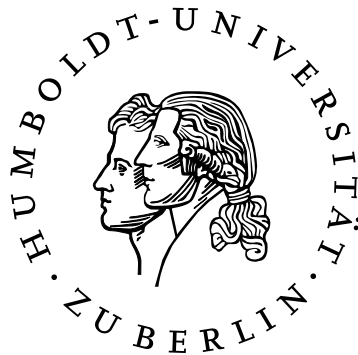


Humboldt-Universität zu Berlin
Institut für Informatik



Studienarbeit

Ein Ansatz einer formalen Semantik der
Business Process Execution Language for Web Services
mit Abstract State Machines

Dirk Fahland
30. Juni 2004

Betreuer: Dr. Axel Martens

Zusammenfassung

In dieser Arbeit stellen wir einen Ansatz zur Definition einer formalen Semantik für die Business Process Execution Language for Web Services (kurz BPEL4WS) von IBM, Microsoft und deren Industriepartnern vor. Zur Formalisierung wählen wir den Abstract-State-Machine-Formalismus (kurz ASM), dessen theoretische Fundierung es uns erlaubt, die Semantik von BPEL4WS auf der selben Abstraktionsebene zu formalisieren, die in der informalen BPEL4WS-Spezifikation vorgegeben ist. Wir werden den inneren Aufbau der Sprache präzise, formal abbilden und damit eine intuitiv und anschaulich nachvollziehbare Entsprechung zwischen den Abläufen eines BPEL4WS-Prozesses gemäß der gegebenen informalen Semantik und unserer formalen Semantik aufzeigen.

Dazu analysieren wir die Struktur von BPEL4WS und zeigen mit welchen Mitteln des ASM-Formalismus diese adäquat, formal erfasst werden und wie in ASM notierte Spezifikationen zu lesen sind. Hierzu werden wir beispielhaft ausgewählte, syntaktische Konstrukte von BPEL4WS nach unserem Ansatz formalisieren.

Die vorliegende Arbeit bezieht sich auf die informale BPEL4WS-Spezifikation v1.1, veröffentlicht am 5. Mai 2003.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Web Services	1
1.2	BPEL4WS	3
1.3	Ansatz für eine formale Semantik mit Abstract State Machines	6
2	Abstract-State Machines	11
2.1	Sequentielle Abstract-State Machines	11
2.2	Asynchrone Verteilte Abstract-State Machines	14
2.3	ASM beschreiben Transitionssysteme	14
2.4	ASM in der BPEL4WS-Spezifikation	16
2.5	Weiteres zu Abstract-State Machines	18
3	Übersetzung von Strukturen	19
3.1	Statische, prozess-beschreibende Strukturen	20
3.2	Abstraktion von Daten	24
3.3	Aktivitätsbaum	26
4	Übersetzung des Verhaltens	27
4.1	Semantische Domäne eines BPEL4WS-Prozesses im ASM-Ansatz	27
4.2	Ausführung eines BPEL4WS-Prozesses	27
4.3	Aktivitätszustände	29
4.4	Sequence	32
4.5	Reply	36
5	Schlußfolgerungen und Ausblick	42
	Literatur	45

1 Einleitung

Die vorliegende Arbeit ist Umfeld der Modellierung verteilter Geschäftsprozesse mit Hilfe von Web Services entstanden und beschäftigt sich mit der *Business Process Execution Language for Web Services* (kurz BPEL4WS) [CGK⁺03], für die bislang keine formale Definition ihrer Semantik existiert.

BPEL4WS ist auf dem Weg Industriestandard zur Modellierung von Geschäftsprozessen zu werden, obwohl die Sprache aufgrund ihrer Komplexität und ihres Aufbaus nicht allein anhand der informalen Dokumente als konsistent und korrekt bewiesen werden kann. Daher ist eine formale Semantik für BPEL4WS dringend erforderlich und die Entwicklung einer solchen ist Gegenstand dieser Arbeit.

Wir analysieren die Struktur von BPEL4WS und beschreiben an Beispielen, wie mit Hilfe von *Abstract-State Machines* (kurz: ASM) [Gur97] eine formale Semantik für BPEL4WS definiert werden kann und welche Prinzipien und Eigenschaften des ASM-Formalismus uns dabei leiten.

Diese Arbeit richtet sich an all Jene, die an BPEL4WS interessiert sind, sowie an die Personen, die einen Einblick in formale Spezifikationen mit ASMs erhalten möchten.

1.1 Web Services

Die zunehmende Vernetzung der Welt hat auch vor Geschäftsprozessen und Arbeitsabläufen nicht halt gemacht. Es ist inzwischen klar ersichtlich, dass Unternehmen ihre Geschäfte in größerem Maße als bisher so weit wie möglich automatisiert abwickeln möchten. Insbesondere soll zwischen verschiedensten und auch wechselnden Kooperationspartnern eine stabile und zuverlässige Kommunikation möglich sein. Dabei entstehen in der Gesamtheit verteilte Systeme, deren jeweilige lokale Arbeitsweise und Implementation verschiedener nicht sein könnte. Daraus erwachsen eine ganze Reihe von Problemen: Unterschiedliche Schnittstellen und Datenformate erschweren die Kommunikation. Es ist schwierig aus einer Vielzahl von Anbietern den geeigneten zu finden. Die Arbeitsweise eines Geschäftspartners kann sich von den Auffassungen der anderen dazu soweit unterscheiden, dass eine Zusammenarbeit unmöglich wird.

Der aktuelle Lösungsansatz für diese Probleme mündet im *Web-Service-Paradigma*: Software kommuniziert direkt mit Software in einer homogenen Kommunikationsschicht. Damit einher geht die Definition des Web Services: Ein *Web Service* ist eine modular abgeschlossene Softwarekomponente, die ihre Funktionalität über eine standardisierte Schnittstelle beschreibt und anbietet [Got00]. Es genügt, die Schnittstelle zu kennen, um den Web Service korrekt anzusprechen. Somit können Details der Implementation versteckt werden.

Ausgehend von den anfangs genannten Problemen wurden in den letzten Jahren von der Industrie drei Basistechnologien entwickelt, um jede Softwarekomponente in einen Web Service zu überführen. Die Software beschreibt ihre Funktionalität anhand einer WSDL-Schnittstelle, bietet sie mittels UDDI an und tauscht Nachrichten über SOAP aus. Letzteres, das *Simple Object Access Protocol*, codiert Nachrichteninhalte in XML und arbeitet auf der Grundlage von Standardprotokollen wie HTTP. Die *Web Service Description Language* (WSDL) [CCMW01] beschreibt die vom Web Service bereit gestellten Operationen als zustandsloses Protokoll zum Nachrichtenaustausch. Jede dieser Operationen kann von

einem anderen Web Service angesprochen werden. Die *Universal Discovery, Description and Integration* (UDDI) entspricht einem zentralen Verzeichnis, welches die Beschreibung und die Adressen der Web Services in XML speichert. Es kann durchsucht werden und liefert Adressen auf Web Services, die die gewünschte Funktionalität anbieten [CDR⁺02]. Dabei kann eine UDDI-Implementation auch selbst über WSDL-Schnittstellen angesprochen werden [CJ03].

SOAP, WSDL und UDDI können als de-facto Standards zur Kommunikation zwischen Softwarekomponenten betrachtet werden [WADH02]. Dennoch erfordert die Aufgabenstellung, die gewonnene Funktionalität weiter auszubauen. So ist es beispielsweise wünschenswert für einen Web Service, weitere Eigenschaften wie Zuverlässigkeit, zeitliches Verhalten oder Datensicherheit zu spezifizieren [Kre03]. Des weiteren besteht seitens der Industrie der Bedarf auch langlebige Interaktionen zwischen Geschäftspartnern automatisiert abzuwickeln sowie mehrere gegebene Web Services zu einem neuen Web Service zu *komponieren* [WADH02]. Keine dieser Aufgaben kann durch die bisher genannten Technologien im Web-Service-Ansatz gelöst werden. Sie bilden jedoch das Fundament für die Technologien, die die Aufgaben lösen können. Aus dieser Situation heraus hat IBM mit seinen Industriepartnern einen *Web Service Technology Stack* entworfen.

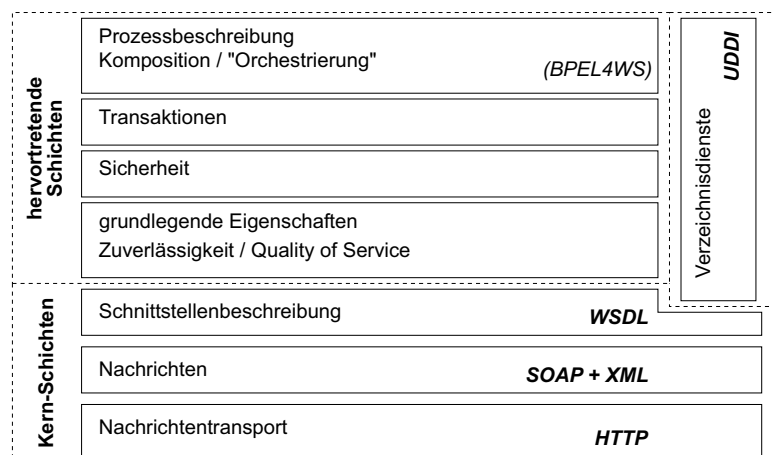


Abbildung 1: Konzeptueller Ansatz eines Web Service Technology Stacks mit den bereits verwendeten Technologien HTTP, XML, SOAP, WSDL, UDDI

Der Stack besteht aus verschiedenen Schichten von denen jede einzelne die Funktionalität der darunterliegenden Schichten nutzt und die eigenen Funktionalitäten den darüberliegenden Schichten zur Verfügung stellt. Alle Schichten sind in XML notiert.

Die Basis des Stacks (*Kern-Schichten* bzw. core layers) bilden die anerkannten Technologien HTTP, XML, SOAP und WSDL – diese sind zum jetzigen Zeitpunkt unter anderem auch Bestandteil der großen Web Service APIs wie das *Java Web Service Developer Pack* von Sun [SUN], *WebSphere Software Development Kit* von IBM [IBM] und *Visual Studio .NET Framework* von Microsoft [RST04].

Die so genannten *hervortretenden Schichten* (emerging layers) erlauben es weitere Eigenschaften eines Web Services zu spezifizieren. Für die Funktionalitäten dieser Schichten

gibt es momentan eine große Zahl von Vorschlägen zur Spezifikation und Implementation. Abbildung 1 stellt einen der derzeitigen, prinzipiellen Ansätze der Gesamt-Architektur dar.

UDDI liegt in diesem Ansatz quer zur Stack-Architektur, da es eher einen (essentiellen) Dienst im Web Service Ansatz darstellt, als eine Technologie zur Implementierung eines Web Services.

Zur obersten Schicht des Stacks in Abbildung 1 gehört die Business Process Execution Language for Web Service. Sie ermöglicht es Web Services abstrakt oder ausführbar zu beschreiben und bestehende Web Services zu einem neuen zu komponieren.

Gleichwohl handelt es sich bei allen Vorschlägen um keine völlig neuen Technologien. Lediglich die Idee einer modularen, standardisierten Architektur ließe sich als Novum auffassen [KL04].

1.2 BPEL4WS

Die Business Process Execution Language for Web Service wurde entworfen, um ganz konkret Geschäftsprozesse innerhalb des Web-Service-Ansatzes zu modellieren. Damit einher geht die Spezifikation ausführbarer Geschäftsprozesse als Web Service sowie die Möglichkeit, die Komposition von Web Services aus der Sicht eines einzelnen Web Services zu beschreiben.

Der elementare Aspekt der Komposition ist hier der Austausch von Nachrichten über WSDL-Schnittstellen. Der spezifizierte Web Service bildet das Bindeglied zwischen den anderen beteiligten Web Services – genannt *Partner* – und verbindet so deren Funktionalität mit der eigenen zu einem neuen Web Service. Hierfür werden Nachrichtenempfang und -versand durch Kontrollflusskonstrukte in eine entsprechende Reihenfolge gebracht. Die Definition des Kontrollflusses ist zwei anderen Sprachen entlehnt: BPEL4WS ist aus den Sprachen *Web Service Flow Language* (WSFL) von IBM [Ley01] und XLANG von Microsoft [Tha01] hervorgegangen. Dabei wurde die graphorientierte Beschreibung des Kontrollflusses aus WSFL mit der block-strukturierten Notation aus XLANG kombiniert und um ein Fehlerbehandlungsmodell erweitert.

Jedes in BPEL4WS geschriebene Programm ist ein *Prozess*, der Datenfluss und Kontrollfluss zueinander in Beziehung setzt. Aus technischer Sicht bilden die *Aktivitäten* die Grundbausteine in BPEL. Sie definieren sowohl Kontrollfluss- als auch Datenfluss-Beziehungen. Wir unterscheiden *elementare* Aktivitäten und *strukturierte* Aktivitäten.

Elementare Aktivitäten beschreiben nicht weiter (sinnvoll) teilbare Arbeitsschritte in dem modellierten Geschäftsprozess: empfangen von Nachrichten (**receive**); auf empfangene Nachrichten antworten (**reply**); asynchronen bzw. synchronen Nachrichtenaustausch mit einem anderen Web Service initiieren (**invoke**); warten (**wait**); Daten manipulieren (**assign**); Fehlerbehandlung explizit aufrufen (**throw**); die Prozessausführung sofort beenden (**terminate**) und nichts tun (**empty**).

