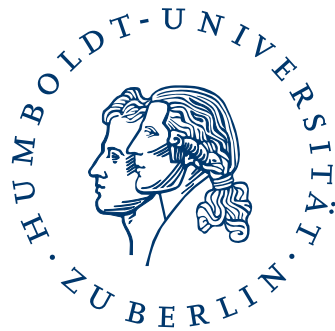


Diplomarbeit

**Transformation von offenen Workflow-Netzen zu
abstrakten WS-BPEL-Prozessen**

Jens Kleine

4. Juli 2007



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

Gutachter:
Prof. Dr. Wolfgang Reisig
Prof. Dr. Karsten Wolf

Zusammenfassung

In dieser Arbeit präsentieren wir eine Transformation von offenen Workflow-Netzen zu abstrakten WS-BPEL-Prozessen. Als Web Services implementierte Geschäftsprozesse sind zunehmend verbreiteter und von größerer finanzieller Bedeutung. Daher müssen sie vor ihrem Einsatz auf wichtige Eigenschaften wie korrekte Terminierung oder Bedienbarkeit überprüft werden. Die häufig eingesetzte Modellierungssprache WS-BPEL ist jedoch auf Grund ihrer fehlenden formalen Semantik nicht analysierbar. Aus diesem Grund existieren Werkzeuge zur Überführung von WS-BPEL-Prozessen in die Petrinetzklasse der offenen Workflow-Netze. Für diese kann auf eine Reihe von Tools zur formalen Analyse von Geschäftsprozessen zurückgegriffen werden. Unsere Transformation ermöglicht es, die Ergebnisse dieser Analysewerkzeuge vollautomatisch in WS-BPEL-Prozesse zurück zu übersetzen. So können unter anderem Beispiele für im Prozess auftretende Fehler und bedienende Partner als WS-BPEL-Code an den Anwender zurückgegeben werden, ohne dass dieser sich dazu mit den, in den Zwischenschritten der Analyse verwendeten, Petri-Netzen auskennen muss. Zudem bietet unsere Transformation die Möglichkeit Geschäftsprozesse graphbasiert als offene Workflow-Netze zu modellieren und diese anschließend automatisch in einen abstrakten WS-BPEL-Prozess zu übersetzen.

Inhaltsverzeichnis

| | |
|---|------------|
| 1. Einleitung | 6 |
| 1.1. Motivation und Zielsetzung dieser Arbeit | 6 |
| 1.2. Aufbau der Arbeit | 8 |
| 2. Grundlagen | 9 |
| 2.1. Petrinetze und offene Workflow-Netze | 9 |
| 2.2. WS-BPEL | 16 |
| 3. Transformation | 20 |
| 3.1. Idee der Transformation | 20 |
| 3.2. Transformationen des oWFN | 23 |
| 3.3. Transformationen des WS-BPEL-Codes | 69 |
| 3.4. Ausführungsreihenfolge der Transformationen | 73 |
| 4. Implementierung und Fallstudie | 76 |
| 4.1. Implementierung | 76 |
| 4.2. Fallstudie | 79 |
| 5. Zusammenfassung und Ausblick | 87 |
| 5.1. Ergebnisse | 87 |
| 5.2. Alternative Ansätze | 88 |
| 5.3. Weitere Arbeit | 88 |
| A. Anhänge | 91 |
| A.1. Erhaltung der Soundness-Eigenschaft durch die Transformationen | 91 |
| A.2. Strukturelle Umformungen | 93 |
| A.3. Nebenläufige Ausführungen | 96 |
| A.4. Fallstudie | 97 |
| A.5. Vergleich mit einem alternativen Ansatz | 98 |
| Abbildungsverzeichnis | 101 |
| Literaturverzeichnis | 102 |
| Erklärung | 104 |

1. Einleitung

1.1. Motivation und Zielsetzung dieser Arbeit

Geschäftsprozesse beschreiben organisatorische Vorgänge, die aus einzelnen Aktivitäten zusammengesetzt sind. Kommunizieren mehrere räumlich oder organisatorisch getrennte Geschäftsprozesse über Nachrichten miteinander, wird von verteilten Geschäftsprozessen gesprochen. Diese werden immer häufiger in Form von Web Services implementiert. Web Services werden üblicherweise nicht in Isolation ausgeführt. Sie werden dazu entworfen, um von anderen Web Service aufgerufen zu werden und ihrerseits Services aufzurufen. Ihre Anzahl und Verbreitung nimmt stetig zu. Auf Grund ihrer großen Bedeutung werden besondere Anforderungen an sie gestellt. Sie sollen zuvor bestimmte Eigenschaften erfüllen und unerwünschtes Verhalten ausschließen. Zu den erwünschten Eigenschaften gehören in den meisten Fällen vor allem die Sicherstellung ihrer korrekten Terminierung (zum Beispiel Soundness [Aal98]) bei der Interaktion mehrerer Web Services und die Gewährleistung ihrer Bedienbarkeit [Sch05]. Diese Eigenschaften müssen überprüft werden, *bevor* die Web Services zum Einsatz kommen. Aktuelle Geschäftsprozessmodellierungssprachen, wie die weit verbreitete *Web Services Business Process Execution Language* [AAA⁺07] oder kurz *WS-BPEL*, sind jedoch gar nicht oder nur im geringen Maße analysierbar.

Petrinetze [Rei82] auf der anderen Seite sind ein Formalismus, für den eine Vielzahl von Analysewerkzeugen bereit steht. Zur Modellierung von kommunizierenden Geschäftsprozessen wurden *offene Workflow-Netze* [MRS05] oder kurz *oWFN* als eine spezielle Netzklasse der Petrinetze eingeführt. Es existiert eine Toolchain [LMSW06] zur Analyse von WS-BPEL-Prozessen. Diese verwendet Werkzeuge um WS-BPEL-Prozesse¹ in offene Workflow-Netze zu überführen. Diese können dann mit bestehenden Tools analysiert werden. Die dabei entstehenden Ausgaben, wie Gegenbeispiele, die Fehler im Prozess demonstrieren oder bedienende Partner eines Prozesses, liegen wieder in Form von offenen Workflow-Netzen vor. Die Toolchain ist bereits in der Lage, automatisiert nicht-triviale Fehler in WS-BPEL-Prozessen aufzuspüren, die bei einer manuellen Suche nur schwer zu entdecken wären.

Abbildung 1.1 zeigt diese Toolchain, die einen WS-BPEL-Prozess in ein offenes Workflow-Netz überführt, welches dann von verschiedenen Analysewerkzeugen, wie Fiona und LoLA untersucht wird. Bisher ist ein Werkzeug (BPEL2oWFN) vorhanden, das einen WS-BPEL-Prozess in ein offenes Workflow-Netz übersetzen und zwei (LoLA, Fiona) zur Analyse dieser Netze. Mit dieser Arbeit wollen wir die in der Abbildung blass dargestellte Lücke schließen und eine Transformation präsentieren, die offene Workflow-Netze in WS-BPEL-Prozesse überführt. Unser Ziel ist es dabei, alle bedienenden Partner des Pro-

¹Wenn im Folgenden von einem WS-BPEL-Prozess gesprochen wird, meinen wir einen Geschäftsprozess, der in WS-BPEL beschrieben ist.

zesses zu erhalten, eine der Struktur des offenen Workflow-Netzes ähnliche und für einen menschlichen Nutzer verständliche Entsprechung des Netzes als WS-BPEL-Code zu erzeugen und diese Transformation im Gegensatz zu bestehenden Werkzeugen (vergleiche [LA06]) vollautomatisch durchführen zu können. Dies vervollständigt die automatisierte Analyse von WS-BPEL-Prozessen und ermöglicht auch Anwendern ohne Petrinetz-Kenntnisse die Nutzung, da für diese ein Resultat im ursprünglichen Eingangsformat erzeugt wird.

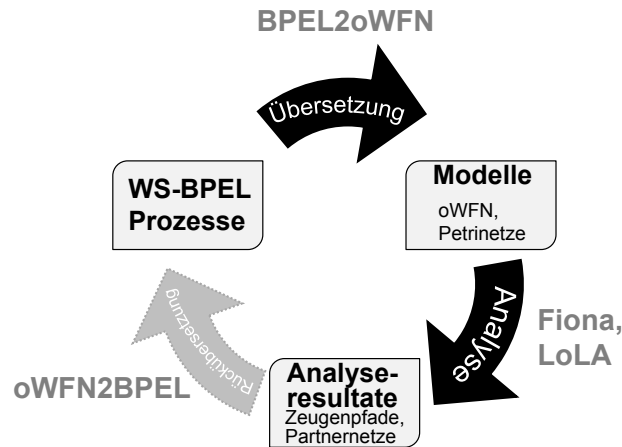


Abbildung 1.1.: Die Toolchain zur Analyse und Optimierung von WS-BPEL-Prozessen.

Da die Modellierung von Geschäftsprozessen hauptsächlich in Form von Graphen stattfindet und nicht direkt als Quellcode, erfüllt die in dieser Arbeit vorgestellte Transformation einen weiteren Zweck. Während WS-BPEL als XML-Quellcode mit vielen komplexen Konstrukten und Einschränkungen schwierig zu modellieren ist, sind Petrinetze eine intuitive, graphbasierte Alternative. Die Modellierung mit Hilfe von Petrinetzen ist in der Regel einfacher, da sie abstrakter ist und weniger Details bekannt sein und festgelegt werden müssen. Mit Hilfe unserer Transformation können Geschäftsprozesse als offene Workflow-Netze modelliert werden, ohne sich dabei der Restriktionen von WS-BPEL bewusst sein zu müssen, und anschließend in einen abstrakten WS-BPEL-Prozess transformiert werden. In diesem brauchen anschließend nur noch Details wie Kommunikationspartner und Attribute der WS-BPEL-Aktivitäten angegeben zu werden, um einen ausführbaren WS-BPEL-Prozess zu erhalten. Die dem der Transformation zugrunde liegenden oWFN ähnliche Struktur des erzeugten WS-BPEL-Codes unterstützt dabei die leichte Orientierung bei der Fertigstellung und Wartung des WS-BPEL-Prozesses.

Unsere Transformation erfolgt dabei in zwei Phasen: In der ersten ersetzen wir schrittweise zuvor definierte Muster im oWFN durch kleinere und reduzieren das Netz so nach und nach auf einen einzelnen Knoten. Bei diesen Reduktionen fügen wir Annotationen zur Repräsentation der entfernten Netzteile in das oWFN ein. Diese Annotationen werden dann in der zweiten Phase optimiert und in einen WS-BPEL-Prozess umgewandelt.

1.2. Aufbau der Arbeit

Wir beginnen unsere Arbeit in Kapitel 2 mit einer Betrachtung der für das Verständnis nötigen Grundlagen. Dazu geben wir die Definitionen von Petrinetzen und offenen Workflow-Netzen an. Zudem werden wir die Einschränkungen festlegen, die offene Workflow-Netze erfüllen müssen, damit sie transformiert werden können. Anschließend geben wir eine kurze Einführung in WS-BPEL und ihre Aktivitäten.

Kapitel 3 widmet sich dann der Transformation von offenen Workflow-Netzen in WS-BPEL-Prozesse. Nach einer kurzen Erläuterung der Transformationsidee werden wir die einzelnen, Annotationen erzeugenden Transformationen detailliert vorstellen. Danach betrachten wir eine Reihe von Korrekturen, die wir auf die erzeugten Annotationen anwenden. Mit der Festlegung der genauen Ausführungsreihenfolge beenden wir das Kapitel.

In Kapitel 4 widmen wir uns der Implementierung der vorgestellten Transformation. Wir erläutern die Fähigkeiten und zusätzlichen Funktionen des daraus entstandenen Werkzeugs und geben ein Beispiel für die vollständige Transformation eines offenen Workflow-Netzes zu einem abstrakten WS-BPEL-Prozess an.

Den Abschluss unserer Arbeit bildet Kapitel 5. Hier fassen wir unsere Ergebnisse zusammen, betrachten alternative Vorgehensweisen aus der Literatur, die sich ebenfalls mit diesem Thema befassen, und diskutieren Ansätze und offene Probleme, die Ausgangspunkte für weitere Forschungsarbeit sein können.

In den Anhängen beweisen wir, dass ein Teil unserer Transformationen die Soundness-Eigenschaft bewahrt. Zudem finden sich hier längere Abschnitte von WS-BPEL-Code, der aus Beispielen in den Kapiteln 3 und 4 hervorgegangen ist. Außerdem vergleichen wir hier unsere Transformation mit dem alternativen Ansatz aus [AL05].

2. Grundlagen

Als Ausgangsbasis für die in dieser Arbeit vorgestellte Transformation verwenden wir offene Workflow-Netze, eine spezielle Klasse der Petrinetze. Wir betrachten daher in Abschnitt 2.1.1 zunächst Petrinetze und die mit ihnen verbundene Terminologie, bevor wir in Abschnitt 2.1.2 die offenen Workflow-Netze definieren. In Abschnitt 2.1.3 formulieren wir die Einschränkungen an offene Workflow-Netze, die wir für unsere Transformation treffen. Als Letztes widmen wir uns in Abschnitt 2.2 der Beschreibungssprache WS-BPEL.

2.1. Petrinetze und offene Workflow-Netze

2.1.1. Petrinetze

Petrinetze [Rei82] sind ein weit verbreiteter mathematischer Formalismus zur Darstellung von dynamischen Systemen.

Definition 1 (Petrinetz)

Wir definieren ein Petrinetz N als das 3-Tupel $N = (P, T, F)$ bestehend aus

- einer endlichen, nicht leeren Menge P von Plätzen,
- einer endlichen, nicht leeren Menge T von Transitionen mit $P \cap T = \emptyset$ und
- einer Menge von Kanten $F \subseteq (P \times T) \cup (T \times P)$.

┘

Zu den Vorteilen von Petrinetzen gehört ihre einfache Darstellbarkeit als gerichtete Graphen. Die Knoten des Graphen sind die Plätze des Netzes, durch Kreise gekennzeichnet, und die Transitionen, durch Rechtecke gekennzeichnet. Die Zustände in einem Petrinetz werden als Markierungen bezeichnet.

Definition 2 (Markierung)

Als Markierung eines Petrinetzes $N = (P, T, F)$ bezeichnen wir die Abbildung $m : P \rightarrow \mathbb{N}$.

┘

Durch diese Abbildung wird jedem Platz des Petrinetzes eine Anzahl von Marken, die wir graphisch durch schwarze, ausgefüllte Kreise darstellen werden, zugewiesen. Es gilt $m_i > m_j$ (bzw. $m_i \geq m_j$) genau dann wenn $\forall p \in P : m_i(p) > m_j(p)$ (bzw. $m_i(p) \geq m_j(p)$). Weiter bezeichnen wir für zwei Knoten x und y mit $F([x, y])$ die charakteristische Funktion der Abbildung F . Ein Petrinetz kann von einer Markierung in eine andere Markierung übergehen. Dieser Zustandsübergang wird als *Schalten* von Transitionen bezeichnet. Zur Definition des Schaltens benötigen wir neben der Markierung noch zwei

weitere Begriffe, die wir zunächst einführen wollen und die uns in dieser Arbeit noch häufiger begegnen werden.

Definition 3 (Vorbereich und Nachbereich)

Wir definieren $\bullet x = \{y \mid [y, x] \in F\}$ als Vorbereich des Knotens $x \in P \cup T$ und $x^\bullet = \{y \mid [x, y] \in F\}$ als seinen Nachbereich. Die Begriffe können kanonisch auf Mengen erweitert werden. Für eine Knotenmenge $A \subseteq P \cup T$ sei $A^\bullet = \bigcup_{a \in A} a^\bullet$ und $\bullet A = \bigcup_{a \in A} \bullet a$. \lrcorner

Definition 4 (Schalten einer Transition)

Eine Transition t ist genau dann aktiviert in Markierung m , wenn für alle Plätze $p \in \bullet t$ gilt: $m(p) \geq 1$.

Eine aktivierte Transition t kann schalten und führt dann von der Markierung m zur Markierung m' , definiert als: $\forall p \in P : m'(p) = m(p) - F([p, t]) + F([t, p])$.

Notiert wird das Schalten der Transition t als: $m \xrightarrow{t} m'$.

Als Schaltfolge oder auch Transitionssequenz bezeichnen wir eine endliche Sequenz von Schaltvorgängen in einem Petrinetz ausgehend von einer Markierung m des Netzes: $m \xrightarrow{t_1} m' \xrightarrow{t_2} m'' \xrightarrow{t_3} \dots \xrightarrow{t_n} m^n$ und als ihre Länge die Anzahl der Schaltvorgänge innerhalb der Schaltfolge. Wir schreiben abkürzend $m \xrightarrow{t_1 t_2 t_3 \dots t_n} m^n$. Mit $m \xrightarrow{\epsilon} m$ bezeichnen wir die leere Schaltfolge. \lrcorner

Aus der Definition folgt Korollar 2.1, welches wir benötigen, um im folgenden Abschnitt einen Beweis führen zu können.

Korollar 2.1 (Monotonie des Schaltens)

Seien m, m', m_1 Markierungen eines Petrinetzes $N = (P, T, F)$ und w eine Schaltfolge. Aus $m \xrightarrow{w} m'$ und $m_1 \geq m$ folgt: Es gibt ein $m_1' \geq m'$ mit $m_1 \xrightarrow{w} m_1'$.

Wir führen zusätzlich noch die folgenden Begriffe und abkürzenden Schreibweisen ein, die wir bei unserer Transformation verwenden werden und zur Definition unserer Transformationseinschränkungen benötigen, bevor wir uns der Klasse der offenen Workflow-Netze widmen.

Definition 5 (Erreichbarkeit)

Eine Markierung m' heißt erreichbar von Markierung m , genau dann wenn eine endliche Schaltfolge $m \xrightarrow{t_i} \dots \xrightarrow{t_j} m'$ im Petrinetz existiert. Wir notieren dies als: $m \xrightarrow{*} m'$. \lrcorner

Definition 6 (Pfad)

Sei $x \in P \cup T$ dann definieren wir die Menge

$$\text{Pfad}(x) = \{y \mid y \in x^\bullet \vee y \in (x^\bullet)^\bullet \vee y \in ((x^\bullet)^\bullet)^\bullet \vee \dots\}. \lrcorner$$

Definition 7 (Zyklus)

Ein Petrinetz $N = (P, T, F)$ besitzt einen Zyklus, wenn ein Knoten $x \in P \cup T$ existiert, für den gilt: $x \in \text{Pfad}(x)$.

Als Inneres eines Zyklus bezeichnen wir alle Knoten $y \in \text{Pfad}(x)$ für die gilt: $x \in \text{Pfad}(y)$. ┘

Abbildung 2.1 zeigt ein Petrinetz mit einem Zyklus. Wählen wir den Platz p_2 als Knoten x , dann ist $\text{Pfad}(x) = \{p_2, t_2, p_3, p_4, t_1\}$, aber p_2 ist nur Teil von $\text{Pfad}(p_2), \text{Pfad}(t_2), \text{Pfad}(t_3)$ und $\text{Pfad}(p_3)$, so dass das Innere des Zyklus aus den Knoten p_2, t_2, p_3, t_1 besteht.

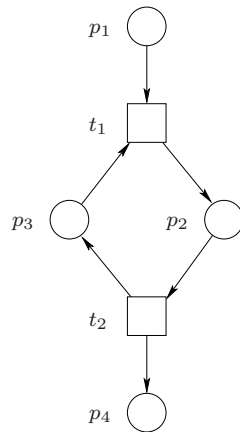


Abbildung 2.1.: Ein Beispiel für einen Zyklus.

Definition 8 (Beschränktheit und Sicherheit)

Ein Platz $p \in P$ eines Petrinetzes $N = (P, T, F)$ ist zusammen mit einer Anfangsmarkierung m_0 beschränkt, falls es ein beliebiges, aber festes $k \in \mathbb{N}$ gibt, so dass in jeder von m_0 erreichbaren Markierung m gilt: $m(p) \leq k$ für alle $p \in P$. Der Platz ist sicher, falls $k = 1$.

Das Petrinetz ist zusammen mit der Anfangsmarkierung beschränkt (sicher), wenn jeder Platz $p \in P$ beschränkt (sicher) ist. ┘

Definition 9 (Tote Transition)

Eine Transition $t \in T$ eines Petrinetzes $N = (P, T, F)$ ist zusammen mit einer Anfangsmarkierung m_0 tot, falls t in keiner von m_0 erreichbaren Markierung schalten kann. ┘

2.1.2. Offene Workflow-Netze

Eine spezielle Klasse der Petrinetze sind die offenen Workflow-Netze oder kurz oWFN [MRS05]. Sie können als eine allgemeinere Form der Workflow-Netze [Aal98] angesehen

werden. Im Gegensatz zu diesen verfügen oWFN jedoch über ein Interface über dessen Plätze der Empfang und Versand von Nachrichten nachgebildet werden kann. Marken auf den Interfaceplätzen modellieren dabei die Nachrichten die zwischen mehreren oWFN ausgetauscht werden. Offene Workflow-Netze können durch ihr Interface miteinander komponiert werden. Dies ermöglicht die Analyse kommunizierender Geschäftsprozesse [RSS05] und die Bedienbarkeit dieser [Sch05]. Ein weiterer Unterschied zwischen Workflow-Netzen und offenen Workflow-Netzen ist die Behandlung von Anfangs- und Endmarkierungen. Während ein Workflow-Netz nur einen markierten Anfangsplatz i besitzt, erlaubt die Definition von oWFN eine beliebige Anfangsmarkierung m_0 . Dem Endplatz o steht in einem oWFN eine Menge von Endmarkierungen Ω gegenüber. Zudem muss jeder Knoten eines Workflow-Netzes auf einem Pfad zwischen i und o liegen. Offene Workflow-Netze unterliegen dieser Einschränkung nicht. Wir betrachten nun die Definition.

Definition 10 (Offenes Workflow-Netz)

Seien

- $N = (P, T, F)$ ein Petrinetz,
- $P_i, P_o \subset P$ mit $\bullet P_i = P_o \bullet = \emptyset, P_i \cap P_o = \emptyset$,
- m_0 eine Markierung von N , die Anfangsmarkierung, und
- Ω eine Menge von Markierungen von N , die Endmarkierungen,

dann bezeichnen wir N zusammen mit P_i, P_o, m_0 , und Ω als ein offenes Workflow-Netz oder kurz oWFN. ┘

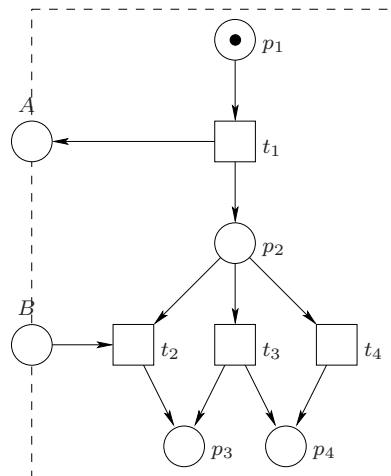


Abbildung 2.2.: Beispiel für ein offenes Workflow-Netz.

Ein oWFN lässt sich aufteilen in sein Interface und sein inneres Netz. Das Interface besteht aus den Eingangsplätzen P_i und den Ausgangsplätzen P_o . Das innere Netz ist das oWFN ohne die Plätze des Interfaces und ohne die Kanten von und zu diesen Plätzen. In den abgebildeten Beispielen in dieser Arbeit werden wir Knoten des inneren Net-

zes mit Kleinbuchstaben (p, t, e, u, p_1, \dots) und Plätze des Interfaces mit Großbuchstaben (A, B, \dots) benennen. Abbildung 2.2 zeigt ein Beispiel für die graphische Darstellung eines oWFN. Das Netz besitzt ein Interface bestehend aus dem Ausgangsplatz A und dem Eingangsplatz B . Die Plätze des Interfaces werden auf einem, das oWFN umgebenden, gestrichelten Rahmen eingezeichnet, die Knoten des inneren Netzes innerhalb dieses Rahmens. Das innere Netz setzt sich zusammen aus den Plätzen p_1, p_2, p_3 und p_4 und den Transitionen t_1, t_2, t_3 und t_4 . Die Anfangsmarkierung ist $m_0 = [p_1]$ und die Menge der Endmarkierungen $\Omega = \{[p_3, p_4]\}$.

2.1.3. Einschränkungen

Wir schränken die Klasse der oWFN, die wir mit unserer Transformation in abstrakte WS-BPEL-Prozesse umwandeln, ein. Dies ist notwendig, weil sich nicht für alle mit Hilfe von oWFN modellierbaren Teilnetze entsprechende Nachbildungen in WS-BPEL-Prozessen erstellen lassen oder von der WS-BPEL Spezifikation [AAA⁺07] besondere Bedingungen gestellt werden. Dies ist vor allem bei Zyklen der Fall. Hinzu kommt, dass unsere Einschränkungen auch oWFN ausschließen, deren Transformationen verklemmende WS-BPEL-Prozesse erzeugen würden. Alle oWFN jedoch, die diesen Einschränkungen genügen, können durch unsere Transformation vollautomatisch in abstrakte WS-BPEL-Prozesse umgewandelt werden.

Die wichtigste Einschränkung, die wir treffen, ist die Forderung, dass das Innere des oWFN¹ die *Soundness* Eigenschaft erfüllt. Die Soundness-Eigenschaft wurde zuerst von van der Aalst in [Aal98] für Workflow-Netze eingeführt. Sie stellt Anforderungen an die Dynamik eines Netzes und sichert dessen ordnungsgemäßes Terminieren. Das Erreichen eines Endzustands wird dabei garantiert, und es bleiben nach dem Erreichen eines solchen keine Marken, außer denen der Endmarkierung, im Netz zurück. Zudem schließt sie unerwünschtes Verhalten, wie Deadlocks, Livelocks und tote Transitionen, aus. Die Eigenschaft ist für die Modellierung von Geschäftsprozessen angepasst, bei der das ordnungsgemäße Beenden von Prozessen und ihre Deadlockfreiheit besonders wichtig sind.

Wir passen für diese Arbeit, auf Grund der Unterschiede zwischen Workflow-Netzen und oWFN, die Definition der Soundness-Eigenschaft an. Dazu betrachten wir die Abläufe im inneren Netz des oWFN und schränken seine Anfangsmarkierung auf einen markierten Platz i mit leerem Vorbereich ein. In der Menge der Endmarkierungen erlauben wir nur noch Markierungen in denen Plätze mit leerem Nachbereich markiert sind.

Definition 11 (Soundness)

Gegeben ein oWFN bestehend aus dem Petrinetz $N = (P, T, F, \cdot)$ zusammen mit dem Interface P_i, P_o , der Anfangsmarkierung m_0 und der Menge von Endmarkierungen Ω .

Forderungen an die Struktur des oWFN:

1. Platz $i \in P$ ist der einzige Platz, der in der Anfangsmarkierung markiert ist:

$$\forall p \in (P \setminus \{i\}) : m_0(p) = 0 \wedge m_0(i) = 1$$

¹Wenn wir im Rest dieser Arbeit von dem oWFN sprechen, meinen wir damit ein oWFN, auf das unsere hier vorgestellte Transformation angewendet wird.

2. Alle Plätze, die Teil einer Endmarkierung sind, sind in den Endmarkierungen mit genau einer Marke markiert und alle anderen Plätze sind nicht markiert:

$$\forall m_\Omega \in \Omega : \forall o \in P \text{ mit } [o] \in m_\Omega : m_\Omega(o) = 1$$

3. Alle Plätze, die Teil einer Endmarkierung sind, besitzen einen leeren Nachbereich:

$$\forall o \in P : \exists m_\Omega \in \Omega \text{ mit } [o] \in m_\Omega : o^\bullet = \emptyset$$

Forderungen an das Verhalten des inneren Netzes:

1. Aus jeder erreichbaren Markierung heraus existiert eine Schaltfolge, die zu einer der Endmarkierungen führt:

$$\forall m : (m_0 \xrightarrow{*} m) \Rightarrow (m \xrightarrow{*} m_\Omega) \text{ mit } m_\Omega \in \Omega$$

2. Von der Anfangsmarkierung m_0 aus sind keine Markierungen erreichbar, in denen alle Plätze einer Endmarkierung markiert sind, außer dieser Endmarkierung selbst:

$$\forall m : (m_0 \xrightarrow{*} m \wedge m \geq m_\Omega) \Rightarrow (m = m_\Omega) \text{ mit } m_\Omega \in \Omega$$

3. Das innere Netz enthält keine toten Transitionen:

$$\forall t \in T \exists m, m' : m_0 \xrightarrow{*} m \xrightarrow{t} m'$$

Wir nennen das oWFN *sound*, genau dann wenn es die Forderungen an die Struktur und an das Verhalten erfüllt. ┘

Aus der klassischen Definition der Soundness-Eigenschaft in [Aal98] folgt, dass ein Workflow-Netz, das *sound* ist, auch beschränkt ist. Ein offenes Workflow-Netz, das nach unserer Definition *sound* ist, ist ebenfalls beschränkt. Wir passen den Beweis aus [Aal98] an:

Beweis. Angenommen das oWFN N ist *sound*, aber nicht beschränkt. Da N nicht beschränkt ist, muss es zwei Markierungen m_i und m_j geben, so dass $m_0 \xrightarrow{*} m_i$, $m_i \xrightarrow{*} m_j$ und $m_i > m_j$, wegen der Monotonie des Schaltens. Da N jedoch *sound* ist, muss es eine Schaltfolge σ geben, so dass $m_i \xrightarrow{\sigma} m_\Omega$ mit $m_\Omega \in \Omega$. Daher muss eine Markierung m existieren, so dass $m_j \xrightarrow{\sigma} m$ mit $m > m_\Omega$. So ist es nicht möglich, dass N *sound* und nicht beschränkt ist. □

Da es unser Ziel ist, bei unserer Transformation einen fehlerfreien WS-BPEL-Prozess zu erzeugen, fordern wir zudem, dass kein Zustand erreichbar sein darf, in dem zwei oder mehr Transitionen aktiv sind, in deren Vorbereich sich derselbe Platz aus P_i befindet. Wie wir in Abschnitt 2.2.3 sehen werden, definiert die WS-BPEL Spezifikation [AAA⁺07] einen Laufzeitfehler, falls mehrere Aktivitäten gleichzeitig dieselbe Nachricht empfangen können. Daher können wir dieses im oWFN mögliche Verhalten nicht nachbilden.

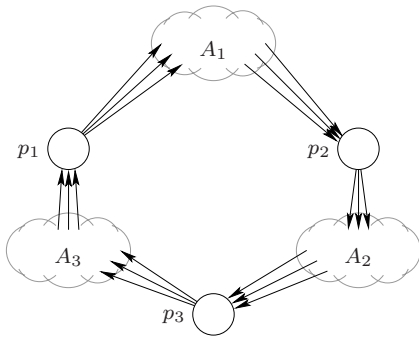
Unsere letzte Einschränkung betrifft den Aufbau von Zyklen. In den Abschnitten 2.2.5, 3.2.4, 3.2.6 und 3.2.7 werden wir auf die Beschränkungen bezüglich Zyklen in WS-BPEL

eingehen. Um uns an diese Beschränkungen halten zu können, erlauben wir nur Zyklen, die zwischen den Ein- und Ausgangsplätzen² nur Teilnetze besitzen, die die Soundness-Eigenschaft erfüllen. Die Vor- und Nachbereiche von Transitionen aus dem Inneren eines Zyklus dürfen nur Plätze enthalten, die sich ebenfalls im Inneren des Zyklus befinden. Keine Transition im Nachbereich eines Ausgangs darf Teil des Nachbereichs eines weiteren Ausgangs desselben Zyklus sein.

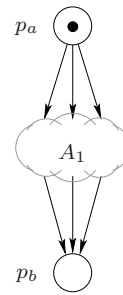
Wir formulieren diese Einschränkung wie folgt: Zyklen im oWFN müssen einem bestimmten Aufbau folgen und aus einer alternierenden Abfolge von Plätzen $p_1, p_2, \dots, p_n \in P$ und Teilnetzen $A_1, A_2, \dots, A_n \subset (T \cup P)$ mit $n \in \mathbb{N}$ bestehen. Jedes Teilnetz A_i muss dabei zusammen mit zwei Plätzen p_a und p_b , für die gilt:

- $\bullet p_a = p_b \bullet = \emptyset$,
- $p_a \bullet = p_i \bullet \cap A_i$ und
- $\bullet p_b = \bullet p_{i-1} \cap A_i$,

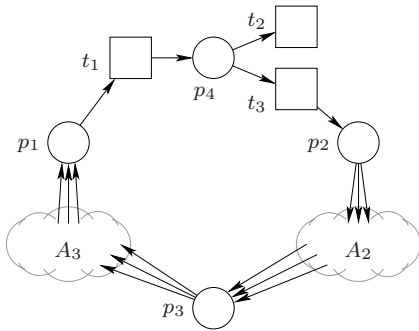
einer Anfangsmarkierung $m_0 = [p_a]$ und nur einer Endmarkierung $m_\Omega = [p_b]$ die Soundness-Eigenschaft erfüllen. Für die Plätze gilt $\forall p_i, p_j \in \{p_1, p_2, \dots, p_n\} : p_i \bullet \cap p_j \bullet = \emptyset$.



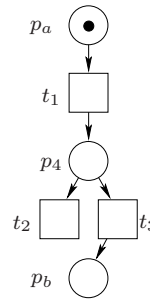
(a) Beispiel für einen Zyklus mit drei Teilnetzen.



(b) Ein einzelnes Teilnetz.



(c) Beispiel für einen ungültigen Zyklus.



(d) Ein Teilnetz, das nicht sound ist.

Abbildung 2.3.: Beispiele für die Einschränkung des Aufbaus von Zyklen.

²Eine Definition von Ein- und Ausgängen von Zyklen findet sich in Abschnitt 3.2.6.

Abbildung 2.3(a) zeigt einen Zyklus mit drei Plätzen (p_1, p_2, p_3) und drei Teilnetzen (A_1, A_2, A_3). Die Teilnetze sind durch Wolken repräsentiert. Kanten zwischen den Komponenten des Zyklus sind lediglich Beispiele für das Vorhandensein von derart gerichteten Kanten. In diesem Beispiel ist ihre Anzahl beliebig, solange mindestens eine Kante vorhanden ist. In Abb. 2.3(b) ist das Teilnetz A_1 zusammen mit den Plätzen p_a und p_b dargestellt, das die Soundness-Eigenschaft erfüllen muss. Abbildung 2.3(c) zeigt einen Zyklus in dem das Teilnetz A_1 angegeben ist. Wie in Abb. 2.3(d) zu sehen, erfüllt der Zyklus nicht die geforderten Einschränkungen, da das Teilnetz nicht sound ist.

2.2. WS-BPEL

Die *Web Services Business Process Execution Language* [AAA⁺07] oder kurz *WS-BPEL* ist eine weit verbreitete und anerkannte Beschreibungssprache für auf Web Services basierende Geschäftsprozesse.

2.2.1. Abstrakte und ausführbare Prozesse

In WS-BPEL können zwei Arten von Prozessen beschrieben werden. In der Spezifikation [AAA⁺07] wird unterschieden zwischen abstrakten Prozessen, den *WS-BPEL Abstract Processes*, und ausführbaren Prozessen, den *WS-BPEL Executable Processes*. So wie offene Workflow-Netze beschreibenden Charakter besitzen, so auch die abstrakten Prozesse. Sie können benutzt werden, um das beobachtbare Verhalten eines ausführbaren Prozesses zu formulieren oder um eine Vorlage zu liefern, aus der durch Vervollständigung ausführbare Prozesse erstellt werden können. Dazu können Teile ihres Verhaltens bei der Spezifikation eines Prozesses verdeckt oder ausgelassen werden. Zudem können sie den gesamten Funktionsumfang von ausführbaren Prozessen nutzen. Ein ausführbarer WS-BPEL-Prozess muss vollständig spezifiziert werden und kann auf einer *WS-BPEL runtime engine* ausgeführt werden.

2.2.2. Aktivitäten

Die WS-BPEL-Aktivitäten unterteilen sich in Basisaktivitäten und strukturierte Aktivitäten. Die Gruppe der Basisaktivitäten führt elementare Schritte des Prozesses aus. Dazu zählt vor allem der Nachrichtenaustausch, für den mehrere Aktivitäten zur Verfügung stehen. Für die anderen Basisaktivitäten nutzen wir in dieser Arbeit die Möglichkeit diese in abstrakten Prozessen als verdeckt anzugeben. Die strukturierten Aktivitäten von WS-BPEL definieren die kausale Ordnung der Basisaktivitäten, indem sie diese und weitere strukturierte Aktivitäten einbetten. Zu den strukturierten Aktivitäten zählen unter anderem **sequence** (sequentielle Ausführung der eingebetteten Aktivitäten), **flow** (nebenläufige Ausführung der eingebetteten Aktivitäten und Definition einer Halbordnung auf diesen), **while** (wiederholte Ausführung von eingebetteten Aktivitäten), **if** (bedingte Ausführung einer eingebetteten Aktivität) und **pick** (bedingte Ausführung einer eingebetteten Aktivität auf Basis von externen Nachrichten oder Zeitereignissen).

Die strukturierte Aktivität `scope` bietet die Möglichkeit mit ihr verknüpften Elemente zur Fehler- und Ereignisbehandlung und nur für die `scope`-Aktivität gültige Variablen zu nutzen. Die äußerste `scope`-Aktivität eines Prozesses ist die `process`-Aktivität. Nach der Beendigung dieser Aktivität ist der gesamte Geschäftsprozess abgearbeitet.

Wir werden im Folgenden die für diese Arbeit wichtigen Aktivitäten und Aspekte von WS-BPEL genauer vorstellen. Dabei beschränken wir uns auf Details, die dazu beitragen, dass wir in Kapitel 3 eine Auswahl treffen können, welche dieser Aktivitäten wir während der Transformation des oWFN verwenden. Unsere folgende Betrachtung gliedert die Aktivitäten daher nach ihren Verwendungsmöglichkeiten. Umfangreiche Aspekte von WS-BPEL, wie die Fehlerbehandlung und der Umgang mit Variablen, die wir bei der Transformation nicht benötigen werden, werden wir nicht detaillierter erklären.

2.2.3. Kommunizierende Aktivitäten

In WS-BPEL sind zwei Aktivitäten spezifiziert, die in der Lage sind, Nachrichten zu senden und drei weitere Aktivitäten, die Nachrichten empfangen können. Wir stellen zunächst die Aktivität `invoke`³ vor, die Nachrichten versendet.

