# Controller-Synthese für Services mit Daten

Dissertation
zur Erlangung des akademischen Grades
doctor rerum naturalium
(Dr. rer. nat)

im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät der
Humboldt-Universität zu Berlin

von
Dipl.-Math. oec. Franziska Bathelt-Tok

Präsidentin der Humboldt-Universität zu Berlin
Prof. Dr.-Ing. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät
Prof. Dr. Elmar Kulke

---

Gutachter/innen:

1. Prof. Dr. Wolfgang Reisig, Humboldt-Universität zu Berlin
2. Prof. Dr. Dr. h. c. Frank Leymann, Universität Stuttgart
3. Prof. Dr. Matthias Weidlich, Humboldt-Universität zu Berlin

Tag der mündlichen Prüfung: 22.11.2017

## Selbständigkeitserklärung

Ich erkläre, dass ich die Dissertation selbständig und nur unter Verwendung der von mir gemäß §7 Abs. 3 der Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät, veröffentlicht im Amtlichen Mitteilungsblatt der Humboldt-Universität zu Berlin Nr. 126/2014 am 18.11.2014 angegebenen Hilfsmittel angefertigt habe.

---

Franziska Bathelt-Tok

# Inhalt

# Teil I
# Allgemeine Grundlagen

# 1 Über diese Arbeit

In diesem Kapitel beschreiben wir zunächst den Kontext der vorliegenden Arbeit (Abschnitt 1.1). Darauf aufbauend formulieren wir das Problem, welches in dieser Arbeit betrachtet wird (Abschnitt 1.2). Anschließend geben wir einen kurzen Überblick über die von uns erreichten Resultate (Abschnitt 1.3) und stellen unsere Lösung auf einer abstrakten Ebene dar (Abschnitt 1.4). Abschließend, erläutern wir den Aufbau der vorliegenden Arbeit (Abschnitt 1.5).

## 1.1 Kontext der Arbeit

Die steigende Nachfrage an immer komplexeren Systemen in verschiedenen wirtschaftlichen Bereichen, wie Geschäftsprozesskoordinierung, Softwaretechnik oder Geräteentwicklung erfordert Strategien, die Wartbarkeit und Wiederverwendbarkeit unterstützen. An diesem Punkt setzt das Paradigma der *serviceorientierten Architekturen* (*SOAn*) an. Diesem zufolge bestehen große Systeme aus weniger komplexen, miteinander interagierenden Komponenten, die im Folgenden als *Services* bezeichnet werden. Diese Services werden autonom entwickelt und können bei Bedarf mit anderen komponiert werden, um eine gewünschte Funktionalität zu erreichen. Dies wird durch die Tatsache ermöglicht, dass Services über sogenannte *Interfaces* verfügen, die die Kommunikation mit der Umgebung bzw. anderen Services durch den Austausch von Nachrichten erlauben.

Besonders in sicherheitskritischen Bereichen ist es unentbehrlich die Korrektheit der Komposition sicherzustellen, da eine fehlerbehaftete Interaktion zu hohen finanziellen Einbußen oder sogar zu lebensbedrohlichen Situationen führen kann.

Im Allgemeinen ist die Korrektheit der Komposition verschiedener Services jedoch nicht per se gegeben. Betrachtet man zum Beispiel zwei Services $S_1$ und $S_2$, so kann ihre Komposition $S_1 \otimes S_2$ schon an inkompatiblen Interfaces scheitern. Dies kann beispielsweise der Fall sein, wenn der Typ einer von $S_1$ ausgehenden Nachricht nicht durch $S_2$ interpretiert bzw. verarbeitet werden kann. Abgesehen von den eventuell auftretenden Daten-Inkompatibilitäten, ist es zudem möglich, dass $S_1 \otimes S_2$ nicht das gewünschte Verhalten aufweist. Es

könnte beispielsweise passieren, dass die Services sich gegenseitig blockieren und keiner der beiden seine Aufgabe beenden kann. Dies wird als *Deadlock* bezeichnet.

Aus diesem Grund ist es zum einen wichtig, die zu garantierenden Verhaltenseigenschaften des komponierten Systems im Vorfeld formal zu definieren, beispielsweise mit Hilfe temporaler Logik. Zum anderen müssen Kompositionsmethoden die durch die unabhängige Entwicklung auftretenden Interface-Inkompatibilitäten behandeln. Da die einzelnen Services durch unterschiedliche Anbieter bereitgestellt werden, ist im Allgemeinen eine interne Veränderung des jeweiligen Services nicht möglich. Daher greift man in solchen Fällen auf die Entwicklung einer Zwischenkomponente, dem sogenannten *Controller* bzw. *Adapter*, zurück. Dieser wirkt als Mediator und passt den Typ der Nachrichten an bzw. leitet sie an die entsprechenden Stellen weiter. Ziel dieser Arbeit ist es, einen Controller $C$ so zu synthetisieren, dass das komponierte System $S_1 \otimes C \otimes S_2$, die im Vorfeld formal definierten Verhaltenseigenschaften sicherstellt und somit korrekt ist.

Insbesondere dieses Korrektheitskriterium ermöglicht eine Anwendung auf sicherheitskritische Systeme, wie sie häufig im medizinischen Bereich zu finden sind. Die Ausrichtung auf den medizinischen Bereich ist dabei nicht einschränkend zu verstehen, sondern begründet sich in der Tatsache, dass diese Arbeit im Rahmen einer Promotion in dem DFG-finanzierten Graduiertenkolleg SOAMED[1] entstand. SOAMED beschäftigt sich mit der Anwendung serviceorientierter Architekturen im Medizin- und Gesundheitsbereich mit dem Ziel, formal fundierte Lösungen für domänenspezifische Probleme zur Verfügung zu stellen.

Da der medizinische Sektor stetig komplexer und immer gefragter wird, werden die aufkommenden Fragestellungen immer vielschichtiger. Sie reichen von der Optimierung von Behandlungsplänen über Datensicherheit und Datenschutz bis hin zur Gewährleistung einer sicheren Interoperabilität von medizinischen Geräten.

In all diesen Einsatzbereichen bestehen die betrachteten Systeme aus einzelnen Komponenten, in denen einzelne Funktionalitäten gekapselt sind und die unabhängig von einander entwickelt wurden. Der Aufbau solcher Systeme folgt somit dem SOA-Paradigma. Durch die unabhängige Entwicklung wird auf der einen Seite eine leichtere Wiederverwendbarkeit und Wartbarkeit erreicht. Auf der anderen Seite rückt jedoch die Sicherstellung einer zuverlässigen und korrekten Kommunikation zwischen den Komponenten immer mehr in den Vordergrund.

In dieser Arbeit fokussieren wir uns auf die sicherheitskritische Domäne der Gewährleistung der Interoperabilität von medizinischen Geräten durch Service-Kompositionstechniken. Dafür existieren zwei Gründe. Zum einen streben Hersteller medizinischer Geräte eine Vormachtsstellung an, indem sie eine Anbindung an Geräte anderer Hersteller nicht unterstützen. Die Käufer werden

[1] Service-orientierte Architekturen zur Integration Software-gestützter Prozesse am Beispiel des Gesundheitswesens und der Medizintechnik, www.soamed.de

dadurch gezwungen weitere Geräte desselben Herstellers zu erwerben, obwohl funktionsgleiche Geräte von anderen Herstellern viel kostengünstiger wären. Durch die so entstehende Abhängigkeit ist es den Herstellern möglich, die Preise zu erhöhen und somit den Profit zu maximieren. Die Gewährleistung einer sicheren Interoperabilität medizinischer Geräte, unabhängig von ihren Herstellern, könnte eine solche Monopolposition aufbrechen und die Kosten für die Käufer verringern.

Auf der anderen Seite wächst die Nachfrage nach immer komplexeren, verteilten eingebetteten Systemen beispielsweise im Bereich der Diagnostik. Diese komplexeren Systeme ermöglichen eine vielfältige Abklärung von Erkrankungen, erfordern jedoch auch eine sichere und zuverlässige Interoperabilität zwischen den medizinischen Geräten. Die Konzepte aus dem Bereich der automatisierten Service-Komposition, die für SOAs entwickelt wurden, sind hinsichtlich der Synthese von Controllern, die datenabhängige, funktionelle und sicherheitskritische Eigenschaften sicherstellen, sehr vielversprechend.

Die Anwendbarkeit dieser Konzepte wird aktuell noch durch die fehlende Datenbehandlung während der Synthese behindert. In der Literatur, die sich mit Service-Komposition durch Controller-Synthese beschäftigt, werden datenabhängiges Verhalten, sowie der durch Nachrichteninhalte bedingte Austausch von Daten, nicht betrachtet. Daher sind existierende Ansätze bisher nur in der Lage Controller zu synthetisieren, die datenunabhängige Eigenschaften garantieren können. Eine Möglichkeit dem entgegen zu treten, besteht darin, das System im Nachhinein manuell auf datenabhängige Eigenschaften zu überprüfen bzw. den Controller um Datenabhängigkeiten zu erweitern. Dies ist jedoch sehr zeitaufwändig, kostenintensiv und fehleranfällig. Auf dieser Lücke liegt der Fokus unseres Ansatzes zur Controller-Synthese, der Verhaltenseigenschaften bei der Komposition von Services mit Daten garantiert. Daher hat er das Potential im Bereich der Gewährleistung von Interoperabilität höchst nutzbringend zu sein.

## 1.2 Problem

In dieser Arbeit betrachten wir allgemein die folgende Fragestellung:

*Wie können wir für zwei gegebene Services und einer Menge von Eigenschaften einen Controller automatisch synthetisieren, sodass das komponierte System, bestehend aus den Services und dem Controller, alle Eigenschaften erfüllt?*

Bevor wir in Abschnitt 1.2.2 die Problemstellung konkretisieren, klären wir zunächst ein paar grundlegende Begriffe (Abschnitt 1.2.1).

### 1.2.1 Grundlegende Begriffe

Wir haben bereits in Abschnitt 1.1 einige Begriffe genannt, auf die wir nun genauer eingehen.

Service: Ein *Service* ist eine Komponente, die durch ein Interface mit anderen Komponenten bzw. der Umgebung durch Nachrichtenaustausch kommunizieren kann. Jeder Service stellt eine gewisse Funktionalität zur Verfügung.

Service-Komposition: Services können *komponiert* werden, um gemeinsam eine Aufgabe zu lösen. Dies geschieht entweder durch das direkte Verbinden ihrer Interfaces oder durch die Bereitstellung eines Controllers, der mit den Interfaces der Services verbunden wird und die Kommunikation steuert. Bei dem so entstehenden System handelt es sich wiederum um einen Service mit einem möglicherweise leerem Interface.

Controller: Ein *Controller* ist ein spezieller Service, der mit anderen Services komponiert wird, den Nachrichtenaustausch zwischen diesen Services steuert und dadurch ihr Zusammenspiel koordiniert.

Eigenschaft: Eine *Eigenschaft* definiert ein Merkmal eines Systems, in unserem Fall eines Services. Wir betrachten hierbei funktionale und datenabhängige Eigenschaften.

funktional: Eine *funktionale Eigenschaft* beschreibt, was der Service tun soll.

datenabhängig: Eine *datenabhängige Eigenschaft* beschreibt, wie sich der Service bei einem bestimmten Nachrichteninhalt verhalten soll.

Korrektheit: Ein Service wird in dieser Arbeit als *korrekt* bezüglich einer Eigenschaft bezeichnet, wenn er diese Eigenschaft erfüllt.

### 1.2.2 Forschungsfragen

In der Literatur existieren bisher viele Ansätze zur Service-Komposition, die jedoch einige Probleme aufweisen, aus denen sich die folgenden Forschungsfragen für diese Arbeit ergeben.

1. Wie kann man datenabhängige Eigenschaften spezifizieren?
2. Wie kann man Services mit Daten und ihre asynchrone Kommunikation modellieren?
3. Wie kann man aus gegebenen Service-Modellen $S_1, S_2$ und einer datenabhängigen Eigenschaft $\varphi$ automatisiert einen Controller $C$ synthetisieren, sodass $S_1 \otimes C \otimes S_2$ korrekt bezüglich $\varphi$ ist?

## 1.3 Resultate

In diesem Abschnitt fassen wir kurz die Hauptresultate dieser Arbeit zusammen, die aus der Beantwortung der in Abschnitt 1.2.2 gestellten Forschungsfragen resultierten.

### 1.3.1 Formale Spezifikation datenabhängiger Eigenschaften mittels temporaler Logik

In Abschnitt 4.1 schlagen wir eine Untermenge der Baumzeitlogik CTL (engl.: *Computation Tree Logic* (*CTL*)) als Formalismus zur Spezifikation datenabhängiger Eigenschaften vor. CTL ist eine anerkannte temporale Logik, die oft zur Spezifikation von Eigenschaften verteilter Systeme genutzt wird. Diese von uns definierte Untermenge nennen wir *Restricted Computation Tree Logic* (*RCTL*) .

### 1.3.2 Formale Modellierung von Services mit Daten und ihrer Kommunikation mittels offener, algebraischer Petrinetze

In Abschnitt 3.3 schlagen wir *Algebraische Petrinetze (APNs)* als Formalismus zur Modellierung von Services mit Daten vor. Dies ist auch der Formalismus der Wahl um den gewünschten Controller zu repräsentieren. Der Vollständigkeit halber sei an dieser Stelle erwähnt, dass die von uns betrachteten APNs Relationen wie $<, \leq, \geq$ und $>$ enthalten können. Um diese zu handhaben, setzen wir für unsere Beispiele voraus, dass es sich bei den Datenwerten um Zahlen handelt und dass die Relationen so zu interpretieren sind, wie es auf Zahlen üblich ist. In Abschnitt 4.2 erweitern wir die algebraischen Petrinetze, damit die in RCTL spezifizierten Eigenschaften mit den Service-Modellen in Verbindung gesetzt werden können. Solche Netze nennen wir *RCTL-Netze*.

### 1.3.3 Automatisierte Synthese korrekter Controller

In Kapitel 5 schlagen wir einen fünfstufigen Ansatz zur automatisierten Controller-Synthese vor, der korrekt-per-Konstruktion ist. Die Synthese erfolgt dabei direkt aus den sicherzustellenden Eigenschaften und den Service-Modellen. Dieser Prozess erfolgt ohne manuellen Aufwand und verringert so die Fehleranfälligkeit des Systems. Der resultierende Controller stellt sicher, dass das komponierte System, bestehend aus Services und dem Controller, alle Verhaltenseigenschaften erfüllt.

## 1.4 Lösung

Nachdem wir die Hauptresultate im vorangegangenen Abschnitt zusammengefasst haben, gehen wir in diesem Abschnitt kurz darauf ein, wie wir diese Resultate erreichen.

Zur Beantwortung der zentralen Forschungsfrage (Frage 3. in Abschnitt 1.2.2) entwickeln wir einen Ansatz, mit dessen Hilfe wir aus gegebenen Services und einer Menge von Eigenschaften automatisiert einen korrekten Controller synthetisieren können.

Dazu nehmen wir an, dass das Verhalten jedes Services als offenes, algebraisches Petrinetz gegeben und die sicherzustellenden Eigenschaften als RCTL - Formeln spezifiziert sind.

*Algebraische Petrinetze (APNs)* gehören zur Klasse der high-level Petrinetze. Sie vereinigen die leicht verständlichen Repräsentationstechniken von low-level Petrinetzen mit den fundierten Möglichkeiten von algebraischen Spezifikationen, die es erlauben Datentypen und -modifikationen abstrakt zu definieren. Damit ist es möglich, sowohl den Kontrollfluss als auch den Datenfluss gleichzeitig zu modellieren. Obwohl APNs ausdrucksstärker sind als low-level Petrinetze, können sie dennoch als Spezialisierung letzterer aufgefasst werden. Dadurch ist es möglich, die gut untersuchten und fundierten Analyse- und Verifikationsmethoden von low-level Petrinetzen auch auf algebraische Petrinetze anzuwenden.

Während APNs sehr gut geeignet sind, um Services mit Daten zu modellieren, ist es mit ihnen nicht möglich, in probater Weise (datenabhängige) Eigenschaften zu spezifizieren. Aus diesem Grund schlagen wir vor, dafür eine Untermenge von CTL zu nutzen, die wir als *RCTL* bezeichnen. Die bereits in Abschnitt 1.3.1 erwähnte Tatsache, dass CTL oft zur Verifikation verteilter Systeme genutzt wird, ist insbesondere für sicherheitskritische Bereiche von großer Bedeutung, da hier Simulationen und Tests nicht ausreichend sind, um alle potentiellen Fehler aufzudecken. Grund dafür ist, dass beim Testen und Simulieren das System nur eine gewisse, endliche Menge an Testfällen durchläuft. Diese Testmenge deckt jedoch nicht notwendiger Weise alle potentiell eintretenden Szenarien ab. Im Gegensatz dazu ist die Verifikation, die beispielsweise durch das sogenannte Model-Checking erreicht wird, vollständig. Das bedeutet, dass für jede Eingabe alle Ausführungsmöglichkeiten des Systems hinsichtlich einer gegebenen Eigenschaft untersucht werden. Dies bedarf jedoch sehr viel Berechnungsaufwand bzw. ist bei unendlichen Datendomänen nicht mehr umsetzbar, da in diesem Falle unendlich viele potentielle Eingaben existieren.

In unserem Ansatz verzichten wir auf bekannte Verifikationstechniken und synthetisieren einen Controller direkt aus den gegebenen Services und den Eigenschaften.

Dafür erweitern wir initial das Konzept der algebraischen Petrinetze, indem wir Transitionen temporale Pfad-Operatoren zuweisen und in späteren Schrit-

ten deren Semantik beachten. Diese so erweiterten APNs bezeichnen wir im weiteren Verlauf als *RCTL-Netze*.