Strukturierte Aktivitäten ordnen die Arbeitsschritte in den Kontrollfluss ein. Sie haben jeweils eine rekursive Struktur, d.h. jede strukturierte Aktivität enthält mindestens eine weitere (*innere*) Aktivität. Die strukturierten Aktivitäten eines Prozesses ergeben zusammen die Definition des Kontrollflusses für den modellierten Prozess: sequentielle Ausführung der enthaltenen Aktivitäten (**sequence**); nebenläufige Ausführung der enthaltenen Aktivi-

täten (**flow**) – die Ausführungsreihenfolge der inneren Aktivitäten kann durch *Links* näher spezifiziert werden (**link**), was einen gerichteten Graphen beschreibt; bedingte, wiederholte Ausführung (**while**); bedingte Verzweigung des Kontrollflusses (**switch**); warten auf ein Ereignis (erhalten einer Nachricht oder erfüllen einer Zeitbedingung) und verzweigen des Kontrollflusses anhand des Ereignisses (**pick**) sowie einen Teil des Prozesses hinsichtlich des Kontroll- und des Datenflusses im *Scope* kapseln (**scope**). Der Scope bietet des weiteren die Mechanismen zur *Fehlerbehandlung* (**faultHandlers**), *Kompensationsbehandlung* (**compensationHandler**), welche bereits abgeschlossene Arbeitsschritte rückgängig macht und der *Ereignisbehandlung* (**eventHandlers**), die nebenläufig auf Nachrichten und zeitbedingte Ereignisse reagiert.

Jeder Prozess definiert dem Blockkonzept von XLANG streng folgend genau eine Aktivität, die alle anderen Aktivitäten rekursiv enthält. Diese *Wurzel-Aktivität* ist immer ein Scope.

Um ein besseres Gefühl für die Sprache zu vermitteln, wollen wir kurz die (syntaktische) Struktur eines BPEL4WS-Prozesses erläutern. Diese beinhaltet eine Reihe weiterer Definitionen, auf die wir ebenfalls kurz eingehen. Jede Prozess-Definition wird mit einer Reihe von Definitionen eingeleitet: Die Schnittstellenbeschreibungen in WSDL werden typischerweise importiert, da sie zumeist als eigene Dokumente separat abrufbar sind. Gleiches gilt häufig für Typ-Definitionen. Im Beispiel in Listing 1 ist dieser Import im **process**-Element als **xmlns:lns**-Attribut notiert.

Es folgen die Definitionen, die spezifisch für diesen Prozess sind. Hierzu zählt eine Spezifikation der Eigenschaften, die ein möglicher Kommunikationspartner bzgl. einer bestimmten Interaktion erfüllen muss (**partnerLinks**), sowie globale Variablendeklarationen (**variables**). Beides stützt sich auch auf die importierten Definitionen der Typen und Schnittstellen.

Ist dieser Teil abgeschlossen, beginnt die Definition der Aktivitäten. Da die Wurzel-Aktivität stets ein Scope ist, werden dessen XML-Tags nicht notiert, sondern die innerhalb eines Scopes zulässigen Elemente direkt definiert: neben einer Aktivität sind dies die Handler. Listing 1 zeigt in verkürzter Form den Quelltext des *LoanApproval-Prozesses* aus der BPEL4WS-Spezifikation [CGK⁺03].

Eine Reihe von Aktivitäten wird nebenläufig (**flow**) ausgeführt. Links (**links**) ordnen einige der Aktivitäten. Zunächst wird eine Nachricht empfangen (**receive**). Eine andere Nachricht wird erst versendet (**invoke**), nachdem die erste Nachricht vollständig empfangen wurde. Diese Reihenfolge ist durch den Link "**receive-to-assess**" und die Elemente **source** und **targets** definiert. Am Ende des Prozesses (die Ordnung gebenden Links wurden nicht mit notiert), antwortet (**reply**) dieser Prozess an den Partner (**partnerLink="customer"**), der die erste Nachricht versandt hat.

Listing 1: Verkürzter Quelltext des LoanApproval-Prozesses

```
<process name="loanApprovalProcess"
  xmlns:lns="http://loans.org/wsd1/loan-approval" ...>
  <partnerLinks>
    <partnerLink name="customer"
      partnerLinkType="lns:loanPartnerLinkType"
      myRole="loanService"/>
```

```

    ...
    </partnerLinks>

<variables>
  <variable name="request" messageType="lns:creditInformationMessage"/>
  <variable name="error" messageType="lns:errorMessage"/>
  ...
</variables>

<faultHandlers>
  <catch faultName="lns:loanProcessFault"
    faultVariable="error" faultMessageType="lns:errorMessage">
    <reply partnerLink="customer" ... />
  </catch>
</faultHandlers>

<!-- activities start here -->
<flow>
  <links>
    <link name="receive-to-assess"/>
    ...
  </links>

  <receive partnerLink="customer" variable="request" createInstance="yes" ...>
    <sources>
      <source linkName="receive-to-assess">
        ...
      </source>
      ...
    </sources>
  </receive>
  <invoke ...>
    <targets>
      <target linkName="receive-to-assess"/>
    </targets>
    ...
  </invoke>
  <assign>
    ...
  </assign>
  ...
  <reply partnerLink="customer" ...>
    ...
  </reply>
</flow>
</process>

```

Für alle hier beschriebenen Konstrukte der Sprache sowie deren Anbindung an die unterliegenden Schichten des Web-Service-Technology-Stacks existiert bislang lediglich eine

XML-basierte Syntax und eine in englischer Prosa verfasste informale Semantik [CGK⁺03]. Aufgrund der Entwicklungsgeschichte von BPEL4WS und der Vielzahl mächtiger Konstrukte [WADH02], die in den normalen Kontrollfluss eingreifen, stellt sich die Frage nach der Konsistenz und Korrektheit der Semantik und schließlich nach der industriellen Einsetzbarkeit der Sprache.

Die Frage der Konsistenz und Korrektheit kann nicht anhand der informalen Dokumente beantwortet werden. Vielmehr ist es notwendig ein formales Modell der Sprache – also eine formale Semantik – zu erstellen, um die Frage beantworten zu können. Der Weg, wie dieses formale Modell generiert werden kann, soll in dieser Arbeit beschrieben werden.

1.3 Ansatz für eine formale Semantik mit Abstract State Machines

Ziel einer Formalisierung einer Programmiersprache ist stets, die Verhalten und die Eigenschaften der mit dieser Sprache geschriebenen Programme formal zu fassen. Jedes gemäß der Sprache korrekte Programm hat durch die Regeln der Sprache eine Semantik, die wir meist als Mengen von Abläufe verstehen, wobei ein Ablauf eine Folge von Zuständen des Programms ist. In unserem Fall sind die Regeln der Sprache vollständig informal gegeben. Folglich sind auch die Abläufe nur durch eine informale Beschreibung gegeben.

Im formalen Modell gibt es ebenfalls Abläufe. Die Abläufe des formalen Modells müssen den Abläufen der informalen Beschreibung (gemäß der gewählten Formalisierung) entsprechen: Jeder relevante Aspekt eines Ablaufs des informalen Systems wird im formalen Ablauf erfasst und jedes im formalen Ablauf erkennbare Verhalten findet sich auch in der informalen Beschreibung. Die formalisierten Abläufe folgen dabei strengen Regeln, eben jenen, die das formale Modell vorgibt. Zu jedem Zustand lässt sich (unter Auflösung von Nichtdeterminismus) genau ein Nachfolgezustand bestimmen.

Die Aufgabe der Formalisierung ist, die Abläufe formal zu erfassen. Dies lässt sich auf mehrere Arten erreichen. Bevor wir unseren Ansatz hier einordnen können, müssen wir aber erst noch einige Vorarbeit leisten.

1.3.1 Grundsätzliche Überlegungen zur Struktur von BPEL4WS

Unser Ansatz einer formalen Semantik für BPEL4WS soll den Aufbau der Sprache selbst widerspiegeln. Die im vorigen Abschnitt vorgestellten Aktivitäten stellen *Konzepte* zur Kommunikation, Steuerung des Kontrollflusses und Manipulation von Daten dar. Jedem Konzept wohnt eine gewisse Struktur und ein prinzipielles Verhalten inne.

Die BPEL4WS-Spezifikation grenzt die Konzepte hinsichtlich ihrer Funktionalität klar voneinander ab. Die Struktur und das Verhalten eines Konzepts einer Aktivität (*Aktivitätskonzept*) ist unabhängig von den anderen Aktivitätskonzepten definiert. Diese Unterteilung wollen wir in der formalen Semantik ebenfalls beibehalten. Offensichtlich können die Konzepte aber keine isolierten Einheiten beschreiben, schließlich besteht ein BPEL4WS-Prozess aus mehreren Aktivitäten, deren jeweilige Ausführung abhängig von der Ausführung der anderen Aktivitäten ist (z.B. für eine sequentielle Abfolge).

Das Zusammenspiel zwischen den Aktivitäten ist aus konzeptueller Perspektive einheitlich, in der Spezifikation aber nur implizit definiert. Wir werden die Koordinierung des

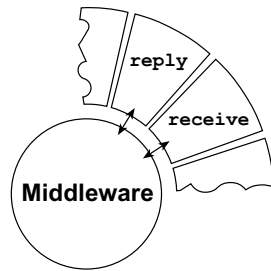


Abbildung 2: Konzeptuelle Struktur von BPEL4WS

Zusammenspiels als ein eigenes Konzept von BPEL4WS auffassen, welches wir als unterliegendes, verbindendes *Middlewarekonzept* bezeichnen wollen (Abbildung 2).

Das Middlewarekonzept definiert letztlich eine einheitliche Schnittstelle und ein einheitliches Protokoll über das die Aktivitäten von BPEL4WS miteinander kommunizieren.

1.3.2 Grundsätzliche Überlegungen zur Semantik von BPEL4WS

In diesem Teilabschnitt stellen wir einige Überlegungen zur Formalisierung der Semantik vor, die auf der Struktur von BPEL4WS basieren. Lesern, denen BPEL4WS unbekannt ist, sei empfohlen diesen Teilabschnitt zu überspringen und am Ende hierher zurückzukehren.

Die in der BPEL4WS-Spezifikation [CGK⁺03] definierten Konzepte beschreiben alle möglichen Ausführungen der durch sie definierten Aktivitäten.

Beim Ablauf eines Programms werden jedoch nicht Konzepte angewandt und ausgeführt, sondern vielmehr *konkrete Ausprägungen* dieser. Letztere sind immer durch eine konkrete Definition eines BPEL4WS-Prozesses gegeben: BPEL4WS-Prozesse sind Programme, die Konzepte definieren die Regeln, nach denen ein Prozess abläuft.

Genau diese Konstellation wollen wir auch in der formalen Semantik wiederfinden und daher die Beschreibung auf der Ebene der Konzepte und die Beschreibung ihrer konkreten Ausprägungen unterscheiden. Das heißt, wir werden die Konzepte selbst formalisieren. Das Middlewarekonzept werden wir naheliegenderweise auf der gleichen Abstraktionsebene wie die Aktivitätskonzepte formalisieren.

Das bedeutet im Endeffekt, dass wir zu *einer* formalen Beschreibung der Konzepte gelangen, die durch geeignete Abbildungen für jeden korrekt definierten BPEL4WS-Prozess gültig ist. Was „geeignet“ bedeutet und wie wir sowohl die Konzepte, als auch ihre Ausprägungen adäquat formalisieren, ist Gegenstand dieser Arbeit.

1.3.3 Abstract-State Machines

Für eine korrekte Formalisierung müssen wir im Auge behalten, dass der Schritt von der informalen zur formalen Semantik stets in allen Einzelheiten nachvollziehbar ist. Genauer gesagt sollten wir zum einen die Semantik auf der Basis eines wohl-fundierten Formalismus

so präzise übersetzen, dass diese mit mathematischen Mitteln analysierbar ist. Zum anderen benötigen wir einen geeigneten Grad an Abstraktion, so dass wir Anforderungen der Spezifikation von Details der Implementation trennen und gleichzeitig die informale mit der formalen Semantik vergleichen können.

Egon Börger sieht im *Abstract-State-Machines-Formalismus* einen Ansatz, der obige Eigenschaften erfüllt [Bo95]. Auf den ersten Blick spezifiziert jede Abstract-State Machine (ASM) ein Pseudocode-Programm, dessen Variablen und Funktionen *induktiv definierte Terme* sind. Diese Notation hat jedoch einen strengen formalen Hintergrund: Jede ASM modelliert ein Transitionssystem; jeder *Zustand* des Transitionssystems ist eine Algebra. Jeder notierte Term wird in einem Zustand *interpretiert*. Genauer: jedes Symbol des Terms bildet die Interpretation auf eine Funktion bzw. eine Konstante in der Algebra ab. Damit lässt sich induktiv für jeden Term ein Wert im aktuellen Zustand berechnen.

