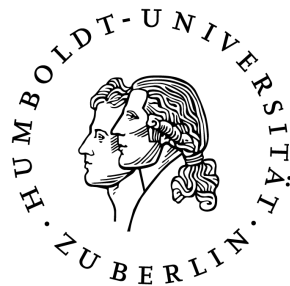


# Implementierung einer Petri-Netz-Semantik für BPEL

Diplomarbeit

Sebastian Hinz

18. März 2005



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

Gutachter:  
Prof. Dr. Wolfgang Reisig  
Dr. habil. Karsten Schmidt



## Zusammenfassung

BPEL ist eine Modellierungssprache zur Beschreibung von verteilten Geschäftsprozessen mit Webservices. Um mit formalen Methoden die Sprache selbst und in BPEL modellierte Prozesse verifizieren zu können, wird eine formale Semantik benötigt. Auf der Basis von Petrinetzen wurde bereits eine solche Semantik entwickelt. Um die Analyse eines Prozesses zu ermöglichen wird ein Werkzeug benötigt, das die Transformation des Prozesses in ein Petrinetz übernimmt.

In der vorliegenden Arbeit wird ein solches Werkzeug entwickelt, vorgestellt und seine Funktionsweise erläutert.



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>7</b>
1.1	Problemstellung . . . . .	7
1.2	Lösungsidee . . . . .	7
<b>2</b>	<b>Grundlagen</b>	<b>9</b>
2.1	BPEL . . . . .	9
2.2	Petrinetze . . . . .	11
2.3	Petrinetz-Semantik für BPEL . . . . .	13
2.4	Werkzeuge . . . . .	14
<b>3</b>	<b>Beschreibung von BPEL2PN</b>	<b>17</b>
3.1	Eingabeformat . . . . .	17
3.2	Zielformate . . . . .	17
3.2.1	LoLA . . . . .	17
3.2.2	Info-Datei . . . . .	18
3.2.3	BPEL-Datei mit ids . . . . .	18
3.2.4	dot . . . . .	19
3.2.5	Benennung der Knoten . . . . .	19
3.3	Aufruf . . . . .	20
<b>4</b>	<b>Implementierung</b>	<b>23</b>
4.1	Prinzip . . . . .	23
4.2	Umsetzung . . . . .	27
4.2.1	BPELDoc . . . . .	27
4.2.2	PetriNet . . . . .	28
4.2.3	Transformer . . . . .	30
4.2.4	BPEL2PN . . . . .	30
4.3	Transformation der Datenobjekte . . . . .	31
4.4	Transformation der Aktivitäten . . . . .	32
4.4.1	Grundsätzlicher Ablauf . . . . .	32
4.4.2	invoke . . . . .	33
4.4.3	assign . . . . .	34
4.4.4	compensate . . . . .	35
4.5	Transformation der Link-Semantik . . . . .	37
4.6	Transformation von scope und process . . . . .	40
4.6.1	Besondere scope-Muster . . . . .	44
4.6.2	Transformation des FaultHandlers . . . . .	45
4.6.3	Transformation des CompensationHandlers . . . . .	45
<b>5</b>	<b>Zusammenfassung und Ausblick</b>	<b>47</b>
5.1	Einschränkungen . . . . .	47
5.2	Weitere Arbeit . . . . .	48



## Abbildungsverzeichnis

1	Ablaufschema von BPEL2PN . . . . .	8
2	Weiterreichen eines Schnittstellenplatzes an ein Kindmuster . . .	25
3	Weiterreichen eines Schnittstellenplatzes an mehrere Kindmuster	26
4	Klassendiagramm von BPEL2PN . . . . .	27
5	Klassenbaum der BPEL-Aktivitäten . . . . .	28
6	Simulation einer Lesekante . . . . .	29
7	Resetkante von PR nach T2 . . . . .	30
8	Einbettung der <code>invoke</code> -Aktivität in einen <code>scope</code> . . . . .	34
9	Überführung eines <code>assigns</code> in eine <code>sequence</code> . . . . .	36
10	Einfach verkettete Liste der Link-Plätze in verschachtelten <code>flows</code>	38
11	Stack-Alternative im <code>CompensationHandler</code> für K-1 Kindscopes	46



# 1 Einleitung

## 1.1 Problemstellung

Die *Business Process Execution Language for Web Services (BPEL)* ist eine Modellierungssprache für verteilte Geschäftsprozesse auf Basis von Webservices. Entwickelt wurde sie in den letzten Jahren unter der Schirmherrschaft von IBM und Microsoft und befindet sich mittlerweile im Standardisierungsverfahren. Leider wird BPEL in der offiziellen Spezifikation [ACLL03] nur informal, das heißt umgangssprachlich, beschrieben. Lediglich die Syntax ist durch ein XML-Schema definiert. Eine mathematisch fundierte Semantik existiert nicht. Um aber mit formalen Methoden in BPEL modellierte Prozesse zu analysieren und um eventuell in BPEL vorhandene Inkonsistenzen finden und nachweisen zu können, ist eine formale Semantik unerlässlich.

Eine Möglichkeit um verteilte Prozesse formal zu beschreiben stellen Petrinetze dar, die sich auch schon in der Modellierung von Geschäftsprozessen bewährt haben [vdA98]. Des Weiteren existieren für Petrinetze eine Reihe von Analyse-Werkzeugen zur Verifikation z.B. durch Model-Checking.

So ist es also naheliegend, auf Basis von Petrinetzen eine formale Semantik für BPEL zu entwickeln. 2004 wurde von Christian Stahl in [Sta04] eine solche Semantik vorgestellt. Andere Ansätze basieren unter anderem auf Prozessalgebra [Fer04], Automaten [FBS04] oder auf Abstract-State-Machines (ASM) [FR04], [FGV04].

Theoretisch ist nun die Möglichkeit gegeben, einen in BPEL modellierten Geschäftsprozess in ein Petrinetz zu transformieren und anschließend mit einem geeigneten Werkzeug wie z.B. dem Model-Checker LoLA [Sch00] zu analysieren. Durch die hohe Komplexität der Petrinetz-Semantik ist jedoch eine Transformation von Hand sehr aufwendig und fehleranfällig.

Aus diesem Grund ist ein Werkzeug erforderlich, das die Transformation eines gegebenen, in BPEL spezifizierten Geschäftsprozesses (*BPEL-Prozess*) in ein Petrinetz automatisiert. Die Entwicklung eines solchen Werkzeugs, das sich auf die in [Sta04] vorgestellte Petrinetz-Semantik stützt, ist der Gegenstand der vorliegenden Arbeit.

## 1.2 Lösungsidee

Um die Transformation eines BPEL-Prozesses in ein Petrinetz zu automatisieren, wird in der objektorientierten, plattformunabhängigen Programmiersprache Java ein entsprechendes Werkzeug (*BPEL2PN*) entwickelt.

In Abbildung 1 wird der Aufbau des Werkzeuges illustriert. Es müssen drei Hauptaufgaben erfüllt werden:

**Einlesen** Zunächst muss ein BPEL-Prozess eingelesen und in einer geeigneten Datenstruktur im Speicher gehalten werden. Da BPEL eine XML-basierte Sprache ist, kann hier auf vorhandene Hilfsmittel (JDOM, siehe 2.4) zurückgegriffen werden.

## 1 Einleitung

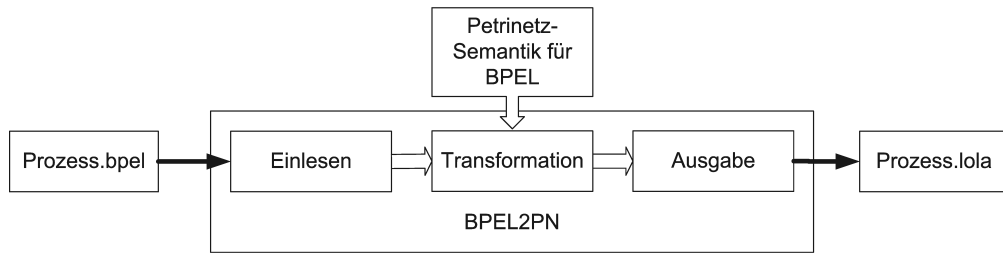


Abbildung 1: Ablaufschema von BPEL2PN

**Transformation** Der komplexeste Teil ist es, die eigentliche Transformation zu automatisieren. Die in [Sta04] gegebene Petrinetz-Semantik für BPEL hat einen modularen Aufbau, das heißt, jedes BPEL-Konstrukt wird durch ein eigenes Petrinetz modelliert. Diese einzelnen Muster werden dann miteinander entsprechend der Prozess-Struktur zu einem einzigen Petrinetz verschmolzen. Im Rahmen der vorliegenden Arbeit wird ein Konzept entwickelt, diese Verschmelzung programmtechnisch umzusetzen.

**Ausgabe** Abschließend muss das erzeugte Petrinetz in geeigneter Form ausgegeben werden. Um eine Analyse mit dem Model-Checker LoLA zu ermöglichen, soll hier eine Datei im LoLA-Format erzeugt werden.

In der vorliegenden Arbeit wird der Datenaspekt der Petrinetz-Semantik vernachlässigt. Es sollen lediglich Low-Level-Netze erzeugt werden. Eine spätere Integration des Datenaspekts und eine damit verbundene Erweiterung auf High-Level-Netze muss bei der Entwicklung von BPEL2PN jedoch eingeplant werden. Sie soll Gegenstand einer weiteren Arbeit sein.

## 2 Grundlagen

### 2.1 BPEL

Die Business Process Execution Language for Web Services (BPEL) ist eine XML-basierte Modellierungssprache zur Beschreibung von verteilten Geschäftsprozessen auf der Basis von Webservices. Die genaue Spezifikation kann in [ACLL03] nachgelesen werden. Hier sollen nur einige grundlegende Konzepte der Sprache kurz umrissen werden, die zum Verständnis der folgenden Kapitel notwendig sind.

Eine *BPEL-Datei* beschreibt das Verhalten eines Webservice, dessen Schnittstelle in einer *WSDL-Datei* (Web Services Description Language [CCMW01]) beschrieben wurde. Um mit anderen Webservices (*Partnern*) kommunizieren zu können, werden Nachrichten ausgetauscht, die gesendet und empfangen werden können. Während BPEL nur die Beschreibung eines Prozesses bereitstellt, wird ein konkreter, nach einer solchen Beschreibung ausgeführter Prozess als *Prozessinstanz* bezeichnet.

Eine BPEL-Datei ist ein XML-Dokument (Extensible Markup Language [BPCM<sup>+</sup>04]), dessen Struktur durch ein XML-Schema beschrieben ist. Jedes BPEL-Sprachkonstrukt entspricht dabei einem XML-Element, das durch Attribute genauer spezifiziert werden kann.

Im Folgenden wird ein kurzer Überblick über alle BPEL-Sprachkonstrukte gegeben. Ausführliche Erklärungen und die verfügbaren Attribute jedes Konstrukts können in [ACLL03] nachgeschlagen werden.

**Basisaktivitäten** sind Aktivitäten, die nicht rekursiv verschachtelt werden können. Im Einzelnen sind das:

- **empty** (Tut nichts.)
- **wait** (Stoppt den Kontrollfluss für eine spezifizierte Zeit.)
- **assign** (Weist einer Variablen einen Wert zu.)
- **receive** (Empfängt eine Nachricht eines Partners.)
- **reply** (Sendet eine Nachricht an einen Partner.)
- **asynchrones invoke** (Ruft einen Partner auf.)
- **synchrones invoke** (Ruft einen Partner auf und wartet auf dessen Antwort.)
- **throw** (Signalisiert einen internen Fehler im Prozess.)
- **terminate** (Beendet die *Prozessinstanz*.)
- **compensate** (Ruft den `CompensationHandler` aller eingebetteten, bzw. eines spezifizierten `scope` auf. `compensate` kann nur im `FaultHandler` oder `CompensationHandler` eingebettet sein.)

**Strukturierte Aktivitäten** können rekursiv verschachtelt werden und definieren eine kausale Ordnung der Basisaktivitäten. Im Einzelnen sind das:

- **sequence** (Beinhaltet eine Anzahl von inneren Aktivitäten, die sequentiell ausgeführt werden.)
- **flow** (Beinhaltet eine Anzahl von inneren Aktivitäten, die nebenläufig ausgeführt werden.)
- **switch** (Beinhaltet eine Anzahl von inneren Aktivitäten von denen aufgrund einer Datenauswertung genau eine ausgeführt wird.)
- **pick** (Ähnlich dem **switch**, allerdings wird die auszuführende Aktivität durch externe Nachrichten oder Alarmereignisse gewählt.)
- **while** (While-Schleife: Eine Aktivität wird so oft ausgeführt, bis die Schleifenbedingung nicht mehr gilt.)
- **scope** (Bettet eine Aktivität ein und bindet sie an einen **FaultHandler**, einen **CompensationHandler** und optional einen **EventHandler** an. Ein **scope** ist ein Sichtbarkeitsbereich für in ihm deklarierte Variablen und **CorrelationSets**. Ein **scope** in dem ein oder mehrere weitere **scopes** eingebettet sind, wird als *Vaterscope* dieser **scopes** bezeichnet. Dementsprechend heißen die eingebetteten **scopes** *Kindsscopes* ihres *Vaterscope*. Dabei spielt die Verschachtelungstiefe keine Rolle.)

**FaultHandler** Der **Fault Handler** ist eine Komponente des **scope**. Im laufenden Prozess auftretende Fehler werden an ihn signalisiert, damit er sie behandeln kann. Der standardmäßige **FaultHandler** ruft die **CompensationHandler** aller *Kindsscopes* in der umgekehrten Reihenfolge ihrer Abarbeitung auf und reicht den Fehler anschließend an den **FaultHandler** des *Vaterscope* weiter. Alternativ dazu kann ein nutzerdefinierter **FaultHandler** spezifiziert werden, der je nach Art des auftretenden Fehlers eine bestimmte BPEL-Aktivität ausführt.

**CompensationHandler** Der **CompensationHandler** ist eine Komponente des **scope**. Tritt ein Fehler auf, so ist der **CompensationHandler** dafür zuständig, bestimmte, durch die Aktivitäten des **scope** hervorgerufene Effekte, wieder rückgängig zu machen (zu *kompensieren*). Standardmäßig werden dazu die **CompensationHandler** aller *Kindsscopes* in umgekehrter Reihenfolge ihrer Abarbeitung aufgerufen. Alternativ kann ein nutzerdefinierter **CompensationHandler** definiert werden, der eine BPEL-Aktivität einsetzt.

**EventHandler** Im **EventHandler** können Aktivitäten definiert werden, die ausgeführt werden, wenn spezifizierte Ereignisse eintreten. Es werden zwei Arten von Ereignissen unterschieden: Alarmereignisse (**onAlarm-Events**), die ausgelöst werden, wenn eine vorgegebene Zeit seit Beginn der Ausführung des **scope** verstrichen ist, und Nachrichtenereignisse (**onMessage-Events**), die ausgelöst werden, wenn eine bestimmte Nachricht eintrifft.

**Link-Konzept** In einer `flow`-Aktivität können Links definiert werden. Sie dienen der Synchronisation nebenläufiger Aktivitäten. Jeder Link verbindet eine `source`- (Start-) mit einer `target`- (Ziel-) Aktivität. Ein Link kann entweder undefiniert sein oder einen booleschen Wert (`transitionCondition`) haben. Ist eine Aktivität `target`-Aktivität eines oder mehrerer Links und haben alle eingehenden Links einen definierten Wert, wird eine `joinCondition` ausgewertet. Ist sie wahr („`true`“), so wird die Aktivität ausgeführt, andernfalls nicht.

Wird eine Aktivität aufgrund einer zu „`false`“ ausgewerteten `joinCondition` oder eines nicht abgearbeiteten Zweiges einer `switch`- oder `pick`-Aktivität nicht ausgeführt, so wird für alle eventuell vorhandenen ausgehenden Links die `transitionCondition` auf „falsch“ gesetzt. So können alle `target`-Aktivitäten geeignet auf den Fehler reagieren. Dieses Verhalten wird als *Dead-Path-Elimination* bezeichnet.

**Variable** Analog zu anderen Programmiersprachen bietet auch BPEL die Möglichkeit, Daten in Variablen zu speichern. Sie werden in einem Sichtbarkeitsbereich (`scope`) durch einen eindeutigen Namen und einen Datentyp definiert.

