbeit die *semantische Kompatibilität* der Komponenten gefordert. Zwei Workflow-Module sind semantisch kompatibel, wenn das komponierte System bedienbar ist. Wesentlich für den dynamischen Aufbau verteilter Geschäftsprozesse ist die *Austauschbarkeit* von Komponenten. Diese Eigenschaft wurde auf eine Simulationsbeziehung zwischen deren Kommunikationsgraphen zurückgeführt und ist somit ebenfalls effektiv entscheidbar. Ein wichtiger Arbeitsschritt vor der Veröffentlichung eines Web Service ist die *Abstraktion* von internen Details. Ein vereinfachtes Modell eines Web Service lässt sich direkt aus dem Bediengraphen des Moduls ableiten.

Weitere Forschungen zielen zum einen auf die Adaption der Methode auf eine konkrete Syntax – insbesondere auf die praxisrelevante Sprache BPEL4WS (siehe Kapitel 4). Zum anderen ist die Verbesserung der Analysemethoden ein wichtiges Anliegen, um dem Problem der Zustandsraumexplosion durch geeignete Reduktionstechniken zu begegnen (siehe Kapitel 5).

4 Eine Petrinetz-Semantik für BPEL4WS

Autor: Christian Stahl

4.1 Ansatz der Semantik

In diesem Kapitel geben wir einen Einblick in unser Konzept einer Petrinetz-Semantik für die *Business Process Execution Language for Web Services* (BPEL4WS) geben. Dazu klären wir nachfolgend die Frage, warum eine Semantik für BPEL4WS benötigt wird. In den Abschnitten 4.1.2 und 4.1.3 präsentieren wir die Idee der Übersetzung sowie einen Vorschlag zur Automatisierung der Transformation. Anhand von Beispielen veranschaulichen wir in den Abschnitten 4.2 und 4.3 die Übersetzung von BPEL4WS in Petrinetze. Einen Ausblick auf die gesamte Übersetzung der Sprache BPEL4WS geben wir abschließend in Abschnitt 4.4.

4.1.1 Warum eine Semantik?

In der Einführung wurde begründet, warum die Web Service-Technologie in absehbarer Zeit eine wichtige Rolle spielen wird und dass BPEL4WS auf dem Weg ist, ein de facto Standard zur Beschreibung von Web Services zu werden. Wir wollen in diesem Abschnitt motivieren, warum es sinnvoll ist, erstens eine Semantik für BPEL4WS zu entwerfen und zweitens dazu Petrinetze zu verwenden.

Im Jahr 2002 ist die Sprache BPEL4WS aus der Vereinigung der *Web Service Flow Language* (kurz: *WSFL*) [Ley01] von IBM und *XLANG* [Tha01] von Microsoft hervorgegangen. BPEL4WS kombiniert sowohl graphbasierte Kontrollstrukturen aus WSFL als auch blockbasierte Kontrollstrukturen aus XLANG. Zusätzlich wurde die Sprache um Konstrukte für die Fehlerbehandlung und die Kompensation von Prozessteilen erweitert. Damit stehen der Sprache mächtige Konzepte zur Verfügung. In [WADH02] heben die Autoren die Ausdrucksstärke von BPEL4WS hervor, stellen aber auch zwei Nachteile fest: Zum einen sei die Sprache sehr komplex und besitze zu viele überlappende Konstrukte. Als zweites kritisieren die Autoren die nicht immer klare Semantik der Sprache. Für BPEL4WS existiert bis zum jetzigen Zeitpunkt keine formale Semantik. Die Spezifikation [ACD+02] beschreibt die Sprache und ihre Konzepte anhand von Beispielen lediglich in textueller Form und legt mittels einer XML-Schema-Definition die Syntax von BPEL4WS fest.

Es ist daher das Ziel unserer Forschungen, eine vollständig formale Semantik für BPEL4WS auf Basis von Petrinetzen zu entwickeln. Petrinetze [Rei85] sind ein ausdrucksstarker mathematischer Formalismus. Aufgrund der graphischen Repräsentation sind Petrinetze leicht erlernbar und verständlich [Aal98]. Weiterhin existieren für Petrinetze viele Algorithmen zur formalen Analyse, wie Verifikation (z.B. im Bereich der Web Services [Mar04]) und Model Checking (z.B. [Sch00a]). Deshalb verwenden wir diesen Formalismus zum Erstellen der BPEL4WS-Semantik.

Im nächsten Abschnitt erklären wir die Idee, wie man BPEL4WS in Petrinetze transformieren kann.

4.1.2 Von BPEL4WS zu Petrinetzen

Wir suchen nach einem Ansatz, um die Sprache BPEL4WS in Petrinetze zu überführen. Unser Ziel ist es, jeden beliebigen in BPEL4WS spezifizierten Prozess in ein Petrinetz übersetzen zu können. Dazu orientieren wir uns am Aufbau von BPEL4WS. Jeder BPEL4WS-Prozess entsteht durch das Zusammenstecken von Aktivitäten. Deshalb übersetzen wir als erstes jede Aktivität der Sprache in ein Petrinetz, welches wir im Folgenden als *Muster* der entsprechenden Aktivität bezeichnen wollen. In Analogie zu den Aktivitäten muss ein Muster mit anderen Mustern zusammengesteckt werden können. Um diese Eigenschaft zu garantieren, benötigt jedes Muster eine definierte *Schnittstelle*. Weiterhin müssen die Muster auch parametrisiert werden können, denn in BPEL4WS gibt es Aktivitäten, die aus Konstrukten bestehen, die durch Parameter bestimmt werden. Unter einem Konstrukt verstehen wir in diesem Zusammenhang beispielsweise eine `Variable`[1] oder eine innere Aktivität. Das entsprechende Muster muss diese Aktivität für jede vorgegebene Anzahl von `Variablen` bzw. inneren Aktivitäten abbilden können, wobei der Parameter die Anzahl angibt. Beim Zusammenstecken und Schachteln der Aktivitäten dürfen keine Verklemmungen auftreten, die im entsprechenden BPEL4WS-Prozess nicht ebenfalls vorkommen. Mit anderen Worten, die Übersetzung muss eigenschaftserhaltend sein. Damit bewahren wir die Äquivalenz zwischen BPEL4WS und dem Petrinetz-Modell. Die Sammlung aller Muster bildet die *Petrinetz-Semantik* für BPEL4WS.

Wir haben die eben geschilderte Idee umgesetzt und jede BPEL4WS-Aktivität in ein Muster übersetzt. Damit sind wir in der Lage, durch Schachteln und Zusammenstecken der Muster jeden in BPEL4WS spezifizierten Prozess in ein Petrinetz abzubilden. Um in einem weiteren Schritt den als Petrinetz vorliegenden Prozess analysieren zu können, benötigen wir ein geeignetes Datenformat, das den Analyse-Werkzeugen als Eingabeformat dient. Dieses Datenformat soll uns helfen, die Transformation von BPEL4WS nach Petrinetze zu automatisieren. Im nächsten Abschnitt stellen wir das von uns verwendete Datenformat PNML und den Automatisierungsansatz vor.

4.1.3 Automatisierte Übersetzung

Ein geeignetes Datenformat für Petrinetze wird benötigt, denn die Algorithmen für die Analyse arbeiten auf den Datenstrukturen der Netze. Wir verwenden die *Petri Net Markup Language* (kurz: *PNML*) [WK03], ein XML-basiertes Austauschformat für Petrinetze, das sich zur Zeit in der Standardisierung befindet. Die Grundlage für jedes in PNML notierte Petrinetz bildet die *Petri Net Type Definition* (kurz: *PNTD*), die die Eigenschaften des entsprechenden Petrinetz-Typen beinhaltet. Es ist außerdem mit PNML möglich, ein Petrinetz aus Modulen zusammenzusetzen (siehe [KW01]). Ein Modul besitzt eine Schnittstelle, auf die außerhalb des Moduls zugegriffen werden kann. Die Schnittstelle ermöglicht das Zusammenstecken von Modulen zu einem Petrinetz.

[1]Diesen TypeWriter-Schrifttyp verwenden wir für alle BPEL4WS-Bezeichner.

Schauen wir uns nun den automatisierten Übersetzungsansatz eines BPEL4WS-Prozesses in ein Petrinetz an. Ausgehend von den Mustern definieren wir einen Netztyp und erstellen für ihn eine PNTD. Im nächsten Schritt überführen wir jedes Muster in ein PNML-Modul von diesem Netztyp.

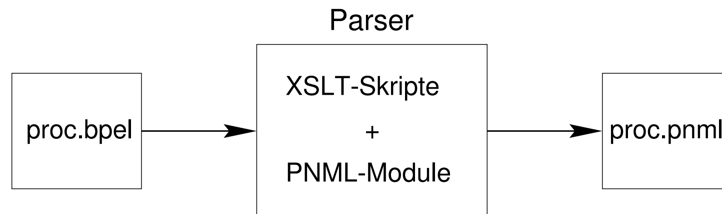


Abbildung 4.1: Transformation von BPEL4WS nach PNML

Das Zusammenstecken sowie das Schachteln der Module ermöglicht die Schnittstelle der Module. Sowohl PNML als auch BPEL4WS sind XML-basiert. Aus diesem Grund können wir für die Übersetzung eines BPEL4WS-Prozesses in ein Petrinetz im PNML-Datenformat die Transformationssprache *Extensible Stylesheet Language Transformations* (kurz: *XSLT*) [Cla99] verwenden. Abbildung 4.1 veranschaulicht den Ablauf der Übersetzung:

Die linke Box zeigt den in BPEL4WS spezifizierten Prozess proc.bpel[2], der dem Parser als Eingabe gegeben wird. Symbolisiert wird der Parser durch die mittlere Box. Für die Übersetzung des Prozesses benötigt man die PNML-Module und den Parser selbst. Letzterer besteht aus einer Sammlung von XSLT-Skripten. Informal beschrieben, wird bei der Transformation jede Aktivität in proc.bpel durch ein entsprechendes PNML-Modul ersetzt. Die einzelnen Module werden über ihre Schnittstelle miteinander verknüpft. Als Ergebnis erhält man ein Petrinetz proc.pnml in PNML-Format. Das soll die rechte Box in der Abbildung verdeutlichen.

Im nächsten Abschnitt zeigen wir den allgemeinen Aufbau der Muster am Beispiel des Musters für die Aktivität `receive`. Darauf aufbauend stellen wir die Übersetzung der `link`-Semantik und der `Dead-Path-Elimination` vor. Im letzten Abschnitt geben wir einen Ausblick auf die gesamte Übersetzung der Sprache BPEL4WS.

4.2 Übersetzung einer Aktivität

In diesem Abschnitt erläutern wir am Beispiel der BPEL4WS-Aktivität `receive` den Grundaufbau und die graphische Notation der Muster. Wir werden dazu nur kurz die Funktion des `receive` erläutern, nähere Informationen sind der Spezifikation [ACD$^+$02] oder dem Tutorial [DK02] zu entnehmen.

4.2.1 Aktivität am Beispiel: receive

Die Aktivität `receive` wartet auf eine von einem anderen Web Service gesendete Nachricht. Sie spezifiziert, über welchen Nachrichtenkanal die Nachricht empfangen wird und in welcher

[2] Wir verwenden diesen serifenlosen Schrifttyp, wenn wir uns auf Bezeichnungen innerhalb von Abbildungen beziehen.

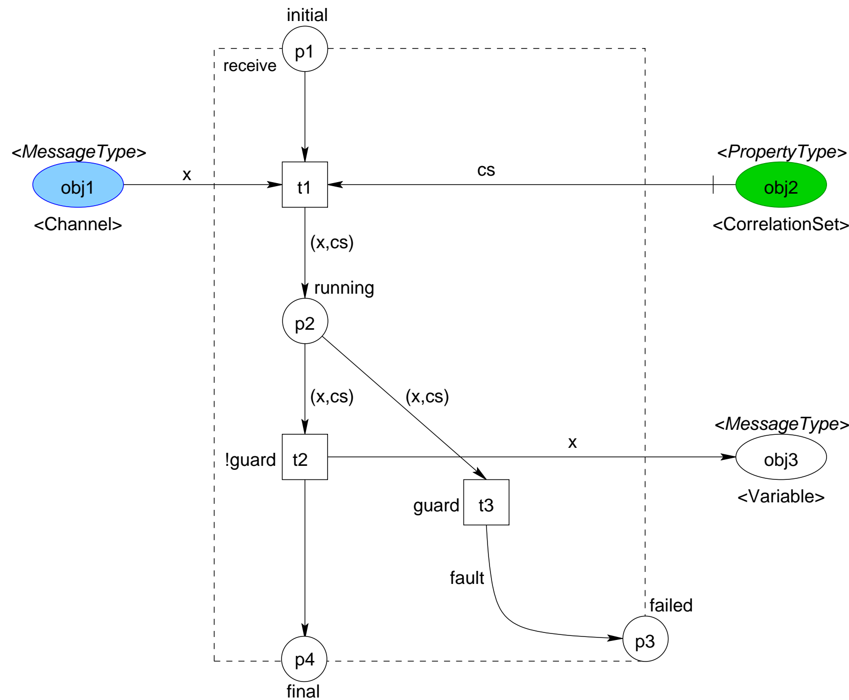


Abbildung 4.2: receive-Muster

`Variable` der Nachrichteninhalt abgelegt wird. Der Nachrichtenkanal beschreibt seinerseits den Typ der Nachricht. Da ein BPEL4WS-Prozess in Instanzen ausgeführt wird, muss jede Nachricht eindeutig einer Prozessinstanz zuordenbar sein. Zu diesem Zweck erhält jede Nachricht ein `CorrelationSet`, das heißt, eine die Instanz identifizierende ID. Bei der Ausführung der `receive`-Aktivität können Standardfehler auftreten.

Im Folgenden erläutern wir das allgemeine Schema der Muster am Beispiel des in Abbildung 4.2 dargestellten `receive`-Musters.

Wir verwenden die gängige graphische Notation für Petrinetze (vgl. [Rei85] und [WWV+97]): Kreise und Ellipsen symbolisieren die Plätze, Quadrate die Transitionen und Pfeile die Kanten. Zusätzlich verwenden wir auch Beschriftungen in unseren Mustern. Dazu gehören in erster Linie die *Bezeichner*. Jeder Transition und jedem Platz wird dabei ein Bezeichner zugeordnet, der sich graphisch im Inneren des Platzes bzw. der Transition befindet, z.B. p1 oder t1. Aufgrund seiner Eigenschaft - ein Bezeichner ist immer eindeutig im Muster - hilft er, auf das betreffende Element im Petrinetz zu verweisen. Es gibt ausgezeichnete Plätze und Transitionen, die einen sprechenden Namen besitzen, wie initial oder final. Dieser Bezeichner soll helfen, die Aufgabe des Elementes im Petrinetz zu verstehen und ist in der Graphik außerhalb des entsprechenden Elementes angeordnet. Weiterhin sind auch einige Kanten beschriftet. Dabei handelt es sich um Variablen im Sinne des Petrinetzes, die im Zuge des Schaltens mit Werten belegt werden, z.B. die einfachen Variablen `x` und cs sowie das Tupel von Variablen (x, cs). Die letzte Form einer Beschriftung, die wir hier vorstellen wollen, ist die *Aktivierungsbedingung* [WWV+97], ein boolescher Ausdruck, der zu einer Transition gehört und dementsprechend graphisch auch außerhalb dieser Tran-

sition angeordnet ist. Um eine Aktivierungsbedingung von den restlichen Beschriftungen zu unterscheiden, schreiben wir sie in geschweifte Klammern, z.B. {guard}. Auf die genaue Funktion der Aktivierungsbedingung gehen wir an geeigneter Stelle noch einmal genauer ein.

Ein Muster ist durch eine gestrichelte Linie eingerahmt. Dieser Rahmen grenzt das Muster von seiner Umgebung ab. Innerhalb des Rahmens befindet sich die Struktur des Musters. Sie besteht in Abbildung 4.2 aus drei Transitionen (t1, t2, t3) und einem Platz (p2). Auf dem Rahmen liegt die Schnittstelle zu angrenzenden bzw. umgebenden Mustern. Jedes Muster hat einen Startplatz (initial) und in den meisten Fällen einen Endplatz (final). Der Kontrollfluss verläuft beginnend bei initial vertikal von oben nach unten. Im `receive`-Muster endet er entweder in final oder in failed. Kann die entsprechende Aktivität des Musters einen Fehler erzeugen, dann existiert wie in Abbildung 4.2 eine Schnittstelle failed, über die der aufgetretene Fehler zum zuständigen `Fault Handler` weitergereicht wird.

Außerhalb des Rahmens liegen Objekte, die global bezüglich des umgebenden `scopes` sind. Diese Objekte sind graphisch als Highlevel-Plätze, das heißt, durch Ellipsen dargestellt. Das `receive`-Muster besitzt drei Objekte: obj1, obj2 und obj3. Oberhalb eines solchen Platzes ist die *Sorte* des Objektes [WWV+97] definiert. Eine Sorte ist eine Menge, aus der die Marken stammen, die auf dem Platz liegen und ankommen. Unterhalb der Ellipse steht die Definition, welche *Rolle* das Objekt im Petrinetz einnimmt. Die Rolle ist dabei unabhängig von der Sorte des Objektes. Zur graphischen Veranschaulichung steht die Sorte immer innerhalb von spitzen Klammern und die Rolle wird in eckige Klammern eingefasst. obj1 ist von der Sorte <MessageType> und nimmt im Muster die Rolle [Channel] also den Nachrichtenkanal ein. Das zweite Objekt enthält Marken der Sorte <PropertyType> und hat die Rolle [CorrelationSet]. Mit obj3 modellieren wird eine `Variable` ([Variable]), wobei die Marken auf dem Platz von der Sorte <MessageType> sind. In der BPEL4WS-Prozessbeschreibung werden die Objekte zu Beginn des Prozesses bzw. sogar schon vor dem Prozess definiert. Aus diesem Grund bilden wir sie als globale Elemente ab. Die Kommunikation verläuft horizontal, z.B. von obj1 zu t1.

Als nächstes erläutern wir den Ablauf des `receive`-Musters. Dabei gehen wir auch auf die Werte ein, die eine Variable annehmen kann.

4.2.2 Der Ablauf des receive-Musters

Wir modellieren das `receive`-Muster als Petrinetz in Abbildung 4.2 wie folgt: Die Aktivität empfängt eine Nachricht (t1) und schreibt entweder den Nachrichteninhalt in die `Variable` (t2) oder es tritt ein Fehler auf (t3). Schauen wir uns einen Ablauf des Netzes genauer an:

Ist die entsprechende Aktivität im BPEL4WS-Prozess aktiviert, liegt in Abbildung 4.2 eine Marke auf p1. Zu Prozessbeginn wird ein `CorrelationSet` erstellt, indem auf dem Platz obj2 eine Marke vom Typ <PropertyType> liegt[3]. In BPEL4WS wird von einem `CorrelationSet` nur gelesen. Somit sollte im Petrinetz die Marke auf dem Platz liegen bleiben. Wir setzen diese Anforderung mittels einer Lesekante [Web02] zwischen obj2 und t1 um, symbolisiert durch einen kleinen senkrechten Strich am Anfang der Kante. Wie der

[3] Die zweite Möglichkeit, dass das CorrelationSet erst während des Prozesses initialisiert wird, betrachten wir in dieser Arbeit nicht.

Name schon sagt, wird beim Schalten die Marke des Platzes obj2 nur gelesen und nicht konsumiert.

Beim Ablauf des receive-Musters unterscheidet man zwei Szenarien, die in Abhängigkeit von der Auswertung der Aktivierungsbedingung {guard} eintreten. Wird sie positiv ausgewertet und die Transition ist konzessioniert, dann ist die Transition aktiviert und kann schalten. Die Aktivierungsbedingung {guard} beschreibt einen Fehlerfall.

Im ersten Szenario liegt auf dem Nachrichtenkanal eine Nachricht und wie oben erwähnt, ist auch das CorrelationSet markiert. Die Variablen x und cs werden mit dem Nachrichteninhalt bzw. dem CorrelationSet belegt. Nach dem Schalten von t1 wird {guard} evaluiert. Der boolesche Ausdruck ist von cs und x abhängig, aber nicht von der der Variablen v, die mit dem Inhalt von obj3 belegt wird. Wird {guard} positiv ausgewertet, das heißt, ein Fehler tritt bei der Abarbeitung des receive auf, schaltet t3. Dabei wird der Variablen fault der Name des aufgetretenen Fehlers z.B. correlationViolation zugewiesen und auf den Platz failed produziert. Damit endet das erste Szenario.

Wird hingegen {guard} zu falsch ausgewertet, das heißt, {!guard} wird positiv ausgewertet, schaltet t2 und produziert eine Marke auf final. Weiterhin wird die Variable x mit dem Nachrichteninhalt belegt und auf obj3 gelegt. Beim Schreiben einer Variable wird deren Inhalt durch den neuen Wert ersetzt. In unserem Modell setzen wir das mit der Schlinge an t2 um.[4] Dies ist das zweite Szenario des Musters. In beiden Fällen ist das receive abgearbeitet worden.

4.3 Übersetzung der link-Semantik

Im letzten Abschnitt haben wir am Beispiel der Aktivität receive die Transformation in ein Petrinetz vorgestellt. Dabei haben wir auch die Schnittstelle einer Aktivität beschrieben. Davon ausgehend stellen wir in diesem Abschnitt die Übersetzung der link-Semantik und der Dead-Path-Elimination in Petrinetze vor.

4.3.1 link-Semantik

In diesem Abschnitt geben wir eine kurze Einführung in das Konzept der links sowie Dead-Path-Elimination (kurz: DPE) der Sprache BPEL4WS. Für mehr Informationen sei wieder auf die Spezifikation [ACD+02] und ein Tutorial [CDK03] verwiesen.

BPEL4WS definiert mit der Aktivität flow ein Konstrukt, das aus einer endlichen Anzahl von inneren Aktivitäten besteht, die nebenläufig ausgeführt und am Ende wieder synchronisiert werden. Innerhalb eines flow kann mit Hilfe von links zusätzlich eine Ordnung, mit anderen Worten, eine Synchronisation zwischen den Aktivitäten, definiert werden. Ein link hat eine source- und eine target-Aktivität, die beide über den link miteinander verbunden sind. Dabei bildet die source-Aktivität den Startpunkt und die target-Aktivität das Ende. Eine Aktivität ist nicht sowohl source als auch target eines links, allerdings erlaubt die Spezifikation, dass eine Aktivität source und target beliebig vieler unterschiedlicher links sein darf. Die source-Aktivität eines links kann dessen Wert durch einen booleschen Ausdruck – die transitionCondition – spezifizieren. Die target-Aktivität

[4] Zu Prozessbeginn werden alle Variablen mit einem Nullwert vorinitialisiert.

dagegen definiert ihre Aktivierung über die joinCondition. Im Standardfall verlangt diese, dass mindestens ein eingehender link den Wahrheitswert true liefert.

Wird in einem BPEL4WS-Prozess eine Aktivität beendet, welche source eines links ist, wird der link aktiviert und der Wert durch die Auswertung der transitionCondition ermittelt. Damit steht der Status des links fest. Die dazugehörige target-Aktivität beginnt die Abarbeitung erst, wenn der Status aller eingehenden links feststeht und die joinCondition ausgewertet wurde. Erfolgt die Auswertung zu false, wird die Aktivität nicht abgearbeitet und die transitionCondition jedes ausgehenden links wird auf false gesetzt. Dieses Verhalten bezeichnet man als Death-Path-Elimination. Mit anderen Worten, man propagiert das Scheitern der Ausführung einer Aktivität an alle target-Aktivitäten. Das setzt sich solange fort, bis eine joinCondition wieder zu true ausgewertet wird. Man erreicht damit, dass die target-Aktivitäten von dem Fehler Kenntnis bekommen und durch die Auswertung der joinCondition darauf geeignet reagieren können.

Nachfolgend stellen wir nacheinander die Muster für eine target-Aktivität und eine source-Aktivität eines links vor. Abschließend repräsentieren wir das Muster einer Aktivität, die sowohl source- als auch target-Aktivität ist.

4.3.2 Muster für die target-Aktivität

Betrachten wir das Muster für die target-Aktivität eines links genauer. Abbildung 4.3 stellt eine beliebige Aktivität X dar, die target zweier links (inLink1 und inLink2) ist. Die äußere gestrichelte Linie rahmt das link-Muster ein, die innere die eingebettete Aktivität X. Von letzterer ist nur die Schnittstelle sichtbar, die Struktur dagegen ist versteckt. Die Schnittstelle besteht aus zwei Plätzen - initial (p4) und final (p5). Letzterer ist zugleich der final-Platz der link-Aktivität. Das Konzept des Verschmelzens von Schnittstellen geht auf die Arbeit von Kindler und Weber [KW01] zurück. Wir verwenden es, um das Netz nicht unnötig zu vergrößern, denn alternativ hätte man die beiden final-Plätze sequentiell mit einer Transition verbinden können.

Wenden wir uns jetzt der link-Aktivität zu. Ihre Schnittstelle besteht aus den Plätzen initial (p3) und final (p5). Außerhalb des Musters liegen die beiden link-Plätze p1 und p2. Auch wenn sich die beiden Plätze außerhalb des Rahmens befinden, sind es keine Objekte. Ein Ablauf für das Muster sieht jetzt wie folgt aus:

Abhängig von der Auswertung der Aktivierungsbedingung {joinCond} betrachten wir zwei Szenarien. Der Ausdruck {joinCond} modelliert die joinCondition, die durch die Prozessbeschreibung gegeben ist. Zu ihrer Auswertung werden die Marken der beiden links, die beide vom Typ Boolean sind, herangezogen. Somit werden die beiden Variablen b1 und b2 immer mit einem Wahrheitswert belegt. Der Wert der Marke auf p1 bzw. p2 ist durch die transitionCondition des jeweiligen links erzeugt worden, siehe dazu Abschnitt 4.3.3. Im ersten Szenario sind die Plätze p1, p2 und p3 markiert. Damit modellieren wir, dass im Prozess der Status aller eingehenden links bekannt ist und die Aktivität zur Ausführung bereit ist. Wenn die Belegung von b1 und b2 den booleschen Ausdruck {joinCond} zu wahr evaluiert, wird eine Marke auf p4 produziert und die innere Aktivität ausgeführt. Wenn bei deren Abarbeitung kein Fehler auftritt, gelangt die Kontrollflussmarke auf den Platz final.

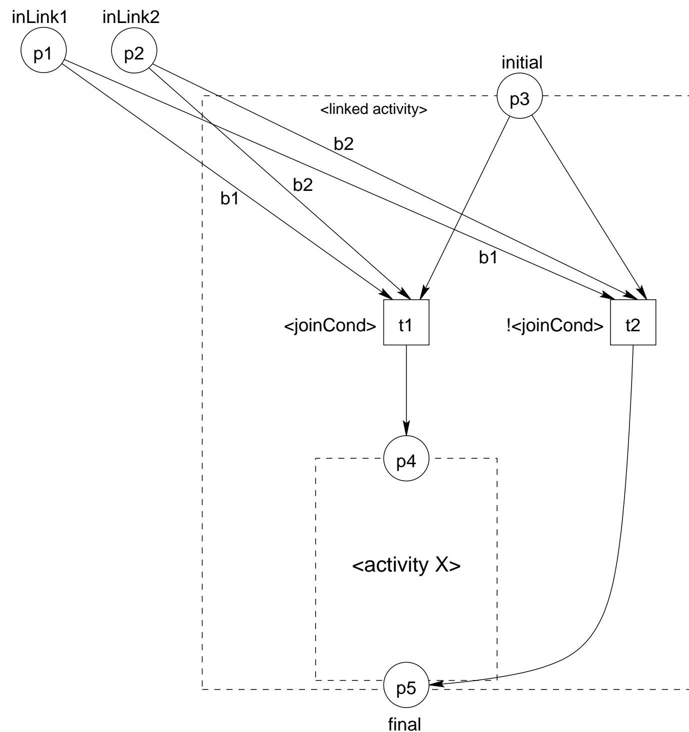


Abbildung 4.3: Muster einer Aktivität X, die target zweier links ist

Ergibt die Evaluierung der `joinCondition` {joinCond} falsch, wird die Aktivität X nicht ausgeführt.

4.3.3 Muster für die source-Aktivität

Als nächstes schauen wir uns das Muster für die `source`-Aktivität in Abbildung 4.4 an. Der äußere Rahmen begrenzt die `link`-Aktivität, der innere die innenliegende Aktivität X. Sowohl X als auch die `link`-Aktivität haben als Schnittstelle je drei Plätze. Das sind initial, final und negLink. Die gepunktete Linie zwischen p2 und p4 symbolisiert, dass die beiden Plätze verschmolzen werden.

Wird beispielsweise die Bedingung in einem `switch`-Zweig zu `false` ausgewertet, dann wird die Aktivität des Zweiges nicht ausgeführt. Dafür werden aber die `transitionConditions` aller ausgehenden `links` der Aktivität, wie auch die ihrer (eventuell vorhandenen) inneren Aktivitäten auf `false` gesetzt. Wir setzen diese Anforderung mit Hilfe des negLink-Platzes um. In Abbildung 4.4 propagieren wir mit dem negLink-Platz ein `false` an die beiden ausgehenden `links` und produzieren eine Marke auf p3. Dieser Platz ist aber nur in der Schnittstelle enthalten, wenn X das Muster einer strukturierten Aktivität ist, die eine `source`-Aktivität enthält. Letzteres nehmen wir für unser Beispiel an. Die Muster aller Basisaktivitäten sowie von strukturierten Aktivitäten, die keine innere `source`-Aktivität enthalten, besitzen keinen negLink-Platz.

Außerhalb der `link`-Aktivität liegen die beiden Plätze der ausgehenden `links` outLink1 und outLink2. Diese beiden Plätze sind genau die Eingangsplätze des Musters für die ent-

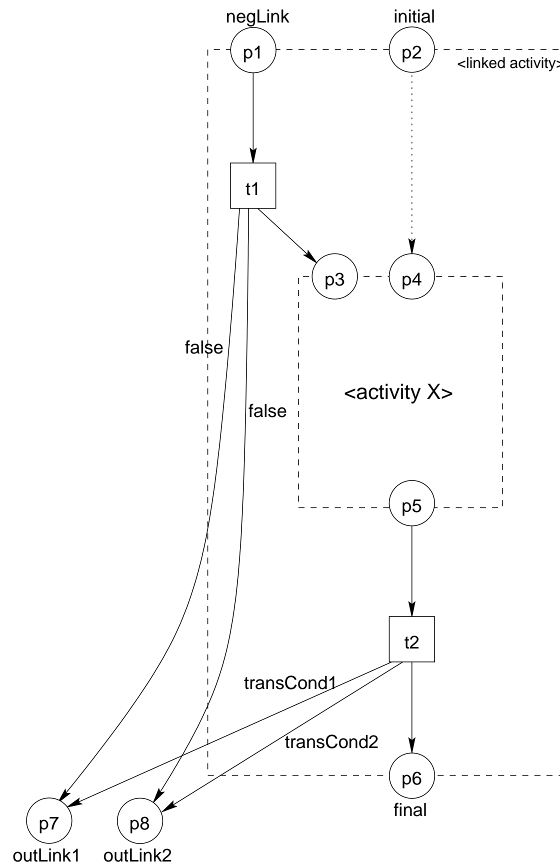


Abbildung 4.4: Muster einer Aktivität X, die source zweier links ist

sprechende `target`-Aktivität (vgl. p1 und p2 in Abbildung 4.3). Sie sind beide von der Sorte Boolean, das heißt, outLink1 und outLink2 werden immer mit einem Wahrheitswert belegt. Der Wahrheitswert ergibt sich aus der Auswertung der `transitionCondition` des jeweiligen `links`. Schauen wir uns die beiden möglichen Szenarien an:

Alternativ liegt auf p1 oder p2 eine Marke. Der andere Fall, dass beide Plätze markiert sind, tritt nicht ein. Beginnen wir mit dem Szenario des positiven Kontrollflusses. Auf p2 liegt eine Marke und aufgrund des Verschmelzens von p2 und p4 wird die Aktivität X ausgeführt. Wird sie fehlerfrei abgearbeitet, ist p5 markiert. Mit dem Schalten von t2 werden die Variablen transCond1 bzw. transCond2 mit der durch die Prozessbeschreibung gegeben `transitionCondition` des jeweiligen `links` belegt. Der Inhalt der Variablen wird auf die Plätze p7 und p8 produziert. Weiterhin gelangt eine Marke auf p6.

Das zweite Szenario betrachtet die Situation, dass sich das `link`-Muster in einem Zweig des Prozesses befindet, dessen Aktivität nicht ausgeführt wird. Stellen wir uns vor, es handelt sich hierbei um den weiter oben angesprochenen `switch`-Zweig. Somit muss an alle `source`-Aktivitäten ein `false` propagiert werden. In diesem Fall ist p1 markiert. Mit dem Schalten von t1 wird auf die beiden ausgehenden `links` eine Marke mit dem Wert `false` sowie eine Marke auf p3 produziert. Letztere Marke soll das `false` an eine weiter innen lie-

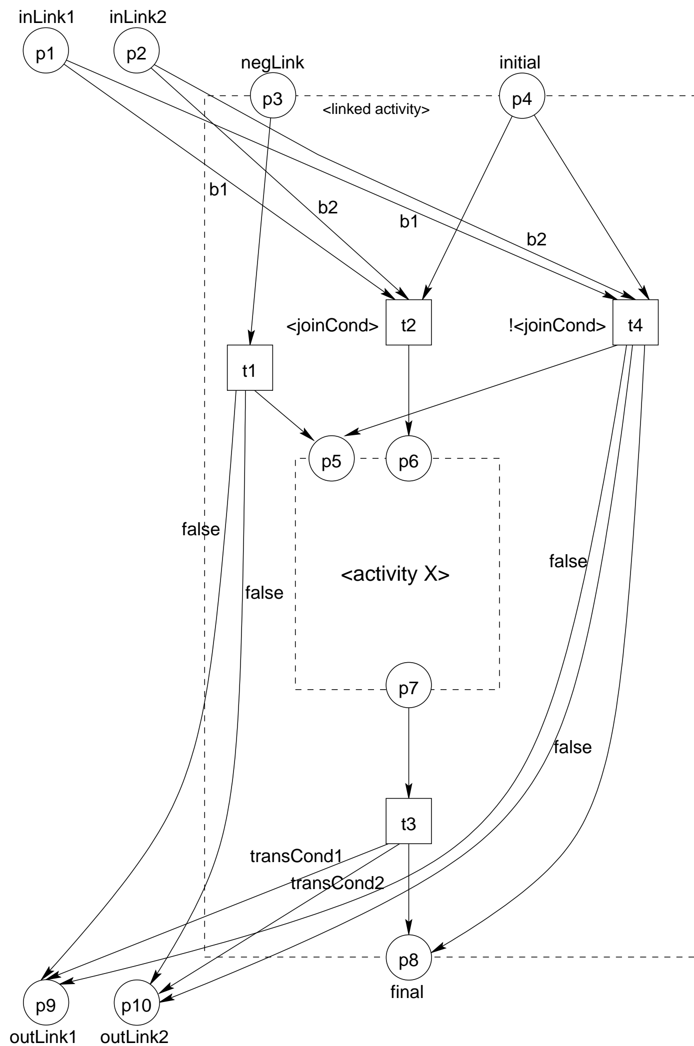


Abbildung 4.5: Muster einer Aktivität X, die target und source je zweier links ist

gende `source`-Aktivität propagieren. Das weitere Verhalten der `DPE` unterliegt somit nicht mehr der Kontrolle des `source`-Elements sondern die Aktivität X ist dafür zuständig.