Der Begriff „Wert“ ist im ASM-Formalismus jedoch völlig abstrakt. Wir sind nicht an einzelnen Werten interessiert, sondern daran, ob zwei Funktionen ein Argument auf den selben Wert abbilden. Das wiederum können wir mittels prädikatenlogischer Formeln syntaktisch auf der Ebene der Terme formalisieren. Im ASM-Formalismus reduzieren wir damit die Betrachtung der Zusammenhänge in einem Zustand auf die Frage, ob zwei Terme gleich interpretiert werden. Davon ausgehend, beschreiben wir Zustandsänderungen und damit *Schritte* syntaktisch durch Zuweisungen an Terme, formuliert in *ASM-Regeln*.

Allerdings steht hinter jedem Symbol eines Terms eine ordentliche mathematische Definition einer Funktion, welche die Zusammenhänge in einem Zustand präzise formalisiert. Jede Zuweisung ist damit eine gleichermaßen präzise Neudefinition der Funktion für ein gegebenes Argument. Durch die Verwendung interpretierter Terme setzen Abstract State Machines den Fokus der Betrachtung darauf, welche Objekte des Systems wie mit anderen zusammenhängen und wie sich diese Verhältnisse ändern.

1.3.4 Vorgehen bei der Formalisierung

Wie schon zu Beginn diesen Abschnitts angedeutet, wollen wir in der formalen Semantik die konzeptuellen Bausteine – die Aktivitätskonzepte und das Middlewarekonzept – der Sprache direkt beschreiben. Der naheliegende Ansatz ist eine unmittelbare Übersetzung der Aktivitätskonzepte und des Middlewarekonzepts in abgegrenzte Einheiten der formalen Semantik. Die Konzepte sind über ihre innere Struktur und ihrem darauf basierenden Verhalten definiert. Strukturelle Beschreibungen finden im ASM-Formalismus in Funktionen ihre Entsprechung; Verhalten, also die Beschreibung von Zustandsänderungen, wird nur durch ASM-Regeln erfasst. Die Verbindung zwischen ASM-Regeln und den Funktionen bilden die über den zugehörigen Funktionssymbolen konstruierbaren Terme.

Aus dem Ansatz einer jeweils in sich abgeschlossenen Übersetzung der Konzepte heraus, dürfen die ASM-Regeln, die das Verhalten eines Aktivitätskonzepts beschreiben auch nur die Funktionssymbole beinhalten, die wir seiner strukturellen Beschreibung oder der strukturellen Beschreibung des Middlewarekonzepts zuordnen. Da sich die Funktionen stets auf die Beschreibung eines Konzepts beschränken, nennen wir sie *konzept-beschreibende* Funktionen.

Von einem makroskopischen Standpunkt aus bildet so die formale Beschreibung eines Konzepts ein *ASM-Modul* mit wohl-definierten Ein- und Ausgabeschnittstellen. Die ASM-Module

aller Konzepte in BPEL4WS ergeben die vollständige *ASM-Definition*, die vollständige konzeptuelle Beschreibung der Sprache. Abbildung 3 stellt diese Idee dar.

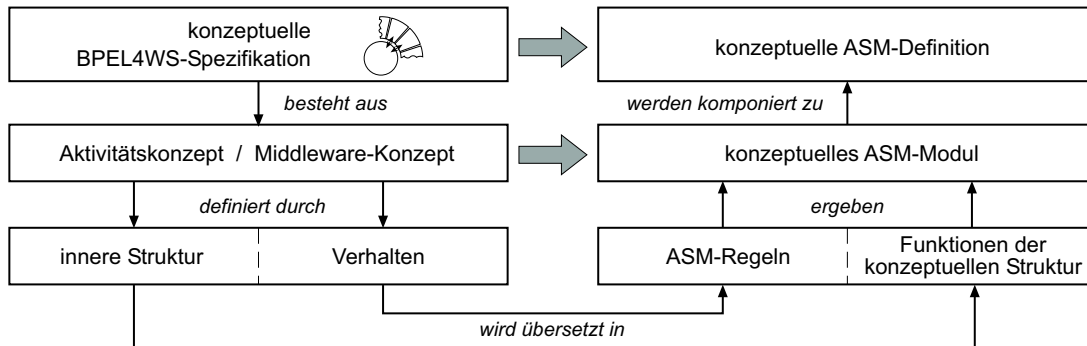


Abbildung 3: Formalisierung der BPEL4WS-Konzepte

Der zweite Aspekt der Sprache, die Abbildung der Konzepte auf konkret definierte BPEL4WS-Prozesse folgt einem anderen Ansatz. Da eine Aktivität in einem BPEL4WS-Prozess lediglich eine spezielle Ausprägung ihres Aktivitätskonzepts ist, die kein neues Verhalten generiert, sondern die möglichen Verhalten laut Konzept für den jeweiligen Spezialfall der Prozessdefinition einschränkt, können wir die spezielle Ausprägung vollständig durch strukturelle Definitionen formalisieren.

Hierfür verwenden wir folglich ebenfalls Funktionen. Zur Unterscheidung nennen wir diese Funktionen *prozess-beschreibend*. Bezüglich der Gesamtheit aller BPEL4WS-Prozesse ist die Definition der prozess-beschreibenden Funktionen nicht einheitlich im Gegensatz zu den strukturellen Funktionen der Konzepte, sondern bezieht sich immer auf eine konkrete *BPEL4WS-Prozess-Definition*. In BPEL4WS wird dieser Aspekt durch die XML-basierte Grammatik erfasst. Wir nehmen uns die Ableitungsregeln, mit denen wir eine Prozess-Definition schrittweise aufbauen können, als Vorbild für ein Verfahren mittels dessen wir prozess-beschreibende Funktionen so definieren, dass sie gegebenen gültigen BPEL4WS-Prozessen entsprechen.

Wir können damit jeden laut BPEL4WS-Spezifikation gültigen BPEL4WS-Prozess auch in unserem ASM-Modell ausdrücken. Dieses Verfahren zur Erzeugung von BPEL4WS-Prozess-Definitionen bildet dann zusammen mit der Semantik der Konzepte die vollständige Formalisierung der BPEL4WS-Spezifikation.

Unser gesamter Ansatz ist in Abb. 4 noch einmal veranschaulicht.

Ob dieser Ansatz zum Erfolg führt, wollen wir in dieser Arbeit an Beispielen überprüfen.

Unser Vorgehen ist wie folgt gegliedert. In Abschnitt 2 geben wir eine informale Einführung in die Syntax und Semantik von Abstract State Machines. Von besonderem Interesse sind hierbei die asynchronen verteilten ASMs. Wir beleuchten dabei auch die Art und Weise, wie ASMs Transitionssysteme implizit beschreiben. Im darauf folgenden Abschnitt 3 charakterisieren wir die Übersetzung der der Semantik zugrundeliegenden Strukturen. Anschließend wird in Abschnitt 4 gezeigt, wie die informale Semantik der Aktivitätskonzepte in ASM-Regeln übersetzt wird. Dabei werden wir auch das verbindende Middlewarekonzept vorstellen. In diesem Abschnitt erläutern wir auch, wie wir die Abläufe eines Prozesses for-

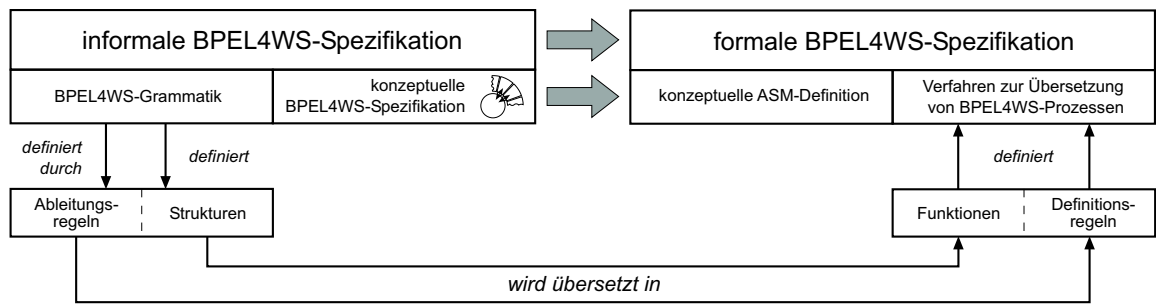


Abbildung 4: Charakterisierung der Übersetzung der BPEL4WS-Spezifikation in ASMs

mal fassen. Abschließend geben wir in Abschnitt 5 eine Zusammenfassung des Ansatzes und eine Betrachtung bezüglich der Eignung von Abstract State Machines zur Formalisierung der Semantik sowie einen Ausblick auf die weitere Arbeit zum Thema.

2 Abstract-State Machines

Dieser Abschnitt gibt eine kurze Einführung in Abstract-State Machines. Leser, die mit diesem Formalismus vertraut sind, sollten zumindest die Abschnitte 2.3 und 2.4 lesen.

Abstract-State Machines (ASMs) wurden 1985 von Yuri Gurevich als ein „Berechnungsmodell [eingeführt], das 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 (insbesondere die Java Virtual Machine) 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 ASMs bilden die einfachste Klasse von Abstract-State Machines. Auf ihnen bauen die restlichen Klassen auf. Die letzte Klasse werden wir für die Semantikdefinition 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.

2.1 Sequentielle Abstract-State Machines

Eine Abstract-State Machine modelliert ein Transitionssystem, dessen Zustände mathematische Strukturen, genauer Σ -Algebren, sind. Die Zustandsübergänge beschreiben 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 eine Menge U und Funktionen $\varphi : U^n \rightarrow U$ 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 mit gleicher Stelligkeit, die wir mit dem selben Symbol f darstellen, verschieden von der Funktionsdefinition in S sein. Um beide Funktionen dennoch zu unterscheiden, 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* . f_S und f_T sind nun für uns unterscheidbar.

In einem Zustand S können wir anhand der Funktionen Beziehungen ausrechnen. Verketteten wir Funktionen, so ergeben sich komplexe Zusammenhänge.

Syntaktisch drücken wir dies mit (wie üblich) *induktiv definierte Termen* aus den Symbolen von Σ aus. Dabei sind 0-stellige Funktionssymbole *Konstantensymbole*.

Jedem Term t können wir in einer Algebra S mit deren Träger U einen Wert $v \in U$

zuordnen, der sich aus der Interpretation seiner Funktionssymbole und der Definition der zugehörigen Funktionen ergibt. Wir nennen daher auch t_S die *Interpretation von t in S* .

Damit können wir nun Terme in einem Zustand vergleichen. Die syntaktische Gleichung

$$f(t_1, \dots, t_n) = t_0 \quad (1)$$

gilt im Zustand S genau dann, wenn die Funktion f_S , angewandt auf die Interpretation der Terme $t_i, i \in [n]^1$ auf die Interpretation von t_0 abbildet; wenn also gilt:

$$f_S(t_{1S}, \dots, t_{nS}) = t_{0S}. \quad (2)$$

Ausgehend von einem Zustand T , in dem zwar wie in S $t_{iS} = v_i = t_{iT}, i \in [n]_0^2$, aber nicht Gleichung (1) gilt, wollen wir erreichen, dass die Gleichung (1) in S 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, \dots, v_n) einfach neu – und zwar syntaktisch notiert als Σ -Assignment, welches sich von Gleichung (1) durch den Zuweisungsoperator $:=$ unterscheidet,

$$f(t_1, \dots, t_n) := t_0. \quad (3)$$

Die semantische Entsprechung ist die Funktionsdefinition in S :

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

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

Mit den obigen Annahmen ist (1) 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 verschiedenen Werten definiert wird: $(f, (t_{1T}, \dots, t_{nT}), t_{0T})$ und $(f, (t_{1T}, \dots, t_{0T}), t_{0T}^*)$ mit $t_{0T} \neq t_{0T}^*$.

Solche Σ -Updates nennen wir *inkonsistent*. Sind mehrere Σ -Updates an einem Übergang beteiligt, so werden alle Funktionen auf einmal neu definiert. Insbesondere hat für

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

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

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

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

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

² $[n]_0 =_{def} \{0, \dots, n\}$

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 interpretieren wir diesen wie das Symbol *undef*:

$$t_S = \text{undef}_S.$$

Für alle anderen der eben genannten Symbole nehmen wir die übliche Interpretation an.

Beobachtung: 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 über Σ 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 sind

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

ebenfalls Regeln.

Existiert im Falle von R_{forall} kein x , das 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 2.4 noch einige Konventionen 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: z.B. **let message = new(BPELMsg) in R**. Diese Notation orientiert sich an der von Börger & Stärck [BS03].

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 $S_0 \in \text{init}$ ein Anfangszustand ist und jeder Übergang von S_i zu S_{i+1} durch den Schritt gegeben ist, der durch Anwenden von R auf S_i definiert ist.

2.2 Asynchrone Verteilte Abstract-State Machines

Eine verteilte ASM besteht prinzipiell aus mehreren sequentiellen 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 der Name des Agenten und R ist die Regel der zugehörigen sequentiellen ASM. Mehrere sequentielle ASM können das selbe Element oder den selben Term verwenden, sowohl lesend, als auch schreibend.

Als Zustand des Gesamtsystems dient *eine* Σ -Algebra. Des Weiteren ist die Regel eines Agenten nicht exklusiv ihm zugeordnet. 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 Mechanismus 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 über den Schritten. Jeder Ablauf des Gesamtsystems hat ein endliches Anfangsstück. Jeder Ablauf eines 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.

