Verteilte Geschäftsprozesse modellieren und analysieren

Wolfgang Reisig Karsten Schmidt
Christian Stahl
Humboldt-Universität zu Berlin, Institut für Informatik,
{reisig, kschmidt, stahl}@informatik.hu-berlin.de

Verteilte Geschäftsprozesse nutzen das Internet, um auf heterogenen Rechnerstrukturen Dienste auszubieten. Modellierungstechniken und Implementierungssprachen für solche Dienste werfen im Vergleich mit herkömmlichen Rechnern grundlegend neue Fragestellungen auf. Wir diskutieren einige davon und zeigen, wie Petrinetze ihre Beantwortung ermöglichen.

1 Einleitung

1.1 Web Services

Konventionelle, monolithische IT-System erweisen sich zunehmend als zu unflexibel, wenn es darum geht, sie neuen Anforderungen anzupassen; insbesondere, wenn einzelne Komponenten aktualisiert, ersetzt, oder in einem neuen Kontext verwendet werden sollen. Um diesen Mangel abzuhelfen, wird seit einiger Zeit das Paradigma der *Service-orientierten Architektur* (SOA) propagiert. Eine solche Software-Architektur ist aus einzelnen *Services* komponiert. Ein Service ist dabei ein selbständiges, ausführbares Stück Software. Services *kommunizieren* miteinander, in zunehmendem Maße mit Hilfe von Nachrichten über das World Wide Web. Deshalb sind *Web Services* (WS) von zentraler Bedeutung.

1.2 Verteilte Geschäftsprozesse

Ein *Geschäftsprozess* (engl. business process) ist ein standardisiertes Verfahren, um organisatorische Vorgänge zu realisieren, wie sie typischerweise in Verwaltungen auftreten. Ein Geschäftsprozess besteht aus einer Menge von *Aktivitäten*, deren Ausführungen kausal miteinander zusammenhängen. Sie beziehen und verarbeiten Informationen aus ihrer *Umgebung*, sie generieren neue Informationen und geben sie an ihre Umgebung ab.

Ein *verteilter Geschäftsprozess* ist eine Menge von Geschäftsprozessen, die räumlich oder organisatorisch getrennt verlaufen und lediglich über Kommunikationskanäle miteinander verbunden sind. Zwei Prozesse eines verteilten Geschäftsprozesses gehören wechselseitig zu ihrer jeweiligen Umgebung, sofern sie einen Kommunikationskanal haben.

1.3 Implementierung verteilter Geschäftsprozesse auf dem Web

Zur Implementierung verteilter Geschäftsprozesse drängt sich das Paradigma der Web Services geradezu auf. Tatsächlich sind verteilte Geschäftsprozesse der derzeit zentrale Anwendungsbereich für Web Services im Bereich der Anwendersoftware. Die Implementierung eines verteilten Geschäftsprozesses auf dem Web wird erheblich leichter, wenn man über eine Sprache verfügt, mit der man Geschäftsprozesse formulieren kann und die auf dem Web verteilt ausführbar ist. Eine solche Sprache ist die *Business Process Execution Language for Web Services (BPEL4WS)*[1] [3], die IBM und Microsoft als Weiterentwicklung von WSFL bzw. XLANG im Juli 2002 vorgeschlagen haben und die inzwischen von einem Konsortium aus über 130 Firmen unterstützt wird.

BPEL basiert auf bekannten Middleware-Technologien, insbesondere dem *Web Service Technology Stack*, bestehend aus Core *Layers* für den Transport und die Formatierung von Nachrichten und *Emerging Layers*, die *Quality of Service* garantieren und letztlich die verteilten Geschäftsprozesse realisieren.

[1] Wir verwenden im weiteren Text die inzwischen geläufige Kurzbezeichnung BPEL.

Ein BPEL-Programm beschreibt den Aufbau eines Geschäftsprozesses innerhalb eines Web Services und spezifiziert zugleich die Interaktion eines einzelnen Geschäftsprozesses mit den Partnern seiner Umgebung.

Der Web Service Technology Stack mit BPEL an der Spitze ist bei weitem nicht der einzige Ansatz, eine Architektur für Web Services zu definieren. Ein konkurrierender Vorschlag ist beispielsweise *Electronic Business XML* (ebXML) von OASIS. BPEL ist jedoch mit Abstand am weitesten verbreitet und anerkannt. Die in dieser Arbeit aufgeworfenen Fragestellungen und Lösungsvorschläge formulieren wir Beispiel von BPEL. Sie haben dennoch einen ganz allgemeinen Charakter und sind durchaus auf andere Sprachen für Web Services übertragbar.

1.4 Zentrale Fragestellungen

Beim Entwurf verteilter Geschäftsprozesse, die als Web Services ausgeführt werden sollen, entstehen spezifische Fragestellungen, die sich in zwei Klassen gliedern: Zum einen geht es um die genauere Bedeutung, also das *Verhalten* verteilter Geschäftsprozesse. Das wird insbesondere kritisch, wenn bereits ausgeführte Aktivitäten (beispielsweise die Buchung eines Fluges) *zurückgesetzt* werden müssen (weil sich beispielsweise später herausstellt, dass kein Hotelzimmer verfügbar ist). Die zweite Klasse betrifft *Eigenschaften* verteilter Geschäftsprozesse, beispielsweise die Möglichkeit, einen Geschäftsprozess sinnvoll zu nutzen. Wir gehen auf beide Aspekte ein:

- zum Verhalten verteilter Geschäftsprozesse: Man unterscheidet hier *positiven* Kontrollfluss, also die Formulierung des beabsichtigten zielführenden Verhaltens, und den *negativen* Kontrollfluss, der das Verhalten im Fehlerfall, insbesondere das Zurücksetzen (Kompensieren) von Aktivitäten betrifft. BPEL trennt beide Aspekte sehr konsequent. (Damit ist – in Teilen – das ACID-Prinzip aus Datenbanken auf Transaktionen verteilter Geschäftsprozesse übertragbar.)

- zu Eigenschaften verteilter Geschäftsprozesse: Wichtig ist der Aspekt der *Bedienbarkeit*: Ein „vernünftiger“ Geschäftsprozess hat einen wohldefinierten Anfangszustand, in dem jede Ausführung beginnt. Entsprechend hat er einen wohldefinierten Endzustand, in dem jede begonnene Ausführung endet. In diesem Endzustand liegen keine Nachrichten mehr auf den Eingangskanälen des Prozesses. Eine „böswillige“ Umgebung kann vernünftiges Verhalten verhindern, indem sie beispielsweise Nachrichten nicht sendet, die der Prozess erwartet. Andererseits muss die Umgebung immer die Möglichkeit haben, eine einmal begonnene Ausführung eines Geschäftsprozesses bis zum Endzustand vorzubringen, also den Web Service zu *bedienen*.

  Eine Umgebung ist selbst wieder ein Geschäftsprozess. Es macht einen Unterschied, ob die bedienende Umgebung ein monolithischer Prozeß ist oder aus unabhängigen Teilprozessen besteht. Im zweiten Fall ist bei der Konstruktion eines Teilprozesses zu unterscheiden, ob er im Zusammenspiel mit anderen, gegebenen Umgebungs-Prozessen „vernünftig“ funktionieren muß, oder ob die Umgebungsprozesse unbekannt sind.

1.5 Die Notwendigkeit der Modellbildung

Im letzten Abschnitt haben wir eine Reihe von Fragen aufgeworfen, die nicht einfach zu beantworten sind. Die Schwierigkeiten beginnen schon damit, dass die Semantik von Sprachen wie BPEL letztendlich bisher nur umgangssprachlich formuliert vorliegt. So sind in BPEL Feinheiten der Fehlerbehandlung, insbesondere das Zurücksetzen von Aktivitäten nicht eindeutig geklärt. Ob BPEL-Programme bedienbar, kombinierbar oder ersetzbar sind, kann aus ihrer gegebenen syntaktischen Darstellung nicht abgeleitet werden.

Man benötigt ein *semantisches Modell* für Geschäftsprozess-Beschreibungssprachen wie z. B. BPEL. Ein solches Modell stellt die Bedeutung der Programme auf der gewählten Abstraktionsebene eindeutig dar und stellt zugleich Techniken bereit, die die aufgeworfenen Fragen nach Bedienbarkeit, Kombinierbarkeit und Ersetzbarkeit von Programmen beantworten helfen.

Wir schlagen in dieser Arbeit ein solches Modell vor. Es verwendet *High-Level Petrinetze*; die Gründe für diese Wahl werden kurz erläutert:

- Geschäftsprozesse verwenden als elementare Objekte und Operationen abstrakte Strukturen, beispielsweise „Antwort auf eine Nachricht“ oder „einen Auftrag stornieren“ ohne die konkrete Repräsentation in konventionellen Datenstrukturen. High-Level Petrinetze unterstützen dieses Konzept.

- Das Paradigma der Geschäftsprozesse berücksichtigt keine Laufzeiten von Nachrichten zwischen einzelnen Prozessen. Insbesondere können sich Nachrichten „überholen“. Die Semantik von Petrinetzen entspricht dem in natürlicher Weise.

- Die Komposition einzelner Geschäftsprozesse entspricht unmittelbar der Komposition von Petrinetzen, indem man Plätze der Petrinetz-Modelle der Prozesse verschmilzt. Aktivitäten verschiedener Prozesse wirken lokal und unabhängig voneinander, genau wie die Transitionen mit disjunkten Vor- und Nachbereichen in einem Petrinetz.

