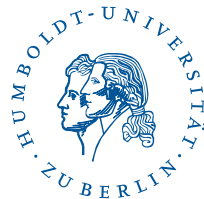


„Semantische Fundierung der
Web-Service-Beschreibungssprache WSCI“

Diplomarbeit

Humboldt Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik



von Stephan Weißleder
betreut durch Dr. habil. Karsten Schmidt

22. November 2004

Zusammenfassung

Wir beschäftigen uns in dieser Arbeit mit verschiedenen Beschreibungsmöglichkeiten für *Web Services*. Im Vordergrund stehen dabei die *Web-Service*-Beschreibungssprache *WSCI* und ihre Eigenschaften. Wir werden eine Semantik entwickeln, die Stärken und Schwächen der zugrunde liegenden Spezifikation [AAF+02] aufzeigt und mit deren Hilfe wir die Eigenschaften von *WSCI* überprüfen können.

Inhaltsverzeichnis

1	Einleitung	5
1.1	Motivation	5
1.2	Umsetzung	6
2	Grundlagen	8
2.1	WSDL	8
2.1.1	Überblick	8
2.1.2	Beispiel	10
2.2	Petrinetze	12
3	Web Services	15
3.1	Sichtweisen	15
3.1.1	Lokale Sichtweise	15
3.1.2	Globale Sichtweise	16
3.1.3	Vergleich	16
3.2	Sprachen	17
3.2.1	BPEL	17
3.2.2	WSCI	19
3.2.3	Vergleich	20
4	Entwicklung einer Semantik zu WSCI	22
4.1	Bausteine	22
4.2	Hierarchie der Elemente aus WSCI	26
4.3	Herangehensweise bei der Entwicklung der Semantik	28
4.4	Prozessinstanzsteuerung	30
4.4.1	Lebenszyklusprozess	30
4.4.2	Lebenszykluskontext	33
4.4.3	Lebenszykluskontrolle	36
4.4.4	Nachrichtempfang	37
4.4.5	Rufempfang	39
4.5	Fehlerbehandlung	40

4.5.1	Ereigniskontext	40
4.5.2	ExceptionHandler	43
4.5.3	DefaultExceptionHandler	45
4.5.4	MessageHandler	45
4.5.5	TimeoutHandler	47
4.6	Komplexe Aktivitäten	49
4.6.1	All	49
4.6.2	Choice	51
4.6.3	Foreach	52
4.6.4	Sequence	53
4.6.5	Switch	54
4.6.6	Until	56
4.6.7	While	57
4.7	Atomare Aktivitäten	58
4.7.1	Action	59
4.7.2	Delay	64
4.7.3	Empty	64
4.7.4	Fault	65
4.7.5	Call	66
4.7.6	Spawn	67
4.7.7	Join	68
4.8	Beispiele für die Transformation von WSCI in Petrinetze	70
4.8.1	Dynamisches Interface in WSCI	70
4.8.2	Darstellung in Petrinetz-Bausteinen	71
4.8.3	Eigenschaften der Beispiele	73
4.8.4	Vereinfachungsmöglichkeiten	74
5	Abschließende Betrachtungen	76
5.1	Zusammenfassung	76
5.2	Ausblick	78

Abbildungsverzeichnis

2.1	WSDL ER-Modell	9
2.2	Petrinetz	14
4.1	Baustein allgemein	23
4.2	Verknüpfung von Bausteinen	25
4.3	Top-Level Elemente	26
4.4	Komposition von Web Services	27
4.5	ActivitySet	28
4.6	Kombination von Prozess und Kontext	28
4.7	Modellierung eines Kontextes	30
4.8	Petrinetz-Baustein für <i>Lebenszyklusprozess</i>	32
4.9	Petrinetz-Baustein für <i>Lebenszykluskontext</i>	34
4.10	Petrinetz-Baustein für <i>Lebenszykluskontrolle</i>	37
4.11	Petrinetz-Baustein für <i>Nachrichtempfang</i>	38
4.12	Petrinetz-Baustein für <i>Rufempfang</i>	39
4.13	Petrinetz-Baustein für <i>Ereigniskontext</i>	42
4.14	Petrinetz-Baustein für <i>ExceptionHandler</i>	44
4.15	Petrinetz-Baustein für <i>DefaultExceptionHandler</i>	45
4.16	Petrinetz-Baustein für <i>MessageHandler</i>	46
4.17	Petrinetz-Baustein für <i>TimeoutHandler</i>	47
4.18	Petrinetz-Baustein für <i>All</i>	50
4.19	Petrinetz-Baustein für <i>Choice</i>	52
4.20	Petrinetz-Baustein für <i>Foreach</i>	53
4.21	Petrinetz-Baustein für <i>Sequence</i>	54
4.22	Petrinetz-Baustein für <i>Switch</i>	55
4.23	Petrinetz-Baustein für <i>Until</i>	56
4.24	Petrinetz-Baustein für <i>While</i>	58
4.25	Petrinetz-Baustein für <i>Notification</i>	60
4.26	Petrinetz-Baustein für <i>One-Way</i>	61
4.27	Petrinetz-Baustein für <i>Request-Response</i>	62
4.28	Petrinetz-Baustein für <i>Solicit-Response</i>	63
4.29	Petrinetz-Baustein für <i>Delay</i>	64

4.30	Petrinetz-Baustein für <i>Empty</i>	65
4.31	Petrinetz-Baustein für <i>Fault</i>	65
4.32	Petrinetz-Baustein für <i>Call</i>	66
4.33	Petrinetz-Baustein für <i>Spawn</i>	67
4.34	Petrinetz-Baustein für <i>Join</i>	68
4.35	<i>Web Service</i> Nr. 1	72
4.36	<i>Web Service</i> Nr. 2	73

Kapitel 1

Einleitung

1.1 Motivation

Prozesse unterschiedlichster Art bestimmen den Arbeitsablauf in jedem Unternehmen. Deren Wettbewerbsfähigkeit hängt maßgeblich von der Effizienz dieser Prozesse ab. Über das Internet kommunizieren Prozesse weltweit und plattformübergreifend miteinander. Unterschiedliche Kommunikationsprotokolle und Datenformate dürfen dabei also keine Hindernisse sein. *Web Services* bauen auf entsprechend standardisierten Mitteln der Kommunikation auf. Sie werden durch *XML*-basierte Dokumente beschrieben und sowohl die benutzten Nachrichtenformate als auch die Kommunikationsprotokolle sind Standards.

Ein *Web Service* ist eine abgeschlossene, selbsterklärende und modulare Software-Komponente, die über das Internet veröffentlicht, aufgefunden und benutzt werden kann [Moh02].

Das wachsende Angebot an *Web Services* macht ihre Unterscheidung und eine Bewertung anhand ihrer Eigenschaften notwendig. Dazu gehört zum einen die Bewertung praktischer Eigenschaften, welche beispielsweise die Effizienz und den gebotenen Leistungsumfang des betrachteten *Web Service* beschreiben. Zum anderen sind die theoretischen Eigenschaften zu bewerten, zu denen wir Verklemmbarkeit, Lebendigkeit und Determinismus der *Web Services* zählen. Die Verifikation dieser und anderer Eigenschaften stellt einen wichtigen Schritt zur Etablierung von *Web Services* dar.

Mit diesem Ziel arbeitet die Fachwelt an der Entwicklung einer Sprache zur Beschreibung von *Web Services*. Bis jetzt gibt es keinen Standard, lediglich zwei Kandidaten: die Sprachen *Web Service Choreography Interface* (*WS-CI*)[AAF+02] und *Business Process Execution Language for Web Services* (*BPEL*)[ACD+03]. In dieser Arbeit vergleichen wir beide Ansätze

miteinander und entwickeln eine Semantik für *WSCl*.

1.2 Umsetzung

Bei der Planung unserer Arbeit interessierten uns vor allem zwei Dinge: Welche Ergebnisse liegen im Bereich der Entwicklung einer Sprache zur Beschreibung von *Web Services* bereits vor? Welche Möglichkeiten bieten sich uns, eine entsprechende Semantik zu entwickeln?

In Kapitel 2 umreißen wir zunächst alle Grundlagen der Arbeit. Dazu betrachten wir den Formalismus der Petrinetze sowie die Sprachen *WSCl* und *BPEL*. Die beiden letztgenannten Sprachen bauen auf der Sprache *WSDL* auf, die wir ebenfalls betrachten werden. Wir wählen Petrinetze, um die Sprache *WSCl* zu modellieren, da sie Vorteile gegenüber anderen Modellierungsmethoden bieten: Sie sind mathematisch fundiert und betrachten sowohl dynamische als auch zustandsgebundene Aspekte des Modells.

Zwischen den beiden Sprachen *WSCl* und *BPEL* erkennen wir einen grundsätzlichen Unterschied: ihre jeweilige Sichtweise auf *Web Services*. Wir unterscheiden die lokale und die globale Sichtweise, die von uns betrachteten Sprachen setzen jeweils eine dieser Sichtweisen um. In Kapitel 3 werden wir auf die angesprochenen Sichtweisen und auf die beiden *Web-Service*-Beschreibungssprachen eingehen. Wir stellen sie jeweils einzeln vor und vergleichen sie anschließend miteinander.

Im Anschluss daran gehen wir im Kapitel 4 detailliert auf die Sprache *WSCl* ein und entwickeln für sie eine Petrinetz-basierte Semantik. Jeder Bestandteil dieser Semantik hat die Form eines Bausteines und repräsentiert einen Teil der Sprache *WSCl*. Wir beschreiben einleitend den generellen Aufbau eines Bausteines und definieren, wie jeweils zwei Bausteine zu verknüpfen sind. Die interessanten Eigenschaften der entstandenen Modelle können wir aufgrund der Verwendung von Petrinetzen als Modellierungsmethode computergestützt überprüfen. Die Bausteine wurden eigenschaftserhaltend aus *WSCl* erstellt. Damit sind wir in der Lage, Aussagen über Eigenschaften (z.B. Komposition und Austauschbarkeit) von *Web Services* anhand unserer Modelle zu treffen. Der hohe Detaillierungsgrad der Elemente sorgt aber schon bei kleineren Anwendungen für Unübersichtlichkeit. Deshalb stellen wir anschließend Möglichkeiten der vereinfachenden Darstellung vor. Ein einfaches, kapitelübergreifendes Beispiel verdeutlicht die dargelegten Inhalte. Dieses Beispiel umfasst zwei *Web Services*. Ein *Web Service* stellt eine Frage, der zweite antwortet ihm.

Am Ende des 4. Kapitels gehen wir konkret auf die Umwandlung unseres Beispiels in Petrinetz-Bausteine ein. Wir kombinieren mehrere dieser Bau-

steine zu einem Modell des *Web Service*. Mit Hilfe einer von uns implementierten Java-Applikation [JAVASUN] transformieren wir anschließend weitere Beispiel-*Web-Services* entsprechend der *WSCI*-Semantik in Petrinetze. Jedes so erstellte Petrinetz liegt in einem bestimmten Datenformat vor, das es uns ermöglicht, das Netz computergestützt mit Hilfe des ModelCheckers LoLA [Sch00] zu verifizieren. Anschließend gehen wir auf die Ergebnisse der Verifikation mit LoLA ein.

Im fünften Kapitel werden wir den Inhalt der Arbeit zusammenfassen und einen Ausblick auf weiterführende Forschungsmöglichkeiten geben.

Kapitel 2

Grundlagen

Im Mittelpunkt unserer Arbeit steht die Entwicklung einer Semantik für *WSCI*. Die notwendigen Grundlagen dafür legen wir im folgenden Kapitel. Wir werden Petrinetze und die Sprache *WSDL* jeweils kurz vorstellen und mit Beispielen veranschaulichen.

2.1 WSDL

2.1.1 Überblick

Die *Web Service Description Language (WSDL)* [CCMW01] ist eine *XML*-basierte Schnittstellenbeschreibungssprache. Mit ihr definieren wir die statische Schnittstelle eines *Web Service* zu seiner Umwelt.

Die Elemente von *WSDL* unterteilt man in einen abstrakten und in einen konkreten Teil. Der abstrakte Teil abstrahiert von Kommunikationsprotokollen und Datenformaten. In ihm können wir sprachunabhängige Typen, Nachrichten, Operationen und Port-Typen definieren. Diese Definitionen sind wiederverwendbar, können also für verschiedene Konkretisierungen dienen. Der konkrete Teil von *WSDL* benutzt diese abstrakten Definitionen. In ihm definieren wir ein konkretes Protokoll und ein konkretes Datenformat (*binding*). Wir definieren Kommunikationsendpunkte (*port*) und deren Zusammenfassung zu einem *Web Service (service)*. Das *binding* unterstützt bereits die folgenden Protokolle und Datenformate: *MIME* [FB96], *HTTP1.1 GET/POST* [FGM+99] und *SOAP 1.1* ([Mi03], [GHM+03], [HHK+03]). Zusätzlich unterstützt *WSDL* die Verwendung von *XML Schema Specification* ([TBMM01], [BM01]). Abbildung 2.1 zeigt die Verknüpfungen der Elemente aus *WSDL* anhand eines ER-Modells. Die linke Seite der Abbildung zeigt den konkreten, die rechte Seite den abstrakten Teil. Wie wir leicht er-

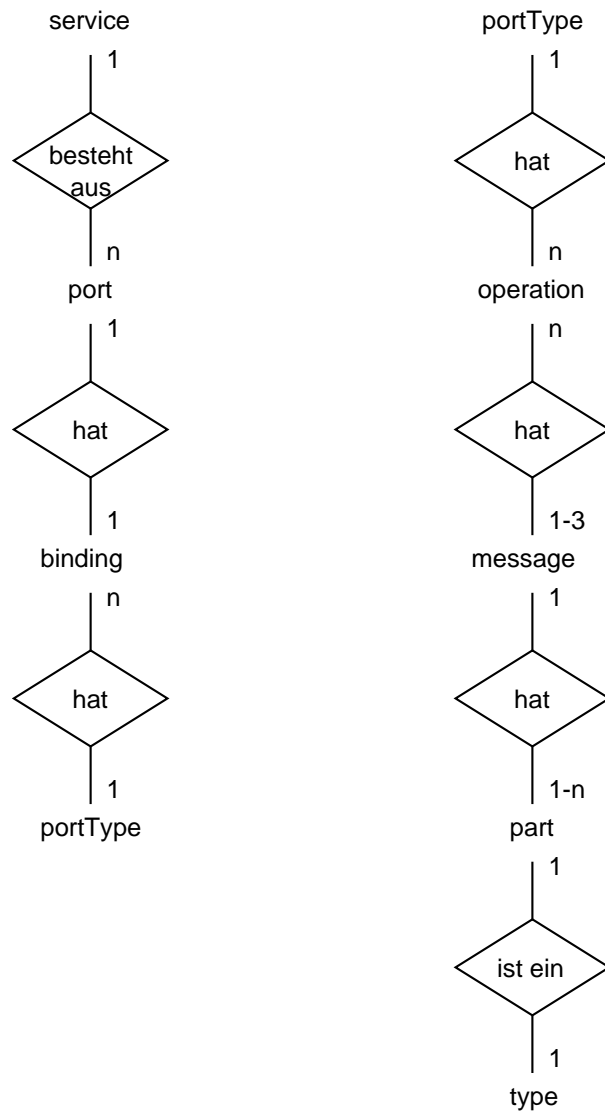


Abbildung 2.1: WSDL ER-Modell

kennen, stehen alle Elemente aus *WSDL* zueinander in Relation. Ein *service* besteht aus einer beliebigen Menge von *ports*. Zu jedem *port* gehört genau ein *binding*. Jedes *binding* besitzt genau einen *portType*. Ein *portType* wiederum kann mehreren *bindings* zugeordnet sein und enthält eine beliebige Anzahl an Operationen (*operation*). Eine *operation* enthält abhängig von ihrem Typ eine, zwei oder drei Nachrichten (*message*). Jede *message* kann in einer beliebigen Anzahl von Operationen verwendet werden und beinhaltet eine beliebige Anzahl von Elementen von beliebigem Typ.

WSDL definiert Nachrichtenformate und legt ihre Verwendung innerhalb der Kommunikation mit einem *Web Service* fest.

2.1.2 Beispiel

Das folgende Beispiel deutet den Aufbau für eine einfache Frage-Antwort-Situation an. Es wird uns auch in den folgenden Kapiteln begleiten. Ausführliche Beispiele finden wir in der Spezifikation für WSDL [CCMW01].

```

00<?xml version="1.0" ?>
01  <definitions name="Kommunikationsbeispiel" >
02
03    <types>
04      <schema targetNamespace="frage.xsd"
05        xmlns="http://www.w3.org/2000/10/XMLSchema" >
06        <element name="Frageinhalt" >
07          <complexType>
08            <all>
09              <element name="Text" type="string"/>
10            </all>
11          </complexType>
12        </element>
13        <element .../>
14      </schema>
15    </types>
16
17    <message name="Frage" >
18      <part name="body" element="Frageinhalt"/>
19    </message>
20
21    <message name="Antwort" >
22      <part ... >
23    </message>

```

```

24
25     <portType name=„AntwortPortType“>
26         <operation name=„AntwortAufFrage“>
27             <input message=„Frage“/>
28             <output message=„Antwort“/>
29         </operation>
30     </portType>
31
32     <binding name=„AntwortSoapBinding“
33         type=„AntwortPortType“>
34         <soap:binding style=„document“
35             transport=„http://schemas.xmlsoap.org/soap/http“/>
36         <operation name=„AntwortAufFrage“>
37             <soap:operation
38                 soapAction=„http://example.com/AntwortAufFrage“/>
39             <input .../>
40             <output .../>
41         </operation>
42     </binding>
43
44     <service name=„AntwortService“>
45         <documentation>My first service</documentation>
46         <port name=„AntwortPort“
47             binding=„AntwortSoapBinding“>
48             <soap:address
49                 location=„http://example.com/antwortservice“/>
50         </port>
51     </service>
52 </definitions>

```

Die wichtigsten Schlüsselwörter in diesem Beispiel sind fett hervorgehoben. Die Typen (*types*) werden von Zeile 3 bis Zeile 15 definiert. In Zeile 9 wird beispielsweise der Inhalt des Elements mit dem Namen Frageinhalt festgelegt. Er besteht aus einem Element namens Text vom Typ String. In Zeile 17 wird eine Nachricht (*message*) definiert. Sie hat genau einen Bestandteil, nämlich ein Element namens body vom eben definierten Typ Frageinhalt. In Zeile 25 wird ein *portType* definiert. Er enthält genau eine Operation (Zeile 26). Diese Operation bekommt ein Nachricht vom in Zeile 17 definierten Typ und beantwortet diese mit einer Nachricht vom in Zeile 21 definierten Typ. Das *binding* wird ab Zeile 32 definiert. Es wird dann im *service*, der ab Zeile 44 definiert ist, in der Definition des *ports* benutzt.

2.2 Petrinetze

Die Theorie der Petrinetze wurde in der Fachliteratur schon sehr ausführlich behandelt und bedarf daher nur einer kurzen Einführung. Der Name des Formalismus geht auf seinen Entwickler Carl Adam Petri [Pet62] zurück. Umfangreiche Abhandlungen sind auch in [Rei85], [Rei90] und [Sta90] enthalten.

Ein Petrinetz besteht aus einer endlichen Menge P von Plätzen p , einer endlichen Menge T von Transitionen t und einer endlichen Menge F von Flusskanten f . P und T sind disjunkt. Ein Petrinetz heißt zusammenhängend, wenn alle seine Plätze und Transitionen über Flusskanten miteinander verbunden sind. Plätze ermöglichen es, einen Zustand zu repräsentieren, Transitionen stellen die möglichen Zustandsveränderungen dar. Jedes Petrinetz ist ein gerichteter, bipartiter Graph. In ihm verbinden gerichtete Flusskanten aus F Elemente aus der Menge P mit Elementen aus der Menge T . Zusätzlich ordnen wir jeder Flusskante ein Element der positiven ganzen Zahlen N zu, welches die Anzahl an zu verbrauchenden bzw. zu erzeugenden Marken bestimmt. Diese Zahl nennen wir das *Gewicht* dieser Flusskante. Es gilt die Relation: $F \subseteq ((P \times T) \cup (T \times P)) \times N$. Für jede Flusskante $f \in F$ stellt $f = (a, b, n)$ dar, dass a der Ursprung, b das Ziel und n das Gewicht von f ist.

Eine Flusskante stellen wir graphisch durch einen Pfeil dar und visualisieren ihr Gewicht durch das entsprechende Symbol. Ist das *Gewicht* gleich 1, lassen wir die Beschriftung weg. Werden zwei Elemente durch zwei Flusskanten entgegengesetzter Richtung verbunden, können wir das durch einen Doppelpfeil darstellen, der in beide Richtungen zeigt. Einen Platz stellen wir mit Hilfe eines Kreises, eine Transition mit Hilfe eines Quadrates dar.

Mit den folgenden Ausführungen skizzieren wir die Funktionsweise der Petrinetze kurz: Plätze sind Speicher für Marken. Den Typ der verwendeten Marken können wir frei definieren. Er bestimmt den Informationsgehalt, den eine Marke in sich trägt. In unserem Fall beschränken wir uns auf die bloße Existenz der Marke. Bezüglich eines Platzes ist also lediglich die Anzahl der Marken von Interesse. Die Abbildung der Menge der Plätze auf die Menge der nicht-negativen ganzen Zahlen nennen wir *Markierung*.

Transitionen erzeugen und verbrauchen Marken auf Plätzen entsprechend dem *Gewicht* der sie verbindenden Flusskanten. Ein Platz p heißt *Vorplatz* einer Transition t , wenn es eine Flusskante $f \in F$ und ein $n \in N$ gibt mit $f = (p, t, n)$. Ein Platz p heißt *Nachplatz* einer Transition t , wenn es eine Flusskante $f \in F$ und ein $n \in N$ gibt mit $f = (t, p, n)$. Eine Transition *hat Konzession* (kann schalten), wenn auf jedem ihrer *Vorplätze* die über das Gewicht der entsprechenden Flusskante definierte erforderliche Anzahl an

Marken vorhanden ist (n). Der *Effekt* des Schaltens einer Transition t besteht darin, dass entsprechend der jeweiligen Flusskante von jedem Vorplatz von t genau n Marken verbraucht und auf jedem Nachplatz von t genau n Marken erzeugt werden (im einfachsten Fall jeweils eine Marke).

Mit diesen Mitteln können wir die möglichen Verhaltensvarianten eines Netzes in Kombination mit einer Anfangsmarkierung berechnen. Dazu gehören unter anderem die *Erreichbarkeit*, die *Verklemmbarkeit* und die *Lebendigkeit* ([Rei85], [Sta90]). *Erreichbarkeit* bezeichnet die Möglichkeit des Eintretens eines Zustandes. *Verklemmbarkeit* ist die Möglichkeit des Erreichens eines Zustandes, von dem aus keine Transition *Konzession hat*. Die Eigenschaft *Lebendigkeit* besagt, dass für jede Transition von jedem Zustand aus ein Zustand *erreichbar* ist, bei dem sie *Konzession hat*. Ein Zustand, für den keine Transition *Konzession hat*, heißt *toter Zustand*. Auf solche Zustände werden wir uns bei der Auswertung der abschließenden Tests konzentrieren.

Bei der Verwendung von Petrinetzen können wir ein beliebiges Abstraktionsniveau wählen. Es gibt keine Festlegung, wofür die benutzten Marken, Plätze oder Transitionen stehen sollen. Das ermöglicht uns beispielsweise die schrittweise konkreter werdende Modellierung eines komplexen Systems. Einfache Petrinetze, deren Marken keine zusätzlichen Informationen tragen, bezeichnen wir auch als Low-Level-Petrinetze. Entsprechend bezeichnen wir ein Petrinetz mit komplexen Marken als High-Level-Petrinetz. Die Färbung von Marken stellt eine einfache Methode dar, Marken unterscheidbar zu machen. Durch die Einbeziehung von Plätzen und Transitionen haftet den Petrinetzen nicht der Nachteil anderer Modellierungsmethoden an, sich von vornherein entweder auf die Betrachtung von Zuständen oder Zustandsübergängen festlegen zu müssen.

Petrinetze besitzen eine einfache graphische Repräsentation. Sie basieren auf Graphen und bieten somit eine theoretisch-fundierte Grundlage für die Simulation, Analyse und Validierung beliebig komplexer Modelle. Abbildung 2.2 zeigt eine Transition, einen Platz und ein Petrinetz mit der Ausgangsmarkierung, die dem Platz P1 genau eine Marke zuweist.

□ Transition

○ Platz

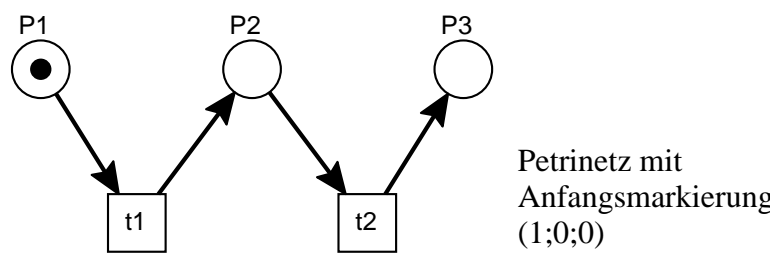


Abbildung 2.2: Petrinetz

Kapitel 3

Web Services

In diesem Kapitel befassen wir uns mit dem Vergleich der lokalen und der globalen Sichtweise auf das Verhalten von *Web Services* und den sie exemplarisch umsetzenden Sprachen *WSCI* und *BPEL* (auch *BPEL4WS*).

3.1 Sichtweisen

Wir haben die Möglichkeit, das Verhalten von verteilten und kommunizierenden Systemen (von *Web Services*) von verschiedenen Standpunkten aus zu beschreiben. Dabei können wir uns auf das Verhalten eines einzelnen *Web Service* konzentrieren oder auf das Zusammenspiel aller beteiligten *Web Services* betrachten. Die lokale Sichtweise beschränkt sich auf einen *Web Service* und betrachtet die direkten Kommunikationspartner - im Folgenden Partner genannt - nur oberflächlich. Die globale Sichtweise hingegen betrachtet alle in einem System enthaltenen *Web Services*.

3.1.1 Lokale Sichtweise

Die lokale Sichtweise auf die Interaktion von *Web Services* konzentriert sich auf genau einen *Web Service*. Wir besitzen alle relevanten Informationen über ihn, insbesondere auch die über sein Verhalten. Dazu gehören der Kontroll- und Datenfluss, also die Reihenfolge der Verarbeitungsanweisungen bzw. die Reihenfolge der Zustände der Variablen. Ein so definierter *Web Service* kann hinsichtlich aller relevanten Informationen untersucht werden.

Das Verhalten der Partner des betrachteten *Web Service* wird ausgeblendet. Alle *Web Services*, zu denen diese evtl. noch Kontakt haben, werden ignoriert. Uns interessiert nur, welche Operationen die Partner dem betrachteten *Web Service* direkt zur Verfügung stellen. Dabei gehen wir implizit

davon aus, dass die Partner die Anforderungen des betrachteten *Web Service* erfüllen. Es kann also nicht zu einer Verklemmung aufgrund eines Fehlers in der Definition eines Partners kommen.

3.1.2 Globale Sichtweise

Die globale Sichtweise betrachtet alle in dem kommunizierenden System enthaltenen *Web Services*. Wir kennen die Verhaltensbeschreibung eines jeden beteiligten *Web Service*. Somit können wir die Interaktion aller Beteiligten untersuchen. Wir können mögliche Widersprüche in den Verhaltensdefinitionen aufdecken und darüber hinaus untersuchen, ob ein *Web Service* einen anderen *Web Service* in seinen Anwendungsgebieten ersetzen kann.

Allerdings geht die globale Sichtweise nicht sehr detailliert auf die Interna der beteiligten *Web Services* ein. Die Beschreibung des Kontrollflusses ist für diese Art der Betrachtung ausreichend. Von weiteren Informationen wird abstrahiert. Damit verlieren wir die Möglichkeit, von diesen Informationen abhängige Eigenschaften zu überprüfen.

3.1.3 Vergleich

Jede der beiden Sichtweisen auf die Interaktion von *Web Services* hat Vor- und Nachteile und ist jeweils für eine andere Anwendung günstig. Wir wählen daher die Art der Sichtweise entsprechend den Anforderungen der jeweiligen Verhaltensbeschreibung.