Im ersten Schritt unseres Ansatzes extrahieren wir aus jedem gegebenen Service-APN eine Menge von RCTL-Formeln, die das Verhalten des Services an den Interfaces spezifizieren. Dieses für die Kommunikation wichtige Verhalten bezeichnen wir im weiteren Verlauf als *beobachtbares Verhalten*.

Das beobachtbare Verhalten sowie die sicherzustellenden Eigenschaften werden im zweiten Schritt in RCTL-Netze übersetzt. Diese RCTL-Netze werden im dritten Schritt zu einem Gesamtnetz komponiert, in dem das Service-Verhalten abgebildet ist und alle Eigenschaften erfüllt sind. Aus diesem Gesamtnetz kann der gewünschte Controller mit Hilfe des von uns definierten Extraktionsschrittes separiert werden. Dazu werden basierend auf Informationen aus den originalen Service-Netzen $S_1, S_2$ nur die Elemente aus dem Gesamtnetz extrahiert die nicht in $S_1$ oder $S_2$ enthalten sind (mit Ausnahme der Interfaceplätze). Bei dem daraus resultierenden Netz handelt es sich um ein RCTL-Netz. Um die Komposition mit den originalen Service-APNs zu ermöglichen, wird im fünften Schritt das RCTL-Netz auf ein APN zurückgeführt. Dabei wird die Semantik der im RCTL-Netz auftretenden Transitions-Label bewahrt.

Zusammenfassend gesagt, schlagen wir einen Ansatz vor, der auf Basis von APNs und RCTL automatisiert einen Controller synthetisiert, um Services mit Daten hinsichtlich spezifizierter Eigenschaften korrekt zu komponieren. Mit unserem Ansatz gehen vier wesentliche Vorteile einher. Erstens unterstützt unser Ansatz durch die Nutzung von RCTL eine geeignete Spezifikation datenabhängiger und funktionaler Eigenschaften. Zweitens haben wir eine beweisbar korrekte Übersetzung von RCTL-Formeln in RCTL-Netze entwickelt. Drittens arbeitet das Synthese-Verfahren vollautomatisch und ist per Konstruktion korrekt. Der resultierende Controller stellt sicher, dass das komponierte System, bestehend aus Services und Controller, die im Vorfeld spezifizierten Eigenschaften erfüllt. Viertens ist der resultierende Controller nach Beendigung des Synthese-Verfahrens als APN gegeben. Dadurch können wir etablierte Verfahren zur Komposition der Services und des Controller nutzen. Daraus resultiert ein APN, auf das wir existierende Analyse- und Verifikationsverfahren anwenden können, um beispielsweise das komponierte System auf die Erfüllung weiterer Eigenschaften zu überprüfen.

## 1.5 Aufbau der Arbeit

Diese Arbeit ist wie folgt strukturiert.

Im *ersten Teil* führen wir allgemeine Grundlagen ein, um die Arbeit in einen Kontext zu setzen und die fundamentalen Konzepte, auf denen sie beruht, zu wiederholen. Dazu erläutern wir in Kapitel 2 die Grundkonzepte *Serviceorientierter Architekturen (SOAs)*, *Algebraischer Petrinetze (APNs)* und der *Computation Tree Logic (CTL)* ein. Zusätzlich präsentieren wir unser

laufendes Beispiel, auf das wir in späteren Kapiteln immer wieder verweisen und an dem wir die einzelnen Schritte unseres Controller-Synthese Prozess erläutern. Um den Neuigkeitswert unseres Ansatzes zu verdeutlichen, diskutieren wir in Kapitel 3 existierende Arbeiten, die sich bereits mit dem Thema Service-Komposition durch Controller-Synthese beschäftigt haben und dabei in gewisser Weise den Umgang mit Daten beachten. Zusätzlich betrachten wir auch bestehende Ansätze für die Überführung von temporaler Logik in operationale Modelle und prüfen Möglichkeiten Service-Verhalten zu modellieren.

Im *zweiten Teil* passen wir die eingeführten Formalismen (APN und CTL) für unsere Bedürfnisse an und definieren dafür zwei neue Formalismen ( Kapitel 4) und Operationen auf diesen (Kapitel 5). Ziel dabei ist es, die Überführung von CTL-Formeln in APNs zu ermöglichen.

Die im zweiten Teil definierten Formalismen und Operationen auf diesen, nutzen wir im *dritten Teil* für die Einführung unseres Prozesses zur Controller-Synthese für Services mit Daten (siehe Kapitel 6). Die praktische Anwendbarkeit verdeutlichen wir anhand von drei Fallstudien (Kapitel 7). Abschließend fassen wir in Kapitel 8 die Hauptresultate dieser Arbeit zusammen, diskutieren sie hinsichtlich der zu Beginn dieses Kapitels gestellten Forschungsfragen und geben einen Ausblick auf neue, sich aus unserer Arbeit ergebene Forschungsfragen.

# 2 Grundlagen

In diesem Kapitel geben wir eine kurze Einführung in die grundlegenden Konzepte, auf denen unser Ansatz zur automatisierten Controller-Synthese für Services mit Daten beruht. Wir starten zunächst mit einer kurzen Wiederholung der Hauptideen *Service-orientierter Architekturen (SOAs)* in Abschnitt 2.1. Wie bereits in Kapitel 1 erwähnt, basiert unser Ansatz auf *algebraischen Petrinetzen (APNs)* und der temporalen Logik *CTL*. Die Grundlagen dafür werden in den Abschnitten 2.2 und 2.3 eingeführt. In Abschnitt 2.4 stellen wir unser laufendes Beispiel vor, anhand dessen wir unseren Ansatz in späteren Kapiteln erläutern werden. Wir fassen das Kapitel in Abschnitt 2.5 abschließend zusammen.

## 2.1 Serviceorientierte Architekturen

Eine *serviceorientierte Architektur* (*SOA*) ist ein Paradigma, das eine Softwarearchitektur in einer abstrakten Weise definiert. Die Grundidee ist es, eine komplexe Funktionalität zu erreichen, indem mehrere Komponenten zusammenarbeiten. Diese Komponenten, genannt *Services*, stellen eigene, weniger komplexe Funktionalitäten zur Verfügung und können über Interfaces mit anderen Services kommunizieren. Durch die Komposition verschiedener Services zur Bearbeitung einer bestimmten Aufgabe, wird eine große Wartbar- und Wiederverwendbarkeit erreicht. In Abschnitt 2.1.1 stellen wir kurz die Hauptmerkmale einer SOA und ihre Vorteile vor.

Die abstrakte Sichtweise ermöglicht eine Anwendung der serviceorientierten Architekturen in verschiedenen praktischen und theoretischen Bereichen. Insbesondere in sicherheitskritischen Domänen, wie der Medizintechnik ist es notwendig, Zuverlässigkeit und Sicherheit zu gewährleisten. In diesen Bereichen ist die Sicherstellung der Interoperabilität zwischen verschiedenen Komponenten ebenso wünschenswert wie schwierig. Die serviceorientierten Architekturen weisen hierbei großes Potential auf, den Schwierigkeitsgrad zu reduzieren.

Um dies zu verdeutlichen, diskutieren wir in Abschnitt 2.1.2 den Zusammenhang zwischen serviceorientierten Architekturen und medizinischen Geräten. Dabei gehen wir vorallem darauf ein, dass das Problem der Gewährleis-

tung einer sicheren Interoperabilität medizinischer Geräte, auf das Problem der Servicekomposition zurückgeführt werden kann.

### 2.1.1 Hauptmerkmale

Obwohl es in der Literatur keine eindeutige Definition von SOAs gibt, gleichen sich existierende Erklärungen, wie sie beispielsweise in [CHT03, PVDH07, Mel10] gegeben werden, in den folgenden Hauptmerkmalen.

Diesen Erklärungen folgend ist eine SOA eine abstrakte Softwarearchitektur bei der Funktionalitäten, wie Methoden oder Anwendungen, in sogenannten Services gekapselt werden. Diese können wiederverwendet werden und sind von Nutzern zugreifbar. Dazu können sie in einer Sammlung von Services, dem sogenannten Service-Vermittler, gesucht und gefunden werden. Außerdem sind sie untereinander komponierbar, sodass daraus ein neuer Service mit einer komplexeren Funktionalität resultiert. Damit ist eine nutzerbezogene Anpassung der Funktionalität möglich.

Wie in Abbildung 2.1 dargestellt ist, existieren drei Hauptrollen in einer SOA.

**Service-Anbieter** Jeder Service wird durch einen Service-Anbieter bereitgestellt. Der Anbieter implementiert die gewünschte Funktionalität eines Sevices und definiert das Interface. Dazu werden die erforderlichen Eingabe- und Ausgabeparamater beschrieben. Die Definition des Interfaces folgt dabei einem standardisierten Protokoll, was eine Veröffentlichung bei einem Service-Vermittler ermöglicht.

**Service-Vermittler** Bei dem Service-Vermittler werden Interface-Definitionen gespeichert. Service-Nutzer können bei diesem nach passenden Services suchen. Sollte diese Suche erfolgreich sein, sendet der Service-Vermittler einen Verweis auf den Service-Anbieter an den Service-Nutzer.

**Service-Nutzer** Der Service-Nutzer ist der Nutzer des Services. Dies kann zum Beispiel eine reale Person oder ein anderer Service sein. Der Service-Nutzer kann bei dem Service-Vermittler nach einem passenden Service suchen. Im positiven Fall erhält der Service-Nutzer einen Verweis auf den Service-Anbieter von dem Service-Vermittler. Mittels dieses Verweises kann der Service-Nutzer den Service-Anbieter kontaktieren und eine Beschreibung der Funktionalität des Services erfragen. Sollte diese mit der gewünschten Funktionalität übereinstimmen, so kann der Service-Nutzer auf den Service aus der Ferne zugreifen.

Wie bereits erwähnt, kann ein Service-Nutzer verschiedene Services komponieren, um eine gewünschte Funktionalität zu erreichen. Dazu stehen zwei Möglichkeiten zur Verfügung. Zum einen kann bei der Suche direkt darauf geachtet werden, dass Interfaces kompatibel sind und gegebene Constraints, wie zum Beispiel eine geeignete Antwortzeit, erfüllt werden. Dies beschränkt jedoch die Anzahl potentiell nutzbarer Services und kann dazu führen, dass

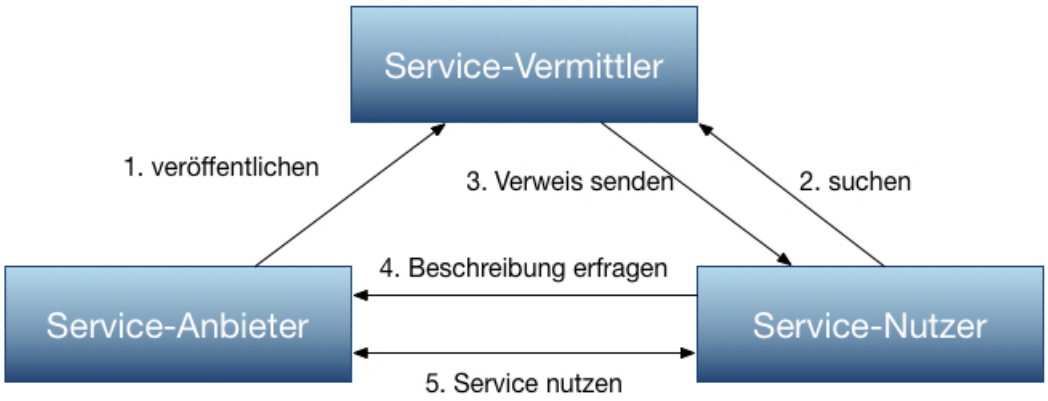



**Abbildung 2.1:** Rollen einer SOA und ihre Zusammenhänge

kein geeigneter Service gefunden werden kann. Zum anderen können Services deren Funktionalität passend ist, die jedoch inkompatible Interfaces besitzen, mit Hilfe einer zentralen Komponente, dem Controller, miteinander komponiert werden. Der Controller interagiert dabei mit allen zu komponierenden Services und steuert die Kommunikation und das Verhalten des komponierten Systems.

Sowohl die Services als auch ihre Komposition haben neben anderen die folgenden Merkmale[PVDH07, Mel10]:

**Abstraktion** Services erlauben nur auf ihre Interfaces und Funktionsbeschreibung einen öffentlichen Zugriff. Das interne Verhalten wird hingegen verschleiert.

**Wiederverwendbarkeit** Funktionalität sollte so weit wie möglich in Services gekapselt werden, um die Wiederverwendbarkeit zu erhöhen.

**Standardisierung** Die Kommunikation zwischen den Services oder der Zugriff auf ihre Interfaces ist standardisiert. Dies wird bei der Implementierung einer SOA durch die Definition eines Protokolls erreicht. Dadurch wird die Nutzung von Services plattformunabhängig und sie können so beliebigen Service-Nutzern zur Verfügung gestellt werden.

**Lose Kopplung** Services können bei dem Service-Vermittler zu jeder Zeit gesucht und dynamisch komponiert werden.

**Komposition** Mit Hilfe ihrer Interfaces können Services miteinander komponiert werden. Nur selten funktioniert diese Komposition direkt, d.h. durch das Verschmelzen der Interfaces. Oft treten hingegen Inkompatibilitäten auf, die die direkte Komposition verhindern. In diesen Fällen werden Controller genutzt. Aktuell bedarf die Synthese korrekter Controller für Services mit Daten jedoch sehr hohen manuellen Aufwand und ist somit fehleranfällig, sowie zeit- und kostenintensiv.

### 2.1.2 Beziehung zwischen SOAn und Geräten

Die Sicherstellung der Interoperabilität zwischen Geräten ist in der medizinischen Domäne eine sehr kritische Aufgabe. Auf der einen Seite werden die Geräte von verschiedenen Herstellern entwickelt und unterstützen nur in äußerst seltenen Fällen die Kommunikation mit Geräten anderer Hersteller. Auf der anderen Seite existieren auch Probleme, die aus den europäischen Richtlinien für Medizinprodukte [Eur07] und der deutschen Umsetzung, dem Medizinproduktegesetz [Ger11] resultieren. Die Richtlinien wurden vom europäischen Parlament entwickelt, um europaweit eine sichere und korrekte Funktionalität von medizinischen Geräten zu gewährleisten. Dazu muss jeder Staat in der EU diese Richtlinien in eigenen Gesetzen umsetzen und die Sicherstellung garantieren. In Deutschland wurde dafür das Medizinproduktegesetz [Ger11] verabschiedet, das Hersteller medizinischer Geräte dazu zwingt, eine vollständige Qualitätssicherung durchzuführen und alle Geräte einem Zertifizierungsprozess zu unterziehen. Ein entscheidendes Problem dabei ist, dass die Kombination aus zwei (zertifizierten) Geräten als neues Gerät verstanden wird, sodass für dieses eine neue Zertifizierung durchgeführt werden muss. Aktuell werden die Geräte während des Zertifizierungsprozesses ausgiebig getestet, was sehr zeitaufwändig ist und nicht garantiert, dass keine Fehler vorhanden sind. Eine Vision ist daher, formale Methoden, mit denen es möglich ist, die Korrektheit der Funktionalität zu verifizieren, zu nutzen um den Zertifizierungsprozess zu vereinfachen und sicherer zu machen.

An dieser Stelle kann die Anwendung serviceorientierter Architekturen sehr hilfreich sein. Insbesondere deren Kompositionskriterium ermöglicht eine parallele Verifikation weniger komplexer Komponenten und ihrer Interaktion. Dies reduziert den Berechnungsaufwand der Verifikation des komponierten Systems. Die Anwendung des Paradigmas wird insbesondere in der medizinischen Domäne ermöglicht, da jedes Gerät eine im inneren versteckte Funktionalität zur Verfügung stellt, die mit anderen durch das Interface kombiniert werden kann. Durch die Komposition verschiedener Geräte entsteht im Sinne des Medizinproduktegesetzes ein neues Gerät mit komplexerer Funktionalität.

Vergleicht man nun diese Punkte mit den Hauptcharakteristika einer SOA, so liegt es nahe, medizinische Geräte als Services und die Sicherstellung ihrer Interoperabilität als Service-Komposition aufzufassen. Folgt man dieser Idee, ist es notwendig, einen korrekten Controller zu synthetisieren, der mit allen involvierten Services kommunizieren kann und dabei im Vorfeld spezifizierte Verhaltensanforderungen garantiert. Mit Hilfe dieses Controllers kann der Weg zur Herstellerunabhängigkeit geebnet werden.

## 2.2 Algebraische Petrinetze

Da wir serviceorientierte Architekturen als grundlegendes Konzept nutzen und dazu medizinische Geräte als Services auffassen, können wir die Herausforderung eine sichere Interoperabilität zu gewährleisten, auf das Problem der

Service-Komposition zurückführen. Dies kann mit Hilfe formaler Methoden angegangen werden. Dazu benötigen wir zunächst einen wohldefinierten Formalismus, der es uns ermöglicht, Service-Verhalten adäquat zu repräsentieren und zu analysieren.

Der gesuchte Formalismus muss, präziser gesagt, in der Lage sein, sowohl den Kontrollfluss als auch den Datenfluss darzustellen. Damit wird erreicht, dass sowohl das Service-Verhalten als auch der Datenaustausch und die Datenmodifikation simultan repräsentiert werden können. Wir möchten jedoch darauf verzichten, das Verhalten eines Services für jeden möglichen Datentyp und alle potentiell aufrufbaren Operationen zur Datenmodifikation zu betrachten. Wir verlangen also von dem Formalismus, dass Datentypen, Operationen und Datenabhängigkeiten auf einer syntaktischen Ebene abgebildet werden können.