1.6 Aufbau der Arbeit

Aus den geschilderten Sachverhalten und Fragestellungen folgt der Aufbau dieser Arbeit: Im zweiten Kapitel wird zunächst die Semantik für Geschäftsprozess-Beschreibungssprachen am Beispiel von BPEL dargestellt. Der positive Kontrollfluss ist dabei vergleichsweise einfach zu handhaben; seine Semantik ist auch schon an anderer Stelle beschrieben worden [6], [5]. Wir konzentrieren uns auf den negativen Kontrollfluss und zeigen insbesondere am Beispiel des *stop-Musters*, wie negativer Kontrollfluss sich über einen verteilten Geschäftsprozess fortpflanzt. Schließlich wird noch einmal validiert, warum die gewählte Semantik angemessen ist.

Im dritten Kapitel schlagen wir eine Vereinfachung der Petrinetzmodelle von Geschäftsprozessen vor mit dem Ziel, die computergestützte Analyse der Modelle zu vereinfachen. Im vierten Kapitel wird am Beispiel der Bedienbarkeit gezeigt, dass das se-

mantische Modell der Petrinetze zur Analyse der in 1.4 diskutierten Eigenschaften von Geschäftsprozessen gut geeignet ist.

2 Verhaltensdefinition

2.1 Überblick zur Petrinetz-Semantik für BPEL

Wir bilden eine *kompositionale* und *hierarchische* Semantik für BPEL. Sie orientiert sich an den Sprachkonstrukten von BPEL, den *Aktivitäten*. Eine Aktivität ist entweder elementar (beispielsweise empfängt oder verschickt sie eine Nachricht) oder aus mehreren Aktivitäten zusammengesetzt. Besonders wichtige zusammengesetzte Aktivitäten sind die *scopes*. Ein scope bindet alle seine Sub-Aktivitäten an eine Fehler- und Kompensationsbehandlung an. Der äußerste scope ist der *process*. Jeder Aktivität wird ein Petrinetz, sein *Muster*, zugeordnet. In Analogie zur Komposition von BPEL-Aktivitäten können auch Muster komponiert werden. Die Semantik eines so komponierten Petrinetzes muss natürlich die Semantik der komponierten Aktivität abbilden. Um diese Eigenschaft zu garantieren, benötigt jedes Muster eine definierte *Schnittstelle*. Desweiteren ermöglicht BPEL die hierarchische Anordnung seiner Aktivitäten, d.h. einige Aktivitäten enthalten eine beliebige Anzahl von Sub-Aktivitäten. Das entsprechende Muster muss die Aktivität für jede vorgegebene Anzahl von Sub-Aktivitäten abbilden können. Ein Parameter legt dabei die Anzahl der Sub-Aktivitäten fest. Wir verwenden im Folgenden anstelle von Hierarchisierung den Begriff *Verfeinerung*. Beim Komponieren und Verfeinern der Muster dürfen nur Verklemmungen auftreten, die im entsprechenden BPEL-Prozess ebenfalls vorkommen. Mit anderen Worten, die Übersetzung muss zentrale, semantische Eigenschaften erhalten. Die Sammlung aller Muster zusammen mit den Regeln für ihre Komposition und ihre Verfeinerung bildet die *Petrinetz-Semantik* für BPEL.

Wir haben die eben geschilderte Idee realisiert und jedes Sprachkonstrukt aus BPEL in ein Muster übersetzt. Damit sind wir in der Lage, durch Verfeinerung und Komposition der Muster jeden in BPEL spezifizierten Prozess in ein Petrinetz abzubilden.

2.2 Muster – am Beispiel von BPEL's receive

Bevor wir auf das receive-Muster, dargestellt in Abb. 1, eingehen, erläutern wir kurz die Notation der Muster. Wir verwenden in unseren Mustern die gängige graphische Notation für Petrinetze (vgl. [7], [16]). Plätze und Transitionen werden mit einem eindeutigen Bezeichner beschriftet, z.B. p1 [2], der sich (entgegen der üblichen Konvention) in dem entsprechenden graphischen Element befindet. Es gibt ausgezeichnete Plätze und Transitionen, die zusätzlich noch einen sprechenden Bezeichner besitzen, z.B. initial, der helfen soll, die Aufgabe des Elementes im Petrinetz zu verstehen. Das Muster selbst ist durch eine gestrichelte Linie eingerahmt. Die Plätze auf dem Rahmen bilden die Schnittstelle zu angrenzenden bzw. umgebenden Mustern. Außerhalb des Rahmens liegen die drei Plätze [Channel], [CorrelationSet] und [Variable], die im umgebenden scope verwaltet werden und dem Muster Objekte schicken ([Channel]), zur Verfügung stellen ([CorrelationSet])

[2] Wir verwenden diesen serifenlosen Schrifttyp, wenn wir uns auf Bezeichner in Abbildungen beziehen.

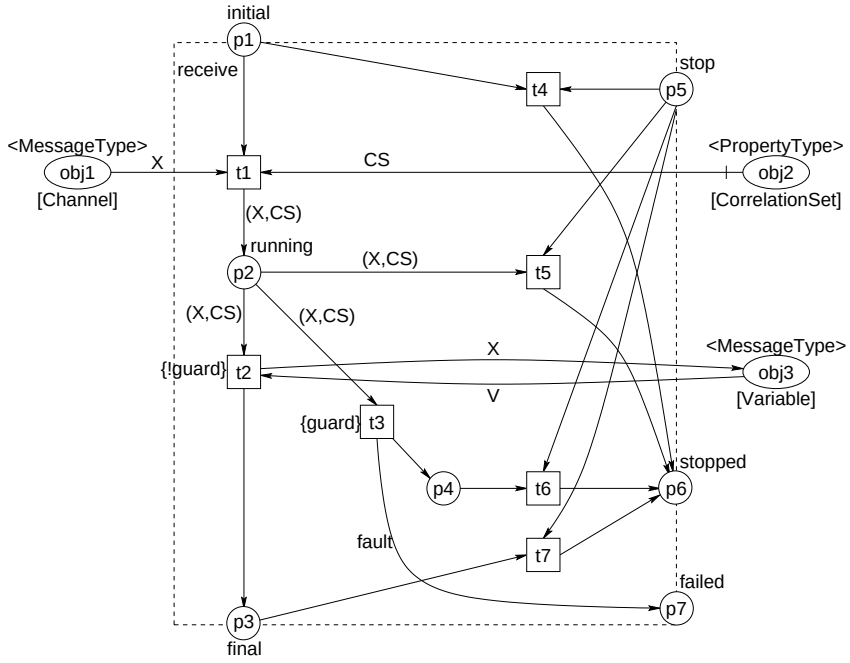

Abbildung 1: receive-Muster

oder speichern ([Variable]). In spitzen Klammern, <MessageType> und <PropertyType>, steht der Typ der entsprechenden Marken.

Innerhalb des Musters sind die Kantenbeschriftungen X und CS *Variablen* im Sinne des Petrinetzes, die im Zuge des Schaltens mit Werten belegt werden. Dabei beginnt eine einfache Variable, z.B. fault, mit einem Kleinbuchstaben und ein Tupel von Variablen, z.B. CS, mit einem Großbuchstaben. Ein Bezeichner, der neben einer Transition angeordnet ist, z.B. {guard}, ist eine Aktivierungsbedingung [16], ein boolescher Ausdruck.

Die BPEL-Aktivität *receive* empfängt über einen Nachrichtenkanal eine Nachricht und vergleicht das CorrelationSet der Nachricht mit dem eigenen CorrelationSet. Das CorrelationSet ist ein Identifikator, der sicherstellt, dass die Nachricht an die richtige Instanz des Prozesses geleitet wird. Anschließend wird der Nachrichteninhalt entweder in eine Variable geschrieben oder ein Fehler signalisiert[3]. Das CorrelationSet der Aktivität receive wird dabei mit dem Prozeßbeginn initialisiert.

Diesem Verhalten der receive-Aktivität ordnen wir das Muster aus Abb. 1 zu. Ist das Muster aktiviert, liegt auf dem *Startplatz* (initial) eine Marke. Wir betrachten zunächst das beabsichtigte Verhalten, den positiven Kontrollfluß, der receive-Aktivität. Dabei wird zunächst mit t1 die Nachricht des Eingabekanals obj1 und das entsprechende CorrelationSet obj2 als Tupel komponiert und auf p2 abgelegt (Belegung der Variablen X und CS mit den Werten von obj1 bzw. obj2). Die Marke auf obj2 wird beim Schalten von t1 nicht konsumiert sondern nur gelesen, denn ein senkrechter Strich kennzeichnet die Kante

[3]Den Fall, dass die Nachricht das CorrelationSet von receive initialisiert, betrachten wir in dieser Arbeit nicht.

zwischen obj2 und t1 als eine Lesekante [15]. Der Fortgang des beabsichtigten Verhaltens hängt nun davon ab, ob ein Fehler aufgetreten ist, ob also beispielsweise das CorrelationSet der Nachricht aus obj1 übereinstimmt mit dem aus obj2. Im Fehlerfall evaluiert die Aktivierungsbedingung guard zu *true* und erzeugt über t3 eine Marke, die über p7 an das Muster zur Fehlerbehandlung weitergereicht wird. Wenn kein Fehler auftritt, wird über t2 die Nachricht aus obj1 der Variable obj3 in der Umgebung des Musters zugewiesen und dort der aktuelle Wert von obj3 gelöscht (Belegung von X und V mit dem Wert von obj1 und obj3). Zugleich wird der Endzustand final erreicht. Das beabsichtigte Verhalten beginnt also in initial und endet mit failed oder final. Es kann jederzeit unterbrochen werden durch ein stop-Signal. In diesem Fall endet das receive-Muster in stopped. Im Fehlerfall *muß* ein stop-Signal kommen, damit das Muster (über p6) terminieren kann.