In Fällen, die eine dritte Art der Verhaltensbeschreibung erfordern, können wir Teile der bereits bekannten Sichtweisen zu einer neuen Sichtweise kombinieren. So können wir die globale und die lokale Sichtweise zusammenführen, um die Verhaltensbeschreibung aller beteiligten *Web Services* und die detaillierte Beschreibung des betrachteten *Web Service* zu erhalten. Weiterhin bietet sich die Möglichkeit, die lokalen Sichtweisen aller betrachteter *Web Services* in einem globalen Modell zu vereinigen. Auf diese Art gewinnen wir aus der Kombination beider Ansätze neue Möglichkeiten der Spezifikation und der Überprüfung von Eigenschaften.

Auf dem Gebiet der Kombinationsmöglichkeiten dieser beiden Ansätze gibt es noch zahlreiche weitere interessante Fragestellungen. Diese sollen jedoch Bestandteil zukünftiger Arbeiten sein.

3.2 Sprachen

Im Folgenden werden wir die beiden Sprachen *BPEL* und *WSCI* vergleichen. Jede dieser Sprachen ist ein Vertreter einer der vorgestellten Sichtweisen auf *Web Services*. Die Sprache *BPEL* ist aufgrund ihrer Konzentration auf die Verhaltensbeschreibung eines einzelnen *Web Service* der Vertreter der lokalen Sichtweise. Entsprechend steht *WSCI* als Vertreter für die globale Sichtweise.

3.2.1 BPEL

Die *Web-Service*-Beschreibungssprache *BPEL* (Business Process Execution Language for Web Services) beschreibt das Verhalten eines *Web Service* und die Interaktion mit dessen Partnern [ACD+03]. Sie ist als XML-basierte Sprache aus Microsofts *XLANG* [Tha01] und IBM's *Web Service Flow Language (WSFL)* [Ley01] hervorgegangen und basiert auf der statischen Schnittstellenbeschreibung eines *Web Service* mit *WSDL*. *BPEL* ergänzt diese um die Verhaltensbeschreibung des *Web Service*.

BPEL setzt das Konzept der lokalen Sichtweise um. Die Beschreibung sämtlicher Partner des betrachteten *Web Service* wird auf die Beschreibung ihrer Schnittstelle reduziert. Die Sprache *BPEL* unterscheidet dabei grundsätzlich zwei verschiedene Arten von *Web Services*: den ausführbaren Prozess (*executable process*) und den abstrakten Prozess (*business protocol*). Ein ausführbarer Prozess beinhaltet alle bekannten, den Ablauf beeinflussenden Fakten und Daten. Wir können alle Abläufe vollständig simulieren, validieren und analysieren. Ein abstrakter Prozess hingegen stellt den Teil an Informationen dar, den Unternehmen bereit sind, an außen stehende Partner weiterzureichen. Der Nutzer dieser abstrakten Prozesse sieht nur die sogenannten externen Aktivitäten. Das sind die Aktivitäten, die für die Kommunikation nach außen zuständig sind. Interne Details bleiben verborgen, von ihnen wird abstrahiert. Somit kann *BPEL* entsprechend der lokalen Sichtweise beide Typen der betrachteten *Web Services* darstellen: den betrachteten *Web Service* und seine Partner.

Die Verknüpfung eines *Web Service* mit seinen Partnern ist typischerweise *peer-to-peer*. Ein *Web Service* in einem Kommunikationsprozess mit seinem Partner ist also sowohl Anbieter als auch Nutzer von angebotener Funktionalität. Wir verknüpfen *Web Services*, indem wir ihre kommunizierenden Operationen einander zuordnen. Dadurch definieren wir die Beziehung eines *Web Service* zu seinen Partnern. Das *BPEL*-Element *partnerLinkType* verknüpft zwei *portTypes* und definiert den Typ einer Verknüpfung. Die Elemente *partnerLink* und *partner* legen konkrete Verknüpfungen zwischen Operationen von *Web Services* fest.

Die Definition eines *Web Service* in *BPEL* setzt sich aus mehreren Elementen zusammen. Diese beschreiben den Kontrollfluss des *Web Service*. Aktivitäten haben einen großen Anteil daran, wir unterscheiden zwischen Basisaktivitäten und komplexen Aktivitäten. Basisaktivitäten stellen die nicht mehr sinnvoll verfeinerbaren Teile des Verhaltens dar (z.B. den Empfang einer Nachricht), komplexe Aktivitäten verbinden diese und definieren ihre kausale Ordnung.

Als Basisaktivitäten gelten *invoke* für das synchrone oder asynchrone Verschicken von Nachrichten, *receive* für deren Empfang und *reply* für das Versenden der entsprechenden Antwortnachricht. Die Aktivität *empty* macht nichts und *wait* lässt Zeit vergehen. *assign* regelt die Zuweisung von Werten zu Variablen. *throw* erzeugt einen Fehler, *terminate* beendet das Verhalten im betroffenen Gültigkeitsbereich und *compensate* kompensiert bereits vollständig ausgeführtes Verhalten.

Die komplexen Aktivitäten enthalten jeweils mindestens eine weitere Aktivität. Sie definieren kein beobachtbares, kommunikatives Verhalten, sondern steuern den Ablauf der enthaltenen Aktivitäten. Zu ihnen zählen *sequence* und *flow* für die sequentielle bzw. parallele Abarbeitung aller enthaltenen Aktivitäten. Innerhalb von *flow* können wir *links* einfügen, die zusätzliche Kausalität definieren. Die Aktivität *while* definiert die wiederholte Abarbeitung der enthaltenen Aktivität und *switch* und *pick* definieren die Auswahl einer Aktivität aus mehreren Aktivitäten auf Basis von Bedingungen bzw. von Ereignissen.

Jede Aktivität und jede Variable ist in einem Gültigkeitsbereich definiert, dem *scope*. Diese können rekursiv weitere *scopes* enthalten, in jedem *scope* können wir Ereignis-, Fehler- und Kompensations-Handler definieren. Jeder dieser Handler definiert das für ein eintretendes Ereignis entsprechende Verhalten. Der Ereignis-Handler reagiert auf eintreffende Nachrichten und auf überschrittene Zeitvorgaben. Der Fehler-Handler reagiert auf auftretende Fehler, der Kompensations-Handler macht die Auswirkungen eines fehlerfrei ausgeführten *scopes* rückgängig. Letzteres ist für die Umsetzung von Transaktionen wichtig.

Da *BPEL* die Ausführung von genau einem Prozess betrachtet, wird implizit von genau einer Instanz dieses Prozesses ausgegangen. Die Betrachtung mehrerer Instanzen innerhalb der Kommunikation mit anderen Prozessinstanzen ist also nicht vorgesehen. Weitere Ausführungen zu *BPEL* sind nachzulesen in [St04] und [Mar03]. *BPEL* wird z.B. durch das Werkzeug *LTSA* [LTSA] mit dem dazugehörigen *BPEL* plug-in [Fos04] unterstützt. Weitere solcher Werkzeuge werden beispielsweise von der PIKOS GmbH angeboten.

3.2.2 WSCI

Die Sprache *WSCI* (Web Service Choreography Interface) beschreibt die Interaktion zwischen *Web Services* [BHM+04]. Sie wurde hauptsächlich von Intalio [WebInt], Sun [WebSun], BEA [WebBEA] und SAP [WebSAP] entwickelt. Im Juni 2002 wurde sie dem *World Wide Web Consortium (W3C)* [WebW3C] vorgelegt [AAF+02], im August 2003 erschien dort ein Working Draft [ABPR03] zu *WSCI*. Mitte 2004 gab es beim *W3C* zahlreiche andere Projekte zur Spezifikation von verteilten Prozessen mit *Web Services*. Die Sprache *WS-CDL* [KBR04] beschäftigt sich beispielsweise mit der Choreographie von *Web Services*. Sie wird in der *W3C Web Services Choreography Working Group* [WSCWG] bearbeitet. Mit der Sprache *BPEL* haben wir einen direkten Konkurrenten von *WSCI* bereits näher vorgestellt. Das zeigt, dass *Web Services* zu einem sich schnell entwickelnden Sektor gehören. Im Hinblick auf die Motivation dieser Arbeit können wir sagen, dass in der oft sehr praxisnahen Entwicklung von Sprachen theoretische Fundierungen von großem Vorteil sind. *WSCI* baut auf *WSDL* auf und ergänzt diese um die Verhaltensbeschreibung des *Web Service*. Sie setzt die Anforderungen der globalen Sichtweise um. Das heißt, dass sie den Kontrollfluss aller beteiligten *Web Services* betrachtet.

WSCI beschreibt das sichtbare Verhalten von *Web Services*. Dazu gehören auch temporale und logische Abhängigkeiten zwischen den ein- und ausgehenden Nachrichten. Die Reihenfolge der Nachrichten und deren Zusammenhang wird ebenso mit einbezogen, wie die Ausnahmebehandlung und das Transaktionsmanagement. Der grundlegende Aufbau von *WSCI* ähnelt dem von *BPEL*. Es gibt jedoch auch Unterschiede. Darauf gehen wir an dieser Stelle nur knapp ein, da wir uns im folgenden Kapitel ausführlich mit dieser Sprache beschäftigen werden.

Eine komplette Übersicht aller partizipierenden *Web Services* bekommen wir durch das Element *model*. Dort werden alle beteiligten *Web Services* referenziert und ihre Operationen miteinander verknüpft. Die Verhaltensspezifikation jedes einzelnen *Web Service* ist im Element *interface* enthalten. Ein *interface* kann mehrere Prozesse enthalten. Das bedeutet auch, dass in einem *Web Service* gleichzeitig mehrere Prozessinstanzen existieren können, die mit verschiedenen, parallel ablaufenden Prozessinstanzen kommunizieren.

In *WSCI* wird nicht beschrieben, welche Prozessinstanz zuerst ausgeführt werden soll. Wir unterscheiden lediglich zwischen Prozessen, die durch den Erhalt einer Nachricht gestartet werden und Prozessen, die ohne einen Impuls von außen ablaufen können. *WSCI* gibt also nicht zwangsläufig eine konkret auszuführende Prozessinstanz vor, sondern beschreibt eine Menge von ausführbaren Prozessen.

Ebenso wie in *BPEL* gibt es in *WSCI* atomare Aktivitäten zur Kommunikation (*action*) und Elemente, die nichts tun (*empty*), Zeit vergehen lassen (*delay*) oder einen Fehler erzeugen (*fault*). Eine komplexe Aktivität ermöglicht die parallele oder sequentielle Ausführung der enthaltenen Aktivitäten (*all, sequence*), sowie die Auswahl von Aktivitäten anhand von Bedingungen oder eintretenden Ereignissen (*switch, choice*) oder definiert Schleifen (*while, until*). Sie bietet ebenfalls die Möglichkeit, die enthaltenen Aktivitäten für jedes Element einer bestimmten Menge jeweils einmal auszuführen (*foreach*).

Ein Gültigkeitsbereich in *WSCI* heißt *context*. Mehrere *contexts* sind in einem *process* enthalten, von denen wiederum mehrere in einem *interface* zusammengefasst werden können. Ebenso wie *BPEL* definiert *WSCI* Konstrukte für Fehlerbearbeitung und Transaktionsmanagement.

Bis zum Jahr 2004 gab es keine technische Unterstützung für *WSCI*. Die Sprache baut jedoch auf Standards wie *WSDL* und *XML* auf. Besonders für *XML* sind viele Werkzeuge verfügbar.

3.2.3 Vergleich

Jede Sprache repräsentiert die ihr zugeordnete Sichtweise offensichtlich ausreichend. Die Details der Modellierung eines *Web Service* weichen jedoch an einigen Stellen voneinander ab. Wir machen die Unterschiede jetzt deutlicher.

Die Sprache *BPEL* betrachtet den Datenfluss des betrachteten *Web Service* vollständig. *WSCI* konzentriert sich auf den Kontrollfluss, ermöglicht aber auch die ausführliche Betrachtung des Datenflusses. Sowohl *WSCI* als auch *BPEL* sind in viele Richtungen erweiterbar. Im Bereich der Daten ist die Darstellung durch *BPEL* prinzipiell ausführlicher, *WSCI* kann dies durch entsprechende Erweiterungen aber ebenfalls erfüllen.

BPEL ermöglicht es weiterhin, innerhalb von parallel auszuführenden Aktivitäten *links* zu definieren, die zusätzliche Kausalität einbringen. Dieses Konstrukt ist nicht in *WSCI* enthalten. *links* verkomplizieren den Ablauf. Sie müssen sehr genau untersucht werden, da sie die Quelle für das Erreichen zahlreicher ungewollter, aber doch erreichbarer Zustände sind. Die Darstellung eines *links* können wir in *WSCI* durch den Versand einer Nachricht von der entsprechenden Prozessinstanz an sich selbst nachbilden.

In *BPEL* wird ein Gültigkeitsbereich (*scope*) als Aktivität behandelt. Die Definition eines gekapselten Bereiches hat also an der gleichen Stelle zu erfolgen wie seine Benutzung. So definierte Gültigkeitsbereiche sind nicht wieder verwendbar. Die Definition in *WSCI* hat diesen Mangel nicht. Ein *process* wird dort innerhalb seines Vaterkontextes definiert, unabhängig von der Stelle, an der er gebraucht wird. Seine Ausführung hängt davon ab, von welcher

Stelle er gerufen wird. Dies geschieht über *call* oder *spawn*. An dieser Stelle wird die Intention von *WSCI* sichtbar, das Verhalten unabhängig von der Anzahl von Prozessinstanzen zu definieren.

BPEL hingegen hat den Makel, nur die gerade betrachtete Prozessinstanz beschreiben zu können. Das wirkt sich insbesondere bei der Kombination von Verhaltensbeschreibungen verschiedener *Web Services* nachteilig aus. In diesem Punkt hebt sich *WSCI* von *BPEL* ab, da *BPEL* nicht alle Konstrukte aus *WSCI* sinngemäß nachbilden kann. Wenn wir zum Beispiel eine dynamische Anzahl von Instanzen eines Partner-*Web-Services* parallel ablaufen lassen, stoßen wir mit *BPEL* aufgrund der eingeschränkten Wiederverwendbarkeit an die Grenzen der Sprache.

Das Fazit des Vergleiches der betrachteten Modellierungsmethoden ist, dass beide ihre jeweilige Sichtweise erfüllen. Alle *BPEL*-Modelle sind durch *WSCI*-Modelle darstellbar. Der umgekehrte Fall trifft nicht zu.

Wir werden uns im Folgenden nur noch mit *WSCI* auseinandersetzen, speziell mit der für diese Sprache entwickelten Semantik.

Kapitel 4

Entwicklung einer Semantik zu WSCI

In diesem Kapitel entwickeln wir eine Semantik für die Sprache *WSCI*. Mit dieser werden wir zum einen Unklarheiten und Widersprüche in der Spezifikation zu *WSCI* aufdecken, zum anderen können wir damit die in der Sprache *WSCI* beschriebenen *Web Services* bezüglich ihrer Eigenschaften untersuchen. Wir werden jedes durch die Spezifikation von *WSCI* vorgegebene, relevante Element und jede wichtige, implizit gegebene Information durch einen Baustein modellieren. Alle Bausteine zusammen bilden die entwickelte *WSCI*-Semantik. Die Abbildung von *WSCI* auf die Bausteine ist eigenschaftserhaltend. Somit haben wir eine Methode gewonnen, die Eigenschaften von Modellen in *WSCI* auf Basis der entwickelten Bausteine zu überprüfen.

4.1 Bausteine

Im Folgenden stellen wir den Aufbau der verwendeten Bausteine und deren Verknüpfung vor.

Definition 1 (Baustein) *Ein Baustein ist ein zusammenhängendes Petri-netz.*

Definition 2 (Schnittstelle eines Bausteines) *Sei M_{PT} die Menge der Plätze und Transitionen eines Bausteines. Die Schnittstelle S dieses Bausteines ist eine Teilmenge von M_{PT} .*

Die Schnittstelle eines Bausteines enthält genau diejenigen Elemente, die für die Verknüpfung mit anderen Bausteinen vorgesehen sind. Wir heben sie

visuell durch einen den Baustein umschließenden Rahmen hervor. Die Menge der Elemente, die auf diesem Rahmen liegen, bilden die Schnittstelle des Bausteines. Innerhalb des Rahmens liegen die Plätze und Transitionen, welche die internen Elemente des Bausteines darstellen. Diese werden nur mit anderen Elementen desselben Bausteines verknüpft. Alle außerhalb des Rahmens dargestellten Objekte gehören nicht zum betrachteten Baustein und

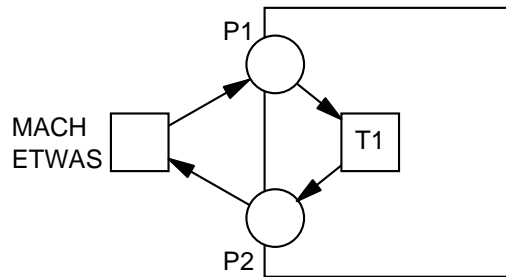


Abbildung 4.1: Baustein allgemein

sollen lediglich die Verknüpfung mit Schnittstellenelementen anderer Bausteine verdeutlichen. Sie werden nur bei komplizierten Zusammenhängen zur besseren Übersichtlichkeit dargestellt. Abbildung 4.1 zeigt die Darstellung eines Bausteines. Der Baustein umfasst $P1$, $P2$ und $T1$. Die Plätze $P1$ und $P2$ gehören zu der Schnittstelle des Bausteines. Die außerhalb des Rahmens liegende Transition nennt den Namen der mit $P1$ und $P2$ zu verknüpfenden Transition (MACH ETWAS).

Definition 3 (Subschnittstelle eines Bausteines) Sei S die Schnittstelle eines Bausteines. Jede Menge $S_U \subseteq S$ heißt Subschnittstelle dieses Bausteines.

Mit Hilfe von Subschnittstellen definieren wir eindeutig, welche Transitionen mit welchen Plätzen im Zuge der Verknüpfung von Bausteinen verbunden werden. Dies soll zu größerer Übersichtlichkeit führen. Eventuelle Ausnahmen erklären wir am konkreten Modell. Jeder Baustein wird mit mehreren anderen Bausteinen verknüpft. Die detaillierte Verknüpfung von Transitionen und Plätzen wird mit Hilfe ihrer Namen beschrieben. Die folgenden Definitionen legen die Benennung von Schnittstellenelementen fest.

Definition 4 (Name eines Platzes) Der Name eines Platzes hat die Struktur $I(F)$, wobei I den informalen und F den formalen Teil des Namens darstellen. Der formale Teil F des Namens besteht aus mindestens einem Bezeichner. Mehrere Bezeichner werden durch einen Schrägstrich getrennt.

Definition 4 legt die Benennung von Plätzen fest. Da ein Platz innerhalb einer Schnittstelle mitunter verschiedene Funktionen übernimmt, kann er entsprechend mehrere Bezeichner erhalten. Ein Bezeichner identifiziert einen Platz innerhalb der Verknüpfung zweier Bausteine eindeutig. Lediglich bei dem Baustein *Lebenszykluskontext*¹ weichen wir an einer Stelle davon ab. Dort wählen wir aus Gründen der Übersichtlichkeit eine andere Bedeutung für verschiedene Bezeichner eines Platzes. Die Bezeichner der Plätze bestehen aus einem Buchstaben, gefolgt von einer Zahl. Die Buchstaben unterteilen die Elemente der Schnittstellen in verschiedene Kategorien, die Zahlen dienen nur ihrer Unterscheidung. Hierbei steht der Buchstabe *I* für die auszuführende Initialisierung der Prozessinstanz, *K* für das Stoppen (kill) und *E* für das Beenden der Instanz. Mit dem Buchstaben *L* bezeichnen wir Elemente, die zwar zur Steuerung des Lebenszyklus gehören, aber weder *I*, *K* noch *E* eindeutig zuzuordnen sind. *X* bezeichnet die zur Ausführung (execution) der Prozessinstanz benötigten Elemente, *D* charakterisiert alle Elemente zum Senden von Nachrichten (data). Mit dem Buchstaben *H* bezeichnete Elemente werden mit EreignisHandlern oder mit dem *Ereigniskontext*² verknüpft.

Definition 5 (Name einer Transition) *Der Name einer Transition hat die Struktur $IF_1\dots F_k$ ($k \in \mathbb{N}$), wobei I den informalen Teil und jedes F_i ($i = 1, \dots, k; k \in \mathbb{N}$) einen formalen Teil des Namens darstellt. Jeder formale Teil besteht aus Bezeichnern für verschiedene Plätze, sie trennenden Kommas, einem Minuszeichen und sie umschließenden Klammern.*

Jeder formale Teil innerhalb der Bezeichnung einer Transition bezeichnet die mit dieser Transition zu verknüpfenden Vor- und Nachplätze. Alle vor dem Minuszeichen stehenden Bezeichner (durch Kommas getrennt) stellen Vorplätze dar, alle danach stehenden Bezeichner die Nachplätze. Innerhalb eines Namens für eine Transition kann es mehrere formale Teile geben. Von diesen wird entsprechend des Typs des anzuschließenden Bausteines ein formaler Teil ausgewählt. Treten mehrere formale Teile auf, so unterscheiden wir sie anhand eines vorangestellten Buchstabens. Zur Verfügung stehen vier unterschiedliche Buchstaben. Die Buchstaben *A*, *H*, *K* und *P* stehen für die Verknüpfung mit einer Aktivität³, mit einem EreignisHandler⁴, mit einem Kontext-Element (*Lebenszykluskontext*⁵ oder *Ereigniskontext*⁶) bzw. mit dem

¹siehe Seite 33

²siehe Seite 40

³siehe ab Seite 49

⁴siehe ab Seite 40

⁵siehe Seite 33

⁶siehe Seite 40

*Lebenszyklusprozess*⁷. Falls ein in dem Namen einer Transition vorkommender Platzbezeichner nicht in der entsprechenden Subschnittstelle vorhanden ist, wird für ihn in dieser Verknüpfung keine Flusskante erzeugt. Das kann z.B. der Fall sein, wenn eine Transition zu mehreren Subschnittstellen gehört und dementsprechend mit Plätzen verschiedener Bausteine verbunden wird.

Definition 6 (Verknüpfung zweier Bausteine) *Ein Baustein B_1 wird mit einem Baustein B_2 verknüpft, indem man die dafür vorgesehene Subschnittstelle aus B_1 mit der dafür vorgesehenen Subschnittstelle aus B_2 verknüpft. Dies geschieht, indem alle Transitionen der beiden Subschnittstellen entsprechend ihrer Namen mit den zugehörigen Plätzen der jeweils anderen Subschnittstelle durch Flusskanten verbunden werden.*

Definition 6 legt die Verknüpfung von Bausteinen fest. Zwei Bausteine werden über jeweils genau eine ihrer Subschnittstellen miteinander verknüpft. Bei der Verknüpfung zweier Bausteine bleiben ihre Schnittstellen erhalten. Bausteine bleiben also auch dann eine Einheit für sich, wenn sie mit anderen Bausteinen verknüpft wurden. Sie werden nicht miteinander verschmolzen.

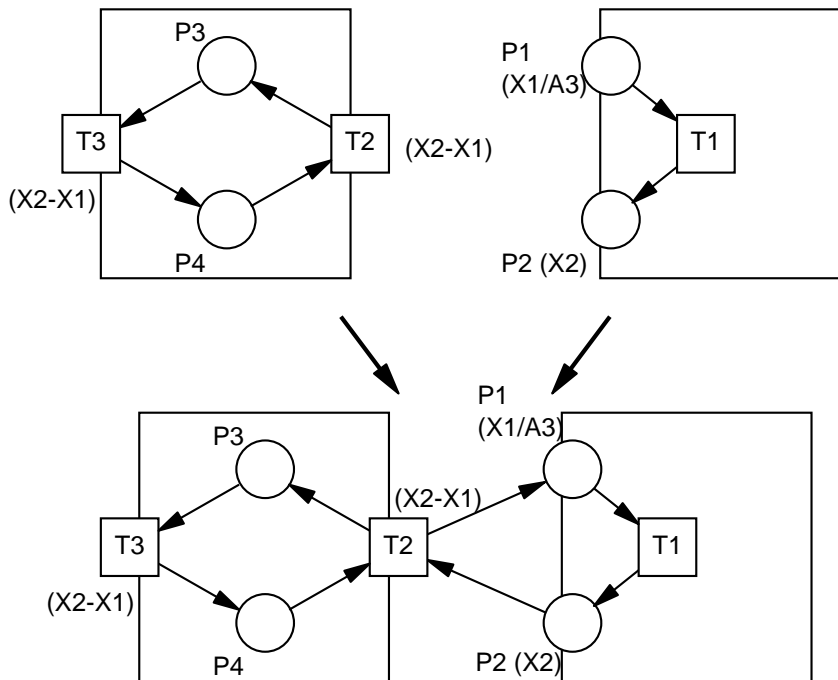


Abbildung 4.2: Verknüpfung von Bausteinen

⁷siehe Seite 30

Entsprechend der Definition von Flusskanten in einem Petrinetz werden nur Transitionen mit Plätzen und Plätze mit Transitionen verbunden. In Abbildung 4.2 werden die beiden oberen Bausteine verknüpft, das Ergebnis der Verknüpfung ist darunter abgebildet. Der rechte Baustein besitzt genau eine Schnittstelle. Sie entspricht der gesamten Schnittstelle des Bausteines. Die für die Verknüpfung der Bausteine verwendete Schnittstelle des linken Bausteines enthält nur die Transition $T2$. Bei dieser Verknüpfung interessieren uns also nur die Namen aus dieser Schnittstelle. Dementsprechend wird der Name von $T3$ nicht berücksichtigt. $P1$ besitzt die beiden Bezeichner $X1$ und $A3$, $P2$ besitzt den Bezeichner $X2$. Die Transition $T2$ besitzt den formalen Teil $(X2 - X1)$. Der Platz $P1$ wird hier also durch seinen Bezeichner $X1$ definiert, $T2$ erhält einen zusätzlichen Vorplatz mit dem Bezeichner $X2$ und einen zusätzlichen Nachplatz mit dem Bezeichner $X1$.

4.2 Hierarchie der Elemente aus WSCI

WSCI basiert auf den in *WSDL* definierten Elementen. Daher ist das Wurzelement jeder *WSCI*-Datei *wsdl:definitions*. Generell ist *WSDL* keine zwingende Voraussetzung. *WSCI* könnte auf jeder Sprache basieren, welche die Ausdrucksstärke von *WSDL* besitzt. Zur Zeit ist das aber nur *WSDL* selbst.

Unter dem Wurzelement liegen die vier top-level Elemente aus *WSCI*. Dazu gehören *interface*, *selector*, *correlation* und *model* (siehe Abbildung 4.3). Alle anderen Elemente dürfen nur in diesen vier Elementen definiert werden. Für jedes der vier *WSCI* top-level Elemente bilden die verwendeten Namen einen eigenen Namensraum. So dürfen z.B. Namen aus *interface* genauso lauten wie Namen aus *correlation*. Die qualifizierte, namensraumübergreifende Bezeichnung für ein Element erhalten wir durch Voranstellen des Namensraumes.

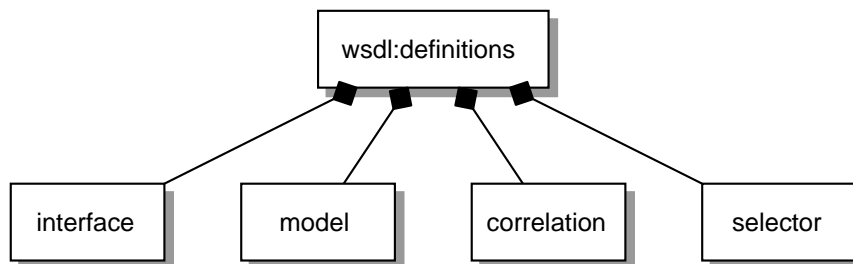


Abbildung 4.3: Top-Level Elemente

Die vier top-level Elemente aus *WSCI* referenzieren sich gegenseitig. Das top-level Element *model* verbindet die Schnittstellen der verwendeten *Web*

Services. Diese werden jeweils über ein Element *interface* dargestellt. Innerhalb eines *interface* werden mehrere Prozesse definiert. Ein Prozess wird über ein Element vom Typ *process* dargestellt, er wird immer in einem Kontext ausgeführt und kapselt die in ihm enthaltenen Aktivitäten. Diese sind in atomare und komplexe Aktivitäten unterteilt. Erstere führen die eigentlichen Aktionen aus, während letztere den Ablauf steuern. Kommunizierende Aktivitäten werden durch *connect* verknüpft. Das Element *connect* liegt direkt unter dem Element *model*. Die Abbildungen 4.4 und 4.5 visualisieren diese Zusammenhänge.