4.3.4 Muster einer Aktivität mit source- und target-Element

Abschließend betrachten wir in Abbildung 4.5 den komplexen Fall, dass eine Aktivität sowohl `source` als auch `target` verschiedener `links` ist. Diese Abbildung ergibt sich fast vollständig aus den Abbildungen 4.3 und 4.4. Hinzugekommen sind lediglich drei Kanten: t4p5, t4p9 und t4p10. Sie werden beim Schalten der Transition t4 verwendet.

Für den Ablauf des Musters gibt es drei mögliche Szenarien. Im ersten Fall liegt eine Marke auf p3 und alle ausgehenden `links` werden auf `false` gesetzt. Ist dagegen p4 markiert, treten die beiden anderen Szenarien abhängig von der Auswertung der Aktivierungsbedingung {joinCond} auf. Es ist wieder ausgeschlossen, dass p3 und p4 gleichzeitig markiert sind.

Wir wollen ausschließlich auf das Szenario eingehen, dass {joinCond} zu falsch evaluiert wird. Mit anderen Worten schaltet dann t4 und die hinzugefügten Kanten „treten in Aktion". Die beiden anderen Fälle sind unverändert zu den zwei vorherigen Abschnitten geblieben, in denen wir sie erläutert haben. In unserem Szenario sind p1, p2 und p4 markiert. Wie man sieht, ist die Auswertung von {joinCond} abhängig von den beiden Variablen b1 und b2, die mit dem booleschen Wert des jeweiligen links belegt werden. Ergibt die Evaluierung falsch, das heißt, die joinCondition ist nicht erfüllt, schaltet t4. Dabei wird eine Marke auf final produziert und die beiden ausgehenden links werden auf false gesetzt. Desweiteren wird das Scheitern der Bedingung dem Muster der inneren Aktivität X bekannt gegeben, indem eine Marke auf negLink (p5) produziert wird, um ein false an alle in der inneren Aktivität befindlichen source-Aktivitäten zu propagieren. Wir nehmen dabei wieder an, dass X das Muster einer strukturierten Aktivität ist, die mindestens eine source-Aktivität einbettet.

Wir haben die link-Semantik sowie die DPE nach Petrinetze überführen können. Anhand dieses noch kleinen Beispiels ist gut erkennbar, wie das Zusammenstecken der Muster funktioniert. Es ist relativ einfach möglich, aus den Mustern einen Prozess zu bauen. Allerdings wird die graphische Repräsentation des Netzes, wie Abbildung 4.5 verdeutlicht, schnell sehr komplex und damit unübersichtlich.

4.4 Ausblick

In diesem Kapitel haben wir motiviert, warum wir eine formale Semantik für BPEL4WS benötigen und eine Einführung gegeben, wie die Semantik mit Hilfe von Petrinetzen umsetzbar ist. Unser Ansatz umfasst das Erstellen von Petrinetz-Mustern für jedes BPEL4WS-Konzept. Durch den Aufbau unserer Muster, insbesondere ihrer Schnittstelle, ist es möglich, sie nach dem „Baukastenprinzip" zusammenzustecken und zu komponieren. Damit können wir jeden in BPEL4WS spezifizierten Prozess in ein Petrinetz transformieren.

Unser Muster-Konzept haben wir beispielhaft anhand der Aktivität receive vorgestellt. Dabei sind wir auf den allgemeinen Aufbau, wie die Schnittstelle, die Objekte und die Struktur einer Aktivität eingegangen. In der darauf folgenden Übersetzung der link-Semantik und der DPE haben wir das Zusammenstecken von Mustern vorgeführt.

In allen Beispielen haben wir aus Gründen der Einfachheit und der Übersichtlichkeit auf die Anbindung der Muster an das *Transaktionsmanagement* verzichtet. Um dieses Konzept umzusetzen, erweitern wir die Schnittstelle und fügen Subnetze in die Muster ein. Damit gewährleisten wir beim Auftreten eines Fehlers den Abbruch des Kontrollflusses und das Abziehen der Marken aus den Netzen. Die vollständige Petrinetz-Semantik für BPEL4WS, inklusive der Überführung aller anderen Aktivitäten sowie von Fault Handler, Compensation Handler und Event Handler in Petrinetz-Muster findet sich in [Sta04].

5 Reduktion von BPEL4WS-Petrinetzen

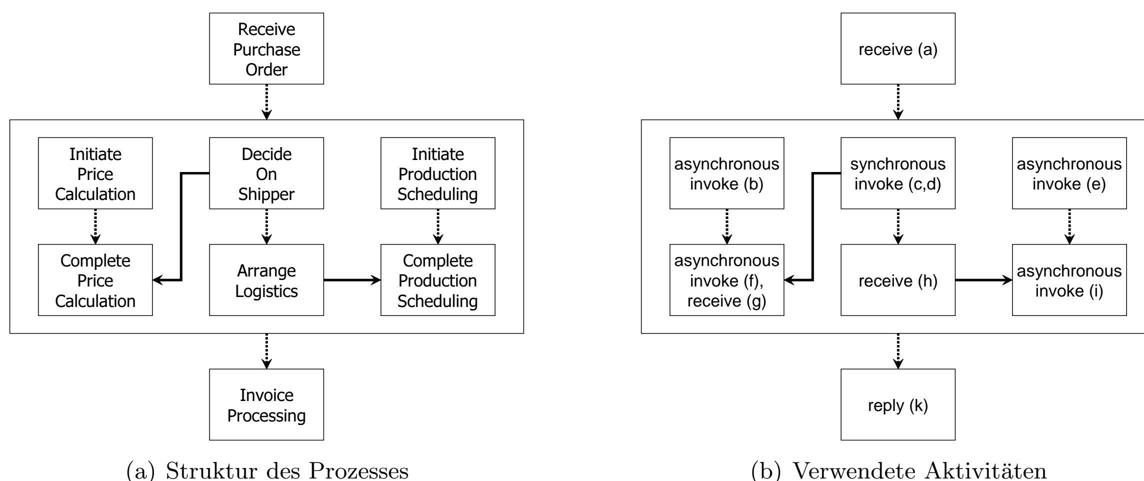
Autor: Thomas Heidinger

5.1 Notwendigkeit der Reduktion

In diesem Kapitel möchten wir auf die Notwendigkeit einer Reduktion der BPEL4WS-Petrinetze aus Kapitel 4 hinweisen und zwei verschiedene Reduktionstechniken vorstellen. Wie wir in Kapitel 4 bereits zeigten, haben wir eine Petrinetzsemantik für BPEL4WS entwickelt, die es uns ermöglicht, einen in BPEL4WS modellierten Geschäftsprozess auf der Basis von Mustern in ein Petrinetz zu übersetzen.

Es stellt sich nun die Frage, wofür die auf diese Weise gewonnenen Netze geeignet sind. Sicherlich dienen sie als formale und damit rechtskräftige Beschreibung eines Web Service, aber was läge näher, als sie auch für Analysezwecke zu verwenden. Es existieren viele Analyseprogramme, die auf Petrinetzen arbeiten. Wollen wir zum Beispiel untersuchen, ob ein Web Service *bedienbar* [Mar04] ist, dann wäre der nahe liegendste Weg, das BPEL4WS-Prozessmodell musterbasierd in ein Petrinetz zu übersetzen und dieses Petrinetz mit dem Wombat4ws [Mar03b] auf Bedienbarkeit zu untersuchen. Im folgenden wollen wir klären, ob dieser Weg erfolgversprechend ist. Betrachten wir dazu den Web Service, der als einleitendes Beispiel in der BPEL4WS-Spezifikation [ACD+02] erläutert wird. Dieser Web Service dient zur Bearbeitung der Bestellung eines Kunden.

Abbildung 5.1(a) zeigt die Struktur des BPEL4WS-Prozesses. Nachdem der BPEL4WS-Prozess die Bestellung des Kunden erhalten hat (Receive Purchase Order), werden drei Aufgaben parallel initiiert: die Berechnung des Endpreises für die Bestellung (Initiate/Complete



(a) Struktur des Prozesses

(b) Verwendete Aktivitäten

Abbildung 5.1: Purchase Order Process

Price Calculation), die Auswahl eines Spediteurs (Decide On Shipper/Arrange Logistics) und die zeitliche Planung der Produktion und des Versandes (Initiate/Complete Production Scheduling). Zwischen diesen drei Aufgaben existieren Abhängigkeiten, so dass sie nicht vollständig parallel ausgeführt werden können. Um den endgültigen Preis zu berechnen, muss der Spediteur bekannt sein, und für die Fertigstellung des Ausführungsplanes wird der Versandtermin benötigt. Nachdem diese Aufgaben erledigt sind, kann die Rechnung an den Kunden gesendet werden (Invoice Processing).

Abbildung 5.1(b) verdeutlicht die zur Konstruktion des BPEL4WS-Prozesses verwendeten Aktivitäten ergänzt um einen Bezeichner für den Kommunikationskanal, über den die jeweilige Aktivität Nachrichten versendet oder empfängt (z. B. Aktivität Receive Purchase Order erhält die Bestellung über den Kanal a). Mit diesen Bezeichnern können wir später die Beziehung zum Petrinetz herstellen.

Wenn wir diesen einfachen Geschäftsprozess nach dem in Kapitel 4 beschriebenen Verfahren musterbasiert in unsere Petrinetze übersetzen, dann entsteht ein Netz mit mehr als 160 Transitionen und 190 Stellen. Zwar lassen sich Netze dieser Größe mit modernen Model-Checking-Werkzeugen unter Zuhilfenahme von Reduktionsverfahren (wie z. B. der *Partial Order Reduction* [Val88]) noch in angemessener Zeit analysieren. Dennoch zeigt dieses Beispiel, dass die entstehenden Netze sehr schnell zu komplex werden, um sie effizient analysieren zu können. Der Grund für die Komplexität liegt in der vollständigen und strukturerhaltenden Transformation des BPEL4WS-Prozesses in ein Petrinetz. Das heißt, es wird jeder Sachverhalt des spezifizierten Geschäftsprozesses im Petrinetz wiedergegeben, einschließlich der *Fehlerbehandlung* und der *Dead Path Elimination*. In vielen Fällen ist aber diese Menge an Informationen für eine spezielle Analyseaufgabe zu detailliert. Möchten wir z. B. nur untersuchen, ob dieser Web Service bedienbar ist, dann würde das Netz aus Abbildung 5.2 ausreichend sein:

Das reduzierte Netz aus Abbildung 5.2 besteht aus nur 10 Transitionen und 15 Stellen und lässt sich daher viel einfacher und effizienter analysieren. Jede der Transitionen sendet oder empfängt eine Nachricht, angedeutet durch die Symbole *!* und *?*, sowie den Bezeichner des Kommunikationskanals. Die gestrichelten Linien gruppieren Transitionen und sollen so die Beziehung zu den Aktivitäten des BPEL4WS-Prozesses hervorheben.

In den folgenden zwei Abschnitten sollen zwei unterschiedliche Herangehensweisen vorgestellt werden, die es ermöglichen, kleinere Netze zu erhalten.

5.2 Reduktionsregeln

Eine Möglichkeit, die Komplexität der BPEL4WS-Petrinetze zu verringern, besteht darin, die generierten Petrinetze mit Hilfe von Transformationsregeln in kleinere Netze zu überführen. Wichtig hierbei ist, dass die Transformation unsere zu analysierende Eigenschaft erhält. Wenn das Originalnetz bedienbar ist, dann muss auch das reduzierte Netz bedienbar sein und umgekehrt. Im folgenden sollen zwei Regeln exemplarisch vorgestellt werden, die es erlauben, eine Stelle und eine Transition aus einem Netz zu streichen und beide Netze bedienungsäquivalent lassen. Weiterführende Regeln zur Reduktion von Petrinetzen finden sich in [Mar04] und [Sta90].

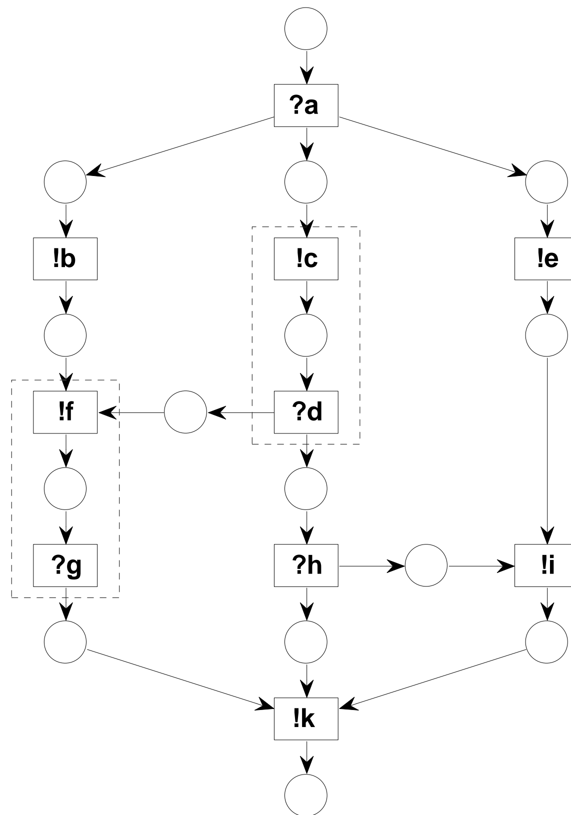
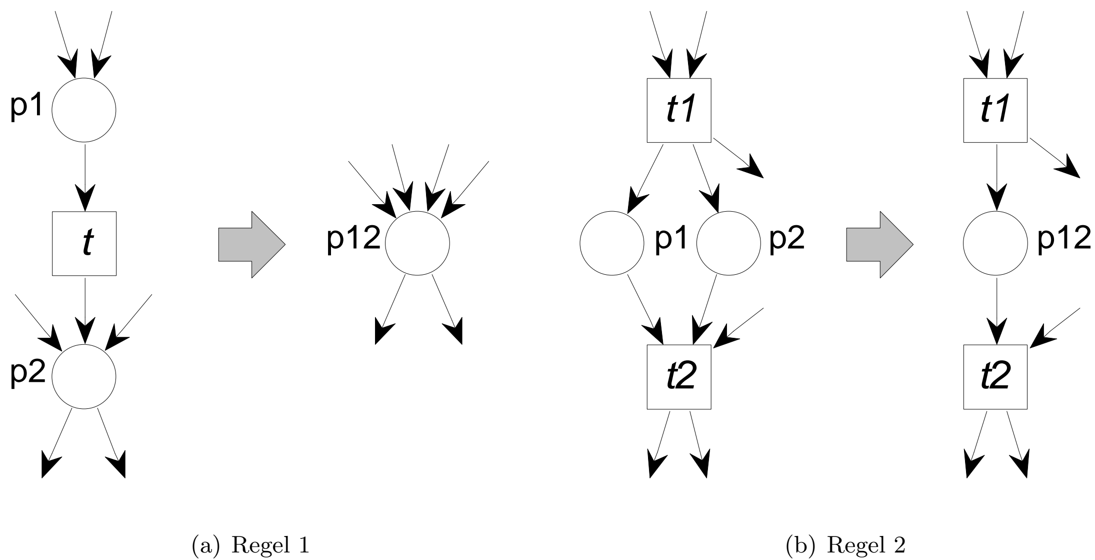


Abbildung 5.2: reduziertes Netz



(a) Regel 1

(b) Regel 2

Abbildung 5.3: Reduktionsregeln

Regel 1

Hat eine Transition, die nicht mit der Umgebung kommuniziert, genau eine nicht verzweigende Stelle im Vorbereich und genau eine Stelle im Nachbereich, dann darf diese Transition entfernt und die beiden Stellen fusioniert werden. Abbildung 5.3(a) verdeutlicht diese Regel.

Regel 2

Wenn zwei Stellen den gleichen Vorbereich und den gleichen Nachbereich haben, dann darf eine der Stellen entfernt werden. Abbildung 5.3(b) verdeutlicht diese Regel.

Der modulare Aufbau unserer Pattern ist der Grund dafür, warum wir solche Muster in den generierten Petrinetzen erhalten werden, die wir reduzieren können. Die Übersetzung betrachtet den allgemeinen Fall, der viele Möglichkeiten wiederspiegeln muss. In der Realität liegt jedoch ein Spezialfall vor, in der zum Beispiel nicht jede Aktivität mit ein- und ausgehenden Links versehen sein muss. Abbildung 5.4 zeigt ein Anwendungsbeispiel für Regel 1. An den grau hinterlegten Stellen können wir die Reduktionsregel 1 anwenden. Die gestrichelten Pfeile in Abbildung 5.4(b) sollen andeuten, dass die auf diese Weise verbundenen Stellen verschmolzen wurden.

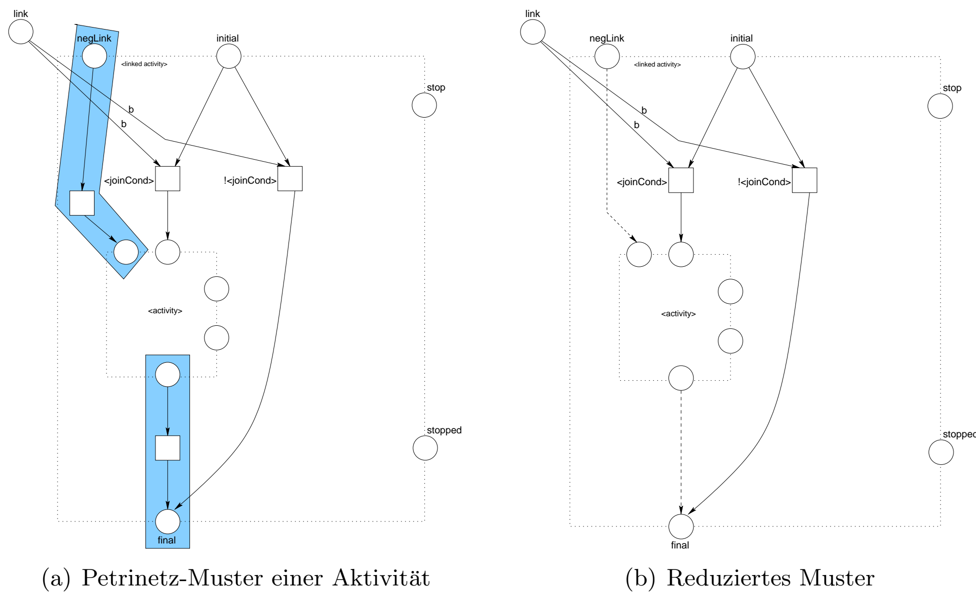
Mit den Regeln zur Fusion bzw. Eliminierung von Netzelementen ist es möglich, ein gegebenes Petrinetz in ein *bedienungsäquivalentes*, jedoch einfacher strukturiertes Petrinetz zu transformieren. Allerdings hat diese Methode auch ihre Grenzen. Trotz umfangreicher Sammlungen von Mustern und Regeln lassen sich immer wieder Beispiele finden, bei denen regelbasiert keine weitere Reduktion möglich ist, obwohl ihre Struktur noch unnötig kompliziert ist. Darüber hinaus sind die Regeln in umfangreichen Sammlungen i. Allg. nicht in beliebiger Reihenfolge anwendbar, das heißt, es bedarf eines heuristischen Vorgehens, um optimale Ergebnisse zu erreichen. Aus diesem Grund betrachten wir mit dem *Slicing* noch ein weiteres Verfahren der Reduktion.

5.3 Slicing

Program-Slicing dient zur Beantwortung der Frage: „Welche Anweisungen können die Berechnungen einer anderen Anweisung beeinflussen?“. Ursprünglich wurde Slicing 1979 als Hilfsmittel beim Debugging entwickelt [Wei79], denn Programmierer machen bei der Fehlersuche auch nichts anderes als von fehlerhaften Zwischenergebnissen rückwärts nach verursachenden Anweisungen zu suchen. Heute stellt sich die Frage in den verschiedensten Anwendungen, doch besonders im Bereich des Software-Reengineering.

Die Technik des Slicing entstammt der statischen Programmanalyse [Tip95, NNH99]. Man versteht darunter das Streichen aller Programmteile, die keinen Einfluss auf eine zu analysierende Eigenschaft, d. h. auf die betrachtete Anweisung haben. Das so entstandene Teilprogramm ist ein *Slice* des ursprünglichen Programms.

Beim Slicing paralleler Programme [Che93, Che97] ergeben sich Schwierigkeiten, da die Ausführungsreihenfolge dynamisch ist. Für sequentielle Programme hingegen ist Slicing bedeutend einfacher. Slices lassen sich durch iteratives Lösen von Datenflussgleichungen [Wei84] oder durch eine Erreichbarkeitsanalyse in Programmabhängigkeitsgraphen



(a) Petrinetz-Muster einer Aktivität (b) Reduziertes Muster

Abbildung 5.4: Beispiel für Regel 1

[FOW87] berechnen. Solche Graphen bestehen aus Knoten, die die Anweisungen des Programms repräsentieren und Kanten, die die Kontrollfluss- und Datenabhängigkeit abbilden.

Den zweiten Weg veranschaulichen wir in Abbildung 5.5. Um den Slice für die Anweisung `write(a)` zu erhalten, müssen wir nur die Abhängigkeiten für `write(a)` rückwärts verfolgen. Das reduzierte Programm für die Anweisung `write(a)` findet sich in Abbildung 5.6.

Aktuelle Forschungen zielen darauf ab, diese Technik in der Analyse von BPEL4WS-Prozessen anzuwenden. Die Idee ist es, vor der Überführung in ein Petrinetz den für die konkrete Analyseaufgabe notwendigen Teilprozess (d. h. Slice) zu identifizieren und somit nur die relevanten Informationen zu übertragen. Auf diese Weise entstehen von vornherein kleinere BPEL4WS-Petrinetze.

Abstrahiert man von Daten, dann sind z. B. für die Analyse der *Bedienbarkeit* nur die kommunizierenden Aktivitäten `invoke`, `receive`, `reply` und `pick` sowie die strukturierten Aktivitäten von Bedeutung. Andere Aktivitäten wie zum Beispiel ein `wait`, `assign` oder

Abbildung 5.6: Slicing

Abbildung 5.5: Programm

`empty` tauschen keine Nachrichten mit einem Partner aus, beeinflussen also den *Kommunikationsgraphen* [Mar04] nicht und können demzufolge ignoriert werden. Es besteht daher kein Anlass, sie in ein BPEL4WS-Petrinetz zu überführen, es sei denn, sie nehmen durch `links` Einfluss auf die Kontrollstruktur.

Für vielfältige Analysen ist lediglich der positive, fehlerfreie Ablauf des Prozesses relevant. Daher brauchen z. B. die `FaultHandler` und deren Anbindung an den Prozess ebenfalls bei der Übersetzung nicht berücksichtigt zu werden. Die genannten Beispiele sind erst der Anfang einer Sammlung von Aktivitäten, die bei der Übersetzung ignoriert werden können.

Zur vollen Entfaltung kommt die Methode des Slicings jedoch erst, wenn die modellierten Daten im Prozess berücksichtigt werden. Für Fragestellungen nach dem Einfluss von Nachrichteninhalten auf interne Entscheidungen oder dem Wertebereich einer Variable, so dass ein vorgegebener Ablauf möglich ist, ist die Fokussierung auf die relevanten Teile des Prozesses unerlässlich. Dieser Bereich ist bisher kaum betrachtet worden und stellt eine der wesentlichen Aufgaben zukünftiger Forschung dar.

5.4 Ausblick

In diesem Kapitel haben wir gezeigt, dass die generierten Petrinetze für Analysezwecke wenig geeignet sind, da sie zu komplex sind. Der Grund hierfür ist in der Modularität und Vollständigkeit der Übersetzung zu finden. Um kleinere Netze zu erhalten, können wir rein syntaktisch die Netze mit Hilfe von Reduktionsregeln minimieren. Diese Regeln müssen unsere zu analysierende Eigenschaft erhalten. Eine andere Möglichkeit besteht darin, von vornherein von Details des BPEL4WS-Prozesses zu abstrahieren und diese Details dann nicht in unsere Petrinetze zu übersetzen. Es wird angestrebt, in der Abstraktion von Details des BPEL4WS Prozesses so großzügig wie möglich zu sein und dennoch präzise Aussagen über den Prozess treffen zu können. Das Finden von geeigneten Reduktionsregeln und Aktivitäten, die nicht übersetzt werden müssen, sind Gegenstand aktueller Forschungen unseres Lehrstuhls. Ziel ist es, Petrinetze zu erhalten, die sich effizient mit Model-Checking-Werkzeugen wie LoLA [Sch00b] oder Wombat4ws [Mar03b] analysieren lassen.

6 Eine ASM-Semantik für BPEL4WS

Autor: Dirk Fahland

6.1 Ansatz für eine formale Semantik

Dieses Kapitel zeigt den Ansatz zur Formalisierung der *Business Process Execution Language for Web Services* (BPEL4WS) mit *Abstract State Machines* (ASMs). Das Ziel ist es, die Sprache BPEL4WS in ein mathematisches Modell zu überführen, anhand dessen die Konsistenz geprüft werden kann. Grundsätzlich verfolgen wir den Ansatz, die Semantik von allen konkreten BPEL4WS-Prozessen gelöst zu formulieren. Sie soll für sich stehen und in jeder Situation die Elemente der Sprache korrekt beschreiben. Konkrete Prozesse können dann ausgeführt werden, in dem die allgemein formalisierte Semantik auf die geeignet übersetzte Prozessdefinition angewandt wird. Was „geeignet" bedeutet, soll ebenfalls in diesem Papier erläutert werden.

Für das erste Verständnis genügt es, ASMs als eine Menge von Pseudocode-Programmen (Regeln) zu verstehen, deren Variablen und Funktionen (induktiv definierte) Terme sind. Ziel der ASM-Semantik ist die allgemeine Formalisierung der in BPEL4WS definierten Konstrukte, so dass wir von jedem konkreten BPEL4WS-Prozess abstrahieren können. Die einmal gegebene formale Semantik soll in allen Situationen und Ausprägungen gültig sein. Hierzu trennen wir allgemeingültige Konstrukte der Sprache von für einen BPEL4WS-Prozess spezifischen Konstrukten. Letztere sowie das Zusammenspiel der beiden werden wir ebenfalls formalisieren, um damit auch konkrete Prozesse zu beschreiben. Damit ist es uns möglich die generellen Konzepte von BPEL4WS zu betrachten und Eigenschaften unabhängig von Einzelfällen zu untersuchen.

Die grundsätzliche Vorgehensweise der Formalisierung soll im Folgenden charakterisiert werden. Bei näherer Betrachtung der informalen BPEL4WS-Semantik lässt sich diese in die Aktivitäten als ihre Grundbausteine zerlegen. Die Aktivitäten existieren nahezu unabhängig voneinander: Die Semantik einer Aktivität ist nicht von der Semantik einer anderen abhängig. Die Interaktion lässt sich über ein allgemeines, verbindendes Grundgerüst modellieren.

Jede Aktivität wiederum kann in ihr *Verhalten* (aktive Systemkomponenten) und in die ihr zugrundeliegenden *Strukturen* (passive Systemkomponenten) zerlegt werden. Die Strukturen zerfallen in zwei Klassen. Die erste Klasse der *statisch definierten Strukturen* ist durch die Repräsentation der BPEL4WS-Prozess-Definition in XML gegeben. *Implizit definierte Strukturen* bilden die zweite Klasse. Sie sind durch die (informale) Semantik bedingt und müssen aus der Spezifikation heraus gelesen werden. In jedem Fall lassen sich alle strukturellen Elemente und die Beziehungen untereinander in Terme überführen.

Die Semantik einer Aktivität wird jeweils in *ASM-Regeln* formalisiert. Diese Regeln verwenden ausschließlich Terme, die auch zur Struktur der Aktivität bzw. zum zugrundeliegenden Gerüst gehören.

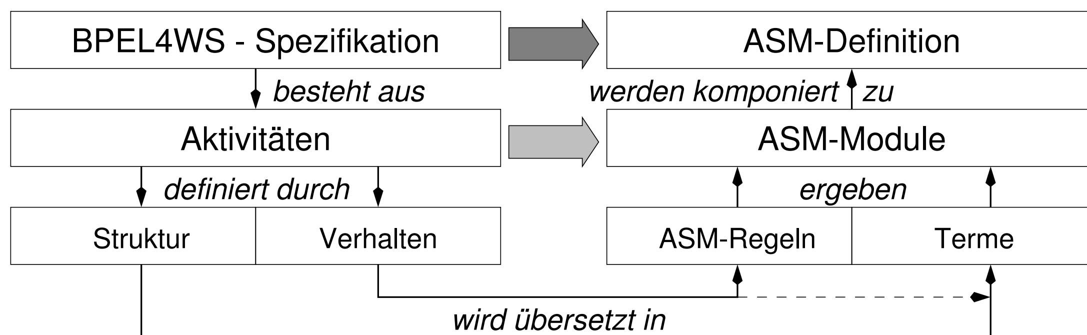


Abbildung 6.1: Übersetzung der BPEL4WS-Spezifikation in ASMs

Alle ASM-Regeln für eine Aktivität ergeben zusammen ein *ASM-Modul*, welches wir als abgeschlossene Komponente der ASM betrachten. Ein ASM-Modul ist die vollständige formale Beschreibung der Semantik der zugehörigen Aktivität. Somit gibt es eine direkte Beziehung zwischen der informalen Semantik einer Aktivität und der formalen Beschreibung. Alle ASM-Regeln für alle Aktivitäten werden zur vollständigen *ASM-Definition* komponiert. Die ASM-Definition ist die formale Semantik für BPEL4WS. Dieses Vorgehen ist in Abb. 6.1 noch einmal veranschaulicht.

Dieser Teil der Arbeit gliedert sich wie folgt. In Abschnitt 6.2 geben wir eine informale Einführung in die Syntax und Semantik von Abstract State Machines. Von besonderem Interesse sind hierbei die Asynchronen Verteilten ASMs. In dem folgenden Abschnitt 6.3 charakterisieren wir die Übersetzung, der der Semantik zugrundeliegenden Strukturen. Anschließend wird in Abschnitt 6.4 gezeigt, wie die informale Semantik von Aktivitäten in ASM-Regeln übersetzt wird. Dabei werden wir auch das verbindende Grundgerüst vorstellen. Abschließend geben wir in Abschnitt 6.5 eine kurze Zusammenfassung des Ansatzes und den Ausblick auf die weitere Arbeit zur ASM-Semantik.

6.2 Abstract State Machines

Dieser Abschnitt gibt eine kurze Einführung in *Abstract State Machines (ASMs)*. Leser, die mit diesem Formalismus vertraut sind, sollten sich zumindest den Abschnitt 6.2.3 durchlesen.

Abstract State Machines (ASMs) wurden 1985 von Yuri Gurevich als ein „Berechnungsmodell [eingeführt], dass mächtiger und allgemeiner ist, als Standard-Berechnungsmodelle“ [Gur85]. ASMs haben in der Zwischenzeit breite Anwendung als Spezifikationsmethode gefunden, u.a. bei SDL-2000, C/C++, Java und Prolog (s. ASM-Webseite [ASMa]).

Der ASM-Formalismus definiert mehrere Klassen von Abstract State Machines. *Sequentielle ASMs* führen einen Zustandsübergang nach dem anderen anhand genau eines Schrittes aus. In *parallelen ASMs* ist ein Zustandsübergang durch viele parallele Teilschritte gegeben. Die Klasse der *verteilten asynchronen ASMs* charakterisiert den Zustandsübergang durch Teilschritte beliebig vieler Agenten. Sequentielle Abstract State Machines bilden die

einfachste Klasse von ASMs. Auf ihnen bauen die restlichen Klassen auf. Die letzte Klasse werden wir für die ASM-Semantik verwenden.

Bevor wir uns näher mit der Übersetzung nach ASM befassen, geben wir eine informale Definition der Abstract State Machines an. Wir beginnen mit den sequentiellen ASMs und erweitern davon ausgehend auf asynchrone, verteilte ASMs.

6.2.1 Sequentielle Abstract State Machines

Eine ASM modelliert ein Transitionssystem. Ein Zustand einer ASM ist eine mathematische Struktur, genauer eine *Σ-Algebra*. Ein Zustandsübergang beschreibt anhand eines komplexen Schritts die Veränderung des neuen gegenüber dem alten Zustand. Um dies zu verstehen, betrachten wir zunächst die Zustände selbst.

In einem Zustand, also einer Algebra, werden Zusammenhänge zwischen Objekten des Systems durch *Funktionen* beschrieben. Alle (möglichen) Objekte sind in einer Menge U, dem *Universum* (oder *Träger*) des Zustands, zusammengefasst.

Eine Σ-Algebra S stellt uns also eine Menge U und Funktionen $\varphi : U^n \to U$ über dieser Menge zur Verfügung. Die Signatur Σ definiert *Funktionssymbole* (z.B. f), die wir benutzen, um Funktionen zu notieren. Eine Funktion und ihr Symbol sind nicht identisch – wichtig ist dies in Bezug auf unterschiedliche Zustände, also unterschiedliche Algebren.

Eine Funktion ist in einem Zustand S für die darin vorkommenden Elemente des Trägers definiert. In einem zweiten Zustand T kann die Funktionsdefinition, die wir mit dem selben Symbol f darstellen, verschieden von der Funktionsdefinition in S sein. Daher bezeichnen wir mit f_S die in S definierte Funktion, die durch f syntaktisch repräsentiert wird: f_S ist die *Interpretation von f in S*.

In S können wir anhand der Funktionen Beziehungen ausrechnen. Verketten wir Funktionen, so ergeben sich komplexe Zusammenhänge. Aus syntaktischer Sicht konstruieren wir (wie üblich) *induktiv definierte Terme* aus den Symbolen von Σ, wobei 0-stellige Funktionssymbole *Konstantensymbole* sind. Für jede so notierte Verkettung von Funktionen können wir (angefangen bei den Konstanten) den Wert eines beliebigen Terms berechnen und vergleichen:

$$f(t_1, \ldots, t_n) = t_0 \tag{6.1}$$

gilt im Zustand S, wenn die Funktion f_S, angewandt auf die Werte $v_i, i \in [n]$[1], die sich durch Berechnung der Terme $t_i, i \in [n]$ ergeben, auf den Wert von t_0 v_0 abbildet:

$$f_S(v_1, \ldots, v_n) = v_0. \tag{6.2}$$

Für $v_i, i \in [n]_0 =_{def} \{0, \ldots, n\}$ schreiben wir auch $t_{iS}, i \in [n]_0$, die *Interpretation von t_i in S*. Es gilt natürlich $v_0, \ldots, v_n \in U$.