Das Muster aus Abb. 1 folgt einer Reihe von Konventionen, die das lesen erleichtern. Das beabsichtigte Verhalten, der positive Kontrollfluß von *initial* nach *final* bzw. nach *failed*, verläuft entlang vertikaler Pfeile. Die Kommunikation zwischen dem Prozeß und seinen Partnern verläuft entlang horizontaler Pfeile. Alle anderen Pfeile, das sind die Pfeile, die vom Platz stop weg bzw. zum Platz stopped hin verlaufen, bezeichnen den negativen Kontrollfluß.

Die BPEL-Spezifikation verlangt, dass der positive Kontrollfluss in einem Prozess abgebrochen wird, wenn beispielsweise ein Fehler bei der Abarbeitung signalisiert wird. Darüber, wie dieser Abbruch durchgeführt werden soll, gibt die Spezifikation keine Auskunft. Um diesen Aspekt der Spezifikation zu modellieren, war es für uns notwendig, einige Modellierungsentscheidungen zu treffen, die wir nachfolgend erläutern. Wir sehen diese Entscheidungen als *eine* Möglichkeit der Umsetzung, ob sie die bestmögliche ist, können wir zum jetzigen Zeitpunkt noch nicht sagen.

Wir modellieren das Abbrechen des positiven Kontrollflusses durch das *Abziehen* der Marken aus allen Mustern vom BPEL-Prozessmodell. Um das Abziehen der Marken zu steuern, haben wir das scope-Muster um eine Steuerungskomponente, das *stop*-Muster, erweitert[4]. Zu diesem Muster gibt es kein BPEL-Äquivalent. Das stop-Muster initiiert das Abziehen der Marken, indem an alle sich in Ausführung befindlichen Muster das Signal stop gesendet wird. Aufgrund der Verfeinerung der BPEL-Sprachkonstrukte werden die Muster rekursiv abgeräumt. Nach dem Abziehen der Marken steuert das stop-Muster eine eventuelle Fehlerbehandlung im scope.

Der Kontrollfluss im receive-Muster kann mit Hilfe eines eingebetteten Subnetzes, der *stop-Komponente*, beendet werden. Die stop-Komponente besteht aus den Transitionen t4 – t7, die an die Schnittstellenplätze stop und stopped angeschlossen sind. Soll der positive Kontrollfluss im receive-Muster abgebrochen werden, liegt eine Marke auf stop. In [13] haben wir bewiesen, dass mit Hilfe der stop-Komponenten jeder Prozess abgebrochen werden kann. Analog zum receive-Muster, enthalten auch die Muster aller anderen BPEL-Aktivitäten und des Compensation Handler eine stop-Komponente.

[4]Es reicht nicht, nur das Muster für process zu erweitern, da in BPEL jeder scope isoliert beendet werden muss.

2.3 Das stop-Muster

Bevor wir die Steuerung des Prozessabbruchs genauer betrachten, treffen wir nachstehende Sprachkonventionen für einen scope A, der einen Sub-scope B enthält: Wir bezeichnen B als den *Kindscope* von A und umgekehrt bezeichnet A den *Elternscope* von B. In diesem Abschnitt sei A ein beliebiger scope, der (wie es die BPEL-Spezifikation verlangt) einen Fault Handler und einen Compensation Handler enthält. Das in A enthaltene stop-Muster ist in Abb. 2 dargestellt. Es übernimmt die Steuerung über den scope A, wenn eine in A enthaltene Aktivität einen Fehler signalisiert (fault_in), A von seinem Elternscope zum Abbrechen aufgefordert wird (ft_in) oder im Kontrollfluss des Prozesses eine terminate-Aktivität aktiviert wird und den gesamten Prozess (und damit auch A) abbricht. Im letzten Fall ist entweder terminate oder terminate_up markiert, je nachdem, wo in der Hierarchie des BPEL-Prozesses sich die terminate-Aktivität befindet. Alle weiteren Plätze oben auf dem Rahmen sind Zustandsplätze über die das stop-Muster den aktuellen Zustand des scope A abruft. Die Zustände sind dabei größtenteils dem *Business Agreement Protocol* (BAP) [2] nachempfunden. Das BAP legt einen Satz von Signalen fest, mit denen ein Elternscope mit seinem Kindscope kommuniziert. Über die Plätze unten auf dem Rahmen wird der Fault Hander von A bzw. A's Elternscope aufgerufen. Mit Hilfe der Plätze (auf der rechten Seite) stop und stopped wird A's Sub-Aktivität beendet, wohingegen die Plätze cleanCH und ch_cleaned die Marken aus A's Compensation Handler abziehen.

Die Kante zwischen p10 und t6 unterscheidet sich graphisch und semantisch von den anderen Kanten. Sie konsumiert alle Marken von p10, egal ob 0 oder mehr Marken auf p10 liegen. Graphisch ist sie durch einen durchsichtigen Kreis an ihrem Ursprung hervorgehoben. In der Literatur findet man dazu den Begriff *reset*-Kante [4].

Beim Ablauf des stop-Musters treten alternativ neun Szenarien ein (vgl. [13]). Wir betrachten beispielhaft das Szenario, in dem das receive-Muster die einzige Sub-Aktivität des scope A ist: Signalisiert das receive-Muster einen Fehler, d.h. in Abb. 1 ist failed markiert, dann gelangt die Marke auf den Platz fault_in des stop-Musters. Durch Schalten von t4 initiiert das stop-Muster den Abbruch des positiven Kontrollflusses in A und eine Marke gelangt auf den Platz stop in Abb. 1, da dieser Platz mit dem Platz stop aus Abb. 2 übereinstimmt. Im receive-Muster kann t6 schalten und signalisiert das erfolgreiche Abräumen dem stop-Muster. Die Marke gelangt auf den Platz stopped in Abb. 2. Obwohl auf p10 keine Marke liegt (receive ist die einzige Sub-Aktivität und kann nur einen Fehler signalisieren), ist t5 aktiviert. Mit dem Schalten von t5 wird der positive Kontrollfluss von A erfolgreich abgebrochen. In den in A enthaltenen Mustern sind alle Marken abgeräumt. Anschließend wird der Fault Handler aktiviert, um den Fehler zu behandeln (t7).

Bei verteilter Abarbeitung mehrerer Sub-Aktivitäten im scope (BPEL-Sprachkonstrukt *flow*) kann auch mehr als ein Fehler signalisiert werden. Nur einer dieser Fehler initiiert aber den Prozessabbruch und wird anschließend vom Fault Handler bearbeitet. Alle anderen Fehler, symbolisiert als Marken auf p10, werden nicht bearbeitet und im Modell beim Schalten von t5 gelöscht. Aus diesem Grund verwenden wir die reset-Kante.

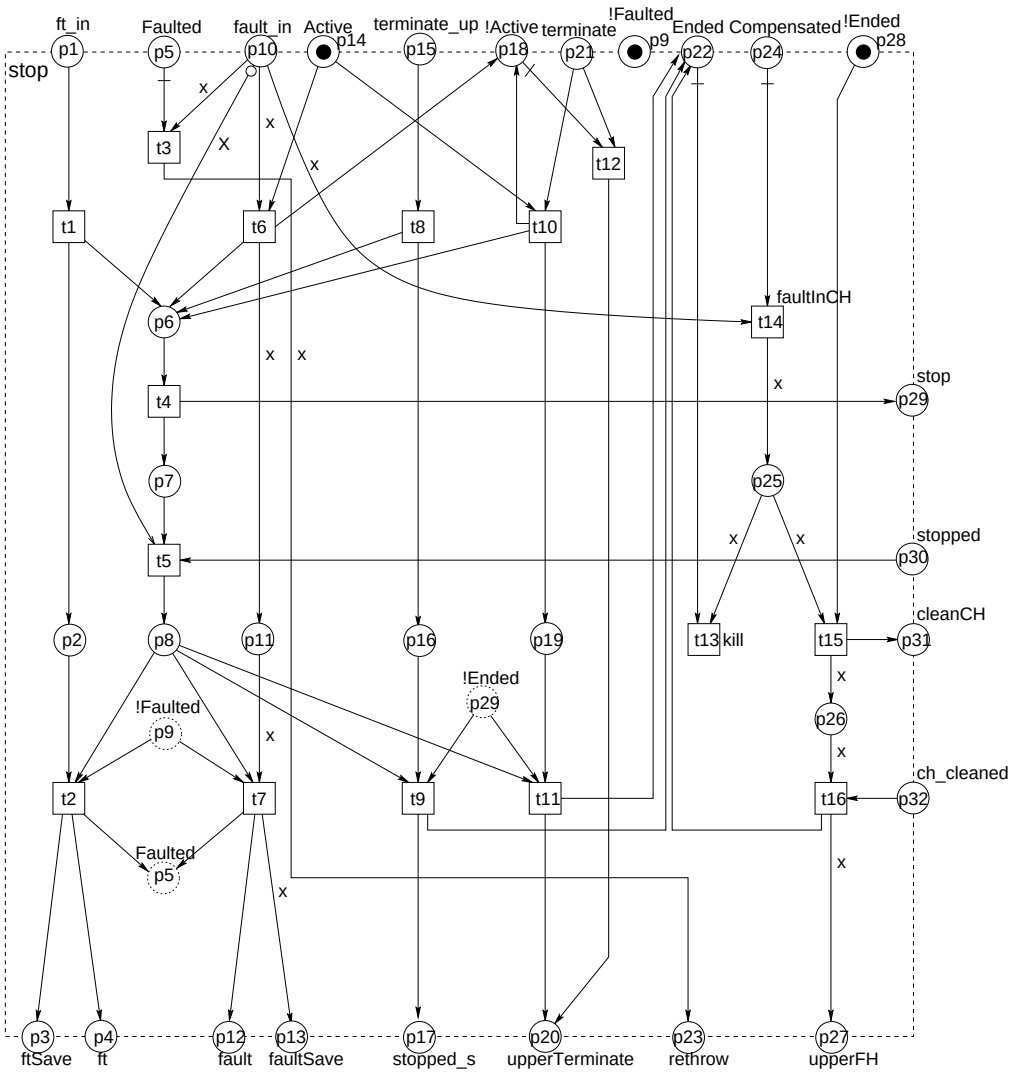

Abbildung 2: stop-Muster

2.4 Weitere Modellierungsentscheidungen bei der Definition der BPEL-Semantik

Betrachten wir nun weitere Konzepte der PetrinetzSemantik, zu deren Umsetzung Modellierungsentscheidungen notwendig waren. Sei dazu A wieder ein scope.