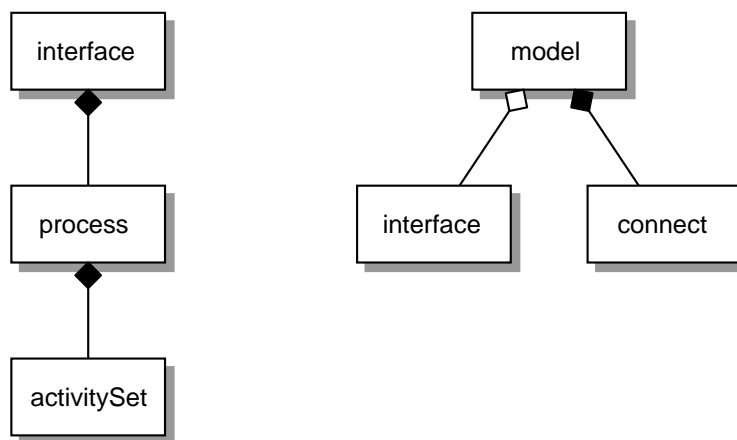


Abbildung 4.4: Komposition von Web Services

Das hier gezeigte Element *activitySet* stellt eine Menge von Aktivitäten (*choice of activities*) mit optionalem Kontext dar (siehe Abbildung 4.5). Ein Kontext kann Eigenschaften (*property*), Ausnahmebehandlung (*exception*) und Transaktionsmanagement (*transaction*) beinhalten. Innerhalb des Kontextes können beliebig viele Prozesse eingefügt werden. Jeder Prozess enthält eine Menge von Aktivitäten (*activitySet*). Das ermöglicht es uns, beliebig viele Prozesse und Kontexte ineinander zu schachteln. In Abbildung 4.6 sehen wir diesen Zusammenhang exemplarisch. *Prozess1* enthält *Kontext1*, welcher wiederum *Prozess2* und zwei weitere Kontexte enthält.

Weiterhin ist es wichtig, zwischen einem Prozess und einer Prozessinstanz zu unterscheiden. Ein *Web Service* kann parallel mit mehreren Partnern kommunizieren. Dabei kann er auch parallel die gleichen Typen von Anfragen bearbeiten. Er erzeugt dann für jede Anfrage eine Instanz desselben Prozesses. Wir haben diesen Fakt bereits beim Vergleich von *WSCI* und *BPEL* hervorgehoben, gehen hier aber noch ein mal darauf ein, da gerade die Instanzbildung für einen Prozess einen wesentlichen Aspekt dieser Arbeit ausmacht.

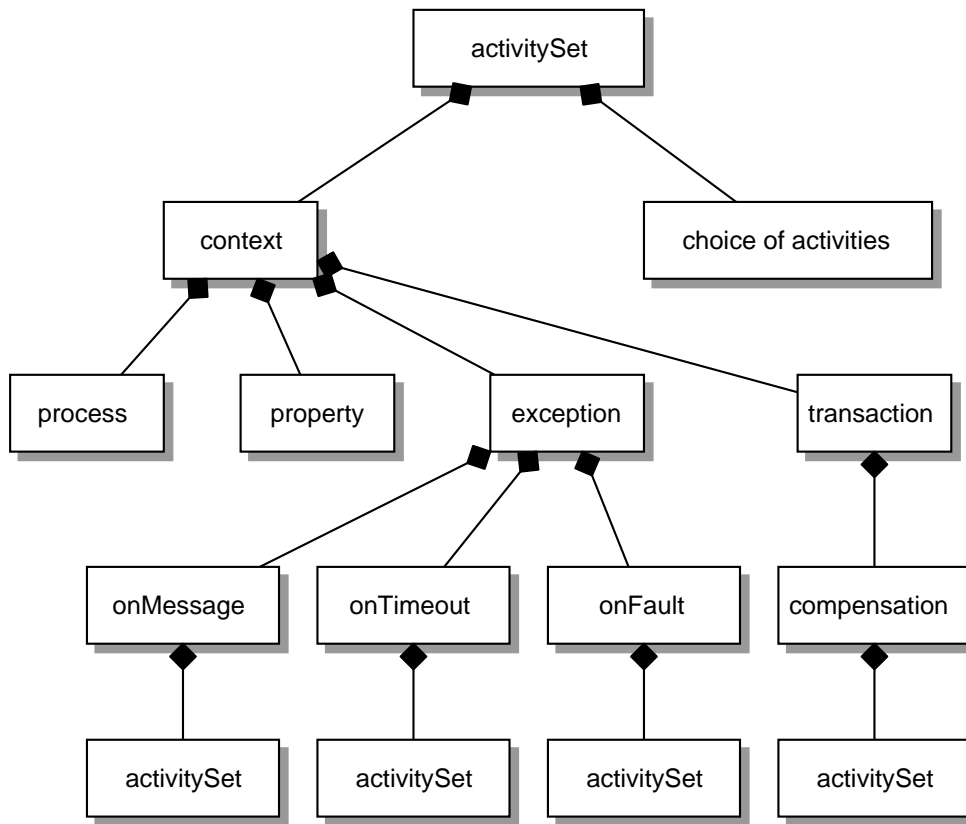


Abbildung 4.5: ActivitySet

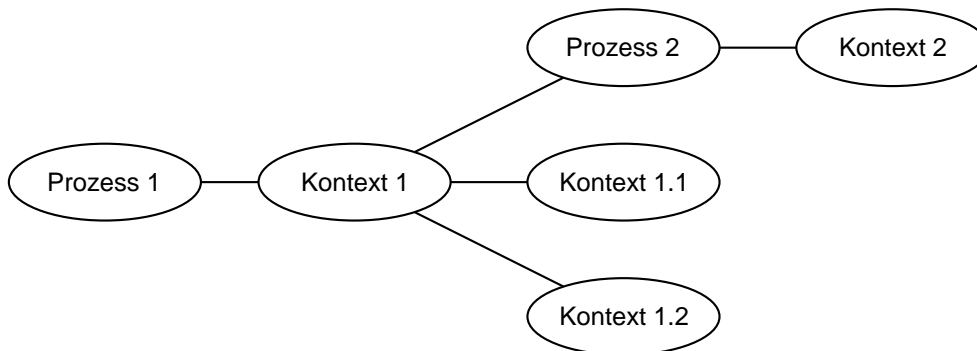


Abbildung 4.6: Kombination von Prozess und Kontext

4.3 Herangehensweise bei der Entwicklung der Semantik

Zuerst überlegen wir uns, welches Element aus *WSCI* das größte darzustellende Element ist. Diese Ansatzweise ermöglicht es uns, alle enthaltenen

Elemente in die Struktur des jeweils nächstgrößeren Elementes einzubetten. Wir beginnen bei dem alle relevanten Informationen umfassenden Element, dem *model*. Das *model* besteht aus Referenzen zu den beteiligten *Web Services (interfaces)* und den Verknüpfungen ihrer Operationen. Es macht wenig Sinn, hierfür einen separaten Baustein zu entwickeln, da keine zusätzlichen Informationen existieren, die wir abbilden könnten. Ein *interface* beinhaltet mehrere Prozesse (*processes*). Es enthält ebenfalls keine weiteren Informationen und muss demnach auch nicht auf einen Baustein abgebildet werden. Diese beiden Elemente sind durch die Verknüpfung der in ihnen enthaltenen Elemente bereits ausreichend beschrieben.

Der Prozess ist nach dem *interface* das nächstgrößere Element. Er enthält mindestens eine Aktivität und einen Kontext. Er steuert den Ablauf der in ihm enthaltenen Aktivitäten. Wir haben bereits hervorgehoben, dass eine Stärke von *WSCI* in der Betrachtung mehrerer Prozessinstanzen besteht. Somit müssen wir unseren Prozess und alle von ihm benutzten Elemente so designen, dass sie mehrere Instanzen gleichzeitig verwalten können. Dazu gehört notwendigerweise auch das Anlegen und Abräumen von Prozessinstanzen, was sich vom Prozess bis zu allen enthaltenen Aktivitäten erstreckt. Der Prozess-Baustein reicht diese Aufgabe also an die in ihm definierten Kontexte weiter, welche wiederum ihre Subkontexte steuern. Diese sind mit den Aktivitäten verbunden und über diesen Weg wird jede einzelne Aktivität gesteuert. Mit dieser so steuerbaren Instanz können wir später sicherstellen, dass an einem bestimmten Punkt im Kontrollfluss keine Aktivität eines Kontextes mehr aktiv sein kann. Das ist z.B. bei der Fehlerbehandlung erforderlich. In diesem Fall werden alle im entsprechenden Kontext liegenden Aktivitäten vor Beginn der Fehlerbehandlung gestoppt. Wir definieren für jeden einzelnen Kontext zwei Steuerbausteine für die Lebenszykluskontrolle: einen für die Aktivitäten und einen für die Ausnahmebehandlung und die an sie angeschlossenen Aktivitäten.

Abbildung 4.7 zeigt die Modellierung eines *WSCI*-Kontextes mit Hilfe von Petrinetz-Bausteinen. Wir sehen drei verschiedene Hierarchien (dargestellt durch die waagrecht verlaufenden Linien). Die obere Hierarchie zeigt die Lebenszykluskontrolle jeder Prozessinstanz, die mittlere Hierarchie stellt den Kontrollfluss durch Verknüpfung von Aktivitäten dar und die untere zeigt die Verknüpfung der Elemente zur Ausnahmebehandlung. *Lebenszykluskontext 1* bedient sowohl den *Ereigniskontext* als auch *Lebenszykluskontext 2*. Ersterer reagiert auf Ereignisse und stoppt alle unter *Lebenszykluskontext 2* liegenden Aktivitäten, falls das auftretende Ereignis ein Fehler ist. Letzterer instanziiert, stoppt und beendet die auszuführenden Aktivitäten. Sowohl die *Lebenszykluskontexte* als auch die auszuführenden Aktivitäten und der *Ereigniskontext* haben Verbindungen zu gleichartigen Elementen in ande-

ren Kontexten. Wenn wir also den *Lebenszykluskontext 2* eines Kontextes beschreiben, meinen wir den entsprechend der Abbildung 4.7 im Kontext angeordneten Baustein *Lebenszykluskontext*. Der Baustein, der einen Prozess darstellt, steuert alle drei Hierarchieebenen.

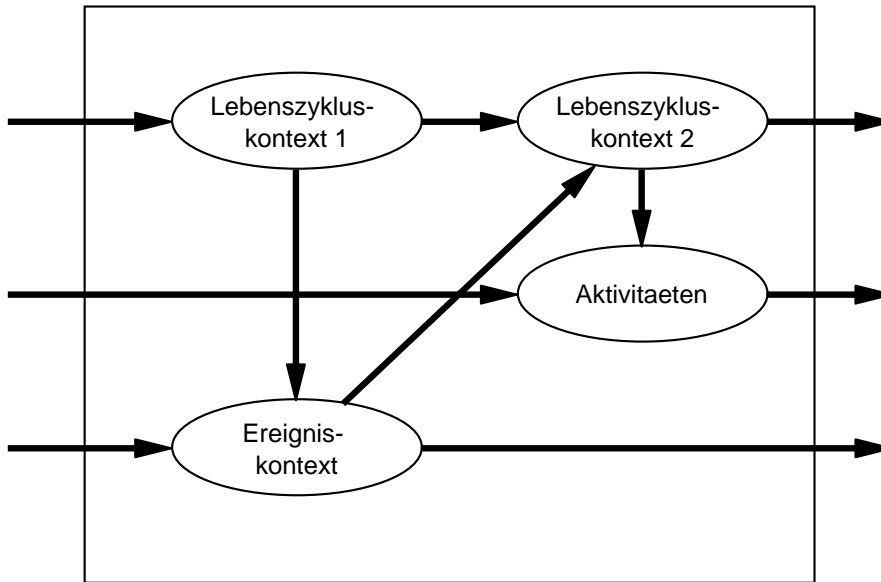


Abbildung 4.7: Modellierung eines Kontextes

4.4 Prozessinstanzsteuerung

Die meisten der in diesem Abschnitt vorgestellten Bausteine stellen jeweils kein Element aus *WSCl* dar, sondern realisieren implizite Forderungen dieser Sprache. Die einzige Ausnahme stellt der Baustein *Lebenszyklusprozess* dar, der das Element *process* aus *WSCl* widerspiegelt. Der Baustein *Lebenszykluskontext* entspricht zusammen mit dem *Ereigniskontext* dem *WSCl*-Element *context*. Die Bausteine *Lebenszykluskontrolle*, *Nachrichtempfang* und *Rufempfang* entsprechen jeweils keinem Element aus *WSCl*.

4.4.1 Lebenszyklusprozess

WSCl definiert ein Element *Process*. Dieses stellt gekapseltes, referenzierbares Verhalten dar. Wir unterscheiden zwischen *Top-level Processes*, die direkt in dem Element *interface* definiert sind, und *Nested Processes*, die in einer komplexen Aktivität definiert sind. Der Start kann durch den Ruf einer Aktivität oder durch den Empfang einer Nachricht ausgelöst werden. Ersteres

kann durch die Aktivitäten *Spawn* oder *Call* geschehen. Die Definition des Prozesses muss sich dazu im selben Kontext oder in einem Vaterkontext der rufenden Aktivität befinden. Nur dann ist der Prozess für die Aktivität sichtbar und kann von ihr gerufen werden. Anschließend wird eine Instanz des Prozesses erzeugt und ausgeführt.

Wir bilden das *WSCI*-Element *Process* auf den Baustein *Lebenszyklusprozess* ab. Dieser Baustein steuert jede Prozessinstanz. Das beinhaltet nach dem Starten durch einen der Bausteine *Nachrichtempfang* (siehe Seite 37) oder *Rufempfang* (siehe Seite 39) das Erzeugen einer Instanz, die Ausführung sowie das Abräumen derselben. Das Erzeugen und das Abräumen von Instanzen wird an einen Baustein vom Typ *Lebenszykluskontext* (siehe Seite 33) weitergegeben. Dieser leitet den entsprechenden Befehl an andere *Lebenszykluskontexte* und an Bausteine vom Typ *Lebenszykluskontrolle* (siehe Seite 36) weiter. Diese sind wiederum mit den ausführenden Bausteinen verbunden und ermöglichen oder verhindern deren Ablauf.

Die Abbildung 4.8 visualisiert den Baustein *Lebenszyklusprozess*. Die folgenden Erklärungen zu einigen Elementen dieses Bausteines dienen dem besseren Verständnis. Der Platz *cs-depot* enthält die Marken, die verschiedene, parallel laufende Prozessinstanzen unterscheidbar machen. Im einfachsten Fall ist das eine Menge von unterschiedlich gefärbten Marken. Ist der Platz *cs-depot* leer, kann keine Instanz ausgeführt werden. Nach der Abarbeitung einer Instanz wird die entsprechende Marke auf den Platz *cs-depot* zurückgelegt. Das ist eine Designentscheidung. Wir könnten den Platz *cs-depot* weglassen. Dann müssten wir aber dafür sorgen, dass der den Prozess rufende Baustein die Unterscheidbarkeit der Instanzen gewährleistet.

Für das Verständnis der Verknüpfungen mit anderen Bausteinen bilden wir Subchnittstellen des Bausteines *Lebenszyklusprozess*. Da vier verschiedene Bausteine mit *Lebenszyklusprozess* verknüpft werden, betrachten wir vier Subchnittstellen. Mit diesem Baustein wird ein Baustein vom Typ *Rufempfang* oder vom Typ *Nachrichtempfang* verknüpft. Weiterhin werden jeweils ein Baustein der Typen *Ereigniskontext* und *Lebenszykluskontext* sowie eine beliebige Aktivität angeschlossen.

Zur Subchnittstelle 1 gehören *cs-depot*, *cs-init*, *cs-finished*, *process kill-out* und *process end-out*. An sie wird die Schnittstelle 2 eines Bausteines vom Typ *Rufempfang* oder *Nachrichtempfang* angeschlossen. Durch eine Transition in dem angeschlossenen Baustein wird gleichzeitig vom Platz *cs-depot* eine Marke konsumiert und auf dem Platz *cs-init* eine Marke erzeugt. So wird *Lebenszyklusprozess* gestartet. Über *cs-finished*, *process end-out* und *process kill-out* wird dem Baustein das Ergebnis des Prozesses mitgeteilt.

Die Subchnittstelle 2 besteht aus *active*, *input-out*, *input-in* und *output*. Sie wird mit der Subchnittstelle 1 des Bausteines der anzuschließenden Ak-

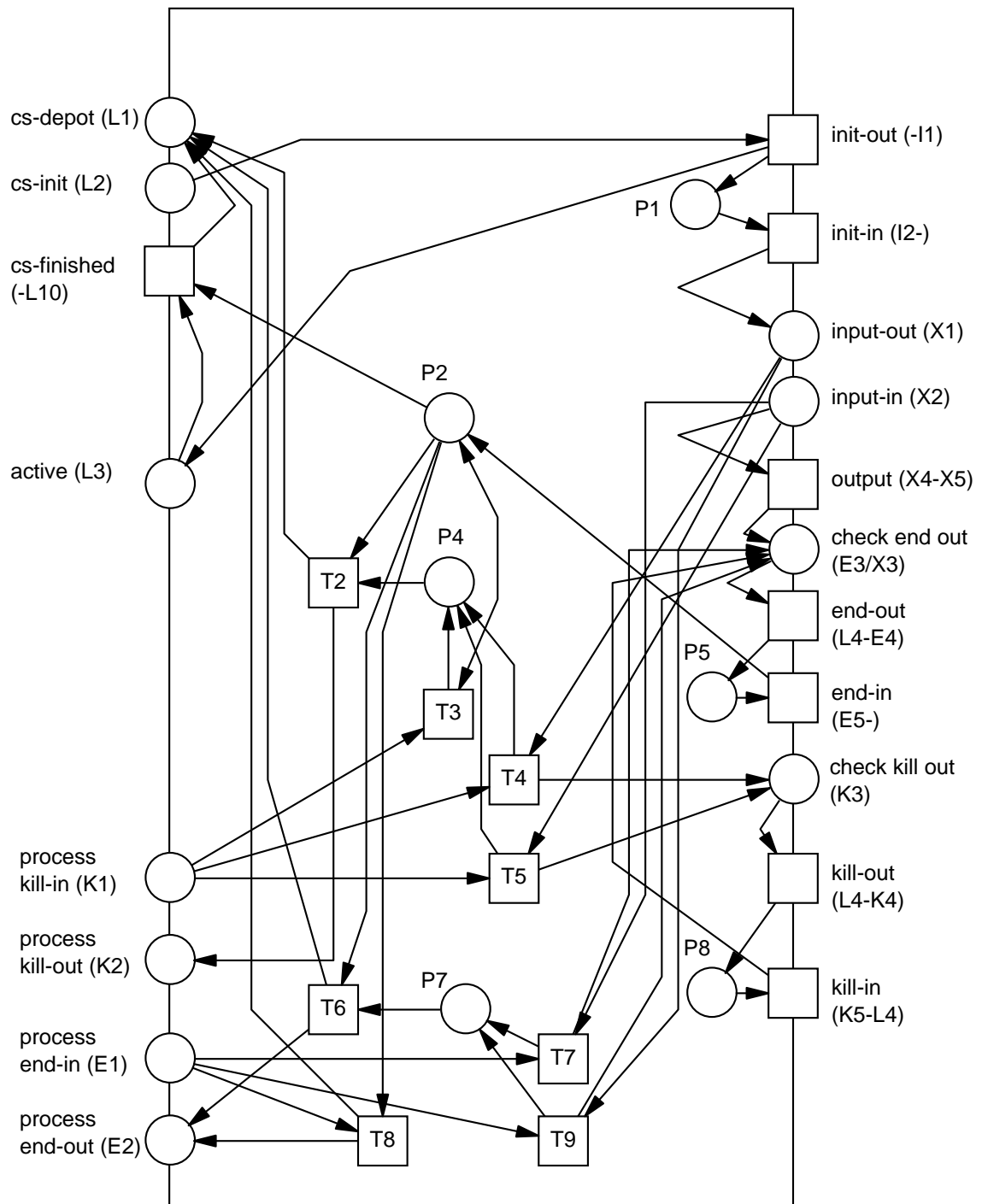


Abbildung 4.8: Petrinetz-Baustein für *Lebenszyklusprozess*

tivität verknüpft. Über die Subchnittstelle 2 wird der Kontrollfluss an die Aktivität geleitet. Eine Marke auf dem Platz *input-out* gibt der Aktivität die Möglichkeit, mit der Abarbeitung anzufangen. Dazu muss auf *active* eine Marke liegen. Auf diese wird lesend zugegriffen. Es wird also eine Marke konsumiert und erzeugt. Wenn die Aktivität mit der Abarbeitung beginnt, produziert sie auf dem Platz *input-in* eine Marke. Mit dem Schalten der Transition *output* ist die angeschlossene Aktivität beendet.

Subchnittstelle 3 setzt sich unter anderem aus *check end out* und *check kill out* zusammen. Über diese Plätze kann ein angeschlossener Baustein vom Typ *Ereigniskontext* abfragen, ob er durch *Lebenszyklusprozess* beendet werden soll. So werden Verklebungen vermieden, wenn einerseits der *Lebenszyklusprozess* den angeschlossenen *Lebenszykluskontext* und andererseits der *Ereigniskontext* den *Lebenszyklusprozess* beenden will. Zur Subchnittstelle 3 gehören weiterhin die Plätze *active*, *process kill-in* und *process end-in*. Leitet ein angeschlossener *Ereigniskontext* eine Fehlermeldung an *Lebenszyklusprozess* weiter, so wird eine Marke von *active* konsumiert und auf *process kill-in* oder *process end-in* erzeugt. Welcher Platz gewählt wird, hängt davon ab, ob auf den eingetretenen Fehler bereits erfolgreich reagiert wurde.

Die Subchnittstelle 4 setzt sich aus *init-out*, *init-in*, *end-out*, *end-in*, *kill-out* und *kill-in* zusammen. Sie wird mit der Subchnittstelle 1 des angeschlossenen *Lebenszykluskontext* verknüpft. Über diese Verknüpfung wird die Initialisierung, das Abräumen und das Stoppen aller enthaltenen Aktivitäten gesteuert.

4.4.2 Lebenszykluskontext

Das Element *Context* aus *WSCI* beschreibt einen Kontext, in dem eine Menge von Aktivitäten ausgeführt werden. Dazu gehören die Menge an Ausnahmefällen und die Aktivitäten, die bei Eintreten der Ausnahmefälle ausgelöst werden. Weiterhin werden in *Context* die Eigenschaften von Transaktionen und zusätzliche *Process*-Elemente definiert. Die Beziehungen von Kontexten innerhalb der Hierarchie des Prozesses benennen wir wie folgt: Ein Kontext ist der Vaterkontext eines anderen Kontextes, wenn der letztere in einer komplexen Aktivität des ersteren definiert wurde. Entsprechend ist ein Kontext ein Kindkontext seines Vaterkontextes. In Abbildung 4.6 auf Seite 28 ist Kontext1 der Vaterkontext von Kontext1.1.

Wir teilen die Aufgaben von *Context* auf verschiedene Bausteine auf. Der Baustein *Ereigniskontext* (siehe Seite 40) definiert das Verhalten im Ausnahmefall. Mit ihm definieren wir auch die Reaktion auf einen Fehler innerhalb einer Transaktion. Die Bausteine zu den in *Context* enthaltenen Aktivitäten und der *Ereigniskontext* werden von den Bausteinen *Lebenszykluskontext* und

Lebenszykluskontrolle gesteuert. Durch diese wird eine Prozessinstanz gestartet, gestoppt und beendet. In Abbildung 4.7 auf Seite 30 ist der Zusammenhang der Bausteine schematisch dargestellt. Der erste *Lebenszykluskontext* steuert alle EreignisHandler und den *Ereigniskontext*. Der zweite, hierarchisch unter dem ersten liegende Baustein steuert die Aktivitäten. Der *Ereigniskontext* reagiert auf Ereignisse (siehe Seite 42). Auf diese Weise ist es im Fehlerfall möglich, erst die Ausführung aller Aktivitäten zu stoppen und anschließend die Fehlerbehandlung zu starten. Die genaue Funktionsweise erklären wir am Baustein.

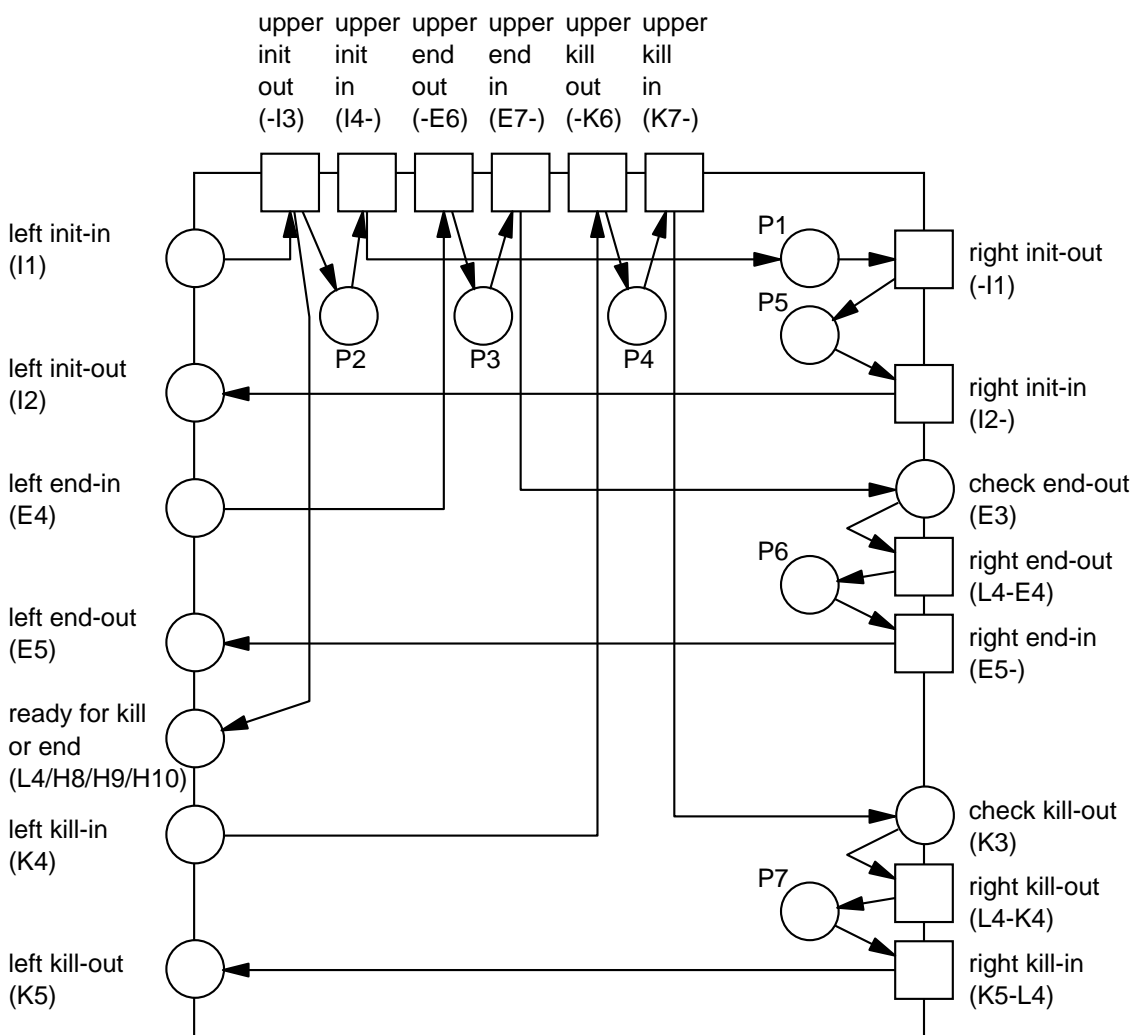


Abbildung 4.9: Petrinetz-Baustein für *Lebenszykluskontext*

Die Namen der Elemente der Schnittstelle beschreiben, ob die Elemente die jeweilige Instanz erzeugen (init), beenden (end) oder abbrechen (kill).

An diesen Baustein können wir bis zu vier Typen anderer Bausteine anschließen. Wir betrachten aber nur drei Schnittstellen, da eine von ihnen von mehreren Bausteinen verwendet wird.