Ausgehend von einem Zustand T, in dem wie in S $t_{iS} = v_i = t_{iT}, i \in [n]_0$ gilt, wollen wir erreichen, dass die Gleichung (6.2) in S auf jeden Fall erfüllt ist. Dies ist nur von f_S abhängig. Daher definieren wir f_S beim Zustandsübergang von T nach S an der Stelle $(v_1, \ldots, v_n)$ einfach neu – und zwar syntaktisch notiert als *Σ-Assignment*

$$f(t_1, \ldots, t_n) := t_0. \tag{6.3}$$

[1] Wir verwenden für Mengen von Indizes die abkürzende Schreibweise $[n] =_{def} \{1, \ldots, n\}$

Die semantische Entsprechung ist die Funktionsdefinition in S:

$$f_S(v_1,\ldots,v_n) =_{def} \begin{cases} v_0, & \text{falls } f(t_1,\ldots,t_n) := t_0 \text{ ein } \Sigma\text{-Assignment} \\ & \text{und für } i \in [n]_0 : t_{iT} = v_i \\ f_T(v_1,\ldots,v_n), & \text{sonst} \end{cases}$$

Die Funktionsdefinition bleibt also für alle Tupel $(v_1,\ldots,v_n)$, die nicht durch $(t_1,\ldots,t_n)$ in T repräsentiert werden, identisch mit f_T. Ein Σ-Assignment $f(t_1,\ldots,t_n) := t_0$ angewandt auf einen Zustand T erzeugt dann das Σ-*Update* $(f,(t_{1T},\ldots,t_{nT}),t_{0T})$, das die partielle Funktionsdefinition beschreibt. Das Σ-Update ist dann am Zustandsübergang *beteiligt*.

Mit den obigen Annahmen ist (6.2) in S per Definition von f_S erfüllt. Den Übergang von T nach S dürfen natürlich mehrere solcher Funktionsdefinitionen beschreiben. Dabei muss beachtet werden, dass niemals die selbe Funktion für die gleichen Argumente mit zwei verschieden Werten definiert wird. Zwei Σ-Updates u_1 und u_2 nennen wir *inkonsistent*, wenn $u_1 = (f,(t_{1T},\ldots,t_{nT}),t_{0T})$ und $u_2 = (f,(t_{1T},\ldots,t_{0T}),t^*_{0T})$ mit $t_{0T} \neq t^*_{0T})$ gilt .
Sind mehrere Σ-Updates an einem Übergang beteiligt, so werden alle Funktionen auf einmal neu definiert. Insbesondere hat für

$$f(g(s_1,\ldots,s_k),t_2,\ldots,t_n) := t_0 \text{ und } \; g(s_1,\ldots,s_k) := s_0$$

die Neudefinition der Interpretation von g keinen Einfluss auf die Neudefinition der Interpretation von f.

Ist $\mathcal{U}_{T\to S}$ die Menge der Σ-Updates, die am Übergang von T nach S beteiligt ist und sind alle Σ-Updates in $\mathcal{U}_{T\to S}$ paarweise konsistent, dann gilt für alle Funktionssymbole f der Signatur Σ:

$$f_S(v_1,\ldots,v_n) =_{def} \begin{cases} v_0, & ((f,(v_1,\ldots,v_n)),v_0) \in \mathcal{U}_{T\to S} \\ f_T(v_1,\ldots,v_n), & \text{sonst} \end{cases}$$

f_S ist die Folgedefinition von f_T aufgrund des Zustandsübergangs von T nach S. Das bedeutet, wir rechnen zunächst alle Σ-Updates in T mittels der Σ-Assignments aus und erzeugen dann die neue Definition anhand der Werte in T simultan.

Σ enthält die Symbole $true, false, undef, =, \neg$ und $\wedge$ deren Interpretation in allen Zuständen gleich ist. Ist für einen Term t^* in einem Zustand S keine Interpretation gegeben, so ist dies gleichbedeutend mit dem Symbol $undef$, d. h.: $t^*_S = undef$. Für alle anderen der eben genannten Symbole nehmen wir die übliche Interpretation an.

Hinweis: Wir können Σ-Updates nur durchführen, wenn die an der Neudefinition beteiligten Elemente des Trägers auch eine syntaktische Repräsentation für ein Σ-Assignment haben. Wir betrachten damit ausschließlich *term-erzeugte* Elemente $u \in U$ des Trägers, also nur solche, für die ein Term t existiert, so dass $t_S = u$ im Zustand S ist.

Ausgehend von Σ-Assignments können wir uns nun der Definition einer Abstract State Machine nähern. Ein *sequentielles ASM-Programm* ist *eine* rekursiv definierte Regel. Jedes Σ-Assignment ist eine Regel. **skip** ist eine Regel (nichts tun). Sei φ ein prädikatenlogischer Ausdruck ohne freie Variablen, $\psi(x)$ ein prädikatenlogischer Ausdruck mit x als einziger freier Variable, x eine Variable, *term* ein Term und *Set* ein Mengensymbol für eine Teilmenge von U. Sind P und Q Regeln, so entstehen durch die Anwendung folgender Verkettungen ebenfalls Regeln:

- R_{par} $\equiv$ P **par** Q (paralleles Anwenden von P und Q),
- R_{if} $\equiv$ **if** φ **then** P **else** Q (bedingtes Anwenden von P, wenn φ im aktuellen Zustand erfüllt ist, sonst Q)
- R_{forall} $\equiv$ **forall** $x \in Set$ **where** $\psi(x)$ **do** P (paralleles Anwenden von P für alle x aus der Menge Set, die ψ erfüllen)
- R_{choose} $\equiv$ **choose** $x \in Set$ **where** $\psi(x)$ **do** P (Anwenden von P für ein nicht-det. ausgewähltes x aus der Menge Set, das ψ erfüllt)
- R_{let} $\equiv$ **let** $x = term$ **in** P (entspricht einer abkürzenden Schreibweise für $term$, x darf auf keiner linken Seite eines Σ-updates stehen)
- R_{new} $\equiv$ **let** $x = new(Set)$ **in** P (häufiger Sonderfall, new angewendet auf eine (in diesem Fall unendliche) Menge Set liefert ein bislang nicht term-erzeugtes Element des Trägers)

Existiert im Falle von R_{forall} kein x, dass φ erfüllt, so wird P überhaupt nicht angewandt. Dieser Fall darf bei R_{choose} nicht auftreten und kann durch ein vorheriges R_{if} abgefangen werden. Wir werden im Abschnitt 6.2.3 noch einige Konvention zur Notation angeben, die die Syntax ein wenig handlicher gestalten.

Hinweis zu R_{new}: Allein auf diese Weise lässt sich die Menge der im Zustand erreichbaren Elemente erweitern: **let** message $= new(BPELMsg)$ **in** R

Eine *sequentielle Abstract State Machine* besitzt eine Signatur Σ, eine ASM-Regel R und eine Menge von Anfangszuständen *init*. Ein *Ablauf* einer sequentiellen ASM ist eine unendliche Folge von Zuständen $S_0 S_1 S_2 \dots$ wobei $S_0 \in init$ ein Anfangszustand ist und jeder Übergang von S_i zu S_{i+1} durch einen Schritt gegeben ist, der durch Anwenden von R auf S_i definiert ist.

6.2.2 Asynchrone Verteilte Abstract State Machines

Verteilte ASMs sind prinzipiell mehrere sequentielle ASMs, wobei jede einzelne sequentielle ASM lokal arbeitet, d.h. die anderen ASMs nicht kennt. Die Asynchronität im System wird dadurch gegeben, dass jede einzelne ASM einen Schritt ausführen kann, wann sie möchte. Es gibt keinen Zwang einen Schritt zu tun oder ihn mit einer anderen ASM zu tun.

Formalisiert wird dieser Ansatz durch eine Menge von Agenten (a, R), von denen jeder einzelne einer sequentiellen ASM entspricht. Hierbei ist $a \in Agent$ der Name des Agenten und R ist die Regel der zugehörigen sequentiellen ASM.

Als Zustand des Gesamtsystems dient *eine* Σ-Algebra. Des Weiteren ist die Regel eines Agenten nicht exklusiv für ihn. Vielmehr können mehrere oder sogar alle Agenten die gleiche Regel ausführen. Wir führen daher ein generisches Symbol *self* ein, das in jeder Regel wie ein Σ-term behandelt wird [BS03]. *self* repräsentiert den Namen des jeweils die Regel anwendenden Agenten. Dieser Mechanimus führt die Lokalität eines Agenten auch auf Regel-Ebene ein. Daher tritt das Universum der Agentennamen auch als Parameter für

Funktionen auf, z.B. verweist jeder Agent auf eine Aktivität (Universum: *Activity*) eines BPEL4WS-Prozesses: $myStatic : Agent \rightarrow Activity$.

Ein *Schritt einer verteilten ASM* entspricht den parallelen Schritten einer endlichen Teilmenge seiner Agenten. Der *Ablauf einer verteilten ASM* ist nun gegeben durch eine Halbordnung. Jeder Ablauf des Gesamtsystems hat einen endlichen Anfang. Der Ablauf eines jeden Agenten ist linear geordnet. Sind zwei Agenten an einem Schritt des Gesamtsystems beteiligt, so müssen die jeweiligen Schritte der Agenten kommutativ sein, d.h. jede Linearisierung des Schrittes liefert den gleichen Folgezustand des Gesamtsystems.

Des Weiteren fordern wir eine Fairness-Bedingung für jeden Agenten: Jeder Agent, der ziehen kann, tut dies auch irgendwann einmal.

6.2.3 ASMs in der BPEL4WS-Spezifikation

Wir legen nun noch einige Konventionen zur Definition der ASM für BPEL4WS fest. Zunächst unterteilen wir den Träger U in *Teiluniversen*, denen wir ebenfalls Symbole zuordnen. Die Interpretation dieser Symbole ist jedoch fest. Die *Signaturen der Funktionen*, die wir definieren, beschreiben abstrakt für ein Funktionssymbol die Abbildung vom Definitionsbereich in U in den Wertebereich in U anhand der Symbole für die entsprechenden Teiluniversen: z.B. ist $myStatic : Agent \rightarrow Activity$ die Signatur für alle Funktionen, die durch $myStatic$ interpretiert werden.

Eine Besonderheit bilden *abstrakte Funktionen*. Von ihnen nehmen wir eine korrekte Definition in jedem Zustand bezüglich einer informalen Charakterisierung an. Sie dienen dazu von nicht-relevanten Teilen der Modellierung zu abstrahieren. Abstrakte Funktionen bilden gewissermaßen die Schnittstelle zwischen formal modellierter Welt und der Umgebung.

Wir fassen gemäß der informalen Semantik zusammen gehörige ASM-Regeln und Funktionen zu *ASM-Modulen* zusammen, wobei wir in dieser Arbeit die Module lediglich informal charakterisieren wollen. Wir werden die Trennlinie zwischen den Modulen so ziehen, dass die aktivitätsspezifischen Funktionen und Regeln jeweils komplett in einem Modul gekapselt sind. Zu der in Abschnitt 6.2.1 gegebenen Syntax führen wir nun einige Notationsregeln ein, die das Lesen der Semantik erleichtern sollen.

- Ein Teil-Universum des Trägers wird mit **Universe** deklariert. Universen werden stets kursiv und mit großem Anfangsbuchstaben gesetzt: **Universe**: *Activity*.
- Funktionen und Σ-Terme werden kursiv gesetzt und beginnen mit einem kleinen Buchstaben. Ein- bzw. Mehrzahl des Namens spiegelt die Abbildung auf ein einzelnes oder mehrere Elemente wieder: $activityParent : Activity \rightarrow Activity$, $activityChilds : Activity \rightarrow \mathcal{P}(Activity)$.
- Variablen werden serifenlos gesetzt und haben einen kleinen Anfangsbuchstaben: reply.
- Die ASM-Schlüsselworte (auch „Operatoren“) werden mit Serifen und fett gesetzt: **forall**.
- Regelnamen werden in Kapitälchen gesetzt: ACTIVITYEXECUTE.

Anstatt des **par**-Operators schreiben wir die parallel auszuführenden Regeln untereinander. Auf **skip** angewandte Operatoren werden weggelassen, dies trifft insbesondere auf **else**-Zweige zu.

Die von **forall**, **if**... **then**... **else**, **choose** und **let** umschlossenen Regeln werden relativ zu diesen in einer neuen Zeile eingerückt geschrieben. Die Einrückung bestimmt zu welchen Operatoren welche Regel gehört.

Folgendes Beispiel soll dies veranschaulichen:

CopyVariableToMsg
  $(\mathsf{pI} \in ProcessInstance, \mathsf{var} \in Variable, \mathsf{msg} \in BPELMsg) \equiv$

if $\mathsf{var} \neq undef$ **then**
  forall $\mathsf{part} \in msgTypeParts(messageType(\mathsf{var}))$ **do**
    $msgPartValue(\mathsf{msg}, \mathsf{part}) := variablePartValue(\mathsf{pI}, \mathsf{var}, \mathsf{part})$
  $messsageMsgType(\mathsf{msg}) := variableMsgType(\mathsf{var})$

Die erste Zuweisung gehört zum **forall**-Operator, die zweite Zuweisung nicht. Der **forall**-Operator und die zweite Zuweisung werden parallel ausgeführt.

Aus Gründen der Lesbarkeit unterteilen wir die Regeldefinitionen. Wir geben einzelne parametrisierte Teilregeln der ASM-Regeldefinition an und referenzieren diese mit einem Regelaufruf. Dies ist vom Verständnis her vergleichbar mit einer Funktionsdefinition und einem Aufruf, stellt jedoch eine Makrodefinition und dessen Verwendung dar.

Der Regelaufruf ersetzt diesen durch den Körper der Regeldefinition. Hierbei werden die im Körper verwendeten Parameter durch die beim Aufruf übergebenen Terme ersetzt. Eine Regel darf mehrfach an verschiedenen Stellen und auch aus verschiedenen Regeln heraus aufgerufen werden, da es sich um einfach Textersetzung handelt. Dies schliesst unmittelbar jedwede rekursive Regeldefinition aus!

Abschließend sei noch eine abkürzende Notation für **choose** gegeben, bei der auch der Fall behandelt wird, in dem es kein Element in der gegebenen Menge gibt, das die Bedingung φ erfüllt. Seien P und Q Regeln,φ ein prädikatenlogischer Ausdruck und x eine Variable. Dann wird

select $\mathsf{x} \in Set$ **where** $\varphi(\mathsf{x})$ **in**
  P
ifnone
  Q

durch folgende Regelkomposition ersetzt:

if $\exists x \in Set : \varphi$ **then**
  choose $\mathsf{x} \in Set$ **where** $\varphi(\mathsf{x})$ **in**
    P
else
  Q

Es handelt sich bei **select** um eine textuelle Abkürzung der **if/choose** Komposition.

6.2.4 Weiteres zu Abstract State Machines

Die hier vorgestellte Syntax und Semantik der Abstract-State Machines wurde in [Gur97] erstmals definiert und im ASM-Buch [BS03] mit einem deduktiven Kalkül versehen. Besonders interessant am ASM-Ansatz ist die Art und Weise, wir die Abläufe einer ASM verstanden werden können: So basiert jeder Zustand einzig und allein auf der Gleichheitsrelation, die angibt, welche Terme als gleich interpretiert werden, und welche nicht. Ein Schritt beschreibt lediglich die Veränderung der Gleichheit, also für welche Terme im Folgezustand die Gleichheit auch gilt. In [Rei03] wird dies für eine Teilklasse der sequentiellen ASMs im Einzelnen dargelegt.

Die Entwicklung des ASM-Formalismus wurde mit dem Ziel vorangetrieben, die Lücke zwischen Berechnungsmodellen und Spezifikationsmethoden zu schließen [Gur]. So hat Y. Gurevich für sequentielle und parallele Algorithmen bewiesen, dass sequentielle ASMs [Gur00] und parallele entsprechend [BG03], die jeweilige Klasse von Algorithmen berechnen können. Für asynchrone verteilte ASMs existiert derzeit kein derartiges Theorem.

Alle bisher vorgestellten Klassen von Abstract-State Machines können keine rekursiven Algorithmen beschreiben. Hierfür wurde der ASM-Ansatz erweitert: Recursive ASMs [GS97] und Turbo ASMs [BS00] ermöglichen dies. Die letztere Erweiterung enthält zudem einige interessante Ansätze zur strukturierten, sequentiellen Komposition, Parametrisierung und Modularisierung großer ASM-Definitionen (vgl. auch [BS03, 2.2.4]).

Gleichzeitig haben sich ASMs als sehr vorteilhaft bei der Formalisierung und Spezifikation von verschiedenen Programmiersprachen erwiesen. Auf diesem Gebiet sind mit ASMs Erfolge erzielt worden, die anderen Formalismen in der gleichen Zeitdauer verwehrt blieben. Prominente Vertreter sind hierzu die formalen Spezifikationen von SDL-2000 [EGG+01] und Prolog [BR94] sowie einige weitere (s. ASM-Website [ASMa]).

Aktuelle Forschungsthemen hinsichtlich ASMs sind Validierungs- und Verifikationsmethoden. Erste Ansätze verfolgen das Ziel eingeschränktere Klassen von ASMs mit spezifischen Eigenschaften zu definieren [Now03]. In einem weitereren Ansatz wird ein Theorem-Beweiser entwickelt, der spezifizierte Eigenschaften anhand der ASM-Definition beweist. Wir können auch auf einige Implementationen von Abstract State Machines zurückgreifen. Allen Ansätzen gemein ist die Ausführbarkeit der spezifizierten Systeme (besonders interessant ASMGofer [ASMb] und AsmL [Asmc]). Einige Werkzeuge bieten eine Anbindung an Model Checker wie SMV, um die ausführbare Spezifikation zu verifizieren [DW00].

Für eine Gesamtübersicht zum Thema Abstract State Machines sei noch einmal die ASM-Website [ASMa] empfohlen.

6.3 Übersetzung von Strukturen

Ein wichtiger Schritt bei der Erstellung der Semantik ist die Übersetzung der in BPEL4WS vorhandenen Strukturen. Wie wir bereits in Abschnitt 6.1 erläutert haben, wollen wir eine allgemeine Formalisierung der Semantik dadurch erlangen, dass wir prozess-spezifische Konstrukte der Sprache von allgemeingültigen Konstrukten in BPEL4WS trennen. Wir ziehen diese Trennlinie hierbei auch durch die der Semantik zugrundeliegenden Strukturen. Die Frage, wo wir diese trennen, und wie wir diese Trennung übersetzen, wollen wir am Anfang des Abschnitts betrachten.

Aktive Komponenten eines Systems werden im ASM-Formalismus in ASM-Regeln dargestellt, passive durch Funktionen. Beide sind über die Verwendung von Funktionen in prädikatenlogischen Bedingungen einerseits und Σ-Updates andererseits miteinander verknüpft.

Dem ASM-Ansatz ist eigen, dass die in den ASM-Regeln formalisierten Teile einer Spezifikation stets allgemeingültig sind. Sie kommen in jedem Ablauf der ASM zum Tragen. Unterschiedliche Abläufe entstehen durch die Wahl des Anfangszustandes. Werden hier Funktionen unterschiedlich definiert, so hat dies Auswirkungen auf die anzuwendenden Regeln und die beteiligten Σ-Updates und somit auf den Ablauf. Daraus folgt, dass alle Teile einer Spezifikation, die die allgemeinen Konstrukte des Systems spezialisieren, nur im Anfangszustand durch entsprechende Funktionsdefinitionen formalisiert werden können.

Dieser Ansatz zur formalen Beschreibung der spezialisierenden Konstrukte greift auch bei den passiven Komponenten des Systems. Wie oben beschrieben, nutzen wir durch ASM-Regeln in einem Ablauf immer wieder neu definierte Funktionen. Wir spezifizieren sie auf der gleichen Ebene wie die Regeln und können dann über Verkettung mit den spezialisierenden Funktionen auch konkrete passive Komponenten beschreiben.

Spezialisierende Konstrukte von BPEL4WS sind in unserem Ansatz alle prozessspezifischen Konstrukte. Genauer gesagt, handelt es sich hierbei um die in der BPEL4WS-Quelltextdatei gegebene Prozess-Definition. Wenn wir die in XML-Form gespeicherten Informationen adäquat in einen entsprechenden Anfangszustand übersetzen, so stehen diese Informationen dann auch den ASM-Regeln zur Verfügung. Die kanonische Übersetzung wollen wir ab Abschnitt 6.3.1 charakterisieren. Da sich die im Quelltext definierten Strukturen während der Ausführung eines Prozesses niemals ändern, nennen wir sie *statische Strukturen*. Die im ASM-Modell definierten Funktionen wollen wir ebenfalls statisch nennen.

Die passiven allgemeingültigen Komponenten der Sprache sind implizit durch die Semantik gegeben. Wie wir bereits weiter oben festgestellt haben, lassen sie sich nicht durch Σ-Updates ausdrücken, sondern durch Funktionen. Zu diesen Komponenten gehören z.B. ausführungsbedingte Zusammenhänge wie Variablenwerte oder der Zustand des Kontrollflusses. Aus diesem Grund wollen wir derartige Strukturen *dynamisch* nennen. Ihre Übersetzung ist nicht kanonisch. Vielmehr werden wir sie explizit an den Stellen definieren, wo ihre Verwendung einsichtig ist. Analog zu den statischen Funktionen nennen wir Funktionen der dynamischen Struktur ebenfalls dynamisch.

6.3.1 Statische Strukturen

Die XML-Repräsentation der BPEL4WS-Prozess-Definition bildet eine sehr gute Grundlage für die Übersetzung in den Anfangszustand anhand von Funktionen, Konstanten und (Teil-)Universen. Wir geben nun einfache Übersetzungsregeln an, anhand derer ein Anfangszustand konstruiert werden kann, der der BPEL4WS-Prozess-Definition entspricht.

XML-Elemente implizieren die Existenz eines eigenen bestimmten Elementes aus einer bestimmten Menge in jedem gültigen Anfangszustand. Wir unterteilen die Trägermenge unserer Algebra zum besseren Verständnis in mehrere disjunkte, endliche und unendliche Universen mit selbsterklärenden Namen. Wir achten bei der Namensgebung im Folgenden weniger auf eine direkte Übersetzung der Objektnamen in den informalen Dokumenten, als auf ihre Funktion.

Als Beispiel diene die Übersetzung von Nachrichtentypen (Message Types) und deren Unterstrukturen (Message Type Parts). Die XML-Notation ist in Listing 6.1 gegeben.

Listing 6.1: Grammatik für Message Type und Message Type Part

```
<message name="..">
  <part name=".." type=".." />*
</message>
```

Tag	Universum
`<message... />`	**Universe**: $MessageType$
`<part... />`	**Universe**: $MessageTypePart$

Jedes `message`-Element und jedes `part`-Element wird durch ein eigenes Element aus der Menge $MessageType$ bzw. $MessageTypePart$ repräsentiert. Betrachten wir folgende konkrete Definition zweier Message Types in Listing 6.2.

Listing 6.2: Beispieldefinition zweier Message Types

```
<message name="my:controlMT">
    <part name="id" type="xsd:integer">
    <part name="cString" type="xsd:string">
</message>

<message name="my:questionMT">
   <part name="id" type="xsd:integer">
   <part name="question" type="my:question">
</message>
```

Für unsere Übersetzung in den Anfangszustand sind die Existenz zweier verschiedener Message Types und die Existenz von insgesamt vier paarweise verschiedenen Message Type Parts relevant. Somit muss in jedem geeigneten Anfangszustand

$$\exists mt_1, mt_2 \in MessageType\ \exists mp_1, \ldots, mp_4 \in MessageTypePart : \\ mt_1 \neq mt_2 \wedge \bigwedge_{i,j \in [4], i \neq j} mp_i \neq mp_j$$

gelten. Wir verwenden hierbei $[n]$ als Abkürzung für die Menge $\{1, \ldots, n\}$.
Kommt eine weitere Message Type Definition hinzu, so müssen wir auch weiterhin sicherstellen, dass alle zu übersetzenden Elemente paarweise verschieden sind.

In einigen Fällen ist die Abbildung der XML-Elemente in Elemente des Anfangszustands auch durch Attribute des XML-Elementes bestimmt. So bilden beispielsweise alle Elemente mit dem Tag `<fault .../>` und dem selben `name`-Attributwert auf das selbe Element ab, auch wenn die `fault`-Elemente mehrfach definiert werden.

Tag	Universum
`<fault name=".." ... />`	**Universe**: $FaultName$

Mit dieser Regel ist die Spezifikation des Anfangszustandes analog zu der für die Message Types. Die *Baum-Struktur* des XML-Dokumentes impliziert Abhängigkeiten zwischen den

Vater- und Kind-Elementen. Diese Abhängigkeiten lassen sich durch Funktionen darstellen. In einer konkreten Übersetzung muss die Funktion so definiert sein, dass sie jeweils von dem Element, das das Vater-Element repräsentiert, auf die Elemente des Trägers abbildet, die die zugehörigen Kind-Elemente repräsentieren.

Die Übersetzung lässt sich damit analog zu den XML-Elementen auch anhand von Tags (und in einigen Fällen auch mit Attributen entsprechend) charakterisieren. Für die Definition der Message Types ergibt sich folgende Abbildung zwischen den Message Types und den Message Type Parts.

$$messageTypeParts : MessageType \to \mathcal{P}(MessageTypePart)$$

Tags	Funktionen
`<message...>`	
`<part .../>+`	*messageTypeParts*
`</messsage>`	

Für das obige Beispiel erweitern wir somit die Anforderung an den Anfangszustand um die Beziehungen zwischen den Message Types un deren jeweiligen Kind-Elementen.

$$\begin{aligned}
&\exists mt_1, mt_2 \in MessageType\ \exists mp_1, \ldots, mp_4 \in MessageTypePart : \\
&\quad mt_1 \neq mt_2 \wedge \bigwedge_{i,j \in [4], i \neq j} mp_i \neq mp_j \\
&\quad \wedge\ messageTypeParts(mt_1) = \{mp_1, mp_2\} \\
&\quad \wedge\ messageTypeParts(mt_2) = \{mp_3, mp_4\}
\end{aligned}$$

Die *Attribute der Elemente* erzeugen Abhängigkeiten außerhalb der Baumstruktur, lassen sich jedoch genauso direkt übersetzen. Auch hier müssen Argumente und Werte der Funktion den entsprechenden Elementen der zugehörigen XML-Elemente genügen.

Wir erweitern unser Beispiel nun auf die Definition von Variablen. Mit der Definition wird jeder Variablen ein Datentyp – gegeben als Message Type, XML-Schema-Type oder elementarer Typ – zugeordnet. Wir betrachten im Folgenden ledoglich die Typisierung durch Message Types, da wir nur diese später benötigen. In den anderen Fällen ist die Formalisierung analog. Listing 6.3 zeigt die Syntax dafür.

Listing 6.3: Grammatik für Variablendefinition

```
<variables>
   <variable name=".." messageType=".." .../>*
</variables>
```

Es ergibt sich folgende Funktion und die entsprechende Übersetzungsvorschrift.

$$variableMsgType : Variable \to MessageType$$

Tags/Eigenschaft	Universum/Funktionen
`<variable .../>`	**Universe**: *Variable*
`<variable messageType=".." .../>`	*variableMsgType*

In einigen Fällen ist die funktionale Abhängigkeit durch mehrere XML-Elemente und Attribute gegeben, so dass die abbildenden Funktionen einen mehrdimensionalen Definitionsbereich haben.

Nehmen wir zu der obigen Definition der Message Types diese Variablendeklaration hinzu (Listing 6.4).

Listing 6.4: Beispiel für Variablendefinition

```
<variables>
   <variable name="varQuestion" messageType="def:questionMT" />
   <variable name="varStop" messageType="def:controlMT" />
</variables>
```

Die Anforderung an den Anfangszustand erweitert sich nun um zwei voneinander verschiedene Variablen und die Abbildung auf deren Message Types.

$$\begin{aligned}
&\exists mt_1, mt_2 \in MessageType\ \exists mp_1, \dots, mp_4 \in MessageTypePart :\\
&\quad mt_1 \neq mt_2 \wedge \bigwedge_{i,j \in [4], i \neq j} mp_i \neq mp_j\\
&\quad \wedge\ messageTypeParts(mt_1) = \{mp_1, mp_2\}\\
&\quad \wedge\ messageTypeParts(mt_2) = \{mp_3, mp_4\}\\
&\wedge\ \exists var_1, var_2 \in Variable :\\
&\quad var_1 \neq var_2 \wedge \bigwedge_{i \in [2]} variableMsgType(var_i) = mt_i
\end{aligned}$$

Wir haben hierbei die Spezifikation für die Message Type mit der für die Variablen lediglich konjunktiv verknüpft. Die Komposition der Anforderungen an den Anfangszustand können wir stets durch Konjunktion der Teilanforderungen ausdrücken. Einzig die quantifizierten Variablen müssen stets den Anforderungen an die paarweise Verschiedenheit genügen.

Wir haben nicht alle Attribute der als Beispiel dargelegten XML-Elemente übersetzt. Dies geschah nicht aus Nachlässigkeit, sondern weil es nicht notwendig ist. Die ausgelassenen Attribute geben Namen oder Typ-Definitionen an.

`name`-Attribute dienen im XML-Schema von BPEL4WS der eindeutigen Referenzierung von Elementen. Diese Referenzierung stellen wir im Anfangszustand explizit durch Abbildungen von Elementen des Trägers auf Elemente des Trägers dar. Die `name`-Attribute dienen uns hierbei während der Übersetzung zur Identifikation der beteiligten XML-Elemente und deren Repräsentation im Anfangszustand – sind aber *im* Anfangszustand selbst nicht von Nöten.

Typ-Attribute (wie bei Message Type Parts) bilden die Grenze der Abstraktion von BPEL4WS selbst. Daher gehen wir darauf nun gesondert ein.

6.3.2 Abstraktion von Daten

Im Folgenden wollen wir darlegen, weshalb Typ-Definitionen und Variablenwerte gar nicht bzw. nur sehr abstrakt modelliert werden. Variablenwerte sind natürlich Gegenstand eines

Ablaufs des Systems. Aus diesem Grund definieren wir im Folgenden auch dynamische Strukturen.

Ziel von BPEL4WS ist es, ausführbare Geschäftsprozesse zu definieren. Dazu gehört auch das Lesen, Schreiben und Manipulieren von komplexen Daten. Basierend auf dem WS-Technology-Stack verwendet BPEL4WS für die Definition und Manipulation eine tiefere Schicht des Stacks. Beides geschieht über geeignete Schnittstellen. Uns kommt es an dieser Stelle darauf an, die Schnittstelle zu modellieren und nicht die dahinterliegende Technologie. Was dies im Einzelnen bedeutet, soll nun erklärt werden.

Für unser Vorhaben kommt uns der ASM-Ansatz erneut zu Hilfe. ASMs betrachten Werte und ihre Gleichheit ausschließlich anhand der Gleichheit von interpretierten Termen, nicht anhand der Gleichheit von Zeichenketten. Gleiches gilt auch bei der Modellierung von Variablenwerten und Datentypen. Ein Variablenwert – egal wie komplex – ist ein einzelnes Element, repräsentiert durch ein abstraktes Element des Teiluniversums $Value$:

Universe: $Value$

Wäre es nötig die Unterstruktur des Wertes zu kennen, so erreichten wir dies mit Hilfe von Funktionen, die die Datentypen in allen Einzelheiten modellieren. In BPEL4WS werden strukturierte Daten stets direkt manipuliert. Die Unterstruktur wird dabei anhand des zugehörigen, strukturierten Datentyps identifiziert. Da Datentypen in BPEL4WS Bäume sind, definiert ein Pfad von der Wurzel zu einem Knoten in diesem Baum eindeutig eine bestimmte Unterstruktur. Funktionen, die diesen Pfad als Argument haben, können so die Werte der Unterstruktur manipulieren, ohne die ganze Struktur zu kennen.

Unter der Annahme, dass alle Manipulationen korrekt bezüglich des Datentyps sind, können wir von den Datentypen und den Zugriffen abstrahieren: Die Modellierung endet auf der Ebene der Message Type Parts. Das heisst, hier befindet sich die letzte explizite Abbildung auf Variablenwerte einer Struktur. Wir wollen dies am Beipsiel von Nachrichten (**Universe**: $BPELMsg$), die zwischen Web Services ausgetauscht werden, zeigen. (Näheres zu diesem Thema findet sich in Abschnitt 6.4.) Jede Nachricht hält für jeden Message Type Part ihres Message Types einen (abstrakten) Wert.

$$\begin{aligned} msgMsgType &: & BPELMsg &\rightarrow MessageType \\ msgPartValue &: & BPELMsg \times MsgTypePart &\rightarrow Value \end{aligned}$$

Die Funktionen $msgPartValue$ und $msgMsgType$ sind Beispiele für dynamische Funktionen: es ist klar, dass Nachrichten ausgetauscht werden, und dass deren Werte eine innere Struktur haben. Welchen Typs die Nachricht ist, kann nur während der Laufzeit definiert werden, da sie erst zu diesem Zeitpunkt überhaupt existiert. Aber damit wird automatisch festgelegt welche innere Struktur sie hat ($msgMsgType$ und $msgTypeParts$) und welche Werte sie zur Laufzeit aufnehmen kann ($msgPartValue$).

Bisweilen ist es nötig, die oben beschriebene Abstraktion zu überwinden und tiefer in die Datenstruktur und ihre Werte zu schauen. Die dafür zuständige Schicht des WS-Technology-Stacks wird aus BPEL4WS mittels Funktionsaufrufen und Parametern angesprochen. Syntaktisch ist dies als Attribut-Wert der manipulierenden Aktivitäten gegeben. Die Parameter der Funktionen, also die Pfade zu den Teilstrukturen des Datentyps, sind durch ihre String-Repräsentation eindeutig gegeben: Zwei gleiche Strings im XML-Text werden durch das

selbe Objekt (**Universe**: $QueryPath$) dargestellt. Die Funktionsaufrufe selbst modellieren wir im ASM-Modell mit abstrakten Funktionen, von denen wir annehmen, dass sie anhand des abstrakten Datenpfades Unterstrukturen korrekt lesen und schreiben. Eine nähere Betrachtung dieser Funktionen findet hier jedoch nicht statt.

6.3.3 Aktivitätsbaum

So, wie das gesamte XML-Dokument, sind auch die Aktivitäten in einer Baum-Struktur definiert. Diese Baum-Struktur impliziert die Aufruf- und Aufführungsbeziehungen der Aktivitäten. Die Wurzel ist ein Scope, syntaktisch gegeben durch das `process`-Tag. Die elementaren Aktivitäten bilden die Blätter des Baumes, da sie keine Kinder mehr haben können.