Interessierte Leser, die eine formale Definition asynchroner, verteilter ASMs ähnlich der sequentieller ASMs erwartet haben, müssen wir enttäuschen. Bislang gibt es in der Forschung keine formale Fundierung für diese Klasse von ASMs.

2.3 ASM beschreiben Transitionssysteme

Wir haben diesen Abschnitt damit begonnen ASMs als Beschreibung von Transitionssystemen einzuführen und kehren zu diesem Aspekt noch einmal zurück. Jede sequentielle ASM beschreibt eine Klasse von Transitionssystemen (init, S, F) , deren jeweilige Zustände S Σ -

Algebren mit der Signatur Σ sind. Die ASM-Regel beschreibt die Zustandsübergangsrelation $F \subseteq S \times S$ [Rei03].

Aus der Art und Weise, wie verteilte ASMs charakterisiert sind, können wir davon ausgehen, dass es eine vergleichbare Betrachtung auch für diese Klasse gibt.

2.3.1 Anfangszustände und die Zustandsübergangsrelation

Die Zustände des Transitionssystems sind durch die Beschreibung der ASM mittels Signatur und Regel lediglich implizit gegeben. Allerdings kommt der Wahl der Anfangszustände eines Transitionssystems eine entscheidende Bedeutung zu. Durch diese Wahl und die gegebene Definition der Zustandsübergangsrelation legen wir implizit die erreichbaren Zustände fest. Diese zu betrachten ist letztlich auch die Aufgabe einer Modellierung mittels Transitionssystem, nur gestattet dies der ASM-Formalismus so nicht.

Wir müssen hier die Perspektive wechseln. Aus einer intuitiven Anschauung heraus können wir auch sagen, dass wir die Übergangsrelation F auf die erreichbaren Zustände eingrenzen, also nur den jeweiligen Fall des Übergangs betrachten, der dem Anfangszustand genügt. Mit anderen Fällen kommt die Übergangsrelation gewissermaßen nie in Berührung.

Durch die Wahl des Anfangszustandes lassen wir nur noch bestimmte Übergänge zu. Diese Analogie zwischen BPEL4WS-Konzepten und ASM-Regeln bzw. BPEL4WS-Prozess-Definition und Anfangszustand werden wir in Kapitel 3 und 4 ausnutzen.

2.3.2 Abstraktion von Schritten und Abläufen

An dieser Stelle können wir einen ersten Beitrag zur Antwort auf die noch offene Frage der Einleitung (Abschnitt 1.3) geben, auf welche Art wir die Abläufe eines Prozesses formal fassen wollen. Diese erwächst aus der Wahl des Formalismus selbst.

Besonders interessant am ASM-Ansatz ist die Art und Weise, wie 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 oder nicht mehr gilt.

Abläufe als Folgen von Zuständen finden wir als Weg in einem Transitionssystem, der in dessen Anfangszustand beginnt, wieder. Die einzelnen Schritte des Weges sind (unter Auflösung von Nichtdeterminismus) durch die Zustandsübergangsrelation eindeutig gegeben. Wie gerade beschrieben, definieren ASM-Regeln implizit eine Zustandsübergangsrelation. Allerdings eine abstrakte. Ein und dieselbe ASM-Regel, mit einem **if**, **choose** bzw. **forall**-Operator, definiert, abhängig von der Interpretation der Funktionssymbole in ihrer Bedingung, unendlich viele konkrete Übergänge von einem Zustand zu einem anderen.

Für eine Σ -Algebra ist gemäß der Semantik von Σ -Updates ihr Nachfolgezustand anhand der ASM-Regeln eindeutig bestimmt. Aufgrund der syntaktischen Abstraktion haben jedoch alle Übergänge etwas gemein. Die über ein Σ -Assignment miteinander verknüpften Terme werden in den jeweiligen Nachfolgezuständen gleich interpretiert, wenn die Bedingungen der Operatoren im Vorgängerzustand als wahr interpretiert wurden.

Aus diesem Aspekt heraus lassen sich Schritte und damit die Abläufe von Systemen über die jeweilige Gleichheit von Termen auf syntaktischer Ebene betrachten. Ein Zustand

dieses „abstrakten Ablaufs“ ist durch Mengen von gleich interpretierten Termen gegeben. Ein Übergang ordnet diese Mengen neu und ist *rein syntaktisch* gegeben: Σ -Assignments verschieben Terme von einer Menge in eine andere oder vereinigen sie, **new**-Operatoren erzeugen neue Mengen.

Auf diesem Abstraktionsniveau gibt nicht länger der ASM-Formalismus einen Ablauf vor, sondern die in ihm definierte Regel. Welche Auswirkungen das auf die Formalisierung von BPEL4WS hat, werden wir in Abschnitt 4 darlegen, wenn wir uns mit den Zustandsübergängen eines BPEL4WS-Prozesses befassen.

2.4 ASM 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.

Zu jedem Funktionssymbol geben wir gleichzeitig eine mathematische Signatur an, die Definitions- und Wertebereich der möglichen, sinnvollen Interpretationen festlegen. Beide Bereiche sind selbstverständlich Teilmengen des Trägers U . Durch diese zusätzliche Annotation der Funktionssymbole erzeugen wir Dank selbsterklärender Symbole ein intuitives Verständnis für die verwendeten Funktionssymbole und -definitionen; so ist beispielsweise aus der Signatur $myStatic : Agent \rightarrow Activity$ bereits ersichtlich, dass jede Interpretation einem Agenten „seine“ Aktivität zuordnet. Ferner unterwerfen wir so die Konstruktion von Termen auf intuitiver Ebene der natürlichen Regel, dass Bild- und Urbildbereich der zugehörigen Funktionen entsprechend zueinander passen müssen.

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.

Ein Beispiel wäre eine Zufallsfunktion $random : \mathbf{N} \rightarrow \mathbf{N}$, für die wir annehmen, dass sie uns eine Zufallszahl $0 \leq random(k) \leq k$ liefert. Eine genau Definition dieser Funktion (so sie denn überhaupt existiert) wäre jenseits einer abstrakten Modellierung.

Wir fassen gemäß der informalen Semantik zusammengehörige 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 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*.
- Funktionssymbole 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

$(pl \in ProcessInstance, var \in Variable, msg \in BPELMsg) \equiv$

```

if var  $\neq$  undef then
  forall part  $\in$  msgTypeParts(messageType(var)) do
    msgPartValue(msg, part) := variablePartValue(pl, var, part)
    messageMsgType(msg) := variableMsgType(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 einfache Textersetzung handelt. Eine rekursive Regeldefinition würde daher nie terminieren und darf somit nicht verwendet werden!

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 der linke Ausdruck durch die rechte Regelkomposition ersetzt:

<pre> select x \in Set where $\varphi(x)$ in P ifnone Q </pre>	<pre> if $\exists x \in Set : \varphi$ then choose x \in Set where $\varphi(x)$ in P else Q </pre>
--	---

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

2.5 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.

Die Entwicklung des ASM-Formalismus wurde mit dem Ziel vorangetrieben, die Lücke zwischen Berechnungsmodellen und Spezifikationsmethoden zu schließen [Gur95]. So hat Y. Gurevich für sequentielle und parallele Algorithmen bewiesen, dass sequentielle ASMs [Gur00] und parallele entsprechend [BG03], die jeweils zugehörige 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 verschiedener 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 weiteren Ansatz wird ein Theorem-Beweiser entwickelt, der spezifizierte Eigenschaften anhand der ASM-Definition beweist. Wir können auch auf einige Implementierungen 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 einen Gesamtüberblick zum Thema Abstract-State Machines sei noch einmal die ASM-Website [ASMa] empfohlen.

3 Übersetzung von Strukturen

Wir haben in Abschnitt 1.3 die Struktur der Sprache BPEL4WS erläutert und in Abschnitt 2.3 eine Analogie zwischen dieser Struktur und der Beschreibung von Transitionssystemen mit ASMs aufgezeigt. Beides werden wir im Folgenden verwenden und einen ersten Blick auf den Aufbau der Formalisierung werfen sowie die Übersetzung der BPEL4WS-Prozess-Definitionen charakterisieren.

Die Aktivitätskonzepte in BPEL4WS basieren auf gewissen lokalen, strukturellen Beschreibungen wie „A reply activity may specify a variable... that contains the message data to be sent in reply.“ [CGK⁺03, 11.4], die für eine konkrete Ausprägung des Konzepts auch eine konkrete Definition erhalten, wie „Reply 'SendConfirmation' antwortet mit Nachrichten-Daten aus der Variable 'finalConfirmation'“.

Auf der Ebene der Konzepte finden wir also Klassen von Funktionen, die wir mittels mathematischer Signatur und Funktionssymbolen wie $replyVariable : Reply \rightarrow Variable$ zur Beschreibung der Struktur des Konzepts formalisieren. Das durch das Konzept gegebene Verhalten wollen wir als ASM-Regel formalisieren. Folglich finden die zugehörigen Funktionssymbole ihren Platz in den ASM-Regeln, die die Zustandsübergangsrelation für dieses Konzept beschreiben.

Der Anfangszustand grenzt, wie in Abschnitt 2.3 beschrieben, die Übergangsrelation ein. Gleiches finden wir in BPEL4WS anhand der BPEL4WS-Prozess-Definition, die uns die möglichen Verhalten der Konzepte für ihre jeweiligen Ausprägungen entsprechend eingrenzen.

Daher formalisieren wir einen konkreten Prozess als Transitionssystem dadurch, dass wir die gegebene allgemeine Übergangsrelation mittels konkreter Funktionsdefinitionen im Anfangszustand des Transitionssystems einschränken. Dass sich die konkreten Funktionsdefinitionen aus der gegebenen BPEL4WS-Prozess-Definition ableiten, ist im Sinne einer korrekten Formalisierung zwingend. Haben wir die Übersetzung eines BPEL4WS-Prozess-Definition in einen entsprechenden Anfangszustand für *alle* möglichen Prozesse sichergestellt, dann können wir guten Gewissens die abstrakteren Funktionsklassen bei der Notation der ASM-Regeln verwenden. Schließlich steht dann jedes Funktionssymbol bezüglich eines (Anfangs-)Zustands für eine korrekte, formale, strukturelle Beschreibung einer strukturellen Eigenschaft eines Konzepts.

Im folgenden Abschnitt beschreiben wir eine Verfahren, mittels dessen wir aus jeder BPEL4WS-Prozess-Definition einen Anfangszustand konstruieren können, der die strukturellen Eigenschaften des Prozesses beschreibt.

Bevor wir uns das Verfahren näher ansehen, sollten wir uns noch überlegen welcher Natur die betrachteten Strukturen sind. Ihnen gemein ist, dass sie als Prozess-Definition in jeder möglichen Ausführung des Prozesses unveränderlich sind. Diese Eigenschaft haben alle Typ-Definition, wie beispielsweise „MessageType X ist ein geordnetes k -Tupel der MessageTypParts Y_1, \dots, Y_k “, Konfigurationen konkreter Ausprägungen eines Aktivitätskonzepts wie „Die Join-Condition von Aktivität A ist 'P and Q'“ und Beschreibungen der Struktur wie „Die Kindaktivität von 'flow2' ist 'receive7'“.

Im Gegensatz dazu ist der Wert einer Variablen, aus der ein Reply seine Antwortnachrichten generiert wohl eine strukturelle Beschreibung, allerdings eine des ausgeführten Prozesses und damit in zwei verschiedenen Ausführungen nicht notwendigerweise die gleiche.

In der ASM-Formalisierung bedeutet dies letztlich, dass alle Funktionen, die wir aus der Prozess-Definition ableiten, statisch sind – die Definition wird durch keinen Schritt verändert. Daher nennen wir sowohl Funktionen, als auch die durch sie formalisierten Strukturen *statisch*. Alle anderen Funktionen sind damit Gegenstand der Veränderung in einem Zustandsübergang, folglich nennen wir sie *dynamisch* und so wollen wir auch die Strukturen nennen, die die Funktionen formalisieren.

Ferner beschreibt eine BPEL4WS-Prozess-Definition zwangsläufig ausschließlich die Struktur des Prozesses und nicht die innere Struktur der Konzepte – die Funktionen, mit denen wir es zu tun haben, sind also allesamt prozess-beschreibend (s. 1.3.4).

3.1 Statische, prozess-beschreibende Strukturen

Was wir in der Einleitung in Abschnitt 1.3 und einführend in diesem Abschnitt versprochen haben, wollen wir nun auch erfüllen. Das Verfahren, mit dessen Hilfe wir zu jeder BPEL4WS-Prozess-Definition eine algebraische Spezifikation eines adäquaten Anfangszustands konstruieren können, orientiert sich an dem Mechanismus, der syntaktisch korrekte BPEL4WS-Prozess-Definitionen erzeugt: der XML-basierten Grammatik.

Für die verschiedenen Konzepte und verwendeten Datenstrukturen definiert die BPEL4WS-Spezifikation jeweils eigene XML-Tags. Jedem Tag sind in einer EBNF-ähnlichen Notation die möglichen Attribute sowie die erlaubten Kind-Tags zugeordnet. Daraus können wir zum einen die Terminal-Symbole der BPEL4WS-Grammatik ablesen, zum anderen sind so auch die (kontext-sensitiven) Ableitungsregeln implizit definiert.

Unser Verfahren beruht auf diesen beiden Aspekten. Anhand der Ableitungsregeln identifizieren wir (Teil-)Universen unseres Zustands sowie Funktionsklassen, die wir mittels mathematischer Signaturen formalisieren. Aus der Ableitung einer Prozessdefinition konstruieren wir uns schrittweise von den Terminalen aufwärts eine axiomatische Charakterisierung der Funktionsdefinitionen, die den definierten Signaturen genügt.

Eine Σ -Algebra S kommt dann als Anfangszustand für eine Ausführung des gegebenen BPEL4WS-Prozesses anhand der ASM-Semantik in Frage, wenn ihre Signatur die Funktionssymbole BPEL4WS-Prozess-Definitionen enthält und die Funktionen von S die axiomatische Charakterisierung erfüllen.

Wir geben nun anhand von Beispielen zwei Typen von Übersetzungsregeln an. Zunächst übersetzen wir die Ableitungsregeln der Grammatik selbst in Universen und Funktionsklassen. Anschließend geben wir schrittweise für eine Ableitung einer konkreten BPEL4WS-Prozess-Definition Forderungen an konkrete Zustände an – als prädikatenlogische, axiomatische Charakterisierung des Anfangszustands. Für diese Regeln unterscheiden wir zwischen den drei wesentlichen Bausteinen XML-basierter Grammatiken – den XML-Elementen, der Baum-Struktur und Attributen von Elementen.

Jedes *XML-Element* besitzt stets ein typgebendes *XML-Tag*. Das Tag bezeichnet in der XML-Grammatik *alle* XML-Elemente dieses Typs. Daraus leiten wir unsere erste Regel ab: Zu jedem Tag der BPEL4WS-Grammatik definieren wir ein zugehöriges unendliches Teiluniversum des Trägers. Alle Teiluniversen sind paarweise disjunkt.

Das XML-Element selbst (in einer konkreten BPEL4WS-Prozess-Definition) definiert die Existenz eines Objekts vom Typ seines Tags.

In jedem Anfangszustand, der der Prozess-Definition genügt, existiert für jedes XML-Element ein eindeutiges Element aus dem zum Tag gehörenden Teiluniversum.

Wir achten bei der Namensgebung der Symbole 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 2 gegeben.

Listing 2: Grammatik für Message Type und Message Type Part

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

Tag	Universum
<code><message... /></code>	Universe: <i>MessageType</i>
<code><part... /></code>	Universe: <i>MessageTypePart</i>

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 3.

Listing 3: 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>
```