In Schnittstelle 1 sind der Platz *ready for kill or end* und alle Plätze enthalten, die mit „left“ beginnen. An diese Schnittstelle wird entweder die Schnittstelle 3 eines anderen *Lebenszykluskontext* oder die Schnittstelle 4 eines *Lebenszyklusprozess* angeschlossen. Das hängt von der Position dieses *Lebenszykluskontext* in der Hierarchie ab. Weiterhin kann ein *Ereigniskontext* angeschlossen werden. Dieser stoppt die Ausführung aller enthaltenen Aktivitäten im Fehlerfall (kill) und re-initialisiert sie, falls der Fehler erfolgreich behandelt werden konnte. Die vier Alternativen für den Schnittstellenplatz *ready for kill or end* haben folgende Bedeutungen: *L4* wird für die Verknüpfung zwischen zwei aufeinanderfolgenden Bausteinen vom Typ *Lebenszykluskontext* verwendet. *H10* bezeichnet von der anzuschließenden Transition aus gesehen den Platz *L4* im *Lebenszykluskontext 1* des Vaterkontextes (siehe Abbildung 4.7 auf Seite 30). Falls es keinen Vaterkontext gibt, wird *H10* nicht berücksichtigt. Die entsprechende Alternative ist an der Transition vermerkt (siehe *Ereigniskontext*). Entsprechend bezeichnet *H8* den *Lebenszykluskontext 1* des eigenen Kontextes und *H9* den *Lebenszykluskontext 2* des eigenen Kontextes.

Falls der aufgetretene Fehler erfolgreich behandelt wurde, werden alle Bausteine unter *Lebenszykluskontext 2* im aktuellen Kontext erst beendet und anschließend neu initialisiert. Das hat zur Folge, dass alle betroffenen Aktivitäten trotz des aufgetretenen Fehlers wieder ausführbar sind.

Wenn der Baustein *Lebenszykluskontext* gestoppt (kill) oder beendet (end) werden soll, wird vom Platz *ready for end or kill* eine Marke konsumiert. Nachdem der kill-Befehl ausgeführt wurde, wird auf diesen Platz wieder eine Marke gelegt. Nach Beendigung des Kontext (end) bleibt dieser Platz ohne Marke. Dass der Platz *ready for end or kill* eine Marke hat, ist eine Bedingung dafür, dass der Baustein *Lebenszykluskontext* gestoppt oder beendet werden kann. So garantieren wir, dass nicht von mehreren Quellen gleichzeitig ein Befehl an den *Lebenszyklusprozess* geht. Das ist z.B. dann der Fall, wenn der Vaterkontext versucht, seinen Kindkontext zu beenden, dieser sich aber nach dem Auftreten eines eigenen Fehler selbst beenden will. Es wird derjenigen Fehlerbearbeitung Vorrang gegeben, welche die Marke von *ready for end or kill* zuerst konsumiert hat.

In Schnittstelle 2 sind alle Transitionen enthalten, deren Name mit „upper“ beginnt. Sie wird an beliebig viele Bausteine vom Typ *Lebenszykluskontrolle* angeschlossen. An jedem dieser angeschlossenen Bausteine ist eine zu steuernde Aktivität angeschlossen.

In Schnittstelle 3 sind alle Transitionen und Plätze enthalten, deren

Namen mit „right“ oder „check“ beginnen. Durch die Marken auf den Plätzen *check kill-out* und *check end-out* kann ein *Ereigniskontext* überprüfen, ob er von seinem Vaterkontext aus beendet werden soll. Dadurch werden Verklemmungen bei der Fehlerbehandlung vermieden. An Schnittstelle 3 werden beliebig viele *Lebenszykluskontexte* angeschlossen, die Anzahl hängt davon ab, wie viele Aktivitäten hier unter einer komplexen Aktivität zusammengefasst werden. Wird kein Baustein angeschlossen, stellt der aktuelle Baustein ein Blatt im Hierarchiebaum dar.

4.4.3 Lebenszykluskontrolle

Der Baustein *Lebenszykluskontrolle* entspricht keinem Element aus *WSCl* direkt. Er ist für die Instanziierung der EreignisHandler, des *Ereigniskontextes* und jeder Aktivität zuständig. Wie auf Seite 33 beschrieben wird er zur Umsetzung des *Context* aus *WSCl* verwendet.

Lebenszykluskontrolle wird mit jedem an ihn angeschlossenen Baustein durch Verknüpfung mit den Plätzen *start*, *active* und *stop* verbunden. Diese Verknüpfung wird in Abbildung 4.10 durch die außerhalb des Rahmens liegenden Plätze angedeutet. Der angeschlossene Baustein wird über die entsprechende Schnittstelle instanziiert, gestoppt und beendet. Die entsprechenden Anweisungen bekommt *Lebenszykluskontrolle* von dem Baustein *Lebenszykluskontext*, der für die Kontrolle des Lebenszyklus eines *WSCl*-Kontextes zuständig ist (siehe Seite 34).

In Abbildung 4.10 sehen wir den Baustein *Lebenszykluskontrolle*. Alle Plätze der Schnittstelle bilden die Subchnittstelle 1. Sie wird mit der Subchnittstelle 2 des *Lebenszykluskontext* verbunden. Über sie werden alle Befehle des Kontextes an den angeschlossenen Baustein weitergeleitet.

Die Transitionen *T1* bis *T5* stellen die Subchnittstelle 2 dieses Bausteines dar. Sie wird mit den Plätzen *start*, *active* und *stop* des angeschlossenen Bausteines verbunden.

Die Intention bei der Gestaltung dieses Bausteines ist die Steuerung des Lebenszyklus des angeschlossenen Bausteines. Nach der Instanziierung sind die Plätze *start* und *active* belegt. Das ist die Anfangsbedingung für die Abarbeitung des angeschlossenen Bausteines. Nach dem Eintreffen des *kill*-Befehls (Schalten von *T4* oder *T5*) liegt auf *active* keine Marke und auf *stop* eine Marke. Da alle für den Kontrollfluss wesentlichen Transitionen der angeschlossenen Bausteine von der Belegung des Platzes *active* abhängen, entspricht das einer Blockade des Bausteines. Mit der Ausführung des *end*-Befehls werden alle benutzten Marken abgeräumt. Nach dem Beenden der Prozessinstanz sind keine Marken dieser Instanz mehr vorhanden. Das mit diesen Bausteinen erzeugte Modell wird also immer vollständig aufgeräumt

und kann beliebig oft hintereinander ausgeführt werden, ohne dabei auf einem Platz Marken zu sammeln.

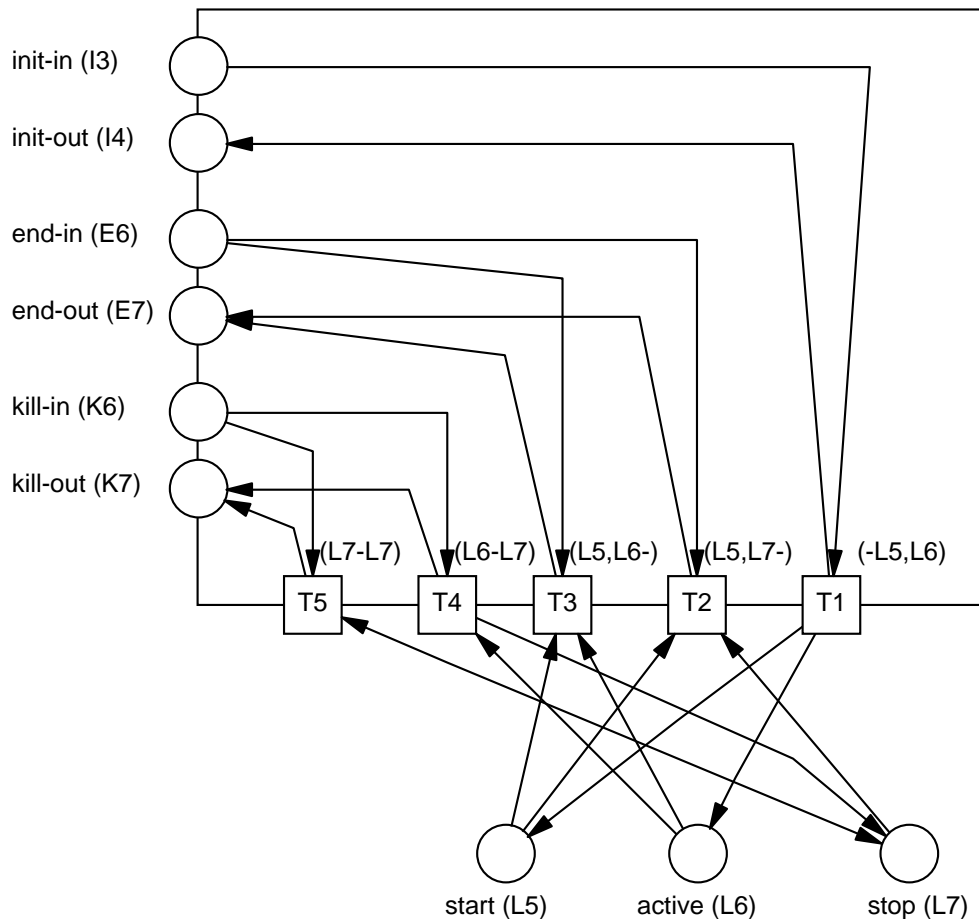


Abbildung 4.10: Petrinetz-Baustein für *Lebenszykluskontrolle*

4.4.4 Nachrichtempfang

Der Baustein *Nachrichtempfang* entspricht keinem Element aus *WSCI*. Er wird für den Fall benutzt, dass das Attribut „instantiation“ den Wert „message“ besitzt. Der entsprechende *Context* in *WSCI* kann damit nur durch den Empfang einer Nachricht gestartet werden. Dann muss die erste auszuführende Aktivität eine nachrichtempfangende, atomare oder eine komplexe Aktivität sein. Für den Fall, dass diese erste Aktivität *All* ist, müssen alle unter ihr liegenden Aktivitäten eine Nachricht empfangen. Die Prozessinstanz startet erst, wenn alle entsprechenden Nachrichten empfangen wurden.

Entsprechend bestimmt die Verwendung des Bausteines *Nachrichtempfang*, dass der angeschlossene Baustein vom Typ *Lebenszyklusprozess* nur durch den Empfang von Nachrichten gestartet werden kann. *Nachrichtempfang* startet den Prozess und stellt den entsprechenden nachrichtempfangenden Bausteinen alle Nachrichten zur Verfügung. Der Baustein *Nachrichtempfang* ist in Abbildung 4.11 dargestellt. Wir betrachten drei Subschnittstellen dieses Bausteines.

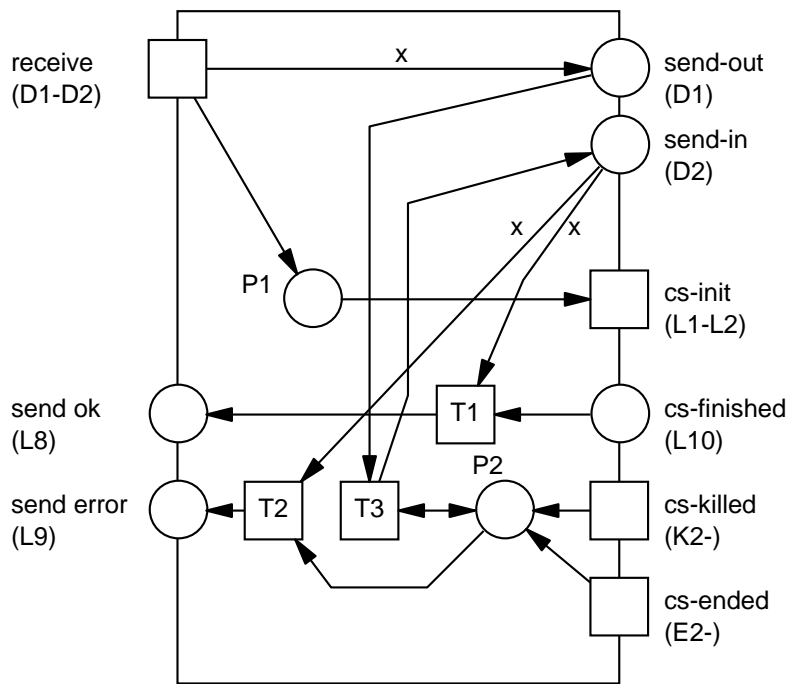


Abbildung 4.11: Petrinetz-Baustein für *Nachrichtempfang*

Subschnittstelle 1 enthält *receive*, *send ok* und *send error*. Diese Subschnittstelle wird mit jeder sendenden Aktivität verknüpft. *receive* startet die Abarbeitung des Prozesses, die beiden Plätze *send ok* und *send error* enthalten die Information über das positive oder negative Ergebnis der Prozessinstanz. Die Transition *receive* schaltet erst, wenn alle benötigten Nachrichten gesendet wurden. Diese Nachrichten werden den empfangenden Aktivitäten durch Subschnittstelle 3 zur Verfügung gestellt. Die Variable x an den Flusskanten im Baustein gibt die Anzahl der für den Start der Prozessinstanz erforderlichen Nachrichten an.

In Subschnittstelle 2 sind alle Elemente enthalten, deren Namen mit „cs“ beginnen. Sie wird mit Subschnittstelle 1 des *Lebenszyklusprozess* verbunden, startet die Abarbeitung des Prozesses und empfängt das Ergebnis.

Die beiden Plätze *send-out* und *send-in* werden in Subchnittstelle 3 zusammengefasst. Sie leiten die eingehende Nachricht an die empfangenden Aktivitäten weiter. Falls der Prozess abbricht, bevor alle Nachrichten abgeholt wurden, werden die verbleibenden Marken durch Transition *T3* konsumiert.

4.4.5 Rufempfang

Auch *Rufempfang* entspricht keinem Element aus *WSCI*. Im Gegensatz zu *Nachrichtempfang* wird dieser Baustein verwendet, wenn das Attribut „instantiation“ den Wert „other“ besitzt. Somit kann *Process* nicht durch den Empfang einer Nachricht, sondern beispielsweise durch den Ruf der komplexen Aktivitäten *Call* oder *Spawn* gestartet werden.

Der Aufbau des Bausteines *Rufempfang* hat dementsprechend Ähnlichkeit zu dem Aufbau des Bausteines *Nachrichtempfang*. Wird er verwendet, kann der angeschlossene Baustein vom Typ *Lebenszyklusprozess* nur durch den Ruf durch eine Aktivität und nicht durch Empfang einer Nachricht gestartet werden. *Rufempfang* startet den angeschlossenen Prozess.

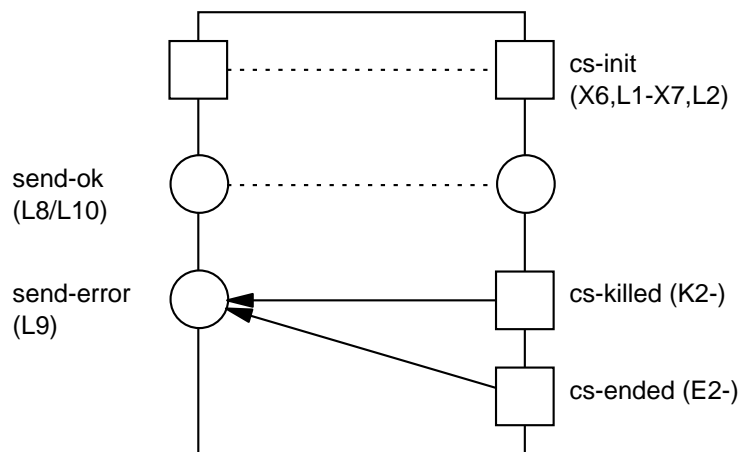


Abbildung 4.12: Petrinetz-Baustein für *Rufempfang*

Der Baustein *Rufempfang* ist in Abbildung 4.12 visualisiert. Die gestrichelten Linien deuten an, dass es sich um ein und dieselbe Transition bzw. ein und denselben Platz handelt. An diesen Baustein werden zwei weitere Bausteine angeschlossen. Wir definieren zwei Subchnittstellen.

Subchnittstelle 1 enthält *cs-init*, *send-ok* und *send-error*. Diese Subchnittstelle wird mit der rufenden Aktivität (z.B. Subchnittstelle 3 von *Request-Response*) und dem *Lebenszyklusprozess* verknüpft. Über sie wird der Prozess gestartet und sein Ergebnis übertragen.

Zu Subchnittstelle 2 gehören *cs-init*, *send-ok*, *cs-killed* und *cs-ended*. Sie wird mit Subchnittstelle 1 des *Lebenszyklusprozess* verbunden.

4.5 Fehlerbehandlung

Mit Aktivitäten beschreiben wir das erwartete Verhalten eines *Web Service*. Unerwartetes Verhalten wird in *WSCI* durch das Element *Exception* dargestellt. Mit den Werten „onMessage“, „onTimeout“ und „onFault“ wird im Attribut *Content* festgelegt, auf welche Art von Ereignis reagiert werden kann. Falls die enthaltenen Aktivitäten Bestandteil einer Transaktion sind, wird in *WSCI* in dem Element *transaction* das entsprechende Kompensationsverhalten definiert. In Abbildung 4.7 auf Seite 30 ist der Aufbau eines Kontextes grob dargestellt. Falls in einer der enthaltenen Aktivitäten ein Fehler auftritt, wird dieser an *Ereigniskontext* weitergeleitet. Dieser stoppt *Lebenszykluskontext 2* und damit alle darunter liegenden Aktivitäten und beginnt anschließend mit der Ausführung des Fehlerverhaltens. Tritt in diesem wiederum ein Fehler auf, so wird der entsprechende Fehler an den *Ereigniskontext* im Vaterkontext weitergeleitet. Läuft die Fehlerbehandlung fehlerfrei ab, wird das Verhalten der wartenden komplexen Aktivität im Vaterkontext fortgesetzt.

Die Fehlerbehandlung in einem Kontext setzt sich aus einem *Ereigniskontext* und beliebig vielen EreignisHandleern zusammen. Ein EreignisHandler ist vom Typ *ExceptionHandler*, *DefaultExceptionHandler*, *MessageHandler* oder *TimeoutHandler*. Drei von fünf hier vorgestellten Bausteinen entsprechen Elementen aus *WSCI*. Ausnahmen sind der *Ereigniskontext* und der *DefaultExceptionHandler*. Ersterer stellt zusammen mit *Lebenszykluskontext* das Element *Context* aus *WSCI* dar. Letzterer existiert für den Fall, dass in *WSCI* kein zu dem Ereignis passender EreignisHandler definiert ist. Er leitet das Ereignis direkt an den Vaterkontext weiter.

4.5.1 Ereigniskontext

In *WSCI* werden im Element *Context* neben den auszuführenden Aktivitäten auch die Reaktionen auf möglicherweise eintretende Ereignisse definiert. In *Context* wird durch je ein Element *Exception* ein Ausnahmefall und die entsprechende Reaktion beschrieben. Die Abarbeitung eines Ereignisses kann nur dann beginnen, wenn gerade kein anderes Ereignis abgearbeitet wird.

Der Baustein *Ereigniskontext* entspricht keinem Element aus *WSCI* direkt. Er stellt zusammen mit *Lebenszykluskontext* das Element *Context* dar (siehe Seite 33). Der *Ereigniskontext* kontrolliert den Ablauf aller Ereignis-

Handler im jeweiligen Kontext. Ein EreignisHandler ist ein Baustein, der auf bestimmte Ereignisse reagiert. Alle Handler werden an den *Ereigniskontext* angeschlossen und über ihn wird weitergeleitet, ob der Ablauf des Handlers erfolgreich war oder ob der Vaterkontext ebenfalls eine Fehlermeldung erhält. Er stellt auch sicher, dass immer höchstens ein Handler aktiv ist. Jeder EreignisHandler kontrolliert vor seinem Start, ob er für das entsprechende Ereignis zuständig ist. Ist kein *ExceptionHandler* für den auftretenden Fehler zuständig, wird er an den Vaterkontext weitergeleitet. In *Ereigniskontext* wird dieses Verhalten durch die Transitionen *throw error-out* und *catch error-out* definiert. Alle Transitionen, die mit *break off* beginnen, ermöglichen den Abbruch des *Ereigniskontext*. Sie haben Konzession, wenn der Vaterkontext seine Kinderkontexte beenden will. Auf diese Weise verhindern wir eine Verklemmungssituation. Die Plätze *start*, *start error handler*, *event handler in progress* und *event handler finished* spiegeln den aktuellen Zustand dieses Bausteines wider. Der Unterschied zwischen *start* und *start error handler* ist, dass vom letzteren aus nur *ExceptionHandler* starten können. Eine Aktivität oder ein fehlererzeugender Subkontext legt dazu die Marke von *start* nach *start error handler*. In Abbildung 4.13 sehen wir den Baustein *Ereigniskontext*. Wir definieren für diesen Baustein fünf Subschnittstellen.

Zur Subschnittstelle 1 gehören *throw error-out*, *break off throw x*, *break off catch x* und *catch error x* (für $x = 1, 2$). Diese Subschnittstelle ist notwendig, um das Eintreten externer Ereignisse (z.B. das Eintreffen einer Nachricht) in den eigenen Kontrollfluss eingliedern zu können. Liefere dieser Vorgang fehlerhaft ab, würden nach der Ausführung des *Web Service* Marken im Modell verbleiben - das Modell wäre nicht aufgeräumt. Aus diesem Grund legen wir viel Wert auf die korrekte Eingliederung externer Ereignisse. Dabei unterscheiden wir drei Fälle. Im ersten Fall wurden die im entsprechenden Kontext befindlichen Aktivitäten noch nicht begonnen, im zweiten Fall sind sie gerade aktiv und im dritten Fall wurden sie bereits beendet. Subschnittstelle 1 wird mit dem *Ereigniskontext* des Vaterkontextes verbunden. Wenn ein solcher nicht existiert, wird er mit *Lebenszyklusprozess* verbunden. Subschnittstelle 1 wird zusammen mit *init finished* mit dem *Lebenszykluskontext 1* des aktuellen Kontextes verknüpft (siehe Benennung *H8*).

Die Subschnittstelle 2 enthält *init finished*, *start init*, *end finished* und *start end*. Sie wird im aktuellen Kontext mit *Lebenszykluskontext 2* verknüpft (siehe Abbildung 4.7 auf Seite 30). Nach dem Schalten der Transition *catch error out 1* oder *catch error out 2* werden alle im entsprechenden *Lebenszykluskontext* enthaltenen Aktivitäten beendet und neu initialisiert. Damit wird die Spezifikation für den Fall des erfolgreichen Behandeln eines Fehlers erfüllt. Danach können übergeordnete Kontexte die Arbeit nach Abarbeitung des Fehlerverhaltens fortsetzen. Dazu gehört im Fall einer Schleife auch das

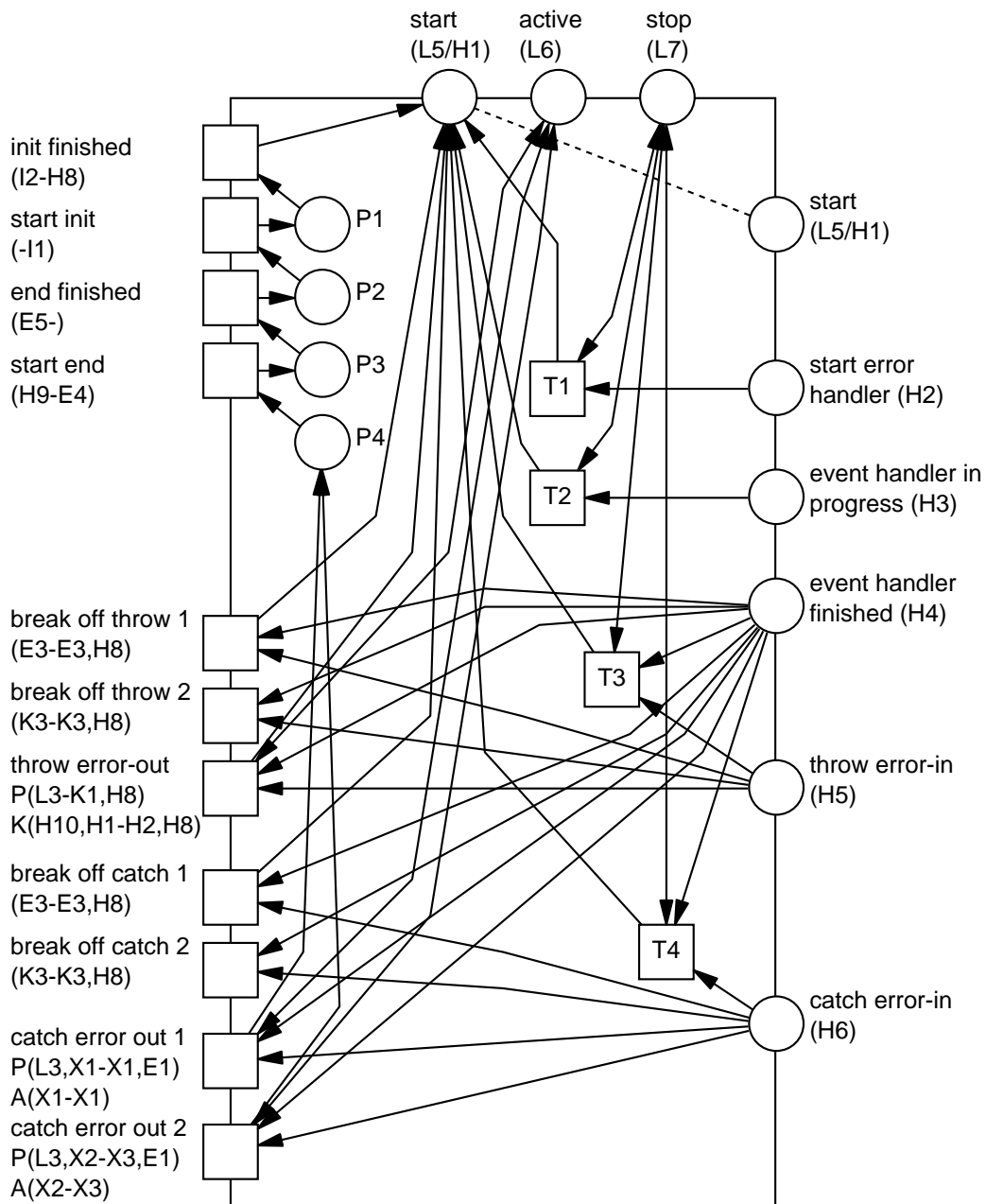


Abbildung 4.13: Petrinetz-Baustein für *Ereigniskontext*

wiederholte Abarbeiten der enthaltenen Aktivitäten.

Die Schnittstelle 3 beinhaltet die Plätze *start*, *active* und *stop*. Sie wird mit der Schnittstelle 2 der angeschlossenen *Lebenszykluskontrolle* verbunden.

In Schnittstelle 4 sind *start*, *start error handler*, *event handler in progress*, *event handler finished*, *throw error-in* und *catch error-in* enthalten. An sie werden alle EreignisHandler angeschlossen. Alle EreignisHandler außer *ExceptionHandler* und *DefaultExceptionHandler* beginnen ihre Abarbeitung vom Platz *start*. Sie sind vom Eintreten externer Ereignisse, wie dem Erreichen eines Zeitpunktes oder dem Empfang einer Nachricht, abhängig. Tritt ein Fehler auf, so konsumiert der Baustein, der den Fehler erzeugt, eine Marke von *start* und produziert eine Marke auf *start error handler*. Gleichzeitig wird vom *Lebenszykluskontext* des *Ereigniskontext* die Marke vom Platz *ready for kill or end* konsumiert. Das führt zu einer „ungestörten“ Fehlerbehandlung. Falls ein Vaterkontext diesen *Ereigniskontext* beenden wollte, müsste er die Abarbeitung von dessen Fehlerverhalten abwarten. Vom Platz *start error handler* wird die Marke dann mit dem Start eines fehlerbehandelnden EreignisHandler konsumiert.

Die gestrichelte Linie zwischen den Plätzen namens *start* zeigt, dass sie ein und denselben Platz darstellen. Der Platz *start* gehört zu zwei Schnittstellen und wird mit *Lebenszykluskontrolle* und EreignisHandlern verbunden.

Schnittstelle 5 beinhaltet *catch error out 1* und *catch error out 2*. Diese Transitionen werden mit der komplexen Aktivität im Vaterkontext verbunden, die die in diesem Kontext befindlichen Aktivitäten steuert. Falls ein hier auftretender Fehler erfolgreich behandelt wurde, wird die Abarbeitung der Aktivität fortgesetzt. Falls eine solche Aktivität nicht existiert, muss an dieser Stelle die Abarbeitung des *Lebenszyklusprozess* fortgesetzt werden (siehe Unterscheidung *A* und *H* auf Seite 24).

Die Spezifikation von *WSCI* macht keine genaueren Angaben, wie mit den an einen EreignisHandler angeschlossenen Aktivitäten umzugehen ist. Ein Fehler innerhalb eines EreignisHandlers führt zum Abbruch der Aktivitäten und zum Weiterreichen des Fehlers an den Vaterkontext. Es ist deshalb eine Designentscheidung, die Schnittstelle der EreignisHandler für Aktivitäten und nicht für Kontexte passend zu gestalten. Letztere würden eine eigene Fehlerbehandlung definieren.