Ein Problem ergab sich bei der Modellierung des Compensation Handler: Wird ein scope A kompensiert, werden die Compensation Handler seiner fehlerfrei abgearbeiteten Kindscopes in der umgekehrten Reihenfolge ihrer Abarbeitung aufgerufen. Somit benötigt das Muster des Compensation Handler in A das Wissen, welche Kindscopes in welcher Reihenfolge kompensiert werden müssen. In [13] haben wir die Verwendung eines Stacks als Speicher der abgearbeiteten Kindscopes motiviert. Dazu wird der Name jedes scope nach seiner fehlerfreien Abarbeitung im Stack des Elternscope gespeichert. Der Stack ist in das Muster des Compensation Handler integriert. Allerdings tritt im Zusammenhang mit dem Stack ein Problem auf, wenn der Fault Handler die Aktivität <*compensate/*> enthält. Diese Aktivität hat die Aufgabe, den scope A zu kompensieren. Um Zugriff auf die Namen aller zu kompensierenden Kindscopes zu erlangen, ruft <compensate/> den Compensation Handler von A auf, um auf den Stack zugreifen zu können. Dieser Aufruf widerspricht der Spezifikation von BPEL, denn ein Fault Handler kann nie den Compensation Handler im eigenen scope aufrufen, ändert aber nichts an den semantischen Eigenschaften von BPEL, wie in [13] bewiesen wurde.

Tritt bei der Abarbeitung des Fault Handler bzw. Compensation Handler ein Fehler auf, wird dieser Fehler in der Semantik nicht direkt an den Elternscope weitergereicht, wie es die BPEL-Spezifikation verlangt. Stattdessen wird der Fehler dem stop-Muster des scope (der den entsprechenden Handler einbettet) signalisiert. Das stop-Muster initiiert das Abräumen der Marken im jeweiligen Handler und reicht erst anschließend den Fehler an den Elternscope weiter. Auch in diesen beiden Fällen wurden die semantischen Eigenschaften von BPEL bewahrt.

Als weiteres Ergebnis haben wir bei der Beschreibung der Szenarien für den Fault Handler einen Deadlock herausgearbeitet: Wird ein scope A mittels forcedTermination zum Beenden gezwungen und beim Kompensieren seines Kindscope tritt ein Fehler auf, dann kann dieser Fehler weder vom Fault Handler in A bearbeitet werden, noch an den Elternscope von A weitergereicht werden. Der Prozess verklemmt somit. Um eigenschaftserhaltend zu BPEL zu sein, tritt dieser Deadlock auch in unserer Semantik auf.

2.5 Validierung der Petrinetz-Semantik

Bei der vorliegenden Semantik stellt sich die Frage nach ihrer Korrektheit. Es ist nicht möglich, formal zu beweisen, dass die Semantik die zentralen, semantischen Eigenschaften der BPEL-Spezifikation bewahrt, da BPEL in einer informalen Spezifikation vorliegt. Aus diesem Grund ist es notwendig, die Semantik auf ihre Plausibilität zu überprüfen. Dazu müssen wir eine hinreichend große Anzahl von BPEL-Prozessen in ein Petrinetz entsprechend der Semantik transformieren und untersuchen, ob das Zusammenspiel der Muster so ist, wie erwartet und ob sich das Modell semantisch äquivalent zum entspre-

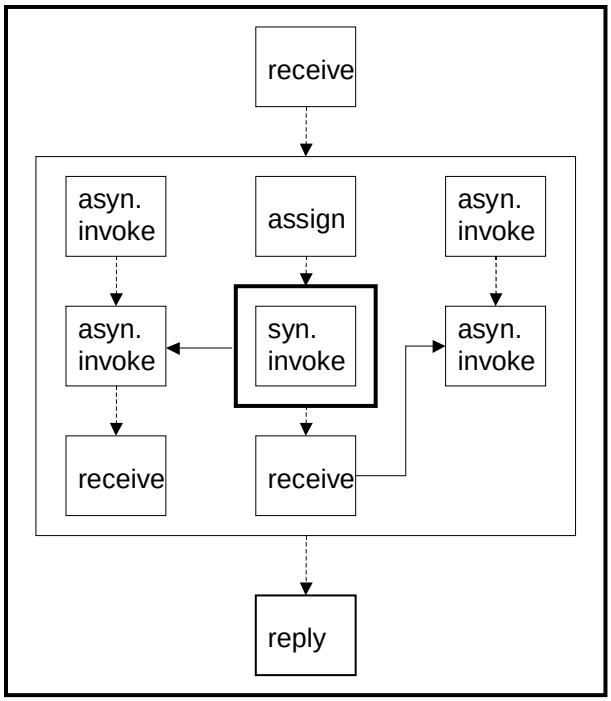

Abbildung 3: Purchase Order Prozess

chenden BPEL-Prozess verhält. Zu diesem Zweck formulieren wir Behauptungen über Eigenschaften unseres Petrinetzmodells. Diese Behauptungen verifizieren wir mit Hilfe des Petrinetz-basierten Werkzeugs LoLA [10] – einem expliziten Model Checker, der Reduktionstechniken wie die Berechnung von Symmetrien [9], Partial Order Reduction mit Hilfe von Stubborn Sets [8] und die Sweep-line Methode [11] implementiert.

In [12] haben wir mit der Validierung der Petrinetz-Semantik begonnen und einen BPEL-Prozess in ein Petrinetz übersetzt. Es handelt sich dabei um eine Modifikation des Purchase Order Prozesses aus der BPEL-Spezifikation [3, S. 14 ff.], visualisiert in Abb. 3. Jede Aktivität ist durch ein Rechteck dargestellt. Einen scope oder einen Prozess visualisieren wir mit einer dickeren Linie. Sequentieller Kontrollfluss ist durch gestrichelte Pfeile gekennzeichnet, Nebenläufigkeit von Aktivitäten durch deren parallele Anordnung. Ein Pfeil mit durchgezogener Linie symbolisiert einen *Link*, ein BPEL-Sprachkonstrukt, das eine Ordnung zwischen zwei BPEL-Aktivitäten festlegt. Betrachten wir den Ablauf: Nachdem der Prozess die Bestellung des Kunden erhalten hat (receive), werden drei Aufgaben parallel initiiert: die Berechnung des Endpreises für die Bestellung (linke Sequenz), die Auswahl eines Spediteurs (mittlere Sequenz) und die zeitliche Planung der Produktion und des Versandes (rechte Sequenz). Zwischen diesen drei Aufgaben existieren Abhängigkeiten, die mit Hilfe der Links modelliert werden. Nachdem die drei Aufgaben erledigt sind, kann die Rechnung an den Kunden gesendet werden (reply).

Obwohl es sich bei diesem Prozess um ein einfaches Beispiel handelt, werden verschiedene Aktivitäten inklusive Fault Handler und Links verwendet. Um die Komplexität

noch etwas zu erhöhen, haben wir das synchrone invoke in einen scope eingebettet, so dass im Falle eines Fehlers eine Fehlernachricht gesendet werden kann. Mit Hilfe dieser Konstruktion konnten wir das Abbrechen eines Kindscope untersuchen. Wir haben in diesem Beispiel von Daten abstrahiert, d.h. eine Nachricht wurde durch eine schwarze Marke modelliert, denn in diesem ersten Test wollten wir unser Hauptaugenmerk auf den Kontrollfluss richten.

Das entsprechend unserer Semantik transformierte Netz hat 158 Plätze und 249 Transitionen und wurde per Hand generiert. Der gesamte Zustandsraum besteht aus 9991 Zuständen. Unter Verwendung von Partial Order Reduction zusammen mit der Sweepline Methode konnte der Zustandsraum auf 1286 Zustände reduziert werden. Die Größe des Zustandsraumes impliziert, dass der so transformierte BPEL-Prozess zu viele Plätze und Transitionen hat, um ihn graphisch zu repräsentieren. Model Checking scheint dagegen ein guter Ansatz zu sein, solche Netze zu verifizieren, denn das Anwenden der Reduktionstechniken war sehr wirkungsvoll.

Wir haben den Prozess nach toten Plätzen und Transitionen untersucht. LoLA berechnete 15 tote Plätze und 101 tote Transitionen. Diese doch erhebliche Anzahl von Plätzen und Transitionen beruht nicht – wie man vielleicht annehmen könnte – auf einem Fehler bei der Transformation des BPEL-Prozesses, sondern ist auf die Muster zurückzuführen: Jedes Muster der Semantik deckt das vollständige Verhalten der entsprechenden Aktivität ab, insbesondere sämtliche Spezialfälle. Beispielsweise enthält der scope aus Abb. 3, der das synchrone invoke einbettet, ein stop-Muster wie in Abb. 2. In diesem Muster ist die Transition `t14` tot. Sie schaltet, wenn ein Fehler bei der Ausführung des Compensation Handler signalisiert wird. Da der scope keine Kindscopes enthält und somit der Compensation Handler bei seiner Aktivierung nichts macht, kann kein Fehler auftreten. Aus diesem Grund enthält das Petrinetz einige überflüssige Konstruktionen, die nie verwendet werden. Es wäre sehr hilfreich für die weitere Arbeit mit der Petrinetz-Semantik, wenn man die Modelle flexibel zur Laufzeit bilden könnte. Mit anderen Worten, wir wollen die Muster erstens genau auf den jeweiligen BPEL-Prozess "maßschneidern", und zweitens auf die Sachverhalte beschränken, die im jeweiligen Kontext benötigt werden.