Aus dieser Definition leiten wir nun eine prädikatenlogische, axiomatische Charakterisierung ab, die von allen in Frage kommenden Anfangszuständen erfüllt sein muss. Uns ist die Existenz zweier verschiedener Message Types und die Existenz von insgesamt vier paarweise verschiedenen Message Type Parts relevant. Somit muss in jedem geeigneten Anfangszustand

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

gelten³.

³Wir verwenden hierbei $[n]$ als Abkürzung für die Menge $\{1, \dots, n\}$.

Kommt eine weitere Message Type Definition in der originalen Prozess-Definition hinzu, so müssen wir auch weiterhin sicherstellen, dass alle zu übersetzenden Elemente paarweise verschieden sind.

Die Charakterisierung verwendet ausschließlich die Symbole der Signatur, die über Teilmengen des Trägers interpretiert werden, sowie gebundene prädikatenlogische Variablen, die keine Symbole der Signatur sind. Die Formel wird entsprechend einer gegebenen Σ -Algebra interpretiert und ist entweder wahr oder falsch.

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
<code><fault name=".." ... /></code>	Universe: <i>FaultName</i>

Mit dieser Regel ist die Spezifikation des Anfangszustandes analog zu der für die Message Types gebildet.

Jedes durch diese Formalisierung erfasste Element der BPEL4WS-Prozess-Definition bildet für uns einen Baustein der strukturellen Beschreibung des Prozesses und wir nennen es daher im Folgenden *Prozess-Struktur-Element*.

Die *Baum-Struktur* des XML-Dokumentes impliziert Abhängigkeiten zwischen den Vater- und Kind-Elementen. Diese Abhängigkeiten lassen sich durch Funktionen darstellen. Die Grammatik definiert in einer implizit gegebenen Regel die zulässigen Vater-Kind-Beziehungen anhand von Tags auf einer abstrakten Ebene. Aus jeder dieser Regeln leiten wir eine Klasse von Funktionen ab, die wir mit einer mathematischen Signatur *beziehungsName : VaterUniversum \rightarrow KindUniversum* notieren.

Wird diese Grammatik-Regel bei der Bildung einer Ableitung angewandt, so definiert sie gleichzeitig XML-Elemente *und* ihre Anordnung im Baum. Verfolgen wir die Ableitung von den Terminalen aufwärts, generiert die Anwendung dieser Regel eine Anforderung an den Anfangszustand.

Für die Elemente des Trägers, die den beteiligen Prozess-Struktur-Elementen entsprechen, bildet die zur angewandten Regel gehörende Funktion jedes zulässigen Anfangszustands vom Vater-Element auf die Kind-Elemente ab.

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.

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

Tags	Funktionen
<code><message ...></code>	<i>messageTypeParts</i>
<code><part .../>+</code>	
<code></message></code>	

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

$$\begin{aligned} & \exists mt_1, mt_2 \in Message\ Type \ \exists mp_1, \dots, mp_4 \in Message\ Type\ Part : \\ & mt_1 \neq mt_2 \wedge \bigwedge_{i,j \in [4], i \neq j} mp_i \neq mp_j \\ & \wedge message\ Type\ Parts(mt_1) = \{mp_1, mp_2\} \\ & \wedge message\ Type\ Parts(mt_2) = \{mp_3, mp_4\} \end{aligned}$$

Die *Attribute der Elemente* erzeugen Abhängigkeiten außerhalb der Baumstruktur, lassen sich jedoch genauso direkt übersetzen. Für jedes Attribut gibt die XML-Grammatik implizit eine Regel an, die die möglichen Beziehung zwischen besitzendem Element und den Elementen, auf die das Attribut verweisen kann, beschreibt – formale wiederum eine Klasse von Funktionen.

Analog zur Baumstruktur generiert die Anwendung der Regel eine Anforderung an den Anfangszustand, die die zulässigen Funktionsdefinitionen entsprechend der verknüpften Prozess-Struktur-Elemente einschränkt.

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 (**Universe:** *XMLSchemaType*) oder elementarer Typ (**Universe:** *ElementaryType*) – zugeordnet. Listing 4 zeigt die Syntax dafür.

Listing 4: Grammatik für Variablendefinition

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

Es ergibt sich folgende Funktion und die entsprechende Übersetzungsvorschrift.

$variableMsgType : Variable \rightarrow Message\ Type \cup XML\ Schema\ Type \cup Elementary\ Type$