**CorrelationSets** Ein BPEL-Prozess wird in Instanzen ausgeführt. Um jede Nachricht, die vom Prozess gesendet oder empfangen wird, eindeutig einer Instanz zuzuordnen, werden sogenannte `CorrelationSets` verwendet. Ein `CorrelationSet` ist eine die Instanz identifizierende ID, die in jeder Nachricht enthalten ist und im `scope` definiert wird. Bei der Verwendung eines `CorrelationSets` können die Attribute `initiate` = „`yes|no`“ und `pattern` = „`in|out|out-in`“ spezifiziert werden. Hat `initiate` den Wert „`yes`“, so wird das `CorrelationSet` durch die erste gesendete oder empfangene Nachricht initialisiert. Hat `initiate` den Wert „`no`“, wird das `CorrelationSet` zu Prozessbeginn initialisiert. `pattern` gibt beim synchronen `invoke` an, ob das `CorrelationSet` zur eingehenden oder ausgehenden Nachricht gehört.

Ein BPEL-Prozess ist ein `scope`, in dem die oben erwähnten Handler sowie eine Aktivität eingebettet sind. In der inneren Aktivität können weitere Aktivitäten beliebig geschachtelt sein.

## 2.2 Petrinetze

In den 60er Jahren wurde von Carl Adam Petri der Grundstein für die Entwicklung von Petrinetzen gelegt [Pet62]. Seitdem wurde viel Literatur über Petrinetze veröffentlicht. Die folgenden Ausführungen basieren auf [Sta04] und [Web03].

Ein Petrinetz wird durch ein Netz grafisch repräsentiert. Es besteht aus zwei Arten von *Knoten* (*Plätze* und *Transitionen*), die durch gerichtete *Kanten* miteinander verbunden sind. Plätze werden als Kreise, Transitionen durch Quadrate repräsentiert.

Formal kann die Struktur eines Petrinetzes wie folgt definiert werden:

**Definition 1 (Petrinetz).** Ein Petrinetz ist ein Tripel  $N = (P, T, F)$  mit

## 2 Grundlagen

- $P$  ist eine Menge von *Plätzen*,
- $T$  ist eine Menge von *Transitionen*,
- $P$  und  $T$  sind disjunkt ( $P \cap T = \emptyset$ ),
- $F \subseteq (P \times T) \cup (T \times P)$  ist eine zweistellige *Flussrelation* (die Menge der Kanten)

Für jedes Element  $x \in S \cup T$  wird  $\bullet x = \{y \in S \cup T \mid (y, x) \in F\}$  als *Vorbereich* und  $x \bullet = \{y \in S \cup T \mid (x, y) \in F\}$  als *Nachbereich* des Elements  $x$  bezeichnet.

Um mittels eines Petrinetzes dynamische Aspekte eines Systems modellieren zu können, werden *Marken* auf Plätzen eingeführt. Grafisch werden solche Marken durch ausgefüllte schwarze Kreise ( $\bullet$ ) repräsentiert. Zusätzlich muss festgelegt werden, wie sich die Marken auf die Plätze verteilen können und nach welchen Regeln sich diese Verteilung ändern kann.

Der Zustand eines Netzes ergibt sich aus der Verteilung der Marken auf die Plätze und wird als *Markierung* bezeichnet.

**Definition 2 (Markierung).** Sei  $P$  die Menge der Plätze eines Petrinetzes  $N$ . Ein Vektor  $M \in \mathbb{N}^P$  über  $P$  heißt *Markierung des Netzes*  $N$ .

Für jede Transition können eine *Schaltbedingung* (Aktivierungsbedingung) und ein *Schalteffekt* definiert werden. Bei erfüllter Schaltbedingung kann eine Transition *schalten*, der Schalteffekt tritt ein. Das Netz wird von einem Zustand in einen anderen überführt. Da die Transitionen für die Zustandsänderung verantwortlich sind, werden sie als *aktive* Komponenten, Plätze als *passive* Komponenten des Netzes bezeichnet.

Durch verschiedene Typen von Markenmengen, Schaltbedingungen und Schalteffekten lassen sich unterschiedliche Arten von Petrinetzen klassifizieren. Zum Verständnis der vorliegenden Arbeit ist die Einteilung in Low-Level und High-Level-Netze ausreichend. Ein Netz mit ununterscheidbaren Marken wird als *Low-Level-Netz* bezeichnet, ein Netz mit unterscheidbaren Marken als *High-Level-Netz*. In einem High-Level-Netz müssen die Sorten der Marken definiert werden und jedem Platz eine Sorte zugewiesen werden, die festlegt, welche Marken auf diesem Platz liegen können.

Der Typ des von BPEL2PN erzeugten Petrinetzes wird als Stellen-Transitions-Netz (*S/T-Netz*) bezeichnet und ist wie folgt definiert:

**Definition 3 (S/T-Netz).** Sei  $N = (P, T, F)$  ein Petrinetz und  $\omega : F \rightarrow \mathbb{N}$  eine Funktion, die jeder Kante ein *Gewicht* zuordnet, dann heißt  $(P, T, F, \omega)$  S/T-Netz.

Beim S/T-Netz handelt es sich um ein Low-Level-Netz, das heißt, die Marken sind ununterscheidbar. Auf jedem Platz können beliebig viele Marken liegen. Die Schaltregel eines S/T-Netzes ist wie folgt definiert:

**Definition 4 (Schaltregel).** Sei  $N = (P, T, F)$  ein Petrinetz und  $M_1$  eine Markierung von  $N$ .

- $t \in T$  ist im Zustand  $M_1$  aktiviert, wenn auf jedem Platz  $p \in \bullet t$  mindestens  $\omega(p, t)$  Marken liegen.
- Eine aktivierte Transition  $t$  kann *schalten*. Wenn  $t$  schaltet, *konsumiert* sie von jedem Platz  $p \in \bullet t$   $\omega(p, t)$  Marken und *produziert* auf jeden Platz  $p' \in t \bullet$   $\omega(t, p')$  Marken. Als Ergebnis des Schaltvorgangs geht das Netz  $N$  in den Zustand  $M_2$  über.

## 2.3 Petrinetz-Semantik für BPEL

Die in [Sta04] vorgestellte Petrinetz-Semantik für BPEL basiert auf einem modularen Konzept: Für jedes BPEL-Konstrukt existiert ein eigenes Petrinetz (*Muster*). Um einen kompletten BPEL-Prozess zu transformieren, müssen Instanzen der einzelnen Muster gebildet und entsprechend der Prozess-Struktur zu einem einzigen Netz verknüpft werden. Um dies zu ermöglichen, wurde für jedes Muster eine Schnittstelle definiert. Eine Schnittstelle besteht aus Plätzen, die beim Zusammensetzen der Muster mit Plätzen der Schnittstelle eines anderen Musters verschmolzen werden.

Im Folgenden soll kurz auf die Besonderheiten der Muster für die einzelnen BPEL-Konstrukte eingegangen werden:

**Basisaktivitäten** Jede Basisaktivität wird durch ein Muster modelliert, das keine weiteren Muster mehr einbetten kann. In der Regel haben alle diese Muster dieselbe Standard-Schnittstelle, bestehend aus:

- **initial** (Beginn des Kontrollflusses im Muster)
- **final** (Ende des Kontrollflusses im Muster)
- **stop** (Dieser Platz wird benötigt, um den Kontrollfluss im Muster zu beenden. So kann das Abräumen aller im Muster befindlichen Marken ausgelöst werden, indem eine Marke auf den **stop**-Platz gelegt wird.)
- **stopped** (Wurden über den **stop**-Platz alle Marken abgeräumt, so liegt auf diesem Platz eine Marke.)
- **failed** (Kann die Aktivität eines Musters einen Fehler erzeugen, so wird die Schnittstelle um den **failed**-Platz erweitert.)

Ausnahmen bilden lediglich die Aktivitäten **throw** (kein **final**-Platz) sowie **terminate** und **compensate** (zusätzliche Plätze zur Anbindung an den **scope**).

**Strukturierte Aktivitäten** Jede strukturierte Aktivität wird durch ein Muster modelliert, das jedem einzubettenden Aktivitätsmuster eine Schnittstelle bereitstellt. Die Schnittstelle einer strukturierten Aktivität entspricht der Standard-Schnittstelle der Basisaktivitäten.

## 2 Grundlagen

**Link-Konzept** Für jeden in einem `flow` definierten Link werden zwei *Link-Plätze* erzeugt (`Link` und dessen Komplementärplatz `!Link`). Ist eine Aktivität `source` oder `target` eines Links, so wird deren Muster in ein Link-Muster eingebettet. Es bindet die Aktivität an die entsprechenden Link-Plätze an und erweitert die Standard-Schnittstelle bei Bedarf um einen `negLink`-Platz zur Dead Path Elimination.

**scope** Im `scope`-Muster werden die Muster der inneren Aktivität sowie der `Event-`, `Fault-` und `CompensationHandler` eingebettet. Eine Besonderheit ist das ebenfalls im `scope` eingebettete Stop-Muster, für das es keine Entsprechung als BPEL-Konstrukt gibt. Es initiiert das Beenden der inneren Aktivitäten und startet den `Compensation-` und `FaultHandler`. Die Schnittstelle des `scope`-Musters beinhaltet neben der oben beschriebenen Standard-Schnittstelle einige zusätzliche Plätze, die mit den Plätzen der Schnittstellen der Vater- und Kindsopes verschmolzen werden. Der Prozess ist ein Spezialfall des `scope` und besitzt ein leicht geändertes Muster.

**Datenobjekte** sind `Variable`, `CorrelationSets` und die Uhr (sie wird für die `wait`-Aktivität und Alarmereignisse benötigt). Datenobjekte werden durch High-Level-Plätze modelliert.

**Nachrichtenkanäle** werden für den Nachrichtenaustausch mit anderen Webservices benötigt und werden wie die Datenobjekte durch High-Level-Plätze modelliert.

Neben den High-Level-Plätzen der Nachrichtenkanäle und Datenobjekte existieren noch weitere High-Level-Plätze, wie z.B. der `failed`-Platz.

### 2.4 Werkzeuge

Während der Entwicklung von BPEL2PN wurden einige zusätzliche Werkzeuge genutzt, die im Folgenden kurz erläutert werden.

#### JDOM

Damit BPEL2PN einen BPEL-Prozess transformieren kann, muss dieser zunächst eingelesen werden.

Da eine BPEL-Datei ein spezielles XML-Dokument [BPCM<sup>+</sup>04] ist, kann hier auf vorhandene Arbeiten zurückgegriffen werden:

JDOM ist ein Open-Source-Projekt, das aus einer Java-Funktionsbibliothek besteht, die ein API (Application Programming Interface) zum Einlesen, Parsen und Manipulieren von XML-Dokumenten zur Verfügung stellt [Har02], [HM04]. Nach dem Einlesen wird ein XML-Dokument von JDOM durch eine Baumstruktur, ähnlich dem DOM (Document Object Model [WAB<sup>+</sup>98]), repräsentiert.

## **LoLA**

Um die von BPEL2PN generierten Petrinetze zu analysieren, wird ein geeignetes Werkzeug benötigt. Am Institut für Informatik der Humboldt-Universität wurde der Model-Checker LoLA entwickelt [Sch00]. LoLA bietet umfangreiche Analysemöglichkeiten für Low-Level-Petrinetze, wie z.B. die Überprüfung auf Deadlocks, tote Transitionen und die Erreichbarkeit gegebener Zustände.

Aber auch High-Level-Petrinetze können von LoLA analysiert werden. Dazu wird ein High-Level-Netz intern in ein Low-Level-Netz überführt.

BPEL2PN erzeugt standardmäßig eine Ausgabe-Datei im LoLA-Format, so dass das generierte Petrinetz mit LoLA analysiert werden kann.

## **dot**

Zur Fehlersuche und Veranschaulichung der generierten Netze kann eine Visualisierung hilfreich sein. Da die Entwicklung und Implementation von zufriedenstellenden Visualisierungsfunktionen sehr aufwendig und umfangreich ist und der Schwerpunkt der vorliegenden Arbeit auf der automatischen Transformation liegt, wird zu diesem Zweck das Werkzeug dot verwendet.

dot wurde von AT&T entwickelt und unter einer Open-Source-Lizenz zur Verfügung gestellt. Es bietet Funktionen zum Visualisieren von gerichteten und ungerichteten Graphen. Eine ausführliche Beschreibung kann in [GKN02] nachgelesen werden.

Auch wenn die Visualisierung von größeren Netzen schnell sehr unübersichtlich wird und auch die Funktion zum Hervorheben von Subnetzen noch fehlerträchtig ist, werden weniger komplexe Netze gut dargestellt.

Aus diesem Grund bietet BPEL2PN die Möglichkeit, eine Ausgabe-Datei im dot-Format zu erzeugen (siehe 3.3), aus der dot durch den folgenden Aufruf eine Bild-Datei erzeugt:

```
dot -Tgif <Eingabe-Datei> -o <Ausgabe-Datei>
```



## 3 Beschreibung von BPEL2PN

BPEL2PN ist ein Werkzeug zum Übersetzen eines BPEL-Prozesses in ein Petrinetz. Das kommandozeilenbasierte Werkzeug benötigt als Eingabe eine BPEL-Datei (siehe 3.1) und kann je nach verwendeten Optionen (siehe 3.3) ein Petrinetz in verschiedenen Formaten ausgeben (siehe 3.2).

Entwickelt wurde BPEL2PN mit dem JDK1.4.2 unter Borland JBuilder X. Da Java eine plattformunabhängige Programmiersprache ist, sollte es auf allen gängigen Betriebssystemen lauffähig sein, die über die Java Runtime Environment 1.4.2 (und höher) verfügen. Getestet wurde es unter Windows XP SP2.

### 3.1 Eingabeformat

Als Eingabedatei für BPEL2PN dient eine BPEL-Datei. Die Endung des Dateinamens sollte (wie üblich) `.bpe1` sein, damit die Namen der Ausgabedateien korrekt gebildet werden können. BPEL2PN setzt voraus, dass die einzulesende BPEL-Datei korrekt ist, also der in [ACLL03] veröffentlichten Sprachbeschreibung entspricht. Grobe Fehler in der Struktur werden von BPEL2PN abgefangen und führen zum Abbruch mit einer Fehlermeldung. Andere Fehler, wie z.B. laut BPEL-Spezifikation [ACLL03] unzulässige Links, können zu undefiniertem Verhalten oder auch zu Abstürzen von BPEL2PN führen.

Zu Debugging-Zwecken wurde ein zusätzliches, optionales Attribut `dummy = „yes|no“` eingeführt, das im XML-Element jeder BPEL-Aktivität definiert sein kann:

- `dummy = „no“` Das Attribut hat keinerlei Auswirkung.
- `dummy = „yes“` Die Aktivität wird nicht regulär transformiert. Stattdessen wird ein stark vereinfachtes Petrinetz erzeugt. Es enthält lediglich die Schnittstellen-Plätze sowie eine einzige Transition, die diese verbindet. Ist die Aktivität ein `scope`, so wird dessen innere Aktivität dennoch transformiert. Die `Event-`, `Fault-` und `CompensationHandler` sowie das `Stop-Muster` werden dann allerdings weder transformiert noch durch ein vereinfachtes Muster ersetzt. Dies ist nützlich um die Komplexität der resultierenden Petrinetze sowohl zu Debugging- als auch zu Analyse-Zwecken zu reduzieren. In einer fortführenden Arbeit könnten vereinfachende Muster entwickelt werden, die zu Analyse-Zwecken geeigneter sind.

### 3.2 Zielformate

BPEL2PN kann, je nach verwendeten Optionen, Ausgabedateien vier verschiedener Formate erzeugen:

#### 3.2.1 LoLA

Um eine Analyse mit LoLA zu ermöglichen, wird von BPEL2PN standardmäßig eine Ausgabe im LoLA-Format erzeugt, wie in [Sch00] beschrieben. Lautet

### 3 Beschreibung von BPEL2PN

der Name der Eingabe-Datei `name.bpel`, so wird die erzeugte LoLA-Datei `name.lo1a` heißen. Die Plätze und Transitionen des Petrinetzes werden jeweils fortlaufend durchnummeriert und durch Voranstellen des Buchstabens 't' bzw. 'p' als Transition (z.B. `t23`) bzw. Platz (z.B. `p128`) kenntlich gemacht.