Folgende Funktionen ergeben sich ganz intuitiv aus der Baumstruktur der Aktivitäten.

$$\begin{aligned} activityParent &: \quad Activity \rightarrow Activity \\ activityChilds &: \quad Activity \rightarrow \mathcal{P}(Activity) \end{aligned}$$

Jedes XML-Element (und damit jede Aktivität) hat genau ein Eltern-Element (bis auf die Wurzel des Baumes). Jedes XML-Element kann mehrere Kind-Elemente haben. Wir heben den Aktivitätsbaum an dieser Stelle explizit hervor, da er bei der Spezifizierung der Aktivitäten noch eine Rolle spielen wird.

6.4 Übersetzung des Verhaltens

Im vorangegangenen Abschnitt 6.3 haben wir die einzelnen Teile der Semantik bezüglich aktiver und passiver sowie allgemeiner und spezialisierender Konstrukte separiert. Weiterhin wissen wir, wie wir die spezialsierenden Konstrukte für unseren Ansatz in statische Strukturen übersetzen. Diese stehen uns nun als entsprechend definierte Funktionen im Anfangszustand für jeden beliebigen BPEL4WS-Prozess zur Verfügung.

Was uns für unsere Semantik noch fehlt sind die allgemeinen aktiven und passiven Teile der Sprache. Beides wollen wir in diesem Abschnitt an Beispielen formalisieren. Dabei beziehen wir die Ergebnisse des vorigen Abschnitts mit ein. Wir übersetzen das Verhalten der Aktivitäten in ASM-Regeln so, dass die einmal definierten Regeln in jedem Spezialfall (lies: in jedem entsprechenden Anfangszustand) gelten. Die Bindung an den Spezialfall gelingt uns über die Funktionssymbole und Signaturen der statischen Strukturen, von deren möglichen konkreten Definitionen wir somit abstrahieren.

Die für die aktiven Teile notwendigen passiven Elemente der Sprache, repräsentiert durch dynamische Strukturen, werden wir parallel mit den ASM-Regeln formalisieren, da sich die dynamischen Strukturen implizit aus dem Verhalten ergeben. Wir vollziehen diesen Ansatz exemplarisch an einer strukturierten Aktivität (Sequence, 6.4.3) und einer elementaren Aktivität (Reply, 6.4.4). Dabei erklären wir auch die hierarchischen Aufrufmechanismen der Aktivitäten. Dies alles steht im Zusammenhang mit der Ausführung eines BPEL4WS-Prozesses, welche wir zunächst charakterisieren wollen.

6.4.1 Ausführung eines BPEL4WS-Prozesses

Was wir bis jetzt von BPEL4WS kennengelernt haben, ist lediglich die statische Struktur einer Prozess-Definition. Diese selbst ist nicht ausführbar. Wir wollen nun im Folgenden beschreiben, wie ein BPEL4WS-Prozess im allgemeinen und in der ASM-Semantik im Speziellen ausgeführt wird.

Ähnlich wie in der Objektorientierten Programmierung wird ein BPEL4WS-Prozess in Instanzen des Prozesses ausgeführt. Allerdings ist der Instanzbegriff hier ein anderer. Eine *BPEL4WS-Prozess-Instanz* zu bilden, bedeutet nicht, die Prozess-Definition zu kopieren, mit Variablenwerten zu belegen und dann in sich auszuführen. Der Ansatz von BPEL4WS ist, die Prozess-Definition als Muster zu verwenden und sich Variablenwerte und Zustand des Kontrollflusses, etc. explizit zu merken. Hintergrund dessen ist, dass die Ausführung eines solchen Prozesses durchaus lange auf externe Ereignisse, z.B. Nachrichten, wartet. Während des Wartens werden die Instanz-spezifischen Daten in einer Datenbank zwischengespeichert und bei Bedarf wieder geladen.

Die Prozess-Definition dient also als Grundgerüst für die darauf ausgeführten Instanzen. Wir wollen nun diesen Ansatz mit ASMs modellieren. Um Instanzen zu unterscheiden, führen wir die Menge der Prozess-Instanzen ein:

Universe: $ProcessInstance$

Die statische Struktur ist bereits bekannt und in den Anfangszustand übersetzt. Wir können auf die statischen Strukturen einfach über die bereits in Abschnitt 6.3 definierten Funktionen zugreifen. In einer Instanz tragen einige der statisch definierten Elemente instanzspezifische Werte, z.B. Variablen und Variablenwerte. Um diesen Elementen einen Wert zuzuweisen, ist lediglich eine Abbildung von der entsprechenden Prozess-Instanz und dem statischen Element auf den Wert nötig:

$$variablePartValue \quad : \quad ProcessInstance \times Variable \times MsgTypePart \quad \rightarrow \quad Value$$

Die Funktion $variablePartValue$ ist das Analogon zu $msgPartValue$ aus Abschnitt 6.3.2 für Variablen. Ein wenig anders ist die Modellierung der Aktivitäten aus Instanzsicht. Während ihre Strukturen ganz genauso die ausführungsbedingten Werte erhalten, ist die Ausführung einer Aktivität in einer Prozess-Instanz völlig unabhängig von der Ausführung der gleichen Aktivität in einer zweiten.

Diese Unabhängigkeit der Ausführung lässt sich innerhalb der ASMs nur durch je einen Agenten modellieren: Für jede Aktivität existiert in jeder Prozess-Instanz ein eigener Agent. Dieser Agent kennt die statische Definition der Aktivität, die er ausführt und die Prozess-Instanz in der er diese Aktivität ausführt. In ASM-Regeln sind Agenten über ihren Namen repräsentiert. Daher ergeben sich diese beiden Abbildungen:

$$\begin{aligned} \mathit{myStatic} &: \mathit{Agent} \rightarrow \mathit{Activity} \\ \mathit{currentInstance} &: \mathit{Agent} \rightarrow \mathit{ProcessInstance} \end{aligned}$$

Diese Abbildungen sind die einzige Verbindung zwischen einem Agenten und dem Teil eines BPEL4WS-Prozesses, den er ausführt. Sie werden zur Laufzeit definiert, wenn die entsprechende Prozess-Instanz erzeugt wird.

Der Mechanismus der Instanz-Erzeugung ist ein wenig kompliziert. Prinzipiell erzeugt eine eintreffende Nachricht eine neue Prozess-Instanz. Sie muss hierbei von einer „instanzerzeugenden“ Aktivität empfangen werden. Die Nachricht kann aber erst empfangen werden, wenn die Instanz bereits existiert, da die empfangende Aktivität bereits auf die Nachricht warten muss. Gelöst wird dieses Problem durch eine unterliegende Middleware, die wir nicht näher betrachten wollen. Nur so viel sei gesagt: die Middleware empfängt zunächst alle Nachrichten für einen Prozess und erzeugt dann abhängig davon neue Instanzen, die im Anschluss die Nachricht selbst vom Framework empfangen können.

6.4.2 Aktivitätszustände

Bevor wir die Aktivitäten spezifizieren, gehen wir noch einmal auf ihr Verhältnis zu den ASM-Agenten ein. Jede Aktivität wird durch genau einen Agenten ausgeführt. Aus der Sicht von BPEL4WS wartet eine Aktivität mit ihrer Ausführung, bis sie die Erlaubnis dazu hat, das heißt *aktiviert* ist. Diese Aktivierung geschieht in BPEL4WS aufgrund des Blockkonzepts streng hierarchisch. Eine Aktivität wird ausschließlich durch die Vater-Aktivität aktiviert. Damit isolieren wir Kontrollflussbeziehungen lokal anhand der Vater-Kind-Beziehungen (s. auch Aktivitätsbaum in Abschnitt 6.3.3).

Im Gegensatz dazu sind ASM-Agenten nicht hierarchisch organisiert, sie kennen einander nicht einmal. Jeder ASM-Agent macht einen Schritt unabhängig vom Verhalten des restlichen Systems: ein ASM-Agent kann jederzeit einen Schritt machen. Wir haben also zwei Aufgaben zu lösen: Wir müssen erstens die Ausführung einer Aktivität durch einen Agenten so weit einschränken, dass die Aktivität auf eine Aktivierung wartet, während der Agent Schritte ohne „Wirkung“ vollzieht. Dies erreichen wir durch einen Zustandsautomaten, den wir – in ASM-Regeln modelliert – den eigentlichen Aktivitätsregeln vorschalten. Als zweites sind die hierarchischen Aufrufbeziehungen zu modellieren. Diese ergeben sich jedoch aus dem Aktivitätsbaum und dem Zustandsautomaten, wenn wir zulassen, dass eine Aktivität den Zustandsautomaten der Kindaktivitäten in Teilen steuern kann.

Wir beginnen zunächst mit dem Zustandsautomaten. Sechs Zustände genügen für unsere Semantik: der Initialzustand *disabled*, der aktivierte Initialzustand *enabled*, *running* als Zustand während der Ausführung der Aktivität; *stopping* für das vorzeitige Terminieren im Fehlerfall und die Endzustände *stopped* (fehlerhaft beendet) und *completed* (fehlerfrei beendet). Jede Aktivität befindet sich in jeder Prozess-Instanz in einem dieser Zustände (endliches Teiluniversum *ActivityState*), modelliert durch die dynamische Funktion *activityState*.

Universe: $ActivityState = \{disabled, enabled, running, stopping, stopped, completed\}$

$$activityState \quad : \quad ProcessInstance \times Activity \quad \rightarrow \quad ActivityState$$

Für jede Aktivität $a \in Activity$ und jede Prozess-Instanz $pI \in ProcessInstance$ gilt im Anfangszustand

$$activityState(pI, a) = disabled. \tag{6.4}$$

Der Zustandsautomat kontrolliert nun, in welchen Zustand die Aktivität versetzt wird, abhängig von ihrem aktuellen Zustand und äußeren Bedingungen. Im Allgemeinen verfügt jede ASM über einen Zustandsautomaten dieses Musters:

if $activityState(\mathsf{pl}, \mathsf{self}) = enabled$ **then**
 StartActivity
if $activityState(\mathsf{pl}, \mathsf{self}) = running$ **then**
 RunActivity
if $activityState(\mathsf{pl}, \mathsf{self}) = stopping$ **then**
 StopActivity

Keinesfalls darf es eine ASM-Regel für eine Aktivität geben die einen Zustandsübergang für diese Aktivität aus einem der Zustände *disabled*, *completed* oder *stopped* modelliert. Zustandsänderungen durch die Aktivität selbst sind also nur aus den Zuständen *enabled*, *running* und *stopping* zulässig. Zusammen mit der Bedingung (6.4) an *activityState* für den Anfangszustand kann somit zunächst keine Aktivität des gesamten Prozesses irgendetwas tun.

Wir wissen nun genug über den Zustandsautomaten und die Kontrolle der Aktivitätsausführung, um die hierarchischen Aufrufbeziehungen zu modellieren.

Die Zustände einer Aktivität können von der Vateraktivität direkt verändert werden. Dies ist in zwei Fällen möglich.

1. Zum Aufrufen einer Kindaktivität, so dass diese ausgeführt wird.
2. Zum vorzeitigen Terminieren der Ausführung einer Kindaktivität im Fehlerfall.

Gleichzeitig überwacht die Vateraktivität ständig den Zustand der Kindaktivität, um so die weitere Ausführung zu steuern. Diese Steuerungsbeschränkung durch die Vateraktivität definieren wir rein semantisch mit ASM-Regeln, unterstützt durch den Aktivitätsbaum, der Beziehungen ersten Grades direkt hierarchisch abbildet.

Die gesamte Ausführung einer Prozess-Instanz ist somit hierarchisch geregelt. Allerdings muss diese Ausführung ersteinmal angestoßen werden. Dies geschieht wiederum durch die, dem Web Service zugrundliegende, Middleware. Sie erzeugt aufgrund eines aktivierenden Ereignisses, z.B. einer eintreffenden Nachricht, eine neue Prozess-Instanz und versetzt in dieser den äußersten Scope, und damit die Wurzel-Aktivität des Aktivitätsbaumes in den Zustand *enabled*.

Die restliche Ausführung der Prozess-Instanz ist dann allein von den Vater-Kind-Beziehungen der Aktivitäten abhänging. Erreicht die Wurzel-Aktivität den Zustand *completed*, so ist die Ausführung dieser Prozess-Instanz beendet.

Links

Der obige Modellierungsansatz betont das strukturierte Blockkonzept von BPEL4WS. Wir müssen jedoch auch die graph-orientierte Beschreibung in unsere Semantik integrieren. BPEL4WS stellt uns hierfür die Aktivität Flow zur Verfügung, deren Kindaktivitäten nebenläufig ausgeführt werden. Ist T ein Teilbaum des Aktivitätbaumes und die Wurzel von T ist ein Flow, so darf zwischen zwei Knoten von T (außer der Wurzel) eine Einschränkung der Ausführungsreihenfolge definiert werden. Dies geschieht mittels des *Link*-Konzepts.

Ein Link legt zwischen einer *Source*-Aktivität und einer von dieser verschiedenen *Target*-Aktivtät fest, dass die Target-Aktivität erst aktiviert wird, wenn die Source-Aktivität

bereits vollständig abgearbeitet (im Zustand *completed*) ist. Wir nennen einen *Link aktiviert*, wenn seine Source-Aktivität abgearbeitet ist. In unserer Formalisierung von BPEL4WS setzen wir das Link-Konzept durch eine weitere Bedingung beim Zustandsübergang von *enabled* nach *running* (*StartActivity*) um.

Das Link-Konzept ergänzt also die hierarchischen Aufrufbeziehungen. Die Schnittstelle hierfür werden wir an den geeigneten Stellen am Beispiel erklären.

Wir wissen nun genug darüber, wie BPEL4WS-Prozesse im Allgemeinen ausgeführt werden. Mit den globalen Zusammenhängen als Grundlage werden wir die Semantik zweier Aktivitäten (Sequence und Reply) im lokalen Kontext modellieren.

6.4.3 Sequence

Die Aktivität Sequence (Tag: `sequence`) ist eine strukturierte Aktivtät. Wir haben sie hier ausgewählt, weil sich anhand ihrer Semantik sehr schön das hierarchische Zusammenspiel der Aktivitäten zeigt. Die Sequence steuert die sequentielle Ausführung ihrer Kindaktivitäten.

Die Syntax der Sequence ist in Listing 6.5 gegeben, wobei *`activity`* für jede beliebige Aktivität stehe.

Listing 6.5: Syntax der Sequence

```
<sequence ...>
  activity+
</sequence>
```

Die Reihenfolge der Kind-Elemente gibt auch die Reihenfolge der Ausführung der Kindaktivitäten an. Diese Abfolge modellieren wir mit einer Liste:

$$\begin{array}{rcl} seqActivityNext & : & Activity \rightarrow Activity \\ seqActivityFirst & : & Sequence \rightarrow Activity \end{array}$$

seqActivityFirst bildet auf die erste Aktivität der Sequence ab. *seqActivityNext* bildet von einer Aktivität auf die Folge-Aktivität in der Sequence ab. Ist *seqActivityNext* undefiniert, so ist das Ende der Liste und damit der Sequence erreicht. *seqActivityFirst* und *seqActivityNext* sind statische Funktionen und entsprechend im Anfangszustand definiert.

Zunächst besitzt jedoch auch die Sequence vor der Ausführung der Semantik den sie steuernden Zustandsautomaten.

```
ExecuteSequence
    (self) ≡
  let seq = myStatic(self) in
    let pl = currentInstance(self) in
      if activityState(pl, seq) = enabled then
        StartActivity(pl, seq)
      if activityState(pl, seq) = running
        RunSequence(pl, seq)
      if activityState(pl, seq) = stopping
```

$\textsc{StopStructured}(\mathsf{pI}, \mathsf{seq})$

Im Normalfall geht eine Sequence direkt von *enabled* nach *running* über. Es gibt zwei Ausnahmen, wo dies nicht geschieht. Im ersten Fall erreicht das Signal zur vorzeitigen Terminierung *activityStop* die Aktivität. Dieses Signal, modelliert als dynamische Funktion, legt für eine Aktivität in einer Prozess-Instanz fest, ob diese Aktivität vorzeitig terminiert werden soll. In diesem Fall darf die sie gar nicht erst ausgeführt werden.

$$activityStop \quad : \quad ProcessInstance \times Activity \quad \rightarrow \quad \{true, false\}$$

In jedem Anfangszustand ist für jede Aktivität *act* und jede Prozess-Instanz pI $activityStop(pI, act) = false$ definiert.

Im zweiten Fall verhindert ein noch nicht aktivierter Link den übergang. Den Zustand der Links wollen wir hier nur mittels einer abstrakten Funktion ermitteln:

$$allLinksEnabled \quad : \quad ProcessInstance \times Activity \quad \rightarrow \quad \{true, false\}$$

bildet auf *true* ab, falls für diese Aktivität *act* in der Prozess-Instanz pI alle Links, die *act* als Target-Aktivität haben, aktiviert wurden; andernfalls auf $false$.

$\underline{\textsc{StartActivity}}$
  $(\mathsf{pI} \in ProcessInstance, \mathsf{act} \in Activity) \equiv$
if $activityStop(\mathsf{pI}, \mathsf{act}) = true$ **then**
  $activityState(\mathsf{pI}, \mathsf{act}) := stopping$
else if $allLinksEnabled(pI, \mathsf{act}) = true$ **then**
  $activityState(\mathsf{pI}, \mathsf{act}) := running$

Die Kontrolle darüber, welche Kindaktivität als nächstes ausgeführt wird obliegt im ASM-Modell einzig der Sequence. Sie muss sich somit auch merken, welche Aktivität gerade die ausgeführte ist.

$$seqCurrentActivity \quad : \quad ProcessInstance \times Sequence \quad \rightarrow \quad Activity$$

seqCurrentActivity bildet als dynamische Funktion stets auf die aktuell ausgeführte Kind-Aktivität ab und ist zu Anfang für alle Prozess-Instanzen und Sequences undefiniert.

$\underline{\textsc{RunSequence}}$
  $(\mathsf{pI} \in ProcessInstance, \mathsf{seq} \in Sequence) \equiv$
if $activityStop(\mathsf{pI}, \mathsf{seq}) = true$ **then**
  $activityState(\mathsf{pI}, \mathsf{seq}) := stopping$
else
  let $\mathsf{current} = seqCurrentActivity(\mathsf{pI}, \mathsf{seq})$ **in**
    if $\mathsf{current} = undef$ **then**
      let $\mathsf{first} = seqActivityFirst(\mathsf{seq})$ **in**
        $\textsc{EnableSequenceChild}(\mathsf{pI}, \mathsf{seq}, \mathsf{first})$
    else if $activityState(\mathsf{pI}, \mathsf{current}) = completed$ **then**
      let $\mathsf{next} = seqActivityNext(\mathsf{current})$ **in**
        if $\mathsf{next} \neq undef$ **then**

$\quad\quad\quad\quad \textsc{EnableSequenceChild}(\mathsf{pl}, \mathsf{seq}, \mathsf{next})$
$\quad\quad\quad \textbf{else}$
$\quad\quad\quad\quad \mathit{activityState}(\mathsf{pl}, \mathsf{seq}) := \mathit{completed}$
$\quad\quad\quad\quad \textsc{ActivateLinks}(\mathsf{pl}, \mathsf{seq})$

Eine Kind-Aktivtät wird aktiviert, in dem sie selbst auf *enabled* gesetzt wird. Gleichzeitig wird zur überwachung der Verweis der Sequence auf die Kind-Aktivität (mittels *seqCurrentActivity*) neu definiert. Dies modelliert folgende Regel:

$\underline{\textsc{EnableSequenceChild}}$
$\quad\quad (\mathsf{pl} \in \mathit{ProcessInstance}, \mathsf{seq} \in \mathit{Sequence}, \mathsf{act} \in \mathit{Activity}) \equiv$
$\quad \mathit{seqCurrentActivity}(\mathsf{pl}, \mathsf{seq}) := \mathsf{act}$
$\quad \mathit{activityState}(\mathsf{pl}, \mathsf{act}) := \mathit{enabled}$

Erreicht die Sequence den Zustand *completed* (in RunSequence), so muss sie noch alle Links, deren Source-Aktivität sie ist, aktivieren. Wir nehmen an, dass dies die Regel ActivateLinks spezifiziert und gehen nicht näher darauf ein.

Ganz analog zur Aktivierung der Kindaktivitäten ist deren vorzeitige Terminierung definiert. Eine Aktivität kann gezwungen werden vorzeitig zu terminieren, wenn Fehler in der Prozess-Instanz auftreten. Hierbei unterbricht jede Aktivität, die das Stop-Signal (*activityStop*) erhält ihre eigene Ausführung und sendet es weiter an ihre Kindaktivitäten. Die Aktivität wartet anschließend darauf, dass alle Kindaktivitäten entweder bereits vollständig ausgeführt wurden (*completed*), oder dass diese und deren Kindaktivitäten vorzeitig terminiert wurden (*stopped*), um dann selbst zu terminieren. Am Beispiel der Sequence wird dieses Verfahren deutlich.

$\underline{\textsc{StopStructured}}$
$\quad\quad (\mathsf{pl} \in \mathit{ProcessInstance}, \mathsf{str} \in \mathit{Activity}_{\mathit{structured}}) \equiv$
$\quad \textbf{if } \forall ch \in \mathit{activityChilds}(\mathsf{pl}, \mathsf{str}) : \mathit{activityState}(\mathsf{pl}, \mathsf{ch}) \in \{\mathit{stopped}, \mathit{completed}\} \textbf{ then}$
$\quad\quad \mathit{activityState}(\mathsf{pl}, \mathsf{str}) := \mathit{terminated}$
$\quad \textbf{else}$
$\quad\quad \textbf{forall } \mathsf{ch} \in \mathit{activityChilds}(\mathsf{str})$
$\quad\quad\quad\quad \textbf{where } \mathit{acitivityState}(\mathsf{pl}, \mathsf{ch}) \in \{\mathit{enabled}, \mathit{running}, \mathit{disabled}\} \textbf{ do}$
$\quad\quad\quad \mathit{activityState}(\mathsf{pl}, \mathsf{ch}) := \mathit{stopping}$

Für *alle strukturierten* Aktivitäten (**Universe**: $\mathit{Activity}_{\mathit{structured}} \subseteq \mathit{Activity}$) der Sprache mit Ausnahme des Scopes und der Fault-, Compensation- und Event-Handler, formalisiert StopStructured das Verhalten beim vorzeitigen Beenden der Aktivität. Gleiches gilt für StartActivity bezüglich *aller* Aktivitäten mit Ausnahme des Scopes und der genannten Handler und den Aktivitäten Pick und Receive, deren Aktivierung noch von externen Ereignissen abhängt. An dieser Stelle wird deutlich wie homogen sich die Aufruf- und Ausführungsbeziehungen in BPEL4WS durch ASMs formalisieren lassen.

Die in den Abschnitten 6.3, 6.4.1 und 6.4.2 definierten Funktionen, sowie *activityStop* und StartActivity und StopStructured bilden das Modul MIDDLEWARE. Es formalisiert das viel zitierte Grundgerüst, auf dem die Semantik der einzelnen Aktivitäten (als Einziges) aufbaut.

Alle in diesem Abschnitt definierten Regeln und Funktionen ergeben das Modul SEQUENCE. Die lediglich abstrakt definierten Funktionen sowie die Regel ActivateLinks werden imporiert. Gleiches gilt für alle Funktionen und ASM-Regeln von MIDDLEWARE. Damit formalisiert SEQUENCE ausschlie"lich Aspekte der Sequence. Ganz analog werden wir nun auch ein Modul für die elementare Aktivität Reply definieren.

6.4.4 Reply

Reply verschickt eine Nachricht als Antwort auf eine erhaltene Nachricht an den Absender (Partner). Als Inhalt wird der Nachricht der Inhalt einer Variable zugewiesen, die durch die Aktivität referenziert wird. Der Versand erfolgt über den Ausgabekanal der Operation, über deren Eingabekanal die vorausgegangene Nachricht einging. Anstatt einer Nachricht darf auch eine Fehlermeldung als Antwort verschickt werden. Listing 6.6 zeigt die Syntax für Reply.

Listing 6.6: Syntax des Replys

```
<reply variable=".." partnerLink=".." portType=".."
       operation=".." faultName=".."?>
   <correlations>?
      <correlation set=".." initiate="yes|no"?>+
   </correlations>
</reply>
```

Aus diesen statischen Definitionen ergeben sich unmittelbar folgende Universen und Funktionen, die – wie in Abschnitt 6.3.1 beschrieben – im Anfangszustand aufgrund eines konkreten BPEL4WS-Prozesses definiert sind.

Universe: *Reply*
Universe: $PartnerLink$
Universe: $PortType$
Universe: *Operation*
Universe: $FaultName$
Universe: $CorrelationSet$

$$\begin{aligned}
replyPartnerLink &: Reply \rightarrow PartnerLink\\
replyPortType &: Reply \rightarrow PortType\\
replyOperation &: Reply \rightarrow Operation\\
replyVariable &: Reply \rightarrow Variable\\
replyFaultName &: Reply \rightarrow FaultName\\
replyCorrelationSets &: Reply \rightarrow \mathcal{P}(CorrelationSet)
\end{aligned}$$

Dazu gilt auch für Reply die allgemeine Struktur für Aktivitäts-Regeln.

<u>ExecuteReply</u>
(self) $\equiv$
let reply = $myStatic$(self) **in**
let pl = $currentInstance$(self) **in**

if $activityState$(pI, reply) $=$ $enabled$ **then**
StartActivity(pI, reply)
if $activityState$(pI, reply) $=$ $running$
RunReply(pI, reply)
if $activityState$(pI, reply) $=$ $stopping$
StopReply(pI, reply)

Die Aktivierung von Reply ist bereits mit StartActivity in Abschnitt 6.4.3 vollständig beschrieben. Wir können also direkt das Verhalten betrachten.

Reply kann nicht in jedem Fall eine Nachricht versenden. Ein spezieller Mechanismus, genannt *Correlation Handling* schränkt den Versand von Nachrichten ein. So dürfen nur Nachrichten versandt bzw. empfangen werden, deren Schlüsselfelder einen bestimmten Wert enthalten. Sowohl die Schlüsselfelder, als auch die Festlegung der Werte werden über das Correlation Handling realisiert. Genügt eine Nachricht nicht den gegebenen Anforderungen, so wird innerhalb des Prozesses ein Fehler ($correlationViolation$) geworfen. Vom Correlation Handling wollen wir an dieser Stelle abstrahieren. Daher verwenden wir hier abstrakte Funktionen, von denen wir annehmen, dass sie gemäß der informalen Semantik den Zustand verändern.

$$correlationSatisfied_{var} : ProcessInstance \times Variable \times \mathcal{P}(CorrelationSet) \rightarrow \{true, false\}$$

Die Funktion $correlationSatisfied_{var}$ gebe $true$ zurück, wenn die Schlüsselfelder, die durch die Correlation Sets gegeben sind, mit den Variablenwerten in der aktuellen Prozess-Instanz übereinstimmen. Gleichzeitig werden noch undefinierte Schlüsselfelder mit den entsprechenden Werten belegt.

<u>ExecuteReply</u>
(pI $\in$ $ProcessInstance$, reply $\in$ $Reply$) $\equiv$
if $activityStop$(pI, reply) $=$ $false$ **then**
let var $=$ $replyVariable$(reply) **in**
let CS $=$ $replyCorrelationSets$(reply) **in**
if $correlationSatisfied_{var}$(pI, var, CS) $=$ $true$ **then**
SendReplyMessage(pI, reply)
$activityState$(pI, reply) := $completed$
else
Throw(pI, reply, $correlationViolation$)
else
$activityState$(pI, reply) := $stopping$

Die Regel Throw spezifiziert das Auftreten eines Fehlers und das „Werfen“ des selben an den umgebenden Scope, der diesen Fehler dann fängt. Der gesamte Fehlerbehandlungsmechanismus ist bei Weitem zu komplex, um ihn hier zu formalisieren. Daher nehmen wir auch von Throw an, dass es gemäß der informalen Semantik arbeitet.

Jede von einem BPEL4WS-Prozess empfangene oder versandte Nachricht ist einzigartig. Daher werden wir auch im ASM-Modell jede zu verschickende Nachricht neu erzeugen.

Anhand des optional definierten Fehler-Namens ($replyFaultName$) ist spezifiziert, ob eine reguläre Nachricht oder eine Fehlernachricht versandt wird. Letztere sind direkt als „Fehler“ definiert und sind jeweils genauso einzigartig. Es existieren für beide Nachrichtenklassen eigene, undendliche große Universen:

Universe: $BPELMsg$
Universe: $Fault$

Aus der Existenz der Nachricht im ASM-Modell folgt noch nicht ihr Inhalt. Diesen erhält sie über dynamische Funktionen, spezialisiert durch statische Funktionen, wie schon in Abschnitt 6.3.2 beschrieben. Fehlernachrichten werden durch Namen ($FaultName$) klassifiziert. Die Funktion $faultFName$ (dynamisch) leistet dies.

$$faultFName \; : \; Fault \; \rightarrow \; FaultName$$

Aus diesen zwei Aspekten der Nachricht im ASM-Modell (Existenz und Inhalt) folgt auch, wie wir den Versand modellieren. Sowohl beim Senden einer regulären Nachricht, als auch einer Fehlernachricht erzeugen wir diese, weisen ihr den Inhalt zu und versenden sie. Letzteres ist für uns im Modell unabhängig von der Zuweisung des Nachrichteninhalts. Daher sind diese beiden Aspekte in zwei parallel komponierten Regeln modelliert.

SendReplyMessage
  (pI $\in ProcessInstance$, reply $\in Reply$) $\equiv$

if $replyFaultName$(reply) $= undef$ **then**
  let message $= new(BPELMsg)$ **in**
    CopyVariableToMsg(pI, $replyVariable$(reply), message)
    SendMessage$_{reply}$(pI, $replyPartnerLink$(reply),
        $replyPortType$(reply), $replyOperation$(reply), message)
else
  let fault $= new(Fault)$ **in**
    CopyVariableToMsg(pI, $replyVariable$(reply), fault)
    $faultFName$(fault) $:= replyFaultName$(reply)
    SendMessage$_{fault}$(pI, reply, fault)

Der Begriff des *Nachrichtenkanals* entsteht durch die Kombination zweier Sichten. Zunächst existiert die *physische Sicht*, die aus der Perspektive eines Web Service auf die physische Adresse der Middleware des anderen Web Service, mit dem Nachrichten ausgetauscht werden, zeigt. Diese Sicht trennt nicht die Kommunikation zwischen zwei bestimmten Instanzen von BPEL4WS-Prozessen oder Web Services. über einen derart beschriebenen physischen Kanal tauschen alle Instanzen der beteiligten Web Services Nachrichten aus. Die zweite Sicht definiert einen *logischen Kanal*, in dem sie für jede Nachricht definiert, welche Instanz des jeweiligen Web Service als Absender bzw. Empfänger fungiert. Wir modellieren beide Sichten.

Die physische Sicht erfordert die Beschreibung von *Adressen* (**Universe**: *Address*) und die Abbildung auf die Middlewareadresse des an der Kommunikation beteiligten Web Service. Dieser ist über den *PartnerLink* (*replyPartnerLink*) des Replys formalisiert. Die statisch definierten PartnerLinks werden erst zur Laufzeit an konkrete Web Services und

damit an Adressen gebunden. Daher modellieren wir die physische Sicht des Kanals mit der dynamischen Funktion

$$partnerAddress \; : \; ProcessInstance \times PartnerLink \;\; \rightarrow \;\; Address$$

Wir haben bis jetzt nicht betrachtet, wie ein PartnerLink an einen konkreten Web Service gebunden wird und wollen dies auch nicht tun. Deshalb verstehen wir *partnerAddress* als abstrakte Funktion, die uns in der gegebenen Prozess-Instanz für einen (definierten) PartnerLink die zugehörige physische Adresse liefert.

Wir sind nun in der Lage den physischen Aspekt des Nachrichtenkanals zu modellieren: der Ausgabe- und der Eingabekanal einer Operation ordnet jeder Operation in dem zugehörigen PortType an der Adresse der Middleware eine Menge von (zu empfangenden) Nachrichten zu.

$$\begin{array}{lcll} portMessages_{reply} & : & Address \times PortType \times Operation & \rightarrow \;\; \mathcal{P}(BPELMsg) \\ portMessages_{fault} & : & Address \times PortType \times Operation & \rightarrow \;\; \mathcal{P}(Fault) \end{array}$$

Die logische Sicht des Kanals ist eine Abbildung der Nachrichten auf Konstrukte, die die Instanz des Absenders und des Empfängers entsprechend identifizieren (vgl. [CB+03]). Aus Gründen der Komplexität der Konstrukte verwenden wir hier nur die Regel SetEndpoints, ohne sie näher zu definieren. Wir nehmen an, dass sie den logischen Kanal für die versandte Nachricht korrekt konstruiert.

Analog zur Existenz einer Nachricht und deren Inhalt sind auch die physische und die logische Sicht des Kanals unabhängig voneinander. Wir modellieren beide Aspekte des Sendens als parallele komponierte Regeln.

$\text{SendMessage}_{reply}$
$(\mathsf{pI} \in ProcessInstance, \mathsf{partnerL} \in PartnerLink, \mathsf{pT} \in PortType$
$\mathsf{op} \in Operation, \mathsf{msg} \in BPELMsg) \equiv$

let address $= partnerAddress(\mathsf{pI}, \mathsf{partnerL})$ **in**
SetEndpoints(pI, partnerL, msg)
$portMessages_{reply}(\mathsf{address}, \mathsf{pT}, \mathsf{op}) :=$
$portMessages_{reply}(\mathsf{address}, \mathsf{pT}, \mathsf{op}) \cup \{\mathsf{msg}\}$

$\text{SendMessage}_{fault}$
$(\mathsf{pI} \in ProcessInstance, \mathsf{partnerL} \in PartnerLink, \mathsf{pT} \in PortType$
$\mathsf{op} \in Operation, \mathsf{fault} \in Fault) \equiv$

let address $= senderAddress(\mathsf{pI}, \mathsf{partnerL})$ **in**
SetEndpoints(pI, partnerL, fault)
$portMessages_{fault}(\mathsf{address}, \mathsf{pT}, \mathsf{op}) :=$
$portMessages_{fault}(\mathsf{address}, \mathsf{pT}, \mathsf{op}) \cup \{\mathsf{fault}\}$

Wir haben bis zu dieser Stelle einigen Aufwand betrieben, um Nachrichten und ihre Inhalte zu separieren sowie Werte von Variablen möglichst abstrakt zu betrachten. Dies wird sich im Rest des Abschnittes auszahlen, da wir nun modellieren, wie eine Nachricht ihren Inhalt

erhält. Dabei bekommen wir noch einmal einen überblick über die Konzepte der ASM-Semantik.

Die an der Zuweisung beteiligten Komponenten des Systems sind die zu versendende Nachricht msg, die Variable var, von der wir unseren Inhalt beziehen und das reply, das die Zuweisung des Variableninhalts an die Nachricht in seiner aktuellen Prozess-Instanz pI spezifiziert. An dieser Stelle treffen noch einmal ASM-Regeln, dynamische und statische Strukturen aufeinander.