Desweiteren berechnete LoLA die 196 (erwarteten) Endzustände des Prozesses. Schliesslich haben wir noch einige musterspezifische Eigenschaften verifiziert, wie beispielsweise „jeder Ablauf der *stop* enthält wird später einmal auch *stopped* enthalten" oder „die target-Aktivität eines Links tritt immer nach der source-Aktivität auf".

Der Validierung schloss sich die Verifikation des Purchase Order Prozesses an. Hier wurden die prozessspezifischen Eigenschaften untersucht, beispielsweise die Terminierung oder das Problem, ob der Prozess dem Kunden immer eine Antwort sendet. Bei der letzten Eigenschaft handelt es sich um eine sehr komplexe temporallogische Formel, die aber von unserem Werkzeug aufgrund des kleinen Zustandsraums problemlos berechnet werden konnte.

Die Resultate bei der Validierung der Semantik und der Verifikation des Prozesses mit Hilfe von LoLA waren durchweg positiv. Zwar ist der untersuchte Prozess relativ klein, aber die Ergebnisse lassen erkennen, dass auch größere Prozesse von LoLA verifiziert werden können, da die Reduktionstechniken auf den so gewonnenen Netzen gut zu arbeiten scheinen. Wir werden fortfahren, weitere und vor allem realistisch große BPEL-

Prozesse zu transformieren und zu analysieren, sobald die Implementierung des Parsers abgeschlossen ist, der einen in BPEL spezifizierten Prozess in ein Petrinetz entsprechend der Semantik übersetzt.

3 Abstrakte Modelle

Der Hauptgrund für die Verwendung von High-Level Petrinetzen für die semantische Fundierung von BPEL ist deren Fähigkeit, Datenaspekte originalgetreu abzubilden. Für die Abbildung des Kontrollflusses reichten dagegen Modellierungsmittel, die bereits von Low-Level Petrinetzen bereitgestellt werden, nämlich Plätze, die ununterscheidbare (in den Abbildungen als schwarze Punkte gezeichnete) Marken tragen. Wegen ihrer geringeren Komplexität eignen sich Low-Level Netze natürlich wesentlich besser für die Analyse relevanter Eigenschaften. Die Kontrollstruktur eines BPEL-Prozesses gewinnt man durch *Abstraktion* von Datenaspekten. Im Modell schlägt sich diese Abstraktion dadurch nieder, dass z. B. auf Kommunikationskanälen statt des Inhalts einer Nachricht nur noch die Anwesenheit einer Nachricht modelliert wird. Marken, die Werte tragen, werden dabei zu ununterscheidbaren, *schwarzen* Marken. Für Nachrichtenkanäle, die den Kontrollfluss des modellierten Geschäftsprozesses besonders stark beeinflussen, kann man einzelne inhaltliche Aspekte dadurch abbilden, dass man den urpsrünglichen Kommunikationskanal in mehrere Kanäle aufteilt. Hat z. B. im High-Level Modell eine Nachricht einen Wahrheitswert als Inhalt, führt man im Low-Level Modell zwei Kanäle ein, wobei die Anwesenheit einer Nachricht in einem der Kanäle eine Nachricht mit Inhalt *true* und eine Nachricht im anderen Kanal eine Nachricht mit Inhalt *false* repräsentiert. Demnach reicht es für die meisten Analysefragestellungen aus, Modelle von Geschäftsprozessen auf der Basis von Low-Level Netzen zu studieren. Die Resultate im nächsten Abschnitt dieses Textes basieren auf *Geschäftsprozess-Netzen* (kurz: *GP-Netzen*), einer Klasse von Low-Level Petrinetzen.

Abb. 4 zeigt ein GP-Netz. In einem GP-Netz gibt es (möglicherweise mehrere) ausgezeichnete *Anfangs-* und *Endstellen.* (in der Abbildung α bzw. ω). Im Anfangszustand sind nur die Anfangsstellen markiert. Den Zustand, in dem genau die Endstellen markiert wird, bezeichnen wir als den *Endzustand.* Jede andere Markierung, die keine Transition aktiviert, also auch eine, wo neben den Endstellen weitere Plätze markiert sind, wird als (unerwünschter) *Deadlock* interpretiert. Diese Sicht hat sich im Bereich Geschäftsprozessmodellierung hervorragend bewährt [14]. Ein GP-Netz hat darüberhinaus *Schnittstellenplätze*, die asynchrone Nachrichtenkanäle zur Kommunikation mit der Umgebung repräsentieren. Sie sind streng unterschieden in *Eingangs-* und *Ausgangskanäle*, liegen also entweder ausschießlich im Vorbereich von Transitionen des Netzes, oder ausschließlich in ihrem Nachbereich.

Die Markierung der Schnittstellenplätze kann nicht nur im GP-Netz selbst, sondern auch von der Umgebung des Prozesses verändert werden. Insbesondere kann eine Umgebung Marken von Ausgangskanälen entfernen und Marken auf Eingangskanälen erzeugen.

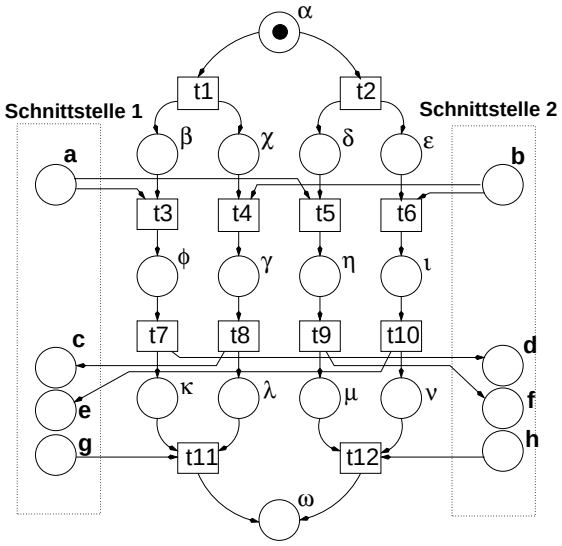

Abbildung 4: GP-Netz. Die umrandeten Plätze sind Schnittstellenplätze. a, b, g und h sind Eingangskanäle, c,d, e und f Ausgangskanäle. Die beiden Umrandungen sollen andeuten, dass für dieses Netz zwei unabhängig agierende Partner vorgesehen sind.

4 Analyse

In den folgenden Unterabschnitten studieren wir *Bedienbarkeit*, eine der in Abschnitt 1 aufgeworfenen Fragestellungen.

4.1 Bedienbarkeit

Ob ein Geschäftsprozess terminiert oder in einen Deadlock gerät, hängt im allgemeinen davon ab, ob sich die Umgebung „protokollgerecht“ verhält. Wir können aber ganz sicher einen Prozess dann als fehlerhaft ausweisen, wenn es für die Umgebung überhaupt keine Möglichkeit gibt, sich so zu verhalten, dass der Prozess garantiert terminiert. Die Umgebung eines Prozesses ist nicht unbedingt ein monolithisches Gebilde. Typischerweise kommuniziert ein Geschäftsprozess (z. B. ein Reisebüro) mit mehreren, autonom agierenden Teilen seiner Umgebung (z. B. einem Kunden einerseits, einer Fluglinie und einem Hotelreservierungsservice andererseits). Einen solchen eigenständigen Teil der Umgebung eines Geschäftsprozesses wollen wir *Partner* nennen. Wir definieren den Begriff *Bedienbarkeit* als die *Existenz* von Partnern, mit denen im Zusammenspiel ein gegebenes GP-Netz immer terminiert.

Um den Begriff des Partners zu formalisieren, müssen wir einige Postulate darüber aufstellen, was wir unter „vernünftigem“ Umgebungsverhalten verstehen wollen. Zunächst postulieren wir, dass eine Umgebung keinen Zugriff auf Interna eines Geschäftsprozesses haben soll. Ein Partner soll daher keinen Zugriff auf die inneren Plätze eines GP-Netzes haben. Weiterhin postulieren wir eine auf Nachrichtenaustausch basierende Kommuni-

kation zwischen Prozess und Umgebung, gestatten also einem Partner Zugriff auf die Schnittstellenplätze eines GP-Netzes. Schließlich wollen wir nichtdeterministisches Verhalten durchaus zulassen.

In unserem Ansatz zum Bedienbarkeitsproblem modellieren wir das Verhalten der Partner durch klassische endliche Automaten. Abb. 5 zeigt zwei solche Automaten. Ein Partner ist möglicherweise nicht mit allen Schnittstellenplätzen des GP-Netzes verbunden, sondern nur mit denen einer Teilmenge M der Menge aller Schnittstellenplätze. Wir sagen: der Partner *benutzt* M. Wenn ein Partner M benutzt, modellieren wir ihn als Automat, der auf dem Alphabet M operiert. Dadurch symbolisieren Übergänge im Automaten Zugriffe auf die benutzten Schnittstellenplätze des GP-Netzes. Ob ein solcher Zugriff das Empfangen oder das Senden einer Nachricht modelliert, geht daraus hervor, ob es sich beim betreffenden Platz um einen Eingangs- oder um einen Ausgangskanal handelt. Einen Übergang mit einem Ausgangskanal des Netzes bezeichnen wir demnach als *Empfangsaktion* und einen Übergang mit einem Eingangskanal des Netzes als *Sendeaktion* des Partners.