Wird beim Aufruf von BPEL2PN die Option `-detailed` gewählt, so werden Informationen über die Herkunft der Plätze und Transitionen mit in deren Namen aufgenommen (siehe 3.2.5).

#### 3.2.2 Info-Datei

Um weitere Informationen über die Herkunft der Knoten des generierten Petrinetzes zu erhalten, kann BPEL2PN eine Info-Datei erzeugen. Lautet der Name der Eingabe-Datei `name.bpel`, so wird die erzeugte Info-Datei `name.info` heißen. Zu jedem Knoten des Petrinetzes können der Info-Datei folgende Informationen entnommen werden:

- Bezeichnung des Knotens wie in der LoLA-Datei
- Art des zugehörigen Musters (z.B. `invoke`, `while`, `scope`, `FaultHandler`, `Stop-Muster`)
- falls vorhanden, Name des zugehörigen Musters (Wert des Attributes `name` der BPEL-Aktivität in der BPEL-Datei. Wird beim Aufruf von BPEL2PN die Option `-ids` gewählt, so wird dem Namen die erzeugte `id` vorangestellt.)
- Name des Knotens im zugehörigen Muster wie in den Abbildungen aus [Sta04] (Bei einigen Mustern weicht die Benennung der Knoten davon ab, Näheres dazu kann in Abschnitt 3.2.5 nachgelesen werden.)
- Bei Plätzen wird, falls vorhanden, eine genauere Beschreibung des Platzes (z.B. `initial`, `final`, `negLink`) mit ausgegeben.
- Ist ein Platz während der Transformation mit anderen Plätzen verschmolzen worden, so wird für ihn eine Liste der mit ihm verschmolzenen Plätze mit jeweils allen hier aufgeführten Informationen erzeugt.

Um das Auffinden eines bestimmten Knotens in der Info-Datei zu erleichtern, sind alle Knoten aufsteigend nach der Bezeichnung in der LoLA-Datei sortiert. Zusätzlich werden die Knoten auch nach den zugehörigen Mustern gegliedert ausgegeben, um zusammengehörende Plätze und Transitionen schnell auffinden zu können.

#### 3.2.3 BPEL-Datei mit ids

Die Verwendung des Attributs `name` in den Aktivitäten eines BPEL-Prozesses ist nicht vorgeschrieben, und die dem Attribut zugewiesenen Werte müssen nicht eindeutig sein. Deswegen ist es nicht sichergestellt, dass jedes in der LoLA- oder Info-Datei aufgeführte Muster eindeutig einer Aktivität im BPEL-Prozess zugeordnet werden kann. Wird beim Aufruf von BPEL2PN die Option `-ids`

gewählt, so wird für jede Aktivität eine eindeutige Identifikations-Nummer generiert und in die Info- und (bei gewählter `-detail`-Option) in die LoLA-Datei mit aufgenommen.

Um über die Identifikations-Nummer auf die zugehörige Aktivität im BPEL-Prozess schließen zu können, wird eine neue BPEL-Datei erzeugt, in der jede Aktivität das zusätzliche Attribut `id = „X“` enthält, wobei `X` die generierte Identifikations-Nummer ist.

Lautet der Name der Eingabe-Datei `name.bpel`, so wird die erzeugte BPEL-Datei `name_ids.bpel` heißen.

### 3.2.4 dot

Um ein erzeugtes Petrinetz grafisch darzustellen, wird von BPEL2PN ein Ausgabe-Format unterstützt, das von `dot` (siehe Abschnitt 2.4 und [GKN02]) gelesen werden kann. Die Bezeichnungen der Knoten sind dabei standardmäßig analog zu den Bezeichnungen in der LoLA-Datei (siehe 3.2.1). Lautet der Name der Eingabe-Datei `name.bpel`, so wird die erzeugte Info-Datei `name.dot` heißen.

Da die erzeugten Petrinetze schnell sehr groß werden, ist die Visualisierung des Netzes eines kompletten Prozesses wenig sinnvoll, da sie recht unübersichtlich ist. In Verbindung mit dem Attribut `dummy` (siehe Abschnitt 3.1) ist es jedoch möglich, Petrinetze zu erzeugen, die gut zur Visualisierung geeignet sind. Zusätzlich sind in Abschnitt 3.3 eine Reihe von Optionen zur Anpassung der `dot`-Ausgabe beschrieben.

### 3.2.5 Benennung der Knoten

In der Info-Datei (bei gewählter `-detail`-Option auch in der LoLA-Datei) werden die Namen der Plätze und Transitionen aufgeführt, wie sie in den Abbildungen aus [Sta04] angegeben sind. Bei einigen Mustern bestehen jedoch aus implementationstechnischen Gründen Ausnahmen:

Bei Mustern die sich untereinander sehr ähneln, bzw. aus Teilmustern zusammengesetzt werden, beziehen sich die Namen der Knoten immer auf die Abbildung des komplexesten Musters. Im Einzelnen handelt es sich dabei um folgende Muster (Die Abbildungsverweise beziehen sich auf [Sta04]):

- Link-Muster: Abb.42
- Stop-Muster: Abb.27, p12 (final) aus Abb.44 heißt p33
- Muster des standardmäßigen `FaultHandlers`: Abb.28, p9 (final) aus Abb.45 heißt p30
- Muster des nutzerdefinierten `FaultHandlers`: Abb.29, p3 (final) aus Abb.46 heißt p30,
- `CompensationHandler`-Muster: Abb.33, t6 aus Abb.32 heißt t7
- `scope/process`-Muster: Abb.23 (`scope`), t17/t18 (Terminated/!Terminated) aus Abb.24 (`process`) heißen t71/t70

### 3 Beschreibung von BPEL2PN

- Die neu hinzugefügten Schnittstellenplätze `!push` (siehe 4.6.3) heißen `p80` (untere Schnittstelle) bzw. `p81` (obere Schnittstelle)

Bei einigen strukturierten Aktivitäten sowie anderen Mustern, die eine variable Anzahl von Netz-Zweigen besitzen, werden die Knoten folgendermaßen benannt: Der Name eines Knotens setzt sich zusammen aus dem Namen des Knotens im linken Zweig in der Abbildung, gefolgt von einem Punkt und einer fortlaufenden Nummer des Zweiges (z.B. `p7.0`, `t11.3`, `p8.2`). Auf dieselbe Art werden auch die Plätze `push`, `!push` und `compScope` des `scope` bezeichnet.

Wird beim Aufruf von BPEL2PN die Option `-detail` gewählt, so werden die Knoten in der LoLA- und der dot-Datei abweichend von Abschnitt 3.2.1 wie folgt bezeichnet:

- Gehört der Knoten zu einem Muster, das keine Aktivität darstellt (z.B. `CompensationHandler`, `Stop-Muster`), wird er folgendermaßen bezeichnet: `[p|t]<lfd nr>.[Scope|Process].<Name des Scope>.<Typ des Musters>.<Knotenname wie in der Abbildung des Musters>`
- Repräsentiert ein Platz eine Variable, ein `CorrelationSet` oder einen Nachrichtenkanal, dann setzt sich dessen Bezeichnung folgendermaßen zusammen: `[var|cor|chnl]<lfd nr>.<Name wie im BPEL-Prozess>`. Der Name eines Nachrichtenkanals setzt sich aus `PartnerLink`, `PortType`, `Operation` und `Richtung` zusammen.
- Der Platz, der die Uhr repräsentiert, heißt immer `Time.Clock`.
- Alle anderen Knoten werden folgendermaßen bezeichnet: `[t|p]<lfd nr>.<Aktivitätstyp>.<Aktivitätsname>.<Knotenname wie in der Abbildung des Musters>`
- Ist die Option `-ids` gewählt, so wird dem Namen jeder Aktivität eine eindeutige Nummer vorangestellt (siehe 3.2.3).

### 3.3 Aufruf

BPEL2PN wird von der Kommandozeile aufgerufen:

```
BPEL2PN Eingabe-Datei [optionen]
```

`Eingabe-Datei` ist eine wie in Abschnitt 3.1 beschriebene BPEL-Datei, die den zu transformierenden BPEL-Prozess enthält.

`optionen` ist optional und kann eine oder mehrere der folgenden Optionen sein:

**-info** Es wird eine Info-Datei erzeugt (siehe 3.2.2).

**-detail** In der LoLA- und dot-Datei wird eine detaillierte Bezeichnung der Knoten erzeugt (siehe 3.2.5).

**-ids** Alle Aktivitäten des BPEL-Prozesses werden mit einer eindeutigen Nummer versehen, die dem Namen der Aktivität vorangestellt wird. Zusätzlich wird eine BPEL-Datei wie in Abschnitt 3.2.3 beschrieben erzeugt.

**-dot** Eine Datei im dot-Format wird erzeugt (siehe 3.2.4).

**-nomerge** Es werden keine Plätze verschmolzen. Statt dessen werden in der dot-Ausgabe die zu verschmelzenden Plätze durch eine spezielle Kante verbunden. Ist diese Option gewählt, so wird keine LoLA-Datei erzeugt.

**-cluster** Jedes BPEL-Konstrukt wird in der dot-Ausgabe als Subnetz gezeichnet. Bei komplexeren Netzen ist die Ausgabe allerdings unbefriedigend, da sich dot noch in der Entwicklung befindet und diesbezüglich noch nicht ausgereift ist.

**-tplname** Statt der Bezeichnung der Knoten wie in der LoLA-Datei, werden die Namen aus den Abbildungen der Muster wie in [Sta04] verwendet.

**-noFH** Es werden keine `FaultHandler` generiert.

**-noCH** Es werden keine `CompensationHandler` generiert.

**-noST** Es werden keine Stop-Muster generiert.

**-v(ersion)** Die Versionsnummer von BPEL2PN wird ausgegeben.

**-h(elp)** Eine Kurzhilfe wird ausgegeben.



## 4 Implementierung

Bereits in Abschnitt 1.2 wurden Einlesen, Transformation und Ausgabe als die drei Hauptaufgaben des zu implementierenden Werkzeugs BPEL2PN identifiziert. Nachfolgend werden das Funktionsprinzip, die Klassenstruktur und einige Besonderheiten der Implementierung erläutert.

### 4.1 Prinzip

Die Idee der Petrinetz-Semantik beruht darauf, jedes BPEL-Sprachkonstrukt in ein eigenes Petrinetz-Muster zu übersetzen und die einzelnen Muster entsprechend der Prozess-Struktur zu einem einzigen Netz zusammenzusetzen. Es ist naheliegend, dieses Konzept auch für BPEL2PN zu übernehmen. So wird für jedes Muster der Semantik eine Methode implementiert, die das Netz des jeweiligen Musters generiert.

Im Folgenden wird zunächst beschrieben, in welcher Reihenfolge die einzelnen Methoden aufgerufen werden. Anschließend wird erläutert, wie die Netze verschiedener Muster durch Verschmelzen ihrer Schnittstellen zu einem einzigen Petrinetz verbunden werden.

#### Aufrufreihenfolge der Methoden

Für die Implementierung sind sowohl ein rekursiver als auch ein iterativer Lösungsansatz denkbar. Da ein BPEL-Prozess auf rekursiven Strukturen basiert, können diese während eines ebenfalls rekursiven Transformationsprozesses direkt übernommen werden. Aufgrund dieses Vorteils wurde der iterative Ansatz nicht weiter verfolgt.

Werden in ein Muster  $M$  die Muster  $M_0, \dots, M_n$  eingebettet, so existieren folgende Ansätze, wie die zu  $M$  gehörende Methode das resultierende Petrinetz generieren kann:

- **Von innen nach außen:** Zuerst werden die Netze von  $M_0, \dots, M_n$  generiert, anschließend das Netz von  $M$ .
- **Von außen nach innen:** Zuerst wird das Netz von  $M$  generiert, anschließend die Netze von  $M_0, \dots, M_n$ .
- **Von innen nach außen und von außen nach innen:** Zuerst wird ein Teil des Netzes von  $M$  generiert, anschließend die Netze der Muster  $M_0, \dots, M_n$  und schließlich der restliche Teil von  $M$ .

Tatsächlich wird hier keiner dieser Ansätze konsequent verfolgt, sondern für jedes Muster der geeignetste gewählt. Ausschlaggebend für die Wahl der Vorgehensweise sind dabei die folgenden Punkte:

- Oft ist ein einbettendes Muster (*Elternmuster*) von der Struktur des eingebetteten (*Kindmuster*) abhängig (siehe 4.5 und 4.6), sodass letzteres erst generiert werden muss, um die benötigten Informationen zu erhalten (von innen nach außen).

## 4 Implementierung

- Bei Mustern einiger strukturierter Aktivitäten muss jedoch zuerst ein Teil des einbettenden Netzes erzeugt werden, um die inneren Aktivitäten daran anbinden zu können (von außen nach innen).

Soweit dies unter Berücksichtigung der beiden oben genannten Punkte möglich ist, wird die Reihenfolge der Erzeugung der einzelnen Muster dem Kontrollfluss des Prozesses angepaßt. Dadurch erhält der Quell-Text der einzelnen Methoden eine nachvollziehbare Struktur.

Das zu erzeugende Petrinetz wird also zum Teil von außen nach innen, und zum Teil von innen nach außen aufgebaut.

### Verschmelzen der Schnittstellen

Um die Netze der einzelnen Muster miteinander zu verbinden, besitzt jedes Muster, wie in Abschnitt 2.3 beschrieben, eine Schnittstelle, deren Plätze mit den Schnittstellenplätzen eines anderen Musters verschmolzen werden.

Das Verschmelzen zweier Schnittstellenplätze  $P_1$  und  $P_2$  kann folgendermaßen beschrieben werden:

- Ein neuer Platz  $P_{neu}$  wird erzeugt.
- Von allen Transitionen der Vorbereiche von  $P_1$  und  $P_2$  wird eine Kante zu  $P_{neu}$  erzeugt.
- Von  $P_{neu}$  wird jeweils eine Kante zu allen Transitionen der Nachbereiche von  $P_1$  und  $P_2$  erzeugt.
- $P_1$  und  $P_2$  werden mit all ihren Kanten gelöscht.

In der Klasse `PetriNet` werden die Objektreferenzen eines neu erzeugten Platzes redundant an verschiedenen Stellen gespeichert. Dadurch können die Methoden zur Ausgabe des Netzes einfacher implementiert werden. Näheres dazu kann der JavaDoc zu `BPEL2PN` entnommen werden.

Durch das redundante Speichern der Plätze würde jedoch das Löschen eines bestehenden Platzes mit all seinen Kanten sehr aufwendig sein. Die rekursive Struktur eines BPEL-Prozesses legt vielmehr ein Vorgehen nahe, bei dem auf ein Verschmelzen von Plätzen, wie oben beschrieben, verzichtet werden kann. So kann auf ein Löschen von Plätzen verzichtet werden.

Die grundlegende Idee besteht darin, dass jeder Schnittstellenplatz nur ein einziges Mal erzeugt wird. Soll ein Schnittstellenplatz eines Musters mit dem eines anderen verschmolzen werden, so wird nun dieser einmal erzeugte Platz an die Netze beider Muster angebunden.

Zu klären bleibt, zu welchem Zeitpunkt ein Schnittstellenplatz erzeugt wird und auf welche Weise die Methoden zum Generieren der einzelnen Muster Zugriff auf die Objektreferenzen ihrer Schnittstellenplätze erhalten. Es werden zwei Fälle unterschieden, die anhand der Abbildungen 2 und 3 im Folgenden besprochen werden. Die Abbildungen zeigen schematisch mehrere auf unterschiedliche Weise ineinander verschachtelte Muster. Gepunktete Linien symbolisieren ein Verschmelzen von Plätzen

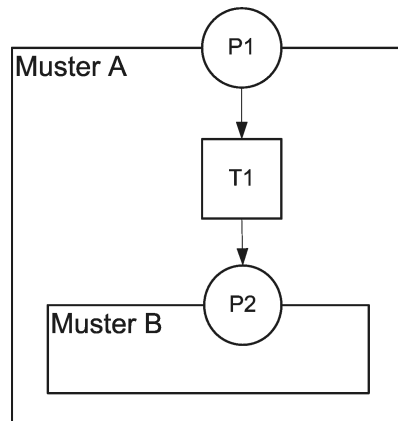


Abbildung 2: Weiterreichen eines Schnittstellenplatzes an ein Kindmuster

**Fall 1:** Weiterreichen eines Schnittstellenplatzes an ein Kindmuster:

Abbildung 2 zeigt den Fall, dass ein Platz der äußeren Schnittstelle von Muster B an das Netz von Muster A angebunden werden muss. Wie oben beschrieben, ruft die Methode zum Generieren des Netzes von Muster A die von Muster B auf. Beide Methoden benötigen hier Zugriff auf P2. Um dies zu erreichen gibt es zwei Möglichkeiten:

- P2 wird in Muster A erzeugt und als Parameter der Methode für Muster B übergeben. (Der Platz wird von außen nach innen *weitergereicht*.)
- P2 wird in Muster B erzeugt und als Rückgabeparameter der Methode für Muster B übergeben, die den Platz dann an ihr Netz anbindet. (Der Platz wird von innen nach außen *weitergereicht*.)