In Abschnitt 2.2.1 stellen wir daher zunächst Petrinetze vor. Dabei handelt es sich um einen wohldefinierten Formalismus, der häufig zur Modellierung des Verhaltens verteilter Systeme genutzt wird. Die Semantik, die durch kausale Abhängigkeiten zwischen Markierungen definiert ist, ermöglicht Analysen und Argumentationen über verschiedene Zustände des Systems. Somit sind Petrinetze für die Modellierung des Kontrollflusses sehr geeignet. Jedoch wird der Datenfluss, also der Austausch von Daten zwischen den Zuständen des Systems, in Petrinetzen nicht explizit dargestellt. Um dieses Problem zu überwinden, können algebraische Petrinetze genutzt werden. Mit ihnen ist es möglich, datenabhängiges Verhalten und Datenmodifikationen auf einer syntaktischen Ebene zu modellieren. Informell betrachtet, handelt es sich dabei um die oben genannten Petrinetze, die mit einer algebraischen Spezifikation angereichert sind, die wir in Abschnitt 2.2.2 einführen. In Abschnitt 2.2.3 geben wir einen Überblick über die Syntax und Semantik von algebraischen Petrinetzen.

## 2.2.1 Petrinetze

Petrinetze in ihrer einfachsten Form, auch Platz-Transitions-Netze oder Low-Level-Netze genannt, wurden von Carl Adam Petri [Pet62] entwickelt. Im Allgemeinen bilden sie einen ausdrucksstarken Formalismus, mit dem diskrete und oft verteilte Systeme modelliert werden können. Genutzt werden sie in verschiedenen Bereichen, wie zum Beispiel zur Modellierung eingebetteter Systeme oder von Netzwerken, aber auch bei der Geschäftsprozessmodellierung. In diesem Abschnitt fassen wir die Hauptcharakteristika der Low-Level-Netze basierend auf [Abe90, Rei10] zusammen.

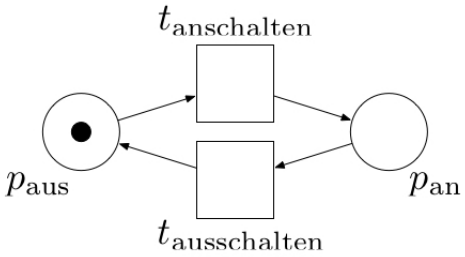
### Syntax

Wir beginnen mit einem kleinen Beispiel, das die Modellierung eines Systems mit Hilfe von low-level Petrinetzen veranschaulicht.

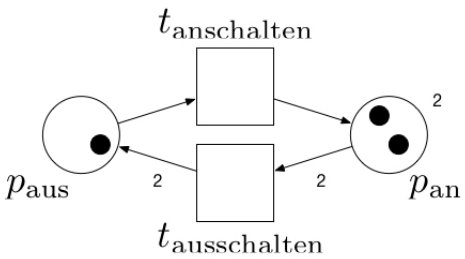
**Beispiel 2.1.** (Lichtschalter)

In diesem Beispiel stellen wir uns das Verhalten eines *Lichtschalters* vor. Dieser kann sich in zwei Zuständen befinden, *an* oder *aus*. Jeder dieser Zustände kann durch eine Aktion, nämlich *anschalten* oder *ausschalten*, eintreten. Dieses Verhalten kann mit Hilfe von low-level Petrinetzen leicht modelliert werden. Dazu werden Zustände, also passive Elemente, als Plätze (Kreise) dargestellt. Die Aktionen, also die aktiven Elemente, werden durch Transitionen (Rechtecke) repräsentiert. Zu Beginn soll der Lichtschalter ausgeschaltet sein. Dies wird durch einen schwarzen Punkt auf dem Platz $p_{\text{aus}}$ dargestellt. Dieser Punkt wird *Marke* genannt. Der dargestellte Zustand des Systems zu Beginn, also dass der Lichtschalter nicht an, sondern und aus ist, wird *initiale Markierung* genannt.

$N_1$ :

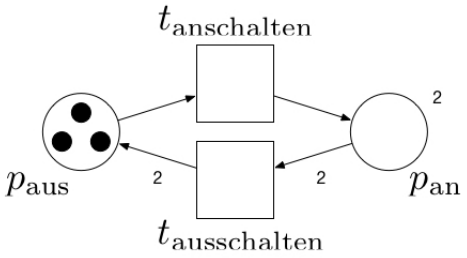



Wenn wir statt einem einfachen Lichtschalter nun eine Steuerung für drei LEDs betrachten, so ist es nicht notwendig, das in $N_1$ dargestellte System drei mal zu zeichnen. Stattdessen können wir einfach drei Marken benutzen. Dies ist in Netz $N_2$ dargestellt. Nehmen wir an, dass in dem System zu Beginn zwei LEDs *an* sind und eine *aus* ist. Außerdem wollen wir fordern, dass immer nur maximal zwei LEDs eingeschaltet sein dürfen. Letzteres kann durch eine Beschriftung mit der Zahl „2“ an dem Platz $p_{an}$ gewährleistet werden. Diese Zahl wird *Kapazität* genannt. Sollte ein Platz mit keiner Zahl beschriftet sein, so wird automatisch eine Kapazität von $\infty$ angenommen. Wollen wir modellieren, dass die Aktion *ausschalten* immer zwei LEDs ausschalten soll, so beschriften wir die Pfeile an den Kanten $(p_{\text{an}}, t_{\text{ausschalten}})$ und $(t_{\text{ausschalten}}, p_{\text{aus}})$ mit der Zahl „2“. Dies wird *Gewicht* genannt. Sollten Kanten nicht mit einer Zahl beschriftet sein, so wird automatisch ein Gewicht von 1 angenommen.
$N_2$ :



Das Gewicht einer eingehenden Kante einer Transition führt zu einer Reduktion der Anzahl der Marken, die sich auf dem Platz befinden von dem die Kante ausgeht, um das Gewicht. Die Anzahl der Marken auf dem Platz in dem eine Kante mündet, erhöht sich hingegen um das Gewicht der Kante. Diese Änderungen der Markenanzahl auf den Plätzen resultiert aus dem sogenannten *Schalten einer Transition* Dieses Schalten einer Transition führt zu einem neuen Zustand des Netzes, der sogenannten *Folgemarkierung*. Beispielsweise führt das Schalten der Transition $t_{\text{ausschalten}}$ im Netz $N_2$ zur Folgemarkierung, die in Netz $N_2'$ dargestellt ist.

$N_2'$ :




Neben der im Beispiel genutzten graphischen Darstellung eines Petrinetzes, existiert auch eine mengentheoretische Repräsentation, die wie folgt definiert ist.

**Definition 2.1** (Petrinetz)**.**
*Ein* ***Petrinetz*** *ist ein 6-Tupel* $N = (P, T, F, C, W, M_0)$ *wobei gilt:*

- $P$ *ist eine nicht-leere, endliche Menge, deren Elemente* ***Plätze*** *genannt werden.*
- $T$ *ist eine nicht-leere, endliche Menge, deren Elemente* ***Transitionen*** *genannt werden.*
- $F \subseteq (P \times T) \cup (T \times P)$ *ist eine* ***Flussrelation*** *zwischen der Menge der Plätze und der Menge der Transitionen und umgekehrt. Ihre Elemente werden* ***Kanten*** *genannt.*
- $C : P \to \mathbb{N}^+ \cup \infty$ *ist eine Abbildung, die jedem Platz eine* ***Kapazität*** *zuweist.*
- $W : F \to \mathbb{N}^+ \cup \infty$ *ist eine Abbildung, die jeder Kante ein* ***Gewicht*** *zuweist.*
- $M_0 : P \to \mathbb{N}$ *ist eine Abbildung, genannt* ***initiale Markierung****, die jedem Platz eine Anzahl von Marken im initialen Zustand des Petrinetzes zuweist.*

**Beispiel 2.2.** (Lichtschalter - Fortsetzung)

Das Petrinetz aus $N_2$ aus Beispiel 2.1 kann nach Definition 2.1 formal wie folgt dargestellt werden.

$$
\begin{aligned}
P &= \{p_{\text{an}}, p_{\text{aus}}\} \\
T &= \{t_{\text{ausschalten}}, t_{\text{anschalten}}\} \\
F &= \{(p_{\text{an}}, t_{\text{ausschalten}}), (t_{\text{ausschalten}}, p_{\text{aus}}), (p_{\text{aus}}, t_{\text{anschalten}}), (t_{\text{anschalten}}, p_{\text{an}})\} \\
C &= \{(p_{\text{an}}, 2), (p_{\text{aus}}, \infty)\} \\
W &= \{((p_{\text{an}}, t_{\text{ausschalten}}), 2), ((t_{\text{ausschalten}}, p_{\text{aus}}), 2), \\
&\quad ((p_{\text{aus}}, t_{\text{anschalten}}), 1), ((t_{\text{anschalten}}, p_{\text{an}}), 1)\} \\
M_0 &= \{(p_{\text{an}}, 2), (p_{\text{aus}}, 1)\}
\end{aligned}
$$

### Ausführungssemantik

Nach der Einführung der Syntax eines Petrinetzes, widmen wir uns nun der Ausführungssemantik. Diese definiert die Dynamik des Systems und somit des Petrinetzes. Grob gesagt ist die Ausführungssemantik eines Petrinetzes durch den Fluss der Marken in ihm charakterisiert. In unserem Beispiel (Beispiel 2.1) ist bereits ein Schritt dieses Flusses dargestellt, nämlich der Übergang von $N_2$ zu $N_2'$. Genauer gesagt, gelangen wir von der initialen Markierung ($M_0 = \{(p_{\text{an}}, 2), (p_{\text{aus}}, 1)\}$) durch das Schalten der Transition $t_{\text{ausschalten}}$ zu der Folgemarkierung $M_1 = \{(p_{\text{an}}, 0), (p_{\text{aus}}, 3)\}$

Um die mathematischen Grundlagen der Ausfürungssemantik zu erläutern, benötigen wir die folgenden Definitionen.

**Definition 2.2** (Vorbereich und Nachbereich)**.**
*Sei $N = (P, T, F, C, W, M_0)$ ein Petrinetz. Dann gilt für alle $x \in T \cup P$ :*

1. *Der **Vorbereich** von $x$ ist definiert als*

$$\bullet x =_{def} \{y \in P \cup T | (y, x) \in F\}.$$

2. *Der **Nachbereich** von $x$ ist definiert als*

$$x\bullet =_{def} \{y \in P \cup T | (x, y) \in F\}.$$

*Wenn $x \in T$ gilt, dann werden die Elemente $\bullet x$ **Eingangsplätze** der Transition $x$ und die Elemente von $x\bullet$ **Ausgangsplätze** der Transition $x$ genannt.*

**Beispiel 2.3.** (Lichtschalter - Fortsetzung)

In Beispiel 2.1 sind die Eingangs- und Ausgangsplätze von $t_{\text{anschalten}}$:

$\bullet t_{\text{anschalten}} = \{p_{\text{aus}}\}$

$t_{\text{anschalten}}\bullet = \{p_{\text{an}}\}$

**Definition 2.3** (Aktive Transition)**.**
*Sei $N = (P, T, F, C, W, M_0)$ ein Petrinetz. Eine Transition $t \in T$ ist **aktiviert bezüglich einer Markierung** $M : P \to \mathbb{N}$ gdw.*

1. $\forall p \in \bullet t.\ W((p,t)) \leq M(p)$ *und*

2. $\forall p \in t\bullet.\ W((t,p)) + M(p) \leq C(p)$.

**Beispiel 2.4.** (Lichtschalter - Fortsetzung)

In Beispiel 2.1 haben wir bereits das Petrinetz $N_2$ mit der initialen Markierung $M_0 = \{(p_{\text{an}}, 2), (p_{\text{aus}}, 1)\}$ eingeführt. In diesem Petrinetz ist Transition $t_{\text{ausschalten}}$ bezüglich der initialen Markierung aktiviert, da folgendes gilt.

1. $W((p_{\text{an}}, t_{\text{ausschalten}})) = 2 \leq 2 = M_0(p_{\text{an}})$
2. $W((t_{\text{ausschalten}}, p_{\text{aus}})) + M_0(p_{\text{aus}}) = 2 + 2 \leq \infty = C(p_{\text{aus}})$

Im Gegensatz dazu ist die Transition $t_{\text{anschalten}}$ nicht bezüglich der initialen Markierung aktiviert, da gilt:

1. $W((p_{\text{aus}}, t_{\text{anschalten}})) = 1 \leq 1 = M_0(p_{\text{aus}})$
2. $W((t_{\text{anschalten}}, p_{\text{an}})) + M_0(p_{\text{an}}) = 1 + 2 \not\leq 2 = C(p_{\text{an}})$

**Definition 2.4** (Schalten einer Transition)**.**
*Sei $N = (P, T, F, C, W, M_0)$ ein Petrinetz. Das **Schalten einer Transition** $t \in T$, die bezüglich einer Markierung $M$ aktiviert ist, führt zu einer Folgemarkierung $M'$, so dass für alle Plätze gilt:*

$$M'(p) =_{def} \begin{cases} M(p) - W((p,t)) & ,wenn\ p \in \bullet t \setminus t\bullet \\ M(p) + W((t,p)) & ,wenn\ p \in t\bullet \setminus \bullet t \\ M(p) - W((p,t)) + W((t,p)) & ,wenn\ p \in \bullet t \cap t\bullet \end{cases}$$

*Das Schalten einer Transition schreiben wir kurz als $M \xrightarrow{t} M'$.*

**Bemerkung 2.5.**
*Wenn eine Transition bezüglich einer Markierung aktiviert ist, kann sie schalten. Sollten mehrere Transitionen bezüglich der selben Markierung aktiviert sein, kann eine beliebige von ihnen schalten.*

**Definition 2.6** (Schaltsequenz)**.**
*Für ein Petrinetz $N = (P, T, F, C, W, M_0)$ mit einer initialen Markierung $M_0$,*

*wird jede Abfolge von nacheinander schaltenden Transitionen beginnend von* $M_0$ ***Schaltsequenz von N*** *genannt. Sie wird mit* $\sigma = M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots$ *bezeichnet.*

*Eine* ***Schaltsequenz von N beginnend bei der Markierung M*** *wird bezeichnet mit*

$$\sigma_M = M \xrightarrow{t_1} M' \xrightarrow{t_2} \ldots$$

*Die* ***Menge der Schaltsequenzen von N*** *(beginnend von* $M_0$*) werden durch* $\sigma^N$ *repräsentiert.*

**Definition 2.7** (Erreichbarkeit einer Markierung)**.**
*Sei* $N = (P, T, F, C, W, M_0)$ *ein Petrinetz.*

*Eine Markierung* $M'$ *in* $N$ *ist* ***in einem Schritt erreichbar von einer Markierung M****, wenn gilt:*

$$\exists t \in T.\ M \xrightarrow{t} M'.$$

*Eine Markierung* $M'$ *ist in einer beliebigen Anzahl von Schritten* ***erreichbar von einer Markierung M****, falls eine endliche Schaltsequenz von* $N$ *beginnend von* $M$ *existiert, geschrieben als* $M \xrightarrow{*} M'$ *, sodass*

$$\exists \sigma_M.\ \sigma_M = M \xrightarrow{t_1} \ldots \xrightarrow{t_n} M', n \in \mathbb{N} \setminus \{0\}.$$

**Definition 2.8** (Erreichbarkeitsmenge)**.**
*Sei* $N = (P, T, F, C, W, M_0)$ *ein Petrinetz. Die Menge*

$$R_N(M_0) = \{M | M_0 \xrightarrow{*} M\},$$

*die alle in Netz* $N$ *von* $M_0$ *aus erreichbaren Markierungen enthält, wird* ***Erreichbarkeitsmenge*** *von* $N$ *bezüglich der initialen Markierung* $M_0$ *genannt.*

**Definition 2.9** (Kantenbeschrifteter Gerichteter Graph mit Initialem Knoten)**.**
*Sei* $A$ *eine endliche Menge. Ein über* $A$ ***kantenbeschrifteter gerichteter Graph mit initialem Knoten*** *ist ein 4-Tupel* $\mathcal{G} = (V, Ed, v_0, l)$ *bestehend aus*

1. *einer endlichen* ***Menge von Knoten*** $V$*,*
2. *einer endlichen* ***Menge von Kanten*** $Ed \subseteq V \times V$*,*
3. *einem* ***initialen Knoten*** $v_0 \in V$*, und*
4. *einer* ***Beschriftungsfunktion*** $l : E \to A$*.*

**Definition 2.10** (Erreichbarkeitsgraph)**.**
*Sei* $N = (P, T, F, C, W, M_0)$ *ein Petrinetz und sei* $R_N(M_0) = \{M | M_0 \xrightarrow{*} M\}$ *die Erreichbarkeitsmenge von* $N$*. Der* ***Erreichbarkeitsgraph*** *für* $N$ *ist ein*

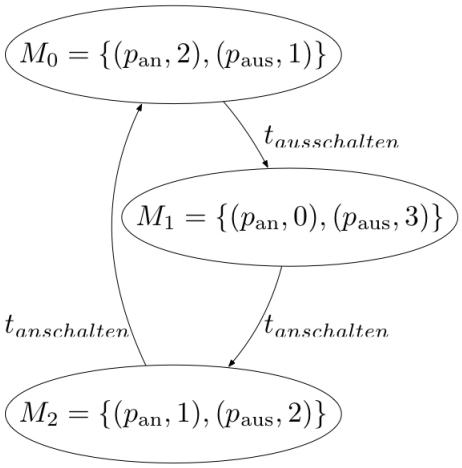
*über $T$ kantenbeschrifteter gerichteter Graph mit initialem Knoten $M_0$. Er ist wie folgt definiert.*

$$RG_N(M_0) = \{R_N(M_0), Ed, M_0, l\} \text{ mit}$$
$$\forall M \in R_N(M_0).\ (M \xrightarrow{t} M' \implies ((M, M') \in Ed \wedge l((M, M')) = t))$$

**Beispiel 2.5.** (Lichtschalter - Fortsetzung)

Betrachten wir das Petrinetz $N_2$ aus Beispiel 2.1. Der Erreichbarkeitsgraph ($RG_{N_2}(M_0)$) dieses Petrinetzes mit initialer Markierung $M_0 = \{(p_{\text{an}}, 2), (p_{\text{aus}}, 1)\}$ ist:

$RG_{N_2}(M_0)$ :




Der Erreichbarkeitsgraph eines Petrinetzes repräsentiert dessen Ausführungssemantik.