Die WS-BPEL-Basisaktivität `invoke` wird benutzt, um Operationen anderer Web Services aufzurufen. Die Aktivität sendet als Daten der Nachricht den Inhalt einer mit anzugebenden Variablen. Die Aktivität ist nach dem Senden der Nachricht beendet. Wir werden im Rahmen dieser Arbeit für jegliche ausgehenden Nachrichten die Aktivität `invoke` verwenden⁴.

Als nächstes betrachten wir die Basisaktivität `receive`, die den Empfang von Nachrichten ermöglicht. Die Daten der Nachricht werden dabei in einer anzugebenden Variablen gespeichert. Die Aktivität wartet so lange ab, bis eine geeignete Nachricht eintrifft oder die Aktivität durch einen Fehler im Prozess abgebrochen wird. Wurde eine Nachricht empfangen und in der Variable gespeichert, ist die Aktivität beendet.

Neben der Aktivität `receive` erlaubt auch die strukturierte Aktivität `pick` den Empfang von Nachrichten⁵. Die Aktivität ist für die ereignisbasierte Auswahl von eingebetteten Aktivitäten gedacht. Daher werden wir diesen Teil des Verhaltens der Aktivität in folgenden Abschnitt im Detail vorstellen, wenn wir uns mit den WS-BPEL-Aktivitäten für bedingtes Verhalten beschäftigen. Das Empfangsverhalten der `pick`-Aktivität entspricht ansonsten dem der `receive`-Aktivität. Die Aktivität benötigt ebenfalls eine Variable,

³WS-BPEL-Aktivitäten und weitere Elemente der Beschreibungssprache werden in dieser Arbeit im `Typewriterfont` gesetzt.

⁴Die ebenfalls zum Nachrichtenversand nutzbare Basisaktivität `reply` ist nur für das Beantworten von Nachrichten vorgesehen. Uns liefert das Interface des oWFN jedoch keine Details bezüglich des Nachrichtenverlaufs zwischen dem beschriebenen Geschäftsprozess und weiteren Geschäftsprozessen. Der Nutzer kann später mit Hilfe von zusätzlichen Informationen `invoke`-Aktivitäten durch `reply`-Aktivitäten ersetzen.

⁵Das mit der strukturierten Aktivität `scope` verknüpfte Element `<eventHandlers>` wird nebenläufig zu den in die `scope`-Aktivität eingebetteten Aktivitäten ausgeführt und ist daher in den von uns betrachteten Fällen nicht einsetzbar.

um die Daten einer empfangenen Nachricht zu speichern und blockiert so lange den Kontrollfluss des Prozesses, bis eines der Ereignisse eintritt.

In WS-BPEL gibt es eine Einschränkung für die Verwendung der empfangenden Aktivitäten. Es können nicht gleichzeitig zwei oder mehr Aktivitäten, seien es `receive` oder `pick` oder eine Kombination aus beiden, auf dieselbe Nachricht warten, die sich durch dieselben Werte in den Attributen *partnerLink*⁶, *portType*, *operation* und *correlationSet* auszeichnet. In diesem Fall tritt der WS-BPEL Standardfehler *bpel:conflictingReceive* ein. Da wir für jede Transition, in deren Vorbereich sich ein Platz aus P_i befindet, eine empfangende Aktivität in den von uns erzeugten WS-BPEL-Code einfügen, haben wir in Abschnitt 2.1.3 die Einschränkung gemacht, dass im oWFN kein Zustand erreichbar ist, in dem mehrere Transitionen schalten können, in deren Vorbereich sich derselbe Platz aus P_i befindet.

2.2.4. Aktivitäten für bedingtes Verhalten

WS-BPEL stellt zwei strukturierte Aktivitäten zur bedingten Auswahl von Zweigen im Ablauf zur Verfügung. Hierbei handelt es sich um die Aktivitäten `if` und `pick`. Letztere haben wir im vorangegangenen Abschnitt bereits angesprochen und werden sie an dieser Stelle detaillierter vorstellen.

Die Aktivität `if` wertet eine oder mehrere Boolesche Bedingungen aus und wählt dadurch eine in sie eingebettete Aktivität aus. Die Aktivität besteht aus dem `<if>`-Element, einem optionalen `<else>`-Element und mehreren optionalen `<elseif>`-Elementen. Kann die mit dem `<if>`-Element verknüpfte Boolesche Bedingung zu wahr ausgewertet werden, wird die mit dem `<if>`-Element verknüpfte Aktivität ausgeführt. Wird die Bedingung zu falsch ausgewertet, wird die Bedingung des ersten `<elseif>`-Elements ausgewertet und so weiter. Hierbei ist die Reihenfolge, in der die `<elseif>`-Elemente angegeben sind, von Bedeutung. Kann keine der Bedingungen zu wahr ausgewertet werden, wird die Aktivität ausgeführt, die mit dem `<else>`-Element verbunden ist. Ist kein `<else>`-Element angegeben, wird keine Aktivität ausgeführt und die `if`-Aktivität beendet. Aus diesem Grund werden wir jedes mal, wenn wir die Aktivität `if` verwenden, ein `<else>`-Element mit angeben. Die `if`-Aktivität ist beendet, wenn die ausgewählte eingebettete Aktivität beendet ist. Wir werden bei der Verwendung der Aktivität `if` während unserer Transformation keine Reihenfolge der eingebetteten Aktivitäten vorgeben. Die Reihenfolgen, in denen Aktivitäten innerhalb der Beispiele dieser Arbeit in Elemente der Aktivität `if` eingebettet werden, sind beliebig vertauschbar. Es bleibt dem Nutzer überlassen, wie er nach der Transformation eines oWFN zu WS-BPEL-Code die Reihenfolgen der Elemente und die dazugehörigen Bedingungen festlegt.

Bei der Aktivität `pick` werden Ereignisse mit der Auswahl der eingebetteten Aktivitäten verknüpft. Bei diesen Ereignissen kann es sich um den Empfang einer bestimmten Nachricht, unter Verwendung des `<onMessage>`-Elements, oder um das Ablaufen eines Timers, unter Verwendung des `<onAlarm>`-Elements, handeln. Es wird nur die Aktivität ausgeführt, die mit dem Ereignis verknüpft ist, welches zuerst eintritt. Die Aktivität `pick`

⁶WS-BPEL-Attribute und Werte werden in dieser Arbeit *kursiv* gesetzt.

ist nach der Ausführung der ausgewählten Aktivität beendet. Jede `pick`-Aktivität muss mindestens ein `<onMessage>`-Element enthalten. Die Aktivität ist beendet, wenn eine der in sie eingebetteten Aktivitäten ausgeführt und beendet wurde.

2.2.5. Nebenläufige Ausführungen

Für die nebenläufige Ausführung von Aktivitäten existiert in WS-BPEL die strukturierte Aktivität `flow`. Mit Hilfe von Links kann eine Halbordnung zwischen den in eine `flow`-Aktivität eingebetteten Aktivitäten festgelegt und Synchronisierungen zwischen ihnen vorgenommen werden. Beim Aufruf der `flow`-Aktivität werden alle in sie eingebetteten Aktivitäten, die nicht Ziel von Links sind oder deren Aktivierungsbedingung bereits zu wahr ausgewertet werden kann, gestartet. Die Aktivierungsbedingungen der Aktivitäten werden mit Hilfe des Zustands der eingehenden Links ausgewertet. Die Aktivität `flow` ist beendet, wenn alle in sie eingebetteten Aktivitäten beendet wurden.

Links werden innerhalb von `flow`-Aktivitäten verwendet, um Synchronisationsbeziehungen zwischen den eingebetteten Aktivitäten festzulegen. Ein Link besteht aus einem `<source>`- und einem `<target>`-Element. Diese sind jeweils Teil einer der eingebetteten Aktivitäten und bestimmen diese zu Quellen und Zielen von Links. Aktivitäten können sowohl Quelle als auch Ziel von mehreren Links sein. Zwischen zwei Aktivitäten darf aber immer nur eine Linkverknüpfung bestehen. Nachdem eine Aktivität beendet ist, wird der Zustand der ausgehenden Links geändert. Erst wenn der Zustand aller eingehenden Links einer Zielaktivität feststeht, wird die Boolesche Aktivierungsbedingung der Aktivität, die durch das optionale Element `<joinCondition>` definiert wird, ausgewertet. Wird die Bedingung zu wahr ausgewertet, wird nun die Zielaktivität ausgeführt.

Links, die innerhalb einer `flow`-Aktivität definiert sind, dürfen nicht denselben Bezeichner tragen. Quelle und Ziel eines Links können selbst in verschiedene strukturierte Aktivitäten eingebettet sein. Jedoch dürfen Links nicht in eine der Aktivitäten zur wiederholten Ausführung hinein oder aus ihnen herausführen. Zusätzlich dürfen Links selbst keine Zyklen im Kontrollfluss bilden.

2.2.6. Zyklen

Die Aktivität `while` führt in sie eingebettete WS-BPEL-Aktivitäten so oft aus, wie eine Boolesche Bedingung zu Beginn jeder Wiederholung zu wahr ausgewertet werden kann. Wird die Bedingung zu falsch ausgewertet, ist die Aktivität beendet. So kann es auch sein, dass es zu keiner Ausführung der eingebetteten Aktivitäten kommt.

Die beiden Aktivitäten `repeatUntil` und `forEach` bieten ebenfalls die Möglichkeit in sie eingebettete Aktivitäten mehrfach auszuführen. Wir werden in Abschnitt 3.2.6 kurz auf sie eingehen.

Damit endet unsere Einführung in WS-BPEL. Im nächsten Kapitel werden wir uns mit der Transformation von oWFN zu abstrakten Prozessen beschäftigen. Dabei werden wir an einigen Stellen kurz weitere Aspekte von WS-BPEL erläutern.

3. Transformation

Mit Hilfe einer Reihe von Transformationen reduzieren wir ein oWFN auf eine Annotation. Auf dieser Annotation führen wir weitere Korrekturen durch, um dann schließlich aus ihr den WS-BPEL-Code erzeugen. Als erstes stellen wir im folgenden Abschnitt 3.1 die Idee dieser Art der Transformation näher vor. Anschließend betrachten wir in Abschnitt 3.2 die Transformationen mit denen wir das oWFN reduzieren und dann in Abschnitt 3.3 die Korrekturen mit denen wir die Annotation weiter verändern. Abschließend widmen wir uns in Abschnitt 3.4 der Anwendungsreihenfolge der Transformationen und Korrekturen.

3.1. Idee der Transformation

Unser Ziel ist die Transformation eines oWFN zu einem abstrakten WS-BPEL-Prozess. Um dieses Ziel zu erreichen suchen wir zuvor definierte Muster im oWFN und ersetzen Teile des Netzes durch annotierte, zu meist kleinere Teilnetze. Als Annotationen für Transitionen und Plätze, die entfernt werden, verwenden wir dabei nicht sofort den finalen WS-BPEL-Code des von uns erzeugten abstrakten Prozesses. Die von uns eingefügten Annotationen enthalten Informationen darüber, welche WS-BPEL-Aktivitäten wir zu diesem Zeitpunkt der Transformation für geeignet halten, den transformierten Teil des oWFN zu repräsentieren. Dies ermöglicht es uns, im Laufe der Transformation an den Annotationen noch Änderungen vorzunehmen, bevor wir den endgültigen WS-BPEL-Code erzeugen. Wir können passendere WS-BPEL-Aktivitäten einfügen, Aktivitäten gruppieren und kapseln oder unnötige vollständig entfernen. So ist es uns möglich, einen kompakteren Code für den WS-BPEL-Prozess zu erzeugen.

Unsere Transformation endet daher auch nicht, wenn das gesamte oWFN zu einem Knoten reduziert werden konnte. Stattdessen werden wir an dieser Stelle mit der so erhaltenen Annotation weitere Transformationen durchführen, die wir im Folgenden als Korrekturen bezeichnen werden. Wir verzichten beispielsweise bis zu dieser Stelle bewusst auf die Verwendung der WS-BPEL-Aktivität `sequence`. Diese häufig zum Einsatz kommende strukturierte Aktivität umgibt mehrere andere Aktivitäten, die sequentiell hintereinander ausgeführt werden. Wir warten hingegen alle Transformation des oWFN und die Korrekturen Annotation ab. Erst dann umgeben wir alle verbleibenden WS-BPEL-Aktivitäten, die nun noch in einer Sequenz stehen, mit der Aktivität `sequence`. Auf diese Art wird nur eine geringe Anzahl an `sequence`-Aktivitäten im WS-BPEL-Code erzeugt.

Wir beschränken uns bei unseren Transformationen nicht auf nur ein Suchmuster für jede WS-BPEL-Aktivität, die wir in die Annotationen einfügen, sondern definieren für einzelne Aktivitäten teilweise mehrere Muster und führen Transformation ein, die das oWFN strukturell umformen um weitere Transformationen zu ermöglichen. Zudem entsteht bei

einer Transformation nicht immer nur eine neue Annotation oder Annotationen, die aus nur einer WS-BPEL-Aktivität bestehen. Teilweise wird für ein Suchmuster ein Konstrukt aus mehreren Aktivitäten in die Annotationen eingefügt.

Unsere Annotationen sind an den auf XML basierenden WS-BPEL-Code angelehnt und ebenso verschachtelt. In der Regel verwenden wir den Namen der Aktivität ohne weitere Elemente und Attribute als Annotation. Bei strukturierten Aktivitäten wird vermerkt, welche Aktivitäten sie einbetten, diese bezeichnen wir als Zweige der Aktivität. Folgen mehrere Aktivitäten aufeinander, bei denen es sich nicht jeweils um einen Zweig einer strukturierten Aktivität handelt, ist dies so zu verstehen, dass die Aktivitäten sequentiell nacheinander ausgeführt werden sollen. Wie zuvor angemerkt, wird die Aktivität `sequence` erst zum Ende der Transformation mit in die Annotationen eingefügt.

Codebeispiel 3.1 zeigt ein Beispiel für Annotationen. Die Aktivitäten `flow`, `while` und `if` stehen in einer Sequenz. Die `flow`- und die `if`-Aktivität besitzen jeweils zwei Zweige mit weiteren Aktivitäten. Die `while`-Aktivität kann nur eine andere Aktivität einbetten, daher ist die Angabe von Zweigen hier nicht nötig. Die in sie eingebetteten Aktivitäten `opaqueActivity` und `invoke` stehen daher ebenfalls in einer Sequenz. Die Bezeichner „A“, „B“, „C“ und „D“ in den `receive`- und `invoke`-Aktivitäten stehen für Nachrichten, die über Interfaceplätze mit diesen Bezeichnern empfangen oder gesendet werden. Die jeweiligen Details der Annotationen erläutern wir während der Transformationen, in denen sie Verwendung finden.

```
<flow>
  Zweig 1 mit Annotation:
  <receive "A" />
  Zweig 2 mit Annotation:
  <invoke "B" />
</flow>
<while>
  <opaqueActivity />
  <invoke "C" />
</while>
<if>
  Zweig 1 mit Annotation:
  <invoke "D" />
  Zweig 2 mit Annotation:
  <opaqueActivity />
</if>
```

Codebeispiel 3.1: Ein Beispiel für Annotationen.

Zu Beginn unserer Transformation sind alle Knoten des oWFN mit einer leeren Annotation versehen. Bei Transformationen, bei denen wir Transitionen aus dem oWFN entfernen, die noch mit keiner WS-BPEL-Aktivität annotiert sind, ändern wir ihre Annotation zuvor in die Aktivität `opaqueActivity`. Durch dieses Vorgehen werden die Transitionen des oWFN als Aktivitäten in den WS-BPEL-Code übernommen. Die Aktivität `opaqueActivity` ist ein Platzhalter für eine ausführbare Aktivität und alle ihre Attribute. Sie steht nur abstrakte WS-BPEL-Prozesse zur Verfügung und kann später durch den Nutzer

unserer Transformation durch die an dieser Stelle beabsichtigte Aktivität ersetzt werden. Neben der Verwendung der Aktivität `opaqueActivity` besteht in abstrakten Prozessen zudem die Möglichkeit Attribute, wie Kommunikationspartner, und Bedingungen der WS-BPEL-Aktivitäten, beispielsweise für Schleifenwiederholungen und bedingtes Verhalten, statt mit tatsächlichen Werten mit dem Wert `opaque` anzugeben. Auch diese Möglichkeit werden wir bei unserer Transformation nutzen und somit das Einfügen von Werten, über die wir anhand des oWFN keine Aussagen treffen können, dem Nutzer überlassen.

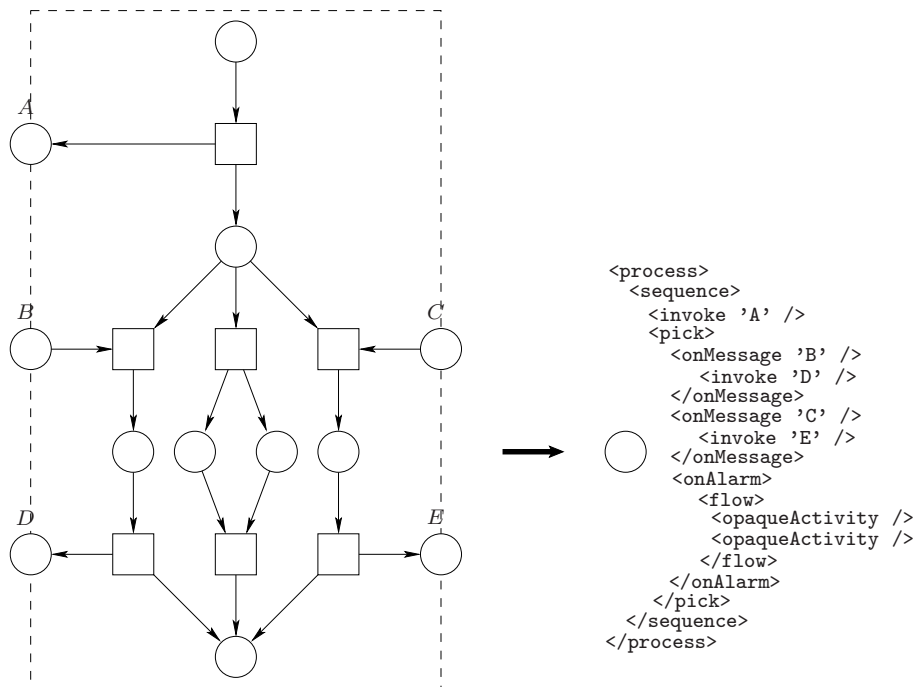


Abbildung 3.1.: Vereinfachtes Beispiel für die Transformation eines oWFN.

Die Transformationen werden wir in einer festgelegten Reihenfolge so lange ausführen, bis das oWFN auf einen annotierten Knoten reduziert wurde. Nach der Ausführung einer Transformation wird die Ausführung bei der ersten Transformation fortgesetzt, so dass spätere Transformationen erst erreicht werden, wenn alle vorherigen nicht mehr ausführbar sind. Ist das oWFN auf einen Knoten reduziert werden die Korrekturen, ebenfalls in einer festgelegten Reihenfolge, auf die erzeugte Annotation angewendet. Aus dieser Annotation entsteht dann in einem letzten Schritt der eigentliche WS-BPEL-Code. Da die Annotationen an die WS-BPEL-Aktivitäten angelehnt und wie diese verschachtelt sind, ist dazu nur eine einfache Ersetzung von Annotation durch den vollständigen WS-BPEL-Code einer Aktivität nötig. Abbildung 3.1 zeigt beispielhaft unsere Transformation in einem Schritt. Auf der linken Seite des Beispiels ist ein oWFN dargestellt und auf der rechten Seite der einzelne, annotierte Knoten, zu dem es reduziert wurde. Zur Vereinfachung haben wir an dieser Stelle auf eine Benennung der Knoten des inneren Netzes verzichtet.

Eine ähnliche Vorgehensweise wurde in [AL05] zur Transformation von sicheren Workflow-Netzen zu BPEL4WS-Prozessen¹ gewählt. Auch dort wird durch die Ersetzung von Suchmustern das gesamte Netz auf einen annotierten Knoten reduziert. Jedoch findet sich in den Annotationen gleich der WS-BPEL-Code, so dass es zu keiner späteren Optimierung kommt. Da Workflow-Netze kein Interface besitzen, ist die Transformation auf spezielle vorgegebene Annotationen des Workflow-Netzes angewiesen, um beispielsweise `pick`-Aktivitäten erkennen zu können. Durch die Beschränkung auf fünf Suchmuster, die jeweils nur eine WS-BPEL-Aktivität erzeugen, ist es notwendig, Nutzerrückfragen in die Transformation zu integrieren. So muss der Nutzer von Hand Teilnetze des Workflow-Netzes in WS-BPEL-Code umwandeln, wenn die Transformation kein Suchmuster mehr anwenden kann. Im Gegensatz dazu reduziert unsere Transformation oWFN ohne Nutzerrückfragen und ohne dass wir uns dabei auf sichere Netze beschränken.

3.2. Transformationen des oWFN

Die Transformationen, die wir im Folgenden beschreiben werden, ersetzen Teilnetze des oWFN durch in den meisten Fällen kleinere und annotierte Teilnetze. Teilweise können mehrere Transformationen das gleiche Teilnetz oder sich überschneidende Teilnetze reduzieren. Wir sprechen in diesem Fall davon, dass diese Transformationen in Konflikt zueinander stehen. Daher werden wir in Abschnitt 3.4 die Ausführungsreihenfolge der Transformationen so festlegen, dass sie optimal ausgenutzt werden. In den folgenden Abschnitten sind sie unabhängig von ihrer Ausführungsreihenfolge thematisch gruppiert. In der festgelegten Ausführungsreihenfolge wird diejenige Transformation gesucht, welche sich als erste auf das oWFN anwenden lässt. Nach deren Ausführung beginnt die Suche von vorn. Dabei werden die Transformationen, die nicht im Konflikt mit Transformationen stehen, die vor ihnen in der Ausführungsreihenfolge stehen, mehrfach ausgeführt und alle anderen nur jeweils einmal.

Als erstes werden wir in Abschnitt 3.2.1 die Transformation des Interface des oWFN betrachten. Diese Transformation muss nur einmal ausgeführt werden, da sie das gesamte Interface des oWFN entfernt und anschließend nicht mehr ausführbar ist. In den Abschnitten 3.2.2–3.2.8 widmen wir uns dann den weiteren Transformationen. Teilweise führen wir dabei bereits Korrekturen auf den erzeugten Annotationen ein, die nach der Ausführung der Transformationen angewendet und in Abschnitt 3.3 beschrieben werden.

3.2.1. Das Interface

Wie wir in Abschnitt 2.1 gesehen haben, besteht das Interface eines oWFN aus Plätzen auf die Marken durch externe Ereignisse produziert oder von denen Marken durch externe Ereignisse konsumiert werden können. Diese Ereignisse repräsentieren den Empfang und

¹Wir erzeugen bei unserer Transformation Prozesse in Version 2.0, die kürzlich als Nachfolgestandard der als *Business Process Execution Language for Web Services* [ACD⁺03] oder kurz *BPEL4WS* bekannten Version 1.1 beschlossen wurde.

das Versenden von Nachrichten und gestatten es mehrere oWFN an ihren Interfaceplätzen zu verschmelzen. Das Interface ist einer der größten Unterschiede zwischen oWFN und Workflow-Netzen. In Geschäftsprozessen, besonders in Form von Web Services, ist der Austausch von Nachrichten von zentraler Bedeutung. Über ihn können mehrere Geschäftsprozesse zusammenarbeiten oder die Leistungen anderer in Anspruch nehmen.

Während dieser Transformation werden wir zum Empfang von Nachrichten die WS-BPEL-Aktivität `receive` verwenden und zum Senden die Aktivität `invoke`. In Abschnitt 3.2.3 werden wir dann einige der hier eingefügten `receive` Annotationen durch `pick` Annotationen ersetzen. Durch dieses Vorgehen können wir alle Plätze des Interface in einer einmaligen Transformation aus dem oWFN entfernen und anschließend mit der Umwandlung des inneren Netzes fortfahren.

Alle Aktivitäten, die Nachrichten empfangen oder senden benötigen die Angabe des Attributs `partnerLink` zur Festlegung des Kommunikationspartners. In dem von uns erzeugten abstrakten Prozess geben wir einen Standardpartner mit dem Bezeichner `generic_pl` für alle sendenden und empfangenden Aktivitäten vor. Der Nutzer kann später die Art, die Inhalte und die Empfänger der Nachrichten selber festlegen und neue Partner definieren. Codebeispiel 3.2 zeigt das dazugehörige `<partnerLinks>`-Element, dass wir in Abschnitt 3.3.1 in die `process`-Aktivität einfügen werden. Im oWFN werden Nachrichten allein durch die Namen der Interfaceplätze repräsentiert. Da uns keine weiteren Informationen über die Inhalte, Typen und Empfänger der Nachrichten bekannt sind, geben wir im von uns erzeugten Prozess den Aktivitäten nur den Namen der Interfaceplätze mit und erzeugen entsprechend benannte Variablen. Die Variablen, die von den sendenden und empfangenden Aktivitäten verwendet werden, deklarieren wir im selben Abschnitt für jeden Interfaceplatz. Als Bezeichner wählen wir dafür den Wortteil `Var_` und den Namen des Interfaceplatzes. Auf Grund der Einschränkung bezüglich der Bezeichner von Variablen in WS-BPEL entfernen wir dabei alle Punkte aus den Namen. In Codebeispiel 3.3 werden die beiden Variablen „Var_A“ und „Var_B“ deklariert, damit diese für kommunizierende Aktivitäten, die aus den „A“ und „B“ genannten Interfaceplätzen entstehen, zur Verfügung stehen. Die Variablen brauchen vor ihrer Verwendung in diesen Aktivitäten nicht initialisiert werden, wenn das Abstract Process Profile verwendet wird.

```
<partnerLinks>
  <partnerLink name="generic_pl"
    partnerLinkType="##opaque"
    myRole="##opaque"
    partnerRole="##opaque" />
</partnerLinks>
```

Codebeispiel 3.2: Ein `<partnerLinks>`-Element.

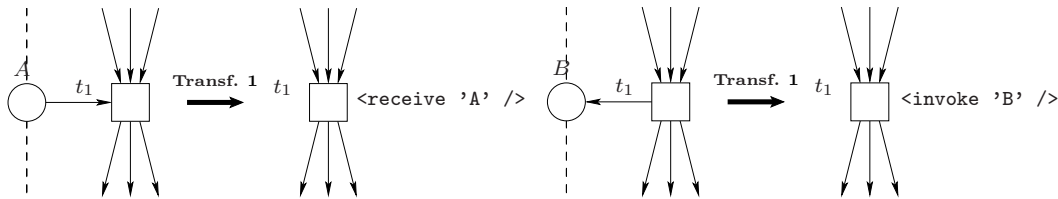
```
<variables>
  <variable name="Var_A"
    element="##opaque" />
  <variable name="Var_B"
    element="##opaque" />
</variables>
```

Codebeispiel 3.3: Variablendeklaration für die Interfaceplätze „A“ und „B“.

Transformation 1

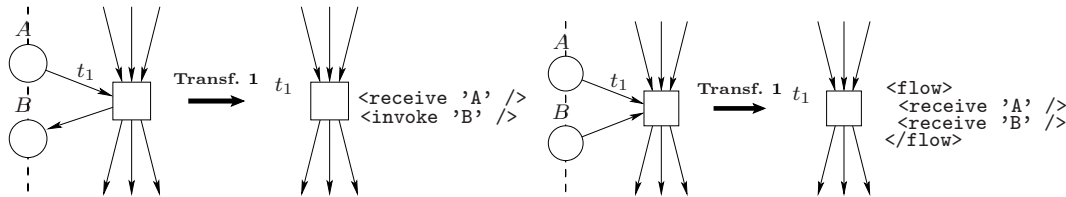
Wir werden mehrere Fälle unterscheiden, in welchen Kombinationen sich die Interfaceplätze im Vorbereich von Transitionen des inneren Netzes befinden können. Im den beiden einfachsten Fällen befindet sich ein Platz aus P_i oder P_o im Vorbereich einer Transition

des inneren Netzes. In diesem Fall können wir die Transition mit einer `receive` oder einer `invoke`-Aktivität annotieren und die Plätze aus dem Vorbereich der Transition entfernen. Abbildung 3.2(a) zeigt ein Beispiel für einen Platz aus P_i (A) und Abbildung 3.2(b) ein Beispiel für einen Platz aus P_o (B) im Vorbereich einer Transition. Wir verwenden in unseren Abbildungen eine abkürzende Annotation. Die Ausdrücke `<receive 'A' />` und `<invoke 'B' />` repräsentieren an dieser Stelle die Aktivitäten `receive` und `invoke`, die eine Nachricht „A“ empfangen beziehungsweise eine Nachricht „B“ senden. Sie werden zum Abschluss unserer Transformationen durch den WS-BPEL-Code in Codebeispiel 3.4 und Codebeispiel 3.5 ersetzt. Auf der rechten Seite der Abbildungen findet sich die annotierte Transition nach der von uns durchgeführten Transformation. Die Abbildungen zeigen jeweils nur die für die Transformationen relevanten Ausschnitte aus dem oWFN. Nur wenn Interfaceplätze vorhanden sind, stellen wir Teile des umgebenden, gestrichelten Rahmens dar. Mit Kanten, die keinen Anfangs- oder Endknoten besitzen, werden wir in dieser Arbeit das Vorhandensein des restlichen oWFN andeuten. In der Regel verwenden wir drei eingehende Kanten, falls für eine Transformation weitere Knoten im Vorbereich eines Knotens erlaubt sind und drei ausgehende Kanten, falls weitere Knoten im Nachbereich eines Knotens erlaubt sind. Dies ist eine für die Darstellung von Petrinetzen übliche Notation, die in ähnlicher Form unter anderem bereits der Autor in [Mur89] verwendet.



(a) Ein einzelner Nachrichteneingang.

(b) Ein einzelnes Sendereignis.



(c) Eine Kombination aus Senden und Empfangen.

(d) Eine Kombination aus zwei Nachrichteneingängen.

Abbildung 3.2.: Beispiele für die Transformation des Interface durch Transformation 1.

```
<receive name="Act_A"
partnerLink="generic_pl"
operation="A"
variable="Var_A" />
```

Codebeispiel 3.4: Ein Beispiel für eine `receive`-Aktivität.

```
<invoke name="Act_B"
partnerLink="generic_pl"
operation="B"
inputVariable="Var_B" />
```

Codebeispiel 3.5: Ein Beispiel für eine `invoke`-Aktivität.

Befindet sich je ein Platz aus P_i und ein Platz aus P_o im Vorbereich einer Transition ergänzen wir diese um die Annotation beider Aktivitäten. Wichtig hierbei ist, dass die Aktivität `receive` vor der Aktivität `invoke` aufgeführt wird. Im oWFN kann die Transition nur schalten, wenn eine Marke vom Eingangsplatz konsumiert werden kann, der Zustand des Ausgangsplatzes spielt dabei keine Rolle. Um dieses Verhalten so exakt wie möglich nachzuahmen, rufen wir im WS-BPEL-Code zuerst die Aktivität `receive` auf. Erst wenn diese erfolgreich eine Nachricht empfangen hat, wird sie beendet und die nächste Aktivität, in diesem Fall die `invoke`-Aktivität wird aufgerufen. Es wird also von dem von uns erzeugten Prozess keine Nachricht gesendet, die im oWFN den Empfang einer anderen Nachricht vorausgesetzt hätte, ohne dass diese Nachricht auch zuvor vom Prozess empfangen wurde. In Abb. 3.2(c) findet sich ein Beispiel für eine Transition, die eine Nachricht vom Platz „A“ konsumiert und eine auf dem Platz „B“ produziert.

Ein oWFN ist in der Lage durch das Schalten einer Transition jeweils eine Marke von mehreren Eingangsplätzen zu entfernen. Dies ist vergleichbar mit dem gleichzeitigen Empfang mehrerer Nachrichten. Da WS-BPEL nicht in der Lage ist, mit der Ausführung nur einer Aktivität mehrere Nachrichten zu empfangen, simulieren wir dieses Verhalten indem wir für jeden Platz aus P_i im Vorbereich einer Transition jeweils eine `receive`-Aktivität in eine `flow`-Aktivität einbetten. Hieraus ergibt sich die Annotation der Transition aus deren Vorbereich wir anschließend die Plätze aus P_i entfernen können. Codebeispiel 3.6 zeigt beispielhaft die Annotation für eine Transition in deren Vorbereich sich mehrere Plätze aus P_i befunden haben und Abb. 3.2(d) zeigt ein Beispiel für die Transformation.

```
<flow>
  <receive "A" />
  <receive "B" />
  ...
</flow>
```

Codebeispiel 3.6: Ein Beispiel für mehrere Empfangsereignisse.

```
<flow>
  <invoke "A" />
  <invoke "B" />
  ...
</flow>
```

Codebeispiel 3.7: Ein Beispiel für mehrere Sendeereignisse.

Ebenfalls ist es in einem oWFN möglich, dass mehrere Ausgangsplätze gleichzeitig mit Marken belegt werden. Wir simulieren den gleichzeitigen Versand mehrerer Nachrichten durch die Annotation mit einer `flow`-Aktivität in die für jeden Platz aus P_o im Nachbereich einer Transition eine `invoke`-Aktivität eingebettet ist. In Codebeispiel 3.7 findet sich beispielhaft die Annotation für eine Transition mit mehreren Plätzen aus P_o im Nachbereich.

Die Verwendung der strukturierten Aktivität `flow` ermöglicht es, die in sie eingebetteten sendenden oder empfangenden Aktivitäten nebenläufig zueinander auszuführen. Dies ist die beste, in WS-BPEL mögliche Approximation an das gleichzeitige Produzieren oder Konsumieren von Marken in einem Petrinetz. Garantieren können wir durch unser Vorgehen, dass alle zu empfangenden Nachrichten eingegangen sein müssen, bevor Nachrichten gesendet werden und dass alle Nachrichten gesendet wurden, bevor durch andere Aktivitäten erneut Nachrichten gesendet oder empfangen werden können.

Neben den gerade genannten Fällen sind noch drei weitere Kombinationsmöglichkeiten möglich. Codebeispiel 3.8 zeigt ein Beispiel für mehrere empfangende und sendende Ak-

tivitäten an einer Transition, Codebeispiel 3.9 eines für eine empfangende und mehrere sendende Aktivitäten und Codebeispiel 3.10 eines für mehrere empfangende und eine sendende Aktivität. Wie zuvor im Beispiel in Abb. 3.2(c) werden alle empfangenden Aktivitäten vor allen sendenden aufgerufen.

```

<flow>
  <receive "A" />
  <receive "B" />
  ...
</flow>
<flow>
  <invoke "X" />
  <invoke "Y" />
  ...
</flow>

```

Codebeispiel 3.8: Ein Beispiel für mehrere empfangende und sendende Aktivitäten.

```

<receive "A" />
<flow>
  <invoke "X" />
  <invoke "Y" />
  ...
</flow>

```

Codebeispiel 3.9: Ein Beispiel für eine empfangende und mehrere sendende Aktivitäten.

```

<flow>
  <receive "A" />
  <receive "B" />
  ...
</flow>
<invoke "X" />

```

Codebeispiel 3.10: Ein Beispiel für mehrere empfangende und eine sendende Aktivität.

Da es sich hierbei um die ersten Transformationen des oWFN handelt, die Annotationen erzeugen und wir jede Transition des oWFN maximal einmal auf diese Art umwandeln können, müssen wir uns noch keine Gedanken darüber machen, dass an den Transitionen bereits Annotationen vorhanden seien könnten. Da nach der Ausführung dieser Transformation alle Interfaceplätze aus dem oWFN entfernt sind, brauchen wir diese Transformation nicht erneut durchzuführen. Die folgenden Transformationen werden wir im Gegensatz dazu immer wieder auf ihre Anwendbarkeit hin testen.

3.2.2. Sequenzen

In diesem Abschnitt betrachten und ersetzen wir Sequenzen in der Struktur des oWFN. Später in Abschnitt 3.3.5 werden wir die WS-BPEL-Aktivität `sequence` nutzen, um sequenzielle Ausführungen von Aktivitäten im WS-BPEL-Code einzubetten. Im WS-BPEL-Code sind alle Aktivitäten mit Ausnahme der Aktivität `process` in strukturierte Aktivitäten eingebettet. Sobald zwei oder mehr Aktivitäten nacheinander ausgeführt werden sollen, müssen sie in die Aktivität `sequence` oder in die ebenfalls strukturierte Aktivität `flow` eingebettet und im letzteren Fall mit Hilfe von Links verbunden werden. Da wir im Verlauf unserer Transformation in den Annotationen der Elemente des oWFN häufiger Aktivitäten aneinanderreihen, umsordieren, ersetzen, entfernen und unter Umständen auch in die Aktivität `flow` einbetten, verzichten wir darauf, bei jedem Auftreten

einer Sequenz von Aktivitäten, diese in eine **sequence**-Aktivität einzubetten. Stattdessen werden wir erst zum Abschluss unserer Transition alle dann noch verbliebenen Aneinanderreihungen von WS-BPEL-Aktivitäten einbetten, wenn keine weiteren Veränderungen mehr vorgenommen werden. Dies hilft zudem dabei kompakteren WS-BPEL-Code zu generieren, in dem keine überflüssigen oder mehrfach ineinander geschachtelten **sequence**-Aktivitäten vorkommen.

Dieses Vorgehen ermöglicht es uns zudem die Transformationen zu verwenden, die wir in diesem Abschnitt vorstellen werden. Wir definieren Transformationen für die Reduktion von Sequenzen im oWFN mit minimaler Länge und führen diese für längere Sequenzen einfach mehrfach aus. Eine Alternative dazu wäre eine Transformationsvorschrift gewesen, die jeweils die Sequenzen mit maximaler Länge im oWFN umwandelt.

Diese und spätere Umformungen des oWFN sind ähnlich den in [Mur89] beschriebenen strukturellen Reduktionsregeln für Petrinetze. Unsere Absicht bei der Ersetzung von Teilnetzen des oWFN ist jedoch in erster Linie nicht die strukturelle Reduktion des Netzes, sondern das Finden einer Repräsentation des ersetzten Teilnetzes durch WS-BPEL-Aktivitäten.