4.5.2 ExceptionHandler

In *WSCI* reagiert ein Element *exception* mit dem Wert „onFault“ auf das Auftreten eines Fehlers. Wenn ein Fehler auftritt, werden alle im aktuellen Kontext liegenden Aktivitäten gestoppt. Danach werden die Aktivitäten zur Fehlerbehandlung ausgeführt. Tritt dabei ein Fehler auf, wird der entsprechende Fehlercode an den *Ereigniskontext* im Vaterkontext geleitet. Dieses Verhalten modellieren wir mit dem Baustein *ExceptionHandler*. Die ausführliche Beschreibung des Verhaltens im Fehlerfall befindet sich auf Seite 40.

Mit *input* wird der *ExceptionHandler* gestartet. Er wird an den *Ereigniskontext* angeschlossen und teilt diesem das Ergebnis seiner Abarbeitung mit. Das kann die erfolgreiche Behandlung oder das erneute Auftreten eines Fehlers sein. In Abbildung 4.14 sehen wir den Baustein für den *ExceptionHandler*. Wir bilden für diesen Baustein vier Subschnittstellen.

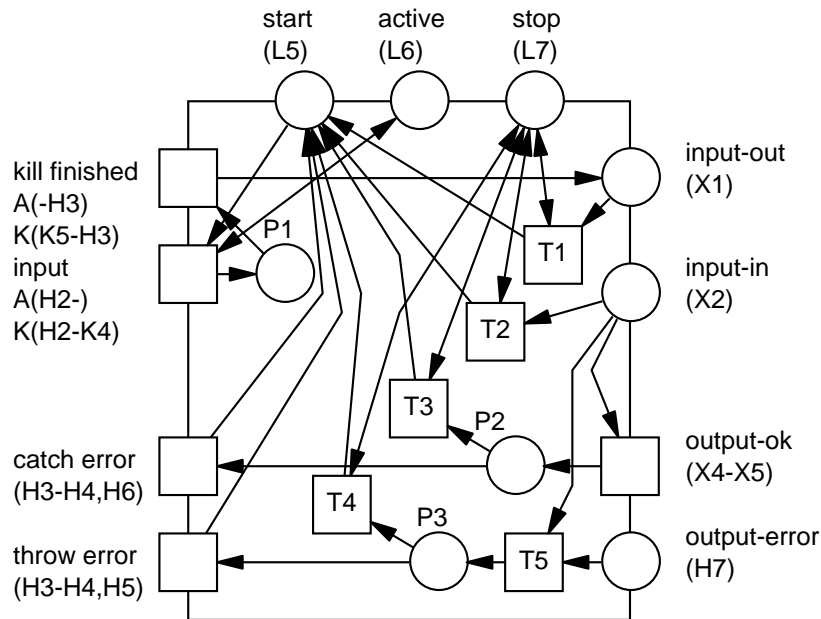


Abbildung 4.14: Petrinetz-Baustein für *ExceptionHandler*

In Subschnittstelle 1 sind *kill finished*, *input*, *catch error* und *throw error* enthalten. An diese Subschnittstelle wird die Subschnittstelle 4 des *Ereigniskontext* oder Subschnittstelle 4 des *Choice* angeschlossen (siehe Bezeichnungen *K* und *A*). Über sie wird der *ExceptionHandler* gestartet und das Ergebnis seiner Abarbeitung übertragen.

Subschnittstelle 2 ist eine Teilmenge von Subschnittstelle 1 und enthält *input* und *kill finished*. Über *input* wird der kill-Befehl im aktuellen Kontext an den *Lebenszykluskontext 2* geschickt (siehe Abbildung 4.7 auf Seite 30). Mit *kill finished* wird auf die Bestätigung der Ausführung des kill-Befehls gewartet. Anschließend wird die Abarbeitung der Fehlerbehandlung gestartet.

Die Plätze *start*, *active* und *stop* bilden die Subschnittstelle 3. Über sie wird die Verknüpfung zu einem Baustein vom Typ *Lebenszykluskontrolle* realisiert.

Subschnittstelle 4 umfasst *input-out*, *input-in*, *output-ok* und *output-error*. Über sie werden die Aktivitäten angesprochen, die zur Abarbeitung des Fehlers zuständig sind. Die Plätze *input-out* und *input-in* starten die mit dem

ExceptionHandler verbundene Aktivität. Über *output-ok* und *output-error* wird das Ergebnis des Ablaufes empfangen.

4.5.3 DefaultExceptionHandler

Der Baustein *DefaultExceptionHandler* entspricht keinem Element aus *WSCI*. Er wird verwendet, wenn auf einen auftretenden Fehler durch keinen anderen *ExceptionHandler* reagiert wird. Er leitet den entsprechenden Fehler einfach an den *Ereigniskontext* im Vaterkontext weiter. In Abbildung 4.15 sehen wir die Umsetzung von *DefaultExceptionHandler*. Wir definieren zwei Subschnittstellen.

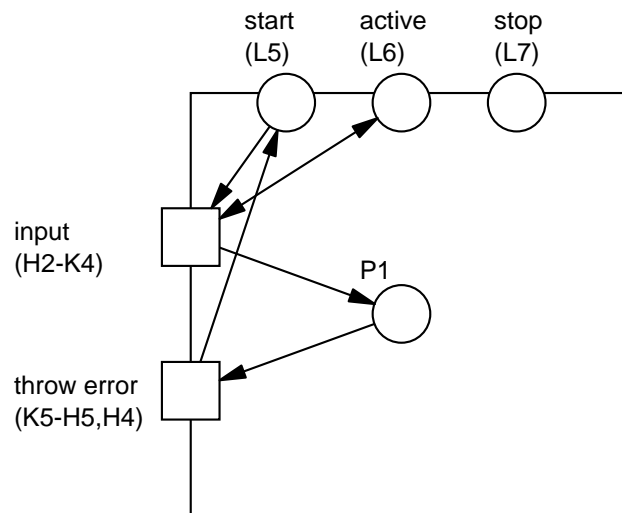


Abbildung 4.15: Petrinetz-Baustein für *DefaultExceptionHandler*

Subschnittstelle 1 enthält die Transitionen *input* und *throw error*. Mit *input* wird im aktuellen Kontext dem *Lebenszykluskontext 2* der kill-Befehl gegeben (siehe Abbildung 4.7 auf Seite 30). Die Transition *throw error* erhält die Bestätigung für die Ausführung des kill-Befehles und liefert einen Fehler an *Ereigniskontext*, der den Fehler an seinen Vaterkontext weiterleitet.

Zu Subschnittstelle 2 gehören *start*, *active* und *stop*. Über sie wird dieser Baustein mit einem Baustein vom Typ *Lebenszykluskontrolle* verbunden.

4.5.4 MessageHandler

Das Element *exception* mit dem Wert „onMessage“ reagiert in *WSCI* auf den Empfang einer Nachricht. An den Empfang der Nachricht ist die Ausführung einer Aktivität gekoppelt. Tritt bei dieser Ausführung ein Fehler auf, wird

der entsprechende Fehler an den angeschlossenen *Ereigniskontext* weitergeleitet. Danach wird der zuständige *ExceptionHandler* ausgeführt. Den Empfang der Nachricht modellieren wir mit dem Baustein *MessageHandler*. Von besonderer Schwierigkeit ist die Eingliederung dieser externen Ereignisse in den internen Ablauf. Im fehlerfreien Fall beeinflusst die Ausführung des *MessageHandlers* den internen Ablauf nicht. Für den Fall eines auftretenden Fehlers wird der interne Ablauf jedoch gestoppt.

MessageHandler löst bei Eintreffen einer Nachricht die Abarbeitung der angeschlossenen Aktivität aus, die das Reaktionsverhalten ausführt. Die Abbildung 4.16 zeigt den Baustein. Wir konzentrieren uns auf drei Subchnittstellen.

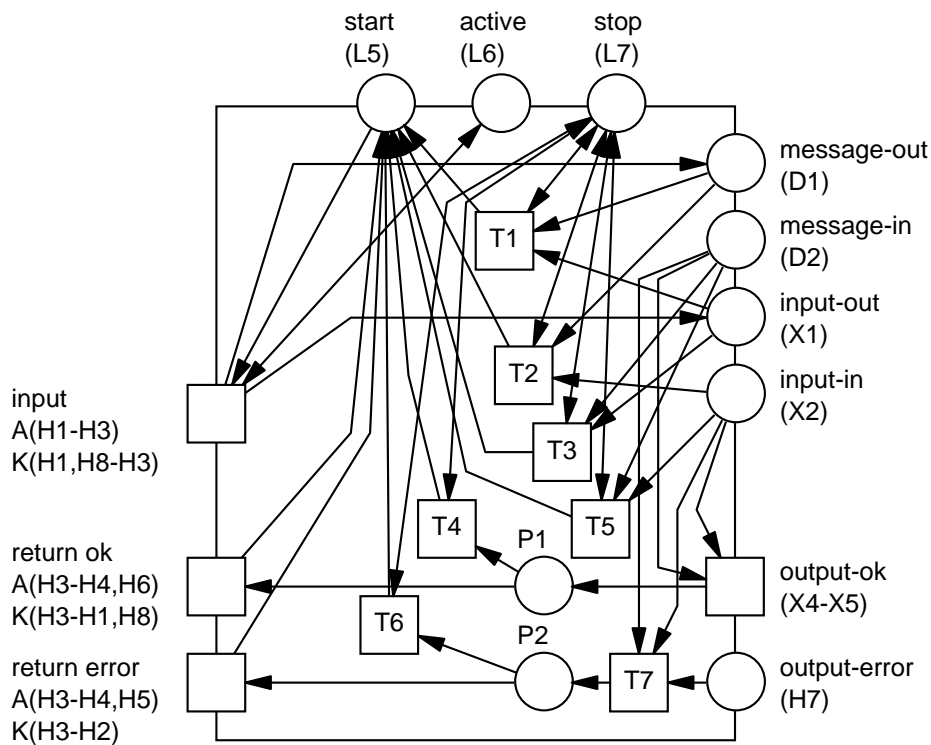


Abbildung 4.16: Petrinetz-Baustein für *MessageHandler*

Subchnittstelle 1 enthält die Transitionen *input*, *return ok* und *catch error*. Über sie wird (wie bei *ExceptionHandler*) die Verbindung zu *Ereigniskontext* oder *Choice* hergestellt.

Zur Subchnittstelle 2 gehören wie bei allen EreignisHandlern *start*, *active* und *stop*. Durch sie wird *MessageHandler* mit einem Baustein vom Typ *Lebenszykluskontrolle* verbunden.

Die Schnittstelle 3 umfasst *message-out*, *message-in*, *input-out*, *input-in*, *output-ok* und *output-error*. Über sie wird die nachrichtempfangende Aktivität mit *MessageHandler* verknüpft.

4.5.5 TimeoutHandler

Wenn eine vorgegebene Zeitschranke überschritten wird, reagiert darauf ein *WSCI-Element Exception* mit dem Wert „onTimeout“. Die Zeitschranke kann durch ein konkretes Datum oder einen Startzeitpunkt zusammen mit einer vorgegebenen Dauer festgelegt werden. Das Element *Exception* löst das mit dem Auftreten dieser Zeitüberschreitung verbundene Verhalten aus.

Wir modellieren dieses Verhalten mit dem Baustein *TimeoutHandler*. Da wir die Zeit in unseren Petrinetzen nicht modellieren, stellen wir diese Bedingung durch Nichtdeterminismus dar. Alternativ dazu können wir zu *input* einen weiteren Vorplatz hinzufügen, der beschreibt, ob die Zeitschranke überschritten wurde. Der Baustein in Abbildung 4.17 setzt das *WSCI-Konstrukt* um. Wir definieren vier Schnittstellen für die Schnittstelle dieses Bausteines.

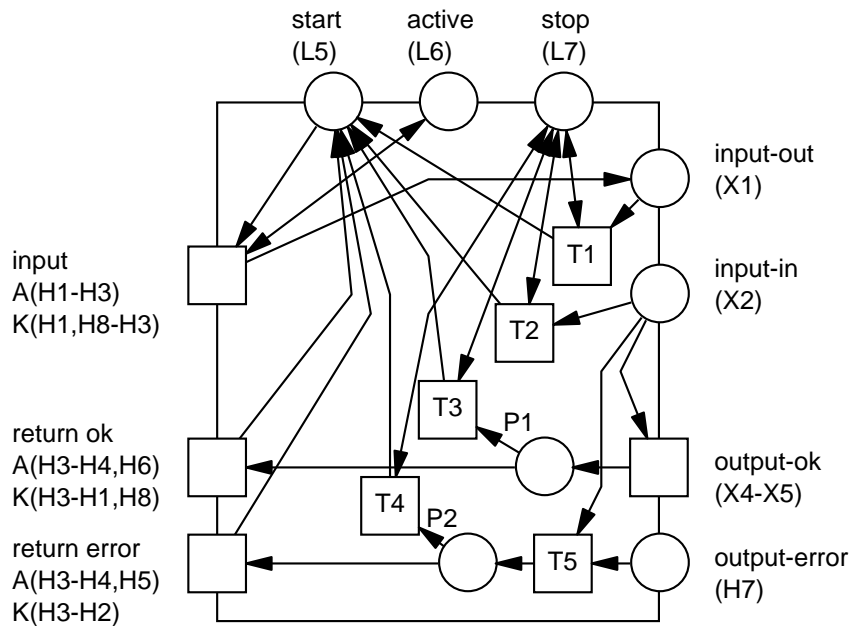


Abbildung 4.17: Petrinetz-Baustein für *TimeoutHandler*

Subschnittstelle 1 enthält *kill finished*, *input*, *catch error* und *throw error*. Diese Schnittstelle wird mit der Schnittstelle 4 des *Ereigniskontext* oder Schnittstelle 4 des *Choice* verknüpft (siehe Bezeichnungen *K* und

A). Über diese Verknüpfung wird dieser *TimeoutHandler* gestartet und sein Ergebnis zurückgeliefert.

Subschnittstelle 2 ist wie bei *ExceptionHandler* eine Teilmenge von Subschnittstelle 1 und beinhaltet *input* und *kill finished*. Mit *input* wird der kill-Befehl an *Lebenszykluskontext 2* im aktuellen Kontext gegeben (siehe Abbildung 4.7 auf Seite 30). Mit *kill finished* wird auf die Bestätigung der Ausführung des kill-Befehls gewartet. Im Anschluss daran wird die Fehlerbehandlung gestartet.

Subschnittstelle 3 enthält die Plätze *start*, *active* und *stop*. Über sie wird die Verknüpfung zu einem Baustein vom Typ *Lebenszykluskontrolle* realisiert.

Subschnittstelle 4 umfasst die Plätze *input-out*, *input-in* und *output-error*, sowie die Transition *output-ok*. Diese werden mit der Aktivität verknüpft, die als Reaktion auf die Zeitüberschreitung ausgeführt werden soll. Über *output-ok* und *output-error* wird das Ergebnis der Aktivität empfangen.

4.6 Komplexe Aktivitäten

In *WSCI* gibt es komplexe Aktivitäten. Sie betten andere Aktivitäten desselben Prozesses ein und steuern diese. Komplexe Aktivitäten stellen somit einen Gegensatz zu den atomaren Aktivitäten dar. Zu den komplexen Aktivitäten gehören *All*, *Choice*, *Foreach*, *Sequence*, *Switch*, *Until* und *While*.

Für jede dieser Aktivitäten entwickeln wir einen Baustein und definieren jeweils zwei Schnittstellen im Voraus. Schnittstelle 1 jedes Bausteines umfasst *input*, *output-out* und *output-in*. Sie wird mit einem Baustein verknüpft, der die Abarbeitung des aktuellen Bausteins steuert. Das kann der Baustein einer anderen komplexen Aktivität oder der Baustein *Lebenszyklusprozess* sein. Über *input* wird der Baustein gestartet, die übrigen Elemente dieser Schnittstelle beenden die Abarbeitung dieses Bausteins. Bei *input* unterscheiden wir, ob die Aktivität direkt mit *Lebenszyklusprozess* oder mit einer anderen komplexen Aktivität verknüpft wurde. Schnittstelle 2 enthält die Plätze *start*, *active* und *stop*. Diese werden mit dem zugehörigen Baustein vom Typ *Lebenszykluskontrolle* verbunden. Diese Schnittstellen werden in allen komplexen Aktivitäten vorausgesetzt und nicht noch einmal beschrieben.

4.6.1 All

WSCI definiert die komplexe Aktivität *All*. Diese führt alle enthaltenen Aktivitäten nebenläufig aus. Das können beliebig viele sein. Wenn ein in einer der Aktivitäten auftretender Fehler erfolgreich behandelt wurde, wird die Abarbeitung von *All* normal fortgesetzt. Dieses Verhalten bilden wir mit dem Baustein *All* nach. Dieser Baustein ermöglicht die nebenläufige Abarbeitung von nur zwei Aktivitäten. Sollen mehr als zwei Aktivitäten nebenläufig ausgeführt werden, kombinieren wir entsprechend viele Bausteine dieses Typs. In Abbildung 4.18 sehen wir den Baustein *All*. Wir fügen in die Menge der Schnittstellen zwei weitere Elemente ein.

Schnittstelle 3 enthält *input1-out*, *input1-in*, *output1* und *output1-finished*. Diese werden mit Schnittstelle 1 der anzuschließenden Aktivität verbunden. Über *output1-finished* wird dem für die anzuschließende Aktivität zuständigen *Ereigniskontext* ermöglicht, den Ablauf vom *All* fortzusetzen. Das ist notwendig, wenn in der enthaltenen Aktivität ein aufgetretener Fehler erfolgreich bearbeitet wurde.

Schnittstelle 4 verhält sich gegenüber der zweiten anzuschließenden Aktivität genauso wie Schnittstelle 3 gegenüber der ersten. Sie enthält alle Elemente der Schnittstelle mit der Zahl 2 im informalen Teil des Namens. In dieser Schnittstelle wurde der Bezeichner *X3* zweimal vergeben. Er dient

dazu, *All* nach einem erfolgreich behandelten Fehler in einer enthaltenen Aktivität fortsetzen zu können. Für Subchnittstelle 4 ist *X3* mit *output2-finished* gleichzusetzen. Gleiches gilt für den Baustein *Sequence*.

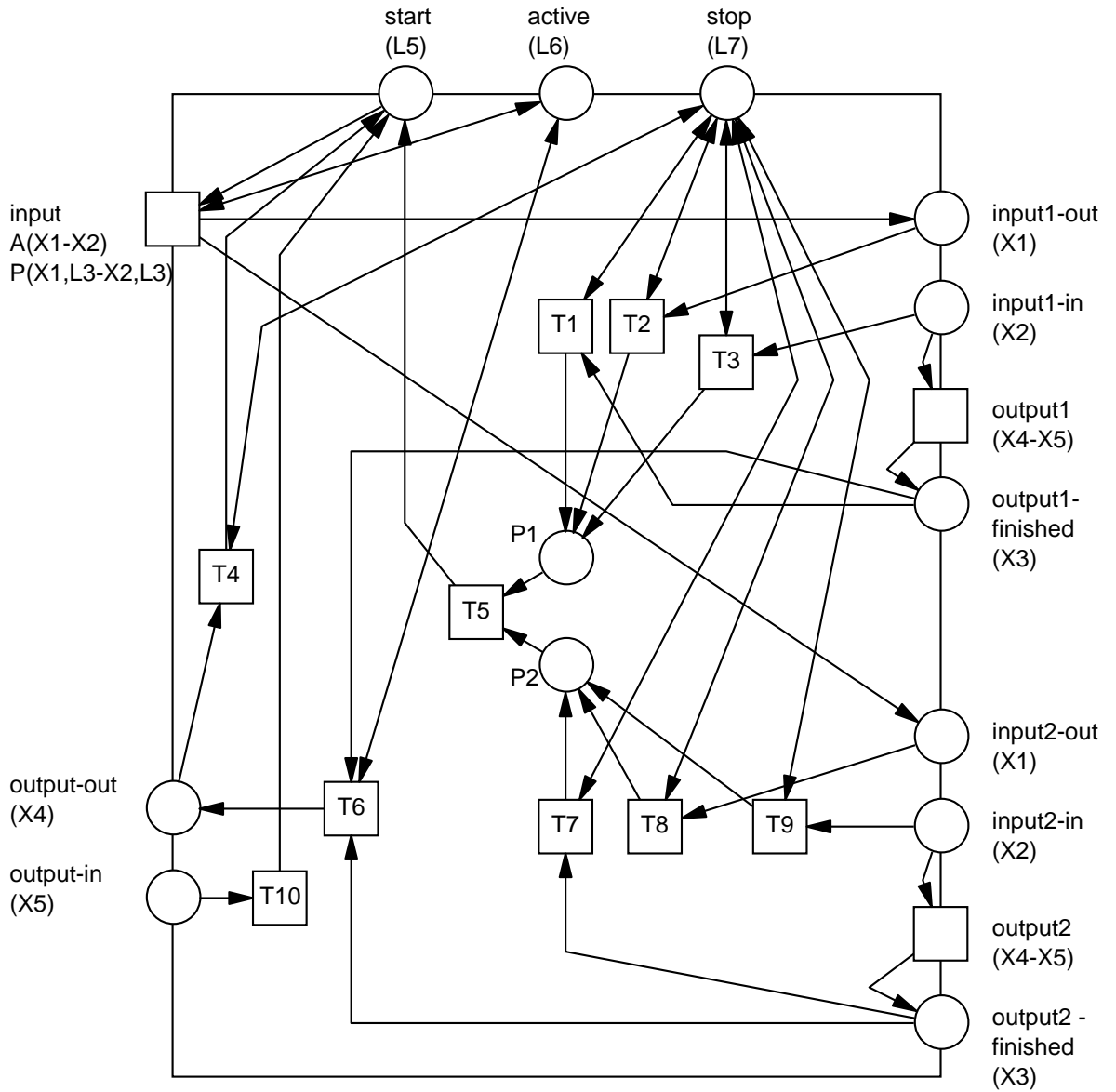


Abbildung 4.18: Petrinetz-Baustein für *All*

4.6.2 Choice

In *WSCI* ist das Element *Choice* definiert. Mit den Elementen *onMessage*, *onTimeout* und *onFault* legen wir die Art der möglichen Ereignisse fest. *Choice* führt genau eines der enthaltenen Elemente aus. Falls sowohl ein Element *Exception* als auch ein Element aus *Choice* auf ein auftretendes Ereignis reagieren kann, wird dem Element aus *Choice* laut Spezifikation der Vorrang gegeben. Dieser Teil der Spezifikation ist sehr schwer umzusetzen. Es wird nicht gesagt, ob das auch für Ereignisse gilt, die aus Kinderkontexten weitergeleitet wurden. Ebenso unspezifiziert ist die Aufteilung der Abarbeitung, wenn zwei Ereignisse gleichzeitig auftreten (z.B. Empfang einer Nachricht und Überschreitung einer Zeitschranke). Aus diesen Gründen haben wir diese Vorrangregelung nicht umgesetzt. Könnten mehrere Elemente in *Choice* auf ein eintretendes Ereignis reagieren, so ist der *WebService* laut Spezifikation mehrdeutig. Durch die Umsetzung dieser Mehrdeutigkeit wird im Baustein Nichtdeterminismus erreicht.

Mit dem Baustein *Choice* modellieren wir die Auswahl des abzuarbeitenden Reaktionsverhaltens. *Choice* reagiert auf ein auftretendes Ereignis mit EreignisHandlern (siehe *Ereigniskontext*). Die EreignisHandler entsprechen hier den Elementen *onMessage*, *onTimeout* und *onFault*.

Das eingetretene Ereignis bestimmt, welcher EreignisHandler ausgeführt wird. Mit *Choice* kann prinzipiell auf die gleichen Ereignisse reagiert werden, auf die auch ein *Ereigniskontext* reagieren kann. Allerdings werden hier die entsprechenden *Lebenszykluskontexte* bei Beginn der Abarbeitung nicht gestoppt. Nach der Ausführung eines EreignisHandlers wird *Choice* beendet.

Abbildung 4.19 zeigt den Baustein *Choice*. Wir definieren zwei weitere Schnittstellen für diesen Baustein.

Subschnittstelle 3 umfasst nur die Transition *exception*. An sie wird der zuständige *Ereigniskontext* oder ein EreignisHandler angeschlossen, über sie wird ein aufgetretener Fehler weitergeleitet. Diese Schnittstelle wird nur dann mit einem EreignisHandler verknüpft, wenn *Choice* als Reaktion auf ein eingetretenes Ereignis auszuführen ist.

Die Plätze *event-out*, *event in progress*, *event-in*, *return-ok* und *return-error* stellen Schnittstelle 4 dar. An sie werden ähnlich wie beim *Ereigniskontext* alle EreignisHandler angeschlossen.

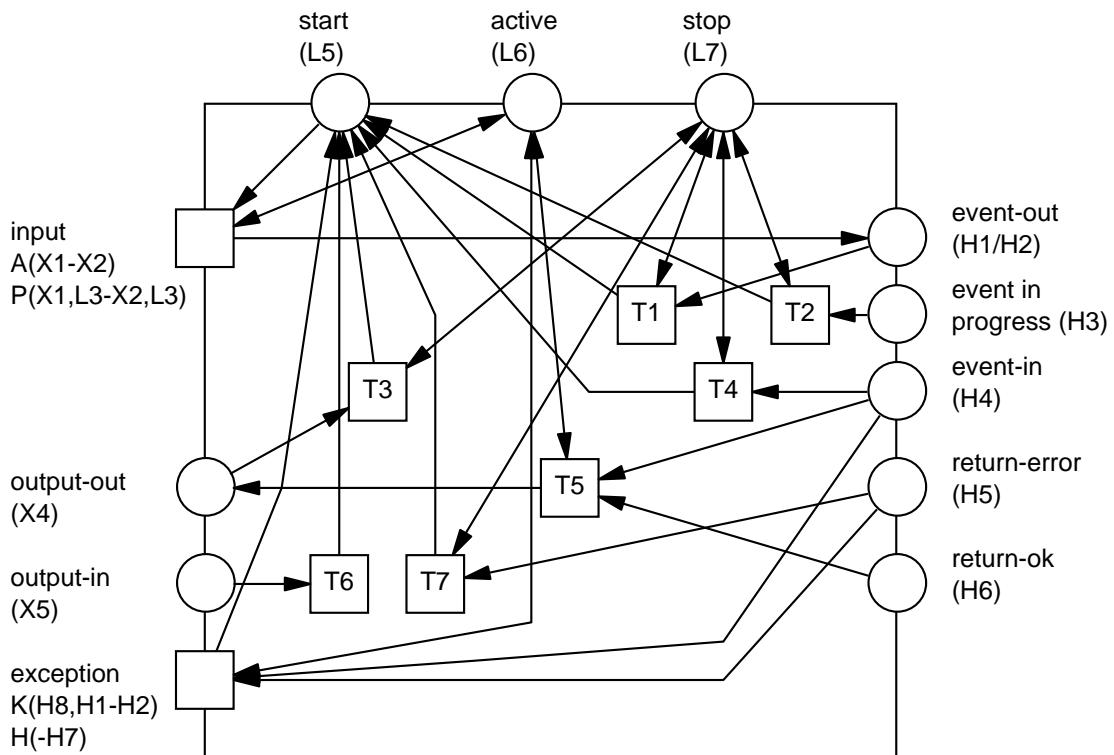


Abbildung 4.19: Petrinetz-Baustein für *Choice*

4.6.3 Foreach

Das Element *Foreach* der Sprache *WSCl* arbeitet die angeschlossene Aktivität für jedes Element einer vorgegebenen Menge genau einmal ab. Diese Menge ist in *WSCl* jedoch nicht näher spezifiziert. In unserem Baustein *Foreach* setzen wir die abzuarbeitende Menge deshalb nicht um. Dadurch entsteht Nichtdeterminismus im entwickelten Baustein. Wir können diese Menge jedoch in den Aufbau des Bausteins einfügen. Das ist an den Transitionen $T3$ und $T5$ möglich. Durch sie wird die wiederholte Abarbeitung erneut gestartet oder gestoppt.

Den Baustein zu *Foreach* sehen wir in Abbildung 4.20. Über die Transition $T3$ wird der Start der angeschlossenen Aktivitäten (ein weiteres Mal) ermöglicht. Die Transition $T5$ beendet die Schleife dann, wenn alle Elemente abgearbeitet wurden. Die Entscheidung, die abzuarbeitende Menge nicht zu modellieren, bedeutet Nichtdeterminismus zwischen den Transitionen $T3$ und $T5$. Alle mit *stop* verbundenen Transitionen dienen zum Abräumen der Marken im Fehlerfall. Wir definieren weiterhin eine zusätzliche Schnittstelle für diesen Baustein.