### Offene Petrinetze

Um die Kommunikation mit der Umgebung zu ermöglichen und den Ansprüchen der serviceorientierten Architekturen gerecht zu werden, betrachten wir eine spezielle Art der Petrinetze, die sogenannten *offenen Petrinetze*. Jedes offene Petrinetz stellt, vereinfacht gesagt, ein Interface zur Verfügung, durch das die Kommunikation unterstützt wird. Bei einem solchen Interface handelt es sich um eine Menge spezieller Plätze.

Im folgenden werden offene Petrinetze basierend auf den Ausführungen von Karsten Wolf [Wol09] definiert.

**Definition 2.11** (Offenes Petrinetz)**.**
*Sei $N = (P, T, F, C, W, M_0)$ ein Petrinetz. Ein* ***offenes Petrinetz*** *ist ein Tupel $ON = (N, P_i, P_o, M_\Omega)$ wobei*

- $P_i \subseteq P$ *eine Menge von* ***Eingangskanälen****,*
- $P_o \subseteq P$ *eine Menge von* ***Ausgangskanälen*** *und*
- $M_\Omega$ *eine Menge von* ***Endmarkierungen*** *ist.*

*Offene Petrinetze erfüllen die folgenden Bedingungen.*

1. *Für alle $p \in P_i$ ist der Vorbereich von $p$ leer.*
2. *Für alle $p \in P_o$ ist der Nachbereich von $p$ leer.*
3. *Für alle $M \in M_\Omega$ ist keine Transition bezüglich $M$ aktiviert.*
4. *Für alle $M \in (M_\Omega \cup \{M_0\})$ enthalten sowohl Eingangs- als auch Ausgangskanäle keine Marken.*
5. *Jede Transition $t \in T$ ist mit maximal einem Eingangs- oder Ausgangskanal über eine Kante verbunden.*

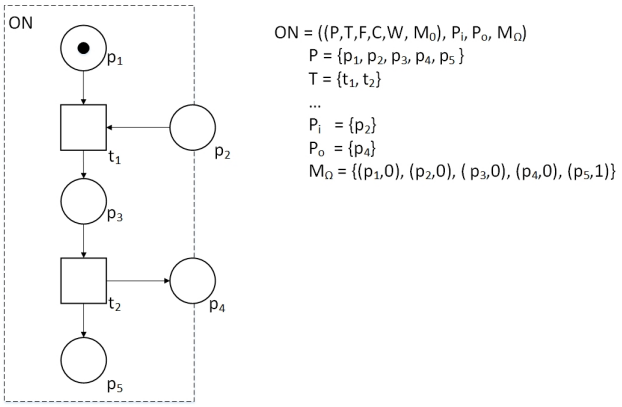
*Eingangskanäle eines offenen Petrinetzes $ON$ werden oft als* ***Interface-Eingangsplätze*** *von $ON$ bezeichnet. Dementsprechend werden Ausgangskanäle oft* ***Interface-Ausgangsplätze*** *genannt.*

*Die Menge von Eingangs- und Ausgangskanälen ($P_i \cup P_o$) wird als* ***Menge von Interface-Plätzen*** *von $ON$ bezeichnet.*

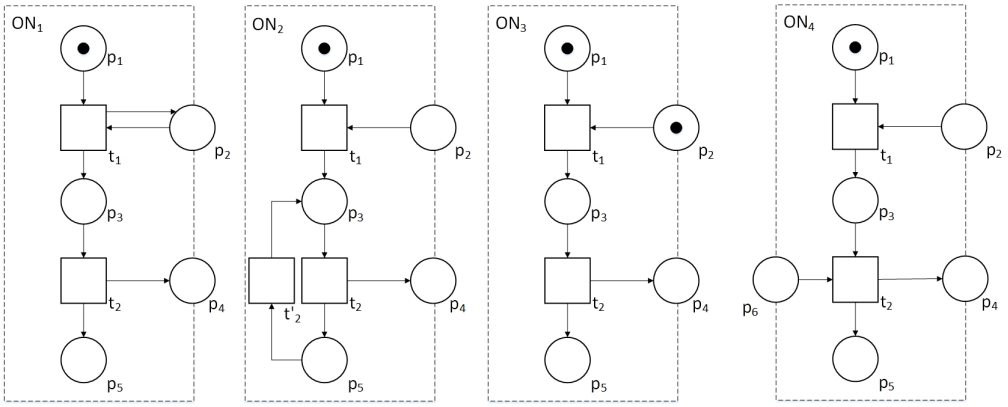
Um eine bessere Intuition dafür zu bekommen, betrachten wir das folgende Beispiel.

**Beispiel 2.6.** (Offenes Petrinetz)

In der folgenden Abbildung wird ein offenes Petrinetz visualisiert. Dieses Netz empfängt über den Interface-Eingangsplatz $p_2$ eine Nachricht und sendet daraufhin eine Nachricht über den Interface-Ausgangsplatz $p_4$. Der Service, dessen Verhalten durch das angegebene Netz modelliert wird, terminiert, wenn die Endmarkierung erreicht wird. Die Endmarkierung $M_\Omega$ wird erreicht, wenn nur $p_5$ eine Marke enthält. Dies bedeutet, dass die Nachricht, die über $p_4$ gesendet wurde, von der Umgebung irgendwann entgegengenommen wird.




In diesem Netz gelten die oben angegebenen Bedingungen. Im folgenden betrachten wir unzulässige Netze, die mindestens gegen eine der Bedingungen verstoßen.




$ON_1$ widerspricht den Bedingungen 1 und 2, da Platz $p_2$ sowohl als Interface-Eingangs- als auch als Interface-Ausgangsplatz genutzt wird. In der mengentheoretischen Darstellung gilt für $p_2 \in P_i$ und $p_2 \in P_o$:

$$F = \{\ldots, (t_1, p_2), (p_2, t_1), \ldots\} \cap \{(t_1, p_2)\} = \{(t_1, p_2)\} \neq \emptyset \text{ und}$$
$$F = \{\ldots, (t_1, p_2), (p_2, t_1), \ldots\} \cap \{(p_2, t_1)\} = \{(p_2, t_1)\} \neq \emptyset$$

$ON_2$ widerspricht der dritten Bedingung, da in der Endmarkierung, in der nur $p_5$ eine Marke enthalten soll, die Transition $t_2'$ aktiviert ist. Aus diesem Grund würde das Netz nicht terminieren, auch wenn $p_5$ eine Marke enthalten sollte.

$ON_3$ widerspricht der vierten Bedingung, da ein Interface-Platz initial markiert ist. Damit würde eine Pseudo-Kommunikation modelliert werden ohne dass ein Kommunikationspartner bekannt ist.

$ON_4$ widerspricht Bedingung 5, da $t_2$ mit zwei Interface-Plätzen ($p_6$ und $p_4$) verbunden ist. Dies wird auch in der mengentheoretischen Darstellung deutlich.

Für $(\bullet t_2 \cup t_2 \bullet) = \{p_3, p_6\} \cup \{p_4, p_5\}$ und $(P_i \cup P_o) = \{p_2, p_6\} \cup \{p_4\}$ gilt:

$$|(\bullet t_2 \cup t_2 \bullet) \cap (P_i \cup P_o)| = |\{p_3, p_4, p_5, p_6\} \cap \{p_2, p_4, p_6\}| = |\{p_4, p_6\}| > 1$$

Offene Petrinetze können mit Hilfe ihrer Interfaces kombiniert werden. Ihre Komposition ist wie folgt definiert.

**Definition 2.12** (Komposition Offener Petrinetze)**.**
*Seien* $ON_1 = ((P_1, T_1, F_1, C_1, W_1, M_0^{ON_1}), P_{i1}, P_{o1}, M_\Omega^{ON_1})$ *und* $ON_2 = ((P_2, T_2, F_2, C_2, W_2, M_0^{ON_2}), P_{i2}, P_{o2}, M_\Omega^{ON_2})$ *zwei offenen Petrinetze.* $ON_1$ *und* $ON_2$ *sind* ***komponierbar*** *gdw.*

- $P_1 \cap P_2 \subseteq (P_{i1} \cap P_{o2}) \cup (P_{i2} \cap P_{o1})$*, und*
- $P_{i1} \cap P_{i2} = \emptyset$*, und*
- $P_{o1} \cap P_{o2} = \emptyset$*, und*
- $T_1 \cap T_2 = \emptyset$*.*

*Die* ***Komposition*** *von zwei komponierbaren, offenen Petrinetzen ist ein offenes Petrinetz* $ON$*, dass wie folgt definiert ist.*
$ON = ON_1 \oplus ON_2 = ((P, T, F, C, W, M_0), P_i, P_o, M_\Omega)$ *mit*

- $P = P_1 \cup P_2$
- $T = T_1 \cup T_2$
- $F = F_1 \cup F_2$
- $\forall p \in P.\ C(p) = \begin{cases} C_1(p) & , \text{ wenn } p \in P_1 \setminus P_2 \\ C_2(p) & , \text{ wenn } p \in P_2 \setminus P_1 \\ min(C_1(p), C_2(p)) & , \text{ wenn } p \in P_1 \cap P_2 \end{cases}$
- $\forall f \in F.\ W(f) = \begin{cases} W_1(f) & , \text{ wenn } f \in F_1 \setminus F_2 \\ W_2(f) & , \text{ wenn } f \in F_2 \setminus F_1 \end{cases}$
- $P_i = (P_{i1} \cup P_{i2}) \setminus ((P_{i1} \cap P_{o2}) \cup (P_{i2} \cap P_{o1}))$,
- $P_o = (P_{o1} \cup P_{o2}) \setminus ((P_{i1} \cap P_{o2}) \cup (P_{i2} \cap P_{o1}))$,
- $M_k = M_k^{ON_1} \oplus M_k^{ON_2}, k \in \{0, \Omega\}$*, wobei*
  $\forall p \in P.\ (M_k^{ON_1} \oplus M_k^{ON_2})(p) = \begin{cases} M_k^{ON_1}(p) & , \text{ if } p \in P_1 \\ M_k^{ON_2}(p) & , \text{ if } p \in P_2 \end{cases}$

**Lemma 2.13.**
*Seien* $ON_1 = ((P_1, T_1, F_1, C_1, W_1, M_0^{ON_1}), P_{i1}, P_{o1}, M_\Omega^{ON_1})$ *und*

*$ON_2 = ((P_2, T_2, F_2, C_2, W_2, M_0^{ON_2}), P_{i2}, P_{o2}, M_\Omega^{ON_2})$ zwei komponierbare, offene Petrinetze. Dann gilt für $k \in \{0, \Omega\}$ und für alle $p \in P_1 \cap P_2$:*

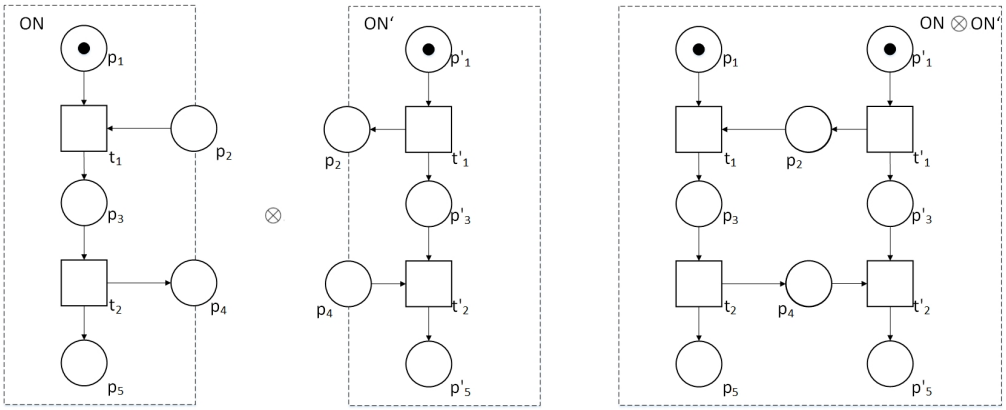
$$M_k^{ON_1}(p) = M_k^{ON_2}(p).$$

*Beweis.*
Sei $k \in \{0, \Omega\}$ und $p \in P_1 \cap P_2$. Dann gilt nach Definition 2.12 $P_1 \cap P_2 \subseteq (P_{i1} \cap P_{o2}) \cup (P_{i2} \cap P_{o1})$. Somit gilt $p \in (P_{i1} \cap P_{o2}) \cup (P_{i2} \cap P_{o1})$. Das bedeutet $p$ ist ein Interface-Platz. Laut Definition 2.11 (Bedingung 4) sind in jedem offenen Petrinetz die Interface-Plätze bezüglich der initialen und der Endmarkierung unmarkiert. Dies impliziert, dass $M_k^{ON_1}(p) = 0 = M_k^{ON_2}(p)$ gilt. □

Um die Komposition zu visualisieren, definieren wir im folgenden Beispiel einen Kommunikationspartner für das Netz $ON$, das wir in Beispiel 2.6 eingeführt haben.

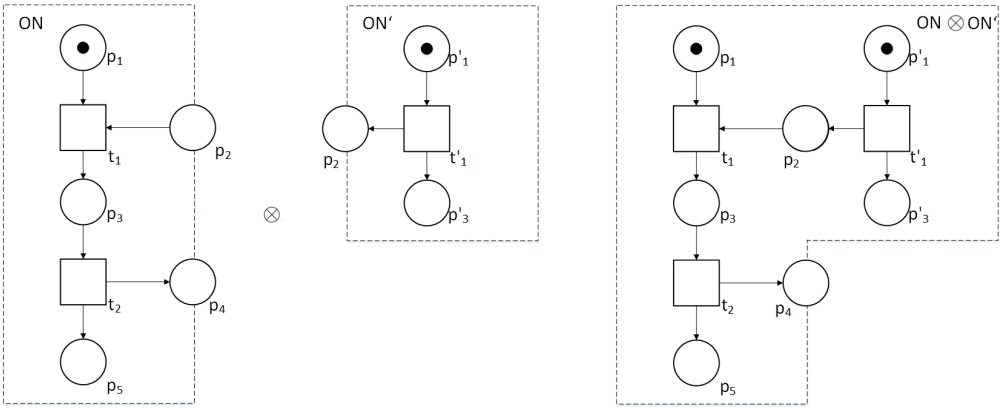
**Beispiel 2.7.** (Komposition offener Netze)
Die folgenden zwei Abbildungen stellen beispielhaft die Komposition von zwei offenen Netzen dar.

In der ersten Abbildung sind die Interface-Plätze kompatibel. Das bedeutet, dass für jeden Interface-Eingangs- bzw. -Ausgangsplatz des ersten Netzes ein passender Interface-Ausgangs- bzw. -Eingangsplatz im zweiten Netz existiert. Dadurch können die Interface-Plätze einfach verschmolzen werden. Dies resultiert in einem offenen Netz mit leerem Interface, in dem die komponierte Endmarkierung durch die gemeinsame Ausführung der beiden Netze erreicht wird. Hierbei ist die komponierte Endmarkierung die Markierung, bei der nur noch auf $p_5$ und $p_5'$ Marken liegen.




Obwohl eine solche Komposition sehr wünschenswert ist, ist es häufiger der Fall, dass die Interfaces zweier Netze nicht vollständig kompatibel sind. In der folgenden Abbildung ist ein Kommunikationspartner $ON'$ dargestellt, der nur ein teilweise kompatibles Interface besitzt. Daraus ergibt

sich eine Restmenge von Interface-Plätzen, die wie folgt berechnet wird (Def. 2.12). $P_o = (P_o^{ON} \cup P_o^{ON'}) \setminus ((P_i^{ON} \cap P_o^{ON'}) \cup (P_i^{ON'} \cap P_o^{ON})) = (\{p_4\} \cup \{p_2\}) \setminus ((\{p_2\} \cap \{p_2\}) \cup (\emptyset \cap p_4)) = p_4$.




### Zusammenfassung

In diesem Abschnitt haben wir die Grundlagen von low-level Petrinetzen zusammengefasst. Dazu haben wir zunächst die syntaktischen Elemente und ihre graphische sowie mengentheoretische Repräsentation vorgestellt. Außerdem haben wir die Ausführungssemantik, die sich aus den Schaltsequenzen des Petrinetzes ergibt, eingeführt. Wir sind anschließend zu den offenen Petrinetzen und deren Komposition übergegangen. Diese dienen uns als Grundlage für die Einführung der algebraischen Petrinetze in Abschnitt 2.2.3. Zunächst betrachten wir jedoch die dafür benötigten algebraischen Spezifikationen.

## 2.2.2 Algebraische Spezifikation

Im Folgenden stellen wir kurz die Grundlagen von algebraischen Spezifikationen vor, wie sie in [EMC+01, Rei91] zu finden sind.