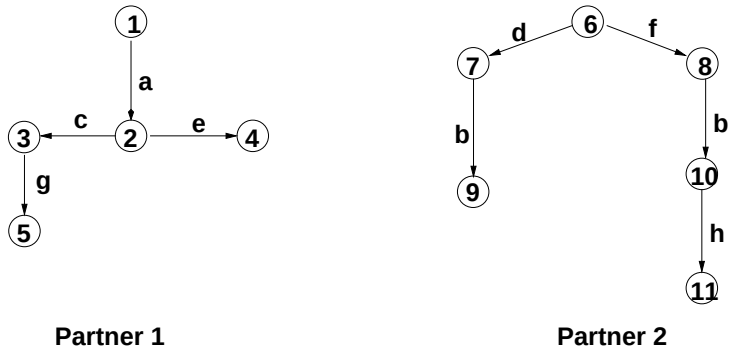

Abbildung 5: Ein mögliches Paar von Partnern des Prozesses aus Abb. 4. Sie bedienen den Prozess. Würde sich Partner 1 allerdings wie Partner 2 verhalten (also durch den gleichen Automaten, nur mit jeweils den Vorgängerbuchstaben beschriftet), so würde das System in dem Zustand verklemmen, wo alle Nachplätze von $t1$ und $t2$ markiert sind.

Das Zusammenspiel eines GP-Netzes G mit einer Menge P von Partnern beschreiben wir als Transitionssystem, das *Kooperationssystem* von G mit P. Abb. 6 zeigt das Kooperationssystem des GP-Netzes aus Abb. 4 mit den beiden Partnern in Abb. 5.

Ein Zustand eines Kooperationssystems ist ein Tupel, das aus einer Markierung des GP-Netzes sowie je einem Zustand der Partner besteht. Jeder Übergang eines Partnerautomaten und jedes Schalten einer Transition des GP-Netzes definiert je einen Übergang im Koordinationssystem. Dabei bewirkt ein mit x beschrifteter Übergang eines Partners, dass eine Marke auf Schnittstellenplatz x des GP-Netzes erzeugt bzw. entfernt wird. Wir verzichten hier auf eine formale Definition.

Bedienbarkeit lässt sich formal definieren als die Frage nach der Existenz einer Menge von Partnern (Automaten) zu einem gegebenen GP-Netz mit der Eigenschaft, dass in dem mit ihnen gebildeten Kooperationssystem von jedem erreichbaren Zustand irgend-

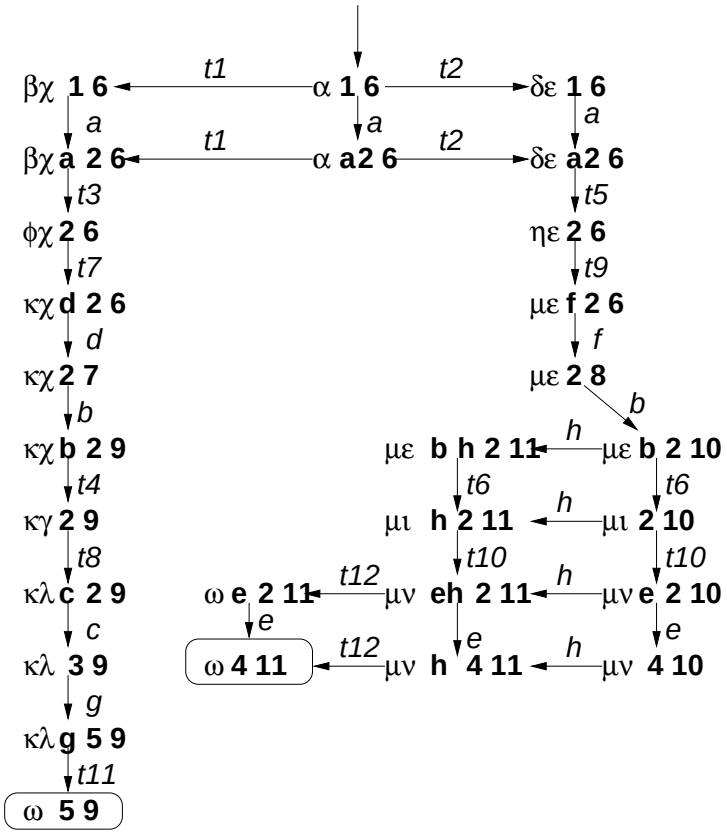

Abbildung 6: Kooperationssystem für das GP-Netz aus Abb. 4 und die Partner aus Abb. 5. Jeder Übergang im Kooperationssystem entspricht einem Übergang im GP-Netz oder einem der Partner. Dabei bewirkt en Übergang eines Partners die Änderung der Markierung eines Schnittstellenplatzes des GP-Netzes.

wann der Endzustand erreicht wird.

Wir unterscheiden für das Bedienbarkeitsproblem drei Fragestellungen. Die erste Fragestellung nennen wir *zentrale Bedienbarkeit*. Hier suchen wir nach einem bedienenden Partner, der Zugriff auf sämtliche Schnittstellen des GP-Netzes hat.

Im zweiten Fall, der *verteilten Bedienbarkeit*, ist die Menge der Schnittstellen in Teilmengen $M_1, \ldots M_k$ partitioniert, und wir suchen nach einer Menge $U = \{U_1, \ldots U_k\}$ von Partnern derart, dass U_i auf dem Alphabet M_i arbeitet und U das GP-Netz bedient.

Der dritte Fall, *lokale Bedienbarkeit*, ist ein Spezialfall verteilter Bedienbarkeit. Hier betrachten wir eine einzelne Teilmenge M_i der Schnittstellen und suchen nach einem Partner, der M_i als Alphabet hat und gemeinsam mit *beliebigen* Partnern für die übrigen Schnittstellenplätze das GP-Netz bedient.

Für alle drei Fragestellungen können wir derzeit Lösungen lediglich für den Fall anbieten, dass das GP-Netz azyklisch ist. Sämtliche in diesem Abschnitt gezeigten Beispielnetze sind azyklisch.

4.2 Entscheidungsverfahren für Bedienbarkeit

Betrachten wir zuerst den zentralen Fall. Zur Lösung des Bedienbarkeitsproblems ist es unser Ziel, einen bedienenden Partner zu konstruieren, wann immer einer existiert. Ein solcher Partner muss mit dem GP-Netz geeignet kommunizieren. Ob eine bestimmte Nachricht in einer bestimmten Situation geeignet oder ungeeignet ist, hängt natürlich davon ab, in welcher Markierung sich das GP-Netz gerade befindet. Die Entscheidung, ob ein Partner eine Nachricht sendet oder auf den Empfang einer Nachricht wartet, muss der Partner aber treffen, ohne direkten Zugriff auf diese Markierung zu haben. Zur Entscheidung kann er lediglich die jeweils aufgelaufene Kommunikation heranziehen. In Kenntnis des (in unserer Fragestellung gegebenen) GP-Netzes kann man aus der aufgelaufenen Kommunikation Rückschlüsse auf die Markierung des GP-Netzes ziehen. Man kann zwar im allgemeinen nicht exakt bestimmen, in welcher Markierung sich das GP-Netz befindet, kann jedoch die Menge der möglichen Netz-Markierungen eingrenzen. Das „Wissen“ eines Partners über bisherige Kommunikation wird in dem jeweils erreichten Zustand des Partners widergespiegelt. Die Rückschlüsse aus der aufgelaufenen Kommunikation auf die Markierungen, in denen sich das GP-Netz befinden kann, können wir also darstellen als eine Abbildung K (*knowledge*), die jedem Automatenzustand q des Partners die Menge derjenigen Markierungen des GP-Netzes zuordnet, in denen sich das Netz befinden *kann*, während der Partner sich in q befindet. Zu gegebenem GP-Netz und einem beliebigen gegebenen Partner können die Werte $K(q)$ aus dem zugehörigen Kooperationssystem abgeleitet werden.

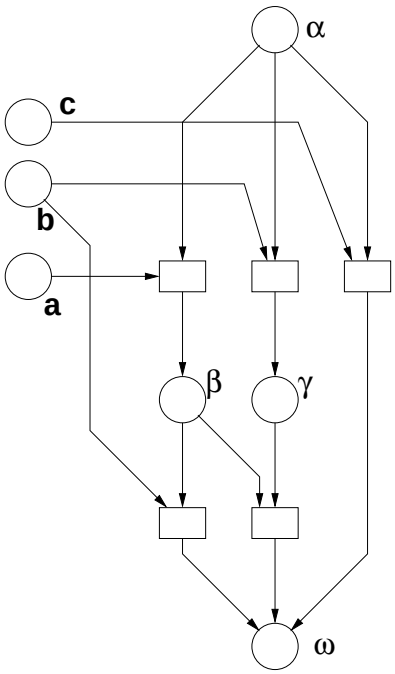

Abbildung 7: Ein Geschäftsprozess-Modell

Mit Hilfe dieser Zuordnung können wir ein Kriterium für bedienende Partner aufstellen. Dazu müssen wir die Deadlocks des GP-Netzes klassifizieren. Wir unterscheiden *interne* und *externe* Deadlocks. Ein Deadlock ist intern, falls jede Netz-Transition einen nicht markierten Vorplatz enthält, der kein Schnittstellenplatz ist. Anderenfalls ist ein Deadlock extern, kann also durch Senden von Nachrichten seitens der Umgebung aufgelöst werden.