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

In einigen Fällen ist der Zusammenhang 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 5: Beispiel für Variablendeklaration

```
<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 Types 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 den Anforderungen an die paarweise Verschiedenheit genügen.

Jede Σ -Algebra mit passender Signatur, die ein Modell für die obige Formel ist, entspricht im formalen Modell der im Beispiel gezeigten Variablen- und Typdefinition und ist somit Kandidat für die Menge der Anfangszustände der Prozesse, die diese Definitionen verwenden.

Die Prinzipien des bis hier vorgestellten Verfahrens lassen sich auch auf andere XML-basierte Grammatiken übertragen; jede Übersetzung ist jedoch immer von der Semantik der Strukturen abhängig, welche sich nicht verallgemeinern lässt.

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 Prozess-Struktur-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.

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 erstmals auch dynamische Funktionen.

Ziel von BPEL4WS ist es, ausführbare Geschäftsprozesse zu definieren. Dazu gehört auch das Lesen, Schreiben und Manipulieren komplexer Daten. Basierend auf dem Web-Service-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 interpretierter Terme, 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 Beispiel von Nachrichten (**Universe:** *BPELMsg*), die zwischen Web Services ausgetauscht werden, zeigen. (Näheres zu diesem Thema findet sich in Abschnitt 4.) Jede Nachricht hält für jeden Message Type Part ihres Message Types einen (abstrakten) Wert.

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

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 mit der Typisierung durch *msgMsgType* wird automatisch festgelegt welche innere Struktur sie hat (*messageTypeParts*, s. voriger Abschnitt) und welche Werte sie zur Laufzeit aufnehmen kann (*msgPartValue*).

Bisweilen ist es nötig, die oben beschriebene Abstraktion zu verfeinern und tiefer in die Datenstruktur und ihre Werte zu schauen. Die dafür zuständige Schicht des Web-Service-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.

3.3 Aktivitätsbaum

Wir heben an dieser Stelle eine Teilstruktur der BPEL4WS-Prozess-Definitionen explizit hervor, da sie bei der Spezifizierung der Aktivitäten noch eine Rolle spielen wird. Wie das gesamte XML-Dokument sind auch die Aktivitäten in einer Baum-Struktur definiert. Diese Baum-Struktur impliziert die Aufruf- und Ausführungsbeziehungen 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. 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, XML-Elemente strukturierter Aktivitäten haben laut BPEL4WS-Grammatik mindestens ein Kind-Element mit Aktivitäts-Tag.

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

Beide Funktionen sind den Ableitungsregel der BPEL4WS-Grammatik zugeordnet, die die Vater-Kind-Beziehungen der Aktivitätselemente beschreiben und werden bei der Bildung der Ableitung auch entsprechend der Regeln in Abschnitt 3.1 definiert.

Der *Aktivitätsbaum* ist nunmehr ein gerichteter, zusammenhängender, zyklensfreier Graph dessen Knoten die in der BPEL4WS-Prozess-Definition beschriebenen Aktivitäten sind und dessen Kanten wir aus den (entsprechend der Ableitung definierten) Funktionen *activityParent* und *activityChild* implizit ableiten können. Die Wurzel ist das Element aus *Activity*, das das `process`-Element repräsentiert.

4 Übersetzung des Verhaltens

Im vorangegangenen Abschnitt 3 haben wir gezeigt, wie wir die prozess-beschreibenden Strukturen einer BPEL4WS-Prozess-Definition in einen äquivalenten Anfangszustand übersetzen. Wir haben uns dabei auf das in Abschnitt 1.3.4 beschriebene Verfahren gestützt, die Konzepte der Sprache als solche auf gleicher Abstraktionsebene zu formalisieren und die Beschreibung eines konkreten BPEL4WS-Prozesses getrennt davon zu erfassen. Letzteres ist nun abgeschlossen und uns stehen alle Funktionssymbole der prozess-beschreibenden Funktionen zur Verfügung. Dank des Verfahrens aus dem vorigen Abschnitt können wir ab jetzt annehmen, dass uns ein Anfangszustand gegeben ist, in dem *alle* diese Funktionssymbole gemäß *irgendeiner* zugehörigen Prozess-Definition interpretiert werden können.

Wir befinden uns auf der Ebene der Konzepte. Deren Verhalten und innere Struktur wollen wir in diesem Abschnitt formalisieren. Das Verhalten der Konzepte begreifen wir als Zustandsübergänge, die wir folglich mit ASM-Regeln abstrakt modellieren. Dem Modul-Prinzip folgend definieren wir hierbei ASM-Regeln und Funktionssignaturen konzeptweise – dadurch wird die Definition einer konzept-beschreibenden Funktion einsichtig, weil wir sie sofort zur Definition des Verhaltens verwenden.

Da der Begriff des Zustandsübergangs eng mit dem des Ablaufs verbunden ist, stoßen wir an dieser Stelle erneut auf die Frage der Einleitung, auf welche Art wir die Abläufe eines Prozesses formal fassen wollen. Diese Betrachtung wird auch *Semantische Domäne* der Formalisierung genannt, da wir für einen syntaktisch gegebenen BPEL4WS-Prozess festlegen, was seine Bedeutung hinsichtlich seiner Abarbeitung ist.

4.1 Semantische Domäne eines BPEL4WS-Prozesses im ASM-Ansatz

Erinnern wir uns der in Abschnitt 2.3.2 ausgeführten Abstraktion von Abläufen von ASMs. Wir ordnen einem BPEL4WS-Prozess einen solchen abstrakten Ablauf zu.

Hierzu übersetzen wir seine Prozess-Definition in einen Anfangszustand, den wir anhand der Funktionssymbole und der bekannten Beziehungen abstrakt erfassen können. Die die Konzepte formalisierenden ASM-Regeln beschreiben abstrakt die Übergänge. Dabei gelten die Regeln für *alle* Prozesse und nicht nur den übersetzten. Die aus der axiomatischen Charakterisierung uns bekannten gültigen Gleichungen genügen, um aus den Regeln genau jene Zustandsübergänge abzulesen, die im gegebenen Prozess möglich sind.

Dieser Ansatz unterscheidet sich fundamental von dem, einen Prozess in eine formalisierte Entsprechung zu „compilieren“, deren Abläufe dann anhand dem Formalismus zugehöriger Regeln, wie beispielsweise der Schaltregel von Petrinetzen, beschrieben werden.

Wir vollziehen unseren Ansatz exemplarisch an einer strukturierten Aktivität (Sequence, 4.4) und einer elementaren Aktivität (Reply, 4.5) nach. 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.

4.2 Ausführung eines BPEL4WS-Prozesses

Was wir bis jetzt von BPEL4WS formalisiert haben, ist lediglich die statische Struktur einer Prozess-Definition. Diese selbst ist nicht ausführbar. Wir wollen nun im Folgenden beschrei-

ben, wie ein BPEL4WS-Prozess im allgemeinen und in der ASM-Semantik im Speziellen ausgeführt wird.

Die statische Prozess-Definition beschreibt u.a. Prozess-Struktur-Elemente, die in einem ausgeführten Prozess laufzeitabhängige Werte aufnehmen können, wie beispielsweise Variablen einen Wert. Weiterhin erlaubt BPEL4WS mehrere Prozesse mit der selben Prozess-Definition gleichzeitig auszuführen. Wie wir diese unterscheiden, beschreibt dieser Abschnitt.

BPEL4WS verwendet den Begriff der *Instanz eines Prozesses*, der sich vom Instanz-Begriff in der objektorientierten Programmierung unterscheidet.

Eine *BPEL4WS-Prozess-Instanz* zu bilden, bedeutet nicht, die Prozess-Definition zu kopieren, mit Variablenwerten zu belegen und dann diese Kopie auszuführen. Der Instanz-Begriff in BPEL4WS orientiert sich vielmehr an der Struktur der Sprache. Alle laufzeitabhängigen Aspekte eines Prozesses sind strukturell durch die Prozess-Definition vorgegeben: Variablen des Prozesses implizieren Variablenwerte, möglicher Kontrollfluss impliziert innere Zustände, etc.; diese laufzeitabhängigen Aspekte können wir durchaus als herkömmliche Variablen (Bezeichner für Speicherstellen) begreifen. Eine Belegung dieser Variablen ist ein Zustand des Prozesses. Eine BPEL4WS-Prozess-Instanz ist dann nichts anderes, als ein mit einem eindeutigen, internen Label (ID) versehener Prozess-Zustand. Durch die ID können wir mehrere Zustände gleichzeitig betrachten und voneinander unterscheiden.

Aufgrund der Eindeutigkeit der IDs können wir die Identität einer Instanz bereits aus seiner ID ableiten. Wir erfassen auf diese Weise *viele*, verschiedene, abgegrenzte Zustände in *einem* Modell. Die Prozess-Definition als statische Struktur existiert hingegen genau einmal [CK04].

Die eindeutigen Label der Instanzen sind ein gedankliches Konstrukt, das wir hier um der Klarheit Willen einführen. In BPEL4WS selbst ist die ID durch die Werte spezifischer, durch die Prozess-Definition bestimmter Prozess-Variablen gegeben. Zwei Prozess-Instanzen haben genau dann unterschiedliche IDs, wenn sie sich in den Werten dieser Variablen unterscheiden. *Correlation Handling* heißt der Mechanismus in BPEL4WS, der diese Zuordnung sichert.

Die einfacher zu erfassende ID einer Instanz modellieren wir direkt und führen die Menge der Prozess-Instanzen ein:

Universe: *ProcessInstance*

Mit dieser Formalisierung können wir nun auch im ASM-Modell Prozess-Struktur-Elementen ausführungsbedingte Eigenschaften zuordnen. Jede ausführungsbedingte Eigenschaft formalisieren wir als Funktion, die in die Menge der entsprechenden Werte der Eigenschaft abbildet. Argument der Funktion ist aber nicht das Prozess-Struktur-Element (z.B. eine Unterstruktur einer Variablen) allein, sondern auch die ID der Instanz, in dem diesem Element der Wert zugeordnet wird:

$$\mathit{variablePartValue} : \mathit{ProcessInstance} \times \mathit{Variable} \times \mathit{MsgTypePart} \rightarrow \mathit{Value}$$

Die Funktion *variablePartValue* ist das Analogon zu *msgPartValue* aus Abschnitt 3.2 für Variablen; das Prozess-Struktur-Element ist durch $\mathit{Variable} \times \mathit{MsgTypePart}$ beschrieben.

Jede Funktion, die eine ausführungsbedingte Eigenschaft eines Prozess-Struktur-Elements beschreibt, hat zwei weitere Charakteristika: Erstens beschreibt sie einen Aspekt einer

Instanz eines BPEL4WS-Prozesses; wir nennen diese Funktionen daher auch *instanz-beschreibend*. Zweitens sind die Definitionen der Funktionen wegen der ausföhrungsbedingten Eigenschaften, die sie beschreiben, auch von Zustand zu Zustand einer Instanz verschieden, sie sind also dynamisch. Ihre Funktionssymbole werden wir daher auch im Folgenden auf der linken Seite von Σ -Assignments finden.

Ein wenig anders ist die Modellierung der Aktivitäten aus Instanzsicht. Ihre Strukturen sind durch den inneren Aufbau des jeweiligen Aktivitätskonzepts definiert. Das Konzept ordnet dem Prozess-Struktur-Element gewisse, zusätzliche, ausföhrungsbedingte Eigenschaften wie interne Zustände, Zustand des Kontrollflusses, etc. zu. Dies können wir bereits modellieren.

Der wirkliche Unterschied zwischen den Laufzeiteigenschaften einer Aktivität und anderen Prozess-Struktur-Elementen findet sich an einer anderen Stelle: Die Ausführung einer Aktivität ist 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 (abstrahiert zum generischen Symbol *self*) repräsentiert. Daher ergeben sich diese beiden Abbildungen:

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

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 von der Middleware empfangen können.

4.3 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 *aktiviert* ist.

4.3.1 Zustandsänderungen & das Blockkonzept

Die Aktivierung der Aktivitäten 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 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 Kind-Aktivitäten 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). Diese Arbeit beschränkt sich auf ein Modell, das keine Scopes enthält; wird das Scope-Konzept integriert wird der Zustandsautomat komplexer. Die Zustände und die Übergänge sind dabei an das *BusinessAgreement Protocol* [CGK⁺03, 20.3] angelehnt. Jede Aktivität befindet sich in jeder Prozess-Instanz in einem der genannten Zustände (endliches Teiluniversum *ActivityState*), modelliert durch die dynamische Funktion *activityState*.

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

$activityState : ProcessInstance \times Activity \rightarrow ActivityState$

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

$$activityState(pI, a) = disabled. \quad (4)$$

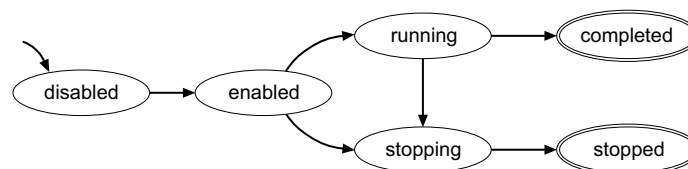


Abbildung 5: Zustandsautomat der BPEL4WS-Aktivitäten

Der Zustandsautomat (Abb. 5) steuert nun, in welchen Zustand die Aktivität versetzt wird, abhängig von ihrem aktuellen Zustand und äußeren Bedingungen. Er gibt den ASM-Regeln, die die Aktivitäten modellieren, eine grundlegende, einheitliche Struktur:

```
if activityState(pl, self) = enabled then  
    StartActivity  
if activityState(pl, self) = running then  
    RunActivity  
if activityState(pl, self) = stopping then  
    StopActivity
```

Die Übergänge finden sich in *StartActivity*, *RunActivity* und *StopActivity* bzw. extern beim Übergang von *disabled* nach *enabled*. 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 (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 Steuerung der Aktivitätsausführung, um die hierarchischen Aufrufbeziehungen zu modellieren.

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

1. Aufrufen einer Kind-Aktivität, so dass diese ausgeführt wird.
2. Vorzeitiges Terminieren der Ausführung einer Kind-Aktivität im Fehlerfall.

Gleichzeitig überwacht die Vater-Aktivität ständig den Zustand der Kind-Aktivität, um so die weitere Ausführung zu steuern.

Diese Steuerungsbeschränkung durch die Vater-Aktivität 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 erst einmal angestoßen werden. Dies geschieht wiederum durch die, dem Web Service zugrundeliegende, 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 abhängig. Erreicht die Wurzel-Aktivität den Zustand *completed*, so ist die Ausführung dieser Prozess-Instanz beendet.

4.3.2 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 Kind-Aktivitäten nebenläufig ausgeführt werden. Ist T ein Teilbaum des Aktivitätsbaumes 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*-Aktivität 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. Ist eine Aktivität Ziel mehrerer Links, so können wir mittels der *Join-Condition* komplexere Aktivierungsbedingungen für die Target-Aktivität angeben, z.B: „mind. ein Link muss aktiviert sein“ oder „Link A und Link B oder Link C müssen aktiviert sein“. Ist keine Join-Condition gegeben, so gibt BPEL4WS standardmäßig die Bedingung „mind. ein Link muss aktiviert sein“ vor.

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 Aktivitätskonzepte (Sequence und Reply) modellieren.

4.4 Sequence

Die Aktivität Sequence (Tag: **sequence**) ist eine strukturierte Aktivität. Wir haben sie für diese Arbeit ausgewählt, weil sich anhand ihrer Semantik sehr schön das hierarchische Zusammenspiel der Aktivitäten zeigt. Die Sequence steuert die sequentielle Ausführung ihrer Kind-Aktivitäten.

Die Syntax der Sequence ist in Listing 6 gegeben, wobei *activity* für jede beliebige Aktivität stehe.

Listing 6: Syntax der Sequence

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

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

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

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. *seqActivityNext* und

seqActivityFirst sind prozess-beschreibende Funktionen, deren Definition uns im Anfangszustand aus der Prozess-Definition gegeben ist.

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
    STOPSTRUCTURED(pl, 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 sie gar nicht erst ausgeführt werden.

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

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

Im zweiten Fall verhindert eine noch nicht erfüllte Join-Condition den Übergang. Den Wert der Join-Condition im aktuellen Zustand wollen wir hier nur mittels einer abstrakten Funktion ermitteln:

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

bildet auf *true* ab, falls die Join-Condition der Aktivität *act* in der Prozess-Instanz *pI* wahr ist; andernfalls auf *false*.

```

STARTACTIVITY
  (pl ∈ ProcessInstance, act ∈ Activity) ≡
  if activityStop(pl, act) = true then
    activityState(pl, act) := stopping
  else if joinConditionEnabled(pl, act) = true then
    activityState(pl, act) := running

```

4.4.1 Aktivierung von Kind-Aktivitäten

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

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

seqCurrentActivity bildet als dynamische Funktion stets auf die aktuell ausgeführte Kind-Aktivität ab und ist zu Anfang für alle Prozess-Instanzen und Sequences undefiniert. Sie ist auch die erste Funktion, die wir kennenlernen, die spezifisch für das Sequence-Konzept definiert ist.

```

RUNSEQUENCE
(pl ∈ ProcessInstance, seq ∈ Sequence) ≡
if activityStop(pl, seq) = true then
  activityState(pl, seq) := stopping
else
  let current = seqCurrentActivity(pl, seq) in
  if current = undef then
    let first = seqActivityFirst(seq) in
      ENABLESEQUENCECHILD(pl, seq, first)
  else if activityState(pl, current) = completed then
    let next = seqActivityNext(current) in
      if next ≠ undef then
        ENABLESEQUENCECHILD(pl, seq, next)
      else
        activityState(pl, seq) := completed
        ACTIVATELINKS(pl, seq)

```

Eine Kind-Aktivität 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:

```

ENABLESEQUENCECHILD
(pl ∈ ProcessInstance, seq ∈ Sequence, act ∈ Activity) ≡
seqCurrentActivity(pl, seq) := act
activityState(pl, act) := 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.

Wir wollen nun eine einfache Eigenschaft der Sequence, die sich aus der informalen Spezifikation ableiten lässt, am formalen ASM-Modell beweisen: *Jede Sequence, die einmal aktiviert ist, terminiert nach endlich vielen Schritten.*

Dabei müssen wir einige vereinfachende Annahmen treffen, da wir in dieser Arbeit nicht alle für die Eigenschaft notwendigen Aspekte der Semantik formalisiert haben.

Behauptung *Wenn eine Sequence in den Zustand enabled versetzt wird (1), ihre Join-Condition zu wahr berechnet wird (2), kein vorzeitiges Signal zur Terminierung die Sequence erreicht (3) und für jede Kind-Aktivität A der Sequence gilt: wenn A in den Zustand enabled versetzt wird, so erreicht A nach endlich vielen Schritten den Zustand completed (4), dann*

erreicht auch die Sequence den Zustand `completed`.

Beweis Gegeben sei ein Zustand S_0 , $seq \in Sequence$ und $pI \in ProcessInstance$. Es seien $a_1, \dots, a_n \in Activity$ die Kind-Aktivitäten von seq in S_0 : $activityChilds(seq) = \{a_1, \dots, a_n\}$. Weiterhin seien in S_0 die Formeln $seqActivityFirst(seq) = a_1$ und für alle $i \in [n - 1] : seqActivityNext(a_i) = a_{i+1}$ sowie $seqActivityNext(a_n) = undef$ erfüllt.

Aus den Voraussetzung (1), (2) und (3) folgt, dass in S_0 $activityState(pI, seq) = enabled \wedge activityStop(pI, seq) = false \wedge joinConditionEnabled(pI, seq) = true$ gilt.