Transformation 2

Wir betrachten zuerst eine Sequenz aus einem Platz, einer Transition und einem weiteren Platz und gehen anschließend kurz auf die weiteren drei Fälle ein, die alle dem gleichen Aufbau folgen. Seien $p_1, p_2 \in P$ und $t_1 \in T$. Wenn gilt, dass

- $p_1 \neq p_2$,
- $p_1^\bullet = \{t_1\}$,
- ${}^\bullet p_2 = \{t_1\}$,
- ${}^\bullet t_1 = \{p_1\}$ und
- $t_1^\bullet = \{p_2\}$,

dann können wir p_1, p_2 und t_1 ersetzen durch einen neuen annotierten Platz p_0 . Für diesen gilt:

- ${}^\bullet p_0 = {}^\bullet p_1$ und
- $p_0^\bullet = p_2^\bullet$.

p_1, p_2 und t_1 werden aus dem oWFN entfernt. Die Annotation am neuen Platz p_0 setzt sich zusammen aus den Annotationen von p_1, t_1 und p_2 . Abbildung 3.3(a) zeigt ein Beispiel mit abgekürzter Annotation für diese Transformation. Auf der linken Seite sind die Transition t_1 und die zwei Plätze p_1 und p_2 vor der Ersetzung zu sehen und auf der rechten Seite findet sich der neue Platz p_0 mit den Annotationen der ursprünglichen drei Elemente.

Wie wir bereits in Abschnitt 3.1 angesprochen haben, fügen wir als Annotation für die entfernte Transition t_1 die Aktivität **opaqueActivity** ein, sollte t_1 bisher nicht annotiert worden sein. Gleiches geschieht in den folgenden drei Fällen beim Entfernen von Transitionen ohne Annotationen.

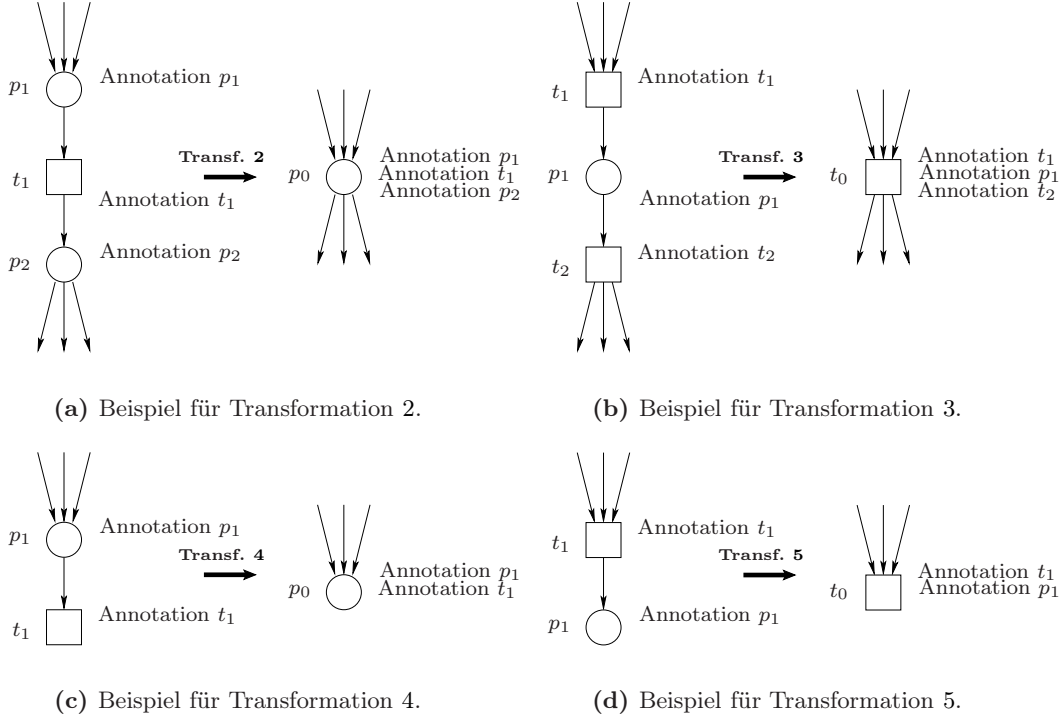


Abbildung 3.3.: Beispiele für die Transformation von Sequenzen.

Transformation 3

Als nächstes betrachten wir eine Sequenz aus einem Platz, einer Transition und einem weiteren Platz. Seien $t_1, t_2 \in T$ und $p_1 \in P$. Wenn gilt, dass

- $t_1 \neq t_2$,
- $t_1^\bullet = \{p_1\}$,
- ${}^\bullet t_2 = \{p_1\}$,
- ${}^\bullet p_1 = \{t_1\}$ und
- $p_1^\bullet = \{t_2\}$,

dann können wir t_1, t_2 und p_1 ersetzen durch eine neue annotierte Transition t_0 . Für diesen gilt:

- ${}^\bullet t_0 = {}^\bullet t_1$ und
- $t_0^\bullet = t_2^\bullet$.

t_1, t_2 und p_1 werden aus dem oWFN entfernt. Die Annotation an der neuen Transition t_0 setzt sich zusammen aus den Annotationen von t_1, p_1 und t_2 . Abbildung 3.3(b) zeigt ein Beispiel mit abgekürzter Annotation für diese Transformation.

Transformation 4

Unsere dritte Transformation behandelt eine Sequenz aus einem Platz und einer Transition deren Nachbereich leer ist. Seien $p_1 \in P$ und $t_1 \in T$. Wenn gilt, dass

- $p_1^\bullet = \{t_1\}$,
- ${}^\bullet t_1 = \{p_1\}$ und
- $t_1^\bullet = \emptyset$,

dann können wir p_1 und t_1 ersetzen durch einen neuen annotierten Platz p_0 . Für diesen gilt:

- ${}^\bullet p_0 = {}^\bullet p_1$ und
- $p_0^\bullet = \emptyset$.

p_1 und t_1 werden aus dem oWFN entfernt. Die Annotation am neuen Platz p_0 setzt sich zusammen aus den Annotationen von p_1 und t_1 . Abbildung 3.3(c) zeigt ein Beispiel mit abgekürzter Annotation für diese Transformation.

Transformation 5

Die letzte Transformation für diesen Abschnitt widmet sich einer Sequenz aus einer Transition und einem Platz dessen Nachbereich leer ist. Seien $p_1 \in P$ und $t_1 \in T$. Wenn gilt, dass

- $t_1^\bullet = \{p_1\}$,
- ${}^\bullet p_1 = \{t_1\}$ und
- $p_1^\bullet = \emptyset$,

dann können wir p_1 und t_1 ersetzen durch eine neue annotierte Transition t_0 . Für diese gilt:

- ${}^\bullet t_0 = {}^\bullet t_1$ und
- $t_0^\bullet = \emptyset$.

p_1 und t_1 werden aus dem oWFN entfernt. Die Annotation an der neuen Transition t_0 setzt sich zusammen aus den Annotationen von t_1 und p_1 . Abbildung 3.3(d) zeigt ein Beispiel mit abgekürzter Annotation für diese Transformation.

Diese vier Transformationen sind ausreichend, um im Verlauf der Transformation alle auftauchenden Sequenzen zu reduzieren. Sequenzen mit mehr als drei Elementen werden in mehreren Schritten umgeformt. Dabei spielt die Reihenfolge, in der die Transformationen ausgeführt werden, keine Rolle. Wir erhalten die gleiche Annotation egal ob bei einer längeren Sequenz zuerst Transformation 2 oder zuerst Transformation 3 angewendet wird. Das Beispiel in Abb. 3.4 verdeutlicht dies. Abgebildet sind zwei der möglichen zweischrittigen Transformationsabläufe. In der oberen Abfolge werden zuerst p_1, t_1 und p_2 zu einem neuen Platz p_0 reduziert und anschließend p_0, t_2 und p_3 erneut reduziert. In der unteren Abfolge werden zuerst t_1, p_2 und t_2 zu einer neuen Transition t_0 reduziert und

anschließend p_1, t_0 und p_3 erneut reduziert. Das Ergebnis ist in allen möglichen Abläufen das gleiche.

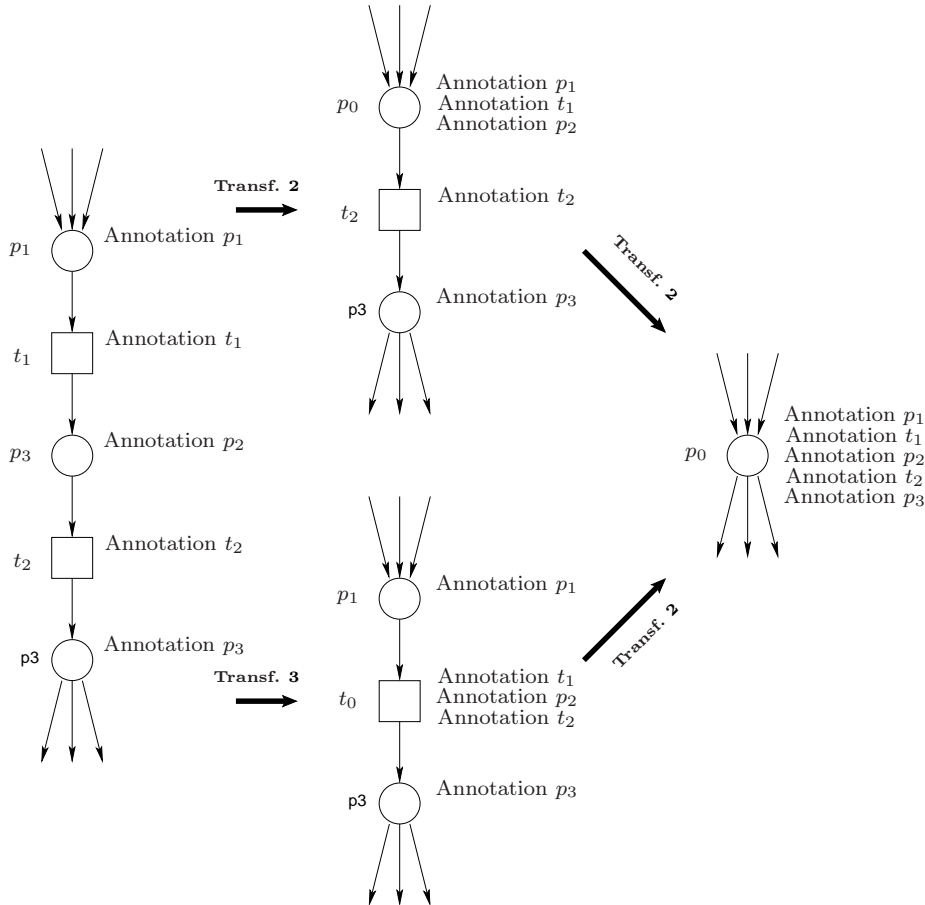


Abbildung 3.4.: Beispiele für verschiedene mögliche Transformationsreihenfolgen.

Die letzten beiden Transformationsvarianten werden insbesondere dann notwendig, wenn das gesamte oWFN auf nur noch zwei Knoten reduziert wurde. Genauso gut, wie diese Transformationen, die Sequenzen mit einem Knoten mit leerem Nachbereich voraussetzen, hätten wir uns stattdessen auch für zwei Transformationen, die Sequenzen reduzieren, die mit einem Knoten mit leerem Vorbereich beginnen, entscheiden können. Bei den späteren Transformationen machen wir uns die hier getroffene Entscheidung zu nutze. In Abschnitt 3.2.3 stellen wir eine Umformung vor, die Transitionen mit leerem Nachbereich benötigt und in Abschnitt 3.2.4 eine Umformung mit Plätzen ohne Nachbereich. Die Ausführung der in diesem Abschnitt eingeführten Transformationen ermöglicht die häufigere Anwendung der folgenden.

3.2.3. Bedingtes Verhalten

Wenn sich in einem Petrinetz der Vorbereich mehrerer Transitionen überschneidet, sprechen wir davon, dass die Transitionen zueinander in Konflikt um die Marken [Sta90] auf den gemeinsamen Plätzen in ihren Vorbereichen stehen. In einem Low-Level Petrinetz ohne Erweiterungen gibt es keine Präferenz für die Auflösung eines solchen Konfliktes. Abbildung 3.5 zeigt ein Beispiel für zwei Transitionen (t_1 und t_2), die in Konflikt um die Marken auf einem Platz (p_1) stehen.

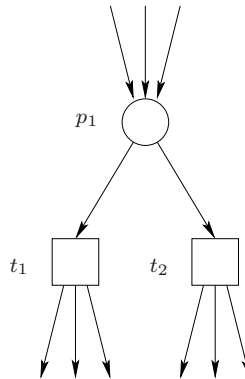


Abbildung 3.5.: Beispiel für einen Konflikt.

Es existieren Erweiterungen für Petrinetze, die Bedingungen, als *Guards* [Jen92] bezeichnet, für das Schalten von Transitionen definieren. Nur wenn diese den Transitionen zugeordneten Bedingungen erfüllt sind, können die dazugehörigen Transitionen schalten. Die von uns betrachteten oWFN enthalten diese Guards nicht, so dass keine der in Konflikt stehenden Transitionen den Vorrang hat. Eine Transformation von oWFN, deren Transitionen Guards zugeordnet sind, ist ebenfalls denkbar. Dabei können die Bedingungen der Guards direkt in die Bedingungen der dazugehörigen Verzweigung im WS-BPEL-Code übertragen. Da diese Bedingungen häufig an Daten, deren Zugriff in WS-BPEL erst ermöglicht werden muss und andere Prozessinformationen geknüpft ist, werden wir an dieser Stelle auf die Details eines solchen Vorgehens verzichten. Wir erzeugen dem Nutzer lediglich eine entsprechende Verzweigung in WS-BPEL und überlassen es ihm, dort die passenden Bedingungen für die Präferenz bei der Auswahl einzutragen.

Korrektur 1

Wie wir in Abschnitt 2.2.4 gesehen haben, stellt WS-BPEL die zwei strukturierten Aktivitäten `if` und `pick` zur bedingten Auswahl von Zweigen im Ablauf zur Verfügung. Wir entscheiden uns an dieser Stelle nicht für eine der beiden strukturierten Aktivitäten, sondern für eine Kombination aus beiden. Da die Aktivität `pick` nur eingesetzt werden kann, wenn mindestens einer der zur Auswahl stehenden Abläufe den Empfang einer Nachricht voraussetzt, werden wir `pick` auch nur in genau diesen Fällen einsetzen. Dies geschieht, indem in einem ersten Schritt erkanntes bedingtes Verhalten mit einem `if` annotiert wird. Am Ende der Transformation, wenn das gesamte oWFN bereits in unsere

Annotationen umgewandelt wurde, werden dann die Zweige aller `if` Annotationen überprüft. Befindet sich in einem der Zweige als erste Aktivität ein `receive`, welches nicht Ziel eines Links ist, dann wird das `if` durch ein `pick` ersetzt. In diesem `pick` werden die `receive`-Aktivitäten, die die erste Aktivität eines Zweiges darstellen und ebenfalls nicht Ziele von Links sind, aus diesen Zweigen entfernt und ihre Eigenschaften werden auf jeweils ein `<onMessage>`-Element übertragen, das mit diesem Zweig verknüpft wird. Die restlichen Zweige bleiben unverändert und werden je mit einem `<onAlarm>`-Element verbunden. Codebeispiel 3.11 zeigt ein Beispiel für eine `if` Annotation, in der es sich bei der ersten Aktivität der beiden Zweige 2 und n jeweils um ein `receive` handelt. In dem Beispiel sollen die beiden `receive`-Aktivitäten die Nachrichten „A“ und „B“ empfangen. Alle anderen Zweige der `if`-Aktivität beginnen mit beliebigen WS-BPEL-Aktivitäten außer der `receive`-Aktivität. Nach der Umwandlung, deren Ergebnis in Codebeispiel 3.12 zu sehen ist, sind die beiden `receive`-Aktivitäten aus den Zweigen 2 und n entfernt worden. Statt ihrer übernehmen nun die `<onMessage>`-Elemente den Empfang der Nachrichten „A“ und „B“. Die restlichen Aktivitäten der Zweige blieben dabei unverändert. Die `if` Annotation wurde in eine `pick` Annotation umgewandelt. Dazu wurde die Position von Zweig 1 verändert, da die `<onMessage>`-Elemente vor den `<onAlarm>`-Elementen aufgeführt werden.

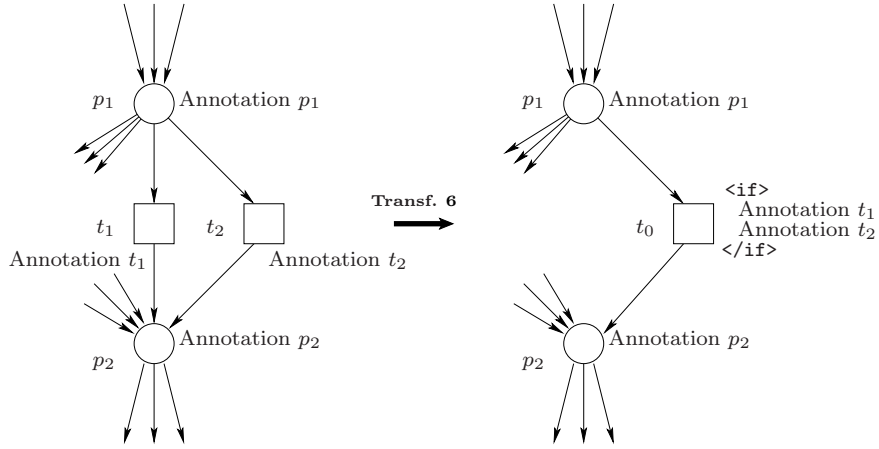
| | |
|---|---|
| <pre> <if> Zweig 1 mit beliebigen Aktivitäten Zweig 2 mit Annotation: <receive "A" /> beliebige Aktivitäten ... Zweig n mit Annotation: <receive "B" /> beliebige Aktivitäten ... Zweig x mit beliebigen Aktivitäten </if> </pre> | <pre> <pick> <onMessage "A"> Zweig 2 mit Annotation: beliebige Aktivitäten </onMessage> ... <onMessage "B"> Zweig n mit Annotation: beliebige Aktivitäten </onMessage> ... <onAlarm> Zweig x </onAlarm> <onAlarm> Zweig 1 </onAlarm> </pick> </pre> |
|---|---|

Codebeispiel 3.11: Eine `if` Annotation, die in eine `pick` Annotation umgewandelt werden kann.

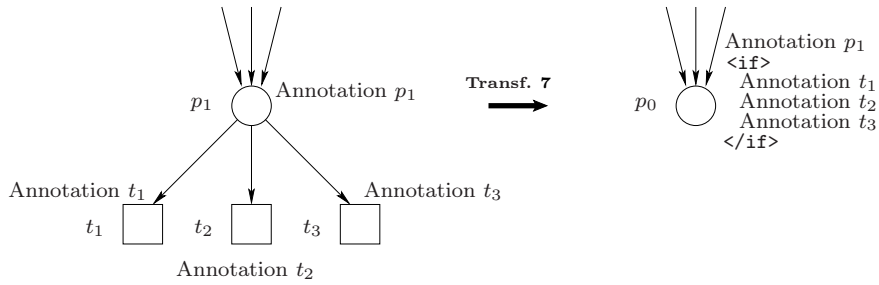
Codebeispiel 3.12: Die neue `pick` Annotation nach Anwendung von Korrektur 1.

Bei unserer Transformation können wir Suchmuster im oWFN nur ersetzen und annotieren, wenn wir für sie eine abgeschlossene WS-BPEL-Aktivität finden können. Im Fall des bedingten Verhaltens bedeutet dies, dass wir sowohl den Anfang, als auch das Ende der gesamten Auswahl und jedes ihrer Zweige kennen müssen. Anderenfalls würden wir den WS-BPEL-Code für die Aktivität `if` nur öffnen können, ohne zu wissen, an welcher Stelle die Aktivität später wieder geschlossen werden muss. Im in Abb. 3.6(a) dargestellten Beispiel von bedingtem Verhalten in einem oWFN bilden auf der linken Seite die Plätze p_1 und p_2 Anfang und Abschluss des Konstrukts. An dieser Stelle können wir demnach

mehrere Elemente des oWFN durch die Annotation mit einer geschlossenen `if`-Aktivität ersetzen, wie auf der rechten Seite der Abbildung zu sehen ist.



(a) Beispiel für die allgemeine Transformation 6.



(b) Beispiel für Transformation 7 im Spezialfall.

Abbildung 3.6.: Beispiele für die Transformation bedingter Verzweigungen.

Transformation 6

Wir betrachten nun die Transformation, die zu dieser Ersetzung führt. Anschließend stellen wir einen Spezialfall der Transformation vor, der besonderer Voraussetzungen bedarf. Bei diesem allgemeinen Fall können sich im Nachbereich, des Platzes der den Anfang und im Vorbereich des Platzes, der den Abschluss des bedingten Verhaltens bildet, noch weitere Transitionen befinden, die an unserer Transformation nicht beteiligt sind. Seien $p_1, p_2 \in P$, $t_1, t_2, \dots, t_n \in T$ und $n \in \mathbb{N}$. Wenn gilt, dass

- $p_1 \neq p_2$,
- $n > 1$,
- $t_1, t_2, \dots, t_n \in p_1^\bullet$,
- $t_1, t_2, \dots, t_n \in \bullet p_2$,

- $\bullet t_1 = \bullet t_2 = \dots = \bullet t_n = \{p_1\}$ und
- $t_1^\bullet = t_2^\bullet = \dots = t_n^\bullet = \{p_2\}$,

dann können wir t_1, t_2, \dots, t_n ersetzen durch eine neue annotierte Transition t_0 . Für diese gilt:

- $\bullet t_0 = \{p_1\}$ und
- $t_0^\bullet = \{p_2\}$.

t_1, t_2, \dots, t_n werden aus dem oWFN entfernt. Die Annotation in Codebeispiel 3.13 entspricht der Annotation an der neuen Transition t_0 . Die bereits angesprochene Abb. 3.6(a) zeigt ein Beispiel mit abgekürzter Annotation für diese Transformation mit zwei Transitionen. Die Annotationen der beiden Transitionen auf der linken Seite des Beispiels werden zu jeweils einem Zweig in der auf der rechten Seite neu eingefügten `if`-Aktivität.

```
<if>
  Zweig mit der Annotation von  $t_1$ 
  Zweig mit der Annotation von  $t_2$ 
  ...
  Zweig mit der Annotation von  $t_n$ 
</if>
```

Codebeispiel 3.13: Die Annotation von t_0 nach Transformation 6.

Durch spätere Transformationen können weitere Teilnetze auf einzelne Transitionen zwischen den Plätzen p_1 und p_2 reduziert werden. In diesem Fall kann die Transformation erneut angewendet werden und transformiert die neu gefundene Transition zusammen mit der früher eingefügten Transition t_0 . Codebeispiel 3.14 zeigt ein Beispiel für eine Annotation, die dabei letztendlich entstehen kann. Wären die Transformationen in einer anderen Reihenfolge ausgeführt worden, so dass später die Transformation nur einmal angewendet worden wäre, wäre eine Annotation wie in Codebeispiel 3.15 zustande gekommen, in der nur eine `if`-Aktivität verwendet worden wäre.

```
<if>
  Zweig mit der Annotation von  $t_i$ 
  Zweig mit Annotation:
    <if>
      Zweig mit der Annotation von  $t_j$ 
      Zweig mit der Annotation von  $t_k$ 
    </if>
</if>
```

Codebeispiel 3.14: Eine Annotation, auf die Korrektur 2 anwendbar ist.

```
<if>
  Zweig mit der Annotation von  $t_i$ 
  Zweig mit der Annotation von  $t_j$ 
  Zweig mit der Annotation von  $t_k$ 
</if>
```

Codebeispiel 3.15: Die Annotation nach Anwendung von Korrektur 2.

Korrektur 2

Um bei unserer Transformation für bedingtes Verhalten stets den gleichen WS-BPEL-Code zu erzeugen und diesen kompakt zu halten, fügen wir eine Korrektur den Transfor-

mationsregeln hinzu. Wie bereits beim nachträglichen Umwandeln von `if` Annotationen in `pick` Annotationen durch Korrektur 1, findet diese Korrektur erst statt, wenn das gesamte oWFN in eine Annotation umgewandelt wurde. Dann werden die Zweige aller `if` Annotationen überprüft. Befindet sich in einem der Zweige als einzige Aktivität eine weitere `if`-Aktivität und ist dieses nicht Ziel von Links, dann wird diese `if`-Aktivität mit dem Umgebenden zusammengelegt. Dies geschieht, indem die Zweige der inneren `if`-Aktivität als zusätzliche Zweige an die Umgebende angehängt werden. So wird aus der Annotation in Codebeispiel 3.14 die Annotation in Codebeispiel 3.15, und wir haben für einen Fall von bedingtem Verhalten im oWFN genau eine `if`-Aktivität in den WS-BPEL-Code eingefügt. Wir werden Korrektur 2 vor der zuvor angesprochenen Korrektur 1 ausführen. So werden mehrere `if` Annotationen zusammengefasst, bevor sie in `pick` Annotationen umgewandelt werden.

Transformation 7

Abschließend betrachten wir noch den Spezialfall, dass wir einen Platz auffinden können, in dessen Nachbereich sich bedingtes Verhalten befindet, der Nachbereich der beteiligten Transitionen in diesem jedoch leer ist. Seien $p_1 \in P$, $t_1, t_2, \dots, t_n \in T$ und $n \in \mathbb{N}$. Wenn gilt, dass

- $n > 1$,
- $p_1^\bullet = \{t_1, t_2, \dots, t_n\}$,
- ${}^\bullet t_1 = {}^\bullet t_2 = \dots = {}^\bullet t_n = \{p_1\}$ und
- $t_1^\bullet = t_2^\bullet = \dots = t_n^\bullet = \emptyset$,

dann können wir p_1 und t_1, t_2, \dots, t_n ersetzen durch einen neuen annotierten Platz p_0 . Für diesen gilt:

- ${}^\bullet p_0 = {}^\bullet p_1$ und
- $p_0^\bullet = \emptyset$.

p_1 und t_1, t_2, \dots, t_n werden aus dem oWFN entfernt. Codebeispiel 3.16 zeigt die Annotation des neuen Platzes p_0 und Abb. 3.6(b) zeigt ein Beispiel mit abgekürzter Annotation für diese Transformation diesmal mit drei Transitionen. Wie am Beispiel zu sehen ist, können wir den Abschluss des bedingten Verhaltens in diesem Fall an den offenen Enden aller Zweige festmachen.

Wie zuvor handelt es sich bei unseren Annotationen lediglich um einige Schlüsselworte, die wir später noch verändern können und zum Abschluss der Transformation durch WS-BPEL-Code ersetzen. Im Fall eines `if` wird der erste Zweig als eingebettete Aktivität in das `<if>`-Element eingefügt, weitere Zweige in `<elseif>`-Elemente und der letzte Zweig in ein `<else>`-Element. Codebeispiel 3.17 zeigt den WS-BPEL-Code, der aus einer `if` Annotation entstanden ist. Der Nutzer muss später die Bedingungen für die Auswahl der Zweige festlegen. Er kann dazu die Reihenfolge der Elemente abändern und selber eines der Elemente zum `<if>`-Element und eines zum `<else>`-Element bestimmen. Die noch einzutragenden Bedingungen sind durch das für abstrakte WS-BPEL-Prozesse definierte

Element `<condition opaque="yes"/>` gekennzeichnet. Im Namen der `if`-Aktivität haben wir den Namen des Platzes, an dem das bedingte Verhalten im oWFN anzutreffen war, mit eingefügt. So ist eine spätere Zuordnung zwischen Elementen des transformierten oWFN und des erzeugten WS-BPEL-Codes leicht möglich. Wie bei der vorherigen Transformationen fügen wir als Annotation für eine entfernte Transition $t_i \in \{t_1, t_2, \dots, t_n\}$ die Aktivität `opaqueActivity` ein, sollte t_i bisher nicht annotiert worden sein. Dies verhindert bei den beiden gerade vorgestellten Transformationen zudem, dass wir einen Zweig in einer `if`-Aktivität erzeugen, der keine Aktivitäten einbettet.

| | |
|---|--|
| <pre>Annotation von p_1 <if> Zweig mit der Annotation von t_1 Zweig mit der Annotation von t_2 ... Zweig mit der Annotation von t_n </if></pre> | <pre><if name="Act_if_p_1"> <condition opaque="yes" /> Aktivitäten von t_1 <elseif> <condition opaque="yes" /> Aktivitäten von t_2 </elseif> ... <elseif> <condition opaque="yes" /> Aktivitäten von t_{n-1} </elseif> <else> Aktivitäten von t_n </else> </if></pre> |
| <p>Codebeispiel 3.16: Die Annotation von p_0 nach Transformation 7.</p> | <p>Codebeispiel 3.17: Der spätere WS-BPEL-Code einer <code>if</code> Annotation.</p> |

Die in diesem Abschnitt eingeführten Transformationsregeln wandeln bedingtes Verhalten im oWFN in Annotationen um, wenn zu einem Konflikt im oWFN auch der passende Abschluss, sei es durch einen weiteren Platz oder durch Transitionen mit leerem Nachbereich gefunden werden kann und die beteiligten Transitionen keine zusätzlichen Plätze im Vor- und Nachbereich haben. Wir werden in Abschnitt 3.2.4 noch einmal bedingtes Verhalten betrachten, wenn wir größere Teilnetze mit nebenläufigen Ausführungen transformieren. Bei dieser Betrachtung werden dann alle weiteren Fälle berücksichtigt.

3.2.4. Nebenläufige Ausführungen

Da es sich bei Petrinetzen um nebenläufige Systeme handelt, ist es möglich, dass nebenläufige Ausführungen auch in einem oWFN auftreten. Im Beispiel in Abb. 3.7 werden die Transitionen t_2 und t_3 nebenläufig zueinander ausgeführt und die Zweige der Ausführungen werden erst durch die Transition t_4 wieder synchronisiert.

Während der Transformation des Interfaces in Abschnitt 3.2.1 haben wir bereits die `flow`-Aktivität benutzt, um den gleichzeitigen Empfang und Versand mehrerer Nachrichten in WS-BPEL nachzubilden. Mit Hilfe der Aktivität `flow` und der Linksemantik sind verschiedenste Abläufe in WS-BPEL ausdrückbar. Sichere und zyklensfreie Petrinetze sind vollständig in eine `flow`-Aktivität umwandelbar, selbst wenn sie keine nebenläufige Ausführungen enthalten. Diese `flow`-Aktivität würde jedoch in den meisten Fällen unnötig

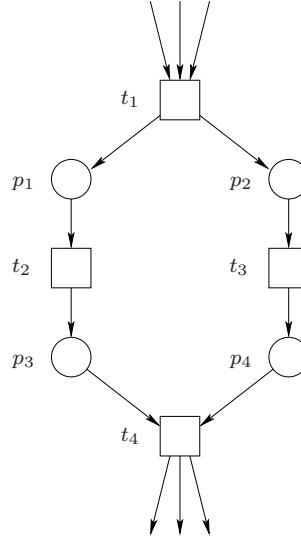


Abbildung 3.7.: Beispiel für nebenläufige Ausführungen.

viele Links beinhalten und dadurch für den Nutzer unverständlich sein können. Aus diesem Grund werden wir die Aktivität `flow` nur dann verwenden, wenn wir nebenläufige Ausführungen abbilden müssen oder eine Nachbildung des Prozesses durch die anderen WS-BPEL-Aktivitäten nicht möglich ist. Des Weiteren werden wir versuchen erst `flow`-Aktivitäten zu erzeugen, die ganz ohne Links auskommen und die Anzahl der Links, die wir dennoch erzeugen müssen, durch spätere Korrekturen wieder zu verringern.

Transformation 8

Die erste Transformation ersetzt eine abgeschlossene nebenläufige Ausführung. Für jeden Zweig der Ausführung wird ein Zweig an die neu eingefügte `flow`-Aktivität angehängt. Da es in diesem Fall keine Abhängigkeiten zwischen den Aktivitäten der einzelnen Zweige gibt, sind an dieser Stelle keine Links notwendig. Alle Aktivitäten innerhalb der Zweige werden weiterhin sequentiell ausgeführt, so können wir vollständig auf Links verzichten. Die Synchronisation aller Zweige findet durch das Beenden der `flow`-Aktivität statt. Seien $t_1, t_2 \in T$, $p_1, p_2, \dots, p_n \in P$ und $n \in \mathbb{N}$. Wenn gilt, dass

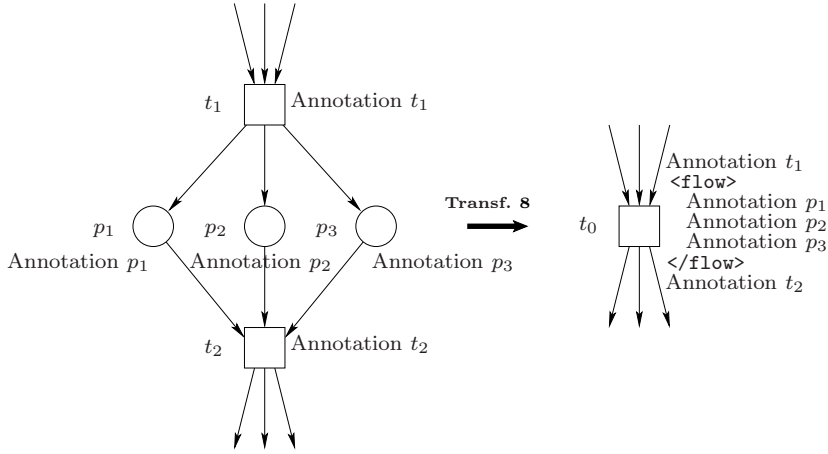
- $t_1 \neq t_2$,
- $n > 1$,
- $\bullet p_1 = \bullet p_2 = \dots = \bullet p_n = \{t_1\}$,
- $p_1 \bullet = p_2 \bullet = \dots = p_n \bullet = \{t_2\}$ und
- $t_1 \bullet = \bullet t_2 = \{p_1, p_2, \dots, p_n\}$

dann können wir t_1, t_2 und p_1, p_2, \dots, p_n ersetzen durch eine neue annotierte Transition t_0 . Für diese gilt:

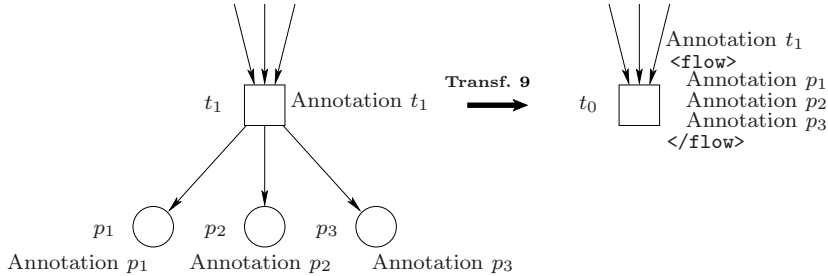
- $\bullet t_0 = \bullet t_1$ und

- $t_0^\bullet = t_2^\bullet$.

t_1, t_2 und p_1, p_2, \dots, p_n werden aus dem oWFN entfernt. Die Annotation in Codebeispiel 3.18 entspricht der Annotation an der neuen Transition t_0 . Abbildung 3.8(a) zeigt ein Beispiel mit abgekürzter Annotation für diese Transformation mit drei Plätzen. Die Annotationen der drei Plätze auf der linken Seite des Beispiels werden zu jeweils einem Zweig in der auf der rechten Seite neu eingefügten flow-Aktivität.



(a) Beispiel für die allgemeine Transformation 8.



(b) Beispiel für Transformation 9 im Spezialfall.

Abbildung 3.8.: Beispiele für die Transformation parallelen Verhaltens.

Transformation 9

Ähnlich wie in Abschnitt 3.2.3 betrachten wir auch an dieser Stelle noch einen Spezialfall, bei dem sich im Nachbereich der Plätze keine weiteren Transitionen befinden. Seien $t_1 \in T$, $p_1, p_2, \dots, p_n \in P$ und $n \in \mathbb{N}$. Wenn gilt, dass

- $n > 1$,
- $\bullet p_1 = \bullet p_2 = \dots = \bullet p_n = \{t_1\}$,
- $p_1^\bullet = p_2^\bullet = \dots = p_n^\bullet = \emptyset$ und

Annotation von t_1
<flow>
 Zweig mit der Annotation von p_1
 Zweig mit der Annotation von p_2
 ...
 Zweig mit der Annotation von p_n
</flow>
Annotation von t_2

Codebeispiel 3.18: Die Annotation von t_0 nach Transformation 8.

$$- t_1^\bullet = \{p_1, p_2, \dots, p_n\}$$

dann können wir t_1 und p_1, p_2, \dots, p_n ersetzen durch eine neue annotierte Transition t_0 . Für diese gilt:

- ${}^\bullet t_0 = {}^\bullet t_1$ und
- $t_0^\bullet = \emptyset$.

t_1 und p_1, p_2, \dots, p_n werden aus dem oWFN entfernt. Die Annotation in Codebeispiel 3.19 entspricht der Annotation an der neuen Transition t_0 . Abbildung 3.8(b) zeigt ein Beispiel mit abgekürzter Annotation für diese Transformation mit drei Plätzen. Auch hier, können wir den Abschluss der nebenläufigen Ausführung an den offenen Enden aller Zweige festmachen.

Annotation von t_1
<flow>
 Zweig mit der Annotation von p_1
 Zweig mit der Annotation von p_2
 ...
 Zweig mit der Annotation von p_n
</flow>

Codebeispiel 3.19: Die Annotation von t_0 nach Transformation 9.

Anders als bei den bisherigen Transformationen annotieren wir bei den beiden gerade vorgestellten auch Plätze mit der Aktivität `opaqueActivity`, sollten sie bisher nicht annotiert worden sein. Dies kann nur geschehen, wenn sich in einem der Zweige zuvor niemals eine Transition befunden hat, denn diese wäre durch die anderen Transformationen annotiert worden. Das bedeutet, dass in diesem Zweig beim Erstellen des oWFN ein einzelner Platz eingefügt wurde. Um nun einerseits keinen leeren Zweig in der neu eingefügten Aktivität `flow` zu erzeugen und da wir andererseits der Auffassung sind, dass der Nutzer, der das oWFN erstellt hat, eine spätere Verwendung dieses Platzes beabsichtigen musste, annotieren wir diese Plätze vor der Transformation mit der Aktivität `opaqueActivity`.

Transformation 10

Für die letzte Transformation betrachten wir sichere und zyklensfreie Teilnetze des oWFN und ersetzen sie durch eine einzige **flow**-Aktivität. Die Teilnetze müssen zyklensfrei sein, da wir, wie bereits in Abschnitt 2.2.5 angesprochen, innerhalb einer **flow**-Aktivität mit Hilfe der Links keine Zyklen bilden dürfen und eine Nachbildung eines solchen Teilnetzes durch eine **flow**-Aktivität damit nicht möglich ist. Sicherheit müssen wir für die Teilnetze voraussetzen, um zu verhindern, dass Transitionen im Teilnetz mehrfach schalten können und dies die mehrfache Ausführung der den Transitionen entsprechenden Aktivitäten in WS-BPEL bedeuten würde. Wir werden in den Abschnitten 3.2.6 und 3.2.7 detaillierter auf die Einschränkungen von WS-BPEL bezüglich der mehrfachen Ausführung von Aktivitäten eingehen.