Dazu betrachten wir zunächst einige fundamentale Definitionen.

**Definition 2.14** (Wörter)**.**
*Sei $\mathcal{A}$ eine nicht-leere Menge.*

1. *Ein (nichtleeres)* ***Wort*** *über $\mathcal{A}$ ist eine Folge $a_1 a_2 \dots a_n$ mit $n \in \mathbb{N}$ und $\forall i \in [1, n].\ a_i \in \mathcal{A}$.*
2. *Für $n = 0$, wird $w$* ***leeres Wort*** *über $\mathcal{A}$ genannt und mit $\lambda$ bezeichnet.*
3. *Für $n \in \mathbb{N}$ wird die* ***Länge eines Wortes*** *$w = a_1 a_2 \dots a_n$ mit $|w| = n$ bezeichnet.*

4. *Für $w_1 = a_1 a_2 \dots a_n$ und $w_2 = b_1 b_2 \dots b_m$ ist die **Komposition** von $w_1$ und $w_2$ definiert als $w_1 w_2 = a_1 a_2 \dots a_n b_1 b_2 \dots b_m$.*

5. *Die **Menge aller Wörter** über $\mathcal{A}$ wird mit $\mathcal{A}^*$ bezeichnet.*

**Definition 2.15** (Kartesisches Produkt)**.**
*Seien $A_1, \dots, A_n$ Mengen. Dann wird*

$$A_1 \times \dots \times A_n := \{(a_1, \dots, a_n) | \forall i \in \{1, \dots, n\}.\ a_i \in A_i\}$$

***kartesisches Produkt** über $A_1 \dots A_n$ genannt.*

**Definition 2.16** (Relation)**.**
*Seien $A_1, \dots, A_n$ nicht-leerre Mengen. Dann wird*

$$R \subseteq A_1 \times \dots \times A_n$$

***n-stellige Relation** über $A_1 \dots A_n$, oder kurz, **Relation** über $A_1 \dots A_n$ genannt.*

*$R$ heißt **homogen**, wenn $\forall i, j \in [1, n].\ A_i = A_j$ gilt.*

**Definition 2.17** (Äquivalenzrelation)**.**
*Sei $R \subseteq A \times A$ eine homogene 2-stellige Relation. $R$ wird **Äquivalenzrelation** über $A$ genannt, gdw.*

1. *$R$ **reflexiv** ist,*
   *also $\forall a \in A.\ (a, a) \in R$ gilt,*

2. *und $R$ **symmetrisch** ist,*
   *also $\forall a, b \in A.\ (a, b) \in R \Rightarrow (b, a) \in R$ gilt,*

3. *und $R$ **transitiv** ist,*
   *also $\forall a, b, c \in A.\ (a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$ gilt.*

**Definition 2.18** (Äquivalenzklasse)**.**
*Sei $R \subseteq A \times A$ eine Äquivalenzrelation über $A$ und sei $a \in A$. Dann wird*

$$[a]_R := \{x \in A | (a, x) \in R\}$$

***Äquivalenzklasse** von $R$ genannt. Wenn $R$ vom Kontext her eindeutig ist, so schreiben wir $[a]$ statt $[a]_R$.*

**Definition 2.19** (Quotient)**.**
*Sei $R \subseteq A \times A$ eine Äquivalenzrelation über $A$. Die Menge aller Äquivalenzklassen von $R$ wird **Quotient** von $A$ bezüglich $R$ genannt und mit*

$$A/R := \{[a]_R | a \in A\}.$$

*bezeichnet.*

**Definition 2.20** (Eigenschaften 2-stelliger Relationen)**.**
*Seien $A$ und $B$ zwei Mengen. Sei $R \subseteq A \times B$ eine 2-stellige Relation über $A$ und $B$. $R$ heißt:*

1. ***linkstotal**, wenn*

$$\forall a \in A.\ \exists b \in B.\ (a, b) \in R$$

2. ***rechtstotal** oder **surjektiv**, wenn*

$$\forall b \in B.\ \exists a \in A.\ (a, b) \in R$$

3. ***linkseindeutig** oder **injektiv**, wenn*

$$\forall a_1, a_2 \in A.\ \forall b \in B.\ ((a_1, b) \in R \wedge (a_2, b) \in R) \Rightarrow a_1 = a_2$$

4. ***rechtseindeutig**, wenn*

$$\forall a \in A.\ \forall b_1, b_2 \in B.\ ((a, b_1) \in R \wedge (a, b_2) \in R) \Rightarrow b_1 = b_2$$

**Definition 2.21** (Abbildung)**.**
*Seien $A$ und $B$ zwei Mengen und sei $f \subseteq A \times B$ eine Relation über $A$ und $B$. $f$ heißt:*

1. ***partielle Abbildung**, gdw. $f$ rechtseindeutig ist. Sie wird bezeichnet mit $f : A \rightharpoonup B$.*
2. ***Abbildung** oder **Funktion**, gdw. $f$ rechtseindeutig und linkstotal ist. Dies wird geschreiben als $f : A \to B$.*
3. ***bijektive** Funktion, gdw. $f$ eine Funktion ist und $f$ sowohl injektiv als auh surjektiv ist.*

**Definition 2.22** (Mengenfamilie)**.**
*Sei $M$ eine Menge von Mengen und sei $I$ eine Menge, deren Elemente **Indizes** genannt werden. $I$ wird auch als **Indexmenge** bezeichnet. Eine **Mengenfamilie** ist eine surjektive Abbildung $A : I \to M$. Sie wird mit $(A_i)_{i \in I}$ repräsentiert.*

Nachdem wir die fundamentalen mathematischen Grundlagen eingeführt haben, wenden wir uns nun den algebraischen Spezifikationen zu. Die Definition baut unter anderem auf dem Konzept der algebraischen Signaturen auf, die wir zunächst betrachten.

**Definition 2.23** (Algebraische Signatur)**.**
*Eine **algebraische Signatur** ist ein Tupel $\Sigma = (S, OP)$ bestehend aus*

1. *einer endlichen, nicht-leeren Menge $S$ , deren Elemente **Sortensymbole** genannt werden, und*

2. *einer endlichen Mengenfamilie* $OP = (OP_{w,s})_{(w,s)\in S^*\times S}$*, wobei für alle* $(w,s) \in S^* \times S$*, die Elemente von* $OP_{w,s}$ *als* ***Operationssymbole*** *bezeichnet werden.*

Der Einfachheit halber nutzen wir im weiteren Verlauf die folgende Notation.

**Notation 2.24.**
*Sei* $\Sigma = (S, OP)$ *eine algebraische Signatur und sei* $f \in OP_{w,s}$ *ein Operationssymbol. Dann schreiben wir für* $f$*,* $f : w \to s$*. Für* $w = \lambda$ *schreiben wir* $f :\to s$ *und nennen* $f$ ***Konstantensymbol****.*

Um dieses Konzept etwas zu veranschaulichen, betrachten wir das folgende Beispiel.

**Beispiel 2.8.** (Algebraische Signatur für $\mathbb{N}$)

Mit Hilfe von Definition 2.23, können wir eine algebraische Signatur für die natürlichen Zahlen angeben.

$$\begin{aligned}
\Sigma_{nat} = \quad & (S, OP) \text{ mit} \\
S = \quad & \{\text{nat}\} \\
OP = \quad & (OP_{w,s})_{\{(w,s)|w\in\{\lambda,\text{nat}\},s\in\{\text{nat}\}\}} \\
& OP_{\lambda,\text{nat}} = \{zero\} \\
& OP_{\text{nat,nat}} = \{\ \text{succ}\ \}
\end{aligned}$$

Um die Lesbarkeit zu vereinfachen nutzen wir im weiteren Verlauf die folgende Darstellung.

$\Sigma_{nat} =$

| | | |
|---|---|---|
| **sorts:** | nat | |
| **operations:** | $zero:$ | $\to$ nat |
| | succ: | nat $\to$ nat |

Wie in diesem Beispiel deutlich wird, ist es mit den algebraischen Signaturen möglich, Datentypen auf einer syntaktischen Ebene zu definieren. Auf dieser Ebene ist jedoch die Semantik unklar, denn es gibt unendlich viele Arten die Syntax zu interpretieren. Aus diesem Grund definiert man die sogenannte *Algebra*, die die algebraische Signatur interpretiert, auf die sich die Algebra bezieht.

**Definition 2.25** ($\boldsymbol{\Sigma}$-Algebra)**.**
*Sei* $\Sigma = (S, OP)$ *eine algebraische Signatur.*
*Eine* ***$\Sigma$-Algebra*** *ist ein Tupel* $A = ((A_s)_{s\in S}, (f_A)_{f\in OP})$ *bestehend aus den folgenden Attributen.*

1. *Für alle Sortensymbole* $s \in S$ *wird die nicht-leere Menge* $A_s$ ***Trägermenge*** *zur Sorte* $s$ *genannt.*

2. *Für alle* $f : s_1 \dots s_n \to s \in OP$ *mit* $n > 0$, *wird* $f_A : A_{s_1} \times \dots \times A_{s_n} \to A_s$ ***Abbildung***, *und mit* $n = 0$ ***Konstante*** *genannt.*

**Beispiel 2.9.** (Beispiel-Algebren für $\Sigma_{nat}$ )

Um die Hauptidee der $\Sigma-$Algebren zu veranschaulichen, definieren zwei verschiedene $\Sigma-$Algebren ($A$ and $B$) zur algebraischen Signatur $\Sigma_{nat}$ (aus Beispiel 2.8).

| $\Sigma_{nat}$ | A | B |
|---|---|---|
| $nat$ | $A_{nat} := \mathbb{N}$ | $B_{nat} := \{0, 2, 4, \dots\}$ |
| $zero :\ \to \text{nat}$ | $zero_A := 0$ | $zero_B := 0$ |
| $succ :\ \text{nat} \to \text{nat}$ | $succ_A(x) := x + 1$ | $succ_B(x) := x + 2$ |

Das Konzept der algebraischen Signaturen kann durch Variablen bzgl. Sorten erweitert werden.

**Definition 2.26** (Algebraische Signatur mit Variablen)**.**
*Sei* $\Sigma = (S, OP)$ *eine algebraische Signatur und sei* $X = (X_s)_{s \in S}$ *eine Mengenfamilie. Für alle Sortensymbole* $s \in S$ *werden die Elemente von* $X_s$ ***Σ-Variablen*** *zur Sorte* $s$ *genannt.*

*Seien* $X$ *und* $OP$ *disjunkt, dann wird das Tripel* $\Sigma = (S, OP, X)$ ***algebraische Signatur mit Variablen*** *genannt.*

Um das Konzept der Algebren zu generalisieren, kann man die Trägermengen der Sorten sowie Operationen auf einer syntaktischen Ebene definieren. Die so entstehenden Konstrukte werden *Terme* genannt.

**Definition 2.27** (**Σ**-(Grund-)Terme)**.**
*Sei* $\Sigma = (S, OP, X)$ *eine algebraische Signatur mit Variablen.*

- *Die Menge* $T_{\Sigma,s}(X)$ *von* ***Σ-Termen zur Sorte*** $s \in S$ ***über*** $X$ *ist induktiv definiert durch*
  1. $\forall x \in X_s.\ x \in T_{\Sigma,s}(X)$
  2. $\forall (c :\to s) \in OP.\ c \in T_{\Sigma,s}(X)$
  3. $\forall (f : s_1 \dots s_n \to s) \in OP.\ \forall t_i \in T_{\Sigma,s_i}(X).\ \forall s_i \in S, i \in \{1, \dots, n\}.$ $f(t_1, \dots, t_n) \in T_{\Sigma,s}(X)$
- $T_\Sigma(X) = (T_{\Sigma,s}(X))_{s \in S}$ *wird Familie von* ***Σ-Termen*** *genannt.*
- *Wenn* $X = \emptyset$, *dann heißt* $T_\Sigma = T_\Sigma(\emptyset)$ *Familie der* ***Σ-Grundterme****.*

Die aus Σ-(Grund-)Termen aufgebaute Algebra wird *Termalgebra* genannt:

**Definition 2.28** (Termalgebra)**.**
*Sei $\Sigma = (S, OP)$ algebraische Signatur. Die* ***Termalgebra*** *$T_\Sigma$ bzgl. $\Sigma$ ist wie folgt definiert.*

1. *Für alle $s \in S$ ist die Trägermenge $T_{\Sigma,s}$ die Menge aller $\Sigma-$Grundterme zur Sorte $s$.*
2. *Für alle Konstanten $f :\to s \in OP$ in $T_\Sigma$ gilt:*
$$f_{T_\Sigma} = f.$$
3. *Für alle Abbildungen $f : s_1 \dots s_n \to s \in OP$ in $T_\Sigma$, für alle $t_i \in T_{\Sigma,s_i}$ und für alle $s_i \in S, i \in \{1, \dots, n\}$ gilt:*
$$f_{T_\Sigma}(t_1, \dots, t_n) = f(t_1, \dots, t_n).$$

**Bemerkung 2.29.**
*Die Termalgebra ist eine* ***initiale Algebra*** *bzgl. $\Sigma$. Das heißt, dass für alle $\Sigma-$Algebren $A$ ein eindeutiger Homomorphismus $h : T_\Sigma \to A$ existiert.*

An dieser Stelle setzen wir voraus, dass der Begriff „Homomorphismus“ bereits bekannt ist und verzichten daher auf dessen Einführung.

**Beispiel 2.10.** (Termalgebra für $\Sigma_{nat}$ )

Um die Idee von Termalgebren zu verdeutlichen, vergleichen wir sie im Folgenden mit der Algebra $A$, die bereits in Beispiel 2.9 vorgestellt wurde.

| $\Sigma_{nat}$ | $T_{\Sigma_{nat}}$ | A |
|---|---|---|
| $nat$ | $T_{\Sigma_{nat},nat}$ | $A_{nat} := \mathbb{N}$ |
| $zero :\ \to$ nat | $zero_{T_{\Sigma_{nat}}} := zero$ | $zero_A := 0$ |
| $succ :$ nat $\to$ nat | $succ_{T_{\Sigma_{nat}}}(t) := succ(t)$ | $succ_A(x) := x + 1$ |

Die Interpretation der Termalgebra erfolgt über die Auswertung der $\Sigma-$Grundterme.

**Definition 2.30** (Auswertung von $\boldsymbol{\Sigma-}$Grundtermen)**.**
*Sei $\Sigma = (S, OP)$ eine algebraische Signatur und sei $A = ((A_s)_{s\in S}, (f_A)_{f\in OP})$ eine $\Sigma$-Algebra. Die* ***Auswertung*** *$eval(A)$* ***von $\Sigma-$Grundtermen in $A$*** *ist eine Familie von Abbildungen:*

$$eval^A = (eval^A_s : T_{\Sigma,s} \to A_s)_{s\in S}\ .$$

*Wir kürzen dies mit $eval^A : T_\Sigma \to A$ ab.*

*Die Abbildungen sind wie folgt definiert.*

1. *Für alle* $f : w \to s \in OP_{w,s}$ *mit* $|w| = 0$ *gilt:*

$$eval_s^A(f) := f_A$$

2. *Für alle* $f : w \to s \in OP_{w,s}$ *mit* $w = s_1 \dots s_n, n > 0$ *und alle* $\Sigma-$*Grundterme* $t_1 \in T_{\Sigma,s_1}, \dots, t_n \in T_{\Sigma,s_n}$ *gilt:*

$$eval_s^A(f(t_1, \dots, t_n)) := f_A(eval_{s_1}^A(t_1), \dots, eval_{s_n}^A(t_n))$$

**Definition 2.31** (Variablenbelegung)**.**
*Sei* $\Sigma = (S, OP, X)$ *eine algebraische Signatur mit Variablen und sei* $A = ((A_s)_{s\in S}, (f_A)_{f\in OP})$ *eine* $\Sigma$*-Algebra. Dann ist eine* ***Variablenbelegung*** *über* $A$ *eine Familie von von Abbildungen*

$$ass^A := (ass_s^A : X_s \to A_s)_{s\in S}.$$

*Wir kürzen dies mit* $ass^A : X \to A$ *ab.*

**Definition 2.32** (Auswertung von $\boldsymbol{\Sigma-}$Termen)**.**
*Sei* $\Sigma = (S, OP, X)$ *eine algebraische Signatur mit Variablen,* $A = ((A_s)_{s\in S}, (f_A)_{f\in OP})$ *eine* $\Sigma$*-Algebra und* $ass^A : X \to A$ *eine Variablenbelegung über* $A$*. Dann ist die* ***Auswertung von*** $\boldsymbol{\Sigma-}$***Termen in*** $\boldsymbol{A}$ ***bzgl.*** $\boldsymbol{ass}$ *eine Familie von Abbildungen*

$$xeval(ass)^A = (xeval(ass)_s^A : T_{\Sigma,s}(X) \to A_s)_{s\in S} \ .$$

*Wir kürzen dies mit* $xeval(ass)^A : T_\Sigma(X) \to A$ *ab.*

*Die Abbildungen sind für alle Sortensymbole* $s \in S$ *rekursiv wie folgt definiert.*

1. *Für alle Variablen* $x \in X_s$ *gilt:*

$$xeval(ass)_s^A(x) := ass_s^A(x).$$

2. *Für alle* $f : w \to s \in OP_{w,s}$ *mit* $|w| = 0$*gilt:*

$$xeval(ass)_s^A(f) := f_A$$

3. *Für alle* $f : w \to s \in OP_{w,s}$ *mit* $w = s_1 \dots s_n, n > 0$ *und alle* $\Sigma-$*Terme* $t_1 \in T_{\Sigma,s_1}(X), \dots, t_n \in T_{\Sigma,s_n}(X)$ *gilt:*

$$xeval(ass)_s^A(f(t_1, \dots, t_n)) := f_A(xeval(ass)_{s_1}^A(t_1), \dots, xeval(ass_{s_n}^A(t_n))$$

Zwei $\Sigma-$Terme können trotz unterschiedlicher syntaktischen Darstellung die gleiche semantische Bedeutung besitzen. Diese Überlegung führt zu dem Begriff der *gültigen* $\Sigma-$*Gleichung.*

**Definition 2.33** ($\boldsymbol{\Sigma}$-Gleichung)**.**
*Sei* $\Sigma = (S, OP, X)$ *eine algebraisch Signatur mit Variablen,* $s \in S$ *eine beliebige Sorte und* $t_l, t_r \in T_{\Sigma,s}(X)$ $\Sigma$*-Terme zur Sorte* $s$*. Dann wird*

$$e \quad \equiv \quad t_l = t_r$$

**Σ− *Gleichung*** *genannt.*

**Definition 2.34** (Gültigkeit einer **Σ**-Gleichung)**.**
*Sei $\Sigma = (S, OP, X)$ eine algebraisch Signatur mit Variablen,*
*$A = ((A_s)_{s\in S}, (f_A)_{f\in OP})$ eine $\Sigma$-Algebra und $e \equiv t_l =_s t_r$ eine $\Sigma-$Gleichung.*
*Dann wird e als **gültig in A** (geschrieben als $A \models e$) bezeichnet, wenn für alle Variablenbelegungen $ass^A : X \to A$*

$$xeval(ass)^A_s(t_l) = xeval(ass)^A_s(t_r)$$

*gilt.*

Mit diesen Definitionen kann man das Konzept der algebraischen Signatur auf den Begriff der algebraischen Spezifikation erweitern.