Im GP-Netz in Abb. 4 sind die Markierungen ωc und $\kappa\mu b$ Beispiele für interne Deadlocks ($\kappa\mu b$ ist nicht erreichbar) und $\beta\chi g$ und $\kappa\lambda\mu\nu$ Beispiele für externe Deadlocks. Unter einigen technischen, die Allgemeinheit nicht einschränkenden Bedingungen an Partner (siehe [1]) gilt dann:

Theorem 1 (Charakterisierung von bedienenden Partnern) *Ein einzelner Partner bedient das GP-Netz genau dann, wenn für alle Zustände q des Partners gilt:*

1. *$K(q)$ enthält keine internen Deadlocks;*
2. *Zu jedem externen Deadlock d in $K(q)$ gibt es einen Übergang des Partners in q, der in d möglich ist.*
3. *Ist der Endzustand des GP-Netzes Element von $K(q)$, so gibt es in q keine Übergänge.*

Dabei ist eine Empfangsaktion a seitens des Partners in einer Markierung m des GP-Netzes möglich, wenn in m eine Marke auf dem Schnittstellenplatz a liegt, während eine Sendeaktion b dann möglich ist, wenn die Zahl der bisher gesendeten Nachrichten die (wegen vorausgesetzter Azyklizität des Netzes berechenbare) Obergrenze für Kommunikationsschritte nicht bereits überstiegen hat.

Dieses Kriterium liefert ein Verfahren zur Konstruktion eines bedienenden Partners: Wir konstruieren den Partner aus einem Automaten, der in jedem Zustand jede mögliche Sende- und Empfangsaktion vorsieht, also eher mehr als das letztlich akzeptable Verhalten umfasst. Diesen Automaten können wir als Baum konstruieren, dessen Tiefe der erwähnten Obergrenze für Kommunikationsschritte entspricht, und in dem von jedem inneren Knoten mit jedem Zeichen des Alphabets (also mit jedem Schnittstellenplatz des GP-Netzes) ein Übergang vorhanden ist. Zu jedem Zustand q dieses Automaten kann man nun den Wert $K(q)$ bestimmen und in $K(q)$ externe und interne Deadlocks bestimmen. Solange es Zustände gibt, die das Kriterium aus Thm. 1 verletzen, werden diese Zustände sowie die bei diesen Zuständen beginnenden Teilbäume gestrichen. Wir vezichten hier auf die Präsentation von Pseudocode und verweisen stattdessen auf das in Abb. 7 dargestellte Beispiel.

Terminiert die Konstruktion mit einem nichttrivialen Automaten (also einer nichtleeren Zustandsmenge), ist das Netz bedienbar. Führt die Konstruktion dagegen zu einem Automaten mit leerer Zustandsmenge, ist das Netz nicht bedienbar.

Diese Konstruktion ist in dieser Form für große Netze sicher nicht durchführbar, weil der Automat, mit dem wir starten, zu viele Zustände hat. Um diesem Problem zu begegnen, haben wir bereits eine Reihe von Reduktionen vorgeschlagen [17], die genau diese Zustandsexplosion bekämpfen.

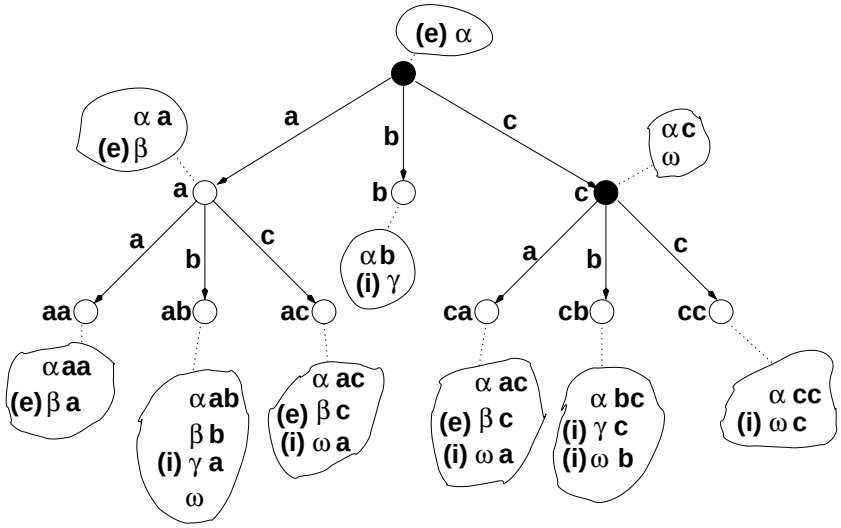

Abbildung 8: Konstruktion eines bedienenden Partners für das GP-Netz in Abb. 7. Gezeigt sind der Automat mit dem man die Konstruktion bedienender Partner beginnt mit den Werten der Wissensfunktion K, wobei externe (e) und interne (i) Deadlocks gekennzeichnet sind. Die Zustände sind mit Sequenzen aus Schnittstellenplätzen beschriftet, zeigen also die aufgelaufene Kommunikation. Wir streichen zunächst alle Knoten, wo ein mit (i) gekennzeichneter Zustand möglich ist. Weiterhin streichen wir den Zustand links unten, weil ein mit (e) gekennzeichneter Zustand auftaucht, der keinen Nachfolger hat (wegen einer vorher berechneten Längenbegrenzung). Nach dem Streichen dieses Zustands muss auch dessen Vorgänger gestrichen werden, weil nun auch er keinen Nachfolger hat, aber einen externen Deadlock möglich macht. Es verbleibt der Automat, der aus den ausgefüllt dargestellten Zuständen besteht. Dieser bedient das Netz in Abb. 7.

Auf der Basis unserer Konstruktion können wir zeigen, dass es zu jedem GP-Netz einen eindeutig bestimmten *liberalsten* bedienenden Partner gibt, also einen, aus dem einzig durch weitere Verhaltenseinschränkung jeder beliebige andere bedienende Partner abgeleitet werden kann.

4.3 Verteilte Bedienbarkeit

Hier gehen wir von einer Partitionierung der Schnittstellen aus, wir setzen also Teilmengen von Schnittstellen als gegeben voraus, so dass jede Schnittstelle zu genau einer dieser Teilmengen gehört. Wir suchen nun für jede Teilmenge M einen Partner der M benutzt, so dass die Menge all dieser Partner das GP-Netz bedient. Dabei kommunizieren die Partner nur mit dem GP-Netz, nicht miteinander (Eine Schar miteinander kommunizierender Partner könnte sonst jede beliebige zentrale Strategie verteilt implementieren, und das verteilte Bedienbarkeitsproblem würde sich auf zentrale Bedienbarkeit redu-

zieren). Die beiden Partner aus Abb. 5 bedienen das Netz in Abb. 4 verteilt für die Partitionierung der Schnittstellenmenge in $\{a, c, e, g\}$ und $\{b, d, f, h\}$.

Wenn ein Netz für eine vorgegebene Partitionierung seiner Schnittstellen von mehreren Partnern verteilt bedient wird, kann man diese Partner zusammen als *einen* Partner auffassen, der das Netz zentral bedient: Mit einem geeigneten Begriff für parallele Komposition ist der zentrale Partner die parallele Komposition der verteilt bedienenden Partner.

Umgekehrt kann man die Frage nach verteilt bedienenden Partnern auffassen als die Frage nach einem zentral bedienenden Partner, der aus Teilen besteht, die jeweils eine der gegebenen Schnittstellenmengen benutzen und sich nicht gegenseitig beeinflussen. Ein Partner U *beeinflusst* einen Partner V, wenn es im zentral bedienenden Partner einen Übergang von einem Zustand q zu einem Zustrand q' gibt mit einem von U benutzten a, und ein von V benutztes b in genau einem der Zustände q oder q' aktiviert ist (also durch den Übergang mit a *aktiviert* oder *deaktiviert* wird).

Theorem 2 (Koordinierte verteilte Bedienbarkeit) *Ein GP-Netz ist bezüglich einer Partitionierung $\{M_1, \ldots, M_k\}$ seiner Schnittstellen verteilt bedienbar genau dann, wenn es einen zentral bedienenden Partner gibt, der so in Teile $T_1, \ldots, T_k$ zerlebgar ist, dass T_i M_i benutzt sich sich die Teile nicht gegenseitig beeinflussen.*

Eine Implementierung der gezeigten Idee führt zu einem Algorithmus, der mit Backtracking arbeitet. Wir verzichten hier auf eine Darstellung dieses Algorithmus. Backtracking als Bestandteil einer Implementation scheint umumgänglich, weil wir es, wie Abb. 9 zeigt, mit *Symmetriebruchproblemen* zu tun haben, für die Backtracking eine der wenigen Lösungsmöglichkeiten darstellt. Das Beispiel zeigt weiterhin, dass es zu einer Partitionierung der Schnittstellen nicht notwendigerweise eine eindeutig bestimmte „liberalste“ Menge von bedienenden Partnern gibt.

4.4 Lokale Bedienbarkeit

Für das Problem der verteilten Bedienbarkeit wird für eine gegebene Partition der Schnittstellen eine entsprechende Menge von Partnern konstruiert die gemeinsam das Netz bedienen. Der Konstrukteur der Partner kann dabei einen Partner in Kenntnis der anderen Partner konstruieren. In Abb. 9 kann er besipielsweise den Partner für $\{b\}$ nur dann so bilden, dass er gar nichts tut, wenn der Partner für $\{a\}$ tatsächlich die Aktion a ausführt.

Hier fragen wir uns nun, inwieweit es sinnvoll und überhaupt möglich ist, zu einer gegebenen Partition der Schnittstellen einen einzelnen der beteiligten Partner *ohne* Kenntnis der anderen Partner zu konstruieren. Wir suchen also einen Partner, der mit *beliebigen* anderen Partnern das gegebene GP-Netz für eine vorgegebene Schnittstellenpartition bedient. In dieser Fassung ist die Fragestellung allerdings trivial: Beliebig „destruktive“ andere Partner können Bedienbarkeit immer verhindern. Wir beschränken uns deshalb auf *kooperative* Partner und definieren zunächst diesen Begriff.