Welche dieser beiden Vorgehensweisen tatsächlich verwendet wird, spielt in den meisten Fällen keine Rolle. Wichtig ist nur, dass beide Methoden dasselbe Verfahren wählen, damit ein Platz nicht sowohl in Muster A als auch in Muster B erzeugt wird.

Einige wenige Plätze sind jedoch nur optional in einer Schnittstelle vorhanden. Je nachdem, ob ein solcher Platz in der Schnittstelle existiert oder nicht, muss die Methode des Elternmusters darauf reagieren und ihr Netz entsprechend anpassen. Solche Plätze können nur von innen nach außen weitergereicht werden, da die Methode des Kindmusters entscheidet, ob der Platz nötig ist oder nicht. Ein Beispiel für einen solchen Schnittstellenplatz ist der `negLink`-Platz (siehe 2.3 und 4.5).

**Fall 2:** Weiterreichen eines Schnittstellenplatzes an mehrere Kindmuster:

Abbildung 3 illustriert den Fall, dass ein Platz P1 der äußeren Schnittstelle mit den Plätzen P2 und P3 in zwei unterschiedlichen inneren Schnittstellen verschmolzen werden muss. Dieser Fall tritt beispielsweise bei den `stop`- und `stopped`-Plätzen auf.

#### 4 Implementierung

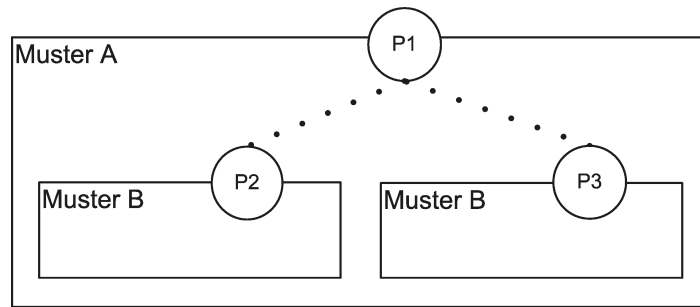


Abbildung 3: Weiterreichen eines Schnittstellenplatzes an mehrere Kindmuster

Hierbei ist es wichtig, dass die Schnittstellenplätze von außen nach innen weitergereicht werden. Würde von innen nach außen weitergereicht, so würde Methode A die Plätze P2 und P3 der Methoden B und C entgegennehmen. Dadurch entstünde ein Konflikt, denn nur einer dieser beiden Plätze könnte in Muster A die Rolle von P1 übernehmen und an das Elternmuster weitergereicht werden.

Die aufgeführten Beispiele machen deutlich, dass die Richtung in einigen Fällen eine wichtige Rolle spielt. Sie kann vom Typ des Schnittstellenplatzes abhängen (z.B. bei `negLink`), als auch von der Struktur des Elternmusters (z.B. bei `stop`).

Die Schnittstelle eines Musters besteht nicht nur aus einem, sondern aus mehreren Plätzen. Würde einer Methode zum Generieren eines Netzes jeder Schnittstellenplatz als einzelner Parameter übergeben werden, so würde das die Aufrufe dieser Methoden sehr komplex und umständlich machen. Deshalb wird für jeden Schnittstellentyp (z.B. `Aktivität`, `FaultHandler`) eine Schnittstellenklasse eingeführt, in der eine Objektreferenz auf jeden Schnittstellenplatz abgelegt werden kann. Jeder Methode zum Generieren eines Netzes wird bei deren Aufruf ein Objekt der passenden Schnittstellenklasse übergeben (*Schnittstellenobjekt*).

Eine aufgerufene Methode überprüft für jeden möglichen Schnittstellenplatz, ob bereits eine entsprechende Objektreferenz im ihr übergebenen Schnittstellenobjekt vorhanden ist. Ist dies der Fall, so bindet sie ihr Netz an diesen Platz an. Wenn nicht, wird ein neuer Platz erzeugt und seine Referenz im Schnittstellenobjekt gespeichert. Dadurch stehen der aufrufenden Methode die Plätze anschließend zur Verfügung.

Die aufrufende Methode kann bestimmen, in welche Richtung ein Schnittstellenplatz weitergereicht wird:

- Schreibt die aufrufende Methode die Objektreferenz eines Platzes ins Schnittstellenobjekt der aufzurufenden Methode, wird dieser Platz von außen nach innen weitergereicht.
- Wird von der aufrufenden Methode ein möglicher Schnittstellenplatz nicht ins Schnittstellenobjekt der aufzurufenden Methode geschrieben, so wird

er von der aufgerufenen Methode erzeugt und von innen nach außen weitergereicht.

Bei den meisten Schnittstellenplätzen spielt es jedoch keine Rolle, in welcher Richtung sie weitergereicht werden. Um trotzdem ein einheitliches Verfahren anzuwenden, wurde festgelegt, dass, soweit es die Struktur des Elternmusters nicht anders erfordert (wie in Fall 2 beschrieben), alle Plätze von innen nach außen weitergereicht werden.

Im Falle des `scope`-Musters kann jedoch der Fall auftreten, dass Schnittstellenplätze über mehrere Verschachtelungsebenen hinweg mit denen anderer Muster verschmolzen werden müssen. Dies wird dadurch realisiert, dass die Schnittstellenobjekte des `scope`-Musters global verfügbar gemacht werden. In Abschnitt 4.6 wird darauf genauer eingegangen.

Alternativ zum Durchreichen der Schnittstellenobjekte wäre ein anderer Ansatz denkbar: Ähnlich zum oben beschriebenen `ScopeInterface` könnten in einer geeigneten globalen Datenstruktur die Schnittstellenplätze allen Methoden zugänglich gemacht werden. Dies würde ein Konzept zur Pflege der Datenstruktur erforderlich machen, das sicherstellt, dass jeder Schnittstellenplatz den passenden Mustern zugeordnet werden kann.

Dieser Ansatz wurde jedoch nicht weiter verfolgt, da das Durchreichen der Plätze durch das rekursive Aufrufen der einzelnen Methoden nahe gelegt wird.

## 4.2 Umsetzung

BPEL2PN wird in vier Hauptklassen implementiert. Dadurch wird eine übersichtliche und dem Prinzip der Objektorientierung genügende Struktur erreicht. Die Funktionalität für jede der drei Hauptaufgaben (siehe 1.2) wurde in jeweils einer Klasse zusammengefasst. Eine Übersicht gibt das UML-Klassendiagramm [Bur97] in Abbildung 4. Eine andere Aufteilung erscheint nicht sinnvoll.

Im Folgenden werden die einzelnen Klassen kurz vorgestellt. Genauere Informationen über Subklassen sowie Methoden und Variablen sind der Dokumentation des Quelltextes zu entnehmen.

### 4.2.1 BPELDoc

Diese Klasse realisiert die interne Repräsentation eines BPEL-Prozesses. Sie beinhaltet Methoden zum Einlesen einer BPEL-Datei und zu deren Überführung

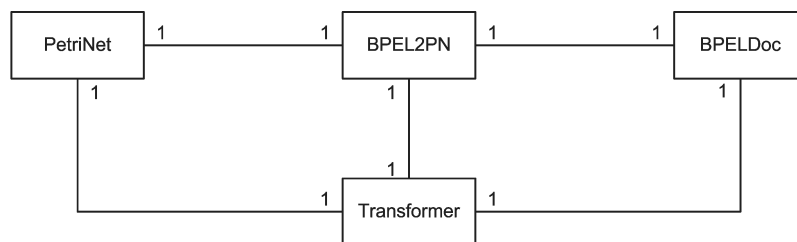


Abbildung 4: Klassendiagramm von BPEL2PN

## 4 Implementierung

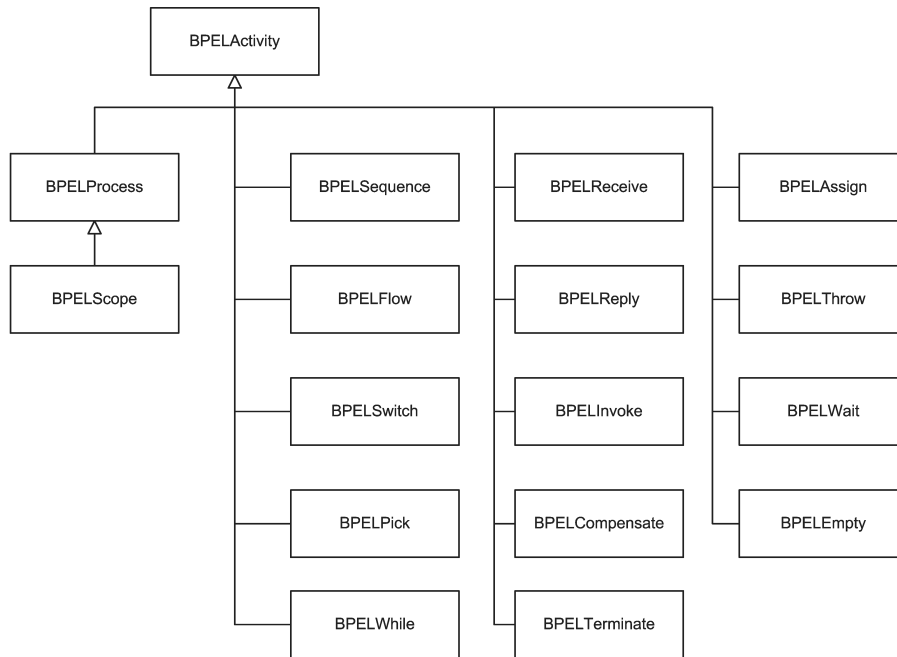


Abbildung 5: Klassenbaum der BPEL-Aktivitäten

in eine interne Repräsentation des BPEL-Prozesses.

Mit Hilfe von JDOM (siehe 2.4) wird eine BPEL-Datei eingelesen und steht anschließend als DOM-ähnliche [WAB<sup>+</sup>98] Baumstruktur im Speicher zur Verfügung. Da diese jedoch universell für XML-Dokumente entwickelt wurde, ist sie recht umständlich zu handhaben. Zwar wäre es möglich, die Transformation in ein Petrinetz direkt aus der JDOM-Struktur vorzunehmen, jedoch würde es den Quelltext sehr unübersichtlich und unflexibel machen. Das würde zu einer höheren Fehleranfälligkeit führen und spätere Erweiterungen wären schwieriger zu implementieren. Deswegen wird der JDOM-Baum in eine *interne Repräsentation* des BPEL-Prozesses überführt, die speziell für die anschließende Transformation in ein Petrinetz konzipiert wurde.

Die Überführung wurde rekursiv realisiert. Dabei existiert für jedes BPEL-Konstrukt eine eigene Methode. Da die einzelnen Konstrukte eines BPEL-Prozesses rekursiv verschachtelt sind, sind andere denkbare iterative Ansätze hier wenig sinnvoll.

Intern wird ein BPEL-Prozess durch eine Baumstruktur dargestellt, deren Knoten alle vom Typ `BPELActivity` sind. Die Blätter des Baumes stellen die Basisaktivitäten dar. Die Wurzel ist vom Typ `BPELProcess`. Abbildung 5 zeigt die dazugehörige Klassenhierarchie.

### 4.2.2 PetriNet

Diese Klasse realisiert neben einer Datenstruktur zum Speichern von Petrinetzen eine Reihe von Methoden, um Plätze, Transitionen und Kanten zu erzeugen und Subnetzen zuzuordnen. Des Weiteren stehen Methoden zur Definition einer Anfangsmarkierung sowie zum Schreiben des erzeugten Netzes als Datei in ver-

schiedenen Formaten (siehe.3.2) zur Verfügung . Einige Besonderheiten dieser Klasse werden hier kurz dargestellt:

**Lesekante** Wird eine Kante von einem Platz P zu einer Transition T als Lesekante definiert, so wird beim Schalten von T die Marke auf P nicht konsumiert.

Da Lesekanten in Low-Level-Netzen nicht existieren, jedoch in einigen Mustern der Petrinetz-Semantik verwendet werden, wird ihr Verhalten, wie in Abbildung 6 illustriert, durch eine Schleife simuliert. Beide Konstrukte sind semantisch äquivalent. So wird durch Schalten von T eine Marke von P konsumiert, aber durch die entgegengesetzte Kante wieder eine Marke auf P produziert.

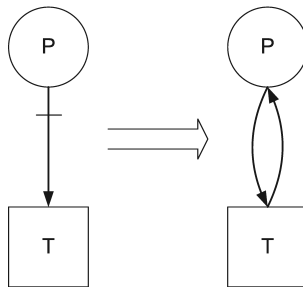


Abbildung 6: Simulation einer Lesekante durch zwei entgegengesetzte einfache Kanten

**Reset-Kante** Im Stop-Muster wird eine Reset-Kante [DFS98] verwendet. In Abbildung 7 ist die Kante von PR nach T2 eine Reset-Kante. Beim Schalten von T2 werden alle Marken von PR abgeräumt. Diese Art von Kanten existiert ebenfalls nicht in Low-Level-Netzen. Da hier keine einfach umzusetzende Low-Level-Lösung existiert wie bei der Lesekante, muss an dieser Stelle auf eine High-Level-Konstruktion zurückgegriffen werden, die von LoLA zu einem Low-Level-Netz entfaltet wird. Jedem Platz P, von dem eine Reset-Kante ausgeht (*Reset-Platz*), wird eine Sorte zugeordnet, die einen Zahlenbereich von null bis zur maximalen Anzahl der Marken auf P darstellt. Zusätzlich werden für P eine Reihe von Funktionen definiert, die beim Schalten von Transitionen, die P im Vor- oder Nachbereich haben, aufgerufen werden. Anhand von Abbildung 7 wird dies erläutert:

- Beim Schalten von T1 wird eine Marke von P1 konsumiert und mittels einer Funktion der Wert von PR inkrementiert.
- Beim Schalten von T2 wird mittels einer Funktion der Wert von PR auf null gesetzt. Auf P2 wird eine Marke produziert. Eine Aktivierungsbedingung stellt sicher, dass T1 nur schalten kann wenn der Wert von PR größer als null ist.
- Beim Schalten von T3 wird mittels einer Funktion der Wert von PR dekrementiert. Auf P3 wird eine Marke produziert. Wie bei T2 stellt auch hier eine Aktivierungsbedingung sicher, dass T3 nur schalten kann wenn der Wert von PR größer als null ist.

## 4 Implementierung

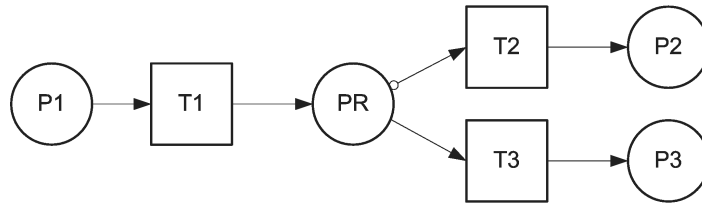


Abbildung 7: Resetkante von PR nach T2

Da die Klasse `PetriNet` jedoch nur für Low-Level-Netze ausgelegt ist, werden die High-Level-Konstrukte nicht direkt im Speicher abgelegt. Es wird lediglich die betroffene Kante als Reset-Kante markiert. Zusätzlich wird für den Platz PR die maximale Anzahl der Marken gespeichert, die auf ihm liegen können. Beim Erzeugen der LoLA-Datei werden dann für jeden Reset-Platz die entsprechenden Sorten und Funktionen ausgegeben. Beim Generieren der LoLA-Ausgabe einer Transition T wird überprüft, ob im Vor- oder Nachbereich von T ein Reset-Platz existiert und ob es sich bei der verbindenden Kante um eine Reset-Kante handelt. So kann die korrekte Schaltregel für T ausgegeben werden.