**Definition 2.35** (Algebraische Spezifikation)**.**
*Eine **algebraische Spezifikation** ist ein Tupel $SP = (\Sigma, E)$ bestehend aus*

1. *einer algebraischen Signatur mit Variablen $\Sigma = (S, OP, X)$ und*
2. *einer Menge von $\Sigma-$Gleichungen $E$ .*

**Beispiel 2.11.** (Algebraische Spezifikation für ℕ)

Um die Idee der algebraischen Spezifikation zu veranschaulichen, erweitern wir die algebraische Signatur $\Sigma_{nat}$ aus Beispiel 2.8 um Variablen, ein Operationssymbol 'add' und drei Gleichungen.

$\Sigma'_{nat} =$

| | | | | |
|---|---|---|---|---|
| **sorts:** | nat | | | |
| **operations:** | $zero$ : | | | $\to$ nat |
| | succ: | | | nat $\to$ nat |
| | add: | | | nat $\times$ nat $\to$ nat |
| **variables:** | $x, y \in$ nat | | | |
| **equations:** | $e_1$ | $\equiv$ | $add(zero, x)$ | $= x$ |
| | $e_2$ | $\equiv$ | $add(x, y)$ | $= add(y, x)$ |
| | $e_3$ | $\equiv$ | $add(x, succ(y))$ | $= succ(add(x, y))$ |

**Beispiel 2.12.** (Gültigkeit einer Σ-Gleichung )
Aufbauend auf der algebraischen Spezifikation aus Beispiel 2.11 nutzen wir die folgende $\Sigma'_{nat}-$Algebra $A$ um die Gültigkeit einer Σ-Gleichung zu veranschaulichen.

| $\Sigma'_{nat}$ | | | A |
|---|---|---|---|
| *nat* | | | $A_{nat} := \mathbb{N}$ |
| *zero* : | | $\rightarrow$ nat | $zero_A := 0$ |
| succ: | | nat $\rightarrow$ nat | $succ_A(x) := x + 1$ |
| add: | | nat $\times$ nat $\rightarrow$ nat | $add_A(x, y) := x + y$ |
| $x, y \in$ nat | | | |
| $e_1$ | $\equiv$ | $add(zero, x) = x$ | |
| $e_2$ | $\equiv$ | $add(x, y) = add(y, x)$ | |
| $e_3$ | $\equiv$ | $add(x, succ(y)) = succ(add(x, y))$ | |

Wir wollen die Gültigkeit der Gleichung $e_1$ in $A$ überprüfen. Es gilt:

$$
\begin{aligned}
& xeval(ass)_{nat}(add(zero, x)) \\
= \; & add_A(xeval(ass)_{nat}(zero), xeval(ass)_{nat}(x)) && \text{(Def. 2.32)} \\
= \; & add_A(zero_A, xeval(ass)_{nat}(x)) && \text{(Def. 2.32)} \\
= \; & add_A(0, xeval(ass)_{nat}(x)) && \text{(Def. of } zero_A\text{)} \\
= \; & 0 + xeval(ass)_{nat}(x) && \text{(Def. of } add_A\text{)} \\
= \; & xeval(ass)_{nat}(x) && \text{(Def. of + for } \mathbb{N}\text{)}
\end{aligned}
$$

Das bedeutet, dass für alle $x \in X_{nat}$ $e_1$ ist gültig in Algebra $A$. Dies schreiben wir als $A \models e$.

Gerade in dem von uns betrachteten, komponentenbezogenen Rahmen ist es wünschenswert zwei algebraische Spezifikationen zu kombinieren. Dies führt zu folgender Definition, für die wir voraussetzen, dass der Begriff „disjunkte Vereinigung“ bereits bekannt ist.

**Definition 2.36** (Komposition von Algebraischen Spezifikationen)**.**
*Seien $SP_1 = (S_1, OP_1, X_1, E_1)$ und $SP_2 = (S_2, OP_2, X_2, E_2)$ zwei algebraische Spezifikationen. Dann ist ihre **Komposition** $SP = SP_1 + SP_2$ wie folgt definiert.*

$$SP = (S_1 \uplus S_2, OP_1 \uplus OP_2, X_1 \uplus X_2, E_1 \uplus E_2),$$

*$\uplus$ steht hierbei für die disjunkte Vereinigung von Mengen.*

**Notation 2.37.**
*Für eine algebraische Signatur mit Variablen $\Sigma = (S, OP, X)$, bezeichnen wir die algebraische Spezifikation $SP = (S \uplus S1, OP \uplus OP_1, X \uplus X_1, E_1)$ mit $\Sigma + (S_1, OP_1, X_1, E_1)$.*

**Definition 2.38** (Modell einer Algebraischen Spezifikation)**.**
*Sei $SP = (\Sigma, E)$ eine algebraische Spezifikation und $A$ eine $\Sigma-$Algebra. $A$ heißt **Modell von SP** or **SP−Algebra**, wenn*

$$\forall e \in E.\ A \models e$$

*gilt.*

**Definition 2.39** (Kongruenz von **Σ−**Grundtermen)**.**
*Sei $SP = (\Sigma, E)$ eine algebraische Spezifikation.*

*Für alle Sortensymbole $s \in S$ heißen zwei $\Sigma-$Grundterme $t_1, t_2 \in T_{\Sigma,s}$ **kongruent** in $SP$, geschrieben als $t_1 \sim^E t_2$, gdw. für alle SP-Algebren $A$ $eval_s^A(t_1) = eval_s^A(t_2)$ gilt.*

**Theorem 2.40.**
*Für alle algebraischen Spezifikationen $SP = (\Sigma, E)$ ist $\sim^E$ eine Äquivalenzrelation.*

*Beweis.*
Um zu beweisen, dass $\sim^E$ eine Äquivalenzrelation nach Def. 2.17 ist, zeigen wir für alle $s \in S$:

1. (Reflexivität) $\forall t \in T_{\Sigma,s}.\ t \sim^E t$
2. (Symmetrie) $\forall t_1, t_2 \in T_{\Sigma,s}.\ t_1 \sim^E t_2 \Rightarrow t_2 \sim^E t_1$
3. (Transitivität) $\forall t_1, t_2, t_3 \in T_{\Sigma,s}.\ t_1 \sim^E t_2 \wedge t_1 \sim^E t_2 \Rightarrow t_1 \sim^E t_3$

Sei $SP = (\Sigma, E)$ im Folgenden eine algebraische Spezifikation, $s \in S$ ein Sortensymbol und $A$ eine $SP-$Algebra.

**Reflexivität**
Sei $t \in T_{\Sigma,s}$ ein $\Sigma-$Grundterm zur Sorte $s$.

Nach Definition 2.39 gilt:

$$t \sim^E t \Leftrightarrow eval_s^A(t) = eval_s^A(t)$$

Da $eval_s^A(t) = eval_s^A(t)$ trivialer Weise wahr ist, ist $t$ kongruent zu sich selbst. Also gilt $t \sim^E t$.

**Symmetrie**
Seien $t_1, t_2 \in T_{\Sigma,s}$ zwei $\Sigma-$Grundterme zur Sorte $s$. Es gilt:

$$
\begin{array}{lcl}
 & t_1 \sim^E t_2 & \\
\Leftrightarrow & eval_s^A(t_1) = eval_s^A(t_2) & (\text{ Def. 2.39}) \\
\Leftrightarrow & eval_s^A(t_2) = eval_s^A(t_1) & (\text{Symmetrie von } =) \\
\Leftrightarrow & t_2 \sim^E t_1 & (\text{ Def. 2.39})
\end{array}
$$

**Transitivität**
Seien $t_1, t_2, t_3 \in T_{\Sigma,s}$ drei $\Sigma-$Grundterme zur Sorte $s$. Unter den Annahmen, dass $t_1 \sim^E t_2$ und $t_2 \sim^E t_3$, gilt:

$$
\begin{array}{lcll}
 & (t_1 \sim^E t_2) \wedge (t_2 \sim^E t_3) & & \\
\Leftrightarrow & (eval_s^A(t_1) = eval_s^A(t_2)) \wedge (eval_s^A(t_2) = eval_s^A(t_3)) & (\text{ Def.2.39}) \\
\Leftrightarrow & (eval_s^A(t_1) = eval_s^A(t_2)) \text{ und } (eval_s^A(t_2) = eval_s^A(t_3)) & (\text{ Def. of } \wedge) \\
\Leftrightarrow & eval_s^A(t_1) = eval_s^A(t_3) & (\text{Reflexivität von } \sim^E) \\
\Leftrightarrow & t_1 \sim^E t_3 & (\text{ Def. 2.39})
\end{array}
$$

□

**Theorem 2.41.**
*Sei $SP = (\Sigma, E)$ eine algebraische Spezifikation. Dann gilt für all $SP-$Algebren $A$, für alle Operationssymbole $f : w \to s' \in OP_{w,s'}$ und alle $\Sigma-$Grundterme zur Sorte $s \in S$ $t_1, t_2 \in T_{\Sigma,s}$:*

$$t_1 \sim^E t_2 \Rightarrow f(\dots, t_1, \dots) \sim^E f(\dots, t_2, \dots)$$

*Beweis.*
Seien $SP = (\Sigma, E)$ eine algebraische Spezifikation, $t_1, t_2 \in T_{\Sigma,s}$ zwei $\Sigma-$Grundterme zur Sorte $s$, $A$ eine $SP-$Algebra und $f : w \to s' \in OP_{w,s'}$ ein Operationssymbol. Unter der Annahme, dass $t_1$ und $t_2$ kongruent sind, gilt nach Definition 2.39 $eval_s^A(t_1) = eval_s^A(t_2)$.

In diesem Fall müssen wir beweisen, dass

$$f(\dots, t_1, \dots) \sim^E f(\dots, t_2, \dots)$$

gilt.

Präziser ausgedrückt, müssen wir zeigen, dass

$$eval_{s'}^A(f(\dots, t_1, \dots)) = eval_{s'}^A(f(\dots, t_2, \dots))$$

gilt.

Der Beweis erfolgt über die Struktur von $f$.

**Fall 1: $w = \lambda$**

$$\begin{array}{lcll} & eval^A_{s'}(f(\ldots, t_1, \ldots)) & \\ = & f_A & \text{(Def. 2.30)} \\ = & eval^A_{s'}(f(\ldots, t_2, \ldots)) & \text{(Def. 2.30)} \end{array}$$

**Fall 2: $w = s_1 \ldots s_n$**

$$\begin{array}{lcll} & eval^A_{s'}(f(\ldots, t_1, \ldots)) & \\ = & f_A(eval^A_{s_1}(\ldots), eval^A_s(t_1), eval^A_{s_n}(\ldots)) & \text{(Def. 2.30)} \\ = & f_A(eval^A_{s_1}(\ldots), eval^A_s(t_2), eval^A_{s_n}(\ldots)) & \text{(Assumption } t_1 \sim^E t_2\text{)} \\ = & eval^A_{s'}(f(\ldots, t_2, \ldots)) & \text{(Def. 2.30)} \end{array}$$

□

Analog zu der Termalgebra, die bezüglich einer algebraischen Signatur (mit Variablen) eine initiale Algebra bildet, kann auch für eine algebraische Spezifikation eine Algebra konstruiert werden, auf die alle SP-Algebren unter zu Hilfenahme der angegebenen $\Sigma-$Gleichungen zurückgeführt werden können. Diese Algebra wird als *Quotiententermalgebra* bezeichnet.

**Definition 2.42** (Quotiententermalgebra)**.**
*Sei $SP = (\Sigma, E)$ eine algebraische Spezifikation und sei $\sim^E$ eine Kongruenzrelation auf $\Sigma-$Grundtermen. Dann heißt, $T_{SP} = T_\Sigma / \sim^E$* ***Quotiententermalgebra*** *von $SP$.*

*Sie ist wie folgt definiert..*

1. *Für alle $s \in S$ gilt: $(T_\Sigma / \sim^E)_s = T_\Sigma, s / \sim^E$.*
2. *Für alle $f : s_1 \ldots s_n \to s \in OP$ und $[t_i] \in T_{\Sigma,s_i} / \sim^E, i \in [1, n]$ gilt:*

$$f_{T_\Sigma/\sim^E}([a_1], \ldots, [a_n]) := [f(a_1, \ldots, a_n)].$$

**Bemerkung 2.43.**
*Die Quotiententermalgebra ist eine* ***initiale Algebra*** *für $SP$. Das bedeutet, für alle $SP-$Algebren $A$ existiert ein eindeutiger Homomorphismus $h : T_{SP} \to A$.*

Der Kongruenzbegriff auf $\Sigma-$Grundtermen (Definition 2.39) kann auf $\Sigma-$Terme $t_1, t_2 \in T_\Sigma(X)$ erweitert werden.

**Definition 2.44** (Kongruenz von $\Sigma-$Termen)**.**
*Für alle $s \in S$ ist die* ***Kongrunez von $\Sigma-$Termen*** *wie folgt definiert:*

$$t_1 \sim^E t_2 \Leftrightarrow \forall ass : X_s \to T_{\Sigma,s}.\ xeval(ass)(t_1) \sim^E xeval(ass)(t_2)$$

**Bemerkung 2.45.**
*In Definition 2.44 wird $T_\Sigma$ als Termalgebra (Def. 2.28) aufgefasst. Dadurch können Variablen, die in den Termen $t_1$ und $t_2$ vorkommen, bei der Ausführung von xeval mit $\Sigma-$Grundtermen belegt werden.*

*Wenn die Auswertung von $t_1$ und $t_2$ in $T_\Sigma$ bzgl. ass ergibt, dass $t_1$ und $t_2$ kongruent sind, dann sind die beiden Terme auch in allen anderen $\Sigma-$Algebren kongruent.*

**Definition 2.46** (Relationale Algebraische Spezifikation)**.**
*Sei $SP = (\Sigma, E)$ eine algebraische Spezifikation. Sei $R$ eine Menge von Symbolen. Jedes $r :< s_1 \dots s_n >\in R$ mit $n > 0$ und $s_1 \dots s_n \in S$ wird* ***Relationssymbol*** *genannt.*

*Das Tripel $SP_R = (\Sigma, E, R)$ wird* ***relationale algebraische Spezifikation*** *genannt.*

**Definition 2.47** (***SP*$_R$**-Algebra)**.**
*Sei $SP_R = (\Sigma, E, R)$ eine relationale algebraische Spezifikation und sei $A$ eine SP-Algebra. Dann ist eine $SP_R$-Algebra das Tupel $A_{SP_R} = (A, (A_r)_{r\in R})$, sodass*

$$\text{für alle } r \in R,\ A_r :< A_{s_1} \dots A_{s_n} > \text{ eine Relation ist.}$$

### 2.2.3 Algebraische Petrinetze

In diesem Abschnitt führen wir, basierend auf den Ausführungen in [Vau87, Rei91, Sch96], die Grundlagen von algebraischen Petrinetzen ein.

Dazu betrachten wir zunächst die syntaktischen Elemente bevor wir zur Ausführungssemantik der algebraischen Petrinetze kommen.

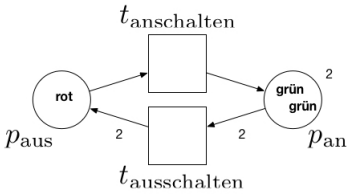
#### Syntax

Wir beginnen mit einem kurzen Beispiel um eine Intuition für die algebraischen Petrinetze zu geben.

**Beispiel 2.13.** (Lichtschalter - Fortsetzung)
In Beispiel 2.1 auf Seite 16 haben wir bereits das Verhalten einer Steuerung für drei LEDs modelliert. Dabei sind wir von drei gleichen LEDs ausgegangen und hatten keine Möglichkeit zwischen ihnen zu differenzieren. Wir erweitern das Modell nun um die Möglichkeit die LEDs zu unterscheiden.

Nehmen wir an, dass wir zwei *grüne* und eine *rote* LED steuern wollen. Dafür benötigen wir die Möglichkeit Marken zu unterscheiden. Eine Option ist es, die Marken explizit zu benennen. Dies ist exemplarisch im folgenden Netz dargestellt.