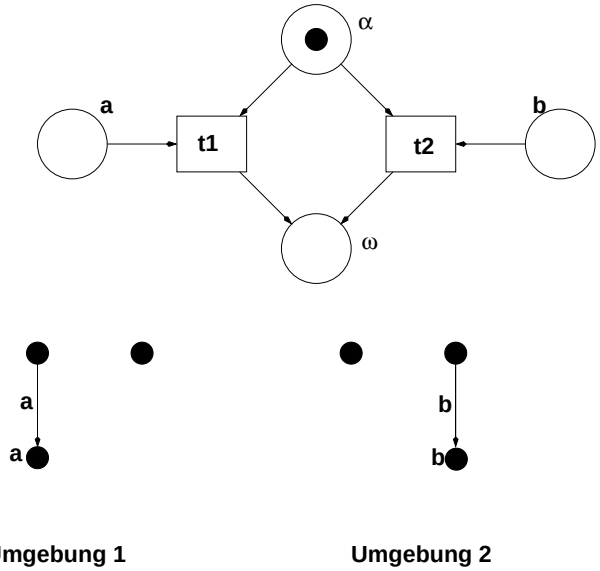

Abbildung 9: Für das oben abgebildete GP-Netz lassen die beiden darunter dargestellten bedienenden Partnerpaare berechnen. Beide bedienen das Netz jeweils verteilt. Obwohl das Netz selbst symmetrisch ist, sind die beiden bedienenden Umgebungen nicht in sich symmetrisch (sondern nur zueinander). Es gibt für das Netz kein symmetrisches Paar bedienender Partner.

Im weiteren sei N ein GP-Netz, $\{M_1, \ldots, M_k\}$ eine Partition seiner Schnittstellen und M_i einer der Schnittstellenmengen. Im weiteren definieren wir, wann ein Partner einen externen Deadlock *beachtet*, wie das auf M_i *reduzierte Netz* N_{M_i} aussieht, wann ein Partner U von N *kooperativ* ist und schließlich, wann ein GP-Netz bezüglich einer Partition seiner Schnittstellen *lokal bedienbar* heißt.

Sei U ein Partner, der M_i benutzt, also, wie in Abschnitt 4.1 definiert, ein endlicher Automat mit Alphabet M_i. Für einen Zustand q von U kann die in Abschnitt 4.2 definierte Menge $K(q)$ externe Deadlocks enthalten. Sei m ein solcher Deadlock, also eine Markierung von N, die ohne Marken auf Eingangskanälen nicht fortsetzbar ist, z.B. die Markierung $\beta\chi$ des Netzes in Abb. 4. U *beachtet* den externen Deadlock m, falls es in U einen Übergang bei q gibt, der in m *möglich* ist, also entweder eine beliebige Sendeaktion oder eine Empfangsaktion von einer bei m markierten Schnittstelle. Es mag verwundern, dass wir in q *nicht* unbedingt eine Sendeaktion auf einen bei m nicht markierten Eingangskanal fordern. Diese Liberalität ist einerseits notwendig, weil U in q Gelegenheit haben soll, vielleicht noch einige Empfangsaktionen auszuführen, oder Sendeaktionen, die später relevant sind, bereits vorzuziehen. Andererseits ist die Liberalität auch ausreichend, weil gegebenenfalls der der externe Deadlock in einem Nachfolgezustand q' von q bestehen bleibt und das Problem also nur aufgeschoben wird. Unendlichern Aufschub können wir durch eine Abschätzung der Höchstzahl der Kommunikationsschritte mit dem (azyklischen!) GP-Netz verhindern.

Das GP-Netz N_{M_i} reduziert N auf die Schnittstellen in M_i. N_{M_i} entsteht aus N durch

Streichen aller Schnittstellen, die nicht in M_i liegen, und der angrenzenden Bögen.

Partner U ist *kooperativ*, wenn er jeden externen Deadlock von N beachtet, bei dem wenigstens ein unmarkierter Eingangskanal in M_i liegt, und N_M zentral bedient. Zum Netz in Abb. 4 ist Partner 1 aus Abb. 5 kooperativ, Partner 2 dagegen nicht, weil dieser den Deadlock $\beta\chi\delta\epsilon$ nicht beachtet, der in $K(6)$ (6 ist der Anfgangszustand von Partner 2) liegt und für den das von Partner 2 benutzte b unmarkiert ist. Partner 2 beachtet diesen Deadlock nicht, weil er im Zustand 6 lediglich die in $\beta\chi\delta\epsilon$ nicht möglichen Empfangsaktionen d und f als Übergänge hat.

Das GP-Netz ist *lokal bedienbar* bezüglich einer Partition $M_1, \ldots, M_k$ der Schnittstellen, falls es zu jedem M_i einen *kooperativen* Partner gibt, der M_i benutzt.

Kooperative Parner sind deshalb interessant, weil kooperative Partner in der Lage sind, mit jedem beliebigen anderen kooperativen Partner zusammenzuarbeiten:

Theorem 3 *Wenn jeder Partner einer Partnermenge kooperativ ist, dann bedient die Partnermenge das gegebene GP-Netz.*

Die Existenz kooperativer Partner können wir, für jede Schnittstellenklasse einzeln, mit einem Algorithmus entscheiden, der mit dem für zentrale Bedienbarkeit fast identisch ist.

Verhalten sich beide Partner aus Abb. 5 nach dem Schema wie Partner 1 (also kooperativ), terminiert das System in Abb. 4, während es in der Markierung $\beta\chi\delta\epsilon$ verklemmen kann, wenn beide sich wie Partner 2 verhalten. Für das Netz in Abb. 9 existieren keine kooperativen Partner, es ist nicht lokal bedienbar.

5 Zusammenfassung

Petrinetze sind ein Ausdrucksmittel, mit dem zahlreiche zentrale Fragestellungen von verteilten Geschäftsprozessen formuliert und gelöst werden können. Datenabhängiger und -unabhängiger Kontrollfluß kann (mit HL- bzw. LL-Netzen) gleichermaßen adäquat behandelt werden. Eine Reihe von Werkzeugen, insbesondere LoLA [10] unterstützt die Analyse solcher Netze auch für größere Fallstudien.

Literatur

[1] ARSTEN SCHMIDT. Controlability of business processes. Tech. Rep. 180, Humboldt-Universität zu Berlin, to appear 2005.

[2] CABRERA, COPELAND, COX, FREUND, KLEIN, STOREY, AND THATTE. *Web Services Transaction.* Vorschlag zur Standardisierung, Version 1.0, Aug. 2002. `http://ibm.com/developerworks/webservices/library/ws-transpec/`.

[3] CURBERA, GOLAND, KLEIN, LEYMANN, ROLLER, THATTE, AND WEERAWARANA. Business Process Execution Language for Web Services, Version 1.1. Tech. rep., BEA Systems, International Business Machines Corporation, Microsoft Corporation, May 2003.

[4] Dufourd, C., Finkel, A., and Schnoebelen, P. Reset nets between decidability and undecidability. In *Proc. 25th Int. Coll. Automata, Languages, and Programming (ICALP'98), Aalborg, Denmark, July 1998* (1998), K. Spies and B. Schätz, Eds., Lecture Notes in Computer Science 1443, Springer, pp. 103–115.

[5] Howard Foster, Sebastian Uchitel, J. M., and Kramer, J. Model-based Verification of Web Service Compositions. In *18th IEEE International Conference on Automated Software Engineering* (October 2003), IEEE Computer Society, pp. 152–163.

[6] Koshkina, M., and van Breugel, F. Verification of business processes for web services. Report cs-2003-11, York University, oct 2003.

[7] Reisig, W. *Petri Nets.*, eatcs monographs on theoretical computer science ed. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1985.

[8] Schmidt, K. Stubborn set for standard properties. In *20th International Conference on Application and Theory of Petri Nets* (1999), Nielsen, M. and Simpson, D., Eds., LNCS 1639, Springer-Verlag, pp. 46–65.

[9] Schmidt, K. How to calculate symmetries of petri nets. *Acta Informatica 36* (2000), 545–590.

[10] Schmidt, K. Lola – a low level analyser. In *International Conference on Application and Theory of Petri Nets* (2000), Nielsen, M. and Simpson, D., Eds., LNCS 1825, Springer-Verlag, p. 465 ff.

[11] Schmidt, K. Automated generation of a progress measure for the sweep-line method. In *Proc. 10th Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2004* (2004), vol. 2988 of *LNCS*, Springer-Verlag, pp. 192–204.

[12] Schmidt, K., and Stahl, C. A petri net semantic for bpel4ws - validation and application. In *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN'04)* (october 2004), E. Kindler, Ed., Universität Paderborn, pp. 1–6.

[13] Stahl, C. Transformation von BPEL4WS in Petrinetze. Diplomarbeit, Humboldt-Universität zu Berlin, April 2004.

[14] van der Aalst, W. M. P. The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers 8*, 1 (1998), 21–66.

[15] Weber, M. *Allgemeine Konzepte zur software-technischen Unterstützung verschiedener Petrinetz-Typen.* PhD thesis, Humboldt-Universität zu Berlin, 2003. URL `http://dochost.rz.hu-berlin.de/dissertationen/weber-michael-2002-12-16/PDF/Weber.pdf`.

[16] Weber, M., Walter, R., Völzer, H., Vesper, T., Reisig, W., Peuker, S., Kindler, E., Freiheit, J., and Desel, J. DAWN. Petrinetzmodelle zur Verifikation Verteilter Algorithmen. Informatik-Bericht Nr. 88, Institut für Informatik, Humboldt-Universität zu Berlin, Nov. 1997.

[17] Weinberg, D. Analyse der Bedienbarkeit. Diplomarbeit, Humboldt-Universität zu Berlin, Oktober 2004.