Das ist zwar eine sehr starre Umsetzung, jedoch wenig aufwendig und für den Zweck ausreichend. Der Ansatz einer alternativen Lösung wird in Abschnitt 5.2 vorgestellt und soll Inhalt weiterer Arbeit sein.

**Verschmelzen von Plätzen** Wie in Abschnitt 4.1 beschrieben, findet kein physisches Verschmelzen von Plätzen statt. Dennoch existiert eine Methode `mergePlace`. Sie verschmilzt jedoch keine existierenden Plätze, sondern ergänzt einen existierenden Platz um die Meta-Information, in welchem weiteren Muster er enthalten ist, sowie seine Bezeichnung in diesem Muster. So wird ein Verschmelzen des Knotens simuliert.

Dies ist nötig, um in der Info-Datei (siehe 3.2) die Information über die verschmolzenen Plätze einschließen zu können sowie in der dot-Ausgabe die Verschmelzung der Knoten zu visualisieren (siehe 3.3).

### 4.2.3 Transformer

Diese Klasse realisiert die eigentliche Transformation. Sie beinhaltet alle entsprechenden Methoden sowie die Subklassen für die Schnittstellen der einzelnen Muster. Aus einem gegebenen `BPELDoc`-Objekt wird in einem `PetriNet`-Objekt das entsprechende Petrinetz erzeugt.

Auf einige Besonderheiten der Implementierung wird in den Abschnitten 4.3 bis 4.6 sowie in der JavaDoc zu `BPEL2PN` eingegangen.

### 4.2.4 BPEL2PN

Diese Klasse beinhaltet die `main`-Methode von `BPEL2PN`. Sie wertet die Kommandozeilen-Parameter aus und instanziiert und verknüpft die Objekte der anderen drei Klassen.

### 4.3 Transformation der Datenobjekte

Da von Daten abstrahiert wird, wird ein Datenobjekt als Low-Level-Platz modelliert. So können in einem Muster weder Daten gelesen noch geschrieben werden. Dieses Verhalten wird lediglich durch eine Low-Level-Marke simuliert. Im Einzelnen handelt es sich dabei um die folgenden Objekttypen, deren Plätze jeweils in unterschiedlichen Methoden der `Transformer`-Klasse erzeugt werden:

**Uhr** Der Platz der Uhr heißt immer `Time.Clock` und wird bei der Transformation des Prozesses erzeugt. Er kann von allen Mustern gelesen werden und wird als globale Variable im `Transformer`-Objekt gespeichert.

**CorrelationSets und Variable** Bei der Transformation eines `scope` werden auch die Plätze der in ihm definierten `CorrelationSets` und Variablen erzeugt. Sie werden im zugehörigen `ScopeInterface`-Objekt gespeichert. Wird innerhalb des Prozesses ein `CorrelationSet` oder eine Variable referenziert, die nicht im aktuellen oder einem Vaterscope definiert ist, so bricht das Programm mit einer Fehlermeldung ab. Wenn in einem Kindscope eine Variable oder ein `CorrelationSet` definiert wird, dessen Name schon im Vaterscope für ein Objekt gleichen Typs vergeben ist, so wird das Objekt im Vaterscope durch das im Kindscope überdeckt.

Wird ein bereits definiertes `CorrelationSet` mit `initiate = „no“` verwendet, so wird es beim Transformieren der entsprechenden Aktivität mit einer Anfangsmarkierung versehen. Ein bisher noch nicht gelöstes Problem entsteht, wenn dasselbe `CorrelationSet` an einer Stelle mit `initiate = „no“` und an anderer Stelle mit `initiate = „yes“` verwendet wird. Dieses Problem kann aber durch statische Analyse gelöst werden, was jedoch Gegenstand weiterer Arbeit sein soll. In der aktuellen Version von BPEL2PN wird ein solches `CorrelationSet` immer mit einer Anfangsmarkierung versehen.

**Nachrichtenkanäle** Die Schnittstelle eines Webservices wird in dessen WSDL-Datei beschrieben [CCMW01]. Dazu wird unter anderem eine Menge von `portTypes` definiert. Jeder `portType` ist eine Menge von abstrakten *Operationen*, die jeweils ein Muster für den Nachrichtenaustausch spezifizieren, das der Webservice unterstützt. Ein Muster legt ein oder zwei Nachrichten fest, wobei zwischen eingehenden und ausgehenden Nachrichten unterschieden wird. Im BPEL-Prozess können sogenannte `partnerLinks` definiert werden, die sich auf einen in der WSDL-Datei definierten `partnerLinkType` beziehen, über den eindeutig ein `portType` referenziert wird.

In der betrachteten Semantik wird der Begriff des *Nachrichtenkanals* verwendet. Ein Nachrichtenkanal wird spezifiziert durch einen `portType`, eine Operation sowie die Richtung und den Namen der Nachricht.

In BPEL2PN ist ein Nachrichtenkanal im BPEL-Prozess durch seinen `partnerLink`, `PortType`, Operation und Richtung implizit definiert. Die Richtung einer Nachricht ergibt sich aus der entsprechenden Aktivität. Bei der ersten Verwendung eines Nachrichtenkanals wird sein Platz erzeugt und global im

## 4 Implementierung

`transformer`-Objekt gespeichert. So steht er allen Aktivitäten zur Verfügung und kann bei Bedarf an das Muster einer Aktivität angebunden werden.

Da die Anbindung des BPEL-Prozesses an andere Webservices von der umzusetzenden Petrinetz-Semantik nicht betrachtet wird und die Nachrichtenkanäle aus dem BPEL-Prozess heraus eindeutig referenziert werden, kann auf ein Einlesen der WSDL-Datei verzichtet werden.

Ein alternativer Ansatz könnte so aussehen, dass vor Beginn der Transformation des BPEL-Prozesses die zugehörige WSDL-Datei eingelesen wird. Daraus könnten die Nachrichtenkanäle extrahiert und eine Liste von Objektreferenzen auf Plätze generiert werden. Das würde allerdings einen erheblichen Mehraufwand bei der Implementierung aufwerfen, aber die Funktionalität des Programms nicht beeinflussen. Lediglich die Bezeichnung der Nachrichtenkanal-Plätze könnte um den Namen der Nachricht ergänzt werden. Deswegen wurde dieser Ansatz nicht weiter verfolgt.

### 4.4 Transformation der Aktivitäten

Die Transformation einer Aktivität wird von der Methode `drawActivity` ausgelöst. Ihr wird das zu transformierende `BPELActivity`-Objekt und ein Schnittstellenobjekt der Klasse `ActivityInterface` übergeben. Sie stellt fest um welche Art von Aktivität es sich handelt und ruft die entsprechende Methode zum Generieren des Musters auf. Der Methode `drawActivity` können über das Schnittstellenobjekt bereits erzeugte Schnittstellenplätze übergeben werden. Anderenfalls werden sie von der entsprechenden Methode direkt erzeugt und über das Schnittstellenobjekt der aufrufenden Methode übergeben.

Theoretisch wäre es auch möglich, auf die `drawActivity`-Methode zu verzichten. Dann müsste jedes Mal, wenn das Netz eines Aktivitätsmusters generiert werden soll, eine umfangreiche Fallunterscheidung gemacht werden, um zu ermitteln, um welchen Aktivitätstyp es sich handelt. Nur so kann die richtige Methode aufgerufen werden. Dieser Aufwand wird durch die `drawActivity`-Methode umgangen.

Der Ablauf zum Erzeugen eines Musters ist bei den meisten Aktivitäten sehr ähnlich und wird in Abschnitt 4.4.1 beschrieben. Auf einige Besonderheiten bestimmter Aktivitäten wird in den Abschnitten 4.4.2 bis 4.4.4 eingegangen. Die Transformation des `scope` als komplexestes Konstrukt der Semantik wird in Abschnitt 4.6 beschrieben.

#### 4.4.1 Grundsätzlicher Ablauf

Zum Erzeugen des Netzes einer Aktivität A werden mittels Methoden der Klasse `PetriNet` (siehe 4.2.2) dessen Plätze, Transitionen und Kanten in einem neuen Subnetz erzeugt.

Jedes Muster wird in einem eigenen Subnetz generiert, in dem der Name und der Typ des Musters gespeichert werden. So stehen diese Informationen bei der anschließenden Ausgabe des Netzes zur Verfügung.

Alternativ könnten Name und Typ auch in den Objekten der Knoten direkt gespeichert werden. Dies würde jedoch die Methoden-Aufrufe zum Erzeu-

gen der Plätze und Transitionen unnötig verkomplizieren und hätte erhebliche Redundanzen zur Folge. Da die Subnetze benötigt werden um die Einträge der Info-Datei nach Mustern zu sortieren, bieten sie sich auch zum Ablegen dieser Informationen an.

Das Muster einer Aktivität A läßt sich in variable und feste Bestandteile unterteilen. Die innere Struktur des Musters steht fest, während die Schnittstelle und die Objekte variabel sind und von den Elternmustern abhängen.

Zunächst müssen die Plätze der Standard-Schnittstelle erzeugt werden, wie in Abschnitt 4.1 beschrieben. Anschließend kann das Netz des zu erzeugenden Musters generiert sowie an die Schnittstelle und die global gespeicherten Plätze der Datenobjekte angebunden werden.

Soll das Muster einer strukturierten Aktivität erzeugt werden, so wird für jedes Kindmuster ein neues Schnittstellenobjekt (`activityInterface`) instanziiert. Muss ein Platz der äußeren Schnittstelle mit der Schnittstelle eines Kindmusters verschmolzen werden, so wird die Objektreferenz dieses Platzes in das neue Schnittstellenobjekt geschrieben. Fordert die Struktur des Musters der zu transformierenden Aktivität, dass ein Schnittstellenplatz von außen nach innen durchgereicht werden muss (z.B. `stop`, siehe 4.1), so wird die Objektreferenz dieses Platzes ebenfalls in das neue Schnittstellenobjekt geschrieben. Zum Erzeugen des Kindmusters wird anschließend die `drawActivity`-Methode aufgerufen, der das neue Schnittstellenobjekt übergeben wird.

### Besonderheiten bei `CorrelationSets`

Die Muster der Aktivitäten, in denen `CorrelationSets` verwendet werden, unterscheiden sich abhängig vom Wert des im `CorrelationSet` angegebenen Attributs `initiate`. Dieser Unterschied ist jedoch so gering, dass durch eine entsprechende Fallunterscheidung nur eine einzige Methode zum Generieren des Musters einer solchen Aktivität nötig ist.

Ist bei einem `CorrelationSet` `initiate = „no“` spezifiziert, so wird eine Anfangsmarkierung auf dem zugehörigen Platz erzeugt.

Werden mehrere `CorrelationSets` verwendet, so wird im generierten Petrinetz nur das erste in der Aktivität angegebene `CorrelationSet` berücksichtigt. Da beim synchronen `invoke` zwei `CorrelationSets` angebunden werden können, werden hier je nach Wert des Attributes `pattern` (siehe 2.1) jeweils nur das erste `CorrelationSet` für die eingehende und das erste `CorrelationSet` für die ausgehende Nachricht berücksichtigt.

#### 4.4.2 `invoke`

Ein synchrones `invoke` kann einen `Compensation`- oder `FaultHandler` beinhalten. Von der betrachteten Petrinetz-Semantik wird für diesen Fall jedoch kein eigenes Muster vorgesehen. Um ein solches `invoke` an seinen `Fault`- bzw. `CompensationHandler` anzubinden, wird es in einen `scope` eingebettet.

Die Methode `drawInvoke` stellt fest, welche der folgenden Ausprägungen des `invoke` vorliegt:

## 4 Implementierung

**Asynchrones invoke** Durch Aufruf der Methode `drawInvokeAsync` wird das entsprechende Petrinetz generiert.

**Synchrones invoke ohne Fault- oder CompensationHandler** Durch Aufruf der Methode `drawInvokeSync` wird das entsprechende Petrinetz erzeugt.

**Synchrones invoke mit Fault- oder CompensationHandler** In diesem Fall muss die `invoke`-Aktivität in einen `scope` eingebettet werden, wie in Abbildung 8 illustriert. Dazu wird ein neues `BPELScope`-Objekt erzeugt, in das die Objektreferenzen der im `invoke`-Objekt enthaltenen `Fault`- und `CompensationHandler` kopiert werden. Als innere Aktivität des `scope`-Objekts wird eine Kopie des zu generierenden `invoke` verwendet. Aus dieser Kopie müssen der `Fault`- und `CompensationHandler` gelöscht werden, da sie sonst erneut in einen `scope` eingebettet werden würde. So würde eine Endlosschleife entstehen. Anschließend wird das Muster des neu erzeugten `scope` durch die entsprechende Methode generiert.

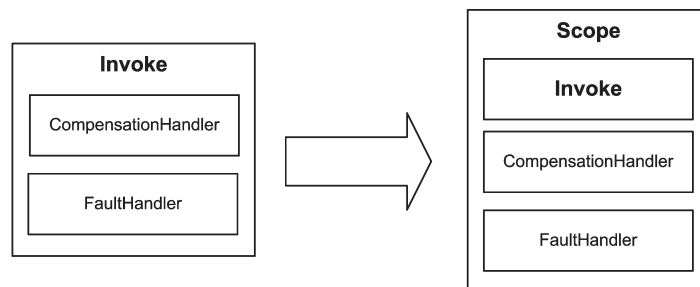


Abbildung 8: Einbettung der `invoke`-Aktivität in einen `scope`

### 4.4.3 assign

BPEL erlaubt in einer einzigen `assign`-Aktivität mehrere Wertzuweisungen. Das in der hier betrachteten Petrinetz-Semantik vorgeschlagene Muster der `assign`-Aktivität berücksichtigt jedoch nur eine einzige. Die Lösung dieses Problems besteht darin, ein `assign` mit mehreren Wertzuweisungen in eine Sequenz von `assigns` mit jeweils nur einer Wertzuweisung zu überführen. Zwei unterschiedliche Arten von Wertzuweisungen müssen betrachtet werden:

- Der Wert einer Variablen (`from-Variable`) wird einer anderen Variablen zugewiesen.
- Ein fester Wert wird einer Variablen zugewiesen.

Da beide Versionen unterschiedliche Muster haben, existiert jeweils eine eigene Methode, die das Petrinetz für ein `assign` mit entsprechender Wertzuweisung generiert.

Da nur eine Wertzuweisung pro `assign` im Muster vorgesehen ist, muss spezifiziert werden, welche der Wertzuweisungen bei der Transformation berücksichtigt werden soll. Dazu existiert im `BPELAssign`-Objekt die Variable `drawCopyIndex`. Sie enthält den Index derjenigen Wertzuweisung, für die das Muster generiert werden soll.

Ähnlich wie beim Erzeugen des `invoke`-Musters übernimmt auch hier eine allgemeine Methode `drawAssign` den Aufruf der spezielleren Methoden. Dabei werden folgende Fälle unterschieden:

**Nur eine Wertzuweisung vorhanden** `drawCopyIndex` des `BPELAssign`-Objekts erhält den Wert null und die Methode zum Generieren des entsprechenden Netzes wird aufgerufen (`drawAssignFrom` bzw. `drawAssignNoFrom`).

**Mehrere Wertzuweisungen vorhanden und `drawCopyIndex` ist nicht definiert**

Ein neues `BPELSequence`-Objekt wird erzeugt. Für jede Wertzuweisung der zu transformierenden `assign`-Aktivität wird eine Kopie des `BPELAssign`-Objekts als innere Aktivität der neuen `sequence` hinzugefügt. In jeder Kopie wird `drawCopyIndex` der Index der jeweils zu transformierenden Wertzuweisung zugewiesen, wie in Abbildung 9 illustriert. Anschließend wird die Methode zum Transformieren der neuen `sequence` aufgerufen.

**Mehrere `copy-Element` vorhanden und `drawCopyIndex` ist definiert** Abhängig von der Art der Wertzuweisung wird die Methode zum Generieren des entsprechenden Netzes aufgerufen.