$N_1$ :




Die initiale Markierung für $N_1$ ist:

$$M_0(p_{an})(x) = \begin{cases} 2 & \text{, falls } x = \text{grün} \\ 0 & \text{, falls } x = \text{rot} \end{cases}$$

$$M_0(p_{aus})(x) = \begin{cases} 0 & \text{, falls } x = \text{grün} \\ 1 & \text{, falls } x = \text{rot} \end{cases}$$

Die Abbildung, die die Anzahl der Vorkommen eines Elementes in einer Menge angibt, wird Multimenge genannt.

**Definition 2.48** (Multimenge)**.**
*Sei $A$ eine Menge. Die Abbildung*

$$\mathcal{M} : A \to \mathbb{Z}$$

*wird* ***Multimenge*** *genannt.*

*Die* ***leere Multimenge*** *$\vartheta$ über $A$ ist wie folgt definiert.*

$$\vartheta_A(a) = 0, \text{ für alle } a \in A$$

*Elemente $a \in A$, die einzeln auftreten, können als* ***einelementige Multimengen*** *$m_a$ über $A$ verstanden werden. Sie sind wie folgt definiert.*

$$m_a(x) = \begin{cases} 1, & \text{wenn } x = a \\ 0, & \text{sonst} \end{cases}$$

*Für zwei Multimengen $\mathcal{M}_1$ und $\mathcal{M}_2$ sind* ***Operationen auf Multimengen*** *wie folgt definiert.*

$$\begin{aligned} (\mathcal{M}_1 + \mathcal{M}_2)(a) &= \mathcal{M}_1(a) + \mathcal{M}_2(a) \\ -(\mathcal{M}_1)(a) &= -(\mathcal{M}_1(a)) \end{aligned}$$

*Hierbei sind $+$ und $-$ wie auf ganzen Zahlen üblich definiert.*

Das Konzept der Multimengen soll nun mit dem Konzept der algebraischen Spezifikationen vereinigt werden.

**Definition 2.49** (Multimengen - Spezifikation)**.**
*Sei $SP = (\Sigma, E)$ eine algebraische Spezifikation. Die* ***Multimengen - Spezifikation*** *$mSP$ ist die mit Multimengen zu jeder Sorte $s \in S$ angereicherte algebraische Spezifikation $SP$. Sie ist wie folgt definiert:*

$$
\left.
\begin{array}{lr}
mSP = SP+ & \\
\textbf{\textit{sorts:}} & m_s \\
\textbf{\textit{operations:}} & \vartheta_s : \rightarrow m_s \\
& MAKE_s : s \rightarrow m_s \\
& +_s : m_s\ m_s \rightarrow m_s \\
& -_s : m_s \rightarrow m_s \\
\textbf{\textit{variables:}} & p, q, r \in m_s \\
\textbf{\textit{equations:}} & +_s(p, \vartheta_s) = p \\
& +_s(p, q) = +_s(q, p) \\
& +_s(p, +_s(q, r)) = +_s(+_s(p, q), r) \\
& +_s(p, -_s(p)) = \vartheta_s
\end{array}
\right\} \quad \forall s \in S.
$$

**Bemerkung 2.50.**
*Sei $A$ eine Menge und $SP = (\Sigma, E)$ eine algebraische Spezifikation. Für jedes Konstantensymbol $f \in SP$ wird ein Term $MAKE_s(f)$ assoziiert. Wird $f$ zu $a \in A$ ausgewertet, so wird $MAKE_s(f)$ zur Menge $\{a\}$ mit $m(a) = 1$ ausgewertet.*

**Bemerkung 2.51.**
*Im weiteren Verlauf der Arbeit beziehen wir uns immer auf Multimengen-Spezifikationen, wenn wir über algebraische Spezifikationen sprechen. Zusätzlich meinen wir Multimengen von Termen, wenn wir Terme behandeln.*

**Notation 2.52.**
*Im weiteren Verlauf nutzen wir die folgenden Notationen.*

1. *Für eine relationale algebraische Spezifikation $SP_R = (\Sigma, E, R)$ wird $mSP$ mit $\hat{SP}_R = (\hat{\Sigma}, \hat{E}, R)$ mit $\hat{\Sigma} = (\hat{S}, \hat{OP}, \hat{X})$ bezeichnet.*
2. *Für Multimengen von Termen lassen wir bei Operationssymbolen den Sortenindex weg und schreiben beispielsweise $\vartheta$ statt $\vartheta_s$.*
3. *Seien $u$ und $v$ zwei Multimengen. Dann schreiben wir $u + v$ für $+(u, v)$. Für $u + (-(v))$ schreiben wir $u - v$.*
4. *Für $MAKE(t)$ schreiben wir $t$.*
5. *Wann immer keine Unklarheiten auftauchen, lassen wir Klammern weg.*

**Definition 2.53.**
*Sei $SP_R = (\Sigma, E, R)$ eine relationale algebraische Spezifikation.*

1. *Wenn die Definition der Multimenge $\mathcal{M}$ auf die natürlichen Zahlen beschränkt ist, also wenn für eine Menge $A$ die Multimenge $\mathcal{M}$ über $A$ definiert ist als $\mathcal{M} : A \to \mathbb{N}$, nennen wir $\mathcal{M}$ **nicht-negative Multimenge**.*

2. *Die Menge **nicht-negativer Operationssymbole** in $mSP$ ist wie folgt definiert*

$$NNS := \hat{OP} \setminus \{-_s | s \in S\}.$$

3. *Die Menge **nicht-negativer Multiterme** von $SP$ ist gegeben durch*

$$T_{\Sigma^+}(X) := \{t \in T_{\hat{\Sigma}}(X) | \exists u \in T_{NNS}(X).\ t \sim^E u\}$$

4. *Nach Definition 2.27 gilt $T_{\Sigma^+} := T_{\Sigma^+}(\emptyset)$ und für alle $s \in S$ gilt:*

$$\begin{aligned} T_{\Sigma^+,m_s}(X) &:= T_{\Sigma^+}(X) \cap T_{\hat{\Sigma},m_s}(X) \\ T_{\Sigma^+,m_s} &:= T_{\Sigma^+,m_s}(\emptyset) \end{aligned}$$

5. *Für zwei Multiterme $t_1, t_2 \in T_{\hat{\Sigma}}(X)$, wird $t_1$ als kleiner oder gleich zu $t_2$ bezeichnet. Es wird wie folgt geschrieben $t_1 \leq^E t_2$, wenn $t_2 - t_1$ nicht-negativ in $SP$ ist.*

**Definition 2.54** (**$SP_R$**-Guard)**.**
*Sei $SP_R = (\Sigma, E, R)$ eine relationale algebraische Spezifikation, seien $t_1, t_2 \in T_{\Sigma,s}(X)$ und sei $r \in R$ ein Relationssymbol. Das Tripel $(t_1, t_2, r)$, auch geschrieben als $[t_1\ r\ t_2]$, wird **$SP_R$-Guard** genannt.*

*Das leere Tripel (-,-,-) heißt **leerer $SP_R$-Guard** und wird mit [ ] bezeichnet.*

**Definition 2.55** (Guardmenge)**.**
*Sei $SP_R = (\Sigma, E, R)$ eine relationale algebraische Spezifikation, seien $t_1, t_2 \in T_{\Sigma,s}(X)$ und sei $r \in R$. Die **$SP_R$-Guardmenge** $G = (G_{\tilde{r}})_{\tilde{r} \in R^*}$ ist wie folgt definiert.*

1. *Für den leeren $SP_R$-Guard $g = [\ ]$ ist $g \in G_\lambda$.*

2. *Für alle $SP_R$-Guards $g = [t_1\ r\ t_2]$ ist $g \in G_r, r \in R$.*

3. *Für alle $g \in G_{\tilde{r}}$ gilt $\neg g \in G_{\tilde{r}}, \tilde{r} \in R^*$.*

4. *Für alle $g \in G_{\tilde{r}}$ und $g' \in G_{\tilde{r}'}$ gilt $g \wedge g' \in G_{\tilde{r}.\tilde{r}'}, \tilde{r}, \tilde{r}' \in R^*$.*

**Definition 2.56** (Auswertung eines Guards)**.**
*Sei $SP_R = (\Sigma, E, R)$ eine relationale algebraische Spezifikation, seien $t_1, t_2 \in T_{\Sigma,s}(X)$, sei $A_{SP_R} = (A, (A_r)_{r \in R})$ eine $SP_R$-Algebra und sei $ass^A : X \to A$ eine Variablenbelegung.*

*Die **Auswertung eines $SP_R$-Guards** in $A_{SP_R}$ bzgl. ass ist eine Familie von Abbildungen, für die gilt:*

$$geval(ass)^{A_{SP_R}} =_{def} (geval(ass)_{\tilde{r}}^{A_{SP_R}} : G_{\tilde{r}} \to \{TRUE, FALSE\})_{\tilde{r} \in R^*}$$

*Diese sind induktiv wie folgt definiert.*

1. *Für* $|\tilde{r}| = 0$ *gilt* $geval(ass)_{\tilde{r}}^{A_{SP_R}}([\,]) = TRUE$
2. *Für* $|\tilde{r}| = 1$*gilt:*

$$geval(ass)_{\tilde{r}}^{A_{SP_R}}([t_1\ r\ t_2]) = TRUE \Leftrightarrow ((xeval(ass)^A(t_1), (xeval(ass)^A(t_2))) \in A_r$$
$$geval(ass)_{\tilde{r}}^{A_{SP_R}}(\neg[t_1\ r\ t_2]) = TRUE \Leftrightarrow ((xeval(ass)^A(t_1), (xeval(ass)^A(t_2))) \notin A_r$$

3. *Für* $|\tilde{r}| > 1$, $g \in G_{\tilde{r}}$ *und* $g' \in G_{\tilde{r}'}$ *gilt:*

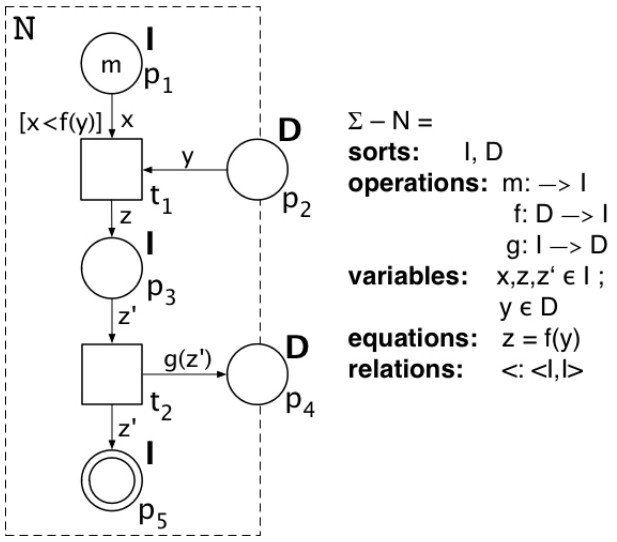
$$geval(ass)_{\tilde{r}}^{A_{SP_R}}(\neg g) = \neg geval(ass)_{\tilde{r}}^{A_{SP_R}}(g)$$
$$geval(ass)_{\tilde{r}.\tilde{r}'}^{A_{SP_R}}(g \wedge g') = geval(ass)_{\tilde{r}}^{A_{SP_R}}(g) \wedge geval(ass)_{\tilde{r}'}^{A_{SP_R}}(g')$$

**Definition 2.57** (Algebraisches Petrinetz)**.**
*Sei* $N = (P, T, F, C, W, M_0)$ *ein Petrinetz und sei* $SP_R = (\Sigma, E, R)$ *eine relationale algebraische Spezifikation.*
*Das Tupel* $AN = (N, (h, M_0, \lambda, guard), E)$ *wird* ***algebraisches Petrinetz*** *über* $SP_R$ *genannt. Es besteht aus*

- *einem Petrinetz* $N$*,*
- *einer Abbildung* $h : P \to S$*, die jedem Platz* $p \in P$ *ein Sortensymbol* $s \in S$ *zuweist,*
- *einer Abbildung* $M_0 : P \to T_{OP^+}$*, die sogenannte initiale Markierung, die jedem Platz eine nicht-negative Multimenge von Grundtermen zuweist,*
- *einer Abbildung* $\lambda : F \to T_{OP^+}(X)$*, die jeder Kante eine Beschriftung mit Termen mit Variablen zuweist,*
- *einer Abbildung* $guard : T \to G$*, die jeder Transition einen Guard zuweist und*
- *einer Menge von* $\Sigma$*-Gleichungen* $E$*.*

**Beispiel 2.14.** (Offenes Algebraisches Petrinetz)

Im Folgenden betrachten wir die graphische Repräsentation eines offenen algebraischen Petirnetzes $(N, (h, M_0, \lambda), E)$.




Es gelten die folgenden mengentheoretischen Darstellungen.

$$\dots$$
$$h := \{(p_1, I), (p_2, D), (p_3, I), (p_4, D), (p_5, I)\}$$
$$M_0 := \{(p_1, \{m\}), (p_2, \emptyset), (p_3, \emptyset), (p_4, \emptyset), (p_5, \emptyset)\}$$
$$\lambda := \{((p_1, t_1), x), ((p_2, t_1), y), ((t_1, p_3), z), ((p_3, t_2), z'), ((t_2, p_4), g(z')), ((t_2, p_5), z')\}$$
$$guard := \{(t_1, [x < f(y)])\}$$
$$\dots$$

Der Einfachheit halber nutzen wir im weiteren Verlauf die folgende Notation.

**Notation 2.58.**
*Sei $SP_R = (\Sigma, E, R)$ eine relationale algebraische Spezifikation.*
*Sei $AN = (N, (h, M_0, \lambda, guard), E)$ ein offenes algebraisches Petrinetz über $SP_R$ mit Platzmenge $P$ und Transitionsmenge $T$ . Für alle $(x, y) \in (T \times P) \cup (P \times T)$ schreiben wir*

$$\xrightarrow{x,y} = \begin{cases} \lambda(x, y) & , \text{ wenn } (x, y) \in F, \\ \vartheta & , \text{ sonst .} \end{cases}$$

### Ausführungssemantik

**Definition 2.59** (Markierung)**.**
*Sei $SP_R = (\Sigma, E, R)$ eine relationale algebraische Spezifikation.*

*Sei $AN = (N, (h, M_0, \lambda, guard), E)$ ein offenes algebraisches Petrinetz über $SP_R$ mit Platzmenge $P$ und Transitionsmenge $T$. Die Abbildung*

$$M : P \to T_{\Sigma^+}$$

*wird **Markierung** genannt.*

**Definition 2.60** (Modus)**.**
*Sei $SP_R = (\Sigma, E, R)$ eine relationale algebraische Spezifikation mit Variablenmenge $X$. Sei $AN = (N, (h, M_0, \lambda, guard), E)$ ein offenes algebraisches Petrinetz über $SP_R$. Die Variablenbelegung*

$$\beta : X \to T_\Sigma$$

*wird **Modus** genannt.*

*$\beta$ kann kanonisch erweitert werden zu $\hat{\beta} : T_{\Sigma^+}(X) \to T_\Sigma$, was induktiv wie folgt definiert ist.*

1. *$\hat{\beta}(\hat{x}) = \beta(x)$ für alle $\hat{x} \in \hat{X}$ und $\hat{x} = MAKE(x)$*
2. *$\hat{\beta}(\hat{c}) = c$ für $\hat{c} :\to s \in T_{\Sigma^+}(X)$ und $\hat{c} = MAKE(c)$*
3. *$\hat{\beta}(f(t_1, \ldots, t_n)) = f(\hat{\beta}(t_1), \ldots, \hat{\beta}(t_1))$, für $f(t_1, \ldots, t_n) \in T_{\Sigma^+}(X)$*

**Definition 2.61** (Aktivierte Transition)**.**
*Sei $SP_R = (\Sigma, E, R)$ eine relationale algebraische Spezifikation.*
*Sei $AN = (N, (h, M_0, \lambda, guard), E)$ ein offenes algebraisches Petrinetz über $SP_R$ mit Platzmenge $P$ und Transitionsmenge $T$. Sei $A_{SP_R} = (T_\Sigma, (A_r)_{r \in R})$ eine $SP_R$- Algebra. Für eine gegebene Markierung $M$, eine Transition $t \in T$ und einen Modus $\beta$, wird $t$ als **aktiviert** bzgl. einer Markierung in einem Modus bezeichnet, genau dann wenn*

$$\forall p \in P.\ \hat{\beta}(\xrightarrow{p,t}) \leq_E M(p)$$
$$\text{und}$$
$$geval(\beta)^{A_{SP_R}}(guard(t)) = TRUE$$

**Definition 2.62** (Schalten einer Transition)**.**
*Sei $SP_R = (\Sigma, E, R)$ eine relationale algebraische Spezifikation.*
*Sei $AN = (N, (h, M_0, \lambda, guard), E)$ ein offenes algebraisches Petrinetz über $SP_R$ mit Platzmenge $P$ und Transitionsmenge $T$. Sei $M$ eine Markierung und $\beta$ ein Modus.*

*Wenn $t$ bzgl. $M$ in Modus $\beta$ aktiviert ist, kann $t$ **in Modus $\beta$ schalten**. Dies führt zur Folgemarkierung $M'$, die für jeden Platz $p \in P$ wie folgt definiert ist.*

$$M'(p) = M(p) - \hat{\beta}(\xrightarrow{p,t}) + \hat{\beta}(\xrightarrow{t,p})$$

*Dies wird mit* $M \xrightarrow{t,guard(t),\beta} M'$ *bezeichnet.*

**Definition 2.63** (Erreichbarkeitsmenge)**.**
*Sei* $SP_R = (\Sigma, E, R)$ *eine relationale algebraische Spezifikation.*
*Sei* $AN = (N, (h, M_0, \lambda, guard), E)$ *ein offenes algebraisches Petrinetz über* $SP_R$ *mit Platzmenge* $P$ *und Transitionsmenge* $T$*. Sei* $G$ *die* $SP_R$*-Guardmenge. Für eine Markierung* $M$ *ist die* ***Erreichbarkeitsmenge*** *bzgl.* $M$ *definiert als die Menge von Markierungen, die von* $M$ *aus erreichbar sind. Diese wird mit* $R_{AN}(M)$ *bezeichnet und es gilt:*

1. $M \in R_{AN}(M)$ *und*
2. *wenn* $M' \in R_{AN}(M)$ *und* $M' \xrightarrow{t,guard(t),\beta} M'', t \in T$*, dann gilt* $M'' \in R_{AN}(M)$*.*

**Definition 2.64** (Erreichbarkeitsgraph)**.**
*Sei* $SP_R = (\Sigma, E, R)$ *eine relationale algebraische Spezifikation.*
*Sei* $AN = (N, (h, M_0, \lambda, guard), E)$ *ein offenes algebraisches Petrinetz über* $SP_R$ *mit Platzmenge* $P$ *und Transitionsmenge* $T$*. Sei* $G$ *die* $SP_R$*-Guardmenge. Der* ***Erreichbarkeitsgraph*** *für* $AN$ *bzgl. der initialen Markierung* $M_0$ *ist ein kantenbeschrifteter gerichteter Graph über* $T \times G$*.*

*Er ist wie folgt definiert.*

$$RG_{AN}(M_0) = \{R_{AN}(M_0), Ed, M_0, l\} \text{ mit } \forall M \in R_{AN}(M_0).$$
$$(M \xrightarrow{t,guard(t),\beta} M' \implies ((M, M') \in Ed \wedge l((M, M')) = (t, guard(t)))$$

#### Zusammenfassung

Durch die Kombination aus Petrinetzen und relationalen algebraischen Spezifikationen ist es möglich mit algebraischen Petrinetzen sowohl den Kontrollfluss als auch den Datenfluss eines Services zu modellieren. Insbesondere die Nutzung der relationalen algebraischen Spezifikation ermöglicht eine Darstellung von Daten auf einer syntaktischen und damit generischen Ebene.