Die für die Zuweisung nötigen statischen Strukturen beschreiben die Typ-Definition von Variable und Nachricht durch $MessageType$, $MessageTypeParts$ und $msgTypeParts$ (vgl. Abschnitt 6.3.1). Die Variable ist typisiert durch $variableMsgType$. Die Nachricht ist ein dynamisch erzeugtes Objekt, sie wird durch $msgMsgType$, einer folgerichtig dynamischen Funktion, typisiert (vgl. 6.3.2).

Nachricht und Variable tragen ihre Werte anhand von dynamischen Funktionen, abhängig von der statischen Struktur ihres Typs. Die Funktionen $msgPartValue$ und $variablePartValue$ haben wir ebenfalls bereits definiert.

Das Verhalten eines jeden Replys bewirkt, dass die Nachricht die selben (statischen) Typ-Informationen wie die Variable erhält und entsprechend der Typisierung die Nachricht die gleichen Werte, wie die Variable trägt. Hierfür weisen wir abhängig von der im Anfangszustand definierten statischen Funktion $msgTypeParts$ die Werte der Substrukturen der Variable als Werte der Substrukturen der Nachricht zu. Die dem Reply zugeordnete Variable muss durch einen Message Type typisiert sein – von dieser statischen Eigenschaft gehen wir hier aus.

COPYVARIABLETOMSG
    $(\mathsf{pI} \in ProcessInstance, \mathsf{var} \in Variable, \mathsf{msg} \in BPELMsg) \equiv$

  if $\mathsf{var} \neq undef$ **then**
    forall $\mathsf{part} \in msgTypeParts(messageType(\mathsf{var}))$ **do**
      $msgPartValue(\mathsf{msg}, \mathsf{part}) := variablePartValue(\mathsf{pI}, \mathsf{var}, \mathsf{part})$
    $msgMsgType(\mathsf{msg}) := variableMsgType(\mathsf{var})$

Alle in diesem Abschnitt definierten Regeln und Funktionen ergeben das Modul REPLY. Die Regel STARTACTIVITY wird vom Modul MIDDLEWARE importiert, ebenso die Funktion *activityStop* und alle weiteren dort definierten und hier verwendeten Funktionen. Des weiteren werden alle abstrakt betrachteten Funktionen und Regeln importiert. REPLY ist völlig unabhängig von SEQUENCE und umgekehrt. Das Zusammenspiel zwischen beiden ergibt sich nur durch MIDDLEWARE.

Die drei in dieser Arbeit definierten Module bilden einen Teil der ASM-Definition für die formale Semantik von BPEL4WS.

6.5 Schlußfolgerungen und Ausblick

Wir haben in diesem Papier an Beispielen gezeigt, welchen Weg wir bei der Formalisierung der Semantik für BPEL4WS mit ASMs einschlagen. Die grundsätzliche Idee ist, die statische Struktur des Prozesses in den Anfangszustand der ASM zu übersetzen. Diese statische Struktur bedingt die Ausführung der allgemein formulierten ASM-Regeln für die

Aktivitäten. Die Ausführung des konkreten Prozesses ist somit vollständig von den ASM-Regeln entkoppelt. Diese wiederum sind in jedem BPEL4WS-Prozess gültig. Eigenschaften der Sprache lassen sich damit auf einer ganz abstrakten Ebene analysieren und beweisen.

Durch die Modularisierung können wir jede Komponente der Sprache (jede Aktivität) einzeln betrachten. Gleichzeitig formalisieren wir auch die Interaktion zwischen den Komponenten in einem eigenen Modul. Mögliche Fehler in der Semantik lassen sich somit eindeutig einem Konzept von BPEL4WS zuordnen.

Was wir hier nicht gezeigt haben, ist neben der Semantik der restlichen Aktivitäten die Modellierung der Fehler-, Kompensations- und Ereignisbehandlung. Daraus ergibt sich auch die Notwendigkeit den Begriff der Prozess-Instanz und der Ausführung dieser zu verfeinern. Ebenfalls noch zu modellieren ist die Middleware, die den Nachrichtenaustausch beschreibt und über die Prozess-Instanzen wacht. Letzteres wird demnächst in einem eigenen Papier veröffentlicht werden.

7 Eine intuitive Graphik für BPEL4WS

Autorin: Daniela Weinberg

7.1 Graphik als Chance

Bilder spielen in unserem Leben eine große und zentrale Rolle. Wir tendieren stets dazu, zunächst ein Bild zu erstellen. Sei es in einem Brainstormingprozess, wo als erstes Ideen gesammelt und bildhaft sortiert werden. Oder aber schlicht in experimentellen Anordnungen der Naturwissenschaften, welche oftmals mit Aufzeichnungsgeräten oder Bildschirmen verbunden sind. Letztlich versuchen wir ein Bild zu erstellen, welches unsere Vorstellungen bzw. Erkenntnisse modelliert.

Wir sehen in der graphischen Darstellung eines Geschäftsprozesses die Chance, einfacher über diesen Prozess reden zu können. Eine bildhafte Repräsentation eines Prozesses kann außerdem auch dazu dienen, zunächst unsichtbare Strukturen sichtbar zu machen. Denn selbst für geübte Menschen ist es sicherlich schwer, einen langen Prozess, welcher in XML Form notiert ist, komplett zu verstehen und alle Strukturen zu überblicken. Oftmals werden die konkreten Strukturen innerhalb eines Prozesses erst durch ein Bild klar. So können beispielsweise Kontroll- und Nachrichtenflüsse rasch nachvollzogen werden, was innerhalb eines XML-Baumes durchaus ein Problem darstellen kann.

Es gibt derzeit keine einheitliche Darstellung eines in BPEL4WS notierten Prozesses. Ein kurzer Blick in das Internet unter dem Gesichtspunkt, mehr über die Geschäftsprozesssprache BPEL4WS zu erfahren, zeigt eine Flut verschiedenster graphischer Darstellungen von Geschäftsprozessen. Im Wesentlichen sollen diese Graphiken dem Leser dazu dienen, den beschriebenen Prozess besser zu verstehen. Hierbei verwendet allerdings jeder Autor eine andere Art der Veranschaulichung.

Wir haben eine graphische Repräsentation der Sprache BPEL4WS entwickelt, die es ermöglicht, jeglichen in dieser Sprache notierten Prozess darzustellen. Jedes Element der Sprache hat eine entsprechende Symbolik. Somit ist die Erstellung einer graphischen Darstellung des Prozesses einfach anhand des XML-Quelltextes möglich. Eine solche Modellierung wollen wir im Abschnitt 7.2.1 mittels eines kurzen Beispiels vornehmen und dabei einige wichtige graphische Elemente vorstellen. Im anschließenden Abschnitt 7.3 werden wir die Konzepte der graphischen Repräsentation einiger ausgewählter Elemente der Sprache BPEL4WS betrachten.

7.2 visualBPEL

Wir wollen, wie bereits beschrieben, eine Möglichkeit schaffen, jeden in BPEL4WS notierten Prozess schnell und präzise graphisch darzustellen. Diese Graphik soll dabei einfach erfassbar sein, ohne dass der Leser die Sprache BPEL4WS selbst kennen muss. Auch soll die graphische Repräsentation möglichst selbsterklärend sein, so dass ein Vorabstudium aller

graphischen Elemente nicht bzw. kaum von Nöten ist. Wir haben hierzu einige Konventionen getroffen:

Prozess Der zu modellierende Geschäftsprozess wird durch ein einfaches Rechteck dargestellt. In dieses Rechteck werden die Elemente des Prozesses notiert. Die Partner des Prozesses werden an der linken bzw. rechten Seite des Rechteckes dargestellt.

Partner Web-Services, welche als Kunden angesehen werden, notieren wir links und jene Web-Services, deren Dienstleistung in Anspruch genommen wird, werden rechts von dem darzustellenden Prozess aus gezeichnet.

Aktivitäten Jegliche Aktivitäten stellen wir stets durch weiße Quadrate dar. Einem solchen Quadrat kann ein weiteres graphisches Element, wie zum Beispiel ein Pfeil, hinzugefügt werden. Dadurch ist es möglich, jede BPEL4WS Aktivität einfach zu modellieren.

Kontrollfluss Den Kontrollfluss innerhalb des Geschäftsprozesses stellen wir von oben nach unten dar.

Datenmengen Die Datenmengen werden durch Ellipsen symbolisiert.

Doch bevor wir hier weiter ins Detail gehen, wollen wir uns ein kleines Beispiel ansehen.

7.2.1 Eine Reiseagentur

Stellen wir uns als erstes folgendes Szenario vor: ein viel beschäftigter Geschäftsmann muss oft von einer Stadt in eine andere reisen. Hierbei nutzt er, wenn es geht, ein Flugzeug. In jeder bereisten Stadt benötigt er dann ein Hotelzimmer und, um stets mobil zu sein, auch einen Mietwagen. Alle notwendigen Buchungen möchte er nun einer Reiseagentur übertragen, die alles für ihn ausführt. Im folgenden wollen wir jene Reiseagentur modellieren.

Beginnen wir als erstes mit dem Grundlegendsten: unsere Reiseagentur arbeitet mit vier verschiedenen Web-Services zusammen und bearbeitet den Auftrag des Kunden in einer Weise, die wir momentan noch nicht weiter spezifizieren wollen. Welche vier Web-Services werden hier benötigt? Zum ersten ist es der Kunde, welcher unseren Reiseservice in Anspruch nimmt. Dann muss die Reiseagentur mit drei Web-Services zusammenarbeiten, die letztlich für das eigentliche Buchen verantwortlich sind. Hierbei handelt es sich um den Airline-Service, den Hotel-Service und den Car-Rental-Service. Von diesen Services wird also eine Dienstleistung in Anspruch genommen. Wir können jetzt erst einmal die allgemeinste Variante unseres Prozesses darstellen (siehe Abbildung 7.1). Im Listing 7.1 ist der BPEL4WS-Quelltext dieser allgemeinsten Formulierung des Prozesses zu sehen.

Wie wir in der Abbildung 7.1 erkennen können, wurden die Schnittstellen zu den einzelnen Web-Services näher spezifiziert. Es wurde der `portType` und die damit assoziierten Operationen angegeben. Bei dem Service des Kunden (`businessMan`) wird der `portType` namens `bookRequestPT` mit der Operation `bookTrip` assoziiert. Des Weiteren ist es möglich, eine Fehlernachricht zu empfangen. Dies wird durch den roten Pfeil symbolisiert. Das

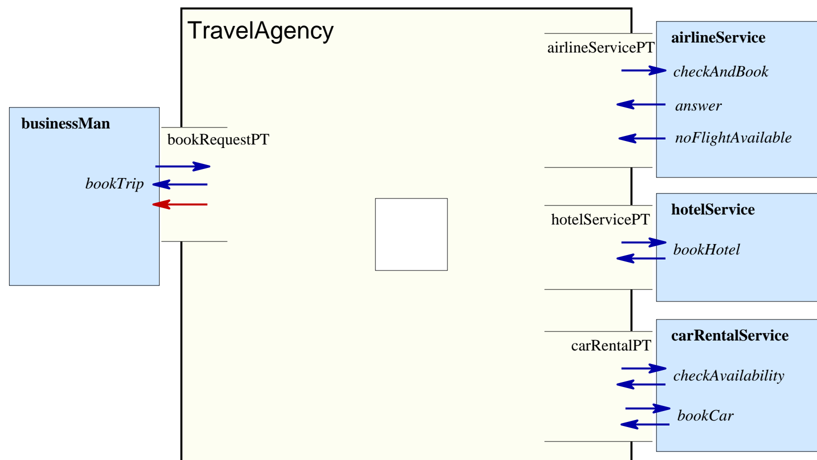


Abbildung 7.1: Der grobe Blick auf die Reiseagentur

Listing 7.2 zeigt den WSDL-Quelltext dieser `portType` Spezifikation. Analog haben wir auch die Modellierung der anderen drei Web-Services vorgenommen.

Listing 7.1: Eine grobe Sicht auf den Prozess

```
<process name="travelAgency"
      ...
     abstractProcess="no">
  <partners>
    <partner name="businessMan" myRole="travelAgency"/>
    <partner name="airlineService" myRole="ticketRequester"/>
    <partner name="hotelService" myRole="hotelTicketRequester"/>
    <partner name="carRentalService" myRole="carRequester"/>
  </partners>

  <variables>
    <variable name="bookRequest" messageType="bookRequestType"/>
    <variable name="bookConfirm" messageType="bookConfirmType"/>
    <variable name="error" messageType="errorType"/>
  </variables>

  activity

</process>
```

Listing 7.2: `portType` Spezifikation des Kunden

```
<portType name="bookRequestPT">
  <operation name="bookTrip">
    <input message="bookTripMsg"/>
    <output message="getDetailsMsg"/>
    <fault name="noFlight" message="noFlightMsg"/>
  </operation>
</portType>
```

Das Bild unserer Reiseagentur zeigt zentral ein weißes Quadrat. Damit haben wir uns schon

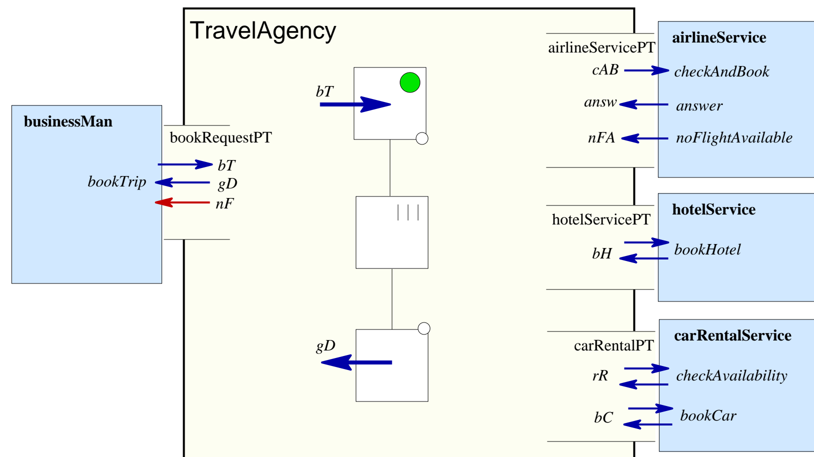


Abbildung 7.2: Kommunikation mit dem Kunden

im Abschnitt 7.2 kurz beschäftigt. Mit diesem Quadrat wollen wir zeigen, dass etwas passiert. Dabei ist uns zunächst in diesem Bild nicht wichtig, was im Speziellen getan wird. Da es jedoch unser Ziel ist, den gesamten Prozess zu modellieren, wollen wir einen Blick in dieses weiße Quadrat werfen. Was tut eigentlich unser Prozess nun mit den Daten, welche er von dem Kunden bekommt? Wie wird mit den anderen Web-Services zusammengearbeitet? Schauen wir zuerst auf die Kommunikation mit dem Kunden. Dieser ist ja der Ausgangspunkt für die Arbeit unserer Reiseagentur. Wir erhalten also eine Nachricht von ihm und die Daten, die für die Buchungen notwendig sind. Als nächstes werden diese Daten verarbeitet und schließlich wird der Kunde über unsere Arbeit informiert.

Schauen wir uns das Bild 7.2 genauer an. Wir sehen jetzt anstelle des weißen Quadrates drei miteinander verbundene leicht modifizierte weiße Quadrate. Das erste und das letzte ist, wie man vielleicht schon selbst hat folgern können, für die Kommunikation mit dem Kunden verantwortlich. Es handelt sich hier konkret um die Aktivitäten `receive` und `reply`. An den Pfeilen stehen kleine Abkürzungen. Durch diese Vereinbarung wollen wir auf zusätzliche Striche verzichten. Das bedeutet, Pfeile mit der gleichen Beschriftung werden miteinander verbunden. Wir verbinden also gedanklich die erste Aktivität (das `receive`) mit dem ausgehenden Pfeil (`bT`) der Operation `bookTrip` und die `reply` Aktivität mit dem eingehenden Pfeil (`gD`) dieser Operation. Der Quelltext von Abbildung 7.2 wird im Listing 7.3 näher spezifiziert.

Listing 7.3: Die äußere Sequenz

```
<sequence>
   <receive partner="businessMan"
            portType="bookRequestPT"
            operation="bookTrip"
            variable="bookRequest"
            createInstance="yes">
   </receive>
   <flow>
   ...
   </flow>
```

```
    <reply partner="businessMan"
           portType="bookRequestPT"
           operation="bookTrip"
           variable="bookConfirm">
    </reply>
</sequence>
```

Die mittlere Aktivität in dieser Sequenz ist ein Piktogramm für eine Menge von nebenläufigen Aktivitäten. In der Sprache BPEL4WS sprechen wir hier von einem `flow`. Mit jenem wollen wir uns nun weiter beschäftigen. Wir zoomen jetzt wieder weiter in die Tiefen unseres Geschäftsprozessen hinein. Was wird bzw. kann jetzt im Laufe der Bearbeitung unserer Kundenanfrage nebenläufig ausgeführt werden? Zunächst einmal ist es nur sinnvoll ein Hotelzimmer zu buchen, wenn ein Flug an den gewünschten Reisetagen erhältlich ist. Die Anfrage, ob ein Auto in einer speziellen Kategorie in diesem Zeitraum verfügbar ist, ist allerdings eher unabhängig vom Buchen des Flugtickets. Die konkrete Bestellung des Wagens sollte jedoch erst dann getätigt werden, wenn der Flug reserviert wurde. Daher werden die Buchung des Flugtickets und des Hotelzimmers von der Reservierung des Mietwagens separat behandelt. Beschäftigen wir uns nun mit dem ersteren. Es wird eine Anfrage an den Airline-Service mit den Daten, die von dem Kunden erhalten wurden, geschickt. Der Airline-Service wird nun entweder einen entsprechenden Flug heraussuchen und buchen oder eine Meldung schicken, dass der gewünschte Flug nicht vorhanden bzw. ausgebucht ist. Im Falle einer positiven Antwort kann nun ein Hotelzimmer über den Hotel-Service gebucht werden. Der Einfachheit halber gehen wir hierbei davon aus, dass die Buchung eines Hotelzimmers immer klappt. Kann der Flug nicht gebucht werden, schickt der Prozess dem Kunden eine Meldung über diesen Fehler zurück.

In der Abbildung 7.3 sehen wir genau den eben beschriebenen Ablauf. Als erstes füh-

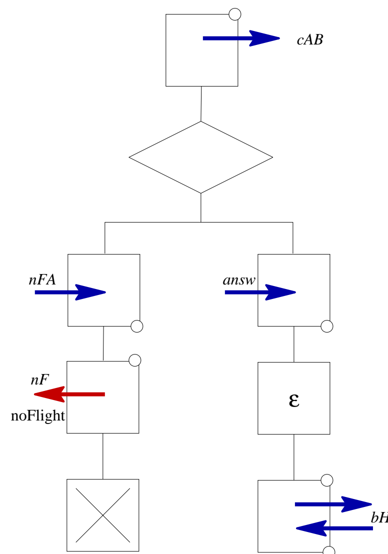


Abbildung 7.3: Buchung von Flug und Hotelzimmer

ren wir ein `invoke` aus und veranlassen somit den Airline-Service den gewünschten Flug herauszusuchen und ggf. zu buchen. Als nächstes warten wir auf eine der beiden möglichen Antworten: Buchung okay oder aber eine Fehlermeldung. Hierzu verwenden wir das `pick` Konstrukt. Wird eine Fehlermeldung empfangen, so wird dem Kunden mit Hilfe eines speziellen `reply` s eine Fehlernachricht geschickt. Wir sehen hier also den Unterschied zwischen einem üblichen `reply`, welches einen blauen ausgehenden Pfeil trägt und jenem `reply`, welches eine Fehlernachricht verschickt und daher einen roten Pfeil mit zusätzlicher Anschrift des Fehlers trägt. Schließlich wird dieser Zweig und somit auch der gesamte Prozess durch ein `terminate` beendet. Im Falle, dass alles gut geht und der entsprechende Flug gebucht werden konnte, wird lediglich noch das Hotelzimmer über den Hotel-Service reserviert. Die `emtpy` Aktivität benötigen wir später für die Synchronisation der beiden nebenläufig ablaufenden Sequenzen. Es wird den Ausgangspunkt für einen `link` bilden, auf den wir noch eingehen werden. Das Auto soll, wie schon gesagt, ja erst gebucht werden, wenn ein Flug reserviert werden konnte. Wie wir in diesem Bild weiterhin sehen können, gibt es zwei Darstellungen für ein `invoke`. Die Sprache BPEL4WS unterscheidet zwischen einem asynchronen und einem synchronen `invoke`. Diese beiden Arten werden dargestellt, indem man bei einem asynchronen `invoke` lediglich einen ausgehenden Pfeil notiert und bei einem synchronen `invoke` sowohl einen ausgehenden als auch einen eingehenden Pfeil notiert. Die Sequenz der eben beschriebenen Aktivitäten ist in Listing 7.4 aufgeführt.

Listing 7.4: Buchen des Flugtickets und des Hotels

```
<sequence>
  <invoke partner="airlineService"
     portType="airlineServicePT"
     operation="checkAndBook"
     inputVariable="bookRequest">
  </invoke>
  <pick>
    <onMessage partner="airlineService"
         portType="airlineServicePT"
         operation="noFlightAvailable"
         variable="error">
      <sequence>
         <reply partner="businessMan"
            portType="bookRequestPT"
            operation="bookTrip"
            variable="error"
            faultName="noFlight">
         </reply>
         <terminate>
         </terminate>
      </sequence>
    </onMessage>
    <onMessage partner="airlineService"
         portType="airlineServicePT"
         operation="answer"
         variable="bookConfirm">
      <sequence>
         <empty>
         </empty>
         <invoke partner="hotelService"
            portType="hotelServicePT"
            operation="bookHotel"
            inputVariable="bookRequest"
            outputVariable="bookConfirm">
```

```
        </invoke>
      </sequence>
    </onMessage>
  </pick>
</sequence>
```

Parallel zu dem eben beschriebenen Ablauf wird der Mietwagen reserviert. Hierbei wird anfangs bei dem Car-Rental-Service angefragt, ob ein Auto in der von dem Kunden gewünschten Kategorie in dem entsprechenden Zeitraum zur Verfügung steht. Ist dies nicht der Fall, nehmen wir an, dass der Service uns eine Nachricht zurücksendet, in der eine andere Kategorie steht. Mit dieser Information senden wir erneut eine Anfrage an den Service - nun aber mit der eben erhaltenen Kategorie. Wir iterieren so lange, bis wir ein Fahrzeug gefunden haben. Jetzt können wir diesen Wagen reservieren. Beide Anfragen realisieren wir durch ein synchrones `invoke`. Die Abbildung 7.4 zeigt diesen Vorgang. Der Quelltext für das Buchen des Mietwagens ist in Listing 7.5 zu sehen.

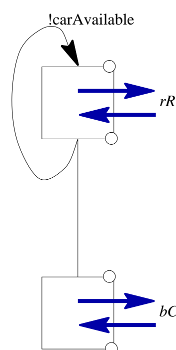


Abbildung 7.4: Reservierung eines Mietwagens

Listing 7.5: Buchung des Mietwagens

```
<sequence>
  <while condition="bookRequest.carAvailable=false">
    <invoke partner="carRentalService"
        portType="carRentalPT"
        operation="checkAvailability"
        inputVariable="bookRequest"
        outputVariable="bookRequest">
    </invoke>
  </while>
  <invoke partner="carRentalService"
      portType="carRentalPT"
      operation="bookCar"
      inputVariable="bookRequest"
      outputVariable="bookConfirm">
  </invoke>
</sequence>
```

Bislang haben wir allerdings außer Acht gelassen, dass es eine Abhängigkeit zwischen der Buchung des Flugtickets und der Reservierung des Mietwagens gibt. Diese Abhängigkeit

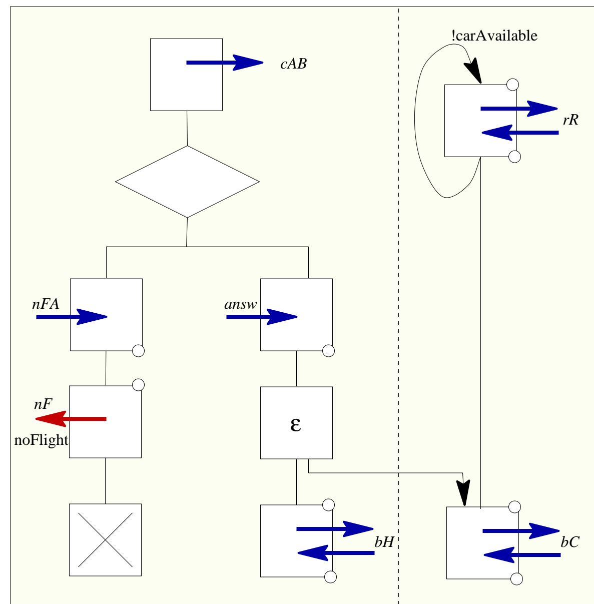


Abbildung 7.5: Die nebenläufigen Aktivitäten

wird durch einen `link` modelliert, der zwischen der positiven Antwort des Airline-Services und der Reservierung des Mietwagens gezogen wird. Um diesen `link` implementieren zu können, haben wir in der ersten Sequenz die `empty` Aktivität genutzt, da die `onMessage` Ereignisse innerhalb eines `pick` Konstruktes nicht Ausgangspunkte für `links` sein können.

In Abbildung 7.5 sehen wir nun den `flow`. Die beiden nebenläufigen Aktivitäten werden durch eine gestrichelte Linie optisch von einander getrennt. Den kompletten Quelltext der nebenläufigen Aktivitäten zeigt nun das Listing 7.6.

Listing 7.6: Die nebenläufigen Aktivitäten

```
<flow>
    <links>
        <link name="flightOkay-bookCar"/>
    </links>

  <sequence>
        <invoke partner="airlineService"
        portType="airlineServicePT"
        operation="checkAndBook"
        inputVariable="bookRequest">
    </invoke>
    <pick>
      <onMessage partner="airlineService"
            portType="airlineServicePT"
            operation="noFlightAvailable"
            variable="error">
        <sequence>
          <reply partner="businessMan"
              portType="bookRequestPT"
              operation="bookTrip"
              variable="error"
```

```
            faultName="noFlight">
          </reply>
          <terminate>
          </terminate>
        </sequence>
      </onMessage>
      <onMessage partner="airlineService"
          portType="airlineServicePT"
          operation="answer"
          variable="bookConfirm">
        <sequence>
        <empty>
          <source linkName="flightOkay-bookCar"/>
        </empty>
        <invoke partner="hotelService"
            portType="hotelServicePT"
            operation="bookHotel"
            inputVariable="bookRequest"
            outputVariable="bookConfirm">
        </invoke>
        </sequence>
      </onMessage>
    </pick>
  </sequence>
  <sequence>
    <while condition="bookRequest.carAvailable=false">
      <invoke partner="carRentalService"
          portType="carRentalPT"
          operation="checkAvailability"
          inputVariable="bookRequest"
          outputVariable="bookRequest">
      </invoke>
    </while>
    <invoke partner="carRentalService"
        portType="carRentalPT"
        operation="bookCar"
        inputVariable="bookRequest"
        outputVariable="bookConfirm">
        <target linkName="flightOkay-bookCar"/>
    </invoke>
  </sequence>
</flow>
```

Zu guter letzt wollen wir jetzt das Piktogramm für die nebenläufigen Aktivitäten aus Abbildung 7.2 durch das eben erarbeitete Bild ersetzen. Damit haben wir den gesamten Prozess der Reiseagentur modelliert (Abbildung 7.6).

Was wir bis dato außer Acht gelassen haben, ist der Datenfluss. Wo werden Variablen geschrieben? Wo werden Daten gelesen? Doch bevor wir uns damit näher beschäftigen, schauen wir uns doch noch einmal die Graphiken an. Wir haben nämlich noch gar nicht darüber gesprochen, was die kleinen Kreise an einigen Aktivitäten aussagen sollen. Diese kleinen Kreise an den rechten Seiten von Aktivitäten dienen als Verbindungsstelle zu Variablen bzw. Datenmengen. Hierbei ist die Anordnung der Kreise zu beachten. Liegt eine solche Verbindungsstelle an der oberen Seite der Aktivität, wird die damit verbundene Variable gelesen. Ist die Variable mit einem unteren Kreis verbunden, wird sie geschrieben.

Da in unserer Reiseagentur auch der Datenfluss eine Rolle spielt, wollen wir diesen ebenfalls in unserer Graphik vermerken und erhalten nun die Abbildung 7.7.

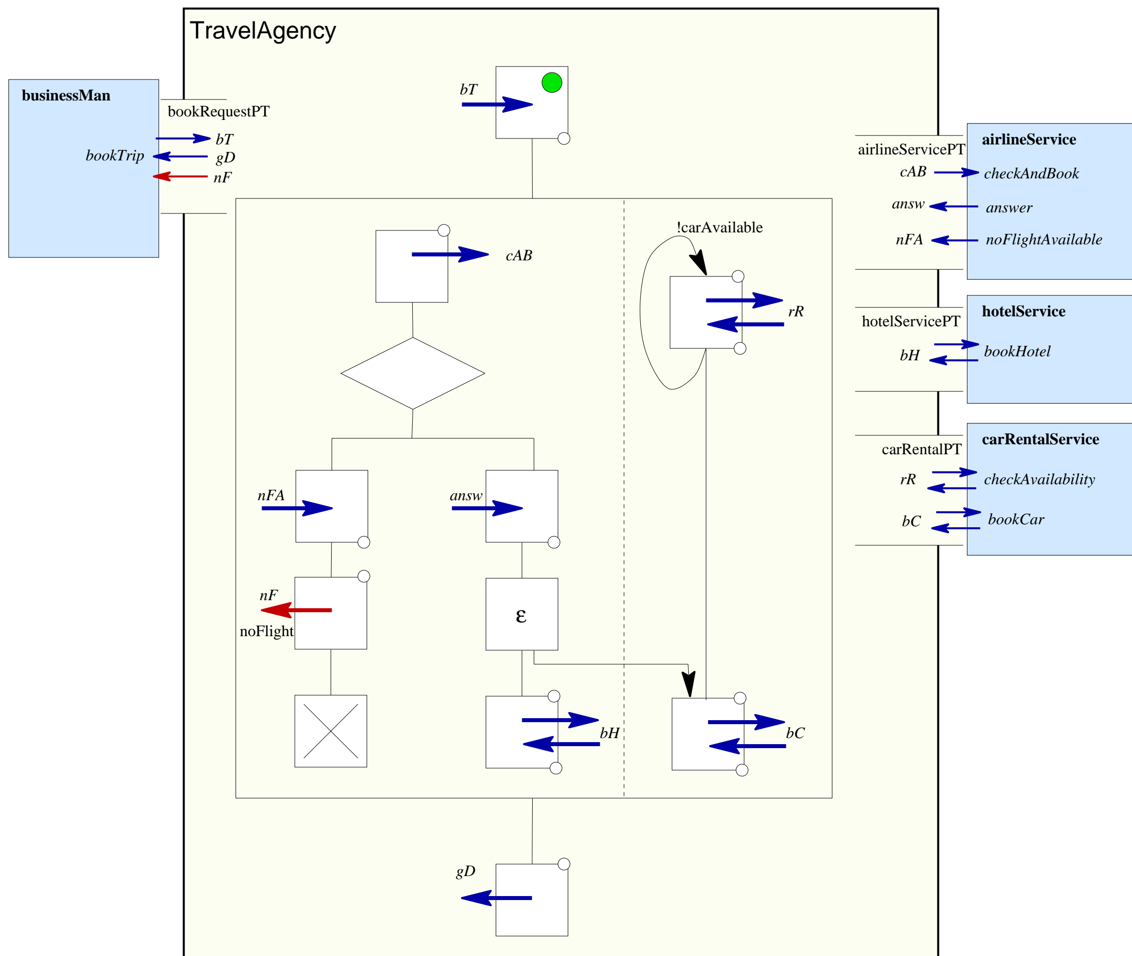


Abbildung 7.6: Der gesamte Prozess

Eine Datenmenge notieren wir stets als Ellipse, die den Namen der Variable in sich trägt.

7.3 Ausgewählte Elemente

In den folgenden Abschnitten wollen wir einige Elemente betrachten, die bereits im Beispiel Anwendung gefunden haben. Eine ausführlichere Beschreibung unter Einbeziehung aller BPEL4WS Elemente kann in [Wei03] nachgelesen werden.

7.3.1 Ein Prozess und seine Partner

Ein BPEL4WS Prozess wird durch eine Box dargestellt. Innerhalb dieser Box werden alle in dem Prozess definierten Elemente notiert. Die Abbildung 7.8 zeigt einen Prozess mit dem Namen „processName“ und einer beliebigen Aktivität ohne jeglicher Verbindung zu Geschäftspartnern.

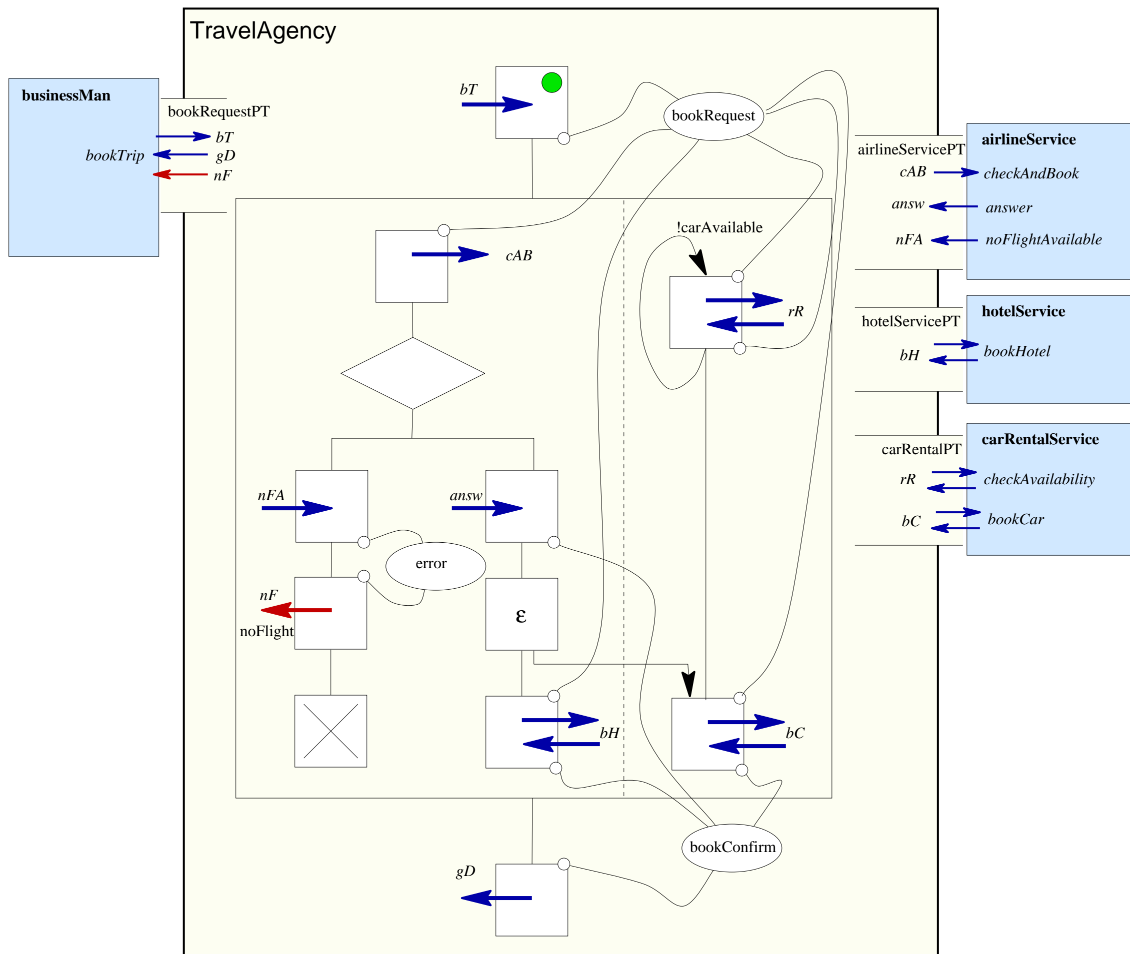


Abbildung 7.7: Der gesamte Prozess mit Datenfluss

Bei den Partnern eines Prozesses unterscheidet man jene, die als Kunden und jene die als Dienstleister fungieren. Die Kunden notiert man stets links und die Dienstleister rechts von unserem Prozess. Die Verbindung zwischen dem Partner und dem BPEL4WS-Prozess wird durch eine Box hergestellt, welche allerdings nur oben und unten durch Striche dargestellt ist. Die linke und rechte Seite bleibt offen und symbolisiert somit, dass hier Datenaustausch möglich ist. In diese Box wird oben der `portType` Name geschrieben, welcher hier dargestellt werden soll. Jeder `portType` wird mit mindestens einer `operation` assoziiert. Diese `operation` wird innerhalb der Partnerbox notiert. In Höhe der `operation` werden die Pfeile, welche die Eingabe-, Ausgabe- bzw. Fehlernachrichten symbolisieren, gezeichnet. Die Eingabe- und Ausgabenachrichtenverbindungen werden als etwas größere blaue Pfeile und die Fehlernachrichtenverbindung als roter Pfeil dargestellt.