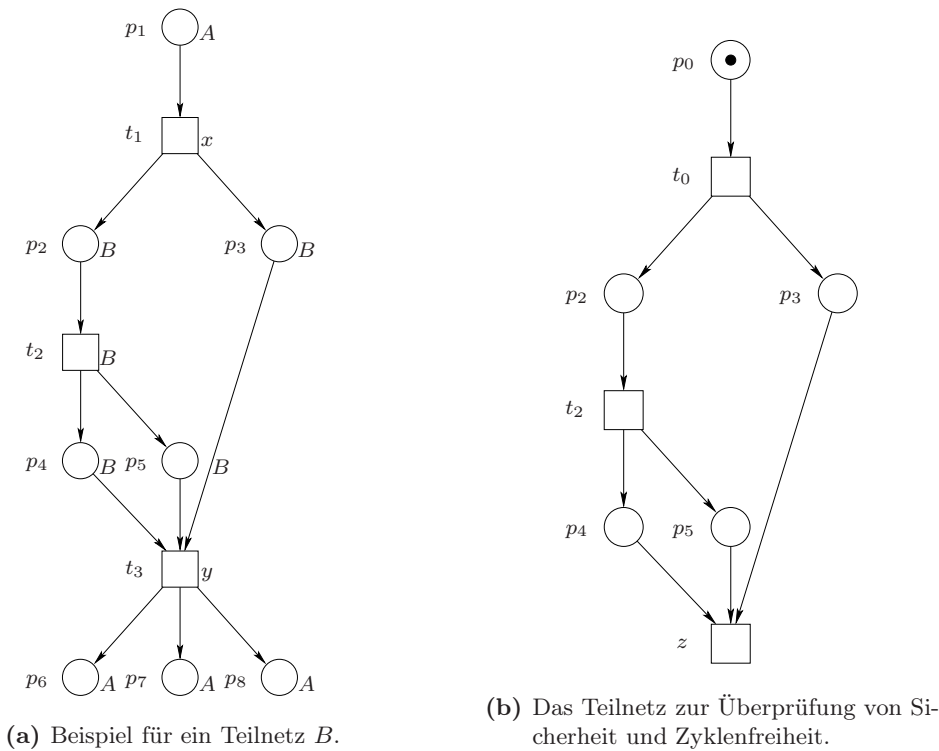


Abbildung 3.9.: Beispiel für ein für Transformation 10 geeignetes Teilnetz.

Wir beginnen unsere Transformation mit der Beschreibung eines geeigneten Teilnetzes B und betrachten dazu zuerst das Beispiel in Abb. 3.9(a). Das Beispiel zeigt neben dem Teilnetz B , das restliche Netz A und die Knoten x und y mit denen wir die Teilnetze voneinander abgrenzen. Drei verschiedene Arten von Teilnetzen sind für die Transformation geeignet. Als erstes betrachten wir den im Beispiel dargestellten Fall. Mit $|B|$ wollen wir die Anzahl der Knoten im Teilnetz B bezeichnen. Seien dazu $x, y \in P \cup T$ und $A, B \subseteq P \cup T$ mit

$$- x \neq y,$$

- $A \cap B = \emptyset$,
- $x, y \notin A \cup B$,
- $A \cup B \cup \{x\} \cup \{y\} = P \cup T$ und
- $|B| \geq 3$.

Für ein geeignetes Teilnetz B muss gelten:

1. $\forall b \in B: b \notin A^\bullet$,
2. $\forall a \in A: a \notin B^\bullet$,
3. $\forall b \in B: b \in \text{Pfad}(x)$,
4. $x \notin B^\bullet$ und
5. $\forall b \in B: b^\bullet \neq \emptyset$.

Aufzählungspunkte 1 und 2 verbieten Kanten zwischen Knoten in A und B . Mit Punkt 3 fordern wir, dass jeder Knoten in B auf einem Pfad liegt, der beim Knoten x beginnt. Durch Punkt 4 schließen wir Zyklen aus und mit Punkt 5 schließlich Knoten ohne Nachbereich innerhalb von B . Warum dies notwendig ist, erläutern wir anhand des Beispiels in Abb. 3.9(a). Nehmen wir weiterhin an, im Nachbereich des Platzes p_4 würde sich der zusätzliche Zweig 1 mit weiteren Transitionen und Plätzen befinden und im Nachbereich von p_6 ein vergleichbarer Zweig 2. Die Knoten von Zweig 1 wären Teil von B und die Knoten von Zweig 2 Teil von A . Mit der Transformation von B würden wir eine **flow**-Aktivität erzeugen, die alle mit den Knoten in B assoziierten Aktivitäten enthält. Eine **flow**-Aktivität ist erst beendet, wenn alle in sie eingebetteten Aktivitäten beendet wurden. Dies bedeutet in diesem Fall, dass Aktivitäten aus Zweig 1 beendet sein müssen, bevor Aktivitäten in Zweig 2 gestartet werden können. Dies würde jedoch nicht den möglichen Abläufen im oWFN entsprechen, in dem die Transitionen aus Zweig 1 nach denen aus Zweig 2 ausgeführt werden können.

Wir beschreiben nun eine weitere Art von Teilnetzen, die Knoten mit leerem Nachbereich gestattet. Hierbei liegen alle Knoten in B auf einem Pfad zu einem Knoten mit leerem Nachbereich. Seien dazu $x \in P \cup T$ und $A, B \subseteq P \cup T$ mit

- $A \cap B = \emptyset$,
- $x \notin A \cup B$,
- $A \cup B \cup \{x\} = P \cup T$ und
- $|B| \geq 3$.

Es muss nun gelten:

- $\forall b \in B: b \notin A^\bullet$,
- $\forall a \in A: a \notin B^\bullet$,
- $\forall b \in B: b \in \text{Pfad}(x)$ und
- $x \notin B^\bullet$.

Als Letztes betrachten wir ein Teilnetz, das das Innere eines Zyklus bildet. Dabei muss es sich bei Knoten x um einen Platz handeln und jeder Pfad in B wieder zu x zurückführen. Seien dazu $x \in P$ und $A, B \subseteq P \cup T$ mit

- $A \cap B = \emptyset$,
- $x \notin A \cup B$,
- $A \cup B \cup \{x\} = P \cup T$ und
- $|B| \geq 3$.

Für ein Teilnetz B muss in diesem Fall gelten:

- $\forall b \in B: b \notin A^\bullet$,
- $\forall a \in A: a \notin B^\bullet$,
- $\forall b \in B: b \in \text{Pfad}(x)$ und
- $\forall b \in B: b^\bullet \neq \emptyset$.

Um auszuschließen, dass einzelne Knoten transformiert werden, betrachten wir nur Teilnetze, die aus mindestens 3 Knoten bestehen. Wir sorgen zudem dafür, dass es zu keiner unnötigen Anwendung dieser Transformation kommt, indem wir die Anwendung dieser Transformation hinter die Anwendung der anderen Transformationen legen werden, die ebenfalls Teilnetze umwandeln können, die die hier geforderten Eigenschaften erfüllen. Da wir die Eigenschaft Sicherheit nur an einem Petrinetz mit einer Anfangsmarkierung überprüfen können, betrachten wir das Teilnetz nun losgelöst vom restlichen oWFN mit zusätzlichen Knoten und einer Anfangsmarkierung. Wir beginnen mit dem Teilnetz B und erweitern es, falls es ein y gibt, um den Knoten z . Für diesen gilt:

- $y \in P \Rightarrow z \in P$,
- $y \in T \Rightarrow z \in T$,
- $\bullet z = \bullet y$,
- $z^\bullet = \emptyset$ und
- $\forall b \in \bullet y \cap B: b^\bullet = (b^\bullet \cup \{z\}) \setminus \{y\}$.

Wenn es sich bei x um einen Platz handelt, erweitern wir das Teilnetz um den mit einer Marke markierten Platz p_0 für den gilt:

- $\bullet p_0 = \emptyset$,
- $p_0^\bullet = x^\bullet \cap B$ und
- $\forall b \in x^\bullet \cap B: \bullet b = (\bullet b \cup \{p_0\}) \setminus \{x\}$.

Wenn es sich bei x jedoch um eine Transition handelt, erweitern wir das Teilnetz um eine Transition t_0 und um den mit einer Marke markierten Platz p_0 für die gilt:

- $\bullet p_0 = \emptyset$,
- $p_0^\bullet = \{t_0\}$,
- $\bullet t_0 = \{p_0\}$,

- $t_0^\bullet = x^\bullet \cap B$ und
- $\forall b \in x^\bullet \cap B : \bullet b = (\bullet b \cup \{t_0\}) \setminus \{x\}$.

Wenn $x \in B^\bullet$, dann kann es sich bei dem Teilnetz nur um den gesamten Inhalt eines Zyklus handeln. In diesem Fall erweitern wir das Petrinetz um den Platz p_z für den gilt:

- $\bullet p_z = \bullet x \cap B$
- $p_z^\bullet = \emptyset$,
- $\forall b \in x^\bullet \cap B : \bullet b = (\bullet b \cup \{p_z\}) \setminus \{x\}$ und
- $\forall b \in \bullet x \cap B : b^\bullet = (b^\bullet \cup \{p_z\}) \setminus \{x\}$.

Dieses vom restlichen Netz A losgelöste Teilnetz B muss nun noch sicher und zyklensfrei sein, damit wir es mit den folgenden Transformationsschritten in eine `flow` Annotation umwandeln können.

Da wir, wie am Anfang dieses Abschnitts begründet, die Anzahl und Komplexität von `flow`-Aktivitäten im erzeugten WS-BPEL-Code möglichst gering halten wollen, werden wir immer nur das Teilnetz transformieren, in dem die längste im Teilnetz mögliche Schaltfolge kürzer oder gleichlang mit den längsten möglichen Schaltfolgen aller anderen diese Definition erfüllenden Teilnetze ist. Nach der Transformation eines Teilnetzes werden zuerst die anderen Transformationen erneut ausgeführt.

Wir werden während dieser Transformation mehrfach die WS-BPEL-Aktivität `empty` in die Annotationen einfügen. Die Aktivität führt keine Funktionalität aus. Ihre in der WS-BPEL Spezifikation [AAA⁺07] vorgesehene Einsatzzwecke sind die Ausführung zur Unterdrückung von Fehlern und die Verwendung als Synchronisationspunkt während der Ausführung einer `flow`-Aktivität. Wir werden die `empty`-Aktivität während unserer Umwandlung in mehreren Transformationen einsetzen, um Transitionen zu kennzeichnen, die wir zusätzlich in das oWFN einfügen. So kann der Nutzer später erkennen, dass es sich bei einer Aktivität nicht um die Repräsentation einer im oWFN vorhandenen Transition handelt. Zudem werden wir in der folgenden Transformation `empty`-Aktivitäten als Quelle und Ziel von Links benutzen. Dazu fügen wir sie vor und hinter die vorhandenen Annotationen von Knoten und als Zweige von bedingtem Verhalten ein. Einige der Korrekturen, die wir vorstellen werden, und die auf den erzeugten Annotationen arbeiten, können nur ausgeführt werden, wenn die betroffenen Aktivitäten nicht Quelle oder Ziel von Links sind. Daher sorgen die hier bei der Erzeugung der Links eingefügten `empty`-Aktivitäten dafür, dass die anderen Aktivitäten nicht zu Quellen und Zielen von Links werden. Korrektur 5 wird später, nachdem die anderen Korrekturen ausgeführt wurden, alle `empty`-Aktivitäten wieder aus den Annotationen entfernen, die nicht für die Synchronisation innerhalb einer `flow`-Aktivität benötigt werden.

Im Folgenden werden wir Aktivierungsbedingungen für alle Aktivitäten festlegen, die Ziel von Links sind. Da wir auch bedingtes Verhalten mit in die `flow`-Aktivität übernehmen, setzen wir das Attribut `suppressJoinFailure` der Aktivität `process` auf den Wert `yes`. Dadurch wird der Fehler `bpel:joinFailure` in jeder Aktivität unterdrückt. Ansonsten würde dieser Fehler auftreten, wenn eine der Aktivierungsbedingungen einer Aktivität innerhalb einer `flow`-Aktivität zu falsch ausgewertet wird. Alternativ dazu könnten wir in jeder

Aktivität, die über eine Aktivierungsbedingung verfügt, das Attribut einzeln auf den Wert *yes* setzen.

Wir widmen uns erneut dem Beispiel in Abb. 3.9(a), das ein unsere Voraussetzungen erfüllendes Teilnetz B zeigt. Transition t_1 ist hierbei x und Transition t_3 ist y . Die Plätze p_1, p_6, p_7 und p_8 bilden somit die Menge A , während p_2, p_3, p_4, p_5 und t_2 die Menge B bilden. In Abb. 3.9(b) ist das Teilnetz zu sehen, an dem die Sicherheit und Zyklensfreiheit getestet wird und auf das die folgenden Transformationsschritte anzuwenden werden. Aus dem Petrinetz wurden A, x und y entfernt und dafür p_0, t_0 und z eingefügt. Mit einer Marke auf p_0 kann die Sicherheit des so entstandenen Petrinetzes überprüft werden.

Betrachten wir nun die einzelnen Schritte der Transformation. In einem ersten Schritt ergänzen wir die Annotationen aller Knoten des Teilnetzes. Wir annotieren jede Transition, die bisher noch nicht annotiert wurde, mit der Aktivität `opaqueActivity`. Wir fügen anschließend vor und nach der Annotation jedes Knoten, der bereits annotiert wurde, jeweils eine `empty` Annotation ein. Alle Plätze, die bisher nicht annotiert wurden, werden mit der Aktivität `empty` annotiert. Diese `empty`-Aktivitäten werden uns später als Quelle und Ziel für Links dienen.

Als nächstes annotieren wir, wie wir es in Abschnitt 3.2.3 angekündigt haben, bedingtes Verhalten innerhalb der `flow`-Aktivität. Dazu betrachten wir alle Plätze, die mehr als eine Transition in ihrem Nachbereich besitzen und fügen für diese `if`-Aktivitäten ein. Seien $p_1 \in P$, $t_1, t_2, \dots, t_n \in T$ und $n \in \mathbb{N}$. Wenn gilt, dass

- $n > 1$,
- $p_1^\bullet = \{t_1, t_2, \dots, t_n\}$ und
- $p_1 \in \bullet t_1, p_1 \in \bullet t_2, \dots, p_1 \in \bullet t_n$

dann können wir Kanten aus dem Teilnetz entfernen. Es gilt:

- $p_1^\bullet = \emptyset$ und
- $\bullet t_1 = \bullet t_1 \cap \{p_1\}, \bullet t_2 = \bullet t_2 \cap \{p_1\}, \dots, \bullet t_n = \bullet t_n \cap \{p_1\}$.

Codebeispiel 3.20 zeigt die neue Annotation von p_1 und Codebeispiel 3.21 die von einem $t_i \in \{t_1, t_2, \dots, t_n\}$. Ähnlich wie bei den Codebeispielen in Abschnitt 3.2.1 stellen die Ausdrücke `<source 'i' />` und `<target 'i' />` eine Kurzschreibweise für die Repräsentation von Quellen und Zielen von Links dar. Links müssen innerhalb einer `flow`-Aktivität einen eindeutigen Namen besitzen. Wir nummerieren an dieser Stelle die Links durch und verwenden die Nummer als eindeutigen Bezeichner. Diese Nummerierung setzen wir bei der nächsten Erzeugung von Links fort. Wir haben also die Annotation des Platzes p_1 um eine `if` Annotation ergänzt, die für jede der Transitionen t_1, t_2, \dots, t_n einen Zweig mit einer neuen `empty` Annotation enthält. Die Kanten zwischen dem Platz und den Transitionen haben wir durch einen Link ersetzt, der eine der `empty`-Aktivitäten als Quelle und die erste Aktivität aus einer der Transitionen, bei denen es sich nach unserem letzten Schritt nun ebenfalls nur noch um `empty`-Aktivitäten handeln kann, als Ziel ersetzt.

Annotation von p_1

```

<if>
  <empty>
    <source "1" />
  </empty>
  <empty>
    <source "2" />
  </empty>
  ...
  <empty>
    <source "n" />
  </empty>
</if>

```

Codebeispiel 3.20: Die Annotation von p_1 .

```

<empty>
  <target "i" />
</empty>
weitere Annotationen von  $t_i$ 

```

Codebeispiel 3.21: Die Annotation eines t_i .

Durch die Ergänzung der Annotation von p_1 um eine if-Aktivität ist nun keine `empty`-Aktivität die letzte Aktivität der Annotation von p_1 mehr. Jedoch gibt es nun von p_1 aus auch keine weiteren ausgehenden Kanten, weswegen wir keine Links erzeugen müssen, die die letzte Aktivität der Annotation als Quelle benutzen müssten. Es bleibt also bei der von uns beabsichtigten Verwendung von `empty`-Aktivitäten als Quelle und Ziel von Links. Damit wir diese Umwandlung vornehmen können, darf die letzte Annotation von p_1 nicht bereits Quelle von Links sein. Es ist jedoch während dieses Teils der Transformation nicht möglich, dass ein Platz ein zweites Mal umgewandelt werden muss oder dass die Annotation bereits Quelle von Links ist.

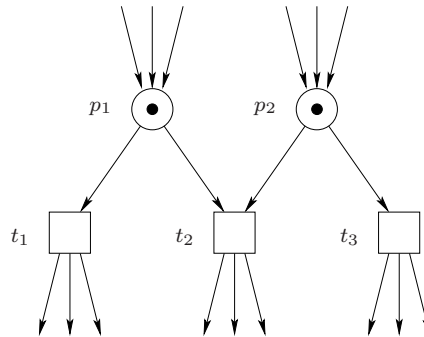


Abbildung 3.10.: Beispiel für einen Konflikt, der die Ausführung von WS-BPEL-Aktivitäten verhindern kann.

Zu beachten ist, dass die an dieser Stelle betrachteten Transitionen t_1, t_2, \dots, t_n neben dem Platz p_1 weitere Plätze im Vorbereich besitzen können. Dieser Fall war bei der getrennten Betrachtung von bedingtem Verhalten in Abschnitt 3.2.3 von uns ausgeschlossen worden. Es können Aktivitäten im WS-BPEL-Prozess dadurch nicht ausgeführt werden, dass sich im Vorbereich einer Transition mehrere Plätze befinden, die ihrerseits mehrere Transitionen in ihren Nachbereichen besitzen. Das Beispiel in Abb. 3.10 zeigt die Plätze p_1 und p_2 , die sich im Vorbereich der Transitionen t_1, t_2 und t_3 befinden. Nehmen wir an, auf den beiden Plätzen wird jeweils nur die dargestellte Marke produziert. Wenn eine der Transitionen t_1 oder t_3 schaltet, kann anschließend nur noch die andere (t_3 beziehungsweise t_1) einmal schalten, nicht jedoch t_2 . Im oWFN schalten die Transitionen nicht-

deterministisch. Im WS-BPEL-Prozess jedoch, findet eine deterministische Entscheidung durch die den Plätzen zugewiesenen `if`-Aktivitäten statt. In diesem Beispiel könnte sich die bedingte Auswahl, die wir mit Platz p_1 verbinden, für die anschließende Ausführung der Aktivitäten, die wir mit Transition t_1 verbinden, entscheiden. Entscheidet sich nun die mit p_2 verbundene bedingte Auswahl unabhängig davon für die anschließende Ausführung der mit t_2 verbundenen Aktivitäten, kommt es zu keiner Ausführung der Aktivitäten, da deren Aktivierungsbedingungen nicht erfüllt sind. Weil es nicht möglich ist, dieses Problem durch die Anordnung der WS-BPEL-Aktivitäten zu umgehen, muss der Nutzer auf die Angabe geeigneter Bedingungen für die `if`-Aktivität achten. Wir fügen dazu an betroffenen Stellen einen warnenden Kommentar in den WS-BPEL-Code ein.

Eine Alternative zur Verwendung von `if`-Aktivitäten wäre die Angabe von `<transition-Condition>`-Elementen innerhalb der `<source>`-Elemente. Diese könnten durch Auswertung einer Booleschen Bedingung gezielt ausgewählte ausgehende Links auf den Wert `false` setzen. Wir haben uns für die Verwendung der `if`-Aktivität entschieden, weil wir so eine einheitliche Behandlung von bedingtem Verhalten sicherstellen. Auch können diese `if`-Aktivitäten, wie die in Abschnitt 3.2.3 erzeugten, durch Korrektur 1 durch `pick`-Aktivitäten ersetzt werden.

In einem weiteren Schritt ersetzen wir alle Kanten des Teilnetzes durch Links. Dabei nehmen wir als Quelle der Links die letzte Aktivität in der Annotation des Knotens an dem die Kante beginnt und als Ziel die erste Aktivität in der Annotation des Knotens an der die Kante endet. Dabei geben wir im WS-BPEL-Attribut `<joinCondition>` für das `<targets>`-Element bei Transitionen, die Ziel von mehreren Links sind, eine Boolesche UND-Verknüpfung aller Links an und bei Plätzen, die Ziel von mehreren Links sind, eine Boolesche ODER-Verknüpfung aller Links. Befinden sich im Vorbereitungsbereich einer Transition des Teilnetzes mehrere Plätze, dann kann die Transition nur schalten, wenn auf allen Plätzen eine Marke produziert wurde. Dies entspricht einer Ausführung der letzten Aktivitäten in den Annotationen der Plätze. Da die gerade erzeugten Links diese Aktivitäten als Quelle benutzen, wird die erste Aktivität einer Transition durch die UND-Verknüpfung in der Aktivierungsbedingung nur dann ausgeführt, wenn alle diese Aktivitäten ebenfalls ausgeführt wurden. Da wir ein sicheres und zyklensicheres Teilnetz umwandeln, kann auf jedem Platz nur einmal eine Marke produziert werden, selbst wenn Plätze des Teilnetzes mehrere Transitionen im Vorbereitungsbereich besitzen. Daher verwenden wir für die Aktivierungsbedingung für die ersten Aktivitäten an diesen Plätzen eine ODER-Verknüpfung aller eingehenden Links.

Im letzten Schritt bei der Erzeugung der neuen Annotation, die das Teilnetz ersetzen wird, erzeugen wir eine `flow`-Aktivität und legen für jeden Knoten des Teilnetzes einen Zweig in dieser an. In diesen Zweig wird die Annotation des Knotens übernommen und der Knoten wird aus dem Teilnetz entfernt. Die Reihenfolge in der dies geschieht ist dabei nicht wichtig. Nach diesem Schritt bleibt nur die Annotation der `flow`-Aktivitäten zurück, die wir nun in das ursprüngliche oWFN einfügen können.

Abhängig davon, ob es sich bei den Knoten x und y um Plätze oder Transitionen handelt, entscheiden wir, welche neuen Knoten wir für die erzeugte `flow` Annotation in das oWFN einfügen müssen. Abbildung 3.11 zeigt in Beispielen die im Folgenden aufgeführten sie-

ben verschiedenen Möglichkeiten für die Einbindung der transformierten Teilnetze. In der Abbildung repräsentieren die Wolken die Teilnetze A und B . Lediglich zur Vereinfachung der Darstellung wurde das Teilnetz A an einigen Stellen in zwei Wolken aufgeteilt. Die gebogenen Kanten sind Beispiele für das mögliche Vorhandensein von Kanten. Die Unterscheidung dieser sieben Möglichkeiten ist notwendig. Im Anhang A.3 geben wir zu jeder Möglichkeit ein Beispielpartikelnetz an.

Falls $x \in P$ fügen wir eine neue Transition t_0 in das oWFN ein, die mit der **flow**-Aktivität annotiert wird und es gilt:

- $\bullet t_0 = \{x\}$ und
- $x^\bullet = (x^\bullet \cap A) \cup \{t_0\}$.

Falls es ein y gibt und $y \in P$ gilt zudem:

- $t_0^\bullet = \{y\}$ und
- $\bullet y = (\bullet y \cap A) \cup \{t_0\}$.

Wenn $x \in B^\bullet$ dann gilt stattdessen:

- $t_0^\bullet = \{x\}$ und
- $\bullet x = (\bullet x \cap A) \cup \{t_0\}$.

Falls jedoch $y \in T$ fügen wir einen weiteren Platz p_0 in das oWFN ein und es gilt:

- $\bullet p_0 = \{t_0\}$,
- $p_0^\bullet = \{y\}$ und
- $\bullet y = (\bullet y \cap A) \cup \{p_0\}$.

Falls $x \in T$ fügen wir einen neuen Platz p_0 in das oWFN ein, der mit der **flow**-Aktivität annotiert wird und es gilt:

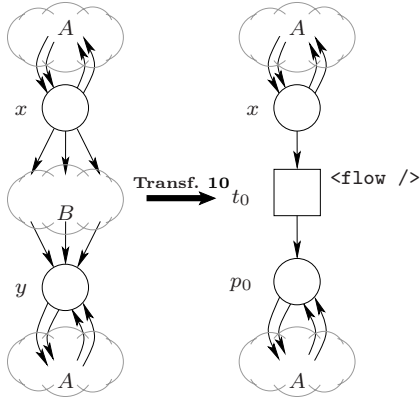
- $\bullet p_0 = \{x\}$ und
- $x^\bullet = (x^\bullet \cap A) \cup \{p_0\}$.

Falls es ein y gibt und $y \in T$ gilt zudem:

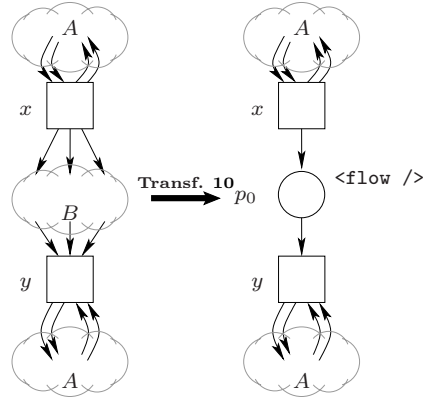
- $p_0^\bullet = \{y\}$ und
- $\bullet y = (\bullet y \cap A) \cup \{p_0\}$.

Falls jedoch $y \in P$ fügen wir eine weitere Transition t_0 in das oWFN ein, die wir mit einer **empty**-Aktivität annotieren und es gilt:

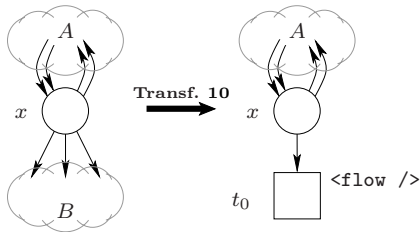
- $\bullet t_0 = \{p_0\}$,
- $t_0^\bullet = \{y\}$ und
- $\bullet y = (\bullet y \cap A) \cup \{t_0\}$.



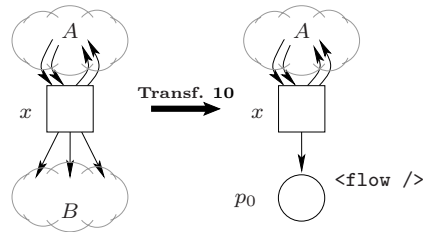
(a) Transformation mit $x, y \in P$.



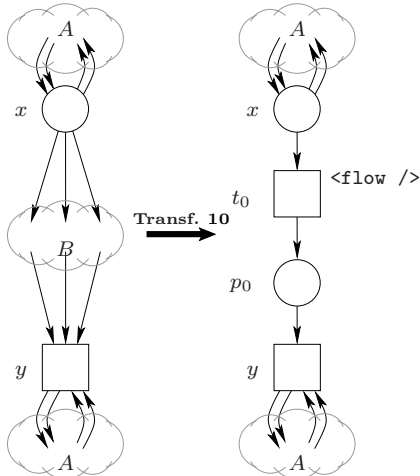
(b) Transformation mit $x, y \in T$.



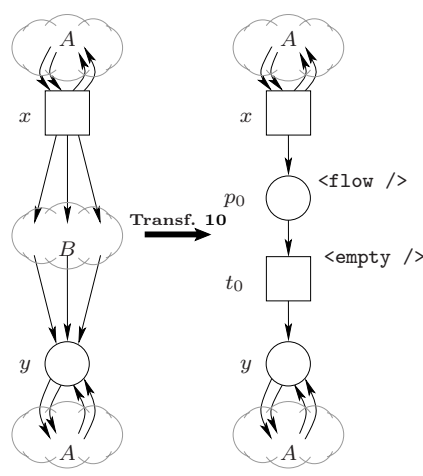
(c) Transformation mit $x \in P$.



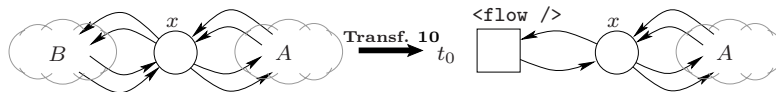
(d) Transformation mit $x \in T$.



(e) Transformation mit $x \in P$ und $y \in T$.



(f) Transformation mit $x \in T$ und $y \in P$.



(g) Transformation mit $x \in B^*$.

Abbildung 3.11.: Beispiele für die Transformation von Teilnetzen durch Transformation 10.

3.2.5. Zwischenfazit

Mit den Transformationen, die wir bisher in den Abschnitten 3.2.1–3.2.4 beschrieben haben, können wir bereits alle zyklensicheren und sicheren oWFN, die den weiteren Einschränkungen aus Abschnitt 2.1.3 genügen, in abstrakte WS-BPEL-Prozesse umwandeln. Durch die Transformationen in den folgenden beiden Abschnitten werden wir diese Lücke schließen können. Zunächst betrachten wir in Abschnitt 3.2.6 die Transformation von Zyklen, bevor wir uns in Abschnitt 3.2.7 der mehrfachen Ausführung von Aktivitäten widmen, die durch nicht sichere Plätze im oWFN entsteht. Die letzten beiden Transformationen stellen wir anschließend in Abschnitt 3.2.8 vor. Sie dienen der strukturellen Umformung des oWFN und gestatten eine gezieltere Anwendung der anderen Transformationen.

3.2.6. Zyklen

Zyklen stellen bei der Umwandlung der oWFN eine besondere Schwierigkeit dar. In WS-BPEL gelten Einschränkungen bezüglich der mehrfachen Ausführung von Aktivitäten. So kann eine Aktivität nur unter zwei Bedingungen ein weiteres Mal ausgeführt werden. Entweder geschieht dies, indem sie in eine der drei strukturierten Aktivitäten `while`, `repeatUntil` und `forEach` eingebettet wird oder indem sie innerhalb des mit der strukturierten Aktivität `scope` verbundenen Elements `<eventHandlers>` definiert wird. Wie wir bereits in Abschnitt 3.2.4 gesehen haben, ist es nicht möglich, Zyklen unter Verwendung von Links innerhalb einer `flow`-Aktivität nachzubilden. Zudem haben wir in Abschnitt 2.2.5 angesprochen, dass keine Links in eine dieser Aktivitäten hinein oder hinaus führen dürfen. Eine Möglichkeit, alle diese Einschränkungen zu umgehen, ist es, die Aktivitäten, die innerhalb eines WS-BPEL-Prozesses mehrfach ausgeführt werden sollen, auch mehrfach in den WS-BPEL-Code mit aufzunehmen. Von dieser Möglichkeit können wir jedoch nur Gebrauch machen, wenn wir die maximale Anzahl der Ausführungen kennen, denn so oft muss auch die Aktivität mit aufgenommen werden. Wir werden in Abschnitt 3.2.7 Transformationen betrachten, die so vorgehen. In diesem Abschnitt betrachten wir zunächst Zyklen, über deren Ausführungshäufigkeit keine Aussage getroffen werden kann.

Von den Aktivitäten, die wir in Abschnitt 2.2.6 vorgestellt haben, werden wir für die Transformation von Zyklen nur die Aktivität `while` verwenden. Die Aktivität `repeatUntil` ist ungeeignet, weil sie nicht die Möglichkeit bietet, dass die eingebetteten Aktivitäten gar nicht ausgeführt werden. Da wir anhand des oWFN keine Rückschlüsse auf die Notwendigkeit einer Zählervariablen ziehen können und die nebenläufige Ausführung von Zyklen an dieser Stelle nicht benötigen werden, ist die Verwendung der `forEach`-Aktivität nicht notwendig. Wir verwenden ebenfalls nicht das mit der `scope`-Aktivität verbundene Element `<eventHandlers>`, das ebenfalls eingebettete Aktivitäten mehrfach ausführen kann. Die Ausführungen der mit Ereignissen verbundenen Zweige des Elements können nur durch externe Nachrichten oder zeitliche Ereignisse gestartet werden, so dass keine Kontrolle über die Ausführungen der Aktivitäten von innerhalb desselben WS-BPEL-Prozesses möglich ist. Beim mehrfachen Eintreten der Ereignisse werden auch die Zweige mehrfach ausgeführt. Wie wir in Abschnitt 2.2.3 bereits angesprochen haben,

wird das Element `<eventHandlers>` zudem nebenläufig zu den in die `scope`-Aktivität eingebetteten Aktivitäten ausgeführt.

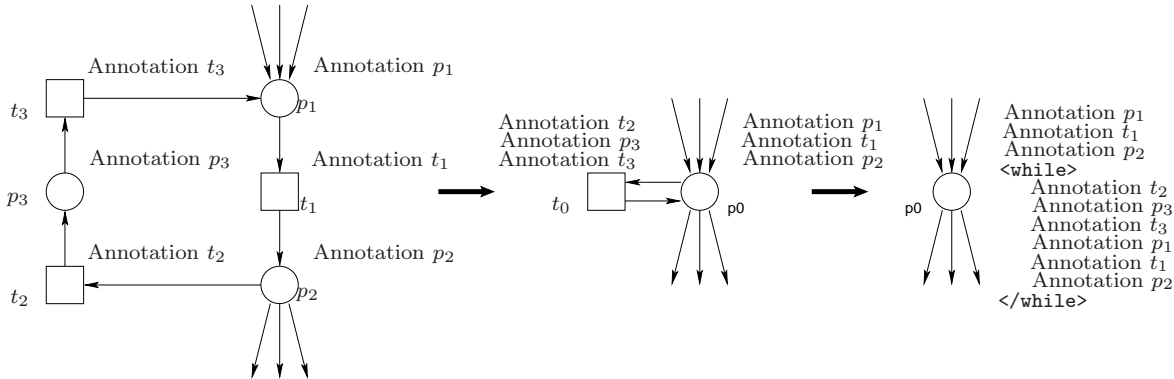


Abbildung 3.12.: Beispiel für einen einfachen Zyklus und dessen Transformation.

Bevor wir die Transformation von Zyklen vorstellen, betrachten wir zunächst das in Abb. 3.12 dargestellte Beispiel für einen Zyklus und die von uns bei dessen Transformation beabsichtigte entstehende Annotation. In der Mitte der Abbildung ist der aus dem Platz p_1 und der Transition t_1 bestehende Zyklus zu sehen. Die nicht abgebildeten Transitionen im Vor- und Nachbereich von p_1 sind nicht Teil des Zyklus. Nehmen wir an, im oWFN wird auf Platz p_1 eine Marke produziert. Zunächst müssen wir die Annotation von p_1 betrachten. Die Aktivitäten, die durch die Annotation des Platzes repräsentiert werden, befinden sich ebenfalls auf dem Zyklus. Keine unserer Transformationen, ermöglicht es, die Annotation von p_1 um Aktivitäten zu ergänzen, die sich nicht auf dem Zyklus befinden. Die einzige Ausnahme bilden die beiden in diesem Abschnitt vorgestellten Transformationen. Aus diesem Grund werden wir anschließend noch Korrektur 3 vorstellen. Auf der linken Seite der Abbildung haben wir zur Verdeutlichung einen möglichen Ursprung für den Zyklus dargestellt. Wie hier zu sehen ist, bestand der Zyklus zuvor aus jeweils drei Plätzen und Transitionen, die mit Hilfe der Transitionen aus Abschnitt 3.2.2 reduziert wurden. Nehmen wir an, aus p_1, t_1 und p_2 wurde der Platz p_0 und aus t_2, p_3 und t_3 die Transition t_0 . Eine Marke auf dem Platz p_0 des zum Teil transformierten oWFN ist somit also gleichbedeutend mit der sequentiellen Aktivierung der Aktivitäten, die durch die Annotation des Platzes repräsentiert werden. Daher finden wir auf der rechten Seite der Abbildung, die das Ziel unserer Transformation in diesem Beispiel darstellt, zuerst eine Kopie der Annotation des Platzes p_0 . Nun gibt es zwei für unsere Betrachtung interessante Fälle. Entweder Transition t_0 schaltet oder eine der anderen Transitionen im Nachbereich von p_0 . Im ersten Fall müssen alle Aktivitäten, die durch die Annotation von t_0 repräsentiert werden, ausgeführt werden. Durch das Schalten von t_0 wird eine neue Marke auf p_0 produziert, wodurch wir an den Beginn unserer Betrachtung zurückkehren und somit den Zyklus einmal durchlaufen haben. Alternativ dazu wird im zweiten Fall der Zyklus verlassen. Wir bilden dieses Verhalten im oWFN nach, indem wir eine `while`-Aktivität einfügen, in der zuerst die Annotation von t_0 und dann die Annotation von p_0 eingefügt wird. Durch spätere Angabe einer Bedingung für diese Aktivität durch den Nutzer wird entschieden, wie viele Wiederholungen der einge-

betteten Aktivitäten stattfinden werden oder ob die eingebetteten Aktivitäten gar nicht ausgeführt werden.

Das Beispiel zeigt die folgende Transformation in ihrem einfachsten Anwendungsfall. Um die vollständige Transformation einfacher beschreiben zu können, führen wir an dieser Stelle die beiden Begriffe *Eingang* und *Ausgang* eines Zyklus ein. Als Eingang wollen wir Plätze auf einem Zyklus bezeichnen, in deren Vorbereich sich Transitionen befinden, die nicht Teil des Zyklus sind. Entsprechend bezeichnen wir mit Ausgang Plätze in deren Nachbereich sich Transitionen befinden, die nicht Teil des Zyklus sind. Im vorangegangenen Beispiel war der Platz p_1 Eingang und Platz p_2 Ausgang des Zyklus. Ein Platz kann gleichzeitig Eingang und Ausgang eines oder sogar mehrerer Zyklen sein. Der Zyklus im Beispiel besitzt nur jeweils den Eingang p_1 und den Ausgang p_2 . Die erste Transformation, die wir in diesem Abschnitt betrachten, reduziert Zyklen beliebiger Größe mit einer beliebigen Anzahl von Ausgängen und genau einem Eingang. Die zweite Transformation transformiert einen Zyklus mit mehreren Eingängen indem der Zyklus abgewickelt wird.