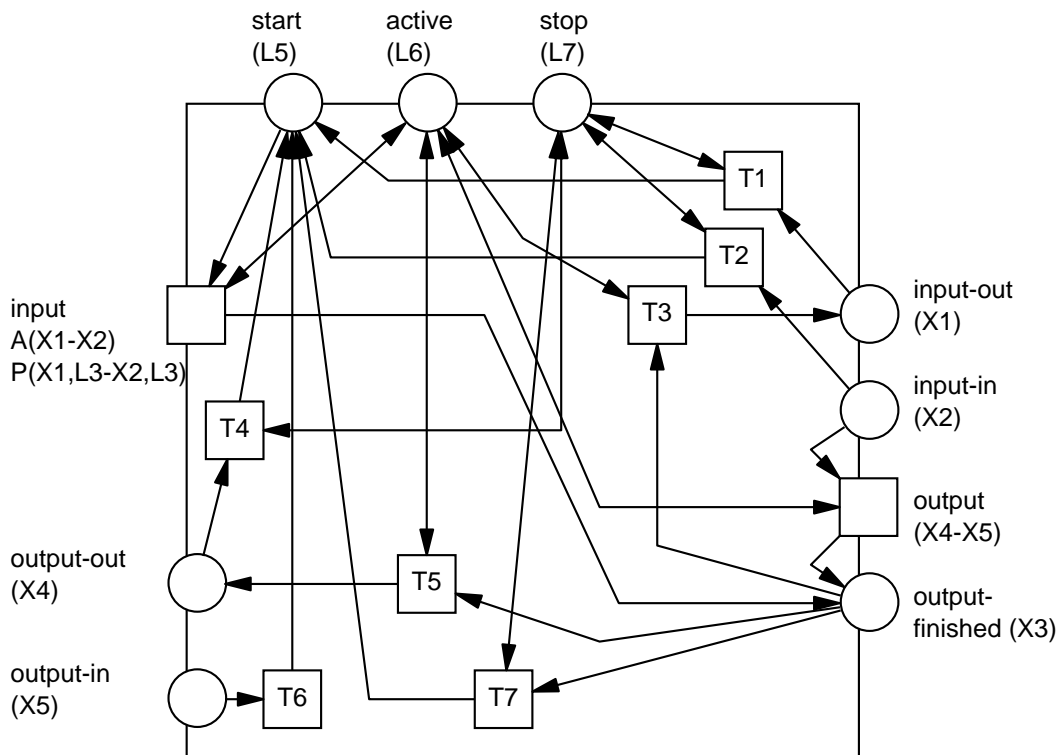


Abbildung 4.20: Petrinetz-Baustein für *Foreach*

Subschnittstelle 3 enthält *input-out*, *input-in*, *output* und *output-finished*. Sie wird benutzt, um *Foreach* mit der anzuschließenden Aktivität zu verknüpfen.

4.6.4 Sequence

In *WSCI* führt das Element *Sequence* alle enthaltenen Aktivitäten nacheinander in der Reihenfolge ihrer Definition aus. Das können beliebig viele sein. Für den Fall eines in einer enthaltenen Aktivität auftretenden, erfolgreich behandelten Fehlers wird die Abarbeitung von *Sequence* fortgesetzt. Wir setzen dieses Verhalten mit dem Baustein *Sequence* um. Ein Baustein dieser Art kombiniert nur zwei Aktivitäten sequentiell. Wenn wir mehr als zwei Aktivitäten hintereinander ausführen wollen, kombinieren wir entsprechend mehrere Bausteine dieses Typs. Die Abbildung 4.21 zeigt den Baustein. Wir definieren zwei weitere Subschnittstellen.

In Subschnittstelle 3 sind *input1-out*, *input1-in*, *output1* und *input2-out* enthalten. Die erste der beiden auszuführenden Aktivitäten wird mit dieser Subschnittstelle verknüpft. Über *input1-out* und *input1-in* wird die ange-

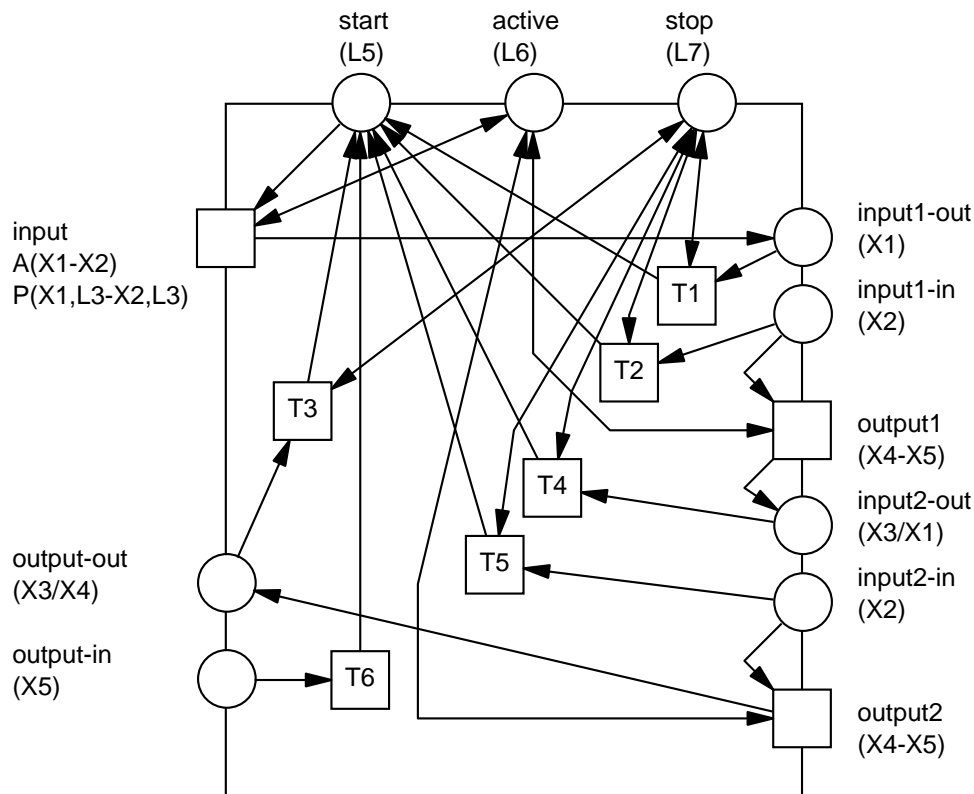


Abbildung 4.21: Petrinetz-Baustein für *Sequence*

geschlossene Aktivität gestartet. Das Schalten der Transition *output1* beendet die erste auszuführende Aktivität erfolgreich. Über *input2-out* wird die Ausführung von *Sequence* im Fall eines erfolgreich behandelten Fehlers in der ersten auszuführenden Aktivität fortgesetzt.

Die zweite auszuführende Aktivität wird mit Subchnittstelle 4 verknüpft und genauso behandelt wie die an Subchnittstelle 3 angeschlossene Aktivität. Zu ihr zählen *input2-out*, *input2-in*, *output2* und *output-out*. In dieser Subchnittstelle wurde der Bezeichner *X3* zweimal vergeben, hier ist lediglich *output-out* mit *X3* gleichzusetzen. *input2-out* erhält nur in Subchnittstelle 3 den Bezeichner *X3*.

4.6.5 Switch

Das Element *Switch* führt in *WSCI* eine spezielle Aktivität aus einer Menge von mehreren Aktivitäten aus. Diese werden durch Elemente vom Typ *Case* dargestellt und beinhalten das auszuführende Verhalten. Die Auswahl basiert auf einer Bedingung, die in *WSCI* durch das Element *Condition* dargestellt

wird. Dieses Element beinhaltet einen Ausdruck, der zu wahr oder falsch ausgewertet wird. Er ist jedoch nicht näher spezifiziert, weshalb wir ihn nicht in unser Modell übernehmen. An dieser Stelle entsteht Nichtdeterminismus in unserem Baustein. Weiterhin haben wir die Unterteilung in *Switch* und *Case* nicht vorgenommen. In *WSCI* steuert *Switch* lediglich die Überprüfung der Elemente vom Typ *Case*. Da diese Überprüfung sequentiell entsprechend der Reihenfolge ihrer Definition ausgeführt wird, können wir *Switch* implizit durch die sequentielle Verknüpfung der *Case*-Elemente erreichen. Wir benötigen also nur einen Baustein zur Darstellung beider Elemente. Wir beschränken uns daher auf einen Baustein *Switch*, der mit weiteren Bausteinen vom Typ *Switch* sequentiell verknüpft wird. Das entsprechend der Bedingung auszuführende Verhalten stellen wir durch eine Aktivität dar.

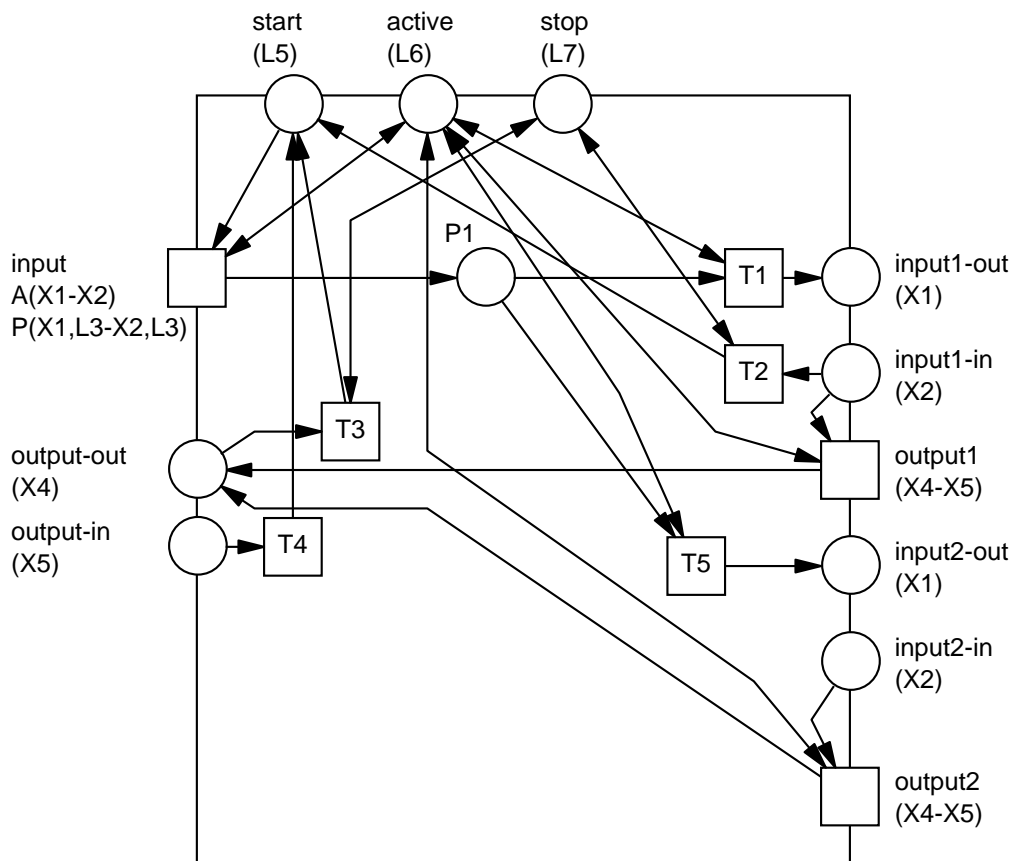


Abbildung 4.22: Petrinetz-Baustein für *Switch*

Abbildung 4.22 stellt den Baustein *Switch* dar. Jede zur Wahl stehende Aktivität wird mit einem Baustein dieser Art verbunden. Die Bausteine vom Typ *Switch* werden wiederum untereinander in der Reihenfolge der Definition

der damit verbundenen Aktivitäten verknüpft. Wir definieren zusätzlich zu den zwei allgemein für komplexe Aktivitäten definierten Subchnittstellen zwei weitere Subchnittstellen.

Subchnittstelle 3 enthält die Plätze *input1-out*, *input1-in* und *output1*. Über sie wird *Switch* mit einem weiteren Baustein vom Typ *Switch* verknüpft.

Subchnittstelle 4 umfasst *input2-out*, *input2-in* und *output2* und verknüpft *Switch* mit der auszuführenden Aktivität.

Ist *P1* markiert, schaltet eine der Transitionen *T1* oder *T5*, je nachdem ob die Bedingung erfüllt ist oder nicht. Die Bedingung wird in unserem Baustein nicht modelliert. Dadurch entsteht im Baustein Nichtdeterminismus. Ist die Bedingung für keinen Fall erfüllt, endet *Switch*, ohne eine Aktivität ausgeführt zu haben.

4.6.6 Until

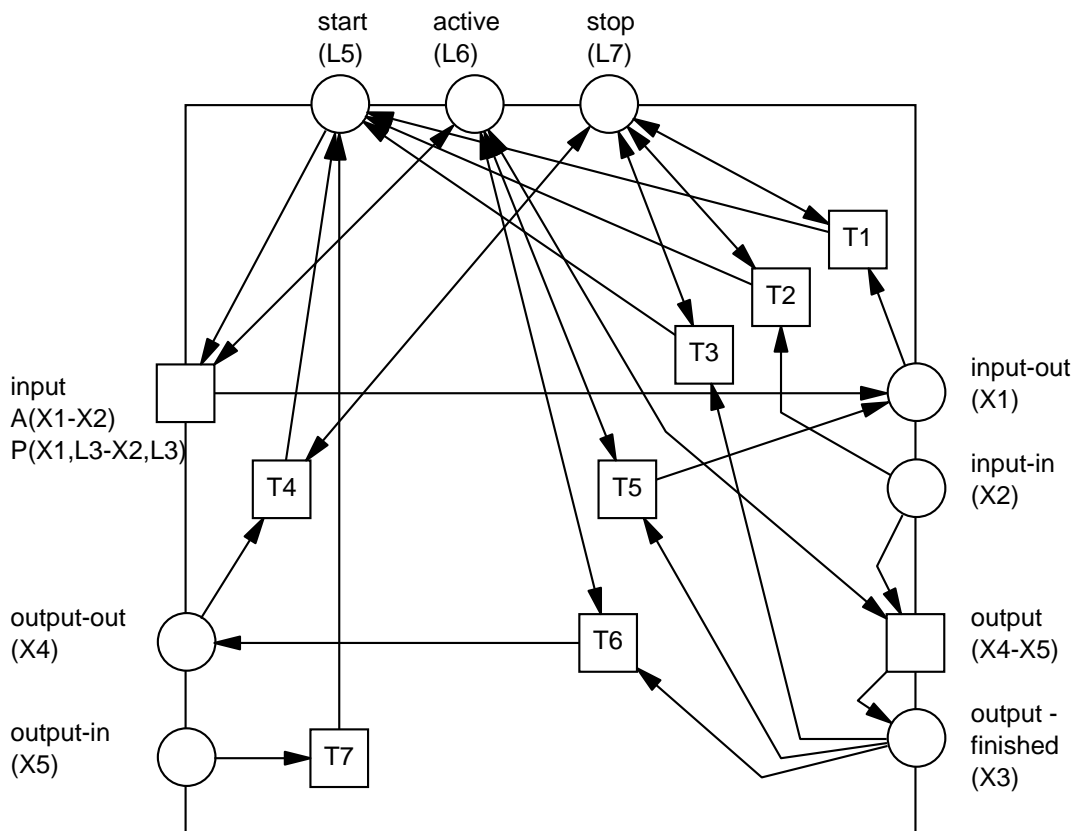


Abbildung 4.23: Petrinetz-Baustein für *Until*

In *WSCI* führt das Element *Until* die angeschlossene Aktivität so oft

hintereinander aus, wie eine bestimmte Bedingung erfüllt ist, mindestens jedoch einmal. Die Bedingung wird wie im Element *Switch* durch ein Element *Condition* dargestellt. Dessen Inhalt beinhaltet einen Ausdruck, der zu wahr oder falsch ausgewertet werden muss, jedoch nicht näher spezifiziert ist. Deshalb modellieren wir dieses Element nicht. So entsteht Nichtdeterminismus im Baustein *Until*. Durch eine genauere Modellierung der Transitionen *T5* und *T6* kann der Nichtdeterminismus beseitigt werden.

Abbildung 4.23 visualisiert den Baustein *Until*. Wenn nach einer Ausführung die Transition *T5* schaltet, so wird die angeschlossene Aktivität erneut ausgeführt. Schaltet die Transition *T6*, so wird *Until* beendet. Alle mit *stop* verbundenen Transitionen räumen im Fehlerfall die restlichen Marken ab. Wir definieren eine weitere Schnittstelle.

In Schnittstelle 3 sind *input-out*, *input-in*, *output* und *output-finished*. Die durch *Until* wiederholt auszuführende Aktivität wird über diese Schnittstelle mit *Until* verknüpft.

4.6.7 While

Das *WSCI*-Element *While* arbeitet die angeschlossene Aktivität so lange ab, wie eine bestimmte Bedingung gilt. Wenn die Bedingung von Anfang an nicht erfüllt ist, wird *While* ohne eine Ausführung der angeschlossenen Aktivität beendet. Wie für *Switch* und *Until* beschrieben, wird die Bedingung für die Ausführung auch im Baustein *While* nicht modelliert. Das führt zu Nichtdeterminismus zwischen den Transitionen *T4* und *T6*.

Abbildung 4.24 zeigt den Baustein. Die Transition *T4* wird geschaltet, wenn die Bedingung für eine weitere Ausführung der angeschlossenen Aktivität erfüllt ist. Das hat eine Ausführung der angeschlossenen Aktivität zur Folge. Ist die Bedingung nicht erfüllt, wird die Transition *T6* geschaltet und *While* erfolgreich beendet. Alle mit *stop* verbundenen Transitionen räumen im Fehlerfall die restlichen Marken ab. Wir definieren eine zusätzliche Schnittstelle.

In Schnittstelle 3 sind die Plätze *input-out*, *input-in* und *output-finished*, sowie die Transition *output* enthalten. Über sie werden die wiederholt auszuführende Aktivität angeschlossen. *input-out* und *input-in* starten die Aktivität, *output* beendet sie. Über *output-finished* kann der *Ereigniskontext* der angeschlossenen Aktivität für den Fall eines erfolgreich behandelten Fehlers das Verhalten in *While* fortsetzen.

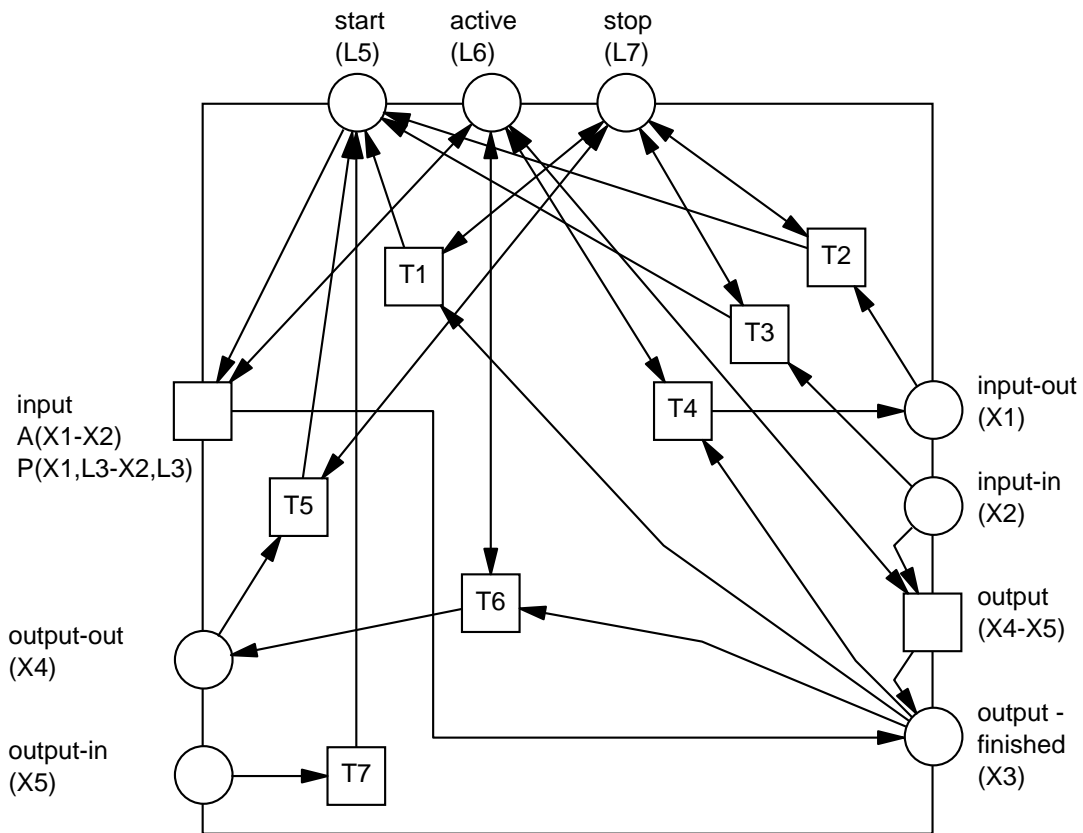


Abbildung 4.24: Petrinetz-Baustein für *While*

4.7 Atomare Aktivitäten

Die Spezifikation für *WSCI* enthält atomare Aktivitäten. Diese sind nicht in weitere Aktivitäten unterteilbar. Sie stellen somit einen Gegensatz zu den komplexen Aktivitäten dar. Zu den atomaren Aktivitäten gehören *Action*, *Delay*, *Empty*, *Fault*, *Call*, *Spawn* und *Join*.

Wir stellen im Folgenden für jedes dieser Elemente einen Baustein vor. Bei der Namensgebung der Schnittstellenelemente haben wir auf gleiche Namen für gleiche Funktionalität geachtet. Die meisten atomaren Aktivitäten stellen ein Blatt im Hierarchiebaum des Prozesses dar. Auf die Ausnahmen gehen wir konkret ein. Wir stellen jetzt die am häufigsten auftretenden Schnittstellenelemente vor und gehen kurz auf ihre Funktion ein. Wenn ein Baustein über das beschriebene Element verfügt, ist die jetzt folgende Erklärung der Funktion für ihn gültig.

Der Ablauf eines Bausteines wird über die Transition *input* gestartet und über die Plätze *output-out* und *output-in* fehlerfrei beendet. Sie werden mit

dem Baustein verbunden, der die Ausführung der Aktivität starten und beenden soll (eine komplexe Aktivität oder *Lebenszyklusprozess*). Mit der Transition *exception* verbinden wir die Aktivität mit dem *Ereigniskontext* oder dem entsprechenden EreignisHandler. Letzteres geschieht nur, wenn diese atomare Aktivität als Reaktion auf ein eingetretenes Ereignis ausgeführt wird. Über die Transition *exception* wird ein Fehler erzeugt und die entsprechende Fehlerbehandlung im umschließenden Kontext ausgelöst. Dazu wird auch die Marke vom Platz *ready for end or kill* vom *Lebenszykluskontext 1 (H8)* des aktuellen Kontextes konsumiert (siehe Seite 40).

Über die Transition *receive* wird eine Nachricht empfangen. Sie wird mit den Plätzen *send-out* und *send-in* des sendenden Bausteines verknüpft. Entsprechend wird über *send-out* und *send-in* eine Nachricht gesendet und mit *receive* des empfangenden Bausteines verknüpft. Die einzige Ausnahme stellt der Start eines Prozesses durch den Empfang einer Nachricht dar. Die sendende Aktivität wird dann mit *receive* von *Nachrichtempfang* verbunden. Die empfangenden Aktivitäten werden mit *send-out* und *send-in* von *Nachrichtempfang* verknüpft.

Die Plätze *start*, *active* und *stop* werden mit der Schnittstelle 2 des angeschlossenen Bausteines *Lebenszykluskontrolle* verknüpft. Ihre Markierung ermöglicht oder verhindert die Ausführung des jeweiligen Bausteines.

4.7.1 Action

In *WSCI* definiert das Element *Action* die Ausführung einer Operation aus *WSDL*. Sie bekommt eine Rolle zugewiesen (Sender/Empfänger) und kann mit jeder der folgenden vier Operationen aus *WSDL* verknüpft werden:

1. *Notification* (Versand einer Nachricht)
2. *One-Way* (Empfang einer Nachricht)
3. *Request-Response* (erst Empfang und dann Versand einer Nachricht)
4. *Solicit-Response* (erst Versand und dann Empfang einer Nachricht)

An eine *Action*, die mit einer *WSDL*-Operation (*Operation*) vom Typ *Request-Response* verbunden ist, kann ein Prozess angehängt werden. Dieser beginnt nach Empfang der Nachricht und endet vor dem Senden der Antwort. Tritt in dem Prozess ein Fehler auf, so schlägt auch *Action* fehl. Im Folgenden stellen wir die Bausteine für jede der vier angegebenen *WSDL*-Operationen vor.

Notification

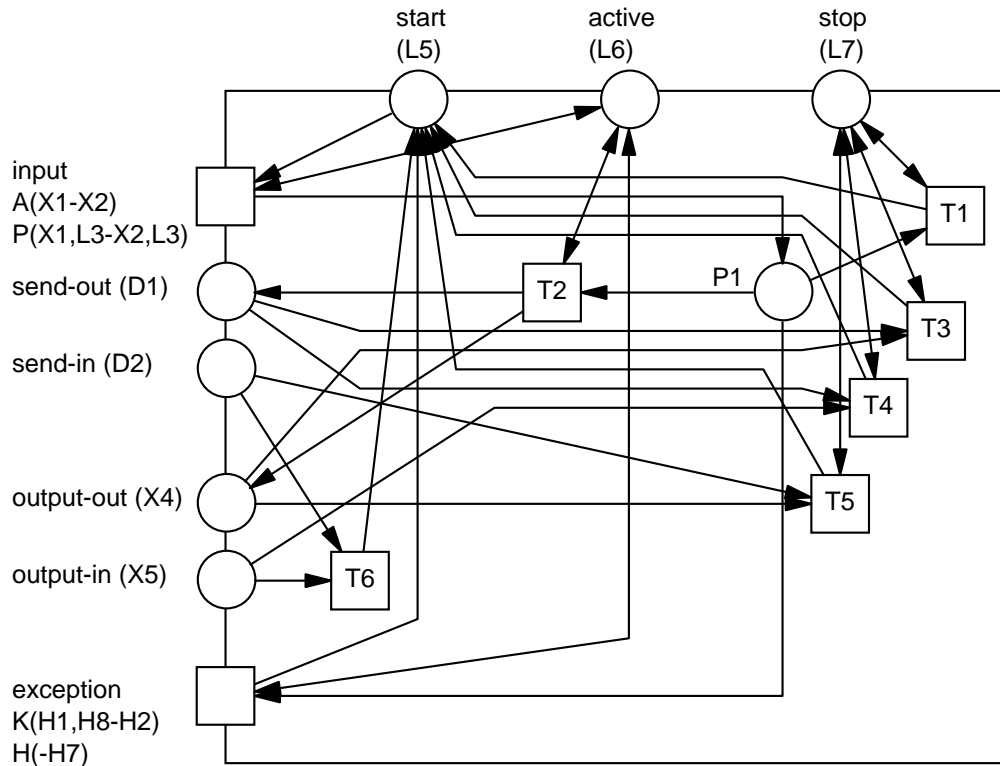


Abbildung 4.25: Petrinetz-Baustein für *Notification*

Abbildung 4.25 zeigt den Baustein für eine *Action*, die mit der *WSDL*-Operation *Notification* verknüpft wurde. Mit *Notification* wird eine Nachricht gesendet. Die Transition *T2* versendet die Nachricht und legt sie auf den Übertragungskanal (Platz *send-out*), von wo aus sie durch den Partner konsumiert werden kann. Nach dem Abholen der Nachricht wird eine Marke auf den Platz *send-in* gelegt. Alternativ kann dem angeschlossenen Kontext über die Transition *exception* das Auftreten eines Fehlers signalisiert werden. Die *Action* kann jederzeit abgebrochen werden. Das geschieht, wenn die angeschlossene *Lebenszykluskontrolle* die Marke auf dem Platz *active* konsumiert und auf dem Platz *stop* eine Marke erzeugt. Abhängig vom bereits angenommenen Zustand schaltet jetzt eine der Transitionen *T1*, *T3*, *T4* oder *T5*. Nach dem Schalten einer dieser Transitionen sind nur noch die Plätze *start* und *stop* belegt. In diesem Zustand kann innerhalb der *Action* keine Transition mehr schalten.