Eine alternative Lösung besteht darin, auf `drawCopyIndex` zu verzichten. Stattdessen müssen in jeder Kopie des `BPELAssign`-Objekts bis auf die jeweils zu transformierende Wertzuweisung alle anderen entfernt werden. Da dieser Ansatz etwas umständlicher zu implementieren ist, wurde der oben beschriebene Ansatz gewählt. Eine weitere Möglichkeit wäre, gänzlich auf das Einbetten der Wertzuweisungen in ein `BPELSequence`-Objekt zu verzichten und direkt durch `drawAssign` das Muster einer `sequence` mit eingebetteten `assign`-Mustern zu generieren. Der damit verbundene Aufwand wäre allerdings unverhältnismäßig groß und würde keine Vorteile gegenüber den anderen Lösungen bringen.

### 4.4.4 `compensate`

`compensate` nimmt als Aktivität eine Sonderstellung ein. Es kann durch den BPEL-Prozess nur im `Compensation`- oder `FaultHandler` eingebettet sein (siehe 4.6.2 und 4.6.3). Zusätzlich wird das `compensate`-Muster im `FaultHandler` und im standardmäßigen `CompensationHandler` direkt eingebettet.

Vier verschiedene `compensate`-Muster werden unterschieden:

- `compensate` ist im `FaultHandler` eingebettet, kein zu kompensierender `scope` ist spezifiziert.
- `compensate` ist im `FaultHandler` eingebettet, ein zu kompensierender `scope` ist spezifiziert.

#### 4 Implementierung

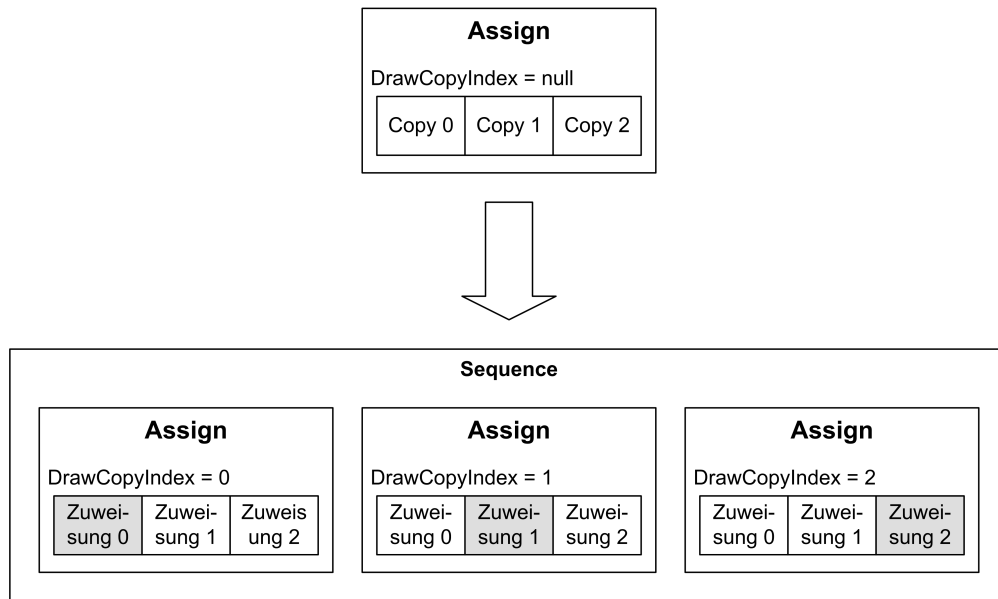


Abbildung 9: Überführung eines `assigns` mit mehreren Wertzuweisungen in eine `sequence`

- `compensate` ist im `CompensationHandler` eingebettet, kein zu kompensierender `scope` ist spezifiziert.
- `compensate` ist im `CompensationHandler` eingebettet, ein zu kompensierender `scope` ist spezifiziert.

Die Struktur dieser Muster ist jedoch immer gleich. Sie unterscheiden sich lediglich in High-Level-Aspekten und der Schnittstelle. Da von High-Level-Aspekten zunächst abstrahiert werden soll, sind hier nur die Unterschiede in den Schnittstellen relevant.

Im `compensate`-Muster wird die Standard-Schnittstelle um zwei Plätze erweitert. Die zusätzlichen Plätze werden je nach Art des Musters mit Plätzen aus dem `Compensation-` oder `FaultHandler` bzw. aus der `scope`-Schnittstelle des eigenen oder eines Kindscope verschmolzen. Näheres dazu ist in [Sta04] nachzulesen, hier soll nur die Umsetzung erläutert werden.

Die Methode `drawCompensate` erzeugt abhängig von den ihr übergebenen Parametern eines der oben aufgelisteten `compensate`-Muster. Da `compensate` eine erweiterte Schnittstelle hat, muss `drawCompensate` ein Objekt der erweiterten Schnittstellenklasse `CompensateInterface` übergeben werden.

Beim Aufruf von `drawCompensate` müssen zwei Fälle unterschieden werden:

- `compensate` ist direkt im `Fault-` oder standardmäßigen `Compensation-Handler` eingebettet. Die Methode zum Generieren des Musters des jeweiligen Handlers bestimmt, welche der vier `compensate`-Arten erzeugt werden soll und welche Schnittstellenplätze demzufolge genutzt werden müssen.

- **compensate** ist in der inneren Aktivität eines benutzerdefinierten **Fault-** oder **CompensationHandlers** eingebettet. Die strukturierte Aktivität in die **compensate** eingebettet werden muss, ist weder dazu ausgelegt, die zusätzlichen Plätze der **compensate**-Schnittstelle zu füllen, noch die Parameter über die Art des zu generierenden **compensate**-Musters zu übergeben.

Diese Funktionalität ließe sich zwar in die jeweiligen Methoden integrieren, was jedoch dazu führen würde, dass alle Methoden zum Erzeugen der Muster strukturierter Aktivitäten um die gleichen Elemente erweitert werden müssten. Die elegantere und weniger aufwendige Lösung besteht darin, eine neue Methode **drawCompensateActivity** einzuführen, die das Bindeglied zwischen **drawActivity** und **drawCompensate** darstellt.

Sie wird von **drawActivity** aufgerufen, wenn ein **compensate** generiert werden soll. Damit ein entsprechendes Schnittstellenobjekt für **drawCompensate** erzeugt und mit den entsprechenden Plätzen gefüllt werden kann, entnimmt sie dem **ScopeInterface**-Objekt (siehe 4.6) des aktuell in der Transformation befindlichen **scope** die Information darüber, in welchem Muster **compensate** eingebettet werden soll (siehe 4.6.2 und 4.6.3). Danach wird **drawCompensate** mit den Parametern zum Generieren des korrekten Musters aufgerufen.

Anschließend werden die Informationen über die Art des erzeugten **compensates** im aktuellen **ScopeInterface** vermerkt. Diese werden von der Methode zum Generieren des **CompensationHandlers** benötigt, um dessen Muster anzupassen.

### 4.5 Transformation der Link-Semantik

Wie in den Abschnitten 2.1 und 2.3 beschrieben, kann jede Aktivität, die in einen **flow** eingebettet ist, **source** oder **target** eines Links sein. Dazu wird das Muster der Aktivität in ein spezielles Link-Muster eingebettet, das das Netz der Aktivität an die Link-Plätze anbindet und bei Bedarf die Schnittstelle um den **negLink**-Platz zur Dead-Path-Elimination erweitert.

Bei der Umsetzung dieses Konzeptes haben sich drei Hauptaufgaben herauskristallisiert, die im Folgenden erläutert werden:

#### Erzeugen und Zwischenspeichern der Link-Plätze

Bevor ein Link an eine Aktivität angebunden werden kann, muss er in einem einbettenden **flow** (*Vaterflow*) deklariert worden sein. Beim Generieren des Petrinetzes einer **flow**-Aktivität werden für jeden in ihr deklarierten Link die beiden Plätze **Link** und **!Link** erzeugt. Die Link-Plätze müssen an folgende Muster angebunden werden:

- Muster der **source**- und **target**-Aktivität des Links
- **process**-Muster (Jeder **!Link**-Platz muss zu Prozessbeginn initialisiert werden.)

#### 4 Implementierung

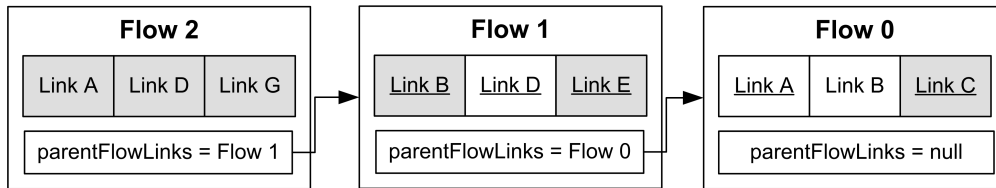


Abbildung 10: Einfach verkettete Liste der Link-Plätze in verschachtelten flows

- **scope-Muster** (Bei der Fehlerbehandlung müssen ausgehende Links des scope auf „false“ gesetzt werden.)

Damit eine Methode zum Erzeugen eines Musters Zugriff auf die benötigten Link-Plätze hat, müssen deren Objektreferenzen in geeigneten Datenstrukturen zwischengespeichert werden:

- Eine Aktivität kann **source** oder **target** jedes Links sein, der in einem einbettenden **flow** deklariert ist. Dabei kann die Aktivität beliebig tief im **flow** verschachtelt sein. Bettet ein **flow** einen weiteren **flow** (*Kindflow*) ein, so wird ein Link des Vaterflows durch einen eventuell vorhandenen gleichnamigen Link eines Kindflows überdeckt.

Dieses Konzept wird durch eine einfach verkettete Liste **flowLinks** umgesetzt, die global im **Transformer**-Objekt gespeichert wird. Beim Generieren des Musters eines **flow** wird dieser Liste ein neues Element angehängt. Es beinhaltet die Namen aller Links des **flow** mit den Objektreferenzen der zugehörigen Link-Plätze. Abbildung 10 zeigt das Beispiel einer solchen Liste für drei ineinander verschachtelte **flows** in denen jeweils drei Links deklariert sind. Dabei ist *Flow 2* Kindflow von *Flow 1* und *Flow 1* Kindflow von *Flow 0*.

Soll nun das Muster einer inneren Aktivität eines **flow** an einen Link angebunden werden, so wird von der Methode **getLinkNodes** die Liste nach einem Link mit entsprechendem Namen durchsucht. Im Beispiel sind in *Flow 2* jedoch nur die in der Abbildung grau markierten Links der Liste sichtbar. Um die Überdeckung der Links zu berücksichtigen muss beim Durchsuchen im obersten Element der Liste (im Beispiel: *Flow 2*) begonnen werden. Sobald der erste Link mit passendem Namen gefunden wurde, wird die Suche abgebrochen, die restlichen Elemente der Liste werden nicht weiter betrachtet. Wird **getLinkNodes** nicht fündig, so bricht BPEL2PN mit einer Fehlermeldung ab.

Wurde das Muster eines **flow** mit allen seinen Kindmustern vollständig generiert, so wird das Element mit den Link-Plätzen des entsprechenden **flow** aus der Liste entfernt, da diese nun nicht mehr benötigt werden. Im Beispiel wird das Element von *Flow 2* gelöscht. In *Flow 1* sind anschließend die in der Abbildung unterstrichenen Links sichtbar.

- Im Prozess-Muster müssen alle **!Link**-Plätze initialisiert werden. Dazu werden in der Methode zum Generieren des **flow**-Netzes die Objektrefe-

renzen aller !Link-Plätze zusätzlich in einer globalen Liste `allLinks` im `Transformer`-Objekt abgelegt. Die Liste `flowLinks` kann hier nicht genutzt werden, da in ihr keine Links gespeichert sind, deren `flow` bereits transformiert wurde.

- Gegebenenfalls müssen bei der Fehlerbehandlung im `scope` noch nicht definierte `source`-Links der inneren Aktivitäten auf „`false`“ gesetzt werden. Um die entsprechenden Link-Plätze an das Netz des `scope` anbinden zu können, wird eine Liste `innerLinks` der entsprechenden Link-Plätze geführt, die im `ScopeInterface` des jeweiligen `scope` gespeichert wird. Die Methode zum Erzeugen des Link-Musters fügt die betroffenen Links der Liste hinzu.

Als Datenstruktur wurde die Liste gewählt, da sie dynamisch um weitere Elemente erweitert werden kann und eine Ordnung der Elemente gewährleistet. Das Array (Feld) wäre ebenso denkbar, allerdings müsste hier manuell die richtige Dimensionierung sichergestellt werden. Beim Set (Menge) hingegen ginge die Ordnung der Elemente verloren. Das Set könnte deswegen nur für `innerLinks` und `flowLinks` verwendet werden, da hier die Ordnung keine Rolle spielt. Zur Vereinheitlichung wurde jedoch auch hier die Liste gewählt.

Es wäre denkbar, statt dieser drei Listen eine einzige zu führen. Allerdings müsste dann für jeden Eintrag zusätzlich die Information darüber gespeichert werden, welche Muster auf den jeweiligen Platz zugreifen können. Zusätzlich muss ein Konzept entwickelt werden, wie mittels dieser einen Liste die Überdeckung realisiert werden kann. Insgesamt erscheint diese Lösung deutlich komplizierter, deswegen wurde von ihr Abstand genommen.

### Dead Path Elimination mit `negLink`-Plätzen

Zur Dead Path Elimination (DPE) kann das Link-Muster die Standard-Schnittstelle um den `negLink`-Platz erweitern.

Steht fest, dass eine innere Aktivität einer `switch`- oder `pick`-Aktivität nicht abgearbeitet werden soll, wird eine Marke auf deren `negLink`-Platz gelegt. Dadurch werden alle ausgehenden Links dieser Aktivität auf „`false`“ gesetzt.

Der `negLink`-Platz wird genau dann im Link-Muster einer `source`-Aktivität erzeugt wenn diese (beliebig tief) in einer `switch`- oder `pick`-Aktivität verschachtelt ist. Wird durch eine strukturierte Aktivität ein Muster mit `negLink`-Platz eingebettet, so muss sie ihre Schnittstelle ebenfalls durch einen `negLink`-Platz erweitern und ihn entweder mit dem inneren `negLink`-Platz verschmelzen oder an ihr Netz anbinden.

Der `negLink`-Platz wird also über die Schnittstellen der einzelnen Muster von innen nach außen weitergereicht, bis die äußerste `switch`- oder `pick`-Aktivität erreicht ist. Dies wird dadurch realisiert, dass die Methoden, die die Netze der strukturierten Aktivitäten generieren, nach dem Erzeugen des Netzes einer inneren Aktivität überprüfen, ob ein `negLink`-Platz in deren Schnittstellenobjekt vorhanden ist. Wenn ja, wird es entsprechend an die äußere Schnittstelle angebunden.

## 4 Implementierung

Die Methoden zum Erzeugen der Muster der Aktivitäten müssen über die Information verfügen, ob die zu transformierende Aktivität in einem `switch` oder `pick` verschachtelt ist. Dazu können sie auf die im `Transformer`-Objekt globale Variable `countSwitchPick` zugreifen, in der die Schachtelungstiefe der `switch` und `pick`-Aktivitäten gespeichert wird. Der Wert von `countSwitchPick` wird von den Methoden `drawSwitch` und `drawPick` vor dem Generieren jeder inneren Aktivität inkrementiert und anschließend wieder dekrementiert. Im `EventHandler`, `CompensationHandler`, `FaultHandler` und in der `while`-Aktivität ist kein `negLink`-Platz in der Schnittstelle des Kindmusters vorgesehen. Um zu verhindern, dass dort ein `negLink`-Platz erzeugt wird, wird vor dem Aufruf der Methode des Kindmusters `countSwitchPick` auf null gesetzt. Anschließend wird `countSwitchPick` wieder der ursprüngliche Wert zugewiesen.

Als Alternative zur globalen `countSwitchPick`-Variable, könnte der Wert ähnlich der Schnittstellenobjekte als Parameter beim Methoden-Aufruf weitergereicht werden. Das würde jedoch die Methodenaufrufe unnötig verlängern und die Methoden, die sonst von der Dead-Path-Elimination nicht betroffen sind, verkomplizieren.