Bei der Anbindung der Partner an den darzustellenden Prozess verlassen wir kurzzeitig unseren Anspruch den Quelltext 1:1 in der Graphik abbilden zu wollen. Wir müssen hier zunächst den Kontext der `input` und `output` Operationen erfassen und auf unseren Prozess

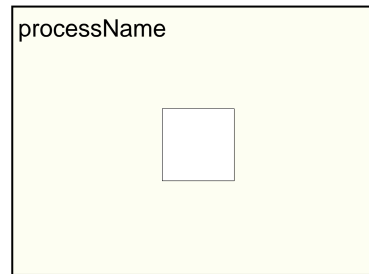


Abbildung 7.8: Ein Prozess mit einer beliebigen Aktivität

applizieren. Dementsprechend notieren wir die Pfeile an der Schnittstelle zwischen dem Partner und dem Prozess als ein- bzw. ausgehende Pfeile. Betrachten wir kurz folgendes Beispiel: In unserem Prozess ist ein `receive` deklariert. Dieses implementiert eine direkte `input` Operation für einen externen Web-Service. In diesem Falle notieren wir bei dem externen Web-Service einen Pfeil, der in Richtung unseres Prozesses zeigt und somit für den Partner als ausgehender Pfeil dargestellt wird. Die Abbildung 7.9 zeigt den Beispielprozess aus Abbildung 7.8 in Verbindung mit zwei Geschäftspartnern.

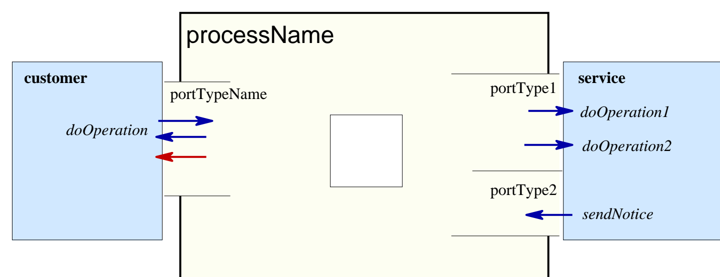


Abbildung 7.9: Ein Beispielprozess mit Geschäftspartnern

7.3.2 Kommunikation mit Partnern

Wie alle Aktivitäten werden auch die folgenden drei, welche für die Kommunikation mit anderen Web-Services genutzt werden, durch ein Quadrat dargestellt. Sie unterscheiden sich lediglich hinsichtlich der ein- bzw. ausgehenden blauen Pfeile. Diese Pfeile zeigen an, ob Nachrichten von einem Partner angenommen werden (eingehender Pfeil) oder ob Nachrichten zu einem Partner gesendet werden (ausgehender Pfeil). Die kleinen weißen Kreise an der rechten Seite des Quadrates deuten an, dass eine Variable gelesen bzw. geschrieben wird. Befindet sich der Kreis an der oberen Seite des Quadrates, wird aus der damit verbundenen Variable gelesen. Ist der Kreis an der unteren Seite, schreibt die Aktivität in der entsprechenden Variablen.

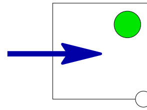


Abbildung 7.10: `receive`

receive Das `receive` verfügt zusätzlich über die Eigenschaft, einen Prozess instanziieren zu können. Dies wird durch einen kleinen grünen Kreis angedeutet, welcher optional in die obere rechte Hälfte des Quadrates notiert werden kann (s. Abbildung 7.10). Das graphische Element verfügt über einen eingehenden blauen Pfeil, da das `receive` Element Nachrichten empfangen kann.

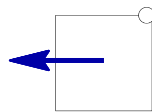


Abbildung 7.11: `reply`

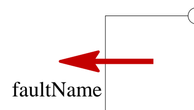


Abbildung 7.12: `reply` mit Fehlernachricht

reply Das graphische Element der Aktivität `reply` verfügt über einen ausgehenden blauen Pfeil, da es Nachrichten verschicken kann (s. Abbildung 7.11). Wird eine Fehlernachricht verschickt, notiert man einen roten Pfeil und versieht diesen zusätzlich mit dem Namen des Fehlers (s. Abbildung 7.12).

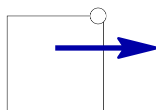


Abbildung 7.13: asynchrones `invoke`

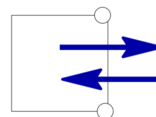


Abbildung 7.14: synchrones `invoke`

invoke Das `invoke` kommuniziert meist nur mit Dienstleistern (Server). Somit werden die von diesem Konstrukt ein- bzw. ausgehenden Pfeile auf der rechten Seite notiert. Diese Darstellung wurde deshalb gewählt, weil in der Prozessgraphik stets die Kunden links und die Dienstleister rechts gezeichnet werden. Je nach dem ob es sich bei dem darzustellenden `invoke` um ein synchrones oder asynchrones `invoke` handelt, notiert man nur einen ausgehenden Pfeil (asynchron, Abbildung 7.13) oder sowohl einen ausgehenden als auch einen eingehenden Pfeil (synchron, Abbildung 7.14).

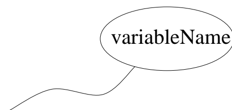


Abbildung 7.15: Variable mit einer Datenverbindung

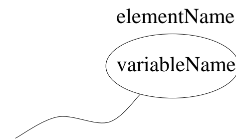


Abbildung 7.16: Variable mit Datenverbindung und gesetztem Attribut `element`

7.3.3 Datenmengen

Variablen sind passive Elemente und dienen nur zum Speichern von Daten. Sie werden graphisch durch Ellipsen dargestellt. Innerhalb der jeweiligen Ellipse steht der Variablenname (s. Abbildung 7.15) und oberhalb der Ellipse werden die Attribute `messageType`, `type` und `element` notiert. Ist die Angabe eines solchen Attributes für das Verständnis des Prozesses nicht von Relevanz, kann auf sie in der Graphik verzichtet werden (s. Abbildung 7.15). In Abbildung 7.16 ist eine Variable notiert, welche das Attribut `element="elementName"` trägt.

7.3.4 Strukturierte Aktivitäten

Strukturierte Aktivitäten können sehr rasch komplex werden. Es bietet sich somit an, Piktogramme einzuführen, um die Übersichtlichkeit der dargestellten Prozesse zu bewahren. Daher besitzt jede strukturierte Aktivität zusätzlich auch ein Piktogramm.

Sequenz

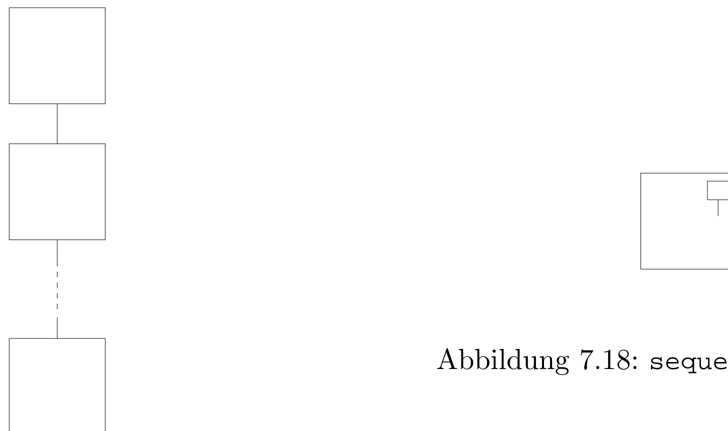


Abbildung 7.18: `sequence` Piktogramm

Abbildung 7.17: `sequence` Schema

In einer Sequenz werden verschiedene Aktivitäten nacheinander ausgeführt. Daher notieren wir diese untereinander. Die einzelnen Aktivitäten sind durch einen einfachen Strich

verbunden. Um eine kognitive Überlastung des Betrachters zu vermeiden, wurde auf eine Pfeildarstellung verzichtet und implizit angenommen, dass der Kontrollfluss stets von oben nach unten gelesen wird. Die Abbildung 7.17 zeigt das Schema einer Sequenz. Des Weiteren kann eine Sequenz auch durch ein Piktogramm (s. Abbildung 7.18) ersetzt werden, um beispielsweise zunächst den groben Ablauf eines Geschäftsprozesses darzustellen.

Nebenläufige Aktivitäten

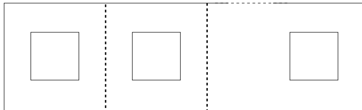


Abbildung 7.19: `flow` Schema

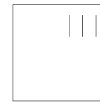


Abbildung 7.20: `flow` Piktogramm

Die nebenläufigen Aktivitäten werden innerhalb einer großen Box durch gestrichelte Linien getrennt gezeichnet. Dabei sollen die gestrichelten Linien andeuten, dass eine parallele Ausführung der jeweils in den abgegrenzten Bereichen liegenden Aktivitäten stattfindet. Die Abbildung 7.19 zeigt schematisch verschiedene parallele Aktivitäten. Das Piktogramm eines `flows` ist in Abbildung 7.20 zu sehen.

While-Schleife

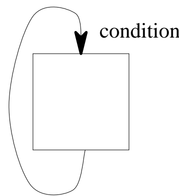


Abbildung 7.21: `while`

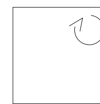


Abbildung 7.22: `while` Piktogramm

Die in der Schleife befindliche Aktivität wird durch einen durchgezogenen Pfeil umschlossen, dessen Pfeilspitze oberhalb der Aktivität mündet. In Höhe der Pfeilspitze wird die Schleifenbedingung geschrieben (s. Abbildung 7.21). Abbildung 7.22 zeigt das Piktogramm für die `while` Aktivität.

Nichtdeterministische Auswahl

Bei dem Element `pick` handelt es sich um eine Aktivität, in der eine Entscheidung getroffen wird. In Abhängigkeit des Auftretens definierter Ereignisse wird eine bestimmte Aktivität ausgeführt. Daher startet das Konstrukt `pick` mit einer Raute. Innerhalb eines `pick` können zwei Arten von Ereignissen definiert werden - ein `onMessage` Ereignis und ein Alarm `onAlarm`. Semantisch verhält sich `onMessage` wie das `receive` Konstrukt und

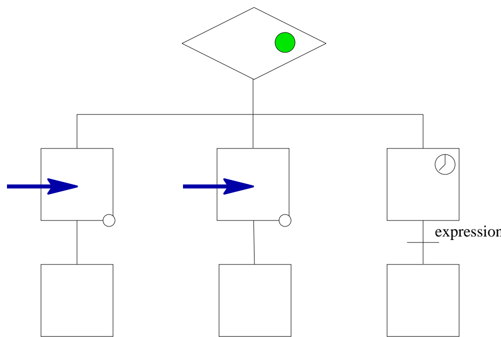


Abbildung 7.23: `pick` Schema

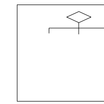


Abbildung 7.24: `pick` Piktogramm

der Alarm wie ein `wait`. Daher werden genau diese beiden Konstrukte auch hier bei einem `pick` genutzt. Ist das Attribut `createInstance="yes"` gesetzt, wird innerhalb der Raute zusätzlich ein grüner Punkt notiert (s. Abbildung 7.23). Das Piktogramm eines `pick` Elementes ist in Abbildung 7.24 zu sehen.

7.4 Zusammenfassung und Ausblick

Wir sehen unsere graphische Repräsentation als Ausgangspunkt für einen Standard der Sprache BPEL4WS. Wie wir eingangs bereits beschrieben haben, ist unsere Graphik konzeptionell durchdacht. Wir haben jedes Element der Sprache BPEL4WS modelliert und sind somit in der Lage den XML-Quelltext eines jeden Geschäftsprozesses, welcher in BPEL4WS formuliert wurde, 1:1 in eine Graphik umzusetzen. Wie viele Fallstudien, so auch das Beispiel aus Abschnitt 7.2.1, belegen, ist unsere Graphik intuitiv erfassbar und bedarf weniger Vorkenntnisse aus dem Bereich der Geschäftsprozessmodellierung.

Es ist möglich, die Konzepte unserer Graphik als Grundlage für einen graphischen BPEL4WS Editor zu nutzen. Somit wird die Arbeit der Entwickler erleichtert, die Geschäftsprozesse in BPEL4WS erstellen. Der Entwickler muss sich nicht mehr mit unübersichtlichen XML-Bäumen beschäftigen. Er hat die Möglichkeit, den Prozess auf einer großen Fläche graphisch übersichtlich zu überschauen. Denn erst in graphischen Repräsentationen werden Strukturmerkmale deutlich, die in einem reinen textbasierten XML-Schema untergehen. Solche Strukturen sind beispielsweise Daten- und Nachrichtenflüsse. Auch komplexe Vorgänge innerhalb von Prozessen, wie nebenläufige Sequenzen, werden schnell sichtbar und kategorisierbar. Das heißt, jegliche Strukturmerkmale eines Geschäftsprozesses lassen sich überblicken und somit leichter verstehen.

Eine einheitliche Graphik ist also von Nöten, um Geschäftsprozesse in BPEL4WS schnell und präzise zu modellieren und um über diese Prozesse dann reden zu können. Wir haben eine solche Graphik für die Sprache BPEL4WS entwickelt. Eine ausführliche Beschreibung der graphischen Repräsentation kann in [Wei03] nachgelesen werden. Dort finden sich auch interessante Beispiele dafür, wie die Graphik aus dem vorliegenden XML-Quelltext des Geschäftsprozesses entwickelt wird.

8 Der Prototyp Wombat4ws

Autor: Axel Martens

Im Rahmen dieser Arbeit ist eine prototypische Implementierung entstanden – das *Workflow Modeling and Business Analysis Toolkit for Web Services* (kurz Wombat4ws). Das Ziel war zum einen die Validierung der entwickelten Konzepte und Methoden. Zum anderen stellt dieser Prototyp den Ausgangspunkt für eine integrierte Entwicklungsumgebung für verteilte Geschäftsprozesse auf Basis von Web Services dar. Aus diesem Grund wird in der weiteren Entwicklung eine Integration von Wombat4ws in die universelle Werkzeug-Plattform eclipse [Gal02] angestrebt.

In diesem Kapitel werden ausgewählte Aspekte der Implementierung vorgestellt. Dazu zählen eine kurze Einführung in das Entwicklungsumfeld und die zugrunde liegende Architektur sowie die Erläuterung der wesentlichen Algorithmen und der Bedienung des Prototyps. Die vollständigen Quellen sowie eine lauffähige Version sind zusammen mit den erwähnten Beispielen frei verfügbar [Mar03b].

8.1 Entwicklungsumfeld

Wombat4ws ist mit der Programmiersprache Java als ein *Anwendungsmodul* des Petrinetz-Kerns (PNK) [KW99] implementiert. Damit war es dem Autor möglich, sich bei der Entwicklung des Prototyps auf die eigentlichen Algorithmen zu konzentrieren ohne die Standardfunktionalität für Petrinetze und eine graphische Bedienoberfläche selbst implementieren zu müssen. Aus diesem Grund werden im folgenden Abschnitt die Struktur und die Konzepte des Petrinetz-Kerns vorgestellt

8.1.1 Der Petrinetz-Kern

Der Petrinetz-Kern ist eine Infrastruktur zum Bau von Petrinetz-Werkzeugen, wurde am Lehrstuhl von Prof. Reisig entwickelt und ist dort frei erhältlich [Mica]. Seine Entwickler verfolgten das Ziel, einem Programmierer die Entwicklung eines petrinetz-basierten Werkzeuges zu erleichtern. Der PNK stellt eine umfangreiche Sammlung von Standardfunktionalität für Petrinetze zur Verfügung: das Parsen und Speichern von Dateien, das Manipulieren und Abfragen von Netzstrukturen und die Visualisierung von Ergebnissen in einem graphischen Editor.

Es gibt bereits eine sehr große Anzahl von petrinetz-basierten Werkzeugen. Für verschiedenste Probleme wurden immer neue Netzklassen entwickelt und auf diesen Netzklassen definierte Algorithmen implementiert. Viele dieser Werkzeuge sind nur für einen Zweck konzipiert und können nicht direkt mit anderen Werkzeugen kombiniert werden – größtenteils aufgrund der heterogenen Datenformate. Der Petrinetz-Kern entstand aus der Beobachtung, dass es ein einheitliches Schema für alle existierenden Petrinetz-Klassen gibt

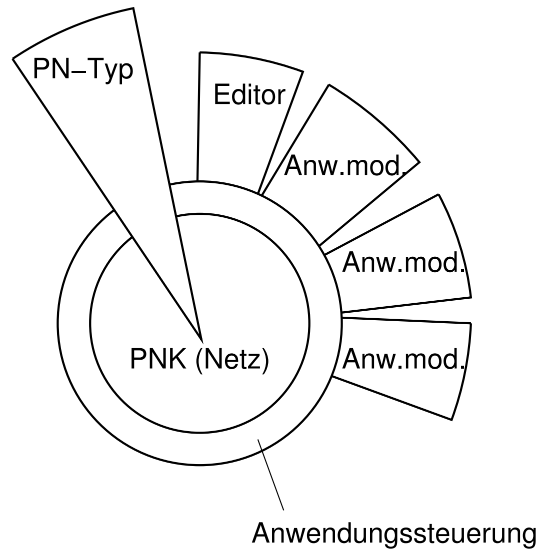


Abbildung 8.1: Architektur des Petrinetz-Kerns

und dass ein System zur Verwaltung deren Modelle die Infrastruktur zur Integration bilden kann. Eine theoretische Fundierung des PNK findet sich in [Web02].

Die Architektur des Petrinetz-Kerns ist bestimmt durch die Forderungen nach Einfachheit, Modularität und Flexibilität. Abbildung 8.1 zeigt eine schematische Darstellung der Architektur, bestehend aus dem eigentlichen Kern, der Anwendungssteuerung und einer Bibliothek von Anwendungsmodulen sowie Petrinetz-Typen.

Im Zentrum steht der eigentliche *Kern*, die Standardfunktionalität für Petrinetze (z. B. Zugriff auf Vor- und Nachbereich der Netzelemente). Dieser Kern wird mit einem *Petrinetz-Typ* – der Beschreibung der verwendeten Petrinetz-Klasse durch ein einheitliches Schema – als Parameter instanziiert. Auf diese Weise entsteht spezifische Funktionalität für diese Netzklasse (z. B. Aktivierung und Schalten von Transitionen).

Ein *Anwendungsmodul* dient z. B. zur Analyse und Manipulation von Netzen oder zur Visualisierung der Ergebnisse der Analyse. Damit beschreibt ein Anwendungsmodul die eigentlichen Algorithmen, die von dem Werkzeug ausgeführt werden sollen. Im Allgemeinen besteht ein PNK-basiertes Werkzeug aus mehreren Anwendungsmodulen, darunter auch solche Anwendungen wie ein Petrinetz-Editor.

Die *Anwendungssteuerung* des PNK verwaltet die Anwendungsmodule, Petrinetz-Typen und Netze. Sie koordiniert damit das entstandene Werkzeug. Der in dieser Arbeit entstandene Prototyp ist ein PNK-basiertes Werkzeug und hat neben dem Editor ein weiteres Anwendungsmodul, auf das wir im weiteren Text noch detaillierter eingehen.

8.1.2 Die Integration

Die Entwicklung eines neuen Anwendungsmoduls mit dem Petrinetz-Kern gestaltet sich denkbar einfach: Der Programmierer muss lediglich ein vorgegebenes *Java-Interface* implementieren und hat damit sofort Zugriff auf die Standardfunktionalität und die Menüstruktur

der Bedienoberfläche. Darüber hinaus ist eine einfache Interaktion mit dem Editor möglich, falls diese Komponente im Werkzeug benötigt wird. Weitere Einzelheiten zur Programmierung mit dem Petrinetz-Kern finden sich in [Micb].

Zur Repräsentation der Daten benutzt der Petrinetz-Kern ein auf XML basierendes Format – die *Petri Net Markup Language* (PNML) [WK03]. Diese Sprache ist aus dem zuvor erwähnten einheitlichen Schema für alle Petrinetz-Klassen abgeleitet und als Vorschlag zur Standardisierung von Petrinetzen eingereicht. Der Entwickler eines neuen Anwendungsmoduls kann durch die modulare Struktur des PNK seine eigenen Datenformate verwenden und erhält mit dem integrierten XML-Parser dabei eine gute Unterstützung. In aktuellen Forschungen wird eine Petrinetz-Semantik für BPEL4WS entwickelt [RW03]. Aufbauend auf diesen Ergebnissen soll der vorgestellte Prototyp angepasst werden, so dass eine direkte Analyse von Modellen in der Syntax von BPEL4WS möglich wird.

Mit dem Petrinetz-Kern können bestehende Werkzeuge in *ein* Werkzeug integriert werden. Dabei werden verschiedene Wege beschritten: die direkte Anbindung über eine Programmier-Schnittstelle des bestehenden Werkzeuges oder den Datenaustausch über Dateien und die Anbindung über Systemrufe. In Wombat4ws wurden das Analysewerkzeug Woflan [VBA00] und das Werkzeug dot [GKN02] zur automatischen Anordnung von Graphen über die zweite Variante integriert.

Der Petrinetz-Kern kann durch seinen modularen Aufbau selbst in ein anderes Werkzeug integriert werden. In weiteren Arbeiten soll die Anbindung an die universelle Werkzeug-Plattform eclipse [Gal02] realisiert werden. Damit wird ein einfacher Austausch von Informationen zwischen dem Wombat4ws und anderen Werkzeugen zur Modellierung und Ausführung von Web Services angestrebt.

8.2 Architektur

Das Wombat4ws besteht wie bereits geschildert aus den Klassen des Petrinetz-Kerns und den neu implementierten Klassen, die ein *Anwendungsmodul* des Petrinetz-Kerns formen. In diesem Abschnitt geben wir einen Überblick über die Architektur dieses Anwendungsmoduls, ohne auf alle Details einzugehen oder gar ein vollständiges Klassendiagramm zu präsentieren. Im darauf folgenden Abschnitt erläutern wir die wesentlichen Algorithmen und die Bedienung des Wombat4ws.

Insgesamt besteht das Anwendungsmodul aus 15 Klassen und umfasst ca. 3000 Code-Zeilen. Die Klassen lassen sich in vier Gruppen einteilen. Die erste Gruppe ist für den Petrinetz-Typ, d. h. die speziellen Eigenschaften der Workflow-Module zuständig. Die zweite Gruppe umfasst den Aufbau und die Analyse von Kommunikationsgraphen. Die Klassen der dritten Gruppe implementieren weitere Algorithmen auf Petrinetzen. Schließlich dienen die Klassen der vierten Gruppe zur Implementierung von Querschnittsaufgaben.

8.2.1 Petrinetz-Typ

Das Modell eines verteilten Geschäftsprozesses entsteht in dieser Arbeit durch die Komposition der Modelle seiner lokalen Komponenten, d. h. durch Komposition von Workflow-Modulen. Für die Analyse ist es notwendig, die speziellen Strukturen und Eigenschaften der

```
<?xml version="1.0" encoding={\"U}TF-8"?> <!DOCTYPE
netTypeSpecification SYSTEM"netTypeSpecification.dtd">
<netTypeSpecification name="WFModule">
  <extendable class="de.huberlin.informatik.pnk.kernel.Net">
     <extension name="firingRule" class="wfmt.FiringRule"/>
  </extendable>
  <extendable class="de.huberlin.informatik.pnk.kernel.Place">
     <extension name="marking" class="wfmt.PTMarking"/>
     <extension name="initialMarking" class="wfmt.PTMarking"/>
     <extension name="nodeType" class="wfmt.NodeType"/>
     <extension name="nodeRef" class="wfmt.NodeRef"/>
     <extension name="runRef" class="wfmt.RunRef"/>
  </extendable>
  <extendable class="de.huberlin.informatik.pnk.kernel.Transition">
     <extension name="nodeType" class="wfmt.NodeType"/>
     <extension name="nodeRef" class="wfmt.NodeRef"/>
     <extension name="runRef" class="wfmt.RunRef"/>
  </extendable>
  <extendable class="de.huberlin.informatik.pnk.kernel.Arc">
     <extension name="inscription" class="wfmt.PTMarking"/>
  </extendable>
</netTypeSpecification>
```

Abbildung 8.2: Spezifikation des Petrinetz-Typs WFModule

Workflow-Module zu berücksichtigen. Deshalb wurde ein eigener Petrinetz-Typ definiert – der Typ WFModule. Abbildung 8.2 zeigt dessen Spezifikation:

Ein Petrinetz-Typ ist definiert über eine Liste von Erweiterungen, die jedes Element eines Netzes von diesem Typ besitzen darf. Die Abbildung 8.2 zeigt den Quelltext der Datei WFModule.xml, die den Petrinetz-Typ der Workflow-Module spezifiziert. Jede Erweiterung (extension) ist einem Netzelement zugeordnet und mit der Java-Klasse verknüpft, welche die Erweiterung implementiert. Alle Klassen stammen aus dem Package wfmt und wurden vom Autor entwickelt. Daher lassen wir im Folgenden den Bezeichner wfmt weg.

Die Java-Klasse FiringRule implementiert die Schaltregel des Petrinetz-Typs WFModule und stellt Operationen zur Verfügung, um die Aktivierung einer Transition zu ermitteln und die Änderung der Markierung durch das Schalten einer Transition zu errechnen. Für die Analyse von Workflow-Modulen ist sowohl das interne als auch das externe Verhalten wichtig. Daher verfügt die Java-Klasse FiringRule über Methoden zum Schalten einer Transition mit und ohne Berücksichtigung der Schnittstellen des Moduls.

Ein Workflow-Modul ist in dieser Arbeit ein Low-Level-Petrinetz und damit kann die Markierung auf jeder Stelle und die Anschrift an jeder Kante durch eine natürliche Zahl repräsentiert werden. Die Java-Klasse PTMarking ermöglicht einen einfachen Zugriff auf diese Datenstruktur und ist daher als Implementierung der Erweiterungen marking und initialMarking – bezogen auf eine Stelle – und der Erweiterung inscription – bezogen auf eine Kante – angegeben.

In den Algorithmen zur Analyse von Workflow-Modulen finden verschiedene Objekte Verwendung: Abläufe, Erreichbarkeits- und Kommunikationsgraphen sowie vereinfachte und umgebende Workflow-Module. Aus Gründen der Wiederverwendbarkeit von vorhandener Funktionalität sind all diese Objekte durch eine einheitliche Datenstruktur abgebildet – als

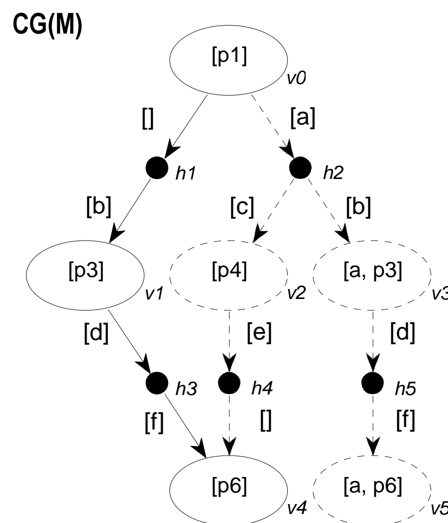


Abbildung 8.3: Ein gefärbter Kommunikationsgraph

Workflow-Module. Daraus wird ersichtlich, dass jedes Element weitere Attribute benötigt, um den Knotentyp zu repräsentieren (z. B. Input-Stelle, Sende-Ereignis, usw.), um den Bezug zu anderen Knoten aufrecht zu erhalten (z. B. Ereignis e_7 repräsentiert das Schalten der Transition t_4) und letztlich um die Bezüge zwischen Bediensequenzen und Abläufen auszudrücken (z. B. Knoten v_3 überdeckt die Abläufe ρ_1 bis ρ_3). Für diese Zusammenhänge werden die Klassen NodeType, NodeRef und RunRef benötigt.

Mit einem Petrinetz-Typ ist ein Dateiformat verknüpft, diese Verknüpfung wird in der Datei toolSpecification.xml festgelegt. In einer weiteren Ausbaustufe sollten die Algorithmen direkt auf der Syntax einer „echten" Modellierungssprache arbeiten – z. B. auf BPEL4WS. Im derzeitigen, prototypischen Stadium ist es dagegen wichtig, schnell und einfach interessante Beispiele zu konstruieren. Es erschien daher sinnvoll, auf der Sprache des Petrinetz-Kerns (PNML) aufzusetzen, sie jedoch zu vereinfachen, so dass eine Manipulationen auf Textebene möglich wird. Das leicht abgewandelte Dateiformat WFModule ist durch die Java-Klasse WFModuleInOut definiert. Daneben findet sich in dieser Klasse eine Methode zum Export von Netzen in das Format des Werkzeuges Woflan.

8.2.2 Kommunikationsgraphen

Das zentrale Objekt in dieser Arbeit ist der Kommunikationsgraph. Drei Java-Klassen sind mit seiner Konstruktion und Analyse befasst: ComGraph, ComNode und State. Im Abschnitt 8.3 gehen wir auf die Implementierung der Methoden dieser Klassen genauer ein. An dieser Stelle präsentieren wir nur die Aufteilung der Aufgaben. Abbildung 8.3 zeigt ein kleines Beispiel für einen Kommunikationsgraphen.

Ein Kommunikationsgraph ist ein *bipartiter Wurzelgraph*. Der Prototyp repräsentiert diese Struktur intern durch ein Petrinetz, wobei alle Knoten des Graphen durch Stellen abgebildet werden und die Kanten des Graphen zu Transitionen in diesem Netz werden.

Die Kanten des Petrinetzes ergeben sich dann kanonisch. Ist ein Knoten *unsichtbar*, dann bekommt die entsprechende Stelle den Typ intern und wird in der graphischen Darstellung als kleiner ausgefüllter Kreis abgebildet (z. B. h1). Jeder Knoten hat eine Farbe: schwarz für noch nicht analysiert, blau für einen Knoten, der zum *Bediengraphen* gehört und rot ansonsten. In dieser Arbeit finden sich ausschließlich Abbildungen in Graustufen. Daher repräsentieren wir die Farbe blau durch eine durchgezogene Linie und die Farbe rot durch eine gestrichelte Linie. Jeder noch nicht analysierte (schwarze) Knoten wird grau dargestellt.

Die Java-Klasse ComGraph verwaltet dieses Netz – genannt cgNet. Daneben verfügt diese Klasse über eine Hash-Tabelle der *sichtbaren Knoten* und implementiert Methoden zum Analysieren des Kommunikationsgraphen. Der eigentliche Aufbau dieses Graphen vollzieht sich in der Java-Klasse ComNode. Die Klasse ComGraph regelt jedoch die Reihenfolge der Knoten, an denen der Graph weiter aufgebaut wird. Dadurch sind verschiedene Suchstrategien möglich.

Ein Objekt der Java-Klasse ComNode verweist auf eine Stelle im cgNet und repräsentiert einen sichtbaren Knoten (z. B. v1). Die Aufgabe dieser Klasse ist es, den Kommunikationsgraphen bis zur nächsten Ebene von sichtbaren Knoten weiter aufzubauen. Wie in Abbildung 8.3 dargestellt, verweist jeder sichtbare Knoten auf eine nicht-leere Menge von Zuständen des Workflow-Moduls (v1 verweist z. B. auf {[p3], [p2,a]}).

Ein Zustand wird durch ein Objekt der Java-Klasse State repräsentiert. Diese Klasse verfügt über Methoden, einen Zustand zu speichern, zu setzen oder zu manipulieren. Für den Aufbau des Kommunikationsgraphen sind die wesentlichen Methoden die Implementierungen der Funktionen INP und NXT. Diese Methoden liefern die in einem Zustand *aktivierten Eingaben* und die Menge der *Folgezustände* eine Zustands.

Ein Objekt der Java-Klasse ComNode benutzt diese Methoden, um die Multimengen von Nachrichten zu ermitteln, die an den ausgehenden Kanten annotiert werden und jeweils die Multimenge von Zuständen zu errechnen, auf die ein nachfolgender Knoten verweist.

8.2.3 Petrinetz-Algorithmen

Zur Analyse des Verhaltens eines Workflow-Moduls wurden neben dem Aufbau des Kommunikationsgraphen weitere Methoden implementiert, die jeweils eine in der Theorie der Petrinetze akzeptierte Darstellung des Verhaltens erzeugen. Dazu zählen der *Erreichbarkeitsgraph* [Sta90], die Menge der *verteilten Abläufe* [Rei85] und der *verzweigende Prozess* (engl. *branching process*, [Ste00]). Diese Darstellungen werden von Methoden der Java-Klasse Generator erzeugt.

Aus dem Kommunikationsgraphen eines Workflow-Moduls lässt sich einerseits eine bedienende Umgebung konstruieren und andererseits eine abstrakte Sicht auf das Workflow-Modul ableiten. Beide Verfahren transformieren den interessanten Teil des Kommunikationsgraphen in ein Workflow-Modul und sind deshalb als Methoden der Java-Klasse Transformator implementiert.