Dann führt die Anwendung von `EXECUTESEQUENCE` auf S_0 zur Anwendung von `STARTACTIVITY` mit $pl = pI$ und $act = seq$ auf S_0 . Daraus folgt, dass im Folgezustand S_1 $activityState(pI, seq) = running$ gilt. Nach der Spezifikation von $seqCurrentActivity$ gilt $seqCurrentActivity(pI, seq) = undef$ im selben Zustand.

Zieht der zu seq gehörende Agent erneut, so wird in `RUNSEQUENCE` `current` an $undef$ gebunden und damit `first` an $seqActivityFirst(seq) = a_1$. Folglich wird `ENABLESEQUENCECHILD` auf S_1 angewandt, wobei pl an pI , seq an seq und act an a_1 gebunden sind: $\varphi_1 \equiv seqCurrentActivity(pI, seq) = seqActivityFirst(seq) = a_1 \wedge activityState(pI, a_1) = enabled$ gilt im Folgezustand S_2 .

Wir verallgemeinern nun von S_2 auf S_2^i , in dem φ_i gelte.

Nach Voraussetzung (4) existiert ein von S_2^i in endlich vielen Schritten erreichbarer Zustand S_3^i in dem $\psi_i = activityState(pI, a_i) = completed \wedge seqCurrentActivity(pI, seq) = a_i \neq undef$ erfüllt ist.

Für die erneute Anwendung von `EXECUTESEQUENCE` und damit `RUNSEQUENCE` durch den Agenten von seq auf S_3^i unterscheiden wir zwei Fälle:

1. $1 \leq i < n$, dann ist $seqActivityNext(a_i) = a_{i+1} \neq undef$, `next` wird an a_{i+1} gebunden und `ENABLESEQUENCECHILD` wird auf S_3^i angewandt, mit den Bindungen: pl an pI , seq an seq und act an a_{i+1} . Im Folgezustand gilt φ_{i+1} , den wir daher mit S_2^{i+1} bezeichnen.
2. $i = n$, dann ist $seqActivityNext(a_n) = undef$, `next` wird an $undef$ gebunden und aus der Regelanwendung folgt, dass $activityState(pI, seq) = completed$ im Folgezustand S_4 gilt.

Unsere Verallgemeinerung von S_2 auf S_2^i ist gerechtfertigt, da der letzte Schritt auch für das mit S_2 identische S_2^1 entsprechend gilt. Wir gelangen so zu einer Folge von Zuständen $S_0 S_1 S_2^1 \dots S_3^1 S_2^2 \dots S_3^2 S_2^3 \dots S_2^n \dots S_3^n S_4$. Zieht der Agent von seq in einem Zustand zwischen S_2^i und S_3^i , so gilt in diesem $activityState(pI, seqCurrentActivity(pI, seq)) \neq completed$ nach Voraussetzung (4) und es findet kein Σ -Update durch diesen Agenten statt.

Damit haben wir die Eigenschaft nachgewiesen. \square

4.4.2 Vorzeitige Terminierung von Kind-Aktivitäten

Ganz analog zur Aktivierung der Kind-Aktivitäten 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 Kind-Aktivitäten (*activityChilids*, s. Abschnitt 3.3). Die Aktivität wartet anschließend darauf, dass alle Kind-Aktivitäten entweder bereits vollständig ausgeführt wurden (*completed*), oder dass diese und deren Kind-Aktivitäten vorzeitig terminiert wurden (*stopped*), um dann selbst zu terminieren.

STOPSTRUCTURED

```

(pl ∈ ProcessInstance, str ∈ Activity_structured) ≡
if ∀ch ∈ activityChilids(str) : activityState(pl, ch) ∈ {stopped, completed} then
    activityState(pl, str) := stopped
else
    forall ch ∈ activityChilids(str)
        where activityState(pl, ch) ∈ {enabled, running, disabled} do
            activityState(pl, ch) := stopping

```

Für *alle strukturierten* Aktivitäten (**Universe:** $Activity_{structured} \subseteq Activity$) 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.

4.4.3 Die Module MIDDLEWARE und SEQUENCE

Die in den Abschnitten 3, 4.2 und 4.3 definierten Funktionen, sowie *activityStop* und STARTACTIVITY und STOPSTRUCTURED bilden das Modul MIDDLEWARE. Es formalisiert das viel zitierte Middleware-Konzept, 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 importiert. Gleiches gilt für alle Funktionen und ASM-Regeln von MIDDLEWARE. Damit formalisiert SEQUENCE ausschließlich Aspekte der Sequence. Ganz analog werden wir nun auch ein Modul für die elementare Aktivität Reply definieren.

4.5 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. Nachrichten werden über den Ausgabekanal der Operation, über deren Eingabekanal die vorausgegangene Nachricht einging, versandt. Anstatt einer Nachricht darf auch eine Fehlermeldung als Antwort verschickt werden. Listing 7 zeigt die Syntax für Reply.

Listing 7: 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 3.1 beschrieben – im Anfangszustand aufgrund eines konkreten BPEL4WS-Prozesses definiert sind.

Universe: *Reply*

Universe: *PartnerLink*

Universe: *PortType*

Universe: *Operation*

Universe: *FaultName*

Universe: *CorrelationSet*

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

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

EXECUTEREPLY

```
(self)  $\equiv$ 
let reply = myStatic(self) in
let pl = currentInstance(self) in
  if activityState(pl, reply) = enabled then
    STARTACTIVITY(pl, reply)
  if activityState(pl, reply) = running
    RUNREPLY(pl, reply)
  if activityState(pl, reply) = stopping
    STOPREPLY(pl, reply)
```

Die Aktivierung von Reply ist bereits mit STARTACTIVITY in Abschnitt 4.4 vollständig beschrieben. Wir können also direkt das Verhalten betrachten.

4.5.1 Erzeugen und Senden von Nachrichten

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 die Bedingungen des Correlation Handlings prüfen.

$$\begin{aligned} & \textit{correlationSatisfied}_{var} : \\ & \textit{ProcessInstance} \times \textit{Variable} \times \mathcal{P}(\textit{CorrelationSet}) \rightarrow \{true, false\} \end{aligned}$$

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. Die Regel SETCORRELATION belege noch undefinierte Schlüsselfelder mit den entsprechenden Werten, worauf wir aus den genannten Gründen nicht im Detail eingehen.

EXECUTEREPLY

```

(pl ∈ ProcessInstance, reply ∈ Reply) ≡
if activityStop(pl, reply) = false then
  let var = replyVariable(reply) in
    let CS = replyCorrelationSets(reply) in
      if correlationSatisfiedvar(pl, var, CS) = true then
        SENDREPLYMESSAGE(pl, reply)
        SETCORRELATIONHANDLING(pl, var, CS)
        activityState(pl, reply) := completed
      else
        THROW(pl, reply, correlationViolation)
  else
    activityState(pl, 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, disjunkte, unendliche 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, wie schon in Abschnitt 3.2 beschrieben. Fehlernachrichten werden durch Namen (*FaultName*) klassifiziert. Die dynamische Funktion *faultFName* 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