### Generieren der Link-Netze

Wie oben bereits erwähnt, wird das Muster einer Aktivität, die über ein- bzw. ausgehende Links verfügt, in ein Link-Muster eingebettet. Dies wird folgendermaßen umgesetzt:

Von der Methode `drawActivity` wird überprüft, ob die Aktivität `source` oder `target` eines Links ist. Ist dies der Fall, so wird der Methode `drawLinkPattern` die zu transformierende Aktivität übergeben. Obwohl in [Sta04] fünf verschiedene Link-Muster aufgeführt sind, gibt es nur diese eine Methode, die dynamisch das korrekte Muster generiert. Damit beim Transformieren der eingebetteten Aktivität diese nicht erneut in ein Link-Muster eingebettet wird (dies würde zu einer Endlosschleife führen), wird `drawActivity` mit dem optionalen Parameter `noLinkPattern` aufgerufen, der das unterbindet.

Das Erzeugen des Link-Musters hätte auch direkt in die Methode `drawActivity` integriert werden können, was sogar einen geringfügig niedrigeren Programmieraufwand bedeutet hätte. In Hinblick auf die Übersichtlichkeit des Quelltextes wurde jedoch eine eigene Methode dafür eingeführt.

Bei der Umsetzung des Link-Konzepts wird davon ausgegangen, dass alle der im zu transformierenden BPEL-Prozess definierten Links gemäß den in [ACLL03] aufgeführten Regeln verlaufen. Im anderen Fall ist das Verhalten von BPEL2PN nicht definiert. Eventuell wird ein fehlerhaftes Petrinetz erzeugt oder ein Programmabsturz provoziert.

## 4.6 Transformation von scope und process

Da der `process` ein Spezialfall des `scope` ist, werden die Netze beider Muster durch die Methode `drawScope` generiert. Auf die Besonderheiten des `process` als Spezialfall des `scope` wird in Abschnitt 4.6.1 eingegangen.

Der `scope` ist eine besondere strukturierte Aktivität. Neben einer inneren Aktivität, werden `Fault-` und `CompensationHandler` sowie optional ein `EventHandler` eingebettet. Die Transformation des `scope` unterscheidet sich in einigen Punkten von der anderer Aktivitäten:

- Schnittstellenplätze müssen über mehrere Einbettungsstufen hinweg verschmolzen werden.
- Im inneren Netz müssen unterschiedliche Arten von Mustern eingebettet werden.
- Die Reset-Kante im Stop-Muster muss berücksichtigt werden (siehe 4.2.2).

Nachfolgend werden diese Punkte näher erläutert.

### Besonderheiten der scope-Schnittstelle

Da das `scope`-Muster nicht nur an seine einbettende Aktivität angebunden ist, sondern zusätzlich auch an seinen Vaterscope und eventuell vorhandene Kindscores, muss es eine besondere Schnittstelle besitzen, die durch die Klasse `ScopeInterface` umgesetzt wird.

Die `ScopeInterface`-Klasse beinhaltet nicht nur die Plätze der `scope`-Schnittstelle. Zusätzlich werden die Plätze der im `scope` deklarierten Variablen, `CorrelationSets` und die `source`-Links der inneren Aktivitäten des `scope` dort gespeichert (siehe 4.5). Da die Methoden zum Generieren der Netze des `scope`, des Link-Musters und der `compensate`-Aktivität auf die Information zugreifen müssen, in welchem Kindmuster des `scope` (innere Aktivität, `Fault-` oder `CompensationHandler`) sie eingebettet sind, wird diese Information ebenfalls in Form von Statusvariablen im `ScopeInterface` abgelegt.

Eine weitere Besonderheit des `scope` im Gegensatz zu anderen strukturierten Aktivitäten besteht darin, dass einige seiner Schnittstellenplätze über mehrere Einbettungsstufen hinweg mit denen anderer Muster verschmolzen werden müssen. Das sind beispielsweise die Schnittstellen aller Kindscores und des Vaterscope. Ein weiteres Beispiel ist der `fault_in`-Platz des Stop-Musters. Er muss mit allen `failed`-Plätzen der inneren Aktivitäten verschmolzen werden. Die in den Abschnitten 4.1 und 4.4 beschriebene Methode kann also nicht angewendet werden, da ein Schnittstellenplatz nur an ein direkt eingebettetes bzw. einbettendes Muster weitergereicht werden kann.

Die Lösung besteht darin, global im `Transformer`-Objekt in der Variablen `currentScope` eine Objektreferenz auf die aktuelle `scope`-Schnittstelle abzulegen. So haben alle Methoden zum Erzeugen der Muster auf die entsprechenden Plätze Zugriff. Zusätzlich wird eine Liste aller `ScopeInterface`-Objekte geführt, deren Transformation noch nicht vollständig abgeschlossen ist. Dies ist nötig um einen Zugriff auf die Plätze der in Vaterscores deklarierten Variablen und `CorrelationSets` zu ermöglichen.

Bevor also von der Methode `drawScope` die innere Aktivität eines `scope` transformiert wird, muss dessen Schnittstellenobjekt mit den Objektreferenzen

## 4 Implementierung

der entsprechenden Plätze gefüllt und global verfügbar gemacht werden. Einige Plätze der `scope`-Schnittstelle werden nur benötigt, wenn in der inneren Aktivität des `scope` mindestens ein Kindscope vorhanden ist. Zusätzlich gibt es auch Plätze, deren Anzahl von der Anzahl der eingebetteten Kindscopes abhängt (siehe 4.6.3). Bevor die innere Aktivität transformiert wird, steht diese Information allerdings noch nicht zur Verfügung. Es wäre also naheliegend, die betroffenen Plätze erst danach zu erzeugen. Dann ist allerdings ein Anbinden dieser Plätze an die Schnittstellen der Kindscopes nicht mehr möglich, da deren Schnittstellenobjekte zu diesem Zeitpunkt nicht mehr in der globalen Liste der `ScopeInterface`-Objekte zur Verfügung stehen. Die Lösung besteht darin, dass `drawScope` vor Aufruf der Methode zur Transformation der inneren Aktivität zunächst die Schnittstelle des Vaterscope (sie steht in der globalen Variablen `currentScope` zur Verfügung) überprüft. Sind die betroffenen Plätze vorhanden, werden sie mit der eigenen Schnittstelle verschmolzen, wenn nicht, werden sie neu erzeugt und sowohl im `ScopeInterface`-Objekt des Vaterscope abgelegt als auch mit der eigenen Schnittstelle verschmolzen.

### Generieren des `scope`-Musters und Anbinden der Kindmuster

Nachdem die Schnittstelle des zu generierenden `scope`-Musters und die Plätze für Variablen und `CorrelationSets` erzeugt wurden, wird durch Aufruf von `drawActivity` das Netz der inneren Aktivität erzeugt. Als nächstes werden, falls vorhanden, durch Aufruf der entsprechenden Methoden die Netze der `EventHandler` generiert. Es folgen `FaultHandler`, `CompensationHandler` und `Stop`-Muster. Hier spielt die Reihenfolge eine wichtige Rolle:

- Das Netz des `Stop`-Musters kann erst generiert werden, nachdem das des `CompensationHandlers` erzeugt wurde, da im Falle des Prozesses der `compensated`-Platz des `CompensationHandlers` mit dem `final`-Platz des `Stop`-Musters verschmolzen wird (siehe 4.6.1). Der `compensated`-Platz kann allerdings erst beim Generieren des `CompensationHandler`-Musters erzeugt werden (siehe 4.6.3).
- Da je nach Art des `FaultHandlers` das Muster des `CompensationHandlers` angepasst werden muss (siehe 4.6.2 und 4.6.3), stehen erst, nachdem das Netz des `FaultHandlers` generiert wurde, die entsprechenden Informationen im `ScopeInterface`-Objekt des aktuellen `scope` zur Verfügung. So kann das Netz des `CompensationHandler` erst nach dem Netz des `FaultHandlers` generiert werden.
- Das Netz des `FaultHandlers` wiederum kann erst generiert werden, nachdem feststeht, ob in der inneren Aktivität des `scope` `source`-Links vorhanden sind oder nicht, da einige Plätze der Schnittstelle nur dann erzeugt werden, wenn dies der Fall ist (siehe 4.6.2).
- Bevor das Netz eines `Fault`- oder `CompensationHandlers` generiert werden kann, müssen die Netze aller Kindscopes erzeugt worden sein. Erst

dann sind alle Plätze in der Schnittstelle des aktuellen `scope` vorhanden, die nötig sind um alle eventuell vorhandenen `compensate Scope`-Aktivitäten anzubinden (siehe 4.6.3 und 4.4.4).

- Ob zuerst die innere Aktivität oder die `EventHandler` generiert werden, spielt keine Rolle. dass zuerst die innere Aktivität generiert wird, wurde willkürlich festgelegt.

Sind alle Kindmuster generiert, so werden die Plätze der `source`-Links des aktuellen `scope` und der inneren Aktivitäten an die Schnittstelle des `Fault-Handlers` angebunden.

### Berücksichtigung der Reset-Kante

An den `fault_in`-Platz des Stop-Musters ist eine Reset-Kante angebunden (siehe 4.2.2). Damit ein korrektes Petrinetz erzeugt werden kann, muss von `drawScope` festgelegt werden, wieviele Marken maximal auf diesem Platz liegen können.

Um eine obere Schranke für dieses Maximum zu ermitteln, wird mittels einer Zählvariablen im `ScopeInterface`-Objekt die Summe folgender Werte berechnet:

- Anzahl der `failed`-Plätze der Kindmuster des `CompensationHandlers`
- Anzahl der `failed`-Plätze der Kindmuster des `FaultHandlers`
- Anzahl der `failed`-Plätze der Kindmuster der inneren Aktivität
- Anzahl der `failed`-Plätze der Kindmuster des `EventHandlers`
- Anzahl der Kindscope (ein Kindscope kann ebenfalls eine Marke auf `fault_in` legen)
- Jeweils 1 für die `failed`-Plätze eines eventuell vorhandenen `Message-EventHandlers` und `CompensationHandlers`

Dabei werden nur die Muster berücksichtigt, die nicht in einem Kindscope des aktuellen `scope` eingebettet sind.

Die genaue maximale Anzahl der Marken, die auf dem `fault_in`-Platz liegen können, kann nur durch statische Analyse ermittelt werden. Das soll Gegenstand weiterer Arbeit sein.

Im Gegensatz zum in [Sta04] vorgestellten `scope`-Muster wurden für die vorliegende Arbeit einige Vereinfachungen vorgenommen, die sich auch auf die Schnittstelle des `scope`-Musters auswirken. Sie werden in Abschnitt 4.6.3 beschrieben.

### 4.6.1 Besondere scope-Muster

Wie schon erwähnt, ist der `process` ein Spezialfall des `scope`. Da er sich nur in einigen wenigen Aspekten vom `scope` unterscheidet, kann das Netz seines Musters ebenfalls von `drawScope` generiert werden. Ein BPEL-Prozess enthält genau einen `process`, in dem alle anderen Muster eingebettet sind.

Die Transformation eines BPEL-Prozesses wird dadurch gestartet, dass die `drawScope`-Methode mit einem speziellen Parameter aufgerufen wird. So verfügt sie über die Information, dass es sich um einen `process` handelt, und kann ein entsprechend angepasstes Netz generieren. Da die Muster von `Compensation`- und `FaultHandler` sowie das `Stop`-Muster ebenfalls an das Prozess-Muster angepasst werden müssen, werden die Methoden zum Generieren der Netze dieser Muster mit einem entsprechenden Parameter aufgerufen.

Die genauen Unterschiede der `scope`- und `process`-Muster werden in [Sta04] genau erklärt.

Im `process`-Muster wurde für die vorliegende Arbeit eine Vereinfachung vorgenommen.

In BPEL ist es möglich, eine abgearbeitete Prozessinstanz zu kompensieren. Dazu muss im Prozess das Attribut `enableInstanceCompensation` den Wert „yes“ haben.

In BPEL2PN wird jedoch für `enableInstanceCompensation` immer der Wert „no“ angenommen. Damit entfallen in der Schnittstelle des `process`-Musters die Plätze `ch_in` und `compensated`. Das hat zur Folge, dass in der Schnittstelle des `CompensationHandler`s der `ch_in`-Platz ebenfalls entfällt und der `compensated`-Platz mit dem `final`-Platz der `process`-Schnittstelle verschmolzen wird.

Für den Fall, dass der `process` keine Kindscopes enthält, und kein nutzerdefinierter `CompensationHandler` definiert ist, kann auf den `CompensationHandler` verzichtet werden, da er keine Funktion mehr hätte. Statt dessen werden die `ch_fh`- und `ch_out`-Plätze des `FaultHandler`s sowie die `clean`- und `cleaned`-Plätze des `Stop`-Musters jeweils miteinander verschmolzen.

In der vorliegenden Arbeit wird die in [Sta04] vorgestellte Petrinetz-Semantik noch um folgenden Aspekt erweitert:

Wird ein `scope` in den `Fault`- oder `CompensationHandler` seines Vaterscope eingebunden, so wird er nicht an dessen `CompensationHandler` angebunden. Aus diesem Grund werden die Plätze `ch_in`, `compensated` sowie `!push` und `push` (untere Schnittstelle) nicht benötigt und dementsprechend auch nicht in der Schnittstelle des `scope` erzeugt. Der `CompensationHandler` wird dabei genauso behandelt wie der `CompensationHandler` des `scope`-Musters.

Um einen solchen `scope` in der LoLA- und der info-Datei (siehe 3.2) kenntlich zu machen, lautet seine Bezeichnung nicht `Scope`, sondern `Scope_FH` bzw. `Scope_CH`.

Anhand der im aktuellen `ScopeInterface`-Objekt abgelegten Statusinformationen, erkennt `drawScope`, ob der zu transformierende `scope` in einen `Compensation`- oder `FaultHandler` eingebettet ist und kann so das korrekte Muster generieren.

### 4.6.2 Transformation des FaultHandlers

Die Petrinetze der in [Sta04] aufgeführten verschiedenen Muster für den `FaultHandler` werden von der Methode `drawFaultHandler` dynamisch erzeugt. Dabei werden folgende Punkte berücksichtigt:

- Die Art des `scope`, in den der `FaultHandler` eingebettet ist (siehe 4.6)
- Standardmäßiger oder nutzerdefinierter `FaultHandler`
- Im `scope` oder in dessen innerer Aktivität eingebettete `source`-Links
- Eine eventuell vorhandene `catchall`-Aktivität (Aktivität die ausgeführt werden soll, wenn keine der anderen Aktivitäten den Fehler behandeln kann)

Die dazu benötigten Informationen, die nicht in der internen Repräsentation des `FaultHandlers` vorhanden sind, werden `drawFaultHandler` beim Aufruf als Parameter übergeben.

Eine eventuell in der inneren Aktivität des nutzerdefinierten `FaultHandlers` verschachtelte `compensate`-Aktivität muss auf die Plätze `ch_fh` und `ch_out` der Schnittstelle des `FaultHandlers` zugreifen können (siehe 4.4.4). Um dies zu ermöglichen, werden Objektreferenzen dieser beiden Plätze vor dem Generieren der inneren Aktivität ins `ScopeInterface`-Objekt des aktuellen `scope` geschrieben.

Unter bestimmten Bedingungen wird kein `CompensationHandler` im `scope` generiert (siehe 4.6.1). Ist dies der Fall, so müssen die beiden Plätze `ch_in` und `ch_fh` verschmolzen werden. Diese Aufgabe übernimmt die `drawFaultHandler`-Methode beim Füllen ihres Schnittstellenobjekts.

### 4.6.3 Transformation des CompensationHandlers

Das in [Sta04] vorgestellte Muster des `CompensationHandlers` verfügt über einen als High-Level-Petrinetz implementierten Stapelspeicher (Stack), in dem die fehlerfrei abgearbeiteten Kindscopes gespeichert werden. Er wird benötigt, um die `CompensationHandler` aller Kindscopes in der umgekehrten Reihenfolge ihrer Abarbeitung aufzurufen.

In der vorliegenden Arbeit soll allerdings nur ein Low-Level-Petrinetz generiert werden (Ausnahme: `fault_in`-Platz des Stop-Musters, siehe 4.6.1 und 4.2.2). Eine Implementierung des Stacks als Low-Level-Netz wäre sehr aufwendig und würde zu einer Zustandsraumexplosion führen, die eine Analyse des generierten Netzes stark erschweren würde. Eine weitere Ausnahme wäre hier denkbar, so könnte der Stack als High-Level-Netz implementiert werden. Da ein High-Level-Netz durch LoLA allerdings zu einem Low-Level-Netz entfaltet wird, bliebe die Zustandsraumexplosion jedoch erhalten.