8.2.4 Querschnittsaufgaben

Neben den eigentlichen Algorithmen umfasst der entstandene Prototyp vier weitere Java-Klassen. Diese dienen der Anbindung an den Petrinetz-Kern, der Visualisierung der Ergeb-

nisse und der Bereitstellung von Datenstrukturen und universell einsetzbaren Methoden. Aus diesem Grund subsumieren sich diese Klassen unter dem Stichpunkt *Querschnittsaufgaben.*

Die Java-Klasse AppMain ist von der Klasse MetaBigApplication des Petrinetz-Kerns abgeleitet und stellt somit die Einbindung der implementierten Algorithmen (als Anwendungsmodul) in das gesamte Werkzeug sicher. In der Klasse AppMain werden die Netze und Graphen verwaltet sowie die Menü-Struktur definiert und mit den Aufrufen der Methoden der zuvor beschriebenen Java-Klassen verbunden.

Zur graphischen und textuellen Darstellung von Ergebnissen der Analyse findet die Java-Klasse WorkSpace Verwendung. Dazu implementiert diese Klasse eine Anbindung an das Werkzeug DOT [GKN02] – ein Open-Source-Graphikwerkzeug, mit dem es möglich ist, aus einer textuellen Darstellung eines Graphen automatisch eine graphische Repräsentation zu erstellen. Dabei unterstützt das Werkzeug DOT verschiedene Formate, u. a. GIF, JPG und Postscript. Die Java-Klasse WorkSpace bietet eine Methode an, ein Petrinetz oder einen Kommunikationsgraphen in das Eingabeformat von DOT zu transformieren, das Werkzeug zu starten und das Ergebnis, d. h. die graphische Repräsentation sofort anzuzeigen. Abbildung 8.4 zeigt einen Screenshot des WOMBAT4WS, darin zu erkennen ist der generierte Kommunikationsgraph.

In der verwendeten Version des Petrinetz-Kerns ist es nicht möglich, die Java-Klassen für das Petrinetz und die einzelnen Netzelemente weiter zu verfeinern, d. h. eine eigene Klasse durch Vererbung abzuleiten. Aus diesem Grund wurde die Java-Klasse NetUtil implementiert, die statische Methoden für häufig verwendete Aufgaben zur Verfügung stellt. Dazu gehören die Verknüpfung von Operationen auf Netzelementen (z. B. der Vorbereich einer Transition) und das Kopieren bzw. Transformieren ganzer Netze.

Die letzte Java-Klasse dieser Gruppe definiert, wie der Name Multiset andeutet, eine Datenstruktur und ist eine Verfeinerung der Klasse java.util.Vector. Sie implementiert typische Operationen auf Multimengen, wie Addition, Durchschnitt und Transformation in eine Menge, und findet daher in den anderen Klassen reichlich Verwendung.

Neben den ausführbaren Dateien des Prototypen WOMBAT4WS sind die Quelltexte, die in dieser Arbeit vorgestellten Beispiele und die Dokumentation des Petrinetz-Kerns verfügbar [Mar03b]. Alle Dateien unterliegen der *GNU General Public License* [Fre91]. Im Zuge der weiteren Entwicklung ist auch die Dokumentation der neu entwickelten Klassen vorgesehen.

8.3 Implementierung

Zu Beginn dieses Abschnittes beschreiben wir kurz die Bedienung des WOMBAT4WS, wohl wissend, dass sich nicht alle Funktionalität des Werkzeuges intuitiv erschließt. Die Einbeziehung des Standard-Editors des Petrinetz-Kerns stellt einen Kompromiss zwischen Aufwand und Ergebnis dar. Im Zuge der Integration in die Werkzeug-Plattform ECLIPSE wird auch das Werkzeug zum Nachweis der Bedienbarkeit selbst bedienbar. Der zweite Teil dieses Abschnittes geht detailliert auf den Aufbau und die Analyse des Kommunikationsgraphen ein.

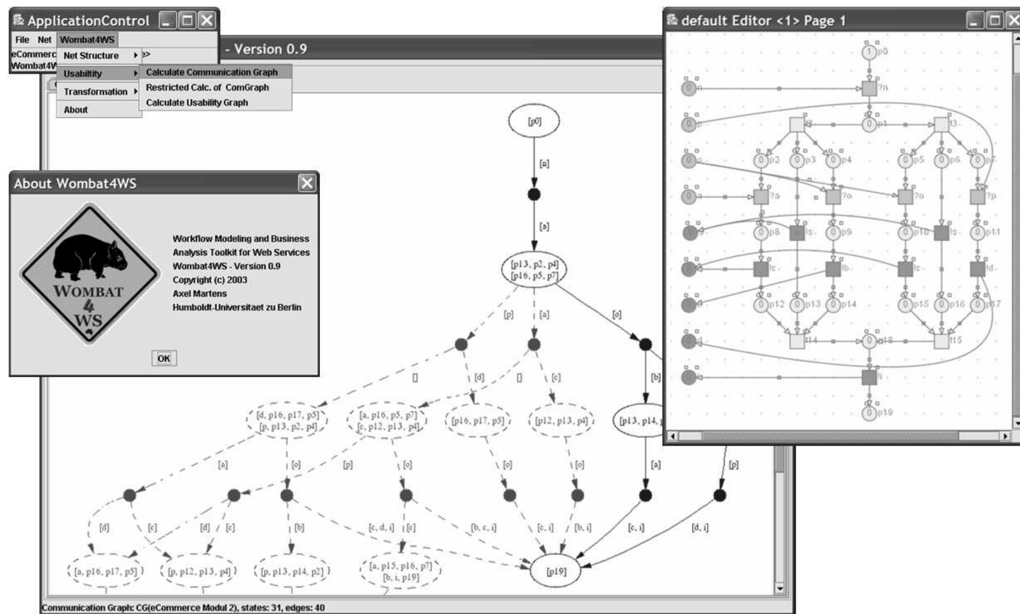


Abbildung 8.4: Screenshot des WOMBAT4WS

8.3.1 Bedienung

Um ein PNK-basiertes Werkzeug zu starten, benötigt man eine Datei, die den Aufbau des Werkzeuges beschreibt – die Tool-Spezifikation (in Fall des WOMBAT4WS die Datei toolSpecification.xml). Mit dieser Datei als Parameter startet man die Klasse ApplicationControl des Petrinetz-Kerns unter Angabe der notwendigen Ressourcen. Zum Umfang des Prototypen gehört eine Datei start.bat, die diesen Aufruf für eine Microsoft-Umgebung ausführt. Befindet sich die Laufzeitumgebung von Java im Suchpfad, so startet das Fenster der ApplicationControl.

Abbildung 8.4 zeigt einen Screenshot des WOMBAT4WS bereits mit einem geladenen Workflow-Modul und einem generierten und analysierten Kommunikationsgraphen. Das Fenster der ApplicationControl ist links oben abgebildet. Dieses Fenster besitzt ein Menü, dass sich jedoch ändern kann, je nachdem, welches Anwendungsmodul aktiv ist. Zu Beginn ist kein Anwendungsmodul aktiv, erst wenn ein Netz geladen oder ein neues erzeugt wird, startet der Editor[1] in einem neuen Fenster und die Menü-Struktur ändert sich. Unter der Menüleiste ist der Name des aktuellen Netzes und des aktiven Anwendungsmoduls aufgeführt.

Im Zentrum steht, wie aus Abbildung 8.1 ersichtlich, das aktuelle Petrinetz, für das nun weitere Anwendungsmodule gestartet werden können. Dazu wählt man den Menüpunkt <Net>/<Start Application>. Falls mehrere Anwendungsmodule geladen wurden, kann man mit dem Menüpunkt <Net>/<Select Application> zwischen diesen wechseln. Gleiches gilt für mehrere Petrinetze.

Alle Beispiele dieser Arbeit sind zusammen mit dem Prototyp verfügbar. In Abbildung 8.4

[1]Der Editor wurde in der Datei toolSpecification.xml als Default-Anwendungsmodul deklariert.

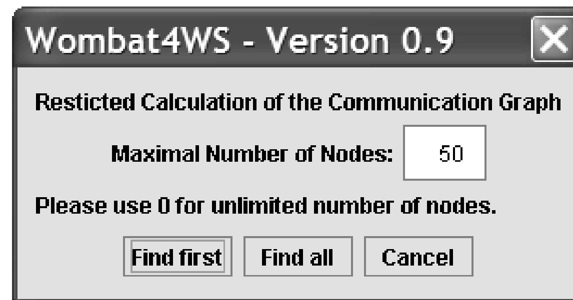


Abbildung 8.5: Beschränkter Aufbau des Kommunikationsgraphen

wurde das Netz eCommerce2.xml geladen. Der Editor des Petrinetz-Kerns bietet die Möglichkeit, das Netz zu manipulieren (d. h. Hinzufügen, Löschen und Ändern von Elementen) und Informationen aus- und einzublenden bzw. auf mehrere Seiten zu verteilen. Weitere Informationen zum Umgang mit dem Editor finden sich in [Micb].

Mit dem Starten des Anwendungsmoduls Wombat4ws erscheint ein neues Fenster – der Arbeitsbereich der Analyse (WorkSpace). In Abbildung 8.4 ist bereits der analysierte Kommunikationsgraph in diesem Fenster dargestellt. Neben dem Erscheinen des neuen Fensters ändert sich auch die Darstellung des Workflow-Moduls im Editor, wie in der Abbildung 8.4 zu sehen ist. Um die Strukturen deutlicher hervorzuheben, werden die Schnittstellen und die damit verknüpften Transitionen eingefärbt, d. h. Input-Stellen werden orange und Output-Stellen werden grün dargestellt. Alle überflüssigen Informationen werden ausgeblendet.

Das Starten des Anwendungsmoduls Wombat führt außerdem zu einer Änderung der Menü-Struktur. Unter dem Eintrag Wombat4ws finden sich drei Gruppen von Einträgen: <Net Structure>, <Usability> und <Transformation>. Hinter dem Eintrag <Net Structure> verbergen sich die Aufrufe der drei im Abschnitt 8.2.3 vorgestellten Methoden. Desweiteren ist eine detaillierte textuelle Ausgabe des Netzes möglich. Die Ausgaben aller Methoden werden im Fenster WorkSpace angezeigt.

Usability

In Abbildung 8.4 ist der Menüpunkt <Usability> ausgewählt. Die drei Einträge starten die Berechnung des Kommunikationsgraphen. Der erste Eintrag startet die Berechnung ohne zusätzliche Randbedingungen. Der Algorithmus terminiert erst dann, wenn der Kommunikationsgraph vollständig aufgebaut und analysiert wurde. Das Ergebnis für diesen Aufruf ist in Abbildung 8.4 dargestellt.

Es ist jedoch möglich, dass der Kommunikationsgraph eines Workflow-Moduls unendlich ist (siehe Abschnitt B.2.2). In diesem Fall würde das Programm beim Aufruf der ersten Variante nicht zu einem Ende kommen bzw. mit einem Überlauf des Speichers die Berechnung abbrechen. Für diesen Zweck ist der zweite Eintrag gedacht. In diesem Fall wird dem Benutzer ein Dialog angezeigt. Dieser Dialog ist in Abbildung 8.5 dargestellt.

Es gibt zwei Möglichkeiten, den Aufbau des Kommunikationsgraphen zu beschränken. Auf der einen Seite kann eine maximale Anzahl von Knoten vorgegeben werden. Wird diese Grenze erreicht endet die Konstruktion und jedes Blatt das Graphen wird als Endzustand

interpretiert. Der Vorteil dieser Methode liegt in der sicheren Terminierung des Algorithmus. Der Nachteil liegt in der mangelnden Intelligenz, d. h. wäre nur ein Knoten mehr nötig gewesen, um die Bedienbarkeit nachzuweisen, dann schlägt der Nachweis fehl. Die vorgegebene Anzahl liegt, wie in Abbildung 8.5 dargestellt, bei 50 Knoten, kann aber beliebig geändert werden – die Eingabe 0 steht dabei für unbeschränkt.

Die zweite Möglichkeit ist, den Aufbau des Kommunikationsgraphen auf den ersten Bediengraphen zu beschränken. Ein Kommunikationsgraph kann beliebig viele Bediengraphen als Teilgraphen enthalten. Bei dieser Variante wird während der Konstruktion jeder maximale Knoten daraufhin untersucht, ob der Pfad von der Wurzel bis zu diesem Knoten ein Bediengraph ist. Die Schaltfläche <Find first> (siehe Abbildung 8.5) startet diese Variante.

Der Vorteil dieser Methode liegt auf der Hand: Jedes bedienbare Workflow-Modul hat mindestens einen endlichen Bediengraphen. Mit einer Breite-Zuerst-Suche gelingt es nach endlicher Zeit, diesen Graphen zu finden. Damit terminiert dieser Algorithmus für jedes bedienbare Modul, falls der endliche Bediengraph in den Speicher passt. Ist ein Modul nicht bedienbar und hat einen unendlichen Kommunikationsgraphen, dann terminiert dieser Algorithmus jedoch nicht. Es bietet sich daher an, beide Beschränkungen zu kombinieren.

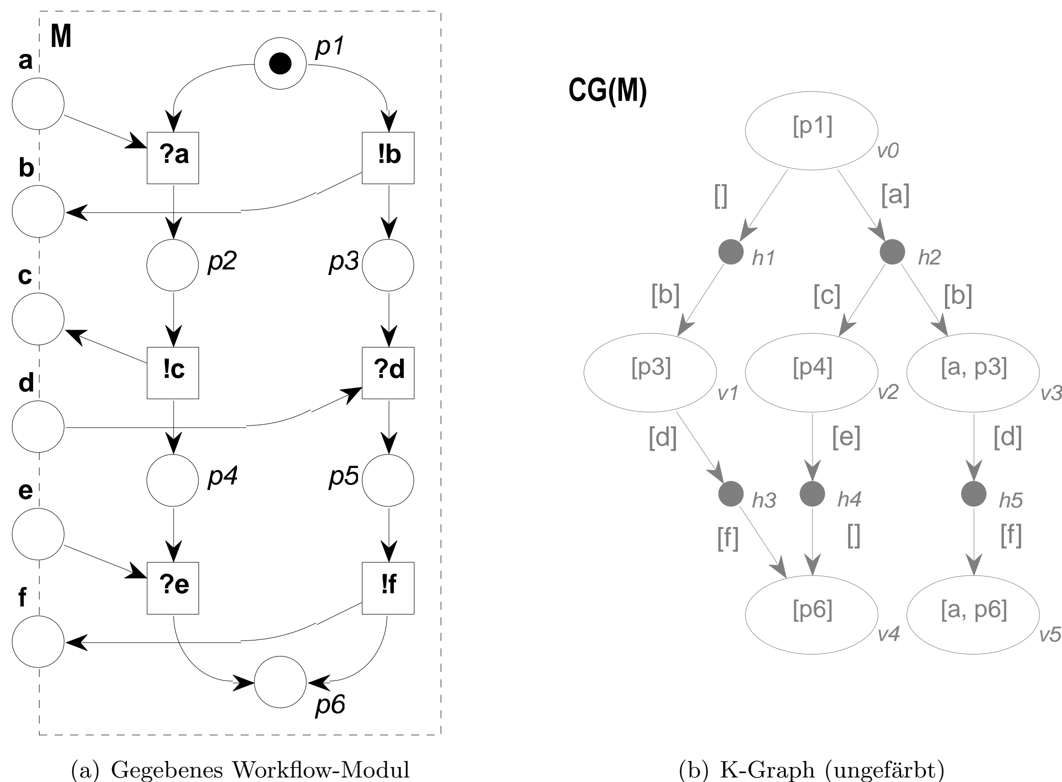
Der dritte Eintrag im Menü veranlasst die gesonderte Darstellung des Bediengraphen, also des blau gefärbten Anteils des Kommunikationsgraphen. Wenn der Kommunikationsgraph schon berechnet wurde, dann wird nur die Darstellung überarbeitet, ansonsten erscheint ebenfalls der Dialog aus Abbildung 8.5, um die Berechnung des Kommunikationsgraphen anzustoßen. Eine unbeschränkte Berechnung erreicht man durch die Eingabe der Anzahl 0 und das Drücken der Schaltfläche <Find all>.

Transformation

Neben den Einträgen <Net Structure> und <Usability> gibt es im Menü noch einen Bereich <Transformation>. Dort verbergen sich die Aufrufe zur Konstruktion einer Umgebung und einer abstrakten Sicht (*Public View*). Die Abwicklung eines wohlgeformt-zyklischen Workflow-Moduls ist z. Zt. noch nicht implementiert. Für die beiden anderen Transformationen wird der Kommunikationsgraph benötigt. Ist dieser nicht vorhanden, erscheint auch hier der Dialog aus Abbildung 8.5.

Die abstrakte Sicht wird nach ihrer Berechnung im Fenster WorkSpace angezeigt. Wird eine Umgebung errechnet, so erscheint im Fenster WorkSpace das komponierte Modul aus dem ursprünglichen Workflow-Modul und der konstruierten Umgebung. Die Schnittstellen zwischen diesen beiden sind jedoch aus folgendem Grund nicht dargestellt: Durch die fehlenden Verbindungen erzielt der Algorithmus zur graphischen Aufbereitung der Netze ein besseres Ergebnis und die Label der Transitionen tragen alle nötigen Informationen in sich.

Für einige wenige (pathologische) Fälle ist die konstruierte Umgebung keine bedienende Umgebung, sondern das Modul ist nicht bedienbar und das komponierte System gerät in einen *Livelock*. Um diese Eigenschaft auszuschließen, startet nach der Konstruktion der Umgebung und damit des komponierten Systems das Analyse-Werkzeug Woflan. Durch einen Druck auf den roten Doppelpfeil startet dann die Analyse der *Soundness* des komponierten Systems. Wenn sie erfolgreich ist, dann ist das Modul bedienbar.



(a) Gegebenes Workflow-Modul

(b) K-Graph (ungefärbt)

Abbildung 8.6: Aufbau des Kommunikationsgraphen

8.3.2 Aufbau von Kommunikationsgraphen

In diesem Abschnitt beschreiben wir die Konstruktion des Kommunikationsgraphen für ein gegebenes Workflow-Modul. Die Implementierung basiert auf dem im [Mar04] vorgestellten Algorithmus und die nachfolgende Beschreibung verwenden deshalb die dort eingeführten Begriffe.

Wir verdeutlichen die Konstruktion des Kommunikationsgraphen am Beispiel des Workflow-Moduls aus Abbildung 8.6(a). Abbildung 8.6(b) zeigt das Ergebnis der Konstruktion noch ohne Analyse, d. h. ohne Färbung des Graphen. Mit dieser Aufgabe befasst sich der nachfolgende Abschnitt 8.3.3.

Für die Konstruktion des Kommunikationsgraphen sind die Java-Klassen ComGraph, ComNode und State zuständig. Die Klasse ComGraph verwaltet eine Liste von Knoten, d. h. Objekte der Klasse ComNode. Jeweils mit dem ersten Element dieser Liste wird die Konstruktion fortgesetzt. Neu erzeugte Knoten werden hinten an diese Liste angehängt. Auf diese Weise implementiert die Klasse ComGraph eine *Breite-Zuerst-Suche*.

Zu Beginn befindet sich in dieser Liste nur der Wurzelknoten (v0). Solange die Liste noch einen Knoten enthält, wird für diesen Knoten die Methode buildGraph() aufgerufen. Abbildung 8.7 zeigt den vereinfachten Quelltext dieser Methode.

Die Methode buildGraph() wird für einen *sichtbaren Knoten* aufgerufen. Ausgehend von diesem Knoten wird die Menge aller *aktivierten Eingaben* mit der Methode getInputs() be-

```
public void buildGraph() {
    inputSet = getInputs();
    while (inputSet.hasMoreElements()) {
        input = inputSet.nextElement();
        child = this.addChild(input);
        statesSet = getStates(input);
        outputSet = getOutputs(statesSet);
        while (outputSet.hasMoreElements()) {
            output = outputSet.nextElement();
            states = selectStates(statesSet, output);

            subChild = comGraph.getNodebyName(states);
            if (subChild != null) {
                child.addChild(subChild, output);
                toAnalyse.add(subChild);
            }
            else {
                subChild = child.addChild(output);
                if (subChild.isDeadEnd()) { toAnalyse.add(subChild); }
                else { comGraph.addNodeToTargets(subChild); }
}   }   }   }
```

Abbildung 8.7: Quelltext der Methode buildGraph()

rechnet – diese Methode ruft für jeden Zustand des aktuellen Knotens die Methode getInputs() der Java-Klasse State auf und fasst die Ergebnisse zusammen. Auf die Klasse State gehen wir im folgenden Text noch genauer ein.

Wie aus Abbildung 8.7 ersichtlich, dienen zwei ineinander geschachtelte while-Schleifen zum Aufbau des Graphen: Für jede aktivierte Eingabe input wird ein *unsichtbarer Knoten* child an den aktuellen Knoten angehängt. Anschließend dient die Methode getStates(input) zur Berechnung der *Folgezustände* bzgl. dieser Eingabe. Auch diese Methode hat eine Entsprechung in der Klasse State.

Aus der Menge der Folgezustände lässt sich die Menge der *möglichen Ausgaben* einfach bestimmen (mit der Methode getOutputs(statesSet)). Für jede dieser Ausgaben output muss eine Kante vom Knoten child wieder zu einem sichtbaren Knoten führen. Dieser sichtbare Knoten umfasst alle Zustände, die mit der vorgegebenen Eingabe und Ausgabe erreichbar sind, zur Auswahl dient die Methode selectStates(statesSet, output). Wir unterscheiden zwei Fälle:

Fall 1: Es gibt bereits einen Knoten im bisher konstruierten Kommunikationsgraphen, der diese Zustände umfasst. Dann wird dieser Knoten (subchild) an den Knoten child angehängt und anschließend die Analyse gestartet. Der nächste Abschnitt betrachtet die Analyse genauer.

Fall 2: Falls es noch keinen derartigen Knoten gibt, wird ein neuer sichtbarer Knoten (subchild) erzeugt. In den meisten Fällen wird der Knoten subchild hinten an die Liste der Klasse ComGraph angehängt. Zur Optimierung der Konstruktion testet vorab die Methode isDeadEnd(), ob es sich bei dem Knoten subchild um ein totes Ende handelt, d. h. einen Knoten, der nur den Endzustand des Workflow-Moduls (v4) oder einen

```
ComNode({[p1]}).buildGraph()
    ComNode({[p1]}).getInputs() = {[], [a]}
    ComNode({[p1]}).getStates([]) = {[b, p3]}
    ComNode({[p1]}).getOutputs({[b, p3]}) = {[b]}
    ComNode({[p1]}).selectStates({[b, p3]}, [b]) = {[p3]}
    ComNode({[p1]}).getStates([a]) = {[a, b, p3], [c, p4]}
    ComNode({[p1]}).getOutputs({[a, b, p3], [c, p4]}) = {[b], [c]}
    ComNode({[p1]}).selectStates({[a, b, p3], [c, p4]}, [b]) = {[a, p3]}
    ComNode({[p1]}).selectStates({[a, b, p3], [c, p4]}, [c]) = {[p4]}
ComNode({[p3]}).buildGraph()
    ComNode({[p3]}).getInputs() = {[d]}
    ComNode({[p3]}).getStates([d]) = {[f, p6]}
    ComNode({[p3]}).getOutputs({[f, p6]}) = {[f]}
    ComNode({[p3]}).selectStates({[f, p6]}, [f]) = {[p6]}
ComNode({[a, p3]}).buildGraph()
    ComNode({[a, p3]}).getInputs() = {[d]}
    ComNode({[a, p3]}).getStates([d]) = {[a, f, p6]}
    ComNode({[a, p3]}).getOutputs({[a, f, p6]}) = {[f]}
    ComNode({[a, p3]}).selectStates({[a, f, p6]}, [f]) = {[a, p6]}
ComNode({[p4]}).buildGraph()
    ComNode({[p4]}).getInputs() = {[e]}
    ComNode({[p4]}).getStates([e]) = {[p6]}
    ComNode({[p4]}).getOutputs({[p6]}) = {[]}
    ComNode({[p4]}).selectStates({[p6]}, []) = {[p6]}
```

Abbildung 8.8: Ablauf-Protokoll der Konstruktion

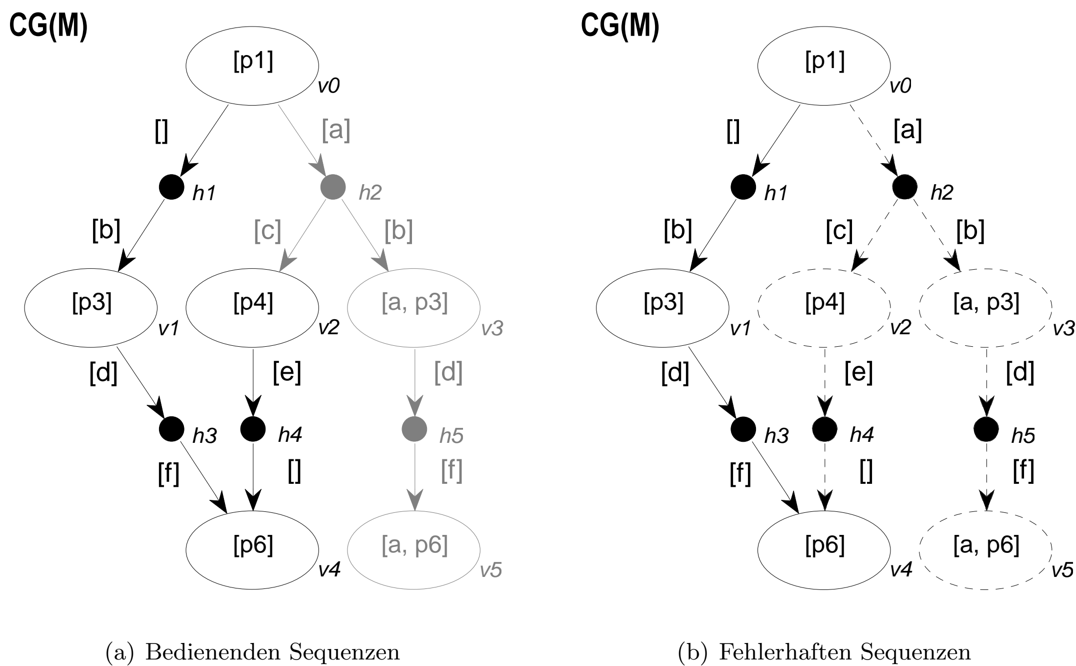
eindeutigen Fehlerzustand umfasst (v5). Falls diese Methode den Wert TRUE liefert, wird stattdessen wie im *Fall 1* die Analyse für den Knoten subchild gestartet.

Abbildung 8.8 zeigt das Ablauf-Protokoll der Konstruktion für das Beispiel aus Abbildung 8.6, wobei statt der Bezeichnung eines Knotens (z. B. v0) die Menge der Zustände dieses Knotens angegeben ist (z. B. {[p1]}).

Wie gesehen, koordiniert die Java-Klasse ComNode die einzelnen Schritte der Konstruktion des Kommunikationsgraphen. Die eigentliche Implementierung dieser Schritte befindet sich in der Klasse State, d. h. die Methoden getInputs() und getStates() implementieren die in [Mar04] vorgestellten Funktionen INP und NXT. Während die Methode getInputs() durch *Tiefe-Zuerst-Suche* genau die Menge der in einem Zustand aktivierten Eingaben ermittelt, muss die Methode getStates() das Problem lösen, einen maximalen Zustand bzgl. eines Ablaufs zu ermitteln, ohne die Abläufe des Moduls explizit zu kennen.

Die Implementierung löst dieses Problem mit einem Trick: Statt zu entscheiden, ob in einer gegebenen Markierung zwei Abläufe verzweigen, testet die Methode isSuccState() auf statische Konflikte innerhalb des Moduls und implementiert auf diese Weise ein stärkeres Kriterium. Durch diesen Trick liefert die Konstruktion u. U. einen größeren Graphen als nötig, im Ergebnis führt die Analyse jedoch zum gleichen Resultat und der Algorithmus kommt ohne die vorherige explizite Konstruktion von ggf. unendlich vielen Abläufen aus.

Während der Konstruktion des Kommunikationsgraphen überwacht die Java-Klasse ComGraph die Anzahl der Knoten und/oder die Existenz eines Bediengraphen. Falls eine der spezifizierten Restriktionen erfüllt ist, endet die Konstruktion. Der folgende Abschnitt er-



(a) Bedienenden Sequenzen

(b) Fehlerhaften Sequenzen

Abbildung 8.9: Färbung des Kommunikationsgraphen

läutert, wie die Analyse des Kommunikationsgraphen noch während der Konstruktion abläuft.

8.3.3 Analyse von Kommunikationsgraphen

Sobald der Kommunikationsgraph eines Workflow-Moduls vollständig vorliegt, lässt sich dieser mit linearem Aufwand analysieren, d. h. in bedienende und fehlerhafte Sequenzen einteilen. Um jedoch die Bedienbarkeit eines Moduls mit einem unendlichen Kommunikationsgraphen nachzuweisen, muss bereits während der Konstruktion eine Analyse stattfinden und ggf. den weiteren Aufbau des Graphen beenden. Im Quelltext der Methode `buildGraph()` (Abbildung 8.7) ist bereits die Stelle ersichtlich, an der die Analyse aufsetzt: Jeder neue Knoten wird in die Liste `toAnalyse` eingetragen. Abbildung 8.9 illustriert die Analyse ausgehend von einem gültigen Blattknoten und ausgehend von einem fehlerhaften Blattknoten.

Während der Konstruktion sind alle Knoten und Kanten des Kommunikationsgraphen initial ungefärbt, d. h. sie tragen das Farbattribut `schwarz` (in Abbildung 8.9 grau dargestellt). Sobald ein Knoten erreicht wird, der ein Blatt des Kommunikationsgraphen ist, wird dieser gefärbt: Der Knoten erhält das Farbattribut `blau` (dargestellt mit einer durchgezogenen Linie), wenn er nur den gültigen Endzustand des Workflow-Moduls umfasst (im Beispiel `[p6]`), ansonsten erhält der Knoten das Farbattribut `rot` (dargestellt mit einer gestrichelten Linie). Ob der aktuelle Knoten ein Blattknoten ist, entscheidet die Methode `isDeadEnd()`. Ein Kriterium ist die Markierung der Endstelle des Workflow-Moduls – allein oder mit weiteren Input-Stellen.

Abbildung 8.9(a) zeigt die Färbung ausgehend von einem blauen Knoten. Für diese Auf-

gabe ist die Methode setBlueUp() der Klasse ComGraph zuständig. Der Knoten v4 ist offensichtlich ein Blattknoten und umfasst nur den gültigen Enzustand des Workflow-Moduls. Jede bedienende Sequenz muss in diesem Knoten enden. Aus diesem Grund erhalten alle eingehenden Kanten das Farbattribut blau. Dann erreicht die Methode setBlueUp() einen unsichtbaren Knoten (z. B. h3). Ein unsichtbarer Knoten erhält das Farbattribut blau, wenn alle ausgehenden Kanten blau gefärbt sind – das ist sowohl für h3 als auch für h4 der Fall. Die eingehende Kante des unsichtbaren Knotens wird blau gefärbt und die Methode setBlueUp() erreicht einen sichtbaren Knoten (z. B. v1). Von diesem Knoten gibt es daher einen Pfad zum Blattknoten und damit erhält auch dieser das Farbattribut blau. Die Methode setBlueUp() wandert so lange rückwärts durch den Graphen, bis der Wurzelknoten (v0) erreicht wird oder ein unsichtbarer Knoten, der mindestens eine nicht-blaue ausgehende Kante besitzt (h2). Wenn der Wurzelknoten erreicht wird, dann ist eine bedienende Sequenz gefunden und mit der entsprechenden Restriktion endet nun die Konstruktion des Kommunikationsgraphen.

Startet die Analyse in einem fehlerhaften Knoten, dann wird die Methode setRedUp() der Klasse ComGraph aufgerufen. Abbildung 8.9(b) zeigt die Färbung ausgehend von Knoten v5. Dieser Knoten umfasst einen Zustand, in dem neben der Markierung der Endstelle noch eine Marke auf der Input-Stelle a liegt. Damit ist eine Sequenz fehlerhaft, wenn sie in diesem Zustand endet und deshalb erhält jede eingehende Kante das Farbattribut rot. Die Methode setRedUp() erreicht den unsichtbaren Knoten (z. B. h5) und auch dieser wird rot gefärbt. Hat ein unsichtbarer Knoten das Farbattribut rot, so müssen auch alle ausgehenden Kanten rot gefärbt werden. Für den Knoten h5 ist das bereits der Fall und die Methode setRedUp() wandert weiter zum sichtbaren Knoten v3. Ein sichtbarer Knoten erhält das Farbattribut rot, wenn alle ausgehenden Kanten rot gefärbt sind. Da diese Forderung für den Knoten v3 zutrifft, setzt sich die Färbung analog zum Knoten v5 fort.

Die Methode setRedUp() endet, sobald ein sichtbarer Knoten erreicht ist, der mindestens eine nicht-rote ausgehende Kante besitzt, d. h. im Knoten v0. Auf dem Weg zum Knoten v0 passiert die setRedUp() den unsichtbaren Knoten h2. Dieser besitzt eine nicht-rote ausgehende Kante. Da der Knoten h2 offensichtlich zu einer fehlerhaften Sequenz gehört, muss diese Information entlang jeder nicht-roten Kante propagiert werden. Dies Aufgabe erledigt die Methode setRedDown(). Diese Methode färbt alle Knoten rot, bis ein sichtbarer Knoten erreicht wird, der mindestens eine nicht-rote eingehende Kante besitzt. Damit terminiert der Algorithmus im Knoten v4. Abbildung 8.9(b) zeigt die endgültige Färbung am Ende der Analyse des Kommunikationsgraphen.

Im Ergebnis zeigt sich, dass das Workflow-Modul aus Abbildung 8.6(a) bedienbar ist, denn es gibt einen Bediengraphen bestehend aus den Knoten v0, h1, v1, h3 und v4. Das Modul ist jedoch nicht vollständig bedienbar, denn nur ein Ablauf wird von der konstruierten Umgebung überdeckt. Der Nachweis der vollständigen Bedienbarkeit ist im derzeitigen Stand des Prototyps noch nicht implementiert.

Fazit: Der Prototyp des Wombat4ws implementiert die wesentlichen Algorithmen zum Nachweis der Bedienbarkeit und zur Transformation von Workflow-Modulen. Damit sind die gewählte Begriffsbildung und die angegebenen Algorithmen im praktischen Einsatz validiert. Mit der Spezialisierung auf eine konkrete Syntax (z. B. BPEL4WS) und einer Integration in ein umfassendes Werkzeug (z. B. eclipse) zielt die weitere Entwicklung auf ein Werkzeug für den Einsatz auch außerhalb des akademischen Umfelds.

Bei der vorliegenden Implementierung des Prototyps lag der Fokus eher auf der prinzipiellen Machbarkeit, denn auf Fragen der Speicher- und Laufzeiteffizienz. Aus diesem Grund finden sich in dieser Arbeit auch sehr wenige Aussagen zur Komplexität der Algorithmen. In der weiteren Entwicklung soll durch den Einsatz von effizienten Methoden und Datenstrukturen und die gezielte Ausnutzung von strukturellen Eigenschaften der Aspekt der Performanz verstärkt betrachtet werden.