One-Way

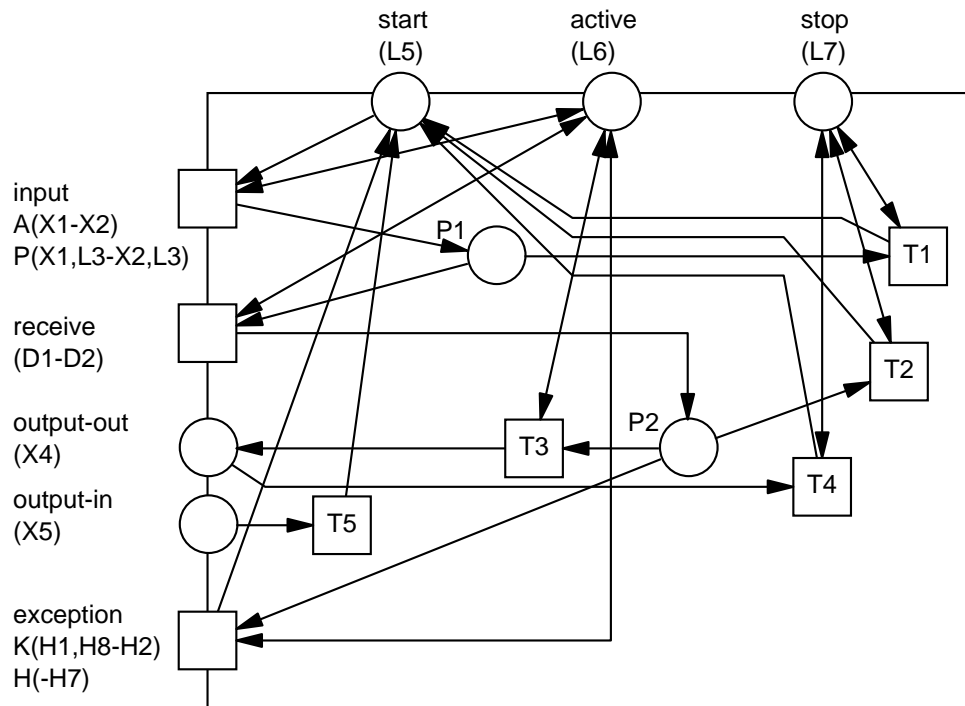


Abbildung 4.26: Petrietz-Baustein für *One-Way*

Abbildung 4.26 zeigt den Baustein für eine *Action*, die mit der *WSDL*-Operation *One-Way* verknüpft wurde. Mit *One-Way* wird eine Nachricht empfangen. Die Transition *receive* holt eine auf einem Übertragungskanal abgelegte Nachricht ab. Im Fall eines Fehlers wird dem *Ereigniskontext* über die Transition *exception* das Auftreten eines Fehlers signalisiert. Falls die *Action* durch den angeschlossenen Baustein *Lebenszykluskontrolle* abgebrochen wird, schaltet eine der Transitionen *T1*, *T2* oder *T4*. Danach sind nur die Plätze *start* und *stop* belegt und keine Transition in dieser Aktivität kann mehr schalten.

Request-Response

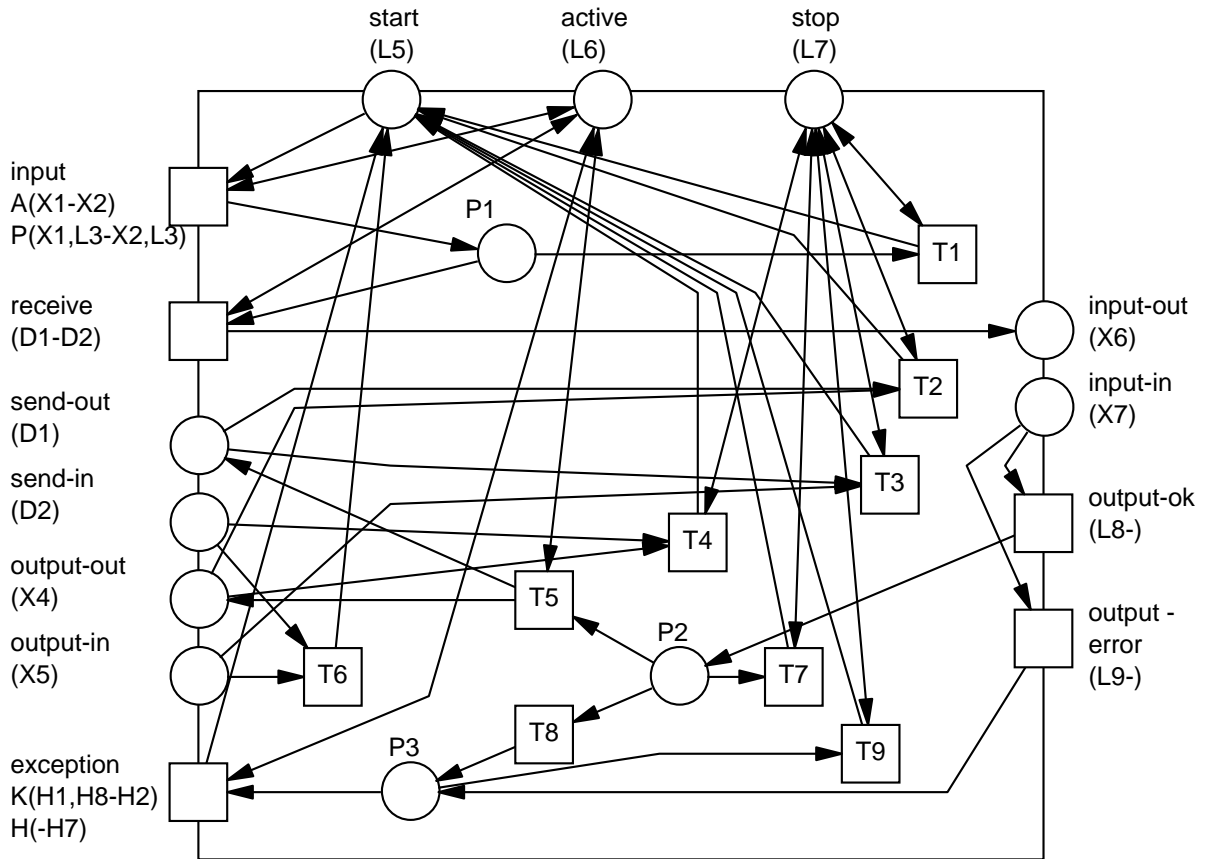


Abbildung 4.27: Petrinetz-Baustein für *Request-Response*

Der Baustein in Abbildung 4.27 modelliert eine *Action*, die mit der *WSDL*-Operation *Request-Response* verknüpft wurde. Durch *Request-Response* wird erst eine Nachricht empfangen und danach eine Nachricht gesendet. Über die Plätze *input-out* und *input-in* und über die Transition *output* können angeschlossene Prozesse ausgeführt werden. Mit der Transition *T5* wird die zu sendende Nachricht auf den Nachrichtenübertragungskanal (Platz *send-out*) gelegt. Nach dem Abholen der Nachricht wird eine Marke auf den Platz *send-in* gelegt. Alternativ kann dem angeschlossenen Kontext über die Transition *exception* das Auftreten eines Fehlers signalisiert werden. Wenn die *Action* durch den angeschlossenen Baustein vom Typ *Lebenszykluskontrolle* abgebrochen wird, schaltet eine der Transitionen *T1*, *T2*, *T3*, *T4* oder *T7*. Diese sorgen dafür, dass danach nur noch die Plätze *start* und *stop* belegt sind. In diesem Zustand kann innerhalb der Aktivität keine Transition mehr schalten.

Solicit-Response

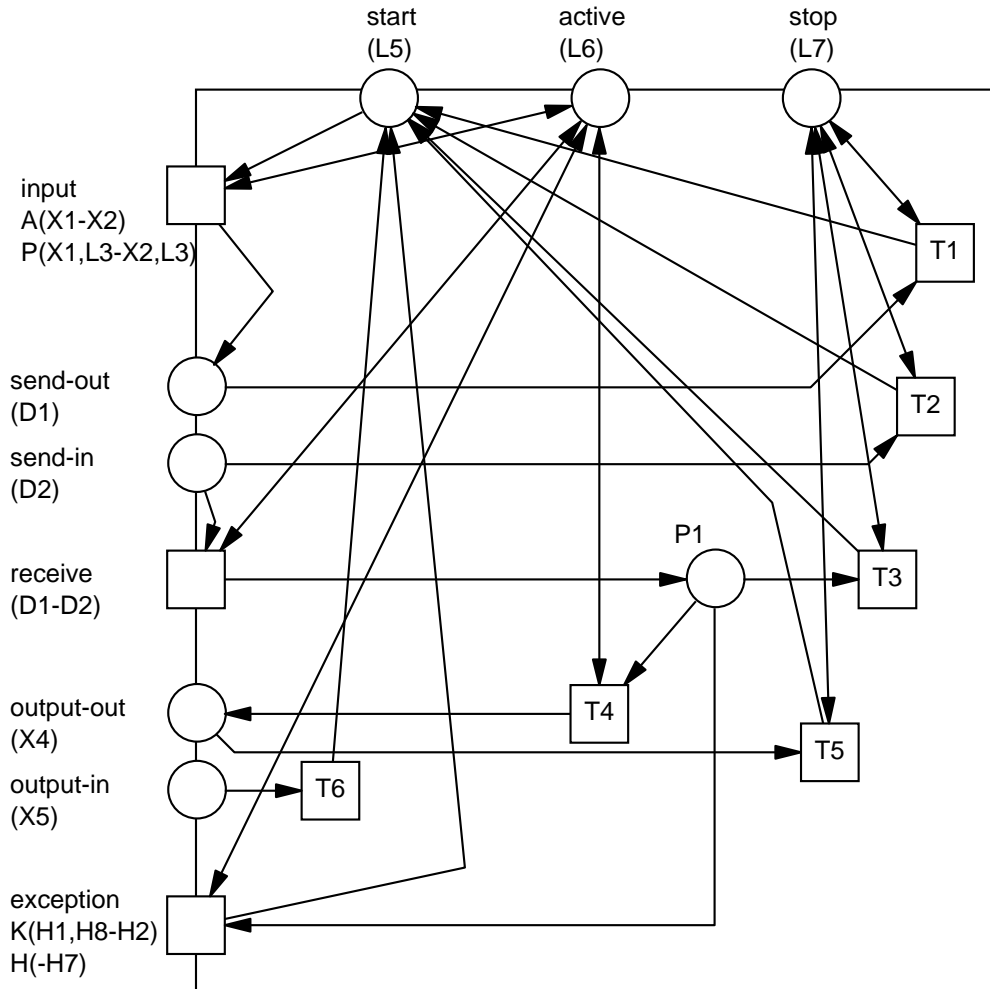


Abbildung 4.28: Petrinetz-Baustein für *Solicit-Response*

Abbildung 4.28 zeigt den Baustein einer mit der *WSDL*-Operation *Solicit-Response* verknüpften *Action*. *Solicit-Response* sendet zuerst eine Nachricht und empfängt danach eine Nachricht. Durch die Transition *input* wird die zu sendende Nachricht auf den Nachrichtenkanal gelegt (Platz *send-out*). Wenn auf dem Platz *send-in* die Bestätigung für das Abholen der Nachricht durch den Partner eingetroffen ist, kann die Aktivität durch die Transition *receive* selbst eine Nachricht empfangen. Alternativ kann dem angeschlossenen *Ereigniskontext* über die Transition *exception* das Auftreten eines Fehlers mitgeteilt werden. Eine der Transitionen $T1$, $T2$, $T3$ und $T5$ schaltet, falls dieser Baustein durch *Lebenszykluskontrolle* gestoppt wird.

4.7.2 Delay

Das *WSCI*-Element *Delay* verzögert den Kontrollfluss um eine angegebene Zeit. Es kann kein Fehler erzeugt werden. Der Einfachheit halber beschränken wir uns in unserem Modell auf den kausalen Zusammenhang, da wir nicht bestimmen können, wie viel Zeit die Ausführung dieser Aktivität wirklich in Anspruch nehmen wird.

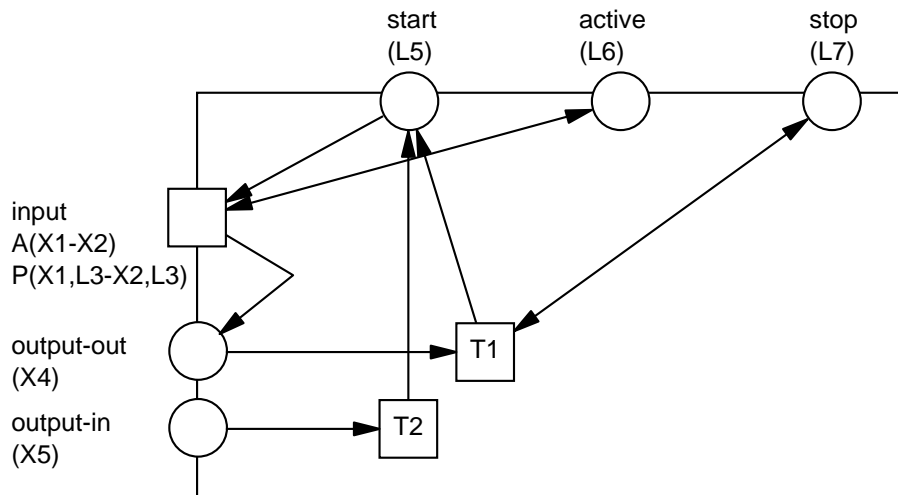


Abbildung 4.29: Petrinetz-Baustein für *Delay*

In Abbildung 4.29 sehen wir den Baustein zu *Delay*. Die Transition *input* erzeugt auf *output-out* eine Marke und ermöglicht somit direkt die Beendigung von *Delay*.

4.7.3 Empty

In *WSCI* ist das Element *Empty* leer. Es führt keine Aktion aus und ist als Platzhalter für später zu definierende Aktivitäten gedacht. Es ist nicht möglich, einen Fehler zu erzeugen.

Der entsprechende Baustein *Empty* leitet lediglich den Kontrollfluss weiter. Der Baustein *Empty* wird in Abbildung 4.30 visualisiert.

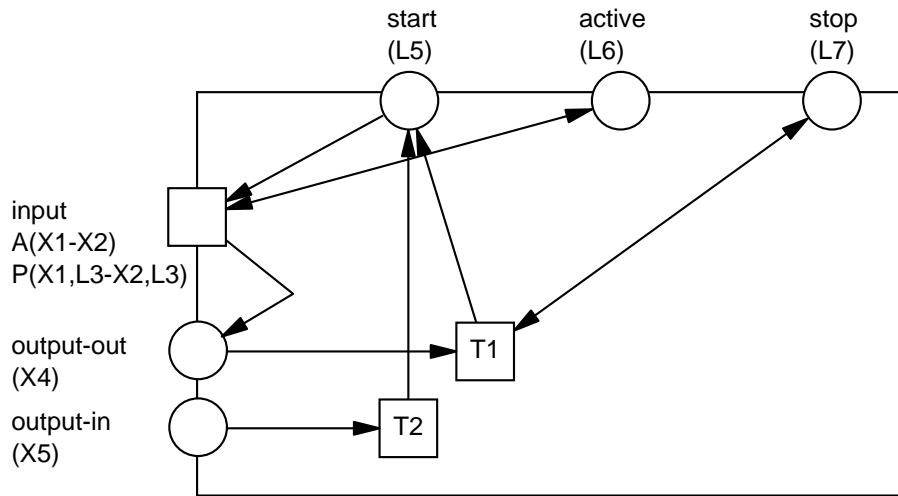


Abbildung 4.30: Petrinetz-Baustein für *Empty*

4.7.4 Fault

Das *WSCI*-Element *Fault* erzeugt einen Fehler. Dieser wird an den aktuellen Kontext gereicht und dort behandelt. Diese Aktivität wird häufig benutzt, um bei einer Zeitüberschreitung in *Exception* einen Fehler zu erzeugen.

In unserem Modell leitet der Baustein *Fault* den Fehler über die Transition *exception* an den entsprechenden *Ereigniskontext* weiter. Abbildung 4.31 zeigt den entsprechenden Baustein.

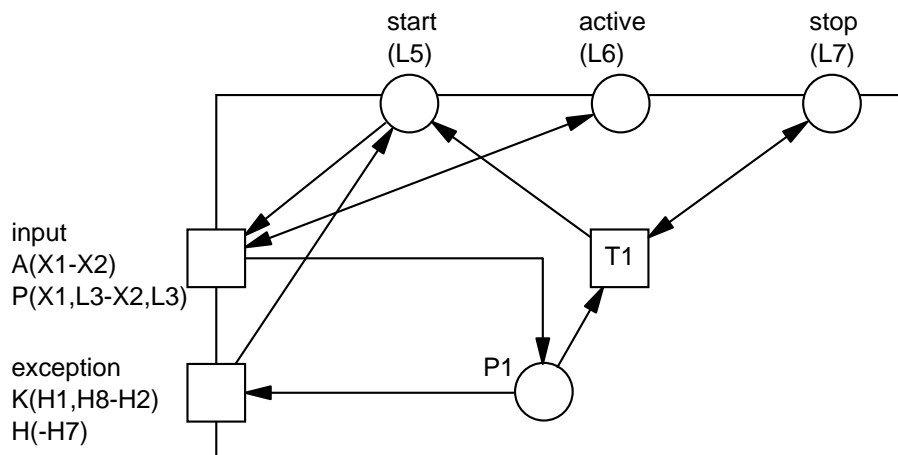


Abbildung 4.31: Petrinetz-Baustein für *Fault*

4.7.5 Call

Die Wiederverwendbarkeit von Verhaltensdefinitionen in der Sprache *WSCl* wird durch das Element *Process* ermöglicht. Dieses kann von den Aktivitäten *Call* oder *Spawn* gerufen werden. Die Definition des Prozesses muss sich dazu im selben Kontext oder in einem Vaterkontext der rufenden Aktivität befinden. Das Element *Call* startet einen Prozess und ist erst beendet, wenn der gerufene Prozess ebenfalls beendet ist.

Der entsprechende Baustein *Call* wird mit einem Baustein vom Typ *Rufempfang* verknüpft, der wiederum an einen Baustein vom Typ *Lebenszyklusprozess* angeschlossen ist. *Call* wartet bis zum Ende des gerufenen Prozesses, erst danach wird der Baustein beendet. Wir definieren für diese Verknüpfung eine zusätzliche Schnittstelle. Diese enthält *input-out*, *input-in*, *output-ok* und *output-error*. Über die ersten beiden Elemente wird der Prozess gestartet, *output-ok* empfängt den positiven Ausgang und *output-error* den negativen (abhängig davon, wie der Prozess ausgegangen ist, schaltet natürlich nur eine der letztgenannten Transitionen). Wurde über *output-error* ein negatives Ergebnis empfangen, leitet *Call* den Fehler an den entsprechenden *Ereigniskontext* weiter. Abbildung 4.32 stellt den Baustein *Call* dar.

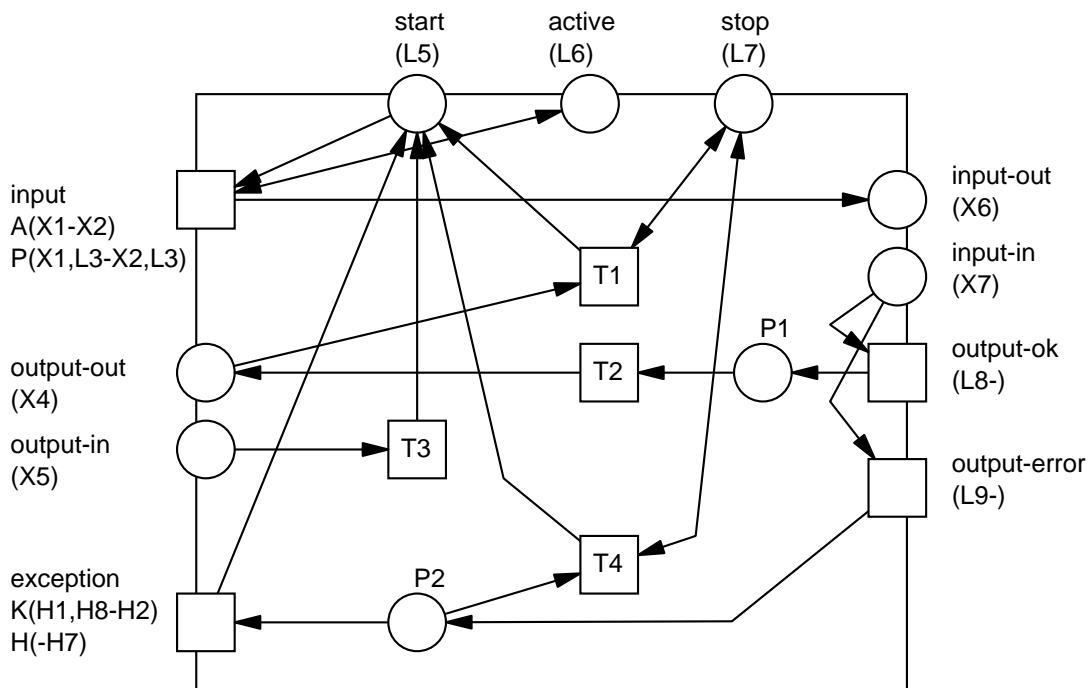


Abbildung 4.32: Petrinetz-Baustein für *Call*

4.7.6 Spawn

In *WSCI* stellt *Spawn* die Alternative zu *Call* dar. Der Unterschied ist, dass *Spawn* nicht bis zum Ende des gerufenen Prozesses wartet, sondern gleich nach dem Start des Prozesses endet.

Der Baustein *Spawn* wird an einen Baustein vom Typ *Rufempfang* angeschlossen, der wiederum an einen Baustein vom Typ *Lebenszyklusprozess* angeschlossen ist. Letzterer steuert den Ablauf in dem gerufenen Prozess. Prozessinstanzen, die so erzeugt werden, werden über die Transitionen *output-ok* und *output-error* abgefangen. Abbildung 4.33 zeigt den Baustein.

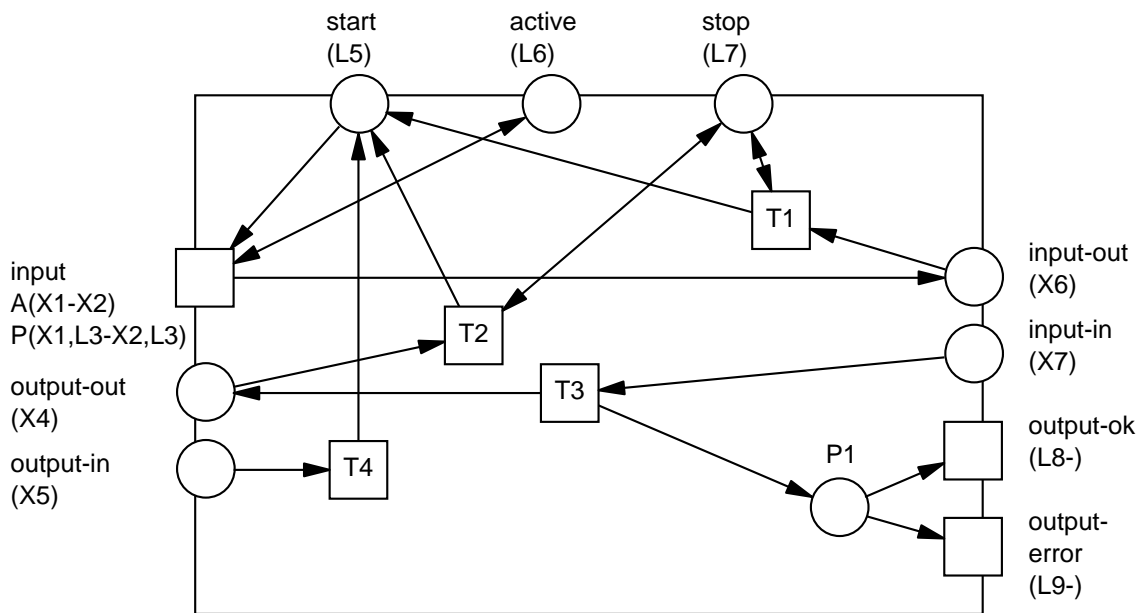


Abbildung 4.33: Petrinetz-Baustein für *Spawn*

Um Konflikte mit später gestarteten Instanzen zu vermeiden, sollte man komplexere Marken verwenden und eine eindeutige Identifizierungsmethode anwenden, sodass nur die richtigen Instanzen abgeräumt werden. Wir erläutern das Designproblem dieses Bausteines im Folgenden kurz.

Spawn muss beliebig oft hintereinander ausführbar sein. Wir können nicht darauf warten, dass der gerufene Prozess beendet wird. Wir müssen die Marke abräumen, die die Nachricht über den positiven oder negativen Ausgang des gerufenen Prozesses enthält. Ansonsten können spätere Prozessinstanzen mit alten, verbliebenen Marken des zu rufenden Prozesses in Konflikt kommen (sie empfangen das Ergebnis der zuvor gestarteten Prozessinstanz). Wir müssen also irgendeinem Baustein die Aufgabe übertragen, die Marke über den Ausgang des gerufenen Prozesses abzuräumen. Im gerufenen Prozess

selbst kann das nicht geschehen. Es könnten sonst auch Marken abgeräumt werden, die noch verarbeitet werden müssen (z.B. weil der Prozess durch *Call* gerufen wurde). Wir können auch keinen weiteren Empfangbaustein (wie z.B. *Rufempfang*) mit *Lebenszyklusprozess* verknüpfen, ebenfalls weil der Prozess auch durch *Call* gerufen werden kann. Einem dem Baustein *Call* eventuell folgenden Baustein *Join* kann diese Aufgabe auch nicht übertragen werden, da es keinen Zwang gibt, ihn tatsächlich zu benutzen, und da der Prozess auch vor der Ausführung von *Join* fehlerbedingt abgebrochen werden könnte. Die Umsetzung besteht also darin, dass *Call* selbst die Marken abräumt. Dadurch können sich auf *P1* beliebig viele Marken sammeln. Wir müssen also alle Marken eindeutig voneinander unterscheiden. Letzteres können wir in einem High-Level-Petrinetz erreichen. Wir verwenden aber Low-Level-Petrinetze, sodass wir die verwendeten Marken in unseren Modellen nicht unterscheiden können.

4.7.7 Join

Das *WSCI*-Element *Join* wartet, bis von einem angegebenen Prozess keine Instanzen mehr existieren. Das gilt für alle durch *Spawn* oder durch den Empfang einer Nachricht gestarteten Prozesse. Dieses *WSCI*-Element bietet somit die Möglichkeit, auf im Hintergrund laufende Subprozesse zu warten.

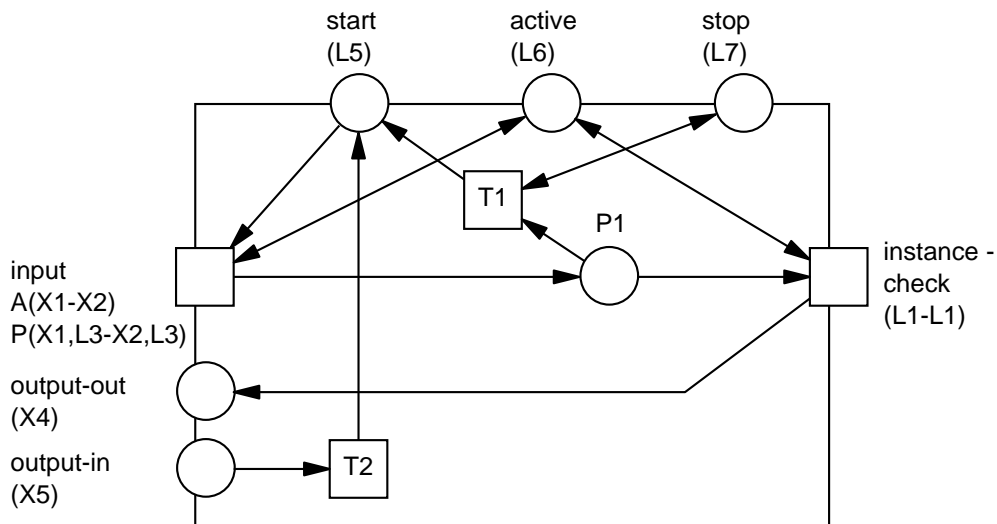


Abbildung 4.34: Petrinetz-Baustein für *Join*

Der Baustein *Join* (Abbildung 4.34) modelliert dieses Verhalten bis auf eine Ausnahme. Über die Transition *instance-check* wird am Baustein *Lebenszyklusprozess* der Platz *cs-depot* abgefragt. Dieser gibt über die gerade

nicht genutzten Instanzen Auskunft (jede Instanz wird dort durch eine Marke repräsentiert). Liegen alle verfügbaren Instanzen auf diesem Platz, gibt es keine aktive Instanz dieses Prozesses mehr und *Join* wird erfolgreich beendet. Das bedeutet, dass auch durch *Call* gerufene Instanzen die Abarbeitung von *Join* verzögern. Die Unterscheidung der Instanzen hinsichtlich der Aktivität, über die sie gerufen wurden, ist in High-Level-Petrinetzen möglich.