Transformation 11

Bei unserer Transformation nehmen wir die Annotationen aller Knoten des Zyklus mit in eine neue **while** Annotation auf. Wir beginnen dabei beim ersten Knoten des Zyklus, der sich im Nachbereich des Eingangsplatzes befindet und enden mit dem Eingangsplatz selbst. Die ursprüngliche Annotation des Eingangsplatzes bleibt vor der neuen **while** Annotation vorhanden. Die Entscheidung, mit Hilfe welches Ausgangs der Zyklus verlassen wird, verschieben wir hinter den Zyklus. Wir können die Entscheidung nicht im Zyklus belassen, ohne selber eine Abbruchbedingung für die entstehende **while**-Aktivität mit anzugeben. Anderenfalls könnte die Aktivität noch weitere Wiederholungen ausführen, obwohl die Entscheidung für einen Ausgang bereits getroffen wurde. Wir entfernen das Innere des Zyklus aus dem oWFN und fügen stattdessen einen neuen Platz p_0 ein. In dessen Nachbereich ordnen wir die Transitionen aus den Nachbereichen der Ausgänge des Zyklus an. Die Annotationen dieser ergänzen wir um die Annotationen der Knoten, die sich im Inneren des Zyklus befunden haben. Der Nutzer muss darauf achten, dass die Abbruchbedingung der **while**-Aktivität berücksichtigt, dass die Aktivitäten der Knoten des Zyklus vom Eingangsplatz bis zum später gewählten Ausgangsplatz nach der Beendigung der Aktivität noch einmal ausgeführt werden. Im Folgenden sind p_1 der Eingang des Zyklus und e_1, e_2, \dots, e_m die Transitionen in den Nachbereichen der Ausgänge. Seien $p_1, p_2, \dots, p_n \in P$, $t_1, t_2, \dots, t_n, e_1, e_2, \dots, e_m \in T$ und $n, m \in \mathbb{N}$. Wenn gilt, dass

- $t_1 \in p_1^\bullet, t_2 \in p_2^\bullet, \dots, t_n \in p_n^\bullet$,
- $t_n \in {}^\bullet p_1$,
- ${}^\bullet p_2 = \{t_1\}, \dots, {}^\bullet p_n = \{t_{n-1}\}$,
- ${}^\bullet t_1 = \{p_1\}, {}^\bullet t_2 = \{p_2\}, \dots, {}^\bullet t_n = \{p_n\}$,
- $t_1^\bullet = \{p_2\}, t_2^\bullet = \{p_3\}, \dots, t_{n-1}^\bullet = \{p_n\}, t_n^\bullet = \{p_1\}$,
- $\{p_1, p_2, \dots, p_n\}^\bullet = \{e_1, e_2, \dots, e_m\} \cup \{t_1, t_2, \dots, t_n\}$ und

$$- \{e_1, e_2, \dots, e_m\} \cap \{t_1, t_2, \dots, t_n\} = \emptyset$$

dann können wir die Plätze p_1, p_2, \dots, p_n und die Transitionen t_1, t_2, \dots, t_n ersetzen durch einen neuen annotierten Platz p_0 . Für diesen gilt:

- $\bullet p_0 = \bullet p_1 \setminus \{t_n\}$,
- $p_0 \bullet = \{e_1, e_2, \dots, e_m\}$ und
- $\bullet e_1 = (\bullet e_1 \cup p_0) \setminus \{p_1, p_2, \dots, p_n\}$, $\bullet e_2 = (\bullet e_2 \cup p_0) \setminus \{p_1, p_2, \dots, p_n\}, \dots, \bullet e_m = (\bullet e_m \cup p_0) \setminus \{p_1, p_2, \dots, p_n\}$.

Die Annotation in Codebeispiel 3.22 entspricht der neuen Annotation am Platz p_0 . Die Annotationen eines $e_i \in p_j \bullet$ ist in Codebeispiel 3.23 aufgeführt. Befindet sich innerhalb einer der Transitionen eine `flow`-Aktivität mit Links, dann müssen die Bezeichner der Links innerhalb der kopierten Annotationen durch noch nicht verwendete Namen ersetzt werden. Falls die bisherige Annotation von p_1 jedoch mit einer oder mehreren `while` Annotationen endet, verwenden wir als neue Annotation für p_0 diejenige in Codebeispiel 3.24. Hierbei wird die neue `while` Annotation an die Liste der bereits vorhandenen angehängt. Codebeispiel 3.25 zeigt die Annotation eines Platzes p_1 , die bereits auf `i while` Annotationen endet, vor der Transformation. Diese Fallunterscheidung ist notwendig, um nach Abschluss der Transformationen Korrektur 3 durchführen zu können. Wie bei den bisherigen Transformationen fügen wir `opaqueActivity` Annotationen für jede bisher nicht annotierte Transition mit in die `while` Annotation ein. Abbildung 3.13 zeigt ein Beispiel mit abgekürzter Annotation für diese Transformation mit drei Plätzen und drei Transitionen auf dem Zyklus. Die Transitionen e_1, e_2, e_3 und e_4 sind Beispiele für die Nachbereiche der Ausgänge des Zyklus. Es könnten Weitere vorhanden sein und sie könnten noch weitere Plätze in ihren Vor- und Nachbereichen besitzen.

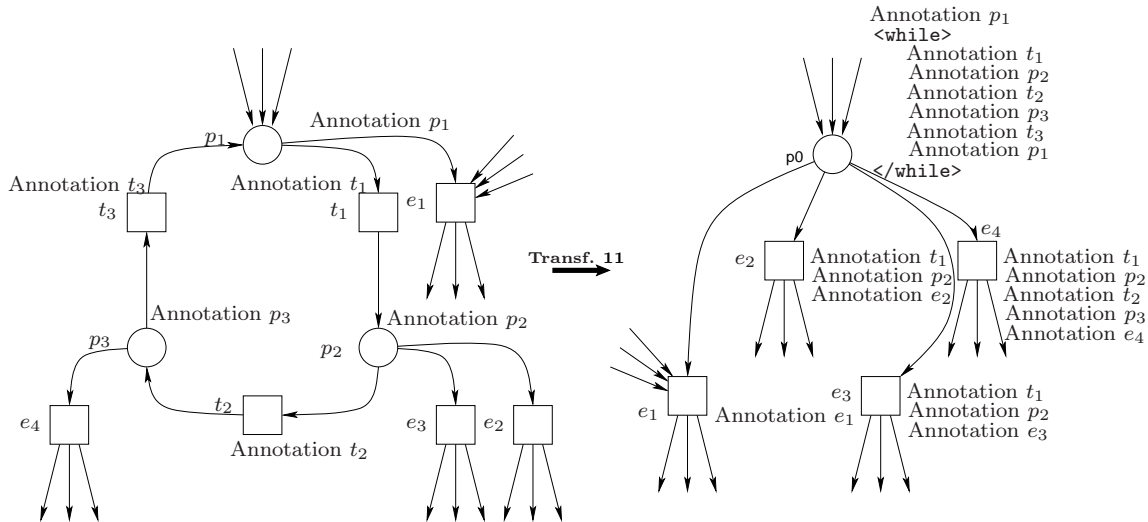


Abbildung 3.13.: Beispiel für die Transformation von Zyklen durch Transformation 11.

Annotation von p_1

```
<while>
  Annotation von  $t_1$ 
  Annotation von  $p_2$ 
  Annotation von  $t_2$ 
  Annotation von  $p_3$ 
  ...
  Annotation von  $t_{n-1}$ 
  Annotation von  $p_n$ 
  Annotation von  $t_n$ 
  Annotation von  $p_1$ 
</while>
```

Codebeispiel 3.22: Annotation von p_0 nach Transformation 11.

beliebige Annotationen von p_1

```
<while>
  erste while Aktivität von  $p_1$ 
</while>
<while>
  zweite while Aktivität von  $p_1$ 
</while>
...
<while>
   $i$ -te while Aktivität von  $p_1$ 
</while>
<while>
  Annotation von  $t_1$ 
  Annotation von  $p_2$ 
  Annotation von  $t_2$ 
  Annotation von  $p_3$ 
  ...
  Annotation von  $t_{n-1}$ 
  Annotation von  $p_n$ 
  Annotation von  $t_n$ 
  beliebige Annotationen von  $p_1$ 
</while>
```

Codebeispiel 3.24: Alternative Annotation von p_0 , wenn bereits `while` Annotationen vorhanden sind.

```
Annotation von  $t_1$ 
Annotation von  $p_2$ 
Annotation von  $t_2$ 
Annotation von  $p_3$ 
...
Annotation von  $t_{j-1}$ 
Annotation von  $p_j$ 
Annotation von  $e_i$ 
```

Codebeispiel 3.23: Annotation von $e_i \in p_j^\bullet$ nach Transformation 11.

beliebige Annotationen von p_1

```
<while>
  erste while Aktivität von  $p_1$ 
</while>
<while>
  zweite while Aktivität von  $p_1$ 
</while>
...
<while>
   $i$ -te while Aktivität von  $p_1$ 
</while>
```

Codebeispiel 3.25: Annotation von p_1 mit vorhandenen `while` Annotationen vor Transformation 11.

Wie bei den bisherigen Transformationen handelt es sich bei unseren Annotationen lediglich um einige Schlüsselworte, die wir später noch verändern können und zum Abschluss der Transformation durch WS-BPEL-Code ersetzen. Codebeispiel 3.26 zeigt den WS-BPEL-Code, der aus der `while` Annotation in Codebeispiel 3.22 entstanden ist. Der Nutzer muss später die Bedingungen für die Anzahl der Ausführungen festlegen. Die noch einzutragende Bedingung ist erneut durch das für abstrakte WS-BPEL-Prozesse definierte Element `<condition opaque="yes"/>` gekennzeichnet. Wenn diese Bedingung vom Empfang einer Nachricht abhängen soll, wie es sich in einem oWFN sehr einfach realisieren lässt, muss der Nutzer den erzeugten WS-BPEL-Code umgestalten und unter Umständen auf die `scope`-Aktivität mit ihrem `<eventHandlers>`-Element zurückgreifen.

Denn wenn eine der anderen empfangenden Aktivitäten einmal ausgeführt wird, kann sie nur durch den Empfang einer Nachricht oder das Auftreten eines Fehlers wieder beendet werden. Ein Zyklus mit einer empfangenden Aktivität kann somit dadurch verklemmen, dass er häufiger ausgeführt wird, als benötigte Nachrichten eintreffen.

```
Aktivitäten von  $p_1$ 
<while name="Act_while_ $p_1$ ">
  <condition opaque="yes" />
  eingebettete Aktivitäten
</while>
```

Codebeispiel 3.26: Der spätere WS-BPEL-Code einer `while` Annotation.

Die folgende Transformation reduziert entgegen unserem allgemeinen Vorgehen nicht die Anzahl der Knoten im oWFN, sondern vergrößert sie sogar. Dies ist notwendig, um Zyklen mit mehr als einem Eingang transformieren zu können. Dazu wickeln wir den Zyklus ab. Wir greifen uns einen der Eingangsplätze (p_1) aus dem Zyklus heraus mit dem der abgewickelte Zyklus beginnt. Diese Auswahl ist nicht deterministisch. Es gibt keine Kriterien, die einen der Eingänge eines beliebigen Zyklus gegenüber den anderen auszeichnen. Auch für die Transitionen (e_1, e_2, \dots, e_m) im Nachbereich der Ausgänge des Zyklus fügen wir Kopien (f_1, f_2, \dots, f_m) ein. Damit es möglich ist, bereits bei der ersten Ausführung der Aktivitäten den Zyklus zu beenden, schaffen wir mit Hilfe dieser neuen Transitionen die Möglichkeit auch vom abgewickelten Zyklus direkt die Nachbereiche der Ausgänge zu erreichen.

Transformation 12

Für die Beschreibung dieser Transformation benennen wir die Eingänge und Ausgänge des Zyklus als r_1, r_2, \dots, r_l und q_1, q_2, \dots, q_k . Wir werden zudem voraussetzen, dass der Zyklus mindestens zwei Eingänge besitzt. Die Anzahl der Ausgänge ist beliebig. Seien $p_1, p_2, \dots, p_n \in P$, $t_1, t_2, \dots, t_n \in T$ und $i, j, k, l, n, m \in \mathbb{N}$. Wenn gilt, dass

- $n > 1$,
- $t_1 \in p_1^\bullet, t_2 \in p_2^\bullet, \dots, t_n \in p_n^\bullet$,
- $t_n \in \bullet p_1, t_1 \in \bullet p_2, \dots, t_{n-1} \in \bullet p_n$,
- $\bullet t_1 = \{p_1\}, \bullet t_2 = \{p_2\}, \dots, \bullet t_n = \{p_n\}$,
- $t_1^\bullet = \{p_2\}, t_2^\bullet = \{p_3\}, \dots, t_{n-1}^\bullet = \{p_n\}, t_n^\bullet = \{p_1\}$,
- $\{q_1, q_2, \dots, q_k\} \subseteq \{p_1, p_2, \dots, p_n\}$,
- $\{r_1, r_2, \dots, r_l\} \subseteq \{p_1, p_2, \dots, p_n\}$,
- $\forall r \in \{r_1, r_2, \dots, r_l\} : \bullet r \setminus \{t_1, t_2, \dots, t_n\} \neq \emptyset$,
- $\forall q \in \{q_1, q_2, \dots, q_k\} : q^\bullet \setminus \{t_1, t_2, \dots, t_n\} \neq \emptyset$,
- $l > 1$,

- $\{q_1, q_2, \dots, q_k\}^\bullet = \{e_1, e_2, \dots, e_m\} \cup \{t_1, t_2, \dots, t_n\}$ und
- $\{e_1, e_2, \dots, e_m\} \cap \{t_1, t_2, \dots, t_n\} = \emptyset$

dann können wir für jedes $e_i \in \{e_1, e_2, \dots, e_m\}$ einen neue Transition f_i in das oWFN einfügen. Die Transition f_i erhält als Annotation eine Kopie der Annotation von e_i . Die Annotationen der e_i werden wie bei der vorherigen Transformation 11 abgeändert. Zusätzlich fügen einen neuen Platz p_0 ein, dessen Annotation ebenfalls der in Transformation 11 angegebenen entspricht. Es gilt:

- $t_n^\bullet = \{p_0\}$,
- ${}^\bullet p_0 = \{t_n\}$,
- $p_0^\bullet = \{e_1, e_2, \dots, e_m\}$,
- ${}^\bullet f_1 = {}^\bullet e_1, {}^\bullet f_2 = {}^\bullet e_2, \dots, {}^\bullet f_m = {}^\bullet e_m$,
- $f_1^\bullet = e_1^\bullet, f_2^\bullet = e_2^\bullet, \dots, f_m^\bullet = e_m^\bullet$,
- ${}^\bullet e_1 = ({}^\bullet e_1 \cup p_0) \setminus \{q_1, q_2, \dots, q_k\}, {}^\bullet e_2 = ({}^\bullet e_2 \cup p_0) \setminus \{q_1, q_2, \dots, q_k\}, \dots, {}^\bullet e_m = ({}^\bullet e_m \cup p_0) \setminus \{q_1, q_2, \dots, q_k\}$.

Abbildung 3.14 zeigt ein Beispiel mit abgekürzter Annotation für die Transformation. In der Abbildung ist auf der linken Seite eine Abwandlung des aus Abb. 3.13 bekannten Zyklus zu sehen. Zusätzlich zu den vier Transitionen e_1, e_2, e_3 und e_4 , die Beispiele für die Nachbereiche der Ausgänge sind, haben wir die Transitionen i_1 und i_2 als Beispiele für die Vorbereiche von Eingängen eingefügt. Die beiden Transitionen könnten selbstverständlich noch weitere Plätze im Nachbereich besitzen. Die Menge $\{q_1, q_2, \dots, q_k\}$ der Ausgänge des Zyklus besteht in diesem Beispiel aus den Elementen p_1, p_2 und p_3 und die Menge $\{r_1, r_2, \dots, r_l\}$ der Eingänge aus den Elementen p_1 und p_3 . Zur besseren Verdeutlichung haben wir die Plätze x_1, x_2, \dots, x_6 im Vor- und Nachbereich von Transition e_1 mit eingezeichnet. Alle Knoten auf der linken Seite der Abbildung besitzen eine ihrer Bezeichnung entsprechende Annotation. Auf der rechten Seite der Abbildung ist das Ergebnis der Transformation dargestellt. Neben den Kopien der Transitionen im Nachbereich der Ausgänge des Zyklus (f_1, f_2, f_3 und f_4) ist der mit der **while**-Aktivität annotierte Platz p_0 hinzugefügt worden. Bei den doppelt abgebildeten Plätzen x_1, x_2, \dots, x_6 in den Vor- und Nachbereichen von e_1 und f_1 handelt es sich um dieselben Plätze, die lediglich aus Platzgründen mehrfach dargestellt werden.

Korrektur 3

Zuletzt betrachten wir die anfangs angekündigte Korrektur. Wie bereits zuvor angesprochen, sind die Transformationen 11 und 12 die einzigen, die es ermöglichen, Eingänge von Zyklen mit Aktivitäten zu annotieren, die sich nicht selbst auf dem Zyklus befinden. Würden wir die Fallunterscheidung während der Transformationen nicht vornehmen, würde das bedeuten, dass wenn ein Platz Eingang von mehreren Zyklen ist und der erste Zyklus mit Hilfe von Transformation 11 oder 12 zu einer **while** Annotation reduziert wurde, diese Annotation in die Annotation mit eingefügt wird, die entsteht, wenn der zweite Zyklus transformiert wird. Abbildung 3.15 zeigt dies anhand eines Beispiels bei

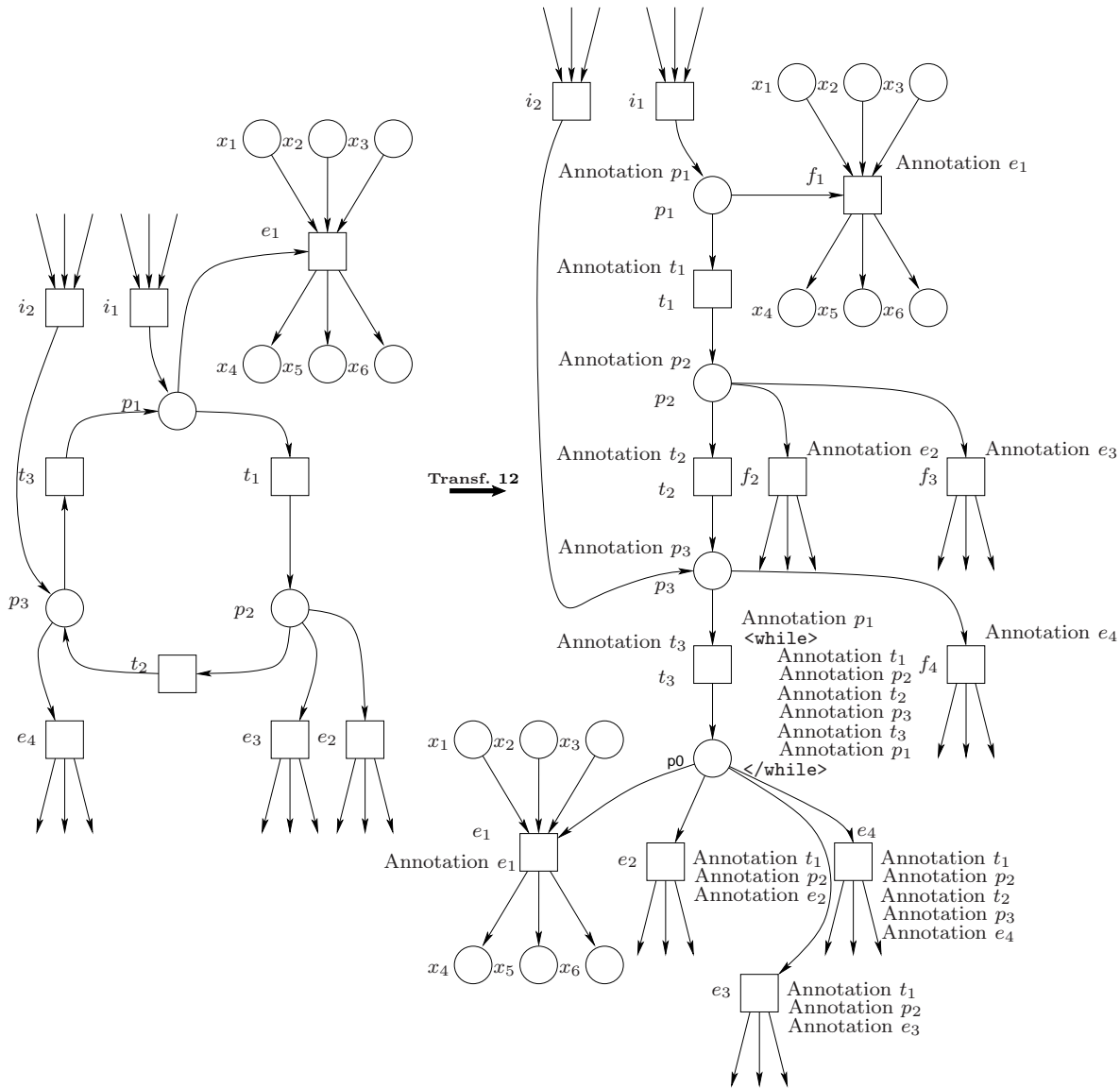


Abbildung 3.14.: Beispiel für Transformation 12 von Zyklen mit mehreren Eingängen.

dem der Platz p_1 Eingang von zwei Zyklen ist und diese in zwei Schritten reduziert werden. Unsere Fallunterscheidung erzeugt eine Aneinanderreihung von `while` Annotationen. Da sich im oWFN zwischen zwei Plätzen, die Eingänge von Zyklen sein können, immer eine Transition befinden muss und wir jede Transition zumindest mit der Aktivität `opaqueActivity` annotieren, können in den Annotationen ansonsten nie mehrere `while` Annotationen aufeinander folgen. Daher können wir alle Gruppen von aufeinander folgenden `while` Annotationen durch die Korrektur ersetzen lassen.

Betrachten wir zunächst die Möglichkeiten im oWFN. Ist ein Platz markiert, der Eingang von mehreren Zyklen ist, kann einer der Zyklen betreten werden. Wird der Zyklus nicht

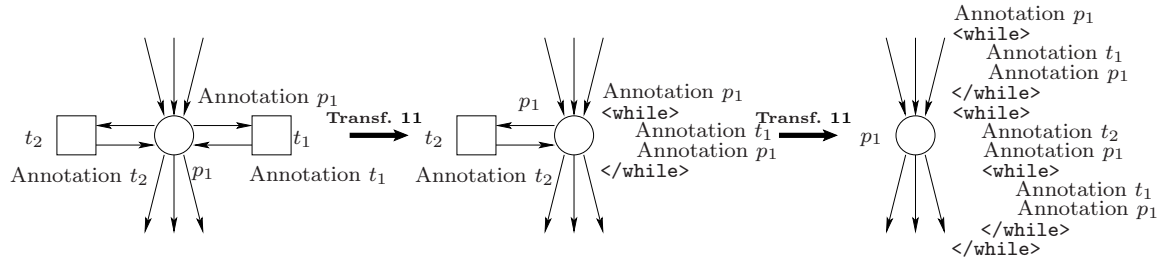


Abbildung 3.15.: Keine gültige Transformation für zwei Zyklen.

verlassen, wird erneut der Eingang markiert und es kann wiederum einer der Zyklen ausgeführt werden. Eine Aneinanderreihung der einzelnen Zyklen im WS-BPEL-Code würde also nicht dem Verhalten im oWFN entsprechen. Daher fügen wir eine neue `while` Annotation ein und in dieser eine `if` Annotation, die als Zweige die Annotationen erhält, die sich zuvor innerhalb der ursprünglichen `while` Annotationen befunden haben. Codebeispiel 3.27 zeigt einen Ausschnitt mit einer Aneinanderreihung von `while` Annotationen, wie sie durch unsere Transformationen entstehen und Codebeispiel 3.28 zeigt den Ausschnitt nach unserer Korrektur.

Wichtig ist ein solches Konstrukt zum Beispiel, um wechselseitigen Ausschluss zu realisieren, bei dem auf eine Ressource nur nacheinander und nie gleichzeitig zugegriffen werden kann. Durch unsere Korrektur werden die Zyklen im erzeugten WS-BPEL-Code durch die `while`-Aktivität mehrfach ausgeführt und nach jeder Ausführung wird mit Hilfe der `if`-Aktivität neu entschieden, welcher Abfolge von Aktivitäten ausgeführt wird. Die verschiedenen Zyklen finden somit weder gleichzeitig, noch sortiert nacheinander statt. Die Korrektur kann nicht angewendet werden, wenn eine der `while`-Aktivitäten Quelle oder Ziel von Links ist.

```

beliebige Annotationen
<while>
  Annotationen der ersten <while> Aktivität
</while>
<while>
  Annotationen der zweiten <while> Aktivität
</while>
...
<while>
  Annotationen der i-ten <while> Aktivität
</while>
beliebige Annotationen
    
```

Codebeispiel 3.27: Die Ausgangsannotation für Korrektur 3.

In Abschnitt 3.2.7 werden wir uns noch einmal mit den besonderen Einschränkungen von Zyklen beschäftigen und mit Transformation 15 Teilnetze innerhalb von Zyklen so reduzieren, dass die beiden in diesem Abschnitt vorgestellten Transformationen auf sie anwendbar werden.

```

beliebige Annotation
<while>
  <if>
    Zweig mit Annotationen der ersten <while> Aktivität
    Zweig mit Annotationen der zweiten <while> Aktivität
    ...
    Zweig mit Annotationen der  $i$ -ten <while> Aktivität
  </if>
</while>
beliebige Annotationen

```

Codebeispiel 3.28: Die Annotation nach Korrektur 3.

3.2.7. Mehrfache Ausführungen von Teilnetzen

Befinden sich mehrere Marken in einem oWFN kann es passieren, dass Transitionen mehrfach schalten können, ohne dass diese Teil eines Zyklus sind. Dies bedeutet, dass die mit diesen Transitionen verknüpften Aktivitäten im WS-BPEL-Prozess ebenfalls mehrfach ausgeführt werden müssen. Wie bereits in Abschnitt 3.2.6 angesprochen, gibt es strukturierte Aktivitäten, die die mehrfache Ausführung von in sie eingebettete Aktivitäten ermöglichen. Da es durch mehrere Marken im oWFN aber zur nebenläufigen Ausführung der Transitionen kommen kann, müssen wir auch die Aktivitäten nebenläufig zueinander ausführen. Daher werden wir in diesem Abschnitt Transformationen einführen, die dafür sorgen, dass Aktivitäten, die mehrfach ausgeführt werden können, auch mehrfach in den WS-BPEL-Code aufgenommen werden.

Bei den Transformationen in diesem Abschnitt suchen wir Plätze, die mehrere Transitionen im Vorbereich besitzen. Diese Plätze, die nach Ausführung der Transformationen aus den Abschnitten 3.2.3 und 3.2.4 noch im oWFN vorhanden sind, sind Kandidaten für den Beginn eines Teilnetzes, das mehrfach ausgeführt werden kann. Wenn es keine Kanten zwischen dem mehrfach ausgeführten Teilnetz und dem Rest des oWFN gibt, können wir das gesamte Teilnetz kopieren. Dies geschieht bei unserer ersten Transformation. Bei der zweiten Transformation können wir dann nur noch den direkten Nachbereich des Platzes kopieren. Die letzte Transformation in diesem Abschnitt wird sich der mehrfachen Ausführung innerhalb eines Zyklus widmen. Wie zuvor bei Transformation 10 sorgen wir durch die Positionierung dieser Transformationen und dadurch dass wir jeweils nur eine von ihnen einmal ausführen, bevor wir wieder die anderen Transformationen aufrufen, dafür, dass nicht unnötig Teile des oWFN kopieren werden.

Transformation 13

Für diese Transformation teilen wir das oWFN in zwei Teilnetze A und B und den Platz p auf. Im Vorbereich von Platz p befinden sich mehrere Transitionen und er ist die einzige Verbindung zwischen A und B . Nach der Aufteilung können wir p zusammen mit B für jede der Transitionen im Vorbereich von p einmal kopieren. Seien $p \in P$, $t_1, t_2, \dots, t_n, u_1, u_2, \dots, u_m \in T$, $A, B \subset P \cup T$ und $n, m \in \mathbb{N}$ mit

- $\bullet p = \{t_1, t_2, \dots, t_n\}$,
- $p^\bullet = \{u_1, u_2, \dots, u_m\}$,
- $A \cap B = \emptyset$,
- $p \notin A \cup B$ und
- $A \cup B \cup \{p\} = P \cup T$.

Wenn gilt, dass

- $n > 1$,
- $B \cap A^\bullet = \emptyset$,
- $B^\bullet \subset B$,
- $\bullet B \subset B \cup \{p\}$,
- $p^\bullet \subset B$
- $\forall b \in B: b \in \text{Pfad}(p)$

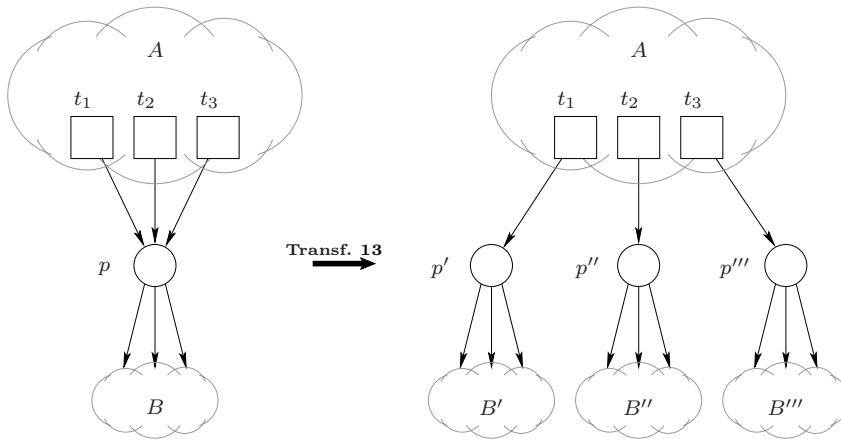
dann fügen wir n Kopien p', p'', \dots, p^n von p und n Kopien B', B'', \dots, B^n von B in das oWFN ein. Für diese gilt $\forall x \in \{1, 2, \dots, n\}$:

- $\bullet p^x = \{t_x\}$,
- $t_x^\bullet = (t_x^\bullet \cup \{p^x\}) \setminus \{p\}$,
- $p^{x^\bullet} = \{u_1^x, u_2^x, \dots, u_m^x\}$,
- $\forall b \in B^x : \forall c \in b^\bullet : b^\bullet = (b^\bullet \cup \{c^x\}) \setminus \{c\}$ und
- $\forall b \in B^x : \forall c \in \bullet b : \bullet b = (\bullet b \cup \{c^x\}) \setminus \{c\}$.

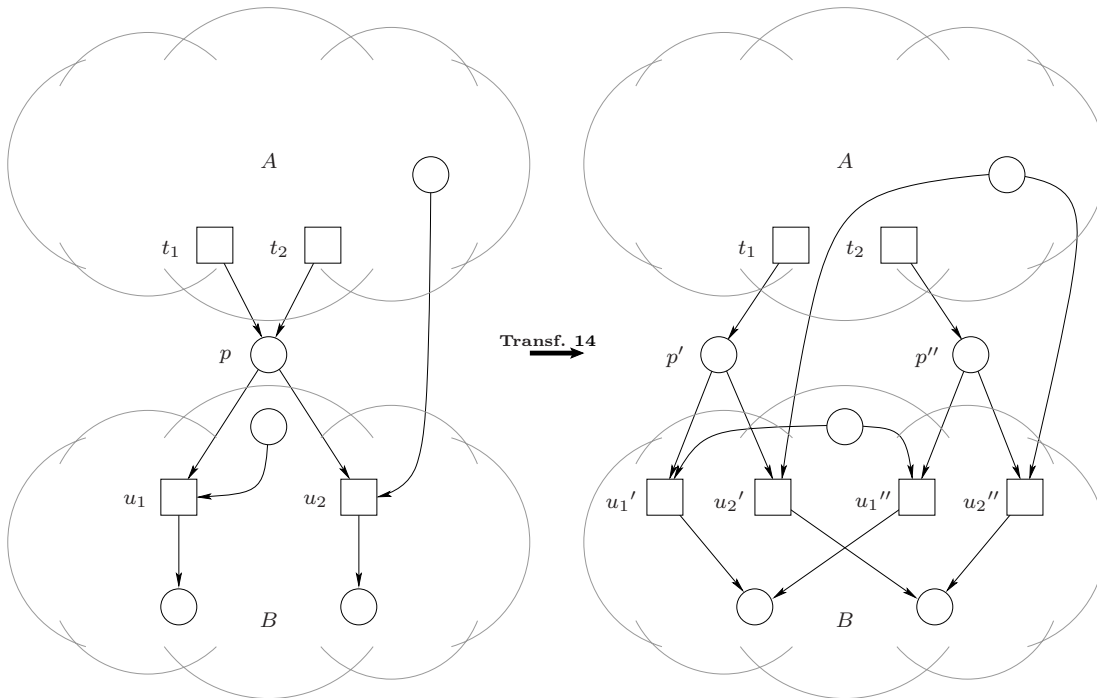
Zudem werden die Annotationen der kopierten Knoten mitkopiert. Kopierte Links innerhalb der Annotationen müssen, falls vorhanden, umbenannt werden, damit sie weiterhin eindeutige Bezeichner besitzen. Anschließend entfernen wir p und B aus dem oWFN. Abbildung 3.16(a) zeigt ein Beispiel für diese Transformation. Auf der linken Seite der Abbildung sind die Teilnetze A und B durch die Wolken dargestellt. Beispielhaft sind drei Transitionen t_1, t_2 und t_3 im Vorbereich des Platzes p zu sehen. Die Plätze im Nachbereich von p und weitere mögliche Kanten an t_1, t_2 und t_3 sind nicht abgebildet. Auf der rechten Seite sind die jeweils drei Kopien von p und B als p', p'', p''', B', B'' und B''' zu sehen.

Transformation 14

Für die zweite Transformation teilen wir erneut das oWFN in zwei Teilnetze A und B und den Platz p auf. Im Vorbereich von Platz p befinden sich mehrere Transitionen. Es sind zudem Kanten von Plätzen in A zu Transitionen in B erlaubt. Nach der Aufteilung können wir p zusammen mit den Transitionen im Nachbereich von p für jede der Transitionen im Vorbereich von p einmal kopieren. Seien $p \in P, t_1, t_2, \dots, t_n, u_1, u_2, \dots, u_m \in T, A, B \subset P \cup T$ und $n, m \in \mathbb{N}$ mit



(a) Beispiel für Transformation 13 ohne Kanten von A nach B .



(b) Beispiel für Transformation 14 mit einer Kante von A nach B .

Abbildung 3.16.: Beispiele für die Transformation bei mehrfachen Ausführungen.

- $\bullet p = \{t_1, t_2, \dots, t_n\}$,
- $p^\bullet = \{u_1, u_2, \dots, u_m\}$,
- $A \cap B = \emptyset$,
- $p \notin A \cup B$ und
- $A \cup B \cup \{p\} = P \cup T$.

Wenn gilt, dass

- $n > 1$,
- $\forall t \in A \cap T, \forall p \in B \cap P : p \notin t^\bullet$,
- $B^\bullet \subset B$,
- $\bullet B \subset B \cup \{p\} \cup (A \cap T)$,
- $p^\bullet \subset B$
- $\forall b \in B: b \in \text{Pfad}(p)$

dann fügen wir n Kopien p', p'', \dots, p^n von p und n Kopien $u_1', u_1'', \dots, u_1^n, u_2', u_2'', \dots, u_2^n, \dots, u_m', u_m'', \dots, u_m^n$ von u_1, u_2, \dots, u_m in das oWFN ein. Für diese gilt $\forall x \in \{1, 2, \dots, n\}$:

- $\bullet p^x = \{t_x\}$,
- $t_x^\bullet = (t_x \bullet \cup \{p^x\}) \setminus \{p\}$,
- $p^{x\bullet} = \{u_1^x, u_2^x, \dots, u_m^x\}$ und
- $\forall u \in \{u_1, u_2, \dots, u_m\}, \forall x \in \{1, 2, \dots, n\} : u^{x\bullet} = u^\bullet \wedge \bullet u^x = \bullet u \wedge \bullet(u^{x\bullet}) = (\bullet(u^\bullet) \cup \{u^x\}) \setminus \{u\} \wedge (\bullet u^x)^\bullet = (\bullet u)^\bullet \cup \{u^x\} \setminus \{u\}$.

Zudem werden erneut die Annotationen der kopierten Knoten mitkopiert. Anschließend entfernen wir p und u_1, u_2, \dots, u_m aus dem oWFN. Abbildung 3.16(b) zeigt ein Beispiel für diese Transformation. Auf der linken Seite der Abbildung sind die Teilnetze A und B durch die Wolken dargestellt. Beispielhaft sind zwei Transitionen t_1 und t_2 im Vorbereich des Platzes p und zwei Transitionen u_1 und u_2 in dessen Nachbereich zu sehen. Ein Platz in A und ein Platz in B aus den Vorbereichen von u_1 und u_2 wurden abgebildet, um als Beispiel für die bei der Transformation neu eingefügten Kanten zu dienen. Auf der rechten Seite sind diese Kanten zusammen mit den jeweils zwei Kopien von p und u_1, u_2, \dots, u_m zu sehen.

Transformation 15

Durch die Bedingungen, dass es keine Kanten vom Teilnetz B zum Platz p und zum Teilnetz A geben darf, sind die vorangegangenen beiden Transformationen nicht anwendbar, wenn sich eine mögliche mehrfache Ausführung innerhalb eines Zyklus befindet. Für unsere letzte Transformation wählen wir ein Vorgehen ähnlich dem bei Transformation 10. Wir suchen ein geeignetes Teilnetz innerhalb eines Zyklus, transformieren dies unabhängig vom restlichen oWFN und fügen anschließend an Stelle des Teilnetzes eine annotierte

Transition in das oWFN ein. Ein Zyklus muss wegen unserer Einschränkungen in Abschnitt 2.1.3 einem bestimmten Aufbau folgen und aus einer alternierenden Abfolge von Plätzen $p_1, p_2, \dots, p_n \in P$ und Teilnetzen $A_1, A_2, \dots, A_n \subset (T \cup P)$ mit $n \in \mathbb{N}$ bestehen. Jedes Teilnetz A_i muss dabei zusammen mit zwei Plätzen p_a und p_b , für die gilt:

- $\bullet p_a = p_b \bullet = \emptyset$,
- $p_a \bullet = p_i \bullet \cap A_i$ und
- $\bullet p_b = \bullet p_{i-1} \cap A_i$,

einer Anfangsmarkierung $m_0 = [p_a]$ und nur einer Endmarkierung $m_\Omega = [p_b]$ die Soundness-Eigenschaft erfüllen. Bei den Plätzen p_1, p_2, \dots, p_n handelt es sich um die Ein- und Ausgänge des Zyklus, so wie wir sie in Abschnitt 3.2.6 eingeführt haben. Abbildung 3.17 zeigt auf der linken Seite einen Ausschnitt aus einem Zyklus. Zwischen den Plätzen p_1, p_2 und p_3 , wir haben ihren Status als Ein- und Ausgänge in diesem Beispiel durch jeweils eine ein- und eine ausgehende Kante symbolisiert, befinden sich die durch Wolken symbolisierten Teilnetze A_1 und A_2 .