9 Abschließende Bemerkungen

Der vorliegende Bericht beschreibt eine Momentaufnahme der Forschungen am Lehrstuhl für Theorie der Programmierung in Bezug auf Web Services und verteilte Geschäftsprozesse. Viele der hier angeschnittenen Fragestellungen werden in anderen, teilweise bereits verfügbaren Arbeiten eingehender betrachtet und umfassend beantwortet. Dazu zählen vor allem die Petrinetz- und ASM-Semantik für BPEL4WS, sowie die graphische Repräsentation und die Analyse von Petrinetz-Modellen. Andere Arbeiten haben ganz neue Fragestellungen hervorgebracht. Im besonderen sind das die Ansätze zum Slicing von BPEL4WS-Prozessen.

Durch den fortschreitenden Prozess der Weiterentwicklung und Standardisierung der Technologien – der dank des engen Kontakts zu Kooperationspartnern bei IBM in Deutschland und Amerika durch die Ergebnisse der Arbeit bereits entscheidend beeinflusst wurde – bleibt dieses Gebiet auf Jahre hinaus ein spannendes und sich dynamisch entwickelndes Forschungsthema und ein Schwerpunkt der Arbeiten am Lehrstuhl für Theorie der Programmierung.

9.1 Erreichte Ergebnisse

Die Arbeiten in der Task Force BPEL4WS haben gezeigt, dass Web Services ein geeigneter Ansatz für die Realisierung verteilter Geschäftsprozesse sind. Mit den verschiedenen Technologien im *Web-Service-Technology-Stack* ist es möglich, einen Geschäftsprozess in allen Aspekten zu beschreiben. Die *Business Process Execution Language* BPEL4WS ist in diesem Zusammenhang auf dem Weg, *der* Industriestandard zur Modellierung von Geschäftsprozessen zu werden, obwohl die Sprache aufgrund ihrer Komplexität und ihres Aufbaus sowie der informalen Spezifikation nicht als konsistent und korrekt betrachtet werden kann. Daher war und ist es das Anliegen der Forschungen, ein formales Fundament und methodische Unterstützung für die Entwicklung Web-Service-basierter Geschäftsprozesse bereitzustellen. Die vorliegenden Ergebnisse – größtenteils noch ausführlicher in gesonderten Arbeiten dargestellt – helfen dem Anwender beim *Verständnis* der Sprache, dienen zur *Präzisierung* der Sprachen und erlauben erste *Analysen* ihrer Modelle.

Verständnis

Die aktuelle Version der Spezifikation von BPEL4WS (Version 1.1 vom 5. Mai 2003 [ACD+02]) umfasst ca. 140 Seiten, wobei nur sehr knappe Erläuterungen und dafür um so mehr Verweise auf andere Technologien des Web-Service-Technology-Stack enthalten sind. Mit der vorliegenden ASM-Semantik ([Fah04]) ist eine Definition der Sprache entstanden, die einen strukturierten Zugang über verschiedene Level der Abstraktion ermöglicht. Sie definiert präzise und verständlich den Bezug zur Middleware und der Schnittstellenbeschreibung mittels WSDL, bietet überschaubare Definitionen für jeden des Sprachbausteine

und erlaubt bei Bedarf Einblicke in die Details der Generierung und Verwaltung von Prozessinstanzen. Wie auch die Definition einer Semantik mit Petrinetzen, hatte diese Arbeit stets die gesamte Sprache mit all ihren Konzepten im Blick und geht somit entscheidend weiter als andere, vergleichbare Ansätze.

Maßgeblich zum Verständnis eines einzelnen Prozesses trägt die entwickelte graphische Repräsentation *visualBPEL* bei ([Wei03]). Diese Graphik ist einfach gehalten und orientiert sich an wenigen, intuitiven Stilmitteln. Dennoch ist sie konzeptionell durchdacht: Jedes Element der Graphik hat einen inhaltlichen Sinn und jedes Element der Sprache BPEL4WS findet sich in der Graphik wieder. Somit sind wir in der Lage, den XML-Quelltext eines jeden Geschäftsprozesses, welcher in BPEL4WS formuliert wurde, 1:1 in eine Graphik umzusetzen. Viele Fallstudien – u. a. Beispiel aus Abschnitt 7.2.1 – belegen, dass unsere Graphik intuitiv erfassbar ist und nur weniger Vorkenntnisse aus dem Bereich der Geschäftsprozessmodellierung bedarf.

Mit einem auf *visualBPEL* basierenden Editor als Komponente einer umfassenden Entwicklungsumgebung für BPEL4WS soll es dem Nutzer ermöglicht werden, auf der gleichen intuitiven Ebene der Darstellung BPEL4WS-Prozesse zu modellieren, zu simulieren und individuelle Eigenschaften zu spezifizieren und zu analysieren. Die Grundlagen hierfür sind mit den vorliegenden Ergebnissen gelegt, die Entwicklung eines solchen Editors ist Gegenstand sich anschließender Forschungen.

Präzisierung

Wie bereits erwähnt, verfolgte auch der Petrinetz-basierte Ansatz das Ziel der Definition einer vollständigen Semantik der Sprache BPEL4WS. Diese Arbeiten, die weit über den in diesem Bericht dargestellten Teil hinausgehen und gesondert in eine Diplomarbeit veröffentlicht sind ([Sta04]), haben mit einer Sammlung von Petrinetz-Mustern dieses Ziel vollends erreicht. Jedes Muster für sich erläutert auf verständliche Weise Struktur und Verhalten eines Sprachelements von BPEL4WS. Das Zusammenspiel der Muster wurde zum einen während der Konstruktion anschaulich dargelegt und zum anderen für die komplexe Eigenschaft des kontrollierten Prozessabbruchs formal bewiesen. Besonders hervorzuheben ist die präzise Herausarbeitung von widersprüchlichen Forderungen und Inkonsistenzen in der ursprünglichen Sprachdefinition.

Dem komplexen Konzept des *scopes* nähert sich in einer weiterführenden Arbeit der Ansatz der *BPEL-Boxen* [Fre04], wobei hier stärker auf eine Trennung der Konzepte Wert gelegt wird. Diese Arbeit erbringt im Nachhinein durch die Betrachtung von (im besonderen langlaufenden) Transaktionen die Motivation für einige auf den ersten Blick verwirrend erscheinende Konzepte in BPEL und trägt somit maßgeblich zum Verständnis bei. Darüber hinaus schafft sie den konzeptionellen Vorbau für bereits angekündigte Modifikationen der Sprachdefinition und bietet den Ausgangspunkt weiterer Forschungen.

Analysen

Die Erfahrung lehrt, dass in den nächsten zwei bis fünf Jahren der Weiterentwicklung viele Konzepte in der Web-Service-Architektur geändert, verfeinert oder ausgetauscht werden. Aus diesem Grund löst sich die Analyse größtenteils von der konkreten Spezifikation und

betrachtet die zugrunde liegenden Konzepte und stützt sich auf Petrinetze zur Modellierung von Web Services. Petrinetze sind eine etablierte Methode zur Modellierung in sich geschlossener Geschäftsprozesse. Sie gestatten es, wie mit der Definition einer Semantik für BPEL4WS gezeigt wurde, die syntaktischen Strukturen existierender Technologien präzise und anschaulich zu unterlegen. Damit wird eine größere Robustheit gegen syntaktische Änderungen bei gleichzeitig enger semantischer Bindung an die Thematik verteilter Geschäftsprozesse erreicht.

Aufbauend auf diesem formalen Fundament ist es möglich, die *Bedienbarkeit* – eine essenzielle Qualitätseigenschaft von Web Services – effektiv und konstruktiv nachzuweisen. Dieser Bericht gibt einen informalen Überblick über die Methode. Alle notwendigen Begriffe wurden in einer Dissertation ([Mar04]) ausführlich hergeleitet und die Algorithmen präzise definiert sowie ihre Korrektheit formal bewiesen. Die Handhabbarkeit der Methode ist durch die prototypische Implementierung der Analyse belegt [Mar03b].

Neben der Analyse eines einzelnen Web Service stehen Beziehungen zwischen verschiedenen derartigen Komponenten im Blickpunkt: Als Voraussetzung für die Komposition von Web Services wird in dieser Arbeit die *semantische Kompatibilität* der Komponenten gefordert. Zwei Workflow-Module sind semantisch kompatibel, wenn das komponierte System bedienbar ist. Wesentlich für den dynamischen Aufbau verteilter Geschäftsprozesse ist die *Austauschbarkeit* von Komponenten. Diese Eigenschaft wurde auf eine Simulationsbeziehung zwischen deren Kommunikationsgraphen zurückgeführt und ist somit ebenfalls effektiv entscheidbar. Ein wichtiger Arbeitsschritt vor der Veröffentlichung eines Web Service ist die *Abstraktion* von internen Details. So lässt sich ein vereinfachtes Modell eines Web Service, das sich nach außen hin genau so verhält wie das Original, direkt aus dem Bediengraphen des Moduls ableiten.

9.2 Weitere Forschungen

Das Ziel dieser Arbeit ist die Entwicklung einer durchgängigen Methode zur Modellierung und Analyse verteilter Geschäftsprozesse basierend auf der Web-Service-Architektur. Gestützt auf formale Methoden wie Petrinetze und ASM ist es gelungen, Kriterien, Regeln und Verfahren zu definieren und zu implementieren, die die wesentlichen Schritte in einem Vorgehensmodell methodisch unterstützen.

Damit ist die Entwicklung jedoch nicht abgeschlossen. Weitere Forschungen zielen auf die tiefgreifende Betrachtung von Datenaspekten insbesondere im Zusammenhang mit verteilten Transaktionen, das Zusammenspiel mehrerer Web Services, die Optimierung der Analysemethoden und die Fragen der Repräsentation, Ergonomie und Dokumentation der Verfahren. Im Folgenden skizzieren wir zu diesen vier Bereichen konkrete Forschungsvorhaben.

Datenaspekte

In der bisherigen Arbeit wurden die Datenaspekte verteilter Geschäftsprozesse nur in geringem Umfang berücksichtigt, um zu einfachen und entscheidbaren Kriterien für die Kontrollstruktur zu gelangen. Durch die Vereinigung der beiden, in diesem Projekt entstandenen Sematikdefinitionen sollen die formalen Modelle stufenweise um Datenaspekte angereichert

werden, wobei angepasste und neue Analysemethoden entwickelt werden. Dazu zählen vor allem Fragen der Kompatibilität von Daten und der Konsistenz von Kontroll- und Datenstruktur. Grundlage für diese Forschungen bildet u. a. [Kin01].

Die Kontrollstruktur bildet die kausale Ordnung aller Aktivitäten eines Geschäftsprozesses ab. Dabei vermengen sich i. Allg. Aktivitäten, die zum Erreichen eines Unternehmensziels wichtig sind, mit solchen, die Ausnahmen und Fehler behandeln und ggf. die Auswirkungen fehlerhafter Aktivitäten kompensieren. Um die Komplexität von Modellen zu begrenzen, erscheint es sinnvoll, diese orthogonalen Aspekte voneinander zu trennen. Ein möglicher Ansatz sieht vor, den so genannten *positiven Kontrollfluss* wie gehabt als Petrinetz abzubilden und für die Fehlerbehandlung und Kompensation andere Techniken der Spezifikation zu entwickeln. Diese Techniken sollten einerseits für den Modellierer intuitiv sein und andererseits automatisch in Petrinetz-Strukturen überführt werden können, so dass eine im Hintergrund ablaufende Analyse auf den bisherigen Verfahren aufsetzen kann. Erste Ansätze zu dieser Thematik finden sich in [DDGJ].

Beide zuvor skizzierten Ansätze laufen zusammen, wenn es um die Unterstützung von verteilten Transaktionen geht. Gerade auf diesem Gebiet werden von Seiten der Industrie zurzeit mit Hochdruck Anforderungen und Vorschläge für Standards entwickelt. Aufgabe der Forschung ist es, das Konzept der Transaktionen als Baustein in ein Gesamtkonzept zu integrieren und die Wechselwirkungen mit anderen Konzepten, z. B. der Fehlerbehandlung, zu beschreiben.

Komposition

Für die Komposition von Web Services gibt es zwei grundsätzliche Ansätze: Zum einen betrachtet man das Zusammenspiel jeweils aus der Perspektive eines beteiligten Web Service. In der Beschreibung dieses Web Service finden sich neben der Modellierung der eigenen Struktur auch Anforderungen an die Partner. Man nennt diese Art der Aggregation auch *Peer-to-Peer-Ansatz*. Die Sprache BPEL4WS ist ein prominenter Vertreter dieser Art der Komposition.

Zum anderen modelliert man das Zusammenspiel von einer übergeordneten Stelle aus. Hierbei liegt der Fokus weniger auf der detaillierten Beschreibung der einzelnen Web Services, als vielmehr auf der Spezifikation ihrer Interaktion in einem globalen Modell. Man nennt diese Art der Komposition auch *Choreographie* von Web Services. Eine Methode zur Beschreibung der Kommunikation zwischen Web Services gemäß dieser Art der Komposition ist die Sprache *Web Service Choreography Interface* WSCI. Sie beschreibt nicht die Implementierung der einzelnen Web Services, sondern lediglich ihr beobachtbares Verhalten.

Da die Schwerpunkte beider Ansätze sich unterscheiden, ist nicht zu erwarten, dass ein Ansatz durch den anderen verdrängt werden wird. Vielmehr stellt sich mit der Kombination der beiden eine wichtige Aufgabe, deren Lösung die Entwicklung Web-Service-basierter Systeme erleichtern wird. Beispiele für konkrete Fragestellungen sind:

- Gegeben sei ein beliebig komplexes WSCI-Modell der Choreographie mehrerer Web Services: Ist ein solches Modell vernünftig – d. h., ist das Modell global betrachtet fehlerfrei und lassen sich die einzelnen Web Services lokal unabhängig implementieren?
- Gegeben sei ein WSCI-Modell und ein BPEL4WS-Modell eines der beteiligten Web

Services: Sind die beiden Modelle konsistent – d. h., ist das BPEL-Modell eine korrekte Implementierung der Spezifikation dieses Web Service (aus dem WSCI-Modell)?

- Gegeben sei die abstrakte Spezifikation mehrerer Web Services mit WSCI und die konkrete Implementierung eines weiteren mit BPEL: Sind diese Web Services kompatibel – d. h., bilden die spezifizierten Web Services eine bedienende Umgebung des implementierten Web Service?

Mit dem zur Verfügung stehenden Arsenal formaler Methoden erscheint es möglich, diese (und ggf. weitere) Fragestellungen effektiv und möglichst effizient beantworten zu können.

Analysemethoden

Eine wichtige Aufgabe weiterer Forschungen ist es, Eigenschaften herauszuarbeiten, die für Modellierer und Benutzer realer Web Services von Relevanz sind. Dazu zählen sowohl temporale Aspekte, wie die generelle Erreichbarkeit von gewünschten Zuständen oder Aktivitäten, als auch betriebswirtschaftliche Kenngrößen wie Durchsatz und Bearbeitungszeit. Neben den bisherigen Methoden bedarf es neuer Verfahren zum Nachweis solcher Eigenschaften. Ein diesbezüglicher Ansatz ist die Integration stochastischer Simulation in das Vorgehensmodell, analog zum Ansatz aus [MR01].

Aber auch die bereits entwickelten Analysemethoden sollten in weiteren Forschungen verbessert werden. In Bezug auf die syntaxbasierten Verfahren kann durch die Beschäftigung mit Fallstudien die Sammlung von Pattern vervollständigt werden. Bei den dynamischen Verfahren stellt die Explosion des Zustandsraums ein gewichtiges Problem dar. Eine Herausforderung für weitere Forschungen ist es, diesem Problem durch geeignete Reduktionstechniken zu begegnen. Als geeignete Grundlage erscheinen die Arbeiten zur *Partial Order Reduction*, z. B. [Val88], [Sch00a] und [Sch02].

Anwendbarkeit

Ein großes Gebiet für die weitere Forschung befasst sich mit der *Ergonomie* der entwickelten Methode. Mit der entstandenen Graphik für BPEL4WS existiert bereits eine angemessene Repräsentation der Prozessmodelle. Weitere Forschungen sollen dazu führen, die Dynamik der Prozessmodelle durch Animation in die Graphik einfließen zu lassen, als auch die Visualisierung der Ergebnisse der Analyse zu ermöglichen.

Weiterhin zählen zur Ergonomie die Handhabung der entwickelten Werkzeuge und der einfache Austausch von Modellen und Informationen mit anderen Werkzeugen. Der entstandene Prototyp ist auf die Funktionalität fokussiert, seine modulare Architektur ermöglicht jedoch sowohl die Erweiterung um zusätzliche Verfahren der Analyse, als auch die Integration in ein umfassendes Entwicklungswerkzeug. In weiteren Arbeiten soll die Anbindung an die universelle Werkzeug-Plattform *eclipse* [Gal02] realisiert werden.

Letztlich gehört zur Ergonomie die Dokumentation der Methode. Anhand von Fallstudien sollen ein Vorgehensmodell (ähnlich [PY02]) und Richtlinien der Modellierung entwickelt und in einem Tutorial beschrieben werden. An dieser Stelle ist der enge Kontakt zu IBM Garant für eine praxisnahe Forschung.

Literaturverzeichnis

[Aal98] van der Aalst, W. M. P.: The Application of Petri Nets to Workflow Management. In: *Journal of Circuits, Systems and Computers* 8 (1998), Nr. 1, S. 21–66

[ACD+02] Andrews ; Curbera ; Dholakia ; Goland ; Klein ; Leymann ; Liu ; Roller ; Smith ; Thatte ; Trickovic ; Weerawarana: *BPEL4WS–Business Process Execution Language for Web Services.* Version 1.1: Vorschlag zur Standardisierung, Juli 2002. – http://ibm.com/developerworks/webservices/library/ws-bpel/

[ACKM02] Alonso, Gustavo ; Casati, Fabio ; Kuno, Harumi ; Machiraju, Vijay: *Web Services.* Springer-Verlag, Berlin, Heidelberg, New York, 2002

[ADLH+02] Atkinson ; Della-Libera ; Hada ; Hondo ; Hallam-Baker ; Klein ; LaMacchia ; Leach ; Manferdelli ; Maruyama ; Nadalin ; Nagaratnam ; Prafullchandra ; Shewchuk ; Simon: *Web Services Security.* Version 1.0: Vorschlag zur Standardisierung, April 2002. – http://ibm.com/developerworks/webservices/library/ws-secure/

[AH02] van der Aalst, W.M.P. ; van Hee, K.M.: *Workflow Management: Models, Methods, and Systems.* MIT Press, Cambridge, MA, 2002

[ASMa] Die ASM Webseite: http://www.eecs.umich.edu/gasm/

[ASMb] Die Webseite zu AsmGofer: http://www.tydo.de/AsmGofer/

[Asmc] Die Webseite zu AsmL: http://www.research.microsoft.com/foundations/asml/

[BCR02] Bellwood, T. ; Clément, L. ; von Riegen, C.: *UDDI – Universal Discovery, Description, and Integration.* Version 3.0. Standard: UDDI.org, September 2002. – http://uddi.org/pubs/uddi_v3.htm

[BEK+00] Box ; Ehnebuske ; Kakivaya ; Layman ; Mendelsohn ; Nielsen ; Thatte ; Winer: *SOAP – Simple Object Access Protocol.* Version 1.1. Standard: W3C, Mai 2000. – http://www.w3.org/TR/SOAP/

[BG03] Blass, A. ; Gurevich, Y.: Abstract State Machines Capture Parallel Algorithms. In: *ACM Transactions on Computational Logic* Vol.4,Issue 4 (2003), Oktober, S. 578–651

[BPMM00] Bray ; Paoli ; McQueen ; Maler: *XML – Extensible Markup Language.* Version 1.0 (Second Edition). Standard: W3C, Oktober 2000. – http://www.w3.org/TR/REC-xml

[BR94] Börger, E. ; Rosenzweig, D.: A Mathematical Definition of Full Prolog. In: *Science of Computer Programming* Bd. 24. North-Holland, 1994, S. 249–286

[BS00] Börger, E. ; Schmid, J.: Composition and Submachine Concepts for Sequential ASMs. In: Clote, P. (Hrsg.) ; Schwichtenberg, H. (Hrsg.): *Computer Science Logic (Proceedings of CSL 2000)* Bd. 1862, Springer-Verlag, 2000, S. 41–60

[BS03] Börger, E. ; Stärk, R.: *Abstract State Machines – A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003. – ISBN 3–540–00702–4

[CB+03] Curbera, A. ; Box, D. [u. a.]: Web Services Addressing (WS-Addressing) / BEA, IBM, Microsoft. 2003. – Specification. http://msdn.microsoft.com/ws/2003/03/ws-addressing/

[CCC+02a] Cabrera ; Copeland ; Cox ; Feingold ; Freund ; Johnson ; Kaler ; Klein ; Langworthy ; Nadalin ; Orchard ; Robinson ; Shewchuk ; Storey: *Web Services Coordination.* Version 1.0: Vorschlag zur Standardisierung, August 2002. – `http://ibm.com/developerworks/library/ws-coor/`

[CCC+02b] Cabrera ; Copeland ; Cox ; Freund ; Klein ; Storey ; Thatte: *Web Services Transaction.* Version 1.0: Vorschlag zur Standardisierung, August 2002. – `http://ibm.com/developerworks/webservices/library/ws-transpec/`

[CCMW01] Christensen, E. ; Curbera, F. ; Meredith, G. ; Weerawarana, S.: *WSDL – Web Services Description Language.* Version 1.1. Standard: W3C, März 2001. – `http://www.w3.org/TR/wsdl`

[CDK03] Curbera, Francisco ; Duftler, Matthew ; Khalaf, Rania: Business Process with BPEL4WS: Learning BPEL4WS, Part 5 / Interantional Business Machines Corporation. 2003. – mar

[Che93] Cheng, J.: Slicing concurrent programs. In: *Automated and Algorithmic Debugging, 1st International Workshop, AADEBUG93* (1993), S. 223–240

[Che97] Cheng, J.: Dependence analysis of parallel and distributed programs and its applications. In: *International Conference on Advances in Parallel and Distributed Computing* (1997)

[Cla99] Clark, James: XSL Transformations (XSLT) Version 1.0 / W3C. URL `http://www.w3.org/TR/xslt`, November 1999. – W3C Recommendation

[DDGJ] Derks, W. ; Dehnert, J. ; Grefen, P. ; Jonker, W.: Customized Atomicity Specification for Transactional Workflow. – Forschungsbericht

[DK02] Duftler, Matthew ; Khalaf, Rania: Business Process with BPEL4WS: Learning BPEL4WS, Part 3 / Interantional Business Machines Corporation. 2002. – oct

[DW00] Del Castillo, G. ; Winter, K.: Model Checking Support for the ASM High-Level Language. In: Graf, S. (Hrsg.) ; Schwartzbach, M. (Hrsg.): *Proceedings of the 6th International Conference TACAS 2000* Bd. 1785, Springer-Verlag, 2000, S. 331–346

[EGG+01] Eschbach, R. ; Glässer, U. ; Gotzhein, R. ; von Lövis, M. ; Prinz, A.: Formal Definition of SDL-2000 - Compiling and Running SDL Specifications as ASM Models. In: *Journal of Universal Computer Science* 7 (2001), November, Nr. 11, S. 1024–1049

[Fah04] Fahland, Dirk: *Ein Ansatz einer formalen Semantik der Business Process Execution Language for Web Services mit Abstract State Machines.*, Humboldt-Universität zu Berlin, Studienarbeit, 2004

[FOW87] Ferrante, J. ; Ottenstein, K.J. ; Warren, J.D.: The program dependence graph and its use in optimization. In: *ACM Trans. Prog. Lang. Syst.* 9 (1987), Nr. 3, S. 319–349

[Fra01] Fraunhofer-Institut für Software- und Systemtechnik: *Workflow-Management und Groupware.* Produkt-Informationen: ISST, 2001. – http://www.do.isst.fhg.de/workflow/produkte/

[Fre91] Free Software Foundation: *GNU General Public License.* Version 2: FSF Inc., 59 Temple Place, Boston, USA, Juni 1991. – http://www.gnu.org/copyleft/gpl.html

[Fre04] Frenkler, Carsten: *Ein Modell zur Integration von Transaktionskonzepten in Geschäftsprozesse mit Petrinetzen.*, Humboldt-Universität zu Berlin, Studienarbeit, 2004

[Gal02] Gallardo, David: *Getting started with the Eclipse Platform.* Whitepaper: IBM developerWorks, November 2002. – http://ibm.com/developerworks/java/library/os-ecov/

[Gar02] Gartner Research: 2002 Emerging Technologies Hype Cycle – Trigger to Peak. In: *Monthly Research Review* (2002), Juni. – Gartner Group Research, Inc.

[GKN02] Gansner, Emden ; Koutsofios, Eleftherios ; North, Stephen: *Drawing graphs with dot.* dot User's Manual.: AT&T Labs Research, Februar 2002. – http://www.research.att.com/sw/tools/graphviz/dotguide.pdf

[Got00] Gottschalk, Karl: *Web Services architecture overview.* Whitepaper: IBM developerWorks, September 2000. – http://ibm.com/developerWorks/web/library/w-ovr/

[GS97] Gurevich, Y. ; Spielmann, M.: Recursive Abstract State Machines. In: *J.UCS: Journal of Universal Computer Science* 3 (1997), April, Nr. 4, S. 233–246

[Gur] GUREVICH, Y.: Evolving Algebras 1993: Lipari Guide. In: *Specification and Validation Methods*

[Gur85] GUREVICH, Y.: A new thesis. In: *American Mathematical Society Abstracts* (1985), August, S. 317

[Gur97] GUREVICH, Y.: May 1997 Draft of the ASM Guide / University of Michigan EECS Department. 1997. – Forschungsbericht. CSE-TR-336-97

[Gur00] GUREVICH, Y.: Sequential Abstract-State Machines Capture Sequential Algorithms. In: *ACM Tranactions on Computational Logic* Vol.1 No.1 (2000), Juli, S. 77–111

[GV02] GIRAULT, Claude (Hrsg.) ; VALK, Rüdiger (Hrsg.): *Petri Nets for System Engineering – A Guide to Modeling Verification and Applications.* Springer-Verlag Berlin Heidelberg New York, Januar 2002. – ISBN 3–540–41217–4

[Hee98] HEERING, Hans: *Objektorientierte Middleware-Konzepte und Infrastrukturen in verteilten Anwendungen.*, AKAD Rendsburg, Hochschule für Berufstätige, Diplomarbeit, Oktober 1998

[Hos02] HOSTMANN, Bill: *Web Services for Business Intelligence – Hype and Reality.* Whitepaper: Gartner Group Research, Inc., Juni 2002

[Kin01] KINDLER, Ekkart: *Systematische Spezifikation und Verifikation von Konsistenzprotokollen.* Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät, Habilitationsschrift, 2001

[Kre01] KREGER, Heather: *WSCA – Web Services Conceptual Architecture.* Whitepaper: IBM Software Group, Mai 2001. – `http://ibm.com/webservices/pdf/WSCA.pdf`

[KW99] KINDLER, Ekkart ; WEBER, Michael: The Petri Net Kernel – An Infrastructure for Building Petri Net Tools. In: *20th International Conference on Application and Theory of Petri Nets – Petri Net Tool Presentations*, College of William and Mary, Williamsburg, Virginia, USA, Juni 1999, S. 10–19

[KW01] KINDLER, Ekkart ; WEBER, Michael: A universal module Concept for Petri nets. In: UND ROBERT LORENZ, Gabriel J. (Hrsg.): *Proceedings des 8. Workshops AWPN*, Katholische Universität Eichstätt, Oktober 2001, S. 7–12

[Ley01] LEYMANN, Frank: *WSFL – Web Services Flow Language.* Whitepaper: IBM Software Group, Mai 2001. – `http://ibm.com/webservices/pdf/WSFL.pdf`

[Mar03a] MARTENS, Axel: On Compatibility of Web Services. In: JUHÁS, Gabriel (Hrsg.) ; KINDLER, Ekkart (Hrsg.): *Petri Net Newsletter*, Nr. 65, Oktober 2003, S. 12–20

[Mar03b] Martens, Axel: Wombat4ws– *Workflow Modeling and Business Analysis Toolkit for Web Services* / Humboldt-Universität zu Berlin. 2003. – Manual. `http://www.informatik.hu-berlin.de/top/wombat`

[Mar04] Martens, Axel: *Verteilte Geschäftsprozesse – Modellierung und Verifikation mit Hilfe von Web Services*, WiKu-Verlag Stuttgart, Dissertation, 2004

[Mica] Michael Weber et al.: *The Petri Net Kernel – An Infrastrucure for Building Petri Net Tools.* Petri Net Kernel Team: Humboldt-Universität zu Berlin. – `http://www.informatik.hu-berlin.de/top/pnk/`

[Micb] Michael Weber et al.: *PNK User's Manual.* Petri Net Kernel Team: Humboldt-Universität zu Berlin. – `http://www.informatik.hu-berlin.de/top/pnk/doku/pnk-guide/quickStart.html`

[Moh02] Mohan, C.: Dynamic E-business – Trends in Web Services. In: Buchmann et al. (Hrsg.): *Proceedings of TES'02*, Springer-Verlag, 2002 (LNCS 2444), S. 1–6

[MR01] Martens, Axel ; Richter, Wolf: *Analysis of Message Flow Systems.* Projektbericht. Institut für Informatik: Humboldt-Universität zu Berlin, 2001

[NNH99] Nielson, Flemming ; Nielson, Hanne R. ; Hankin, Chris: *Principles of Program Analysis.* Springer, 1999

[Now03] Nowack, A.: Deciding the Verificiation Problem for Abstract State Machines. In: Börger, E. (Hrsg.) ; Gargantini, A. (Hrsg.) ; Riccobene, E. (Hrsg.): *Abstract State Machines 2003 – Advances in Theory and Practice* Bd. LNCS 2589, Springer-Verlag, 2003, S. 341–351

[PY02] Papazoglou, M. P. ; Yang, J.: Design Methodology for Web Services and Business Processes. In: Buchmann et al. (Hrsg.): *Technologies for E-Services: Proceedings of TES'02*, Springer-Verlag Heidelberg, 2002 (LNCS 2444), S. 54–64

[Rei85] Reisig, W.: *Petri Nets.* EATCS Monographs on Theoretical Computer Science. Berlin, Heidelberg, New York, Tokyo : Springer-Verlag, 1985

[Rei03] Reisig, W.: On Gurevich's theorem on sequential algorithms. In: *Acta Informatica* Vol. 39, No. 5 (2003), S. pp 273–305

[RW03] Reisig, Wolfgang ; Weber, Michael: *Task Force – Geschäftsprozesssprache BPEL4WS.* Projektbeschreibung. Lehrstuhl Theorie der Programmierung: Inst. für Informatik, Humboldt-Universität zu Berlin, 2003

[Sch00a] Schmidt, Karsten: LoLA – A Low Level Analyser. In: Nielsen, M. (Hrsg.) ; Simpson, D. (Hrsg.): *International Conference on Application and Theory of Petri Nets*, Springer-Verlag, 2000 (LNCS 1825), S. 465 ff.

[Sch00b] SCHMIDT, Karsten: LoLA – a Low Level Petri net Analyzer. / Humboldt-Universität zu Berlin. 2000. – Forschungsbericht. http://www.informatik.hu-berlin.de/~kschmidt/lola.html

[Sch02] SCHMIDT, Karsten: *Explicit State Space Verification.* Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät, Habilitationsschrift, 2002

[Sle01] SLEEPER, Brent: *Defining Web Services.* An Analysis Memo: The Stencil Group, Juni 2001. – http://www.stencilgroup.com/ideas_scope_200106wsdefined.pdf

[Sta90] STARKE, Peter H.: *Analyse von Petri-Netz-Modellen.* Leitfäder und Monographien der Informatik. Stuttgart, Germany : B.G. Teubner Verlag, 1990

[Sta04] STAHL, Christian: *Transformation von BPEL4WS in Petrinetze.*, Humboldt-Universität zu Berlin, Diplomarbeit, April 2004

[Ste00] STEFAN RÖMER: *Theorie und Praxis der Netzentfaltungen als Grundlage für die Verifikation nebenläufiger Systeme.*, Technischen Universität München, Dissertation, 2000

[Tha01] THATTE, Satish: *XLANG – Web Services for Business Process Design.* Initial Public Draft: Microsoft Corporation, Mai 2001. – http://www.gotdotnet.com/team/xml_wsspecs/xlang-c

[Tid00] TIDWELL, Doug: *Web services – the Web's next revolution.* Whitepaper: IBM developerWorks, November 2000. – http://ibm.com/developerWorks

[Tip95] TIP, F.: A survey of program slicing techniques. In: *Journal of programming languages* 3 (1995), Nr. 3

[Val88] VALMARI, Antti: *State Space Generation – Efficiency and Practicality.*, Tampere University of Technology, Dissertation, 1988

[VBA00] VERBEEK, H.M.W. ; BASTEN, T. ; VAN DER AALST, W.M.P.: *Woflan 2.2.* Manual: Eindhoven University of Technology, September 2000. – http://tmitwww.tm.tue.nl/research/workflow/woflan/

[WADH02] WOHED, P. ; VAN DER AALST, W.M.P. ; DUMAS, M. ; TER HOFSTEDE, A.H.M.: Pattern Based Analysis of BPEL4WS. / Queensland University of Technology. Brisbane, Australia, 2002. – QUT Technical report, FIT-TR-2002-04

[Web02] WEBER, Michael: *Allgemeine Konzepte zur software-technischen Unterstützung verschiedener Petrinetz-Typen.*, Humboldt-Universität zu Berlin, Dissertation, 2002

[Wei79] WEISER, M.: *Program slices: formal, psychological and practical investigations of an automatic program abstraction method*, University of Michigan, PhD thesis, 1979

[Wei84] Weiser, M.: Program slicing. In: *IEEE Trans. Softw. Eng.* 10 (1984), Nr. 4, S. 352–357

[Wei03] Weinberg, Daniela: *Graphische Repräsentation von BPEL4WS.*, Humboldt-Universität zu Berlin, Studienarbeit, 2003

[WK03] Weber, Michael ; Kindler, Ekkart: The Petri Net Markup Language. In: Ehrig, H. (Hrsg.) ; Reisig, W. (Hrsg.) ; Rozenberg, G. (Hrsg.) ; Weber, H. (Hrsg.): *Petri Net Technology for Communication Based Systems.*, Springer-Verlag, 2003 (LNCS 2472). – to appear

[WWV+97] Weber, M. ; Walter, R. ; Völzer, H. ; Vesper, T. ; Reisig, W. ; Peuker, S. ; Kindler, E. ; Freiheit, J. ; Desel, J.: DAWN. Petrinetzmodelle zur Verifikation Verteilter Algorithmen. / Institut für Informatik, Humboldt-Universität zu Berlin. 1997. – Informatik-Bericht Nr. 88