4.8 Beispiele für die Transformation von WSCI in Petrinetze

Wir setzen an dieser Stelle die Betrachtung unseres Beispiels einer einfachen Kommunikation zwischen zwei *Web Services* fort. Ein *Web Service* sendet über den Baustein *Solicit-Response* eine Nachricht an den anderen und wartet auf eine Antwort.

Die entsprechende Definition des statischen Teiles sehen wir auf Seite 10. Wir werden sie nun um die Definition des dynamischen Teiles in *WSCI* erweitern. Anschließend deuten wir an, wie die entsprechende Darstellung in Petrinetz-Bausteinen aussieht.

4.8.1 Dynamisches Interface in WSCI

Für ein einfaches Kommunikationsbeispiel definieren wir zwei *Web Services* (*interfaces*), sowie die Verknüpfung ihrer Operationen innerhalb des globalen Modells (*model*). Dabei gehen wir zuerst auf den antwortenden *Web Service* ein.

```
01<wsdl:definitions name = „Kommunikationsbeispiel“...>
02
03  <interface name = „AuskunftsService“>
04
05    <process name = „Auskunft“
06      instantiation = „message“>
07      <action name = „gibAuskunft“
08        role = „AuskunftsService“
09        operation = „AntwortPortType/AntworteAufFrage“>
10        <call process = „BearbeiteFrage“/>
11      </action>
12    </process>
13
14    <process name = „BearbeiteFrage“ instantiation = „other“>
15      ...
16    </process>
17  </interface>
18</wsdl:definitions>
```

Die wichtigsten Elemente sind hervorgehoben. In Zeile 05 beginnt die Definition des Prozesses. Wir erkennen durch „instantiation = message“, dass der Prozess durch den Empfang einer Nachricht gestartet wird. In den

Zeilen 07 bis 11 sehen wir die Definition der enthaltenen *Action* und ihre Referenz auf die benutzte Operation in Zeile 09. Um die Frage angemessen beantworten zu können, leitet der Prozess die Anfrage an den in Zeile 14 nicht näher spezifizierten Prozess weiter. Dieser kann nicht durch Erhalt einer Nachricht gestartet werden („instantiation = other“) und ist somit für außen liegende *Web Services* nicht erreichbar. Nach Abarbeitung dieses internen Prozesses schickt die *Action* eine Antwort. An wen diese Antwort gesendet wird, erkennen wir erst im globalen Modell. Hier werden die Operationen der beteiligten *interfaces* miteinander verknüpft.

```

01<definitions name = „Kommunikationsbeispiel“
02  ... Namensraumdefinitionen >
03
04  <import namespace = „http://example.com/auskunftsservice“
05    location = „http://example.com/auskunftsservice.wsci“/>
06  <import namespace = „http://example.com/kunde“
07    location = „http://example.com/kunde.wsci“/>
08
09  <model name = „FrageAntwort“>
10    <interface ref = „AuskunftsService“/>
11    <interface ref = „Kunde“/>
12
13    <!-- Kunde / AuskunftsService -->
14    <connect operations = „AntwortPortType/AntworteAufFrage
15      FragePortType/StelleFrage“/>
16    <!-- andere connect Elemente -->
17  </model>
18</definitions>

```

Im globalen Modell werden die beteiligten *interfaces* referenziert (Zeilen 10 und 11) und alle auftretenden Operationen verknüpft (Zeilen 14 und 15). Somit haben wir alle Partner festgelegt und können die Eigenschaften dieses Modells überprüfen. Im Folgenden zeigen wir, wie eine Übersetzung in die Petrinetz-Bausteine schematisch aussehen würde.

4.8.2 Darstellung in Petrinetz-Bausteinen

Wir zeigen in Abbildung 4.35 den *Web Service*, der die erste Anfrage an seinen Partner stellt. Wir sehen die Kombination aller enthaltenen Petrinetz-Bausteine. In dem *Web Service* definieren wir die Aktivität *Solicit-Response*. Sie sendet eine Nachricht an den Partner und wartet auf eine Antwort. Sollte

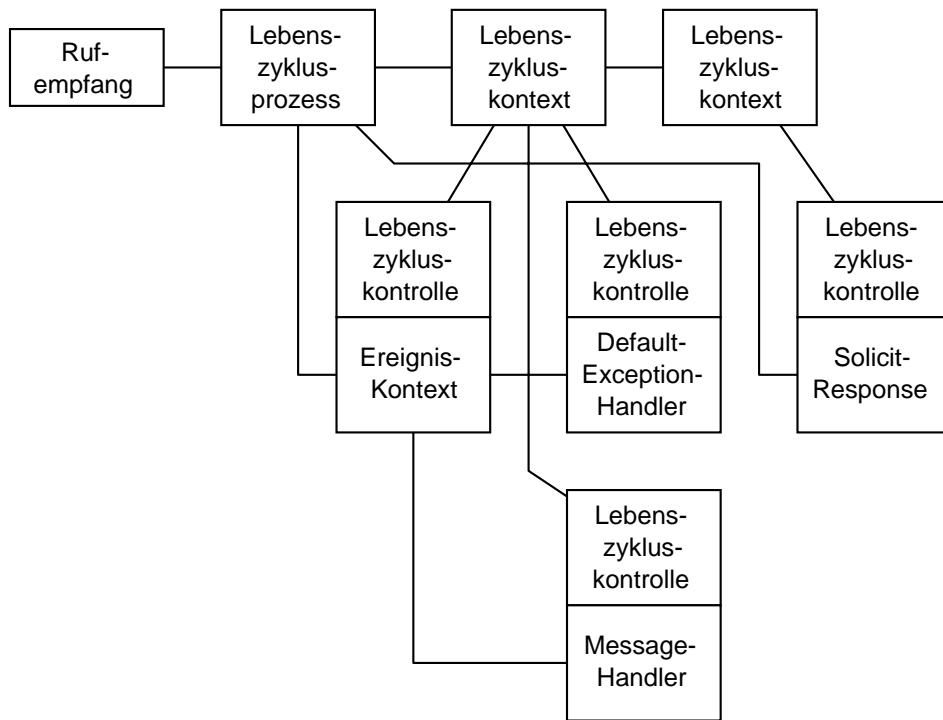


Abbildung 4.35: *Web Service* Nr. 1

dort ein Fehler auftreten, der zu einem fehlerhaften Ende des Partners führt, bekommt der `MessageHandler` eine entsprechende Nachricht und bricht die Aktivität ab.

Abbildung 4.36 zeigt die schematische Darstellung des antwortenden *Web Service*. Er benutzt alle für den Lebenszyklus notwendigen Bausteine, einen `DefaultExceptionHandler` und die *Action Request-Response*. Der Prozess, der von dieser *Action* gerufen wird, ist hier nur durch drei Punkte angedeutet.

Die Kombination dieser beiden *Web Services* wurde vollständig in Petri-netz-Bausteine umgesetzt und getestet. Ein Ergebnis des Tests ist, dass nur erwünschte Endzustände auch tatsächlich Endzustände sind. In einem solchen Zustand kann keine Transition schalten. Bei den Endzuständen handelt es sich ausschließlich um Zustände, in denen alle internen Marken abgeräumt wurden. Nicht abgeräumt werden die Marken auf dem Platz *cs-depot* jedes *Lebenszyklusprozess*. Sie repräsentieren die möglichen Prozessinstanzen und bleiben auch nach dem Ablauf einer Instanz erhalten. Jedes Verhalten führt zu fehlerfreier oder fehlerhafter Kommunikation. In jedem Fall verhält sich das Modell vollständig entsprechend den Erwartungen.

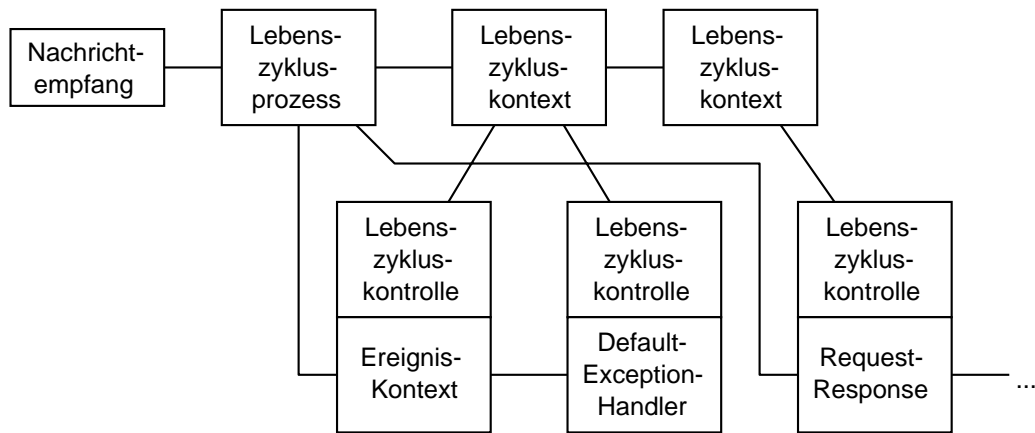


Abbildung 4.36: *Web Service* Nr. 2

4.8.3 Eigenschaften der Beispiele

Unser Ziel in dieser Arbeit war die Entwicklung von Bausteinen für *WSCI*. Das haben wir erreicht. Wir können nun ausnutzen, dass wir zur Modellierung Petrinetze verwendet haben. Die Eigenschaften von Petrinetzen sind computergestützt auswertbar. Für diese Auswertung verwenden wir den Petrinetz-Model-Checker LoLA [Sch00]. Dazu benötigen wir ein Werkzeug, welches einen gegebenen *WSCI*-Prozess in ein unserer Semantik entsprechendes Petrinetz übersetzt und dieses in einem für LoLA lesbaren Format ausgibt. Ein aus unseren Bausteinen zusammengesetzter *Web Service* besitzt selbst bei kleinen Beispielen bereits mehrere hundert Plätze und Transitionen. Es wäre sehr mühselig, jedes Beispiel von Hand einzugeben. Wir haben zu diesem Zweck mehrere Java-Programme [JAVASUN] geschrieben. Jedes Programm erstellt ein Petrinetz und beschreibt den Ablauf einzelner Prozesse oder das Zusammenspiel mehrerer *Web Services*. Dazu entwickelten wir für jeden Baustein eine Klasse und statteten sie mit den notwendigen Methoden aus, um den dargestellten Baustein mit anderen Bausteinen verknüpfen zu können. Bei der Wahl der Beispiele haben wir darauf geachtet, jeden modellierten Baustein wenigstens einmal zu verwenden und möglichst an jeder Schnittstelle mit verschiedenen Bausteinen zu verknüpfen. Den Quellcode ist in [Wei04] verfügbar.

Auf diese Art haben wir ungefähr zehn Beispielprozesse entworfen und mit dem Werkzeug LoLA getestet. Jedes Beispiel umfasst ca. zweihundert Plätze und zweihundert Transitionen. Je nach Komplexität der Beispiele enthält der Erreichbarkeitsgraph des Petrinetzes zwischen zweihundert und mehreren hunderttausend Zuständen. Mehr als zehntausend Zustände haben wir nur mit Beispielen erreicht, die nebenläufig auszuführende Aktivitäten ver-

wenden. Dabei wurde keine Reduktionstechnik verwendet. Wichtigen Transitionen haben wir zusätzliche *Nachplätze* angehängt, um den Kontrollfluss im Wesentlichen nachvollziehen zu können. Die Menge der toten Markierungen entsprach bei jedem Test den Erwartungen. Zum Beispiel entsprechen die Erwartungen an einen einfachen, eine Nachricht sendenden *Web Service*, dass er erfolgreich beendet und die Nachricht geschickt wurde oder dass der *Web Service* einen Fehler meldet und keine Nachricht schickt. Diese ersten Tests der Bausteine verliefen positiv und zeigen zumindest die technische Qualität der Petrinetz-Semantik.

4.8.4 Vereinfachungsmöglichkeiten

Die folgenden Ausführungen zu Vereinfachungsmöglichkeiten schneiden ein sehr großes Gebiet der Modellierung an und dienen nur als Hinweis für die weitere Bearbeitung der Modelle.

Jeder der vorgestellten Bausteine enthält ca. zehn Plätze und zehn Transitionen. Da wir für die Darstellungen eines *Web Service* abhängig von dessen Komplexität sehr viele Bausteine benötigen, ist die Anzahl der Plätze, Transitionen und Flusskanten in einem solchen Modell ebenfalls sehr hoch. Schon bei kleinen *Web Services* können wir von fünfzehn Bausteinen, also von einhundertfünfzig Plätzen und einhundertfünfzig Transitionen ausgehen. Die Analyse durch den Petrinetz-Model-Checker LoLA ergab für einen einfachen *Web Service* zwischen zweihundert und fünftausend erreichbare Zustände. Das ist auf die Komplexität der Prozessinstanzen zurückzuführen. Zu dem Ablauf einer Prozessinstanz gehören auch die Initialisierung und das Beenden derselben. Zusätzliche Komplexität wird durch die Behandlung von auftretenden Fehlern erreicht. In den mit den entwickelten Bausteinen modellierten *Web Services*, die eine parallele Verarbeitung ermöglichen (durch Verwendung eines Bausteines vom Typ *All*), kann die Anzahl der Zustände somit leicht auf mehrere Millionen anwachsen. Spätestens an diesem Punkt geht die Übersichtlichkeit der Modelle verloren. Wie bereits beschrieben, wurde keine Möglichkeit der Vereinfachung genutzt. Nähere Ausführungen zu diesem Thema können in [Sch99] nachgelesen werden.

Wir gehen nun darauf ein, wie man die Übersichtlichkeit verbessern kann. Die Analyse der Eigenschaften von *WSCI*-Modellen kann nur mit den entwickelten Bausteinen betrieben werden. Die folgenden Vereinfachungsmöglichkeiten zielen deshalb lediglich auf eine für den Menschen lesbare Variante, die für keine weiteren Aufgaben mehr herangezogen wird.

Reduzierter Erreichbarkeitsgraph

Eine Variante zur Vereinfachung der Darstellung besteht in der Verwendung reduzierter Erreichbarkeitsgraphen. Diese Methode erfordert ein fertiges Petrinetz-Modell des *Web Service*. Zu diesem konstruieren wir den vollständigen Erreichbarkeitsgraphen, den wir durch das Weglassen unnötiger Knoten vereinfachen. Ein Knoten ist unnötig, wenn sein Vaterknoten im Erreichbarkeitsgraphen den gleichen sichtbaren Zustand darstellt und nicht verzweigend ist. Unter dem sichtbaren Teil eines Zustandes verstehen wir diejenigen Elemente, die nach außen hin sichtbar sind (z.B. ob eine Nachricht gesendet wurde). Die für das Verständnis des Ablaufs wichtigen Zustandsübergänge bleiben also im Erreichbarkeitsgraphen enthalten. Die Überprüfung der Eigenschaften brachte zumindest für unsere ersten Testfälle das Ergebnis, dass nur zwanzig Prozent der erreichbaren Zustände in einem Schritt mehr als einen Folgezustand erreichen konnten. Es sind also nur relativ wenige Zustandsknoten im Erreichbarkeitsgraphen verzweigend. Der mit dieser Methode reduzierte Erreichbarkeitsgraph ist also entsprechend kleiner und übersichtlicher.

Einzelfälle

Für die vereinfachende Darstellung eines *Web Service* bietet es sich an, nicht alle möglichen Abläufe zu zeigen, sondern lediglich ein paar ausgewählte. Dafür sind zahlreiche Modellierungsmethoden geeignet, ein Beispiel sind Sequenzdiagramme. Diese spiegeln den sequentiellen Ablauf von genau einem möglichen Verhalten wider. Anhand solcher Diagramme können wir schnell den Sinn des dargestellten *Web Service* erkennen, ohne alle Einzelheiten überblicken zu müssen.

Kapitel 5

Abschließende Betrachtungen

5.1 Zusammenfassung

In dieser Arbeit haben wir uns der theoretischen Durchdringung der praktisch relevanten Technologie der *Web Services* zugewandt. Wir stellten mit *WSCI* und *BPEL* zwei Modellierungssprachen für *Web Services* vor, prüften deren Vor- und Nachteile und verglichen ihre unterschiedlichen Sichtweisen auf *Web Services*. In diesen Bereichen gaben wir einen Überblick über den derzeitigen Stand der Theorie der *Web Services*.

Wir entwickelten eine Semantik für die *Web-Service*-Beschreibungssprache *WSCI*. Mit dieser Semantik haben wir die Sprache formalisiert. Die dabei verwendeten Petrinetze sind Low-Level-Petrinetze. Aus ihnen zusammengesetzte *Web Services* spiegeln nahezu alle Details des in *WSCI* spezifizierten Verhaltens wider. Dazu gehören die Unterteilung des *Web Service* in Prozesse sowie die Ausführung der entsprechenden Prozessinstanzen inklusive der Fehlerbehandlung und der Reaktion auf externe Ereignisse. Einige Details wurden von uns aufgrund ihrer ungenauen Spezifikation in *WSCI* gesondert betrachtet. Beispiele dafür sind die Menge der abzuarbeitenden Elemente für das *WSCI*-Element *Foreach* sowie die Bedingungen für die *WSCI*-Elemente *Switch*, *Until* und *While*. In den jeweiligen Modellen wurde das Fehlen einer genauen Spezifikation entsprechend durch Nichtdeterminismus umgesetzt.

Mit den von uns entwickelten Modellen haben wir für jeden Prozess die Initialisierung, die Ablaufsteuerung und die Beendigung möglicher Prozessinstanzen umgesetzt. Jede Prozessinstanz wird nach ihrer Abarbeitung in den Bausteinen wieder vollständig entfernt, so dass nach jedem Ablauf keine unerwünschten Marken mehr vorhanden sind. Unsere Modelle setzen sowohl das Standardverhalten als auch das Verhalten im Fehlerfall und die Reaktion auf externe Ereignisse durch die zuständigen EreignisHandler um. Als beson-

dere Herausforderung stellte sich die Eingliederung des Reaktionsverhaltens auf externe Ereignisse in den internen Kontrollfluss dar, da das Auftreten eines solchen Ereignisses oft von keiner inneren Aktion ausgelöst wird und praktisch jederzeit geschehen kann.

Die Übersetzung der Konstrukte der Sprache *WSCI* in Petrinetz-Bausteine geschieht weitestgehend eigenschaftserhaltend, da das Verhalten der Elemente in *WSCI* in den Petrinetz-Bausteinen nachgebildet wurde. Aufgrund bereits genannter Mängel in der Spezifikation von *WSCI* haben wir an einigen Stellen bewusst Nichtdeterminismus erzeugt. Dieser führt zu einer Erweiterung des Zustandsraumes. Alle in *WSCI* erreichbaren Zustände können auch in unserer Petrinetz-Semantik nachgebildet werden. Allerdings wurde in der Übersetzung nicht einfach jedem Element aus *WSCI* ein Petrinetz-Baustein zugeordnet. Es gibt einige Elemente aus *WSCI*, für die wir keinen Petrinetz-Baustein modelliert haben (z.B. *interface* oder *case*), und es gibt viele Bausteine, die wir aus impliziten Voraussetzungen der Sprache *WSCI* entwickelt haben und welche jeweils kein Element aus *WSCI* darstellen (z.B. *Lebenszykluskontrolle*).

Bei der Entwicklung unserer Semantik sind uns in der vorgegebenen Spezifikation von *WSCI* Fehler aufgefallen, die jedoch keine größere Tragweite besitzen. Weiterhin erkannten wir Schwachstellen, die daraus resultieren, dass das Verhalten für bestimmte Fälle sehr ungenau beschrieben wurde. Zum Beispiel ist es erlaubt, dass in einem Kontext mehrere EreignisHandler auf dasselbe eingehende Ereignis reagieren können. Das *interface* in *WSCI* wird dann entsprechend als mehrdeutig erkannt. Wir sind aber der Meinung, dass so etwas nicht erlaubt sein sollte. Man kann diese Mehrdeutigkeit auch sehr gut mit dafür vorgesehenen Elementen aus *WSCI* in einem einzigen EreignisHandler zusammenfassen. Die vorgegebene Spezifikation sagt weiterhin, dass das erste auftretende Ereignis Vorrang vor später auftretenden Ereignissen hat, die Möglichkeit des gleichzeitigen Auftretens zweier Ereignisse wird gar nicht betrachtet. Die angeführten Kritikpunkte schmälern die Spezifikation und sind verbesserungswürdig.

Abschließend haben wir mit dem Werkzeug LoLA und mit selbst geschriebenen Java-Programmen zahlreiche Tests für die von uns entwickelte Semantik durchgeführt. In diesen Tests haben wir auch für komplexe Beispiele durchgehend positive Ergebnisse erzielt. Dies unterstreicht die Qualität der von uns entwickelten Semantik.

5.2 Ausblick

Diese Arbeit hat viele neue Ansätze für nachfolgende wissenschaftliche Arbeiten hervorgebracht. Dazu gehört unter anderem ein vollständiger Vergleich von *BPEL* und *WSCI*. Entsprechend weiterführende Überlegungen können auf Möglichkeiten der Kombination der beiden vorgestellten Beschreibungssprachen abzielen. Das schließt auch die Entwicklung einer neuen Sprache ein, welche die Vorteile von *WSCI* und *BPEL* kombiniert.

Weiterhin haben wir die korrekte Funktionsweise der entwickelten Bausteine durch zahlreiche Tests untermauert. Eine Möglichkeit der Fortführung dieser Arbeit besteht in einer fortgesetzten Validierung der vorgestellten Semantik.

In unserer Arbeit sind wir außerdem auf Möglichkeiten der Vereinfachung der Darstellung eingegangen. Diese beziehen sich jedoch nicht auf die Modelle, sondern versuchen das Wesentliche der dargestellten *Web Services* leichter verständlich zu machen. Eine Vereinfachung der Petrinetz-Bausteine selbst ist aber aufgrund der Komplexität der Sprache *WSCI* nicht erreichbar, ohne dabei an Ausdrucksstärke zu verlieren. Eine Fortführung unserer Arbeit kann also darin bestehen, die Darstellung der Modelle noch weiter zu vereinfachen.

Literaturverzeichnis

- [AAF+02] Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi–Nagy, Ivana Trickovic, Sinisa Zimek. *Web Services Choreography Interface (WSCI) 1.0*. W3C Note vom 8.8.2002 (siehe <http://www.w3.org/TR/2002/NOTE-wsci-20020808>).
- [ABPR03] Daniel Austin, Abbie Barbir, Ed Peters, Steve Ross-Talbot; *Web Services Choreography Requirements 1.0*; W3C Working Draft vom 12.8.2003 (siehe <http://www.w3.org/TR/2003/WD-ws-chor-reqs-20030812>)
- [ACD+03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic and Sanjiva Weerawarana. *Business Process Execution Language for Web Services Version 1.1*. Spezifikation vom 5.5.2003 (siehe <http://www.ibm.com/developerworks/library/ws-bpel/>).
- [BHM+04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris and David Orchard. *Web Service Architecture*. W3C Working Group Note vom 11.2.2004 (siehe <http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/>).
- [BM01] Paul V. Biron and Ashok Malhotra. *XML Schema Part 2: Datatypes*. W3C Recommendation vom 2.5.2001 (siehe <http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/>).
- [CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith and Sanjiva Weerawarana. *Web Service Description Language (WSDL) 1.1*. W3C Note vom 15.3.2001 (siehe <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>).
- [Cha] Mike Chapple. *Databases - The ACID Model* (siehe <http://databases.about.com/library/weekly/aa120102a.htm>).

- [FB96] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. November 1996 (siehe <http://www.ietf.org/rfc/rfc2045.txt>).
- [FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter P. Leach and T. Berners-Lee. *Hypertext Transfer Protocol -- HTTP/1.1*. Juni 1999 (siehe <http://www.ietf.org/rfc/rfc2616.txt>).
- [Fos04] H. Foster. *BPEL4WS plugin* (siehe <http://www.doc.ic.ac.uk/ltsa/bpel4ws/>)
- [GHM+03] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau and Henrik Frystyk Nielsen. *SOAP Version 1.2 Part 1: Messaging Framework*. W3C Recommendation vom 24.6.2003 (siehe <http://www.w3.org/TR/2003/REC-soap12-part1-20030624/>).
and *SOAP Version 1.2 Part 2: Adjuncts*. W3C Recommendation vom 24.6.2003 (siehe <http://www.w3.org/TR/2003/REC-soap12-part2-20030624/>).
- [HHK+03] Hugo Haas, Oisín Hurley, Anish Karmarkar, Jeff Mischkin, Mark Jones, Lynne Thompson and Richard Martin. *SOAP Version 1.2 Specification Assertions and Test Collection*. W3C Recommendation vom 24.6.2003 (siehe <http://www.w3.org/TR/2003/REC-soap12-testcollection-20030624/>)
- [JAVASUN] die Sprache *Java* und darauf aufbauende Technologien (siehe <http://java.sun.com/>)
- [KBR04] Nickolaos Kavantzias, David Burdett and Greg Ritzinger; *Web Service Choreography Description Language Version 1.0*. W3C Working Draft vom 27.4.2004 (siehe <http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/>).
- [Ley01] Frank Leymann. *Web Services Flow Language (WSFL 1.0)*. (siehe <http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>).
- [LTSA] *Labelled Transition System Analyser* (siehe <http://www.doc.ic.ac.uk/ltsa/>)
- [Mar03] Axel Martens. *Verteilte Geschäftsprozesse - Modellierung und Verifikation von Web Services*. Dissertation (PhD thesis), Humboldt Universität zu Berlin 2003.

- [Mi03] Nilo Miltra. *SOAP Version 1.2 Part 0: Primer*. W3C Recommendation vom 24.6.2003 (siehe <http://www.w3.org/TR/2003/REC-soap12-part0-20030624/>).
- [Moh02] C. Mohan. Dynamic E-business - Trends in Web Services. In: BUCHMANN ET AL. (Hrsg.): *Proceedings of TES'02*, Springer-Verlag, 2002 (LN-CS 2444), S.1-6
- [Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.
- [Rei85] W. Reisig. *Petri Nets*. EATCS, Monographs on Theoretical Computer Science, W.Brauer, G. Rozenberg, A. Salomaa (Eds.), Springer Verlag, Berlin, 1985.
- [Rei90] W. Reisig. *Petrinetze, eine Einführung*. Springer-Verlag (1990)
- [Sch00] K. Schmidt. *LoLA - A Low Level Analyser*. *International Conference on Application and Theory of Petri Nets*. Springer-Verlag, 2000 (LN-CS 1825), S. 465 ff. (LoLA: siehe www.informatik.hu-berlin.de/~kschmidt/lola.html)
- [Sch99] Karsten Schmidt. Stubborn set for standard properties. *Proc. 20th Int. Conf. Application and Theory of Petri nets, LNCS 1639*, pages 46-65, 1999.
- [Sta90] Peter H. Starke. *Analyse von Petri-Netz-Modellen*. Humboldt-Universität zu Berlin, B. G. Teubner Stuttgart 1990
- [St04] Christian Stahl. *Transformation von BPEL₄WS in Petrinetze*. Diplomarbeit, Humboldt Universität zu Berlin 2004.
- [Tha01] Satish Thatte. *XLANG - Web Services for Business Process Design*. (siehe <http://www.gotdotnet.com/team/xml-wsspecs/xlang-c/default.htm>).
- [TBMM01] Henry S. Thompson, David Beech, Murray Maloney and Noah Mendelsohn. *XML Schema Part 1: Structures*. W3C Recommendation vom 2.5.2001 (siehe <http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/>).
- [WebInt] *Intalio* - führender Anbieter für Business Process Management Systems (BPMS) (siehe www.intalio.com)
- [WebBEA] *BEA Systems, Inc.* (siehe www.bea.com)

- [WebSAP] *SAP (Systeme, Anwendungen, Produkte in der Datenverarbeitung)* (siehe www.sap.com)
- [WebSun] *Sun Microsystems, Inc.* (siehe www.sun.com)
- [WebW3C] *World Wide Web Consortium* (siehe www.w3.org)
- [Wei04] *Java-Code für die entwickelten Bausteine* (siehe www.informatik.hu-berlin.de/~weissled)
- [WSCWG] *Web Service Choreography Working Group.* (siehe <http://www.w3.org/2002/ws/chor/>)

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Weiterhin erkläre ich hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Instituts für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Berlin, den 22. November 2004

Stephan Weißleder