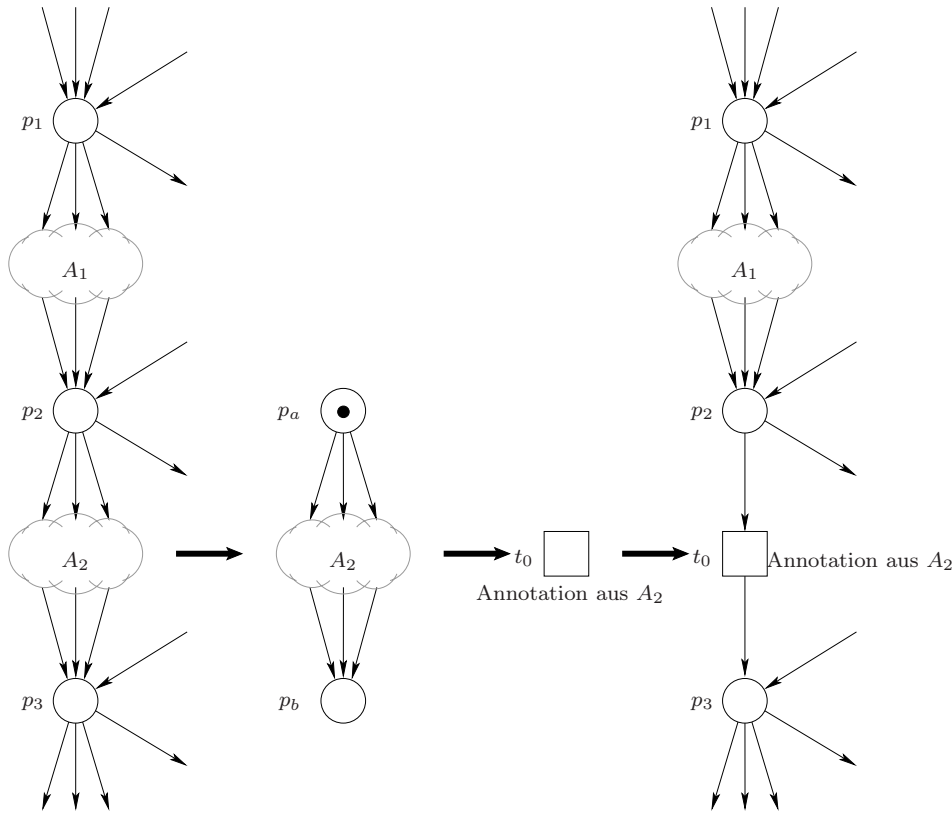


Abbildung 3.17.: Beispiel für Transformation 15 bei mehrfachen Ausführungen auf Zyklen.

Da sich unsere gesamte Transformation mit oWFN befasst, welche die Soundness-Eigenschaft erfüllen, können wir die einzelnen Transformationen auf ein Teilnetz A_i mit den zusätzlichen Plätzen p_a und p_b anwenden. Auf ein solches Teilnetz konnten im bisherigen Transformationsverlauf nur die Transformationen 1, 2, 3, 6, 8, 10, 11, 12, 16, 17 und diese

Transformation selbst angewendet werden. Wurde diese Transformation bereits angewendet, besteht das Teilnetz nur noch aus einer Transition. Wie wir in Anhang A.1 zeigen, erhalten die Ausführungen der anderen aufgeführten Transformationen die Soundness-Eigenschaft für das Teilnetz.

Da sich das Teilnetz nun nicht mehr im Inneren eines Zyklus befindet, können nun auch alle anderen Transformationen angewendet werden. Wir verwenden die Transformationen 2–17. Eine erneute Transformation des Interface ist für das Teilnetz nicht mehr notwendig. Die Transformation wird so lange fortgesetzt, bis das Teilnetz auf einen annotierten Knoten reduziert ist. Die dabei entstehende Annotation dieses Knotens wird auf eine neue Transition t_0 übertragen, die an Stelle des Teilnetzes in den Zyklus im oWFN eingefügt wird. Für die Transition gilt $\bullet t_0 = \{p_i\}$ und $t_0 \bullet = \{p_{i+1}\}$. A_i wird aus dem oWFN entfernt. Weiter gilt nun $p_i \bullet = \bullet p_{i+1} = \{t_0\}$.

Der zweite Schritt des Beispiels in Abb. 3.17 zeigt das Teilnetz A_2 zusammen mit den beiden Plätzen p_a und p_b . Im dritten Schritt ist die annotierte Transition t_0 dargestellt und im vierten ist diese in das oWFN eingefügt worden.

3.2.8. Strukturelle Umformungen des oWFN

In diesem Abschnitt werden wir mit Hilfe zweier Transformationen Strukturen im oWFN umformen, ohne dabei deren Annotationen zu verändern. Um eine vollständige Umwandlung eines oWFN in WS-BPEL-Code zu ermöglichen sind diese beiden Transformationen nicht notwendig. Sie ermöglichen es jedoch die anderen Transformationen gezielter anzuwenden und so das oWFN zu klarer strukturierterem und kompakterem WS-BPEL-Code umzuwandeln. Zwar werden wir die Transformationen jeweils gleich zu Beginn unserer Transformation ausführen, jedoch hilft die Kenntnis der anderen Transformation beim Verständnis der Nützlichkeit dieser Transformationen, so dass wir sie erst jetzt vorstellen werden. Am besten werden die genannten Verbesserungen durch diese beiden strukturellen Umformungen anhand eines Beispiels deutlich. Abbildung 3.18 zeigt auf der linken Seite unseres Beispiels ein Ausgangsnetz und rechts daneben die Schritte einer Transformation dieses oWFN, wie sie durch die Anwendung einer strukturellen Umformung ermöglicht wird.

Einzig Transformation 10 wäre beim Ausgangsnetz anwendbar und würde eine `flow`-Aktivität mit drei `if`-Aktivitäten und 15 Links erzeugen. Wenden wir jedoch eine der beiden strukturellen Umformungen an, die wir in diesem Abschnitt vorstellen werden, erhalten wir ein Petrinetz in dem die gleichen Abläufe möglich und auf das die Transformationen 6 und 8 anwendbar sind. Diese erzeugen eine einfache `if`- und eine `flow`-Aktivität ohne Links. In unserem Beispiel führen wir zuerst Transformation 8 und in einem weiteren Schritt Transformation 6 aus. Nun kann das entstandene Netz durch zwei Ausführungen der Transformation 2 oder durch eine Ausführung der Transformation 2 und eine der Transformation 3 auf einen Platz reduziert werden. Der bei dieser Transformation erzeugte WS-BPEL-Code würde aus einer `sequence`-Aktivität bestehen, in die erst eine `flow`- und anschließend eine `if`-Aktivität eingebettet ist. Das so erzielte Ergebnis ist also übersichtlicher als die zuvor erwähnte `flow`-Aktivität mit drei `if`-Aktivitäten und 15 Links, die ohne diese strukturelle Umformung erzeugt worden wäre. Codebei-

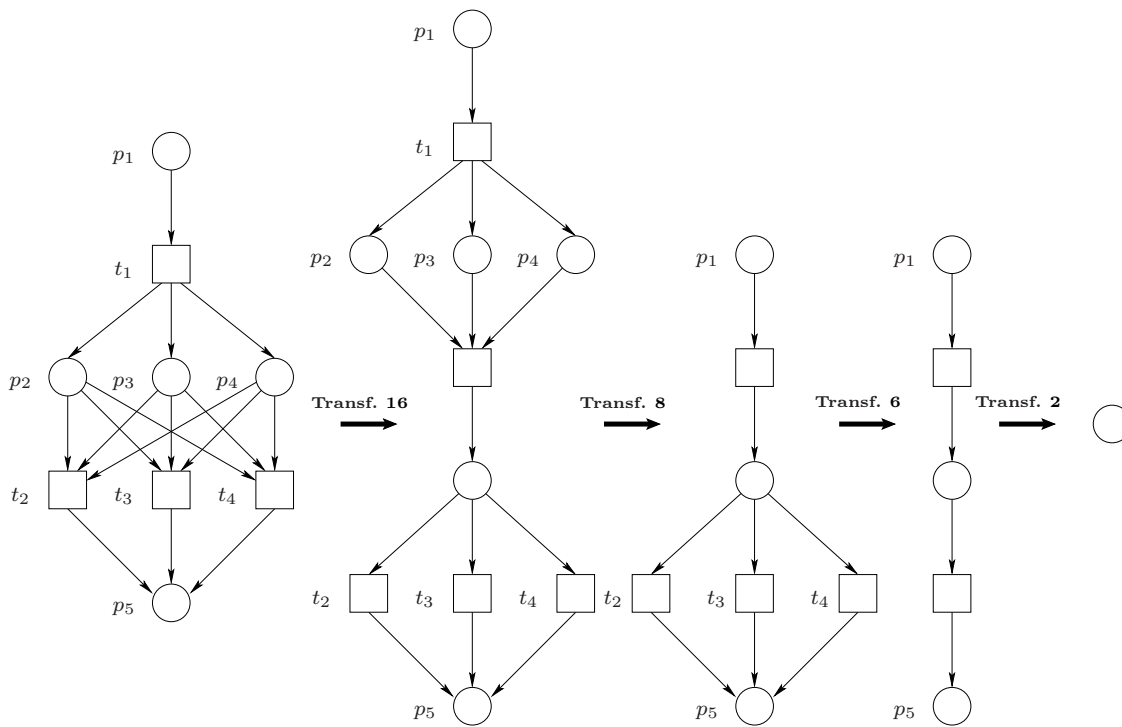


Abbildung 3.18.: Beispiel für die Nützlichkeit von strukturellen Umformungen.

```

<process>
  <sequence>
    <flow>
      <opaqueActivity />
      <opaqueActivity />
      <opaqueActivity />
    </flow>
    <if>
      <opaqueActivity />
    <elseif>
      <opaqueActivity />
    </elseif>
    <else>
      <opaqueActivity />
    </else>
  </if>
</sequence>
</process>

```

Codebeispiel 3.29: Der WS-BPEL-Code der Beispieltransformation.

spiel 3.29 zeigt den WS-BPEL-Code, den die Transformation im Beispiel erzeugen würde, in einer abgekürzten Schreibweise. Wir haben im Codebeispiel der Übersicht halber auf die Benennung der Elemente, die Vergabe von Werten und einige Attribute verzichtet. Ein Codebeispiel für das Ergebnis der Transformation ohne eine vorherige strukturelle Umformung umfasst in dieser abgekürzten Schreibweise 135 Zeilen und findet sich in Anhang A.2. Die erste strukturelle Umformung wird es erlauben, dass wir, wie gerade am Beispiel gesehen, bei der Transformation stellenweise auf die Verwendung von `flow`-Aktivitäten und unübersichtlichen Links verzichten können. Die Anwendung der zweiten strukturellen Umformung wird je nachdem, ob das Teilnetz sicher ist oder nicht, entweder ebenfalls `flow`-Aktivitäten einsparen oder die Anzahl der Kopiervorgänge von Teilnetzen durch die Transformationen in Abschnitt 3.2.7 reduzieren.

Transformation 16

Bei der ersten Transformation fügen wir einen zusätzlichen Platz und eine zusätzliche Transition zwischen eine Menge von m Transitionen und einer Menge von n Plätzen ein. Die Transitionen besitzen alle einen identischen Vorbereich, der aus der Menge der Plätze besteht. Seien $p_1, p_2, \dots, p_n \in P$, $t_1, t_2, \dots, t_m \in T$ und $n, m \in \mathbb{N}$. Wenn gilt, dass

- $n > 1$,
- $m > 1$,
- $\{t_1, t_2, \dots, t_m\} \in \bigcap_{p \in \{p_1, p_2, \dots, p_n\}} p^\bullet$ und
- $\bullet t_1 = \bullet t_2 = \dots = \bullet t_m = \{p_1, p_2, \dots, p_n\}$

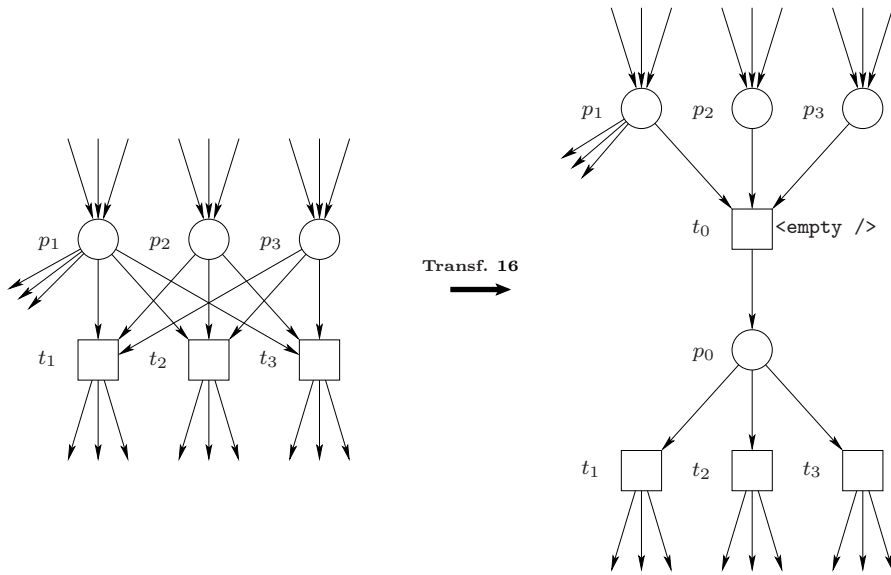
dann können wir eine neue Transition t_0 mit einer `empty` Annotation und einen neuen Platz p_0 einfügen. Für diese gilt:

- $\bullet t_0 = \{p_1, p_2, \dots, p_n\}$,
- $t_0^\bullet = \{p_0\}$,
- $\bullet p_0 = \{t_0\}$ und
- $p_0^\bullet = \{t_1, t_2, \dots, t_m\}$.

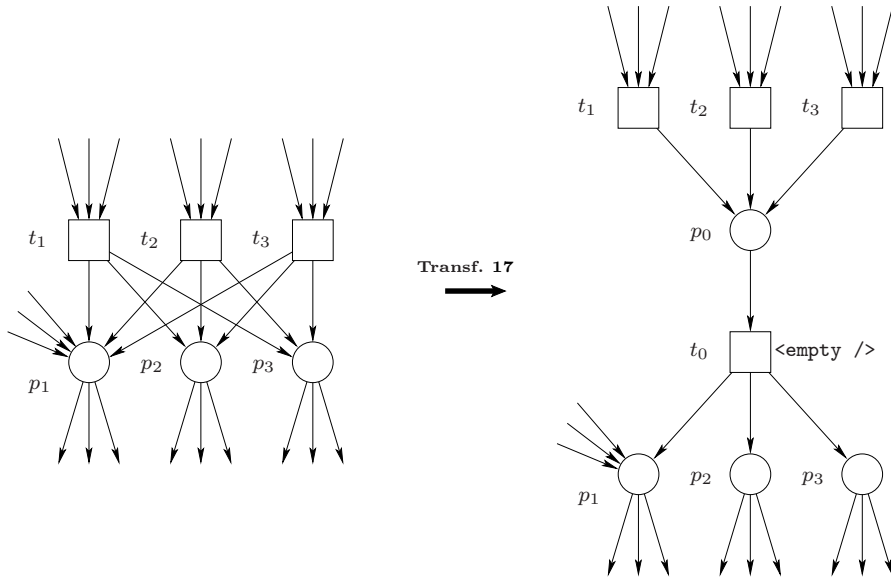
Weiter wird

- $p_1^\bullet = (p_1^\bullet \cup \{t_0\}) \setminus \{t_1, t_2, \dots, t_m\}$, $p_2^\bullet = (p_2^\bullet \cup \{t_0\}) \setminus \{t_1, t_2, \dots, t_m\}$, \dots , $p_n^\bullet = (p_n^\bullet \cup \{t_0\}) \setminus \{t_1, t_2, \dots, t_m\}$ und
- $\bullet t_1 = \bullet t_2 = \dots = \bullet t_m = \{p_0\}$.

Abbildung 3.19(a) zeigt ein Beispiel mit $n = 3$ und $m = 3$ für diese strukturelle Umformung. Aus Gründen der Übersichtlichkeit sind in der Abbildung weitere Transitionen nur im Nachbereich von Platz p_1 angedeutet. Auf der linken Seite ist ein Ausschnitt aus dem oWFN vor der Umformung zu sehen und auf der rechten Seite der gleiche Ausschnitt nach der Umformung mit dem neuen Platz p_0 und der neuen Transition t_1 .



(a) Beispiel für Transformation 16 mit drei Plätzen und drei Transitionen.



(b) Beispiel für Transformation 17 mit drei Plätzen und drei Transitionen.

Abbildung 3.19.: Beispiele für die strukturellen Umformungen.

Vor der Transformation müssen alle Plätze p_1, p_2, \dots, p_n belegt sein, damit eine der Transitionen t_1, t_2, \dots, t_m schalten kann. Nach der Transformation müssen p_1, p_2, \dots, p_n belegt sein, damit die neu eingefügte Transition t_0 schalten kann. Dabei wird eine Marke auf dem neuen Platz p_0 produziert, die von einer der Transitionen t_1, t_2, \dots, t_m konsumiert werden kann, um zu schalten. Bis auf das zusätzliche Schalten von t_0 hat sich somit am Ablauf im oWFN nichts geändert.

Transformation 17

Bei der zweiten Transformation fügen wir ebenfalls einen zusätzlichen Platz und eine zusätzliche Transition zwischen eine Menge von m Transitionen und einer Menge von n Plätzen ein. Die Transitionen besitzen diesmal alle einen identischen Nachbereich, der aus der Menge der Plätze besteht. Seien $p_1, p_2, \dots, p_n \in P$, $t_1, t_2, \dots, t_m \in T$ und $n, m \in \mathbb{N}$. Wenn gilt, dass

- $n > 1$,
- $m > 1$,
- $\{t_1, t_2, \dots, t_m\} \in \bigcap_{p \in \{p_1, p_2, \dots, p_n\}} \bullet p$ und
- $t_1^\bullet = t_2^\bullet = \dots = t_m^\bullet = \{p_1, p_2, \dots, p_n\}$

dann können wir eine neue Transition t_0 mit einer **empty** Annotation und einen neuen Platz p_0 einfügen. Für diese gilt:

- $t_0^\bullet = \{p_1, p_2, \dots, p_n\}$,
- $\bullet t_0 = \{p_0\}$,
- $p_0^\bullet = \{t_0\}$ und
- $\bullet p_0 = \{t_1, t_2, \dots, t_m\}$.

Weiter wird

- $\bullet p_1 = (\bullet p_1 \cup \{t_0\}) \setminus \{t_1, t_2, \dots, t_m\}$, $\bullet p_2 = (\bullet p_2 \cup \{t_0\}) \setminus \{t_1, t_2, \dots, t_m\}, \dots, \bullet p_n = (\bullet p_n \cup \{t_0\}) \setminus \{t_1, t_2, \dots, t_m\}$ und
- $t_1^\bullet = t_2^\bullet = \dots = t_m^\bullet = \{p_0\}$.

Abbildung 3.19(b) zeigt ein Beispiel mit $n = 3$ und $m = 3$ für diese strukturelle Umformung. Aus Gründen der Übersichtlichkeit sind in der Abbildung weitere Transitionen nur im Nachbereich von Platz p_1 angedeutet. Auf der linken Seite ist ein Ausschnitt aus dem oWFN vor der Umformung zu sehen und auf der rechten Seite der gleiche Ausschnitt nach der Umformung mit dem neuen Platz p_0 und der neuen Transition t_1 .

Vor der Transformation konnten alle Transitionen t_1, t_2, \dots, t_m unabhängig voneinander schalten und dadurch jeweils eine Marke auf jedem der Plätze p_1, p_2, \dots, p_n produzieren. Nach der Transformation können immer noch alle Transitionen unabhängig voneinander schalten und produzieren nun jeweils eine Marke auf dem neuen Platz p_0 . Anschließend kann durch das Konsumieren dieser Marke die neue Transition t_0 schalten und produziert

jeweils eine Marke auf jedem der Plätze p_1, p_2, \dots, p_n . Wie bei der vorherigen Transformation hat sich bis auf das zusätzliche Schalten von t_0 am Ablauf im oWFN nichts geändert.

Wie bereits bei anderen Transformationen zuvor annotieren wir bei diesen beiden Transformationen die neuen Transitionen t_0 mit einer **empty**-Aktivität, da sie ursprünglich nicht im oWFN vorhanden waren und nach Möglichkeit nicht im WS-BPEL-Code auftauchen sollen.

3.3. Transformationen des WS-BPEL-Codes

Wir haben im vorherigen Abschnitt bereits einige Transformationen des WS-BPEL-Codes erläutert. In den Abschnitten 4–3.3.5 betrachten wir nun die anderen abschließenden Veränderungen der erzeugten Annotationen. Diese Korrekturen müssen im Gegensatz zu den anderen Transformationen nur einmal durchgeführt werden.

3.3.1. Process-Aktivität

Jeder WS-BPEL-Prozess beginnt mit der Ausführung der strukturierten Aktivität **process**. Bisher ist diese Aktivität in unseren Annotationen noch nicht vorhanden. Diesen Umstand ändern wir mit der ersten Korrektur.

Korrektur 4

Die gesamten, durch die zuvor beschriebenen Transformationen entstandenen, Annotationen werden in die neue Aktivität **process** eingebettet. In Codebeispiel 3.30 haben wir den WS-BPEL-Code abgebildet, der aus einer **process** Annotation entsteht. Neben dem Namen des Prozesses, für den wir im Beispiel den Bezeichner *Act_process* gewählt haben, legen wir mit den Attributen *targetNamespace*, *xmlns*, *xmlns:template* und *abstractProcessProfile* fest, dass es sich um einen abstrakten Prozess handelt. Wie wir im Zusammenhang mit Transformation 10 erklärt haben, setzen wir innerhalb der **process**-Aktivität das Attribut *suppressJoinFailure*. Weitere in WS-BPEL optionale Attribute der **process**-Aktivität haben wir nicht mit angegeben. Außerdem werden innerhalb der **process** die Elemente `<partnerLinks>` und `<variables>` definiert, die im Zusammenhang mit dem Interface benötigt werden und die wir in Abschnitt 3.2.1 erläutert haben. Im Beispiel haben wir eine Variable angelegt.

3.3.2. Empty-Aktivitäten

Während unserer Transformationen haben wir an mehreren Stellen **empty**-Aktivitäten eingefügt, um so Knoten zu kennzeichnen, die im ursprünglichen oWFN nicht vorhanden waren oder um als Quelle und Ziel von Links innerhalb von **flow**-Aktivitäten zur Verfügung zu stehen. Mit Hilfe dieser Korrektur werden wir nun diese **empty**-Aktivitäten teilweise wieder entfernen, so dass sie nicht im erzeugten WS-BPEL-Code auftauchen.

```
<process name="Act_process"
  targetNamespace="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
  xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
  suppressJoinFailure="yes"
  xmlns:template="http://docs.oasisopen.org/wsbpel/2.0/process/abstract/simple-template/2006/08"
  abstractProcessProfile="http://docs.oasisopen.org/wsbpel/2.0/process/abstract/simple-template/2006/08">
  <partnerLinks>
    <partnerLink name="generic_pl"
      partnerLinkType="##opaque"
      myRole="##opaque"
      partnerRole="##opaque" />
  </partnerLinks>
  <variables>
    <variable name="Var_A"
      element="##opaque" />
  </variables>
  eingebettete Aktivitäten
</process>
```

Codebeispiel 3.30: Der WS-BPEL-Code einer `process`-Aktivität.

Korrektur 5

Eine `empty`-Aktivität wird aus den Annotationen entfernt, wenn sie eine von drei Bedingungen erfüllt. Für die erste Bedingung darf die Aktivität nicht Quelle und Ziel von Links sein. Zudem darf durch ihr Entfernen kein leerer Zweig in einer strukturierten Aktivität oder eine `while`-Aktivität, die keine Aktivität einbettet, entstehen. In Codebeispiel 3.31 ist die `empty`-Aktivität zwischen den beiden `opaqueActivity`-Aktivitäten weder Quelle noch Ziel von Links und kann somit einfach entfernt werden.

```
beliebige Aktivitäten
<opaqueActivity />
<empty />
<opaqueActivity />
beliebige Aktivitäten
```

Codebeispiel 3.31: Durch Korrektur 5 entfernbare `empty`-Aktivität.

Wenn eine Aktivität, auf die eine `empty`-Aktivität folgt, nicht Quelle von Links ist und die `empty`-Aktivität nicht Ziel von Links, dann wird die `empty`-Aktivität entfernt, und alle Links, deren Quelle die `empty`-Aktivität war, werden abgeändert, so dass nun die andere Aktivität ihre Quelle ist. In Codebeispiel 3.32 ist die Aktivität `opaqueActivity` nicht Quelle von Links und die `empty`-Aktivität nicht Ziel von Links. Stattdessen ist sie Quelle des Links *i*. In Codebeispiel 3.33 ist nun die Aktivität `opaqueActivity` Quelle des Links *i* und die `empty`-Aktivität wurde entfernt.

```
beliebigen Aktivitäten
<opaqueActivity />
<empty>
  <source "i" />
</empty>
beliebigen Aktivitäten
```

Codebeispiel 3.32: Eine entfernbare `empty`-Aktivität.

```
beliebigen Aktivitäten
<opaqueActivity>
  <source "i" />
</opaqueActivity>
beliebigen Aktivitäten
```

Codebeispiel 3.33: Das Beispiel nach Korrektur 5.

Wenn eine Aktivität auf eine `empty`-Aktivität folgt und nicht Ziel von Links ist und die `empty`-Aktivität nicht Quelle von Links, dann wird die `empty` Aktivität entfernt und alle Links, deren Ziel die `empty`-Aktivität war, werden abgeändert, so dass nun die andere Aktivität ihr Ziel ist. Zudem wird die Aktivierungsbedingung der `empty`-Aktivität auf die andere Aktivität übertragen. In Codebeispiel 3.34 ist die Aktivität `opaqueActivity` nicht Ziel von Links und die `empty`-Aktivität nicht Quelle von Links. Stattdessen ist sie Ziel des Links `i`. In Codebeispiel 3.35 ist nun die Aktivität `opaqueActivity` Ziel des Links `i` und die `empty`-Aktivität wurde entfernt.

```
beliebigen Aktivitäten
<empty>
  <target "i" />
</empty>
<opaqueActivity />
beliebigen Aktivitäten
```

Codebeispiel 3.34: Eine entfernbare `empty`-Aktivität.

```
beliebigen Aktivitäten
<opaqueActivity>
  <target "i" />
</opaqueActivity>
beliebigen Aktivitäten
```

Codebeispiel 3.35: Das Beispiel nach Korrektur 5.

3.3.3. Flow-Aktivitäten

Bei der Anwendung der Transformationen aus Abschnitt 3.2.4 kann es passieren, dass ein Teilnetz durch eine `flow`-Aktivität ersetzt wird, in dem an einem der Knoten bereits eine andere `flow`-Aktivität annotiert ist. Wird dabei aus der vorhandenen `flow`-Aktivität ein eigener Zweig in der neu eingefügten, der nicht Quelle oder Ziel von Links ist, dann können wir die beiden `flow`-Aktivitäten zu einer zusammenfassen. Dies geschieht, indem wir die Zweige der eingebetteten `flow`-Aktivität als Zweige in die einbettende einfügen. Da in der gesamten `flow`-Aktivität die Bezeichnungen von Links eindeutig sein müssen, haben wir bei der Erzeugung der Links zur Benennung der Bezeichner alle Links durchnummeriert. Und wir haben in allen Transformationen, die Kopien von Annotationen erzeugen, die Bezeichner der Links innerhalb der kopierten `flow`-Aktivitäten geändert.

Korrektur 6

Wenn einer der Zweige einer `flow`-Aktivität `A` nur aus einer weiteren `flow`-Aktivität `B` besteht und diese nicht Quelle oder Ziel von Links ist, dann entfernen wir die `flow`-Aktivität `B` und fügen alle ihre Zweige als Zweige in die `flow`-Aktivität `A` ein. Auf Grund der Art, wie wir Links nur innerhalb einzelner `flow`-Aktivitäten erzeugt haben,

kann keine der Aktivitäten innerhalb der `flow`-Aktivität B Quelle oder Ziel von einem Link sein, der die Grenzen der `flow`-Aktivität überschreitet. Codebeispiel 3.36 zeigt eine `flow`-Aktivität mit drei Zweigen. Zweig A_2 besteht nur aus einer weiteren `flow`-Aktivität mit zwei eigenen Zweigen. In Codebeispiel 3.37 ist die `flow`-Aktivität nach der Korrektur abgebildet. Zweig A_2 wurde aus der Aktivität entfernt und die Zweige B_1 und B_2 in die Aktivität eingefügt.

```
<flow>
  Zweig  $A_1$  mit beliebigen Aktivitäten
  Zweig  $A_2$  mit:
  <flow>
    Zweig  $B_1$  mit beliebigen Aktivitäten
    Zweig  $B_2$  mit beliebigen Aktivitäten
  </flow>
  Zweig  $A_3$  mit beliebigen Aktivitäten
</flow>
```

Codebeispiel 3.36: Eine `flow`-Aktivität, die eine weitere einbettet.

```
<flow>
  Zweig  $A_1$  mit beliebigen Aktivitäten
  Zweig  $B_1$  mit beliebigen Aktivitäten
  Zweig  $B_2$  mit beliebigen Aktivitäten
  Zweig  $A_3$  mit beliebigen Aktivitäten
</flow>
```

Codebeispiel 3.37: Die `flow`-Aktivität nach Korrektur 6.

3.3.4. Links

Bei der Erzeugung von `flow`-Aktivitäten wird jeder Knoten des transformierten Teilnetzes als Zweig in die Aktivität eingebettet. Dabei können überflüssige Links erzeugt werden. Diese Korrektur entfernt diese Links wieder aus den Annotationen.

Korrektur 7

Wenn eine Aktivität, die in eine `flow`-Aktivität eingebettet ist, zum einen die letzte Aktivität in einem Zweig der `flow`-Aktivität und zum anderen Quelle von genau einem Link ist und die Aktivität, die Ziel dieses Links ist, nicht Ziel von weiteren Links und zugleich die erste Aktivität eines Zweiges derselben `flow`-Aktivität ist, dann entfernen wir den Link und den Zweig der zweiten Aktivität und fügen den gesamten Inhalt des Zweiges hinter die erste Aktivität ein. Codebeispiel 3.38 zeigt eine `flow`-Aktivität mit drei Zweigen. Zweig 1 endet mit einer `empty`-Aktivität, die Quelle von Link i ist und Zweig 3 beginnt mit einer `empty`-Aktivität die Ziel von Link i ist. Beide `empty`-Aktivitäten sind weder Quelle noch Ziel von weiteren Links. In Codebeispiel 3.39 ist die `flow`-Aktivität nach der Korrektur zu sehen. Die Aktivitäten aus Zweig 3 wurden an Zweig 1 angehängt.

```

<flow>
  Zweig 1 mit beliebigen Aktivitäten und
  <empty>
    <source "i" />
  </empty>
  Zweig 2 mit beliebigen Aktivitäten
  Zweig 3 mit
  <empty>
    <target "i" />
  </empty>
  weitere Aktivitäten von Zweig 3
</flow>

```

Codebeispiel 3.38: Eine `flow`-Aktivität mit einem entfernbaren Link.

```

<flow>
  Zweig 1 mit beliebigen Aktivitäten und
  <empty />
  <empty />
  weitere Aktivitäten von Zweig 3
  Zweig 2 mit beliebigen Aktivitäten
</flow>

```

Codebeispiel 3.39: Die `flow`-Aktivität nach Korrektur 7.

3.3.5. Sequence-Aktivitäten

Wir haben bereits in Abschnitt 3.2.2 erläutert, weswegen wir die strukturierte Aktivität `sequence` erst nachträglich in die von uns erzeugten Annotationen mit einfügen.

```

<sequence>
  <opaqueActivity />
  <flow />
  <while />
  <if />
  <opaqueActivity />
</sequence>

```

Codebeispiel 3.40: Ein Beispiel für eine durch Korrektur 8 eingefügte `sequence`-Aktivität.

Korrektur 8

Folgen in den Annotationen zwei oder mehr Aktivitäten aufeinander, sind also nicht ineinander eingebettet, werden sie in eine `sequence`-Aktivität eingebettet. Codebeispiel 3.40 zeigt ein Beispiel für eine erzeugte `sequence`-Aktivität, in die fünf aufeinander folgende Aktivitäten eingebettet wurden.

3.4. Ausführungsreihenfolge der Transformationen

Nach der Beschreibung aller Transformationen und Korrekturen, betrachten wir nun deren Ausführungsreihenfolge. Wie wir bereits in 3.2 angesprochen haben, stehen die Transformationen und Korrekturen zum Teil zueinander in Konflikt. Daher ist die Festlegung einer Reihenfolge notwendig.

Die Ausführung beginnt mit der einmaligen Reduktion des Interfaces mit Hilfe von **Transformation 1**. Nach Beendigung der Transformation ist das gesamte Interface des

oWFN entfernt. Nun folgen die weiteren Transformationen. Diese werden so lange ausgeführt, bis das oWFN auf einen Knoten reduziert worden ist. Dabei wird diejenige Transformation in der Liste gesucht, welche sich als erste auf das oWFN anwenden lässt. Diese wird entweder einmal oder mehrfach ausgeführt, und anschließend beginnt die Suche von vorn.

1. **Transformationen 16 und 17:** Die beiden strukturellen Umformungen aus Abschnitt 3.2.8 werden zuerst ausgeführt. Sie ermöglichen die Ausführung weiterer Transformationen und stehen nicht in Konflikt zueinander. Jede der Transformationen wird so oft ausgeführt, wie sie angewendet werden kann.
2. **Transformationen 2–5:** Die Transformationen, die Sequenzen im oWFN zusammenfassen, stehen zueinander in Konflikt. Sie können jedoch, wie in Abschnitt 3.2.2 gezeigt, in beliebiger Reihenfolge ausgeführt werden. Jede der Transformationen wird so oft ausgeführt, wie sie angewendet werden kann.
3. **Transformationen 8 und 9:** Die beiden Transformationen aus Abschnitt 3.2.4 für nebenläufige Ausführungen stehen weder untereinander noch mit einer der bisherigen Transformationen in Konflikt, so dass auch sie so oft ausgeführt werden, wie sie angewendet werden können.
4. **Transformation 11:** Die Transformation von Zyklen steht ebenfalls nicht im Konflikt mit den bisherigen Transformationen und wird so oft ausgeführt, wie sie angewendet werden kann.
5. **Transformationen 6 und 7:** Zwar stehen die Transformationen nicht im Konflikt mit den bisherigen Transformationen, jedoch müssen weniger Transformationen und Korrekturen ausgeführt werden, wenn Transformation 7 erst nach den bisherigen ausgeführt wird. Der Grund dafür ist, dass dann von der Transformation unter Umständen größere Teilnetze reduziert werden können.
6. **Transformation 12:** Die Transformation von Zyklen mit mehreren Eingangsplätzen steht im Konflikt mit Transformation 7. Würde diese Transformation nicht erst nach Transformation 7 ausgeführt, würde bedingtes Verhalten innerhalb eines Zyklus als zusätzliche Eingangsplätze interpretiert. Die Transformation wird nach einer Ausführung beendet, damit das strukturell veränderte oWFN auf die erneute Ausführung der bisherigen Transformationen überprüft werden kann.
7. **Transformation 10:** Das Ersetzen sicherer und zyklensfreier Teilnetze, wie wir es in Abschnitt 3.2.4 beschreiben, steht im Konflikt mit fast allen bisherigen Transformationen, deren Suchmuster Teil des von dieser Transformation reduzierten Teilnetzes sein können. Daher wird sie ebenfalls bereits nach einer Ausführung beendet.
8. **Transformation 13:** Die Transformationen, die Kopien von Teilen des oWFN erzeugen, stehen mit den bisherigen im Konflikt. Sie können erst nach Transformation 10 ausgeführt werden, da diese die sicheren Teilnetze aus dem oWFN entfernt und somit keine unnötigen Kopien von Teilnetzen erzeugt werden. Sie werden nach einer Ausführung beendet.

9. **Transformation 14:** Da diese mit der vorherigen Transformation im Konflikt steht und die vorherige einfachere Strukturen erzeugt, wird sie erst nach ihr ausgeführt. Auch sie wird nach einer Ausführung beendet.
10. **Transformation 15:** Durch die Positionierung der Transformation, die nicht sichere Teilnetze innerhalb eines Zyklus reduziert, hinter Transformation 10 wird diese Transformation nicht unnötig auf Teilnetze angewendet, die sicher sind.

Nachdem das oWFN durch die Transformationen auf einen Knoten reduziert worden ist, werden die Korrekturen auf der Annotation des Knotens durchgeführt. Dabei muss jede Korrektur nur einmal so oft ausgeführt werden, wie sie angewendet werden kann.