Aus diesem Grund wird im Rahmen der vorliegenden Arbeit der Stack aus dem Muster des `CompensationHandlers` entfernt und durch ein Alternativmuster ersetzt, wie in Abbildung 11 illustriert.

Durch den fehlenden Stack können nun keine Namen von Kindscopes mehr im `CompensationHandler` gespeichert werden. Trotzdem muss eine Anbindung an

#### 4 Implementierung

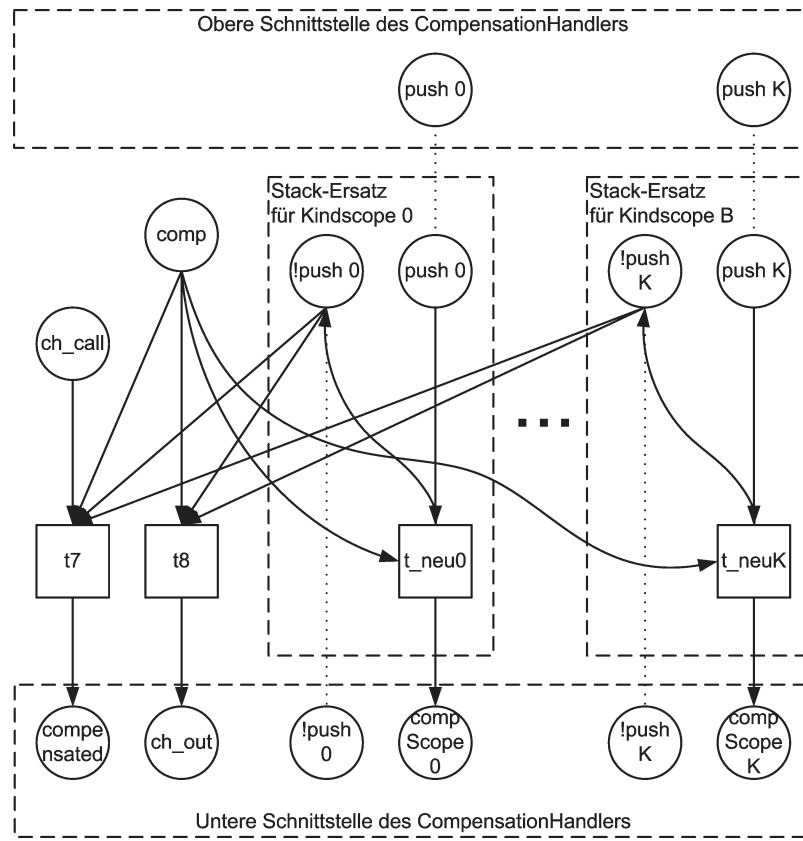


Abbildung 11: Stack-Alternative im CompensationHandler für K-1 Kindsopes

die CompensationHandler der Kindsopes gewährleistet sein. Existierte vorher jeweils nur einziger `push`- und `compScope`-Platz um den CompensationHandler an alle Kindsopes anzubinden, ist nun für jeden vorhandenen Kindscope ein eigener `push`- und `compScope`-Platz in der Schnittstelle des CompensationHandlers vorgesehen. Zusätzlich kommt ein `!push`-Platz für jeden Kindscope hinzu, der den Komplementärplatz zum entsprechenden `push`-Platz darstellt. Durch die Transition `t2` im Prozess-Muster ([Sta04], Abb. 24) muss der `!push`-Platz initialisiert werden. Dementsprechend muss die Schnittstelle des den CompensationHandler einbettenden scope um die jeweiligen Plätze erweitert werden. Statt mittels des Stacks werden die CompensationHandler der Kindsopes nun durch die Transitionen `t_neu0` - `t_neuK` aufgerufen.

Da diese Transitionen nichtdeterministisch schalten, hat diese Vereinfachung zur Folge, dass die CompensationHandler der Kindsopes nun nicht mehr in einer definierten Reihenfolge aufgerufen werden.

Obwohl in [Sta04] verschiedene Muster für den CompensationHandler aufgeführt sind, wird das Netz des CompensationHandlers durch eine einzige Methode (`drawCompensationHandler`) dynamisch generiert. Anhand der im aktuellen `scopeInterface` gespeicherten Statusinformationen erkennt `drawCompensationHandler`, um welche Art von CompensationHandler es sich handelt und setzt das Netz aus den entsprechenden Bausteinen zusammen.

## 5 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde in Java ein Werkzeug (BPEL2PN) entwickelt, das einen in BPEL spezifizierten verteilten Geschäftsprozess in ein Petrinetz überführen kann. Zur Übersetzung wurde die in [Sta04] vorgestellte Petrinetz-Semantik für BPEL verwendet. Um eine Kontrolle und Analyse des generierten Netzes zu ermöglichen, werden verschiedene Ausgabeformate unterstützt, unter anderem ein Format das vom Model-Checker LoLA gelesen werden kann.

Während der Implementation wurden einige Annahmen getroffen und Vereinfachungen vorgenommen. Insbesondere wurde von Daten abstrahiert. In den folgenden Abschnitten sollen noch einmal alle Einschränkungen aufgeführt sowie ein Ausblick auf mögliche weitere Arbeit gegeben werden.

### 5.1 Einschränkungen

Folgende Einschränkungen sind noch in BPEL2PN vorhanden:

- Der zu transformierende BPEL-Prozess muss korrekt sein, das heißt, er muss der BPEL-Spezifikation [ACLL03] entsprechen. Einige grobe Fehler können von BPEL2PN abgefangen werden; das Programm gibt dann eine Fehlermeldung aus und wird beendet. Andere Fehler jedoch können das Erzeugen von inkorrekten Petrinetzen oder sogar Abstürze provozieren.
- Die in [Sta04] vorgestellte Petrinetz-Semantik berücksichtigt keine Serialisierbarkeit von `scopes` [ACLL03]. Entsprechend wurde dieser Aspekt in BPEL2PN auch nicht implementiert.
- Das Prozessattribut `enableInstanceCompensation` wird nicht ausgewertet und immer als „no“ angenommen (siehe 4.6.1). Die Einschränkung läßt sich jedoch umgehen, indem der zu analysierende Prozess als `scope` definiert und in einen `process` eingebettet wird. So kann der umgebende `process` den `CompensationHandler` des `scope` auslösen.
- Durch die vorgenommene Abstraktion von Daten können keine Daten in Variablen geschrieben oder von ihnen gelesen werden. Dadurch werden auch keine Aktivierungsbedingungen von Transitionen ausgewertet, so dass keine definierten Fehler geworfen werden können. Es wird lediglich symbolisch auf die Low-Level-Plätze der Variablen zugegriffen.
- Um eine Zustandsraumexplosion zu vermeiden, wurde der Stack des `CompensationHandlers` durch ein Alternativkonstrukt ausgetauscht. Dadurch können die `CompensationHandler` der `Kindsscopes` nicht mehr in der vorgegebenen Reihenfolge aufgerufen werden. Sie ist nun nichtdeterministisch (siehe 4.6.3). Da durch das Model Checking jedoch alle Möglichkeiten getestet werden, stellt dies keine relevante Einschränkung dar.
- Bei allen Aktivitäten, die `CorrelationSets` verwenden, wird nur das erste in der jeweiligen Aktivität spezifizierte `CorrelationSet` berücksichtigt. Eine Ausnahme bildet das synchrone `invoke`, hier werden bis zu zwei `CorrelationSets` angebunden (siehe 4.4.1)

- Ein `CorrelationSet`, das durch eine Aktivität mit dem Anfangsattribut `initiate = „no“` verwendet wird, wird mit einer Anfangsmarkierung versehen. Dabei wird nicht berücksichtigt, ob es von einer anderen Aktivität mit `initiate = „yes“` verwendet wird. Dieses Problem kann durch statische Analyse gelöst werden.
- Die maximale Anzahl der auf dem `fault_in`-Platz des Stop-Musters liegenden Marken wird nur durch eine obere Schranke approximiert. Durch statische Analyse kann sie genau bestimmt werden.

### 5.2 Weitere Arbeit

Die vorliegende Arbeit bietet viel Raum für weitergehende, auf ihr aufbauende Arbeiten. Im Folgenden werden einige Vorschläge gemacht:

**Integration des Datenaspektes** Bis auf die in den Abschnitten 4.6 und 4.2.2 beschriebene Ausnahme, erzeugt BPEL2PN bisher nur Low-Level-Netze, die den Datenaspekt von BPEL vernachlässigen. Da aber gerade Daten einen erheblichen Einfluss auf den Prozess haben, ist es wünschenswert, sie auch bei der Transformation des Prozesses in ein Petrinetz zu berücksichtigen, wie in [Sta04] beschrieben. Da bei der Entwicklung von BPEL2PN darauf geachtet wurde, dass das Werkzeug erweiterbar ist, wurde der Datenaspekt implizit berücksichtigt und könnte durch folgende Erweiterungen integriert werden:

Die Klasse `Petrinet` (siehe 4.2.2) muss dahingehend erweitert werden, dass sie auch High-Level-Netze erzeugen kann. Dazu müssen neben den Subklassen zum Speichern von Plätzen, Transitionen und Kanten zusätzliche Subklassen eingeführt werden, in denen Sorten und Funktionen gespeichert werden können. Neben den zusätzlichen Methoden zum Erzeugen von Objekten der neuen Klassen, müssen bestehende Methoden und Klassen erweitert werden:

- Die Subklasse `Place` muss die Sorte des Platzes speichern können.
- Die Subklasse `Transition` muss eine Aktivierungsbedingung (Ausdruck, der zu „wahr“ ausgewertet werden muss, damit die Transition schalten kann) speichern können.
- Die Subklasse `Arc` muss eine Funktion, eine Variable und eine Sorte speichern können.
- Die Methoden zum Erzeugen von Objekten dieser drei Klassen (`addPlace`, `addTransition`, `addArc`) müssen um einen optionalen Parameter erweitert werden, damit sie beim Erzeugen der entsprechenden Objekte die zusätzlichen Informationen (Sorte, Aktivierungsbedingung, Variable, Funktion) berücksichtigen können.
- Die `writeLola`-Methoden der `PetriNet`-Klasse sowie der `Place`-, `Transition`- und `Arc`-Subklassen müssen so angepasst werden, dass sie beim Erzeugen der Ausgabe im LoLA-Format die High-Level-Informationen mit ausgeben.

In der Klasse `Transformer` müssen alle Methoden zum Generieren der Petri-netze erweitert werden, deren Muster High-Level-Plätze enthalten. Dazu müssen über die entsprechenden Methoden der Klasse `PetriNet` zunächst Sorten und Funktionen definiert werden. Anschließend können dann den Methoden zum Erzeugen der Plätze, Transitionen und Kanten die High-Level-Informationen als Parameter übergeben werden.

**Statische Analyse** Unter statischer Analyse versteht man die Analyse von Programmen zur Compilezeit. So können allein aus dem Quelltext Informationen über das Programm ermittelt werden. Einen ersten Ansatz zur statischen Analyse von BPEL-Prozessen liefert [Hei03].

Zwei Probleme, die durch statische Analyse gelöst werden könnten, wurden im letzten Kapitel bereits angesprochen:

- Die maximale Anzahl von Marken, die auf dem `fault_in`-Platz des Stop-Musters liegen können, wird in der vorliegenden Arbeit lediglich durch eine obere Schranke approximiert (siehe 4.6). Durch statische Analyse kann ermittelt werden, wieviele Aktivitäten gleichzeitig einen Fehler werfen können, so daß das Maximum genau bestimmt werden kann.
- Wird ein `CorrelationSet` von einer Aktivität mit `initiate = „no“` verwendet, so wird es auch dann mit einer Anfangsmarkierung versehen, wenn eine andere Aktivität es mit `initiate = „yes“` verwendet (siehe 4.4.1). Durch statische Analyse des Prozesses kann ermittelt werden, welche der Aktivitäten zuerst abgearbeitet wird, so daß gegebenenfalls auf diese Anfangsmarkierung verzichtet werden kann.

Eine weitere mögliche Anwendung der statischen Analyse ist die Vereinfachung des resultierenden Netzes, um die Komplexität der anschließenden Analyse zu reduzieren.

Da die in [Sta04] vorgestellten Muster einen allgemeinen Fall modellieren, werden unter Umständen nicht alle Elemente eines Musters in jedem Prozess benötigt. Durch statische Analyse könnten unter anderem Informationen über den Kontrollfluß oder uninitialisierte Variable ermittelt werden. So können speziell an den jeweiligen Prozess angepasste Muster generiert werden, von denen nur die Fälle berücksichtigt werden, die auch tatsächlich eintreten können. Das Ziel ist es, für den jeweiligen Kontext das abstrakteste Muster zu verwenden.



## Literatur

- [ACLL03] Tony Andrews, Francisco Curbera, Frank Leyman, and Kevin Liu. Business Process Execution Language for Web Services, Version 1.1. Technical report, IBM, Microsoft, 2003.
- [BPCM<sup>+</sup>04] Tim Bray, Jean Paoli, C.M.Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). *W3C Recommendation*, 2004.
- [Bur97] Rainer Burkhardt. *UML - Unified Modeling Language*. Addison-Wesley, 1 edition, 1997.
- [CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (wsdl) 1.1. *W3C Note*, 2001.
- [DFS98] C. Dufourd, A. Finkel, and Ph. Schnoebelen. Reset Nets between Decidability and Undecidability. In K. Spies and B. Schätz, editors, *Proc. 25th Int. Coll. Automata, Languages, and Programming (ICALP'98), Aalborg, Denmark, July 1998*, Lecture Notes in Computer Science 1443, pages 103–115. Springer, 1998.
- [FBS04] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting BPEL web services. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 621–630. ACM Press, 2004.
- [Fer04] Andrea Ferrara. *Web Services: A Process Algebra Approach*. Universita die Roma, <http://www.dis.uniroma1.it/ferrara/research/ferraraPA04.pdf>, 2004.
- [FGV04] Roozbeh Farahbod, Uwe Glässer, and Mona Vajihollahi. Specification and Validation of the Business Process Execution Language for Web Services. In *Abstract State Machines*, volume 3052 of *Lecture Notes in Computer Science*, pages 78–94. Springer, 2004.
- [FR04] Dirk Fahland and Wolfgang Reisig. ASM-based semantics for BPEL: The negative Control Flow. Technical Report 179, Humboldt-Universität zu Berlin, December 2004.
- [GKN02] Emden Gansner, Eleftherios Koutsoufios, and Stephen North. *Drawing graphs with dot*. AT&T, [www.graphviz.org](http://www.graphviz.org), 2002.
- [Har02] Elliotte Rusty Harold. *Processing XML with Java*. Addison-Wesley, <http://www.cafeconleche.org/books/xmljava/>, 2002.
- [Hei03] Thomas Heidinger. Statische Analyse von BPEL4WS prozessmodellen. Studienarbeit, Humboldt-Universität zu Berlin, Institut für Informatik, 2003.

- [HM04] Jason Hunter and Brett McLaughlin. *JDOM v1.0 API Specification*. <http://www.jdom.org/docs/apidocs/>, 2004.
- [Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.
- [Sch00] Karsten Schmidt. Lola: A low level analyser. In M. Nielsen and D. Simpson, editors, *Lecture Notes in Computer Science 1825*, volume Proc. 21th Int. Conf. Application and Theory of Petri Nets, Aarhus, Denmark, pages 465–474. Springer, 2000.
- [Sta04] Christian Stahl. Transformation von BPEL4WS in Petrinetze. Diplomarbeit, Humboldt-Universität zu Berlin, Institut für Informatik, 2004.
- [vdA98] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
- [WAB<sup>+</sup>98] Lauren Wood, Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Arnaud Le Hors, and Gavin Nicol. Document Object Model (DOM) Level 1 Specification Version 1.0. *W3C Recommendation*, 1998.
- [Web03] Michael Weber. *Allgemeine Konzepte zur software-technischen Unterstützung verschiedener Petrinetz-Typen*. PhD thesis, Humboldt-Universität zu Berlin, 2003.

## **Erklärungen**

### **Selbständigkeitserklärung**

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Berlin, den 18. März 2005

Sebastian Hinz

### **Einverständniserklärung**

Ich erkläre hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Instituts für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Berlin, den 18. März 2005

Sebastian Hinz