```

(pl ∈ ProcessInstance, reply ∈ Reply) ≡
if replyFaultName(reply) = undef then
  let message = new(BPELMsg) in
    COPYVARIABLETOMSG(pl, replyVariable(reply), message)
    SENDMESSAGEreply(pl, replyPartnerLink(reply),
      replyPortType(reply), replyOperation(reply), message)
else
  let fault = new(Fault) in
    COPYVARIABLETOMSG(pl, replyVariable(reply), fault)
    faultFName(fault) := replyFaultName(reply)
    SENDMESSAGEfault(pl, reply, fault)

```

4.5.2 Nachrichtenkanäle

Der Begriff des *Nachrichtenkanals* (Ein- und Ausgabekanal von Aktivitäten) entsteht durch die Kombination zweier Sichten. Zunächst existiert die *physische Sicht*, die aus der Perspektive eines Web Services auf die physische Adresse der Middleware des anderen Web Services, 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 Services 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 Services. Dieser ist über den *PartnerLink* (*replyPartnerLink*) des Repls 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 ordnen jeder Operation in dem zugehörigen PortType an der Adresse der Middleware eine Menge von (zu empfangenden) Nachrichten zu.

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

Die logische Sicht des Kanals ist eine Abbildung der Nachrichten auf Datenstrukturen, die die Instanz des Absenders und des Empfängers entsprechend identifizieren (vgl. [CB⁺03]). Aus Gründen der Komplexität der Datenstrukturen 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 parallel komponierte Regeln.

$$\begin{array}{l} \hline \text{SENDMESSAGE}_{reply} \\ \hline (pl \in ProcessInstance, partnerL \in PartnerLink, pT \in PortType \\ op \in Operation, msg \in BPELMsg) \equiv \\ \text{let address} = partnerAddress(pl, partnerL) \text{ in} \\ \text{SETENDPOINTS}(pl, partnerL, msg) \\ portMessages_{reply}(address, pT, op) := \\ portMessages_{reply}(address, pT, op) \cup \{msg\} \end{array}$$

$$\begin{array}{l} \hline \text{SENDMESSAGE}_{fault} \\ \hline (pl \in ProcessInstance, partnerL \in PartnerLink, pT \in PortType \\ op \in Operation, fault \in Fault) \equiv \\ \text{let address} = partnerAddress(pl, partnerL) \text{ in} \\ \text{SETENDPOINTS}(pl, partnerL, fault) \\ portMessages_{fault}(address, pT, op) := \\ portMessages_{fault}(address, pT, op) \cup \{fault\} \end{array}$$

4.5.3 Zuweisen von Nachrichteninhalten

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 *pl* spezifiziert. An dieser Stelle treffen noch einmal ASM-Regeln, dynamische und statische Funktionen aufeinander.

Die für die Zuweisung nötigen statischen (genauer: prozess-beschreibenden) Strukturen sind die Typ-Definition von Variable und Nachricht, formalisiert durch *MessageType*, *MessageTypeParts* und *messageTypeParts* (vgl. Abschnitt 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. Abschnitt 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 Reply 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 entsprechend der Einleitung in diesen Abschnitt aus.

COPYVARIABLETOMSG

(*pl* ∈ *ProcessInstance*, *var* ∈ *Variable*, *msg* ∈ *BPELMsg*) ≡

if *var* ≠ *undef* **then**

forall *part* ∈ *messageTypeParts(variableMsgType(var))* **do**

msgPartValue(msg, part) := *variablePartValue(pl, var, part)*

msgMsgType(msg) := *variableMsgType(var)*

4.5.4 Das Modul REPLY

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.

5 Schlußfolgerungen und Ausblick

Wir haben in dieser Arbeit an Beispielen gezeigt, welchen Weg wir bei der Formalisierung der Semantik für BPEL4WS mit Abstract State Machines einschlagen. Die Struktur der Sprache selbst gab uns dabei auch die Architektur der Formalisierung vor: Die Aktivitätskonzepte und das Middlewarekonzept definieren auf abstrakter Ebene die möglichen Verhalten von BPEL4WS-Prozessen. Eine BPEL4WS-Prozess-Definition definiert konkrete Ausprägungen der Konzepte, deren Verhalten dadurch auf den Spezialfall des gegebenen Prozesses eingeschränkt wird.

Das Verhalten der Konzepte können wir nahezu unmittelbar aus der informalen Semantik heraus direkt formalisieren. Die dazu notwendigen Strukturen gewinnen wir zum einen durch Abstraktion von BPEL4WS-Prozess-Definitionen und zum anderen aus impliziten Vorgaben der informalen Spezifikation selbst, zum Beispiel interne Zustände einer Aktivität. Die explizite Formulierung des Middlewarekonzepts ist ein wesentlicher „kreativer“ Anteil an der Formalisierung des Verhaltens.

Dabei bilden wir (wie in Abschnitt 4 gezeigt) das engste allgemeine Verhalten in ASM-Regeln direkt ab, wie beispielsweise die Möglichkeit ein Reply sowohl reguläre als auch Fehlernachrichten aus einer gegebenen Variable verschicken zu lassen.

Die Menge aller zum Konzept gehörenden Verhalten erfassen wir durch Abstraktion der Strukturen, auf denen dieses abläuft. Die Strukturen erscheinen in den ASM-Regeln in Form von zu Termen verknüpften Funktionssymbolen, deren Interpretation aus einem gegebenen BPEL4WS-Prozess erwächst.

Die Funktionssymbole repräsentieren Funktionsklassen mit ganz bestimmten Eigenschaften, welche wir aus dem Übersetzungsverfahren in Abschnitt 3 gewonnen haben. Die Eigenschaften charakterisieren stets ein bestimmtes Verhältnis bzw. eine bestimmte Art der Abhängigkeit zwischen strukturellen Elementen des Prozesses wie „ist Kind-Element von“ oder „hat den Typ.“ Diese Eigenschaften haften als Klasseneigenschaften gleichsam ihrem Funktionssymbol an.

Mit dem in Abschnitt 1.3.3 beschriebenen Prinzip, Verhältnisse und Abhängigkeiten ausschließlich anhand der Gleichheit von Termen zu betrachten, können wir zudem auch die Abläufe des Konzepts anhand seiner ASM-Regel abstrakt nachvollziehen, ohne die konkrete Interpretation der Symbole zu kennen. Dies genügt, um *alle* möglichen Verhalten eines Konzepts in *einer* ASM-Regel zu erfassen und zu verstehen.

Grundlage für die Abstraktion bildet die hinsichtlich der Funktionsklassen korrekte Übersetzung einer beliebigen BPEL4WS-Prozess-Definition in einen zugehörigen Anfangszustand, in dem die den Funktionssymbolen zugeschriebenen Eigenschaften von ihrer Interpretation erfüllt werden. Das Verfahren aus Abschnitt 3, das uns diese Korrektheit garantiert, ist somit wesentlicher Bestandteil unserer Formalisierung. Es bildet von konkreten BPEL4WS-Prozess-Definitionen, die das Verhalten der Konzepte einschränken, ab auf konkrete Funktionsdefinitionen, die das Verhalten des Systems einschränken, welches durch die ASM-Regeln der formalisierten Konzepte beschrieben wird.

Die semantische Domäne der Formalisierung eines BPEL4WS-Prozesses ist damit die Menge seiner Abläufe im ASM-Modell, die sich lediglich aufgrund von Nichtdeterminismus und Eingaben (empfangene Nachricht mit Inhalt und Absender) unterscheiden. Dieses Formalisierungsprinzip unterscheidet sich von anderen Ansätzen zur Formalisierung von Pro-

grammiersprachen. So hat beispielsweise C.Stahl in [Sta04] und [MSW⁺04] eine formale Semantik für BPEL4WS definiert, deren semantische Domäne Petrinetze sind: ein gegebener BPEL4WS-Prozess wird mit einem musterbasierten Ansatz in ein Petrinetz übersetzt. Ein derartig gegenständliches Übersetzungsziel hat unser Ansatz nicht, dessen Stärken vielmehr in der abstrakten Betrachtung der Abläufe liegt.

Anstatt den Formalismus selbst zu bemühen, verstehen wir Zustandsübergänge anhand der für alle Prozesse gleichen ASM-Regeln, welche auf den informal beschriebene Übergängen der Konzepte beruhen. Dadurch gelangen wir zu einer isomorphen Beziehung zwischen einem durch die informale Spezifikation beschriebenen Ablauf und dem zugehörigen Ablauf im formalen Modell – gerade das, was eine Formalisierung leisten soll.

Unser Vorgehen erlaubt uns außerdem, die informale und die formale Semantik auf konzeptueller Ebene zu vergleichen, also auf einer deutlich kleineren Komplexitätsstufe, als der gesamten Semantik der Sprache. Gleichzeitig besteht so die Möglichkeit, Eigenschaften, die wir aus der informalen Beschreibung ableiten und formalisieren, allein anhand der beteiligten Konzepte zu verifizieren und nicht die gesamte Sprache betrachten zu müssen.

Allerdings dürfte der Umfang des Beweises in Abschnitt 4.4.1 für die doch recht einfache Eigenschaft deutlich gemacht haben, dass dafür eine handhabbare Beweismethode nötig ist. Dabei sollten wir bedenken, dass wir für diesen Beweis sogar noch vereinfachende Annahmen hinzugenommen haben. Uns erscheint eine temporale Logik am geeignetsten, um diese Aufgabe zu bewältigen.

Es sei noch bemerkt, dass eine Gruppe an der Simon Fraser University in Vancouver, Canada ebenfalls eine formale Semantik für BPEL4WS mit Abstract-State Machines erstellt [FGV04]. Interessanterweise ähneln sich die beiden Ansätze trotz unabhängiger Arbeit sowohl in ihrer semantischen Domäne, als auch in ihren ASM-Regeln sehr stark. Die Unterschiede beruhen in erster Linie darauf, dass Farahbod, Glässer und Vajihollahi die Regeln in AsmL notieren, um ein ausführbares Modell zu erhalten. Von diesem Schritt haben wir bislang aus Gründen der Klarheit und Verständlichkeit der formalen Semantik Abstand genommen.

Ergänzend hat diese Arbeit auch gezeigt, dass Abstract-State Machines eine Vielzahl von Abstraktionsmechanismen bereit stellen. Dabei sind diese Mechanismen einerseits dem ASM-Ansatz der gleichen Interpretation von Termen geschuldet und andererseits dem Grad der Formalisierung einzelner Komponenten des Systems. Gerade letzteres gibt dem Modellierer die nötigen Freiheiten, die in der Einleitung zitierten Eigenschaften auch zu erfüllen.

Der Modellierer kann wählen, ob eine Unterstruktur eines Objektes relevant für das erste Modell hinsichtlich der Anforderungen ist, oder eher der Implementation zugeordnet wird. Je nachdem kann er die aus der Unterstruktur resultierenden Abhängigkeiten in Funktionen formalisieren oder nicht. In beiden Fällen ist das Objekt an sich (als Element des Trägers) im Modell korrekt formalisiert. Weiterhin hat der Modellierer die Wahl, wie konkret er Funktionssignaturen mit Leben füllen möchte.

Der ASM-Ansatz erlaubt es problemlos auch bei informal definierten Funktionen vernünftig über die Gleichheit von interpretierten Termen zu reden. Auf diese Weise können Aspekte des Systems in das formale Modell des System aufgenommen werden oder durch abstrakte Funktionen der Umgebung zugeschrieben werden. Im zweiten Fall bilden diese Funktionen die Schnittstelle des formalisierten Systems zu seiner Umgebung. Dies ist uns an vielen

Stellen wie beispielsweise den Nachrichtenkanälen hilfreich, ohne dem Verständnis und der Korrektheit Abbruch zu tun. Ist es notwendig, die Schnittstelle formal präzise zu erfassen, so leistet bereits eine konkrete Funktionsdefinition aus Sicht des Systems das Verlangte. Wie diese zustande kommt, ist eine Frage der mit der Schnittstelle zusammenhängenden Spezifikation der Umgebung.

Alle hier gezeigten und verwendeten Abstraktionsmechanismen sind durch das Konzept der Σ -Algebra gegeben und stellen keine Ergänzung des Formalismus dar. Der ASM-Ansatz unterstützt also unmittelbar die Wahl des Abstraktionsgrades, so dass Anforderung und Implementationsdetails klar getrennt werden können.

Zusammenfassend können wir sagen, dass der gewählte Ansatz Erfolg versprechend ist. Wir können nach dem beschriebenen Verfahren eine formale Semantik erstellen, die für alle BPEL4WS-Prozesse gültig ist. Die formalen Grundlagen des ASM-Formalismus erlauben es uns, wichtige Eigenschaften der Sprache für das modellierte System und in dem gewählten Ansatz ganz allgemein für die Sprache an sich *und* für konkrete BPEL4WS-Prozesse zu überprüfen. Die Abstraktionsmechanismen und das Prinzip der Gleichheit von interpretierten Termen machen das Modell dank der selbst-sprechenden Symbole (Bezeichner) intuitiv leicht verständlich. Wir haben damit auch gezeigt, dass Abstract-State Machines für die Formalisierung von BPEL4WS im Allgemeinen und auch entsprechend der Eigenschaften aus Abschnitt 1.3 geeignet sind. Diesem hängt jedoch ein Wermutstropfen an: wichtige Eigenschaften des Modells zu beweisen, ist in der Praxis nahezu unmöglich. Der Formalismus ist zu ausdrucksstark. Wir müssen uns an dieser Stelle mehr auf das intuitive Verständnis des formalen Modells verlassen.

Ausgehend vom hier beschriebenen Ansatz müssen für eine vollständige formale Semantik der Business Process Execution Language for Web Services alle noch verbliebenen Aktivitäten der Sprache modelliert werden. Weiterhin gilt es, die Konzepte der Fehler-, Kompensations- und Ereignisbehandlung in das Modell zu integrieren. 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.

Unsere bisherigen Vorarbeiten zur Semantik von BPEL4WS, innerhalb der BPEL4WS-Taskforce an der Humboldt-Universität zu Berlin [BPT], haben dabei neben einigen unscharfen und unsauberen Formulierungen in der informalen Spezifikation auch einige konzeptionell schwerwiegende Fehler im Bereich des Link-Konzepts sowie der Kompensationsbehandlung aufgezeigt, deren Entdeckung bereits in den Standardisierungsprozess der Sprache eingeflossen sind. Wir werden darauf in einer zukünftigen Arbeit zur formalen ASM-Semantik von BPEL4WS eingehen.

Literatur

- [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/>
- [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
- [Bo95] BÖRGER, E.: Why use Evolving Algebras for Hardware and Software Engineering? In: BARTOSEK, M. (Hrsg.) ; STANDEK, J. (Hrsg.) ; WIEDERMANN, J. (Hrsg.): *SOFSEM'95, 22nd Seminar on Current Trends in Theory & Practice of Informatics* Bd. LNCS 1012, Springer-Verlag, 1995, S. 235–271
- [BPT] Die Webseite der BPEL4WS-Taskforce: http://www.informatik.huberlin.de/top/forschung/projekte/vgp_mit_ws/bpel/
- [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/>
- [CCMW01] CHRISTENSEN, Erik ; CURBERA, Francisco ; MEREDITH, Greg ; WEERAWARANA, Sanjiva: Web Services Description Language (WSDL) 1.1 / Ariba, IBM, Microsoft. 2001. – Specification. <http://www.w3.org/TR/wsdl.html>
- [CDR⁺02] CURBERA, F. ; DUFTLER, M. ; R., Khalaf ; W., Nagy ; N., Mukhi ; S., Weerawarana: Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. In: *IEEE Internet Computing* 6(2) (2002), S. 86–93
- [CGK⁺03] CURBERA, F. ; GOLAND, Y. ; KLEIN, J. ; LEYMAN, F. ; ROLLER, D. ; WEERAWARANA, S. ; THATTE, S. (Hrsg.): Business Process Execution Language for Web Services Version 1.1 / BEA Systems, IBM, Microsoft, SAP, Siebel. 05 May 2003. – Specification. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbiz2k2/html/bpel1-1.asp>

-
- [CJ03] COLGRAVE, John ; JANUSZEWSKI, Karsten: Using WSDL in a UDDI Registry, Version 2.0 / OASIS. 2003. – Technical Note
- [CK04] CURBERA, Francisco ; KHALAF, Ranja: Implementing BPEL4WS: The architecture of a BPEL4WS Implementation. In: *Proceedings of the Grid Workflow Workshop at GGF-10, 2004*
- [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
- [FGV04] FARAHBOD, Roozbeh ; GLÄSSER, Uwe ; VAJIHOLLAHI, Mona: Specification and Validation of the Business Process Execution Language for Web Services / Simon Fraser University, Burnaby B.C. Canada. 2004. – Technical Report. SFU-CMPT-TR-2003-06
- [Got00] GOTTSCHALK, Karl: Web Services architecture overview / IBM developerWorks. 2000. – Whitepaper. <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
- [Gur85] GUREVICH, Y.: A new thesis. In: *American Mathematical Society Abstracts* (1985), August, S. 317
- [Gur95] GUREVICH, Y.: Evolving Algebras 1993: Lipari Guide. In: BÖRGER, E. (Hrsg.): *Specification and Validation Methods*. Oxford University Press, 1995, S. 9–36
- [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 Transactions on Computational Logic* Vol.1 No.1 (2000), Juli, S. 77–111
- [IBM] Die Webseite der WebSphere Software Development Kit: <http://ibm.com/developerworks/webservices/wsdk/>
- [KL04] KOSSMANN, D. ; LEYMANN, F.: Web Services. In: *Informatik Spektrum* 26 (2004), S. 117–128

-
- [Kre03] KREGGER, Heather: Fulfilling the Web Service Promise. In: *Communication of the ACM* 46 (2003), June, Nr. 6, S. 29–34
- [Ley01] LEYMANN, F.: Web Service Flow Language (WSFL 1.0) / IBM Software Group. May 2001. – Specification
- [MSW⁺04] MARTENS, Axel ; STAHL, Christian ; WEINBERG, Daniela ; FAHLAND, Dirk ; HEIDINGER, Thomas: Business Process Execution Language for Web Services – Semantik, Analyse und Visualisierung / Humboldt-Universität zu Berlin. 2004. – Forschungsbericht
- [Now03] NOWACK, A.: Deciding the Verficiation 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
- [Rei03] REISIG, W.: On Gurevich’s theorem on sequential algorithms. In: *Acta Informatica* Vol. 39, No. 5 (2003), S. 273–305
- [RST04] RAPP, Elliot ; SHOHOUD, Yasser ; TAVIS, Matt: New Features for Web Service Developers in Beta 1 of the .NET Framework 2.0. In: *MSDN* (2004), June. – <http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnvs05/html/wsnetfx2.asp>
- [Sta04] STAHL, Christian: Transformation von BPEL4WS in Petrinetze / Humboldt-Universität zu Berlin. 2004. – Diplomarbeit
- [SUN] Die Webseite des Java Web Service Developer Packs: <http://java.sun.com/webservices/jwsdp/index.jsp>
- [Tha01] THATTE, S.: XLANG – Web Service for Business Process Design / Microsoft Corporation. 2001. – Specification
- [WADH02] WOHEDE, P. ; v.D. AALST, W. ; DUMAS, M. ; TER HOFSTEDDE, A.: Pattern Based Aanalysis of BPEL4WS / Queensland University of Technology, Brisbane. 2002. – Analysis