1. **Korrektur 4:** Die `process`-Aktivität wird zuerst eingefügt. Diese Korrektur steht nicht im Konflikt mit den anderen und kann daher beliebig positioniert werden. Die Ausführung zu Beginn der Korrekturen bringt lediglich Vorteile bei der Implementierung.
2. **Korrektur 3:** Das Ersetzen mehrerer aufeinander folgender `while`-Aktivitäten steht nur im Konflikt mit der Erzeugung von `<sequence>`-Aktivitäten.
3. **Korrektur 7:** Das Entfernen von Links wird vor dem Entfernen von `empty`-Aktivitäten ausgeführt, da so mehr `empty`-Aktivitäten entfernt werden können.
4. **Korrektur 5:** Diese Korrektur, die `empty`-Aktivitäten entfernt, muss nach den beiden vorherigen und vor allen weiteren Korrekturen ausgeführt werden.
5. **Korrektur 6:** Das Zusammenführen mehrerer `flow`-Aktivitäten kann nach den vorherigen beiden Korrekturen unter Umständen häufiger ausgeführt werden.
6. **Korrektur 2:** Auch das Zusammenführen mehrerer `if`-Aktivitäten profitiert vom vorherigen Entfernen der `empty`-Aktivitäten.
7. **Korrektur 1:** Die Erzeugung der `pick`-Aktivitäten steht im Konflikt mit der vorherigen Korrektur und muss nach dieser ausgeführt werden.
8. **Korrektur 8:** Die Korrektur, die `sequence`-Aktivitäten einfügt, muss zuletzt ausgeführt werden. Die Zusammenfassungen von Aktivitäten, die die anderen Korrekturen durchführen, würden ansonsten eingefügte `<sequence>`-Aktivitäten wieder unnötig machen.

Wie wir anfangs in Abschnitt 3.1 erläutert haben, als wir die Idee unserer Transformation vorgestellt haben, wird nach den Korrekturen aus der Annotation der eigentliche WS-BPEL-Code erzeugt, indem die Annotationen jeder Aktivität durch ihren WS-BPEL-Code ersetzt werden. In Kapitel 4 beschäftigen wir uns als nächstes mit der Implementierung der Transformation. Dabei zeigen wir ein ausführliches Beispiel für eine vollständige Transformation eines oWFN zu einem abstrakten WS-BPEL-Prozess.

4. Implementierung und Fallstudie

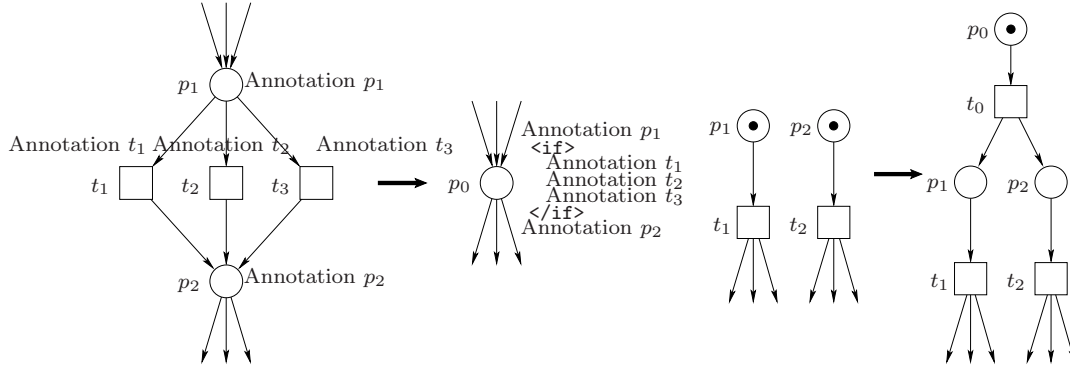
4.1. Implementierung

In diesem Kapitel werden wir auf einige Details und Besonderheiten unserer Implementierung der Transformation eingehen. Wir haben die vorgestellte Transformation im Tool oWFN2BPEL implementiert. Das Werkzeug beherrscht alle von uns vorgestellten Transformationen aus Abschnitt 3.2 und Korrekturen aus Abschnitt 3.3. Es liest ein oWFN ein und gibt für alle Netze, die alle unsere Einschränkungen aus Abschnitt 2.1.3 erfüllen, einen abstrakten WS-BPEL-Prozess aus.

Für den Fall, dass das eingelesene oWFN nicht unseren Einschränkungen genügt, bietet oWFN2BPEL zusätzlich die Möglichkeit, das Netz so weit es mit den Transformationen möglich ist zu reduzieren und ein mit WS-BPEL-Code annotiertes oWFN auszugeben. Dieses kann vom Nutzer weiter verarbeitet werden. Eine weitere Zusatzoption erlaubt es, falls diese nicht benötigt werden, `opaqueActivity`-Aktivitäten aus dem erzeugten WS-BPEL-Code zu entfernen. Dabei gehen wir genauso vor, wie bei der Entfernung der `empty`-Aktivitäten durch Korrektur 5. Auf diese Art wird nur das Kommunikationsverhalten des oWFN übersetzt, ohne dabei alle internen Transitionen zu berücksichtigen.

Zur Optimierung der Laufzeit haben wir zwei zusätzliche Transformationen eingefügt. Die erste transformiert, wie im Beispiel in Abb. 3.12 dargestellt, einen Zyklus, bestehend aus nur einem Platz und einer Kante, zu einer `while`-Aktivität. Dieses Suchmuster kann im oWFN schneller gefunden werden als Zyklen, die aus mehreren Knoten bestehen: Es reicht eine Betrachtung jeder Transition. Diese Transformation wird vor Transformation 11 ausgeführt und ebenfalls so oft angewendet, wie dies möglich ist. Nach erfolgreicher Beendigung der Transformation beginnt, wie bei den anderen Transformationen, die Suche der Ausführungsreihenfolge nach erneut. Die andere zusätzliche Transformation reduziert bedingtes Verhalten auf einen Platz. Das Beispiel in Abb. 4.1(a) stellt dies dar. Die Transformation fasst eine Ausführung der Transformationen 2 und 6 zusammen, falls sich im Nachbereich von Platz p_1 und im Vorbereich von Platz p_2 nur Transitionen befinden, die an der Transformation beteiligt sind. In diesem Fall können alle Knoten direkt auf einen Platz reduziert werden. Wieder ist es einfacher das Suchmuster im oWFN aufzuspüren. Daher wird diese Transformation, ähnlich der zuvor vorgestellten, vor Transformation 6 ausgeführt.

Es wurden von uns zudem zwei Transformation zur strukturellen Umformung implementiert, die es erlauben, ein oWFN einzulesen, das eine Anfangsmarkierung besitzt, die aus mehreren Plätzen mit jeweils nur einer Marke besteht und bei dem Plätze aus der Anfangsmarkierung keinen leeren Vorbereich besitzen. Dazu werden zusätzliche Knoten im Vorbereich dieser Plätze erzeugt, von denen einer der einzige Platz in einer neuen Anfangsmarkierung wird. Abbildung 4.1(b) zeigt ein einfaches Beispiel für eine dieser Zusatzfunktionen. Aus der Ausgangssituation auf der linken Seite mit zwei markierten



(a) Ein Beispiel für die zweite zusätzliche Transformation.

(b) Ein Beispiel für eine strukturelle Umformung durch eine zusätzliche Transformation.

Abbildung 4.1.: Beispiele für zusätzliche Transformationen in oWFN2BPEL.

Plätzen (p_1 und p_2) wird nach der Umformung durch das Einfügen zweier zusätzlicher Knoten (p_0 und t_0) ein Netz mit nur noch einem markierten Platz (p_0), das durch das Schalten der aktivierten Transition t_0 in die Markierung der Ausgangssituation zurückkehren kann.

Der Test auf Sicherheit für ein Teilnetz, den Transformation 10 benötigt, geschieht durch Aufrufe des Petrietz-Model-Checkers LoLA [Sch00].

Es existiert bereits ein Tool namens WorkflowNet2BPEL4WS basierend auf dem Ansatz in [AL05], welches annotierte Workflow-Netze in BPEL4WS 1.1 Prozesse transformiert, jedoch mit Nutzerrückfragen arbeitet und in vielen Fällen auf diese angewiesen ist. Vor der Transformation durch WorkflowNet2BPEL4WS müssen die Workflow-Netze vom Nutzer mit Informationen über Nachrichten und Bedingungen annotiert werden, damit Entscheidungen zur Auswahl von Aktivitäten getroffen werden können. Als Einschränkung für die Workflow-Netze wird lediglich die Erfüllung der Soundness-Eigenschaft und Sicherheit des Netzes gefordert. Zwar erweckt dies den Eindruck, als könnte WorkflowNet2BPEL4WS Zyklen jeglicher Art transformieren, doch ist nur die Transformation von Zyklen bestehend aus einem Platz und einer Transformation, wie wir sie zur Laufzeitoptimierung mit eingefügt haben, implementiert. Größere Zyklen und viele andere Strukturen sind nur durch Rückfragen an den Nutzer des Tools reduzierbar. Bei diesen Rückfragen kann unter Umständen vom Nutzer gefordert werden, den BPEL4WS-Code für das gesamte Workflow-Netz anzugeben. Eine vollautomatische Transformation, wie sie unsere Implementierung bietet, ist somit nur in den seltensten Fällen möglich. Wir werden in Abschnitt 5.2 bei der Betrachtung alternativer Ansätze noch einmal auf WorkflowNet2BPEL4WS eingehen und es in Anhang A.4 an einem Beispiel mit unserer Implementierung vergleichen.

Wir haben unsere Transformation mit Hilfe der unabhängig von dieser Arbeit entwickelten Tools BPEL2oWFN, welches WS-BPEL-Prozesse in die in [Loh07] vorgestellte Petrietzsemantik für WS-BPEL übersetzt, und Fiona [LMSW06], zur Erzeugung

und zum Vergleich von Bedienungsanleitungen, getestet. Wenn ein oWFN mit unserem Tool in einen abstrakten WS-BPEL-Prozess transformiert wurde und anschließend mit BPEL2oWFN zurück in ein oWFN, entstanden dabei nahezu identische oWFN, die sich in den meisten Fällen nur in der Länge von Sequenzen und der Bezeichnung der Knoten unterschieden. Die Bedienungsanleitungen beider oWFN waren in diesen Fällen identisch, selbst wenn sich die Struktur des von BPEL2oWFN erzeugten oWFN von der des ursprünglichen stärker unterschied. Nicht bei jedem Test konnten durch die Kombination dieser Werkzeuge identische Bedienungsanleitungen erzeugt werden. Dies liegt jedoch daran, dass die durch unsere Transformation erzeugten abstrakten WS-BPEL-Prozesse um Bedingungen ergänzt werden müssen, die bei der erneuten Übersetzung mit BPEL2oWFN nicht berücksichtigt werden.

Wir betrachten als Nächstes eine Fallstudie, die einige der Transformationen und Korrekturen an einem längeren Beispiel zeigt und die Fähigkeiten unserer Transformation demonstrieren soll.

4.2. Fallstudie

Wir betrachten nun die schrittweise Transformation des in Abbildung 4.2 dargestellten oWFN mit elf Plätzen und zwölf Transitionen. Zusätzlich besitzt das Netz ein Interface bestehend aus den beiden Eingangsplätzen A und B und den vier Ausgangsplätzen C, D, E und F . Die Anfangsmarkierung ist $m_0 = [p_1]$. Die Menge der Endmarkierungen enthält zwei Elemente: $\Omega = \{[p_8, p_{11}], [p_{10}, p_{11}]\}$. Das oWFN enthält einen Zyklus mit dem Eingang p_5 und den beiden Ausgängen p_6 and p_7 .

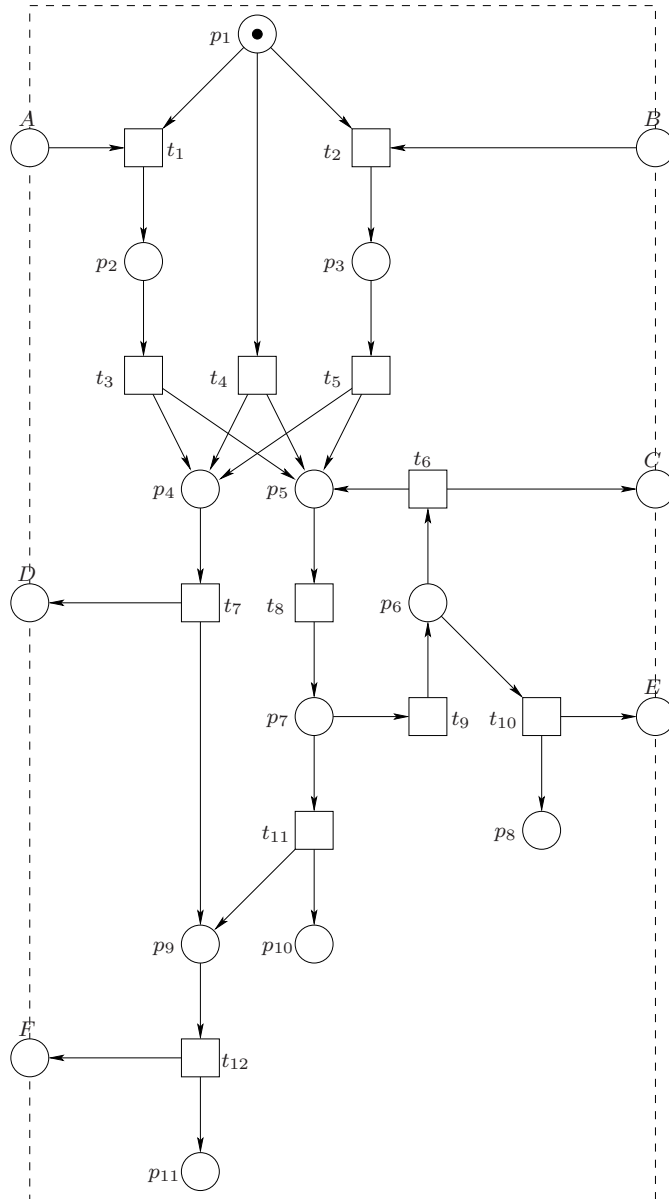


Abbildung 4.2.: Ein oWFN als Ausgangsbasis für unsere Transformation.

Wir beginnen damit, die Transformationen gemäß der in Abschnitt 3.4 festgelegten Ausführungsreihenfolge auf ihre Anwendbarkeit hin zu untersuchen. Da Interfaceplätze vorhanden sind, können wir einmalig Transformation 1 ausführen. Abbildung 4.3 zeigt das transformierte oWFN. Die Interfaceplätze wurden entfernt und `receive`- und `invoke`-Aktivitäten eingefügt. Wir kennzeichnen alle bereits annotierten Knoten mit einer grauen Färbung.

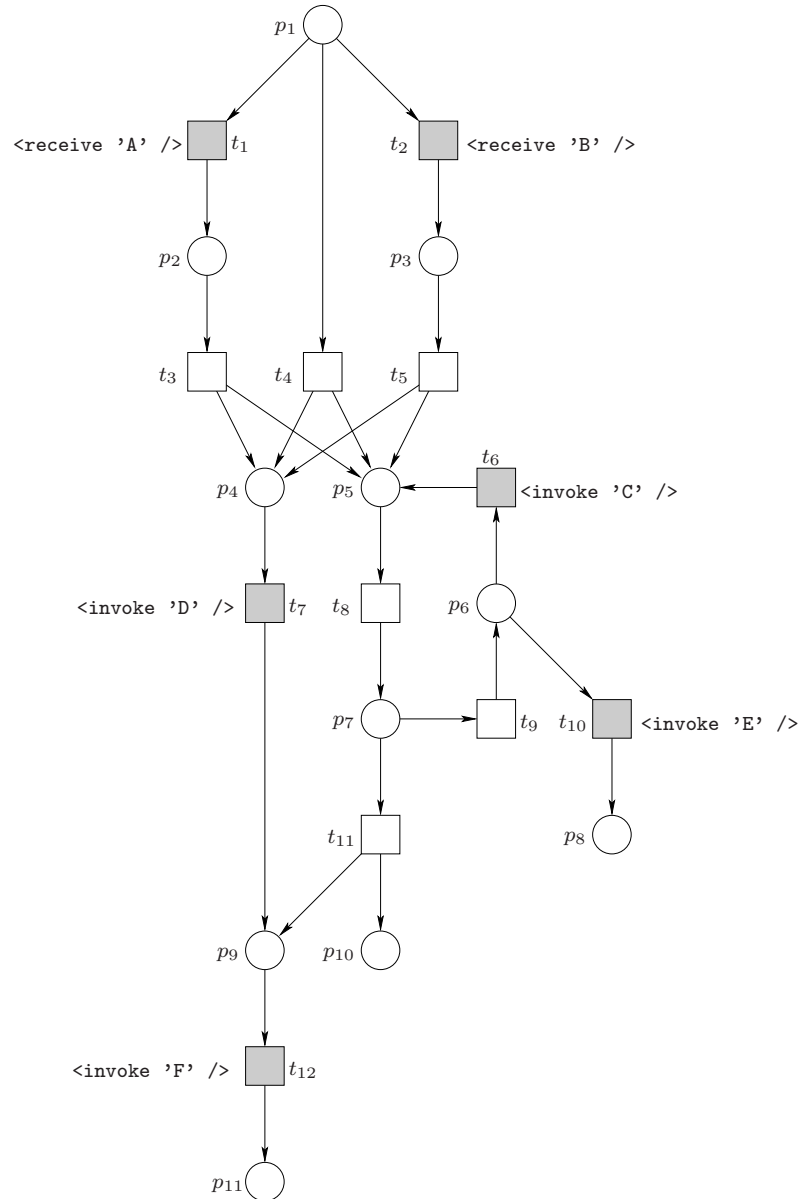


Abbildung 4.3.: Das oWFN nach der Anwendung von Transformation 1.

Die strukturelle Umformung durch Transformation 17 kann nun auf die Transitionen t_3, t_4, t_5 und die Plätze p_4, p_5 angewendet werden. Es entstehen die beiden neuen Knoten p_a und t_a , wie in Abb. 4.4 dargestellt. Transition t_a wird dabei mit einer `empty`-Aktivität annotiert. Wir geben an dieser Stelle die Namen der Knoten als Bezeichner der Aktivitäten zur besseren Übersicht mit an. Nicht mehr mit angegeben sind die WS-BPEL-Aktivitäten aus dem vorherigen Transformationsschritt.

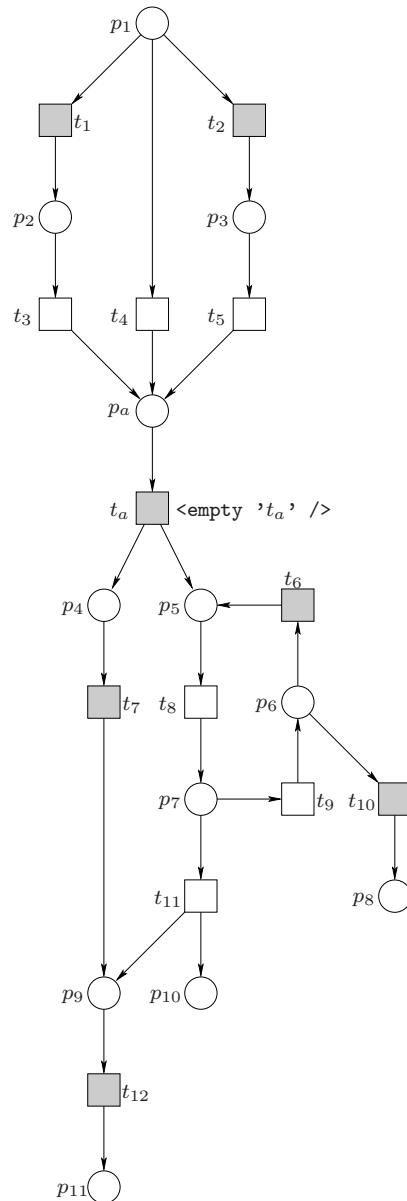


Abbildung 4.4.: Das oWFN nach der Anwendung von Transformation 17.

Nachdem nun keine weitere strukturelle Umformung mehr auf das oWFN angewendet werden kann, werden die in Abschnitt 3.2.2 betrachteten Transformationen für Sequenzen angewendet. Transformation 2 reduziert dabei die Sequenz p_5, t_8, p_7 auf den Platz p_b und die Sequenz p_9, t_{12}, p_{11} auf den Platz p_c . Transformation 3 reduziert die Sequenz t_1, p_2, t_3 auf die Transition t_b und die Sequenz t_2, p_3, t_5 auf die Transition t_c . Zuletzt ist noch Transformation 5 auf die Sequenz t_{10}, p_8 anwendbar, die dadurch auf die Transition t_d reduziert wird. Die neuen Knoten werden gemäß den Transformationsvorschriften mit den vorhandenen WS-BPEL-Aktivitäten und der Aktivität `opaqueActivity` annotiert. Abbildung 4.5 zeigt das oWFN nach diesen Transformationen. Mit grauen Kanten kennzeichnen wir die Zugehörigkeit von Annotationen zu Knoten (zum Beispiel an Platz p_b).

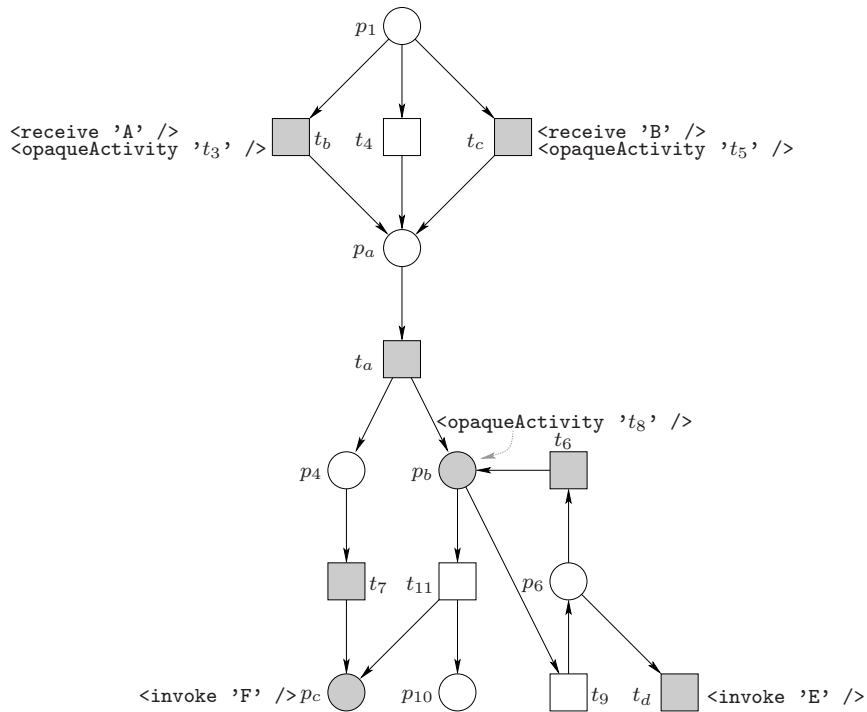


Abbildung 4.5.: Das oWFN nach der Anwendung der Transformationen 2, 3 und 5.

Nun kann Transformation 7 auf die Knoten p_1, p_a, t_b, t_4, t_c angewendet werden und erzeugt eine `if` Annotation. Durch eine erneute Ausführung von Transformation 2 wird das Ergebnis der vorherigen Transformation auf den Platz p_d reduziert. In Abb. 4.6 ist das annotierte oWFN zu sehen. Zum besseren Verständnis haben wir bei der Annotation die `<if>`, `<elseif>` und `<else>`-Elemente der `if`-Aktivität mit angegeben.

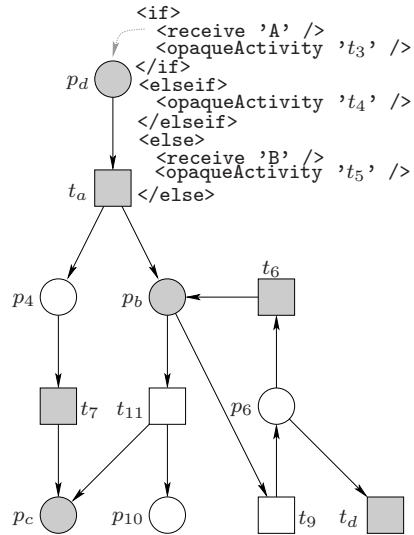


Abbildung 4.6.: Das oWFN nach der Anwendung der Transformationen 7 und 2.

Als nächstes wird durch Transformation 11 der Zyklus bestehend aus den Knoten p_b , t_9 , p_6 , t_6 auf den Platz p_e reduziert. Wie in Abb. 4.7 zu sehen, wird der Platz dabei mit einer `while`-Aktivität annotiert. Die Annotation von Platz t_d wird dadurch zudem um eine `opaqueActivity`-Aktivität ergänzt, die den entfernten Platz p_9 repräsentiert.

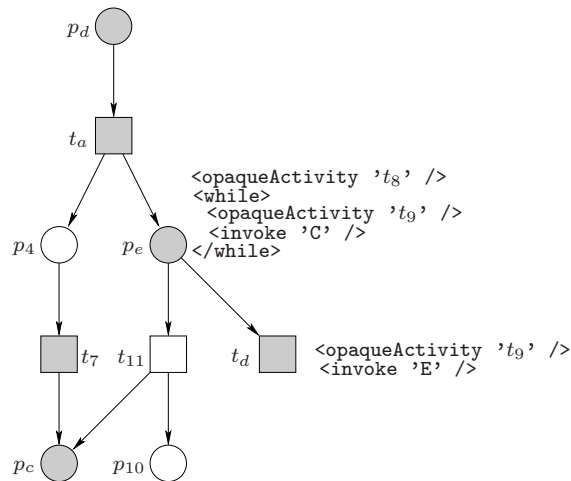


Abbildung 4.7.: Das oWFN nach der Anwendung von Transformation 11.

Die nächste im oWFN ausführbare Transformation ist erst wieder Transformation 13. Das von ihr kopierte Teilnetz besteht nur aus dem Platz p_c , von dem die Kopien p_c' und p_c'' angelegt werden. Beide erhalten die Annotation, die zuvor mit p_c verbunden war. Abbildung 4.8 zeigt die Auswirkungen der beiden Transformationen.

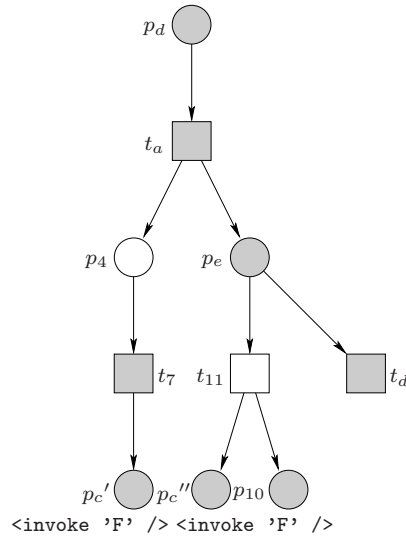


Abbildung 4.8.: Das oWFN nach der Anwendung der Transformationen 3 und 13.

Wieder kann eine im letzten Schritt entstandene Sequenz reduziert werden. Dabei wird Transformation 2 auf die Knoten p_4, t_7, p_c' angewendet und erzeugt den Platz p_f . Die Knoten t_{11}, p_c'', p_{10} können anschließend von Transformation 9 durch eine `flow` Annotation an Transition t_f ersetzt werden. Die so entstehende `flow`-Aktivität bettet als Zweige die Aktivitäten der Plätze beiden Plätze p_c'' und p_{10} ein. Damit kein leerer Zweig in die Aktivität eingefügt wird, erhält in diesem Fall der Platz p_{10} eine `opaqueActivity` Annotation. Abbildung 4.9 zeigt das oWFN nach der Anwendung der beiden Transformationen.

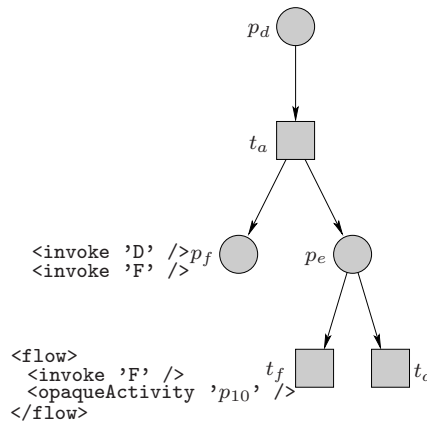


Abbildung 4.9.: Das oWFN nach der Anwendung der Transformationen 2 und 9.

In dem so entstandenen oWFN kann durch Transformation 7 noch eine `if` Annotation, die die Knoten p_e, t_d, t_f zu einem neuen Platz zusammenfasst und durch erneute Anwendung von Transformation 9 eine weitere `flow` Annotation, basierend auf dem neuen Platz und den Knoten t_a, p_f , eingefügt werden. Die verbleibenden beiden Knoten werden durch

Transformation 4 auf einen Platz reduziert. Diese letzten drei Transformationsschritte haben wir nicht mehr abgebildet. Codebeispiel 4.1 zeigt die so entstandene Annotation.

Auf die Annotation kann zunächst Korrektur 4, welche die `process`-Aktivität eingefügt, angewendet werden. Anschließend kann mit Hilfe von Korrektur 5 die `empty`-Aktivität, die aus Transition t_a entstanden ist, entfernt werden. Da sich in der ersten `if`-Aktivität zwei Zweige befinden, die mit jeweils einer `receive`-Aktivität beginnen, kann Korrektur 1 diese durch eine `pick`-Aktivität mit zwei `<onMessage>`-Elementen ersetzen. In einem letzten Schritt werden durch Korrektur 8 fünf `sequence`-Aktivitäten eingefügt. In Codebeispiel 4.2 ist die fertige Annotation zu sehen.

| | |
|---|--|
| <pre> <if> <receive "A" /> <opaqueActivity "t3" /> </if> <elseif> <receive "B" /> <opaqueActivity "t5" /> </elseif> <else> <opaqueActivity "t4" /> </else> <empty "t_a" /> <flow> Zweig 1 mit Annotation: <opaqueActivity "t8" /> <while> <opaqueActivity "t9" /> <opaqueActivity "t6" /> <opaqueActivity "t8" /> </while> <if> <flow> Zweig 1 mit Annotation: <invoke "F" /> Zweig 2 mit Annotation: <opaqueActivity "p10" /> </flow> </if> <else> <opaqueActivity "t9" /> <invoke "E" /> </else> Zweig 2 mit Annotation: <invoke "D" /> <invoke "F" /> </flow> </pre> | <pre> <process> <sequence> <pick> <onMessage "A"> <opaqueActivity "t3" /> </onMessage> <onMessage "B"> <opaqueActivity "t5" /> </onMessage> <onAlarm> <opaqueActivity "t4" /> </onAlarm> </pick> <flow> <sequence> <opaqueActivity "t8" /> <while> <sequence> <opaqueActivity "t9" /> <opaqueActivity "t6" /> <opaqueActivity "t8" /> </sequence> </while> <if> <flow> <invoke "F" /> <opaqueActivity "p10" /> </flow> </if> <else> <sequence> <opaqueActivity "t9" /> <invoke "E" /> </sequence> </else> </sequence> <sequence> <invoke "D" /> <invoke "F" /> </sequence> </flow> </sequence> </process> </pre> |
|---|--|

Codebeispiel 4.1: Die Annotation des letzten Knotens.

Codebeispiel 4.2: Die Annotation nach Anwendung der Korrekturen.

In Anhang A.4 geben wir den vollständigen vom Tool erzeugten WS-BPEL-Code für diese Transformation an. Das von uns implementierte Tool benötigt für diese Transformati-

on 13 Durchläufe durch die Transformationen, die mehrfach ausgeführt werden. Dabei kommt es zu 6 Transformationen des Interface und 15 Transformationen des inneren Netzes vor der Ausführung der Korrekturen. Wie diese Fallstudie anhand eines kleineren Beispiels und durch die Verwendung einiger Transformationen und Korrekturen zeigt, kann durch unsere Transformation in wenigen Schritten aus einem oWFN ein abstrakter WS-BPEL-Prozess erzeugt werden. Dieser ist von der Struktur her dem oWFN sehr ähnlich und auch größere Zyklen mit mehreren Ein- und Ausgängen und Plätze, die nicht sicher sind, wie die Plätze p_9 und p_{11} , können reduziert werden. Durch unsere Korrekturen wird der WS-BPEL-Code zudem weiter optimiert und nur für die Transformation benötigte Hilfsaktivitäten, wie in diesem Fall die `empty`-Aktivität, werden wieder entfernt.

Im folgenden Kapitel werden wir noch einmal unsere Ergebnisse zusammenfassen und einige alternative Ansätze betrachten.

5. Zusammenfassung und Ausblick

5.1. Ergebnisse

Die von uns in Kapitel 3 vorgestellte Transformation ist in der Lage, offene Workflow-Netze vollautomatisch in abstrakte WS-BPEL-Prozesse umzuwandeln. Die oWFN müssen dazu lediglich den wenigen Einschränkungen aus Abschnitt 2.1.3 genügen. Neben der für Geschäftsprozesse wichtigen Soundness-Eigenschaft, spiegeln diese die Einschränkungen von WS-BPEL bezüglich Zyklen wieder. Dennoch gestatten uns die Einschränkung die Transformation komplexer Zyklenstrukturen, sogar solcher, die den wechselseitigen Ausschluss nachbilden. Den so entstandenen abstrakten WS-BPEL-Prozess kann der Nutzer durch wenige zusätzliche Angaben zu einem ausführbaren Prozess vervollständigen. Er kann somit mit Hilfe unserer Transformation die Vorteile nutzen, die die Modellierungs- und Analysemöglichkeiten auf Petrinetzebene bieten.

Zudem zeigt der Vergleich mit einem anderen Transformationswerkzeug in Anhang A.4, dass die Transformation die Möglichkeiten der WS-BPEL-Aktivitäten besser ausnutzt und so kompakteren und leichter verständlichen WS-BPEL-Code erzeugt.

In Kapitel 4 haben wir gesehen, dass unsere Transformation sowohl die bedienenden Partner als auch in vielen Fällen die Struktur des zugrundeliegenden oWFN erhält. Die in dieser Arbeit vorgestellte Transformation und das auf ihr basierende Tool BPEL2oWFN gliedern sich in die vorhandene Toolchain [LMSW06] ein und können somit dazu genutzt werden, WS-BPEL-Prozesse automatisch zu analysieren. Abbildung 5.1 zeigt noch einmal die Toolchain mit der nun durch unsere Transformation geschlossenen Lücke.

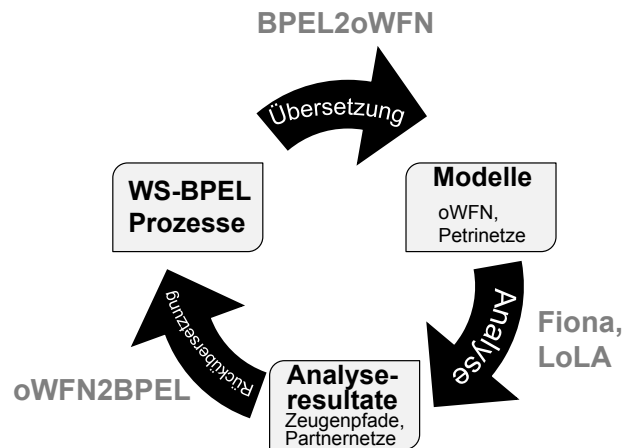


Abbildung 5.1.: Die Toolchain zur Analyse und Optimierung von WS-BPEL-Prozessen.

5.2. Alternative Ansätze

In [MLZ05] kategorisieren die Autoren mögliche Vorgehensweisen bei der Umwandlung von graphbasierten Geschäftsprozessbeschreibungssprachen in blockbasierte. Das Ersetzen von Suchmustern ähnlich unserem Vorgehen wird dort als *Structure-Identification* bezeichnet und das Verfahren, das wir in Transformation 10 in Abschnitt 3.2.4 anwenden, als *Element-Preservation* mit einer bei uns differenzierteren *Element-Minimization*. Eine letzte Strategie mit dem Namen *Structure-Maximization* beschreibt die Kombination der vorher benannten Strategien. Jedoch wird dort lediglich vorgeschlagen, die *Element-Preservation* oder *Element-Minimization* einmalig auf das gesamte Netz anzuwenden, wenn *Structure-Identification* nicht mehr anwendbar ist. Eine Strategie, wie wir sie verwenden, bei der die *Element-Preservation* als wiederkehrendes Element der *Structure-Identification* benutzt wird, wurde nicht benannt. Auch die Separation von Teilnetzen und die einzelne Anwendung der Transformationen auf diese, wie wir sie in Transformation 15 in Abschnitt 3.2.7 verwenden oder die nachträglichen Veränderungen des WS-BPEL-Codes durch unsere Korrekturen wurden bisher nicht als Strategien benannt.

In [AL05] werden als Ausgangsbasis der Transformation zu BPEL4WS Workflow-Netze verwendet. Die Transformation unterscheidet lediglich fünf Suchmuster, denen jeweils der BPEL4WS-Code von genau einer Aktivität zugeordnet wird. Es findet keine Nachbearbeitung des erzeugten Codes statt. Auf Grund der geringen Anzahl an Suchmustern ist die Transformation zudem auf Nutzerrückfragen angewiesen. Dabei werden dem Nutzer Teilnetze des Workflow-Netzes gezeigt und dieser muss den BPEL4WS-Code angeben, der diese Teilnetze ersetzt. Sie werden als Komponenten in einer Bibliothek gespeichert. Wie die Fallstudie der Autoren in [LA06] zeigt, konnten von 100 Workflow-Netzen lediglich 11 ohne Nutzerrückfragen transformiert werden. Erst nach der Aufnahme von 76 zusätzlichen Komponenten in die Bibliothek konnten alle 100 Workflow-Netze zu BPEL4WS-Code transformiert werden. In Anhang A.4 vergleichen wir diesen Ansatz anhand der Transformation eines Beispielnetzes mit der in dieser Arbeit von uns vorgestellten Transformation. Auf Grund der wenigen Suchmuster des Ansatzes verwenden wir für diesen Vergleich ein zyklensfreies Netz, welches auch ohne Nutzerrückfragen transformiert werden kann. Dabei fällt vor allem die häufige Verwendung der `flow`-Aktivität und der Einsatz von Links bei diesem Ansatz auf, da unsere Transformation für dieselben oWFN WS-BPEL-Code ohne Links erzeugt.

5.3. Weitere Arbeit

5.3.1. Vereinfachte Variante

Wir haben in dieser Arbeit versucht, die Anzahl der bei unserer Transformation erzeugten Links möglichst gering zu halten. Ist die Übersichtlichkeit des erzeugten WS-BPEL-Codes nicht von Bedeutung, wenn zum Beispiel nur ein Prozess erzeugt werden soll, der das Verhalten des oWFN bezüglich dem Senden und Empfangen von Nachrichten nachbildet, kann ein alternativer Ansatz gewählt werden. Dazu wird der Erreichbarkeits-

graph des inneren Netzes berechnet. Ist dieser endlich und zyklensfrei, dann kann er in eine `flow`-Aktivität überführt werden, in der die Kanten des Graphen durch Links nachgebildet werden. Neben der schlechten Lesbarkeit des erzeugten WS-BPEL-Codes hat diese Methode zudem den Nachteil, dass sie nur angewendet werden kann, wenn für das innere Netz des `oWFN` ein endlicher und zyklensfreier Erreichbarkeitsgraph existiert. Zur Verfeinerung dieser Methode könnte in weiterer Arbeit versucht werden, Zyklen im Erreichbarkeitsgraphen durch in die `flow`-Aktivität eingebettete `while`-Aktivitäten zu repräsentieren.

5.3.2. Lockerung der Einschränkungen

In Abschnitt 2.1.3 haben wir Einschränkungen für die Klasse der `oWFN`, die wir mit unserer Transformation in abstrakte WS-BPEL-Prozesse umwandeln, formuliert. Diese Einschränkungen könnten durch weitere Arbeiten gelockert werden. Dazu könnten zum Beispiel Kantengewichte zugelassen werden. Denkbar sind auch strukturelle Umformungen, die es ermöglichen eine Anfangsmarkierung zu verwenden, in der mehrere Plätze mit mehreren Marken markiert sind. Dazu muss die Definition der Soundness-Eigenschaft erweitert werden. Für Anfangsmarkierungen mit jeweils nur einer Marke auf jedem Platz der Anfangsmarkierung haben wir, wie in Abschnitt 4.1 angesprochen, bereits entsprechende Umformungen mit in die Implementierung aufgenommen. Wir haben gefordert, dass der Nachbereich aller Plätze, die Teil einer Endmarkierung sind, leer ist. Bei der Analyse von `oWFN` werden jedoch auch Netze betrachtet, bei denen eine Endmarkierung aus einem Platz innerhalb eines Zyklus besteht. Es kann daher auch gewünscht sein, einen WS-BPEL-Prozess zu erzeugen, der dieses Verhalten nachahmt und im Regelfall nicht terminiert. Für den einfachen Fall, dass es sich um einen Platz innerhalb eines Zyklus handelt, ist unsere Transformation in ihrer derzeitigen Form bereits ausreichend. Für Endmarkierungen, die Plätze an anderen Stellen des `oWFN` zulassen, müssten die Transformationen jedoch angepasst werden. Ein besonderes Problem dabei ist, dass im WS-BPEL-Prozess an einer solchen Stelle eine Entscheidung getroffen werden muss, ob der Prozess die nachfolgenden Aktivitäten ausführen soll oder nicht. Während sich das `oWFN` in einer Endmarkierung befindet, würde die Abarbeitung des WS-BPEL-Prozesses einfach fortgesetzt.

5.3.3. Nutzung zusätzlicher Aktivitäten und Variablen

Die Anzahl der WS-BPEL-Aktivitäten, die bei der Transformation verwendet werden, könnte in weiterer Arbeit erhöht werden. Wir haben uns auf einige der in WS-BPEL zur Verfügung stehenden Aktivitäten beschränkt. Zusätzliche Aktivitäten könnten dazu genutzt werden, die Struktur des transformierten `oWFN` genauer zu erhalten. Bei der Modellierung von Geschäftsprozessen ist es nicht untypisch die wiederholte Ausführung eines Zyklus vom Eingang von Nachrichten abhängig zu machen. Dieses Verhalten könnte durch die Verwendung der `repeatUntil`-Aktivität und das Anlegen von Variablen nachgebildet werden. Zum Empfang könnte dann die `pick`-Aktivität benutzt werden. Durch die Angabe einer maximalen Zeitspanne, in der auf den Eingang einer weiteren Nach-

richt gewartet wird, kann diese die weitere Ausführung des Prozesses zulassen, im Gegensatz zur `receive`-Aktivität, die ohne eine eingehende Nachricht den Prozess blockiert. Durch die Analyse des Kontrollflusses von Daten könnten weitere Informationen gewonnen werden, die es zum Beispiel ermöglichen, Nachrichten mit Hilfe der `reply`-Aktivität zu beantworten oder die bei der `invoke`-Aktivität zusätzlich gegebene Möglichkeit des Anfrage-Antwort Nachrichtenaustauschs zu nutzen.

In [AtHKB03] stellen die Autoren das *Milestone Pattern* vor, bei dem die Ausführung einer Aktivität davon abhängig ist, ob ein bestimmter Zustand bereits erreicht wurde und sich das System noch in diesem Zustand befindet. Es wird dabei darauf hingewiesen, dass in zustandsbasierten Systemen, wie Petrinetzen, Aktivitäten deaktiviert werden können, zum Beispiel, indem eine erzeugte Marke von einem Platz wieder konsumiert wird. In nachrichtenbasierten Systemen, wie WS-BPEL, ist dies jedoch nicht vorgesehen. In dieser Arbeit haben wir das Milestone Pattern nicht betrachtet. Unsere in Abschnitt 2.1.3 getroffene Einschränkung bezüglich des Aufbaus von Zyklen verhindert ein Auftreten des Milestone Patterns in den von uns transformierten oWFN. In einer weiteren Arbeit könnten Transformationsstrategien erforscht werden, die ein im oWFN auftretendes Milestone Pattern in WS-BPEL-Aktivitäten überführen. Hierzu könnten automatisch Variablen generiert werden, die Zustände des Systems simulieren.

A. Anhänge

A.1. Erhaltung der Soundness-Eigenschaft durch die Transformationen

Um Transformation 15 ausführen zu können, müssen ausgewählte Teilnetze von Zyklen die Soundness-Eigenschaft erfüllen. In Abschnitt 2.1.3 haben wir dies als Einschränkung für das oWFN gefordert. An dieser Stelle liefern wir den Beweis, dass die Teilnetze die Soundness-Eigenschaft immer noch erfüllen, falls im Verlauf der Transformation alle auf sie anwendbaren Transformationen ausgeführt werden.

Satz A.1 *Teilnetz $A_i = (P, T, F)$ mit $p_a, p_b \in P$, der Anfangsmarkierung $m_0 = [p_a]$ und der Menge der Endmarkierungen $\Omega = \{[p_b]\}$ erfülle die Soundness-Eigenschaft. Nach Anwendung der Transformationen 1, 2, 3, 6, 8, 10, 11, 12, 16 und 17 erfüllt A_i noch immer die Soundness-Eigenschaft.*

Beweis. Wir betrachten die Auswirkungen der einzelnen Transformationen:

- **Transformation 1:** Die Soundness-Eigenschaft wurde für das innere Netz des oWFN definiert. Transformation 1 verändert das innere Netz nicht.
- **Transformationen 2, 3, 6, 8, 16 und 17:** Die Soundness-Eigenschaft für Teilnetze mit den angegebenen Anfangs- und Endmarkierungen entspricht der Definition der Soundness-Eigenschaft für Workflow-Netze in [Aal98]. Für diese hat der Autor in [Aal96] bewiesen, dass ein Workflow-Netz mit einer zusätzlichen Transition, die vom in der Endmarkierung markierten Platz konsumiert und auf den in der Anfangsmarkierung markierten Platz produziert, genau dann sound ist, wenn es lebendig und beschränkt ist. Nach [Mur89] erhalten die strukturellen Reduktionen, die den aufgeführten Transformationen zugrunde liegen, Lebendigkeit und Beschränktheit des Netzes.
- **Transformation 10:** Wir unterscheiden die in Abb. 3.11 dargestellten sieben Fälle:
 - Fall 3.11(a): Das durch die Transformation entfernte Teilnetz B ist mit einer Marke auf Platz x sicher. Da das Netz vor der Transformation lebendig war, müssen Transitionen aus B mit einer Marke auf x schalten können. In B gibt es keine toten Transitionen und keine Knoten mit leerem Nachbereich, daher muss dies zu genau einer Marke auf Platz y führen. Daher verändert die Ersetzung des Teilnetzes durch eine Transition, die eine Marke von x konsumiert und eine Marke auf y produziert, den Ablauf im Netz nicht.
 - Fall 3.11(b): Durch das Schalten von Transition x werden Marken auf Plätzen innerhalb von B produziert. Da das Netz sound ist, kann Transition y nicht tot sein. Das Schalten von x muss demnach dazu führen, dass Marken auf allen

Plätzen des Vorbereichs von y , die sich innerhalb von B befinden, produziert werden. Daher verändert die Ersetzung des Teilnetzes durch einen Platz im Nachbereich von x und im Vorbereich von y den Ablauf im Netz nicht.

- Fälle 3.11(c) und 3.11(d): Diese Fälle können im betrachteten Teilnetz nicht eintreten.
- Fälle 3.11(e), 3.11(f) und 3.11(g): Diese Fälle ergeben sich aus den vorherigen.
- **Transformation 11:** Angenommen das Netz ist vor der Transformation sound und damit beschränkt und lebendig und nach der Transformation erfüllt es die Soundness-Eigenschaft nicht mehr. Dann muss es im Netz nun eine tote Transition oder einen unbeschränkten Platz geben. Wenn eine der Transitionen e_1, e_2, \dots, e_m tot ist, muss sie dies bereits vor der Transformation gewesen sein, was ein Widerspruch zur Annahme wäre. Wird eine Marke auf p_1 produziert, kann anschließend jeder Ausgangsplatz des Zyklus markiert werden. Nach der Transformation ersetzt p_0 die Ausgangsplätze und wird genau dann markiert, wenn p_1 vor der Transformation markiert wurde. Die anderen Plätze in den Vorbereichen von e_1, e_2, \dots, e_m wurden durch die Transformation nicht verändert. Wenn Platz p_0 unbeschränkt ist, muss auch Platz p_1 vor der Transformation unbeschränkt gewesen sein. Das ist ein Widerspruch zur Annahme. Die Transitionen e_1, e_2, \dots, e_m können maximal so häufig schalten, wie Marken auf p_0 produziert werden. Da p_0 nicht unbeschränkt ist, können die Plätze im Nachbereich der Transitionen nach der Transformation ebenfalls nicht unbeschränkt sein.
- **Transformation 12:** Angenommen das Netz ist vor der Transformation sound und damit beschränkt und lebendig und nach der Transformation erfüllt es die Soundness-Eigenschaft nicht mehr. Dann muss es im Netz nun eine tote Transition oder einen unbeschränkten Platz geben. Die Transitionen im Inneren des abgewickelten Zyklus können nach der Transformation nicht tot sein, ohne dass vor der Transformation die Transition im Vorbereich des Eingangsplatzes p_1 tot war. Wird eine Marke auf p_1 produziert, kann anschließend jede Transition im Inneren des abgewickelten Zyklus aktiviert werden. Da keine der Transitionen im Inneren des abgewickelten Zyklus tot ist, kann jeder Platz im Inneren des abgewickelten Zyklus markiert werden. Die Vorbereiche der Transitionen f_1, f_2, \dots, f_m entsprechen den unveränderten Vorbereichen der Transitionen e_1, e_2, \dots, e_m . Daher können die Transitionen ebenfalls nicht tot sein. In den Vorbereichen der Transitionen e_1, e_2, \dots, e_m wurden die Plätze des Zyklus durch den Platz p_0 ersetzt, der markiert werden kann. Daher können auch die Transitionen e_1, e_2, \dots, e_m nicht tot sein. Da keine Transitionen entfernt wurden, die Marken von den Plätzen des Zyklus konsumieren, kann ein Platz des Zyklus nach der Transformation nur unbeschränkt sein, wenn er dies bereits vor der Transformation war. Dies ist ein Widerspruch zur Annahme. Die Transitionen e_1, e_2, \dots, e_m können maximal so häufig schalten, wie Marken auf den Eingangsplätzen des Zyklus produziert werden. Da die Plätze des Zyklus nicht unbeschränkt ist, können die Plätze im Nachbereich der Transitionen nach der Transformation ebenfalls nicht unbeschränkt sein.

□

A.2. Strukturelle Umformungen

Codebeispiel A.1 zeigt in abgekürzter Schreibweise den entstehenden WS-BPEL-Code für das in Abb. 3.18 dargestellte Beispiel, falls die Transformationen zur strukturellen Umformung des oWFN aus Abschnitt 3.2.8 nicht angewendet werden.

```

<process>
  <flow>
    <if>
      <targets>
        <target linkName="Link_16"/>
        <joinCondition>$Link_16</joinCondition>
      </targets>
      <empty>
        <sources>
          <source linkName="Link_9"/>
        </sources>
      </empty>
      <elseif>
        <empty>
          <sources>
            <source linkName="Link_8"/>
          </sources>
        </empty>
      </elseif>
      <else>
        <empty>
          <sources>
            <source linkName="Link_7"/>
          </sources>
        </empty>
      </else>
    </if>
    <if>
      <targets>
        <target linkName="Link_15"/>
        <joinCondition>$Link_15</joinCondition>
      </targets>
      <empty>
        <sources>
          <source linkName="Link_6"/>
        </sources>
      </empty>
    </if>
  </flow>
</process>

```

Codebeispiel A.1: Der WS-BPEL-Code der Beispieltransformation. Teil 1

```
<elseif>
  <empty>
    <sources>
      <source linkName="Link_5"/>
    </sources>
  </empty>
</elseif>
<else>
  <empty>
    <sources>
      <source linkName="Link_4"/>
    </sources>
  </empty>
</else>
</if>
<if>
  <targets>
    <target linkName="Link_14"/>
    <joinCondition>$Link_14</joinCondition>
  </targets>
  <empty>
    <sources>
      <source linkName="Link_3"/>
    </sources>
  </empty>
  <elseif>
    <empty>
      <sources>
        <source linkName="Link_2"/>
      </sources>
    </empty>
  </elseif>
  <else>
    <empty>
      <sources>
        <source linkName="Link_1"/>
      </sources>
    </empty>
  </else>
</if>
<empty>
  <targets>
    <target linkName="Link_12"/>
    <target linkName="Link_11"/>
    <target linkName="Link_10"/>
    <joinCondition>
      $Link_12 and $Link_11 and $Link_10
    </joinCondition>
  </targets>
</empty>
```

Der WS-BPEL-Code der Beispieltransformation. Teil 2

```

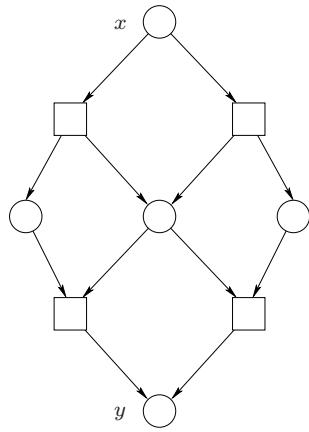
<opaqueActivity>
  <sources>
    <source linkName="Link_16"/>
    <source linkName="Link_15"/>
    <source linkName="Link_14"/>
  </sources>
</opaqueActivity>
<opaqueActivity>
  <sources>
    <source linkName="Link_12"/>
  </sources>
  <targets>
    <target linkName="Link_9"/>
    <target linkName="Link_6"/>
    <target linkName="Link_3"/>
    <joinCondition>
      $Link_9 and $Link_6 and $Link_3
    </joinCondition>
  </targets>
</opaqueActivity>
<opaqueActivity>
  <sources>
    <source linkName="Link_11"/>
  </sources>
  <targets>
    <target linkName="Link_8"/>
    <target linkName="Link_5"/>
    <target linkName="Link_2"/>
    <joinCondition>
      $Link_8 and $Link_5 and $Link_2
    </joinCondition>
  </targets>
</opaqueActivity>
<opaqueActivity>
  <sources>
    <source linkName="Link_10"/>
  </sources>
  <targets>
    <target linkName="Link_7"/>
    <target linkName="Link_4"/>
    <target linkName="Link_1"/>
    <joinCondition>
      $Link_7 and $Link_4 and $Link_1
    </joinCondition>
  </targets>
</opaqueActivity>
</flow>
</process>

```

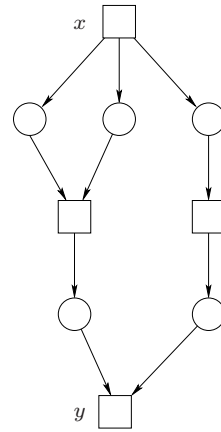
Der WS-BPEL-Code der Beispieltransformation. Teil 3

A.3. Nebenläufige Ausführungen

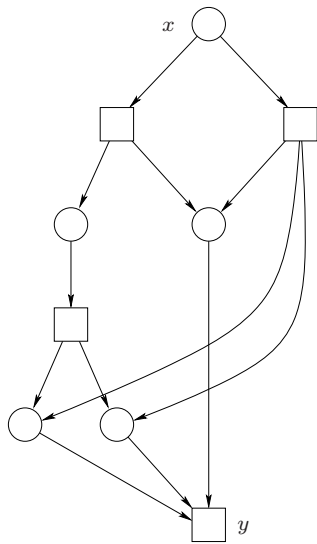
Wir benötigen lediglich vier Teilnetze, um konkrete Beispiele für die in Abbildung 3.11 dargestellten sieben möglichen Ausgangssituationen anzugeben. Abbildung A.1 zeigt die Teilnetze und die Knoten x und y , das restliche oWFN ist nicht abgebildet. Wir verzichten in den Beispielen auf die Darstellung von ein- und ausgehenden Kanten für die Knoten x und y vom und zum restlichen oWFN. Gibt es in den dargestellten Beispielen keine weiteren Kanten von und zum Platz y , handelt es sich zusätzlich um Beispiele für die in den Abbildungen 3.11(c) und 3.11(d) gezeigten Fälle. Wenn es sich im in Abb. A.1(a) zu betrachtenden Beispiel bei x und y um denselben Platz handelt, findet das Beispiel auch Verwendung für den Fall aus Abb. 3.11(g).



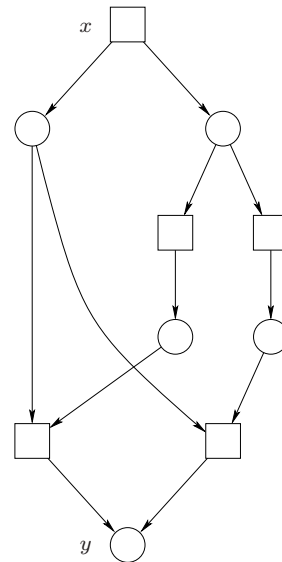
(a) Teilnetz mit $x, y \in P$.



(b) Teilnetz mit $x, y \in T$.



(c) Teilnetz mit $x \in P, y \in T$.



(d) Teilnetz mit $y \in P, x \in T$.

Abbildung A.1.: Beispiele für Teilnetze, die in `flow`-Aktivitäten umgewandelt werden.

A.4. Fallstudie

Codebeispiel A.2 zeigt den vollständigen durch die Fallstudie in Abschnitt 4.2 entstehenden WS-BPEL-Code.

```

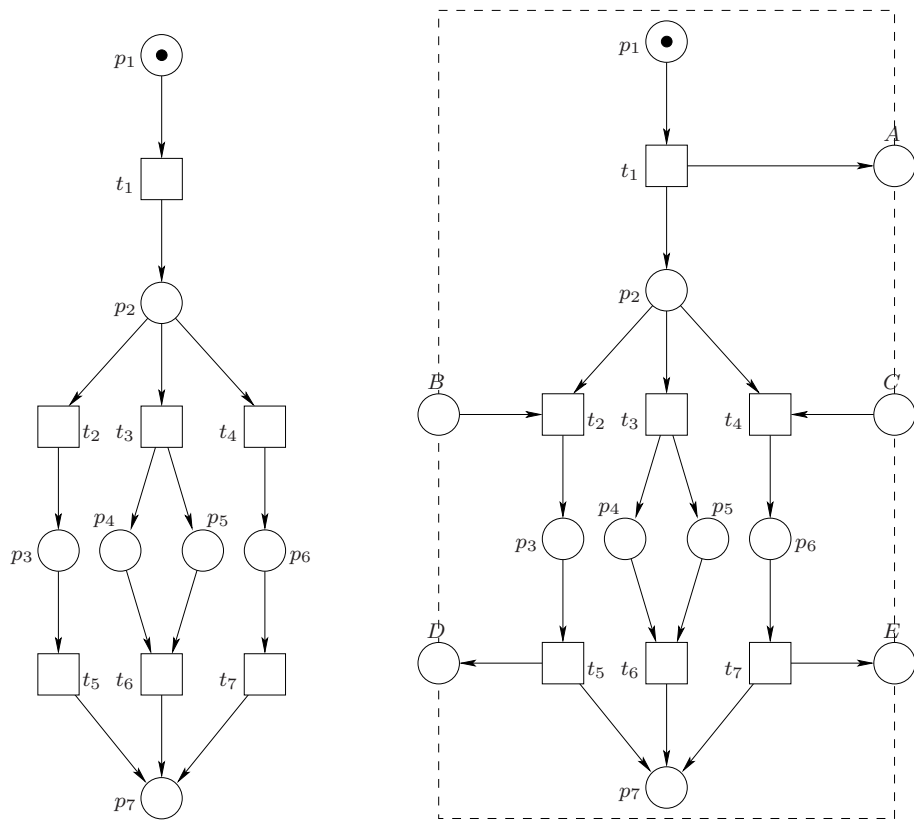
<process name="Fallstudie"
  targetNamespace="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
  xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract" suppressJoinFailure="yes"
  xmlns:template="http://docs.oasisopen.org/wsbpel/2.0/process/abstract/simple-template/2006/08"
  abstractProcessProfile="http://docs.oasisopen.org/wsbpel/2.0/process/abstract/simple-template/2006/08">
  <partnerLinks>
    <partnerLink name="generic_pl" partnerLinkType="##opaque"
      myRole="##opaque" partnerRole="##opaque" />
  </partnerLinks>
  <variables>
    <variable name="Var_A" element="##opaque" /> <variable name="Var_B" element="##opaque" />
    <variable name="Var_C" element="##opaque" /> <variable name="Var_D" element="##opaque" />
    <variable name="Var_E" element="##opaque" /> <variable name="Var_F" element="##opaque" />
  </variables>
  <sequence name="Act_sequence">
    <pick name="Act_if_pl_to_place_for_cross_over">
      <onMessage partnerLink="generic_pl" operation="B" variable="Var_B">
        <opaqueActivity name="Act_t5" />
      </onMessage>
      <onMessage partnerLink="generic_pl" operation="A" variable="Var_A">
        <opaqueActivity name="Act_t3" />
      </onMessage>
      <onAlarm> <for opaque="yes" />
        <opaqueActivity name="Act_t4" />
      </onAlarm>
    </pick>
    <flow name="Act_transition_for_cross_over">
      <sequence name="Act_sequence">
        <opaqueActivity name="Act_t8" />
        <while name="Act_p5_t8_p7"> <condition opaque="yes" />
          <sequence name="Act_sequence">
            <opaqueActivity name="Act_t9" />
            <opaqueActivity name="Act_t6" />
            <opaqueActivity name="Act_t8" />
          </sequence>
        </while>
        <if name="Act_if_p5_t8_p7_open_end"> <condition opaque="yes" />
          <flow name="Act_t11">
            <invoke name="Act_F" partnerLink="generic_pl" operation="F" inputVariable="Var_F" />
            <opaqueActivity name="Act_p10" />
          </flow>
          <else>
            <sequence name="Act_sequence">
              <opaqueActivity name="Act_t9" />
              <invoke name="Act_E" partnerLink="generic_pl" operation="E" inputVariable="Var_E" />
            </sequence>
          </else>
        </if>
      </sequence>
      <sequence name="Act_sequence">
        <invoke name="Act_D" partnerLink="generic_pl" operation="D" inputVariable="Var_D" />
        <invoke name="Act_F" partnerLink="generic_pl" operation="F" inputVariable="Var_F" />
      </sequence>
    </flow>
  </sequence>
</process>

```

Codebeispiel A.2: Der in der Fallstudie generierte WS-BPEL-Code.

A.5. Vergleich mit einem alternativen Ansatz

Wir wollen an dieser Stelle den in [AL05] gewählten Ansatz zur Transformation von Workflow-Netzen zu BPEL4WS Prozessen mit dem in dieser Arbeit vorgestellten Ansatz vergleichen. Wir verwenden dazu das in Abb. A.2(a) dargestellte Petrinetz. Es handelt sich um ein einfaches Beispiel, das sowohl Nebenläufigkeit, als auch bedingtes Verhalten enthält. Es ist so gestaltet, dass sowohl die in [LA06] vorgestellte und in das Process-Mining-Werkzeug ProM [vDdMV⁺05] übernommene Implementierung¹ als auch die von uns in Abschnitt 4.1 angesprochene Implementierung oWFN2BPEL das Netz ohne Nutzereingriffe verarbeiten können.



(a) Ein Petrinetz zum Vergleich der Transformationen.

(b) Das um Interfaceplätze erweiterte Petrinetz.

Abbildung A.2.: Beispiele für Teilnetze, die in `flow`-Aktivitäten umgewandelt werden.

Wir haben zur Gegenüberstellung aus der Ausgabe unseres Werkzeugs die `process`-Aktivität und die Deklarationen von Variablen und des `<partnerLinks>`-Elements entfernt. Die Implementierung von [AL05] in ProM generierte diese Angaben nicht mit. Wir beschränken uns somit bei unserem Vergleich auf den Kern des Prozesses. Codebei-

¹Wir verwenden ein aktuelles Nightly Build von ProM vom 09.05.2007.

spiel A.3 zeigt die Ausgabe durch ProM und Codebeispiel A.4 die gekürzte Ausgabe der Implementierung der in dieser Arbeit vorgestellten Transformation.

Wie in den Codebeispielen zu sehen ist, wurde bei der Transformation nach [AL05] eine `flow`-Aktivität erzeugt. In diese sind mit Hilfe von fünf Links sieben `invoke`- und zwei `sequence`-Aktivitäten eingebettet. Zudem findet sich in der `flow`-Aktivität eine `empty`-Aktivität, die auf Platz p_2 zurückzuführen ist und nebenläufig zu den anderen eingebetteten Aktivitäten ausgeführt wird. Das bedingte Verhalten an Platz p_2 des Petrinetzes spiegelt sich im erzeugten BPEL4WS-Prozess nicht wieder. Es müssten durch spätere Nutzereingriffe `<transitionCondition>`-Elemente innerhalb der `<source>`-Elemente eingefügt werden, um dies zu korrigieren. Die Nebenläufigkeit der Plätze p_4 und p_5 des Petrinetzes ist im BPEL4WS-Prozess nicht vorhanden, stattdessen finden sich an der entsprechenden Stelle zwei Links zwischen denselben Aktivitäten.

Durch unsere Transformation wurde ein WS-BPEL-Prozess mit einer `if`- und einer `flow`-Aktivität, die keine Links verwendet, drei `sequence`- und sieben `opaqueActivity`-Aktivitäten erzeugt. Der WS-BPEL-Code ist übersichtlicher und erhält die Struktur des Petrinetzes besser. Während das bedingte Verhalten im anderen Transformationsergebnis nicht erkennbar war, und es erforderte, dass der Nutzer es innerhalb der erzeugten `flow`-Aktivität von Hand erneut einfügt, ist es bei unserer Transformation ein zentrales Element, vergleichbar mit seiner Position im Petrinetz. Weiter ist die Nebenläufigkeit der Plätze p_4 und p_5 durch zusätzliche Aktivitäten nachgebildet.

```
<flow name="Flow_F3">
  <invoke name="t1">
    <source linkName="link1" />
    <source linkName="link2" />
    <source linkName="link3" />
  </invoke>
  <invoke name="t3">
    <target linkName="link1" />
    <source linkName="link4" />
    <source linkName="link5" />
  </invoke>
  <invoke name="t6">
    <target linkName="link4" />
    <target linkName="link5" />
  </invoke>
  <sequence name="Sequence_F1">
    <invoke name="t2" />
    <invoke name="t5" />
    <target linkName="link2" />
  </sequence>
  <sequence name="Sequence_F2">
    <invoke name="t4" />
    <invoke name="t7" />
    <target linkName="link3" />
  </sequence>
  <empty name="p2" />
</flow>
```

Codebeispiel A.3: Ausgabe der Implementierung nach [AL05] in ProM.

```
<sequence name="Act_sequence">
  <opaqueActivity name="Act_t1" />
  <if name="Act_if_p1_t1_p2_to_p7">
    <condition opaque="yes" />
    <flow name="Act_t3">
      <opaqueActivity name="Act_p4" />
      <opaqueActivity name="Act_p5" />
    </flow>
    <elseif>
      <condition opaque="yes" />
      <sequence name="Act_sequence">
        <opaqueActivity name="Act_t4" />
        <opaqueActivity name="Act_t7" />
      </sequence>
    </elseif>
    <else>
      <sequence name="Act_sequence">
        <opaqueActivity name="Act_t2" />
        <opaqueActivity name="Act_t5" />
      </sequence>
    </else>
  </if>
</sequence>
```

Codebeispiel A.4: Ausgabe der Implementierung unserer Transformation.

Weitere Vergleiche mit größeren Beispielnetzen zeigten ähnliche Ergebnisse. Während bei unserer Transformation kaum `flow`-Aktivitäten erzeugt werden und diese in den meisten Fällen keine Links verwenden, werden bei der Transformation durch die Implementierung von [AL05] in ProM viele Petrinetze vollständig durch `flow`-Aktivitäten repräsentiert. Das in der Fallstudie in Abschnitt 4.2 betrachtete `oWFN` erzeugt beispielsweise nach dem Entfernen des Zyklus eine `flow`-Aktivität mit zwölf Links. Wie in der Fallstudie zu sehen war, erzeugt unsere Transformation keine Links.

Für beide Transformationen verbessern sich die Ergebnisse, wenn der Empfang von Nachrichten mit in das Petrinetz aufgenommen wird. Mit Hilfe einer älteren Implementierung von [AL05] namens `WorkflowNet2BPEL4WS` können annotierte Workflow-Netze betrachtet werden. In diesen Annotationen können wir den Empfang von Nachrichten angeben. Für unsere Implementierung verwenden wir das `oWFN` in Abb. A.2(b) mit zusätzlichen Interfaceplätzen. Die Codebeispiele A.5 und A.6 zeigen erneut die Kernbereiche der erzeugten Prozesse. In beiden Fällen wurde jeweils eine `pick`-Aktivität erzeugt, die zwischen Nachrichtenempfang und zeitbasiertem Ereignis unterscheidet. Der Hauptunterschied liegt nun in der Behandlung der Nebenläufigkeit der Plätze p_4 und p_5 .

```

<sequence name="Sequence_F5">
  <invoke name="t1" />
  <pick name="Pick_F4">
    <onMessage operation="msg1">
      <sequence name="Sequence_F1">
        <invoke name="t2" />
        <invoke name="t5" />
      </sequence>
    </onMessage>
    <onMessage operation="msg2">
      <sequence name="Sequence_F2">
        <invoke name="t4" />
        <invoke name="t7" />
      </sequence>
    </onMessage>
    <onAlarm for="timeout" until="deadline">
      <flow name="Flow_F3">
        <links>
          <link name="t3_t6" />
        </links>
        <invoke name="t3">
          <source linkName="t3_t6" />
        </invoke>
        <invoke name="t6" joinCondition="...">
          <target linkName="t3_t6" />
        </invoke>
      </flow>
    </onAlarm>
  </pick>
</sequence>

```

Codebeispiel A.5: Ausgabe von `WorkflowNet2BPEL4WS`.

```

<sequence name="Act_sequence">
  <invoke name="Act_A"
    partnerLink="generic_pl"
    operation="A" inputVariable="Var_A" />
  <pick name="Act_if_p1_t1_p2_to_p7">
    <onMessage partnerLink="generic_pl"
      operation="C"
      variable="Var_C">
      <invoke name="Act_E"
        partnerLink="generic_pl"
        operation="E" inputVariable="Var_E" />
    </onMessage>
    <onMessage partnerLink="generic_pl"
      operation="B"
      variable="Var_B">
      <invoke name="Act_D"
        partnerLink="generic_pl"
        operation="D" inputVariable="Var_D" />
    </onMessage>
  </pick>
  <onAlarm>
    <for opaque="yes" />
    <flow name="Act_t3">
      <opaqueActivity name="Act_p4" />
      <opaqueActivity name="Act_p5" />
    </flow>
  </onAlarm>
</sequence>

```

Codebeispiel A.6: Ausgabe von `oWFN2BPEL`.

Abbildungsverzeichnis

| | |
|--|----|
| 1.1. Die Toolchain zur Analyse und Optimierung von WS-BPEL-Prozessen. | 7 |
| 2.1. Ein Beispiel für einen Zyklus. | 11 |
| 2.2. Beispiel für ein offenes Workflow-Netz. | 12 |
| 2.3. Beispiele für die Einschränkung des Aufbaus von Zyklen. | 15 |
| 3.1. Vereinfachtes Beispiel für die Transformation eines oWFN. | 22 |
| 3.2. Beispiele für die Transformation des Interface durch Transformation 1. | 25 |
| 3.3. Beispiele für die Transformation von Sequenzen. | 29 |
| 3.4. Beispiele für verschiedene mögliche Transformationsreihenfolgen. | 31 |
| 3.5. Beispiel für einen Konflikt. | 32 |
| 3.6. Beispiele für die Transformation bedingter Verzweigungen. | 34 |
| 3.7. Beispiel für nebenläufige Ausführungen. | 38 |
| 3.8. Beispiele für die Transformation parallelen Verhaltens. | 39 |
| 3.9. Beispiel für ein für Transformation 10 geeignetes Teilnetz. | 41 |
| 3.10. Konflikt, der die Ausführung von WS-BPEL-Aktivitäten verhindern kann. | 46 |
| 3.11. Beispiele für die Transformation von Teilnetzen durch Transformation 10. | 49 |
| 3.12. Beispiel für einen einfachen Zyklus und dessen Transformation. | 51 |
| 3.13. Beispiel für die Transformation von Zyklen durch Transformation 11. | 53 |
| 3.14. Beispiel für Transformation 12 von Zyklen mit mehreren Eingängen. | 57 |
| 3.15. Keine gültige Transformation für zwei Zyklen. | 58 |
| 3.16. Beispiele für die Transformation bei mehrfachen Ausführungen. | 61 |
| 3.17. Beispiel für Transformation 15 bei mehrfachen Ausführungen auf Zyklen. | 63 |
| 3.18. Beispiel für die Nützlichkeit von strukturellen Umformungen. | 65 |
| 3.19. Beispiele für die strukturellen Umformungen. | 67 |
| 4.1. Beispiele für zusätzliche Transformationen in oWFN2BPEL. | 77 |
| 4.2. Ein oWFN als Ausgangsbasis für unsere Transformation. | 79 |
| 4.3. Das oWFN nach der Anwendung von Transformation 1. | 80 |
| 4.4. Das oWFN nach der Anwendung von Transformation 17. | 81 |
| 4.5. Das oWFN nach der Anwendung der Transformationen 2, 3 und 5. | 82 |
| 4.6. Das oWFN nach der Anwendung der Transformationen 7 und 2. | 83 |
| 4.7. Das oWFN nach der Anwendung von Transformation 11. | 83 |
| 4.8. Das oWFN nach der Anwendung der Transformationen 3 und 13. | 84 |
| 4.9. Das oWFN nach der Anwendung der Transformationen 2 und 9. | 84 |
| 5.1. Die Toolchain zur Analyse und Optimierung von WS-BPEL-Prozessen. | 87 |
| A.1. Beispiele für Teilnetze, die in flow-Aktivitäten umgewandelt werden. | 96 |
| A.2. Beispiele für Teilnetze, die in flow-Aktivitäten umgewandelt werden. | 98 |

Literaturverzeichnis

- [AAA⁺07] Alexandre Alves, Assaf Arkin, Sid Askary, Ben Bloch, Francisco Curbera, Yaron Goland, Neelakantan Kartha, Canyang Kevin Liu, Dieter König, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Web services business process execution language version 2.0. committee specification. Technical report, OASIS, January 2007.
- [Aal96] Wil M. P. van der Aalst. Structural characterizations of sound workflow nets. Computing science reports 96/23, Eindhoven University of Technology, 1996.
- [Aal98] Wil 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.
- [ACD⁺03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services, version 1.1. Technical report, IBM, BEA Systems, Microsoft, SAP AG, Siebel Systems, May 2003.
- [AL05] Wil M. P. van der Aalst and Kristian Bisgaard Lassen. Translating workflow nets to BPEL4WS. Technical Report WP 145, Eindhoven University of Technology, August 2005.
- [AtHKB03] Wil M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.
- [Jen92] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Springer-Verlag, 1992.
- [LA06] Kristian Bisgaard Lassen and Wil M. P. van der Aalst. Workflow-Net2BPEL4WS: A tool for translating unstructured workflow processes to readable bpel. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4275 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 2006.
- [LMSW06] Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg. Analyzing interacting BPEL processes. In *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings*, volume 4102 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, September 2006.
- [Loh07] Niels Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0 and its compiler BPEL2oWFN. Informatik-Berichte 212, Humboldt-

- Universität zu Berlin, June 2007.
- [MLZ05] Jan Mendling, Kristian Bisgaard Lassen, and Uwe Zdun. Transformation strategies between block-oriented and graph-oriented process modelling languages. Technical Report JM-2005-10-10, Vienna University of Economics and Business Administration, October 2005.
- [MRS05] Peter Massuthe, Wolfgang Reisig, and Karsten Schmidt. An operating guideline approach to the SOA. In *2nd South-East European Workshop on Formal Methods 2005 (SEEFM05)*, Ohrid, Republic of Macedonia, 2005.
- [Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
- [Rei82] Wolfgang Reisig. *Petrinetze, Eine Einführung*. Springer-Verlag, 1982.
- [RSS05] Wolfgang Reisig, Karsten Schmidt, and Christian Stahl. Kommunizierende Workflow-Services modellieren und analysieren. *Informatik - Forschung und Entwicklung*, pages 90–101, October 2005.
- [Sch00] Karsten Schmidt. LoLA: A low level analyser. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets, 21st International Conference (ICATPN 2000)*, number 1825 in Lecture Notes in Computer Science, pages 465–474. Springer-Verlag, June 2000.
- [Sch05] Karsten Schmidt. Controllability of Open Workflow Nets. In Jörg Desel and Ulrich Frank, editors, *Enterprise Modelling and Information Systems Architectures*, volume P-75 of *Lecture Notes in Informatics (LNI)*, pages 236–249, Bonn, 2005. Entwicklungsmethoden für Informationssysteme und deren Anwendung (EMISA, RWTH Aachen), Köllen Druck+Verlag GmbH.
- [Sta90] Peter H. Starke. *Analyse von Petri-Netz-Modellen*. Teubner, 1990.
- [vDdMV⁺05] Boudewijn F. van Dongen, Ana Karla A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and Wil M. P. van der Aalst. The ProM framework: A new era in process mining tool support. In *ICATPN*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer-Verlag, 2005.

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit „*Transformation von offenen Workflow-Netzen zu abstrakten WS-BPEL-Prozessen*“ selbstständig und ohne fremde Hilfe verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet zu haben.

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

Berlin, den 4. Juli 2007