## 2.3 Computation Tree Logic (CTL)

Nachdem wir im letzten Abschnitt die Grundlagen der algebraischen Petrinetze, als Formalismus zur Modellierung des Service-Verhaltens eingeführt haben, betrachten wir nun *Computation Tree Logic* (*CTL*) als Spezifikationssprache für Verhaltenseigenschaften.

CTL wurde von Clarke et al.[CES86] entwickelt. Hierbei handelt es sich um eine angesehene temporale Logik, die häufig zur Spezifikation und Verifikation von Eigenschaften verteilter Systeme genutzt wird. CTL-Formeln bestehen grob gesagt aus Propositionen, logischen und temporalen Operatoren in Kombinationen mit Pfadquantifizierern. Insbesondere diese Kombination ermöglicht es, kausale Zusammenhänge zwischen Systemzuständen zu spezifizieren. Durch die Modellierung kausaler Zusammenhänge in Form eines Baumes und die Option alle Pfade gleichzeitig zu betrachten, ermöglicht CTL eine Analyse aller potentiellen Ausführungsschritte eines Systems. Dieser Punkt ist essentiell bei der Systemverifikation. In diesem Abschnitt führen wir basierend auf [CES86] die Syntax und Semantik von CTL ein.

### Syntax

Die Syntax der CTL-Formeln ist über atomaren Propositionen definiert. Eine *Proposition* ist eine Aussage, die entweder wahr ($TRUE$) oder falsch ($FALSE$) sein kann. Eine solche Aussage könnte zum Beispiel sein: `Es regnet und es ist bewölkt.` Der Wahrheitswert dieser Proposition, also ob die obige Aussage wahr oder falsch ist, hängt hierbei von den Wahrheitswerten der beiden (Teil-) Propositionen (`es regnet` und `es ist bewölkt`) ab. Bei jeder dieser beiden Propositionen handelt es sich um eine *atomare Proposition.*

**Definition 2.65** (Menge der CTL-Formeln)**.**
*Sei $AP_{CTL}$ eine Menge atomarer Propositionen. Die* ***Menge der CTL-Formeln*** *$\Phi_{CTL}$ über $AP_{CTL}$ ist induktiv wie folgt definiert.*

1. *Wenn $a \in AP_{CTL}$ , dann ist $a \in \Phi_{CTL}$.*
2. *Wenn $\varphi_1$ und $\varphi_2 \in \Phi_{CTL}$, dann gilt $\neg\varphi_1, \varphi_1 \wedge \varphi_2 \in \Phi_{CTL}$.*
3. *Wenn $\varphi_1$ und $\varphi_2 \in \Phi_{CTL}$, dann gilt $AX\varphi_1, EX\varphi_1, A[\varphi_1 U \varphi_2], E[\varphi_1 U \varphi_2] \in \Phi_{CTL}$.*

### Semantik

Die Semantik von CTL kann mit Hilfe eines gelabelten Transitionssystems definiert werden [CES86].

**Definition 2.66** (Knoten-Gelabeltes Transitionssystem)**.**
*Sei A eine endliche Menge. Ein über A* **(Knoten-)gelabeltes Transitionssystem (*LTS*)** *ist ein 4-Tupel $TS = (St, s_0, R, L)$ bestehend aus*

1. *einer endlichen* ***Menge von Zuständen*** *($St$),*
2. *einem* ***initialen Zustand*** *($s_0$),*
3. *einer linkstotalen Relation ($R \subseteq St \times St$), die die* ***Übergänge*** *zwischen den Zuständen definiert, und*
4. *eine* ***labeling Funktion*** *($L : St \to A$), die jedem Zustand des Systems ein Element aus A zuweist.*

Die Semantik von CTL ist durch Zustände und Pfade des zu Grunde liegenden LTSs definiert.

**Definition 2.67** (Pfad eines Gelabelten Transitionssystems)**.**
*Sei $TS = (St, s_0, R, L)$ ein (Knoten-) gelabeltes Transitionssystem. Ein **Pfad von TS** ist eine möglicherweise unendliche Folge von Zuständen $\sigma_{LTS} = (s_0, s_1, \dots)$, so dass*

$$\forall i \in \{0, 1, \dots\}.\ (s_i, s_{i+1}) \in R.$$

**Definition 2.68** (Semantik von CTL)**.**
*Sei $AP_{CTL}$ eine Menge atomarer Propositionen und sei $TS = (St, s_0, R, L)$ ein LTS. Sei $L : St \to \mathcal{P}(AP_{CTL})$ eine labeling Funktion, die jedem Zustand von $TS$ eine Menge atomarer Propositionen zuordnet. Sei $a \in AP_{CTL}$ und $\varphi, \varphi_1, \varphi_2 \in \Phi_{CTL}$. Eine CTL-Formel $\varphi$ ist in Zustand $s$ des LTS $TS$ erfüllt gdw.*

$$TS, s \models \varphi.$$

*Der Operator $\models$ ist induktiv wie folgt definiert:*

| | |
|---|---|
| $TS, s \models a$ | *für* $a \in L(s)$ |
| $TS, s \models \neg\varphi$ | *gdw. nicht*$(TS, s \models \varphi)$ |
| $TS, s \models \varphi_1 \wedge \varphi_2$ | *gdw.* $TS, s \models \varphi_1$ *und* $TS, s \models \varphi_2$ |
| $TS, s \models AX(\varphi)$ | *gdw. für alle Zustände* $s'$ *mit* $(s, s') \in R$ $TS, s' \models \varphi$ *gilt* |
| $TS, s \models EX(\varphi)$ | *gdw. ein Zustand* $s'$ *existiert, so dass* $(s, s') \in R$ *und* $TS, s_i \models \varphi$ *gilt* |
| $TS, s \models A(\varphi_1 U \varphi_2)$ | *gdw. für alle Pfade* $\sigma_{LTS} = (s_0, s_1, \dots)$ *gilt:* $\exists i \geq 0.\ (TS, s_i \models \varphi_2 \wedge \forall j \in [0, i).$ $TS, s_i \models \varphi_1)$ |
| $TS, s \models E(\varphi_1 U \varphi_2)$ | *gdw. ein Pfad* $\sigma_{LTS} = (s_0, s_1, \dots)$ *existiert, so dass* $\exists i \geq 0.\ (TS, s_i \models \varphi_2 \wedge$ $\forall j \in [0, i).\ TS, s_i \models \varphi_1)$ *gilt* |

**Definition 2.69** (Logische Äquivalenz)**.**
*Seien $\varphi_1, \varphi_2 \in \Phi_{CTL}$ und sei $TS = (St, s_0, R, L)$ ein LTS. $\varphi_1$ und $\varphi_2$ heißen **logisch äquivalent** in einem Zustand $s \in St$, gdw.*

$$TS, s \models \varphi_1 \Leftrightarrow TS, s \models \varphi_2$$

*Dies wird bezeichnet mit* $\varphi_1 \equiv \varphi_2$.

**Notation 2.70.** *Seien* $\varphi_1$ *und* $\varphi_2 \in \Phi_{CTL}$ *und sei* $TS = (St, s_0, R, L)$ *ein LTS mit* $L : St \to \mathcal{P}(AP_{CTL})$.

*Wir nutzen die folgenden Notationen [HR04].*

$\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2)$

$\varphi_1 \to \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$

$AF(\varphi_1) \equiv A(True\ U \varphi_1)$

$EF(\varphi_1) \equiv E(True\ U \varphi_1)$

$AG(\varphi_1) \equiv \neg EF(\neg\varphi_1)$

$EG(\varphi_1) \equiv \neg AF(\neg\varphi_1)$

*Die Notationen für die Operatoren* $\vee$ *und* $\to$ *entsprechen denen, die auch in der Propositionalen Logik gelten.*

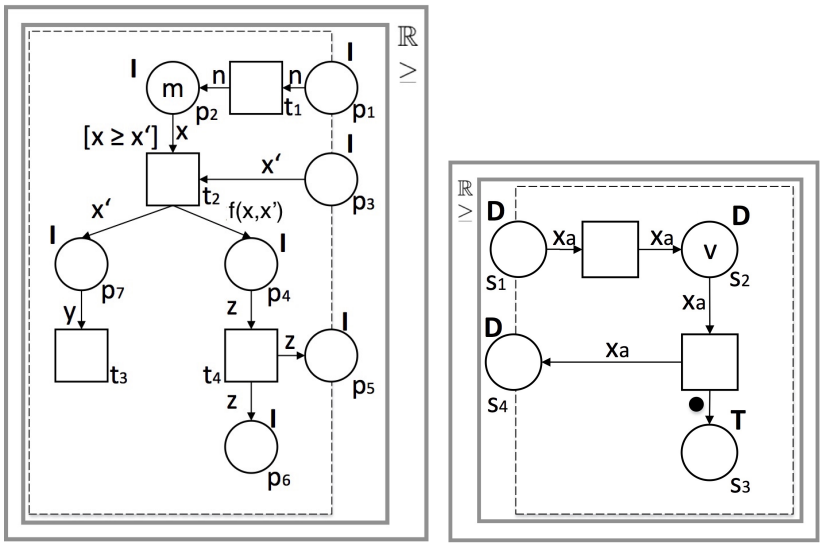
## 2.4 Laufendes Beispiel

Um unseren Ansatz für die automatisierte Controller-Synthese für Services mit Daten zu veranschaulichen, verweisen wir in dieser Arbeit wiederholt auf ein spezielles Beispiel, das wir in diesem Abschnitt vorstellen.

Wir betrachten hierbei ein System, bestehend aus einer Pumpe und einem Schalter. Beide Geräte benötigen die Zusammenarbeit mit der Umgebung um zu funktionieren. Das Hauptziel unseres Ansatzes ist es einen Controller zu synthetisieren, der die autonome Interaktion zwischen dem Schalter und der Pumpe ermöglicht, wobei bestimmte Verhaltenseigenschaften sichergestellt werden müssen. Diese Eigenschaften müssen vorher festgelegt werden.

Das Verhalten der Pumpe und des Schalters ist durch je ein APN modelliert. Diese sind in Abbildung 2.2 dargestellt. Die dazu gehörigen algebraischen Spezifikationen sind in Abbildung 2.3 angegeben. Den beiden algebraischen Netzen sind die reellen Zahlen und die darauf definierte Ordnung zu Grunde gelegt. Das bedeutet, die Interpretationen für die Relationssymbole $<, \leq, \geq$ und $>$ sind nicht frei wählbar, sondern sind so definiert, wie es von den reellen Zahlen her bekannt ist. Durch das zu Grunde legen der reellen Zahlen, die in der realen Welt bei medizinischen Geräten so nicht auftreten, ist es möglich, alle Datentypen die auf Zahlen beruhen, als Interpretation für die Sorten zu nutzen. Dabei bleibt die Interpretation der Relationssymbole erhalten. Im weiteren Verlauf verzichten wir auf die explizite Darstellung dieses Wrappers und beschränken uns auf die Repräsentation der Netzstruktur.

Um eine Intuition für die Funktionalität der Geräte zu bekommen, erläutern wir die einzelnen Netze im Folgenden detaillierter.




**(a)** Pumpe **(b)** Schalter

**Abbildung 2.2:** Laufendes Beispiel - Verhalten als APN

$SP_{\text{Pump}} = (\Sigma_{\text{Pump}}, E_{\text{Pump}})$

$\boldsymbol{\Sigma_{\text{Pump}}}=$

| | |
|---|---|
| **sorts:** | $I$ |
| **vars:** | $x, x', n, z, y \in X_I$ |
| **operations:** | $m : \rightarrow I$ |
| | $f : I \times I \rightarrow I$ |

$\boldsymbol{E_{\text{Pump}}} =$ $z = f(x,x')$, $y = x'$

**(a)** Pumpe

$SP_{\text{Actuator}} = (\Sigma_{\text{Actuator}}, E_{\text{Actuator}})$

$\boldsymbol{\Sigma_{\text{Actuator}}}=$

| | |
|---|---|
| **sorts:** | $D, T$ |
| **vars:** | $x_a \in X_D$ |
| **operations:** | $v : \rightarrow \text{D}$ |
| | $\bullet : \rightarrow \text{T}$ |

$\boldsymbol{E_{\text{Actuator}}} = \emptyset$

**(b)** Schalter

**Abbildung 2.3:** Laufendes Beispiel - Algebraische Spezifikationen

### Pumpe

Die Pumpe besteht aus einem Reservoir, das einen gewissen Anfangsfüllstand $m$ eines Medikamentes aufweist. In dem Netz wird dies als initiale Markierung des Platzes $p_2$ repräsentiert. Die Pumpe verfügt zudem über ein Interface (Interface-Eingangskanal $p_3$), worüber ihr die zu injizierende Menge $x'$ an Medizin mitgeteilt werden kann. Wenn die angegebene Menge $x'$ kleiner ist als Restmenge $x$ im Reservoir ($[x \geq x']$), dann wird die injizierte Menge intern gespeichert (durch das Markieren des Platzes $p_7$) und die sich noch im Reservoir befindende Menge aktualisiert (durch die Berechnung von $f(x, x')$)

Die aktualisierte Menge wird anschließend der Umgebung über den Interface-Ausgangsplatz $p_5$ mitgeteilt. Die Pumpe erreicht ihren Finalzustand, wenn der Platz $p_6$ markiert wird und alle Marken von den Interface-Plätzen konsumiert wurden. Um einen Überlauf des internen Speichers der Pumpe zu verhindern, können die Werte bzw. genauer gesagt die Marken von dem Platz $p_7$ entfernt werden, indem die Transition $t_3$ schaltet.

Um die Pumpe über den aktualisierten Füllstand zu informieren oder das Reservoir aufzufüllen und den Füllstand auf den Ausgangswert zurückzusetzen, kann der Interface-Platz $p_1$ genutzt werden. Diese Nachricht wird dann als initiale Markierung für den nächsten Durchlauf genutzt.

#### Schalter

Der Schalter sendet einen konstanten Wert, der als initiale Markierung $v$ auf dem Platz $s_2$ repräsentiert ist, an die Umgebung mittels des Interface-Platzes $s_4$. Der Schalter kann durch eine Nachricht über den Interface-Kanal $s_1$ zurückgesetzt werden. Dies kann jedoch, aufgrund der Kapazitätsangabe 1 für den Platz $s_2$, nur geschehen, wenn der Platz $s_2$ unmarkiert ist

#### Ziel

Für dieses Beispiel ist es unser Ziel, einen Controller zu synthetisieren, der

1. Werte, die durch den Schalter über den Interface-Kanal $s_4$ gesendet werden, an den Interface-Kanal $p_3$ der Pumpe weiterleitet und gegebenfalls den Typ der Nachricht anpasst,
2. eine Fehlermitteilung an die Umgebung sendet, sollte die Restmenge im Reservoir der Pumpe zu gering sein
3. und der die Pumpe und den Schalter zurücksetzt, nachdem die Fehlermitteilung geschickt und das Reservoir der Pumpe aufgefüllt wurde.

## 2.5 Zusammenfassung

In diesem Kapitel haben wir die grundlegenden Konzepte und Formalismen vorgestellt, auf denen unser Ansatz zur automatisierten Controller-Synthese für Services mit Daten beruht. Dazu haben wir zunächst die Basiskonzepte der *serviceorientierten Architekturen* vorgestellt, die die zu Grunde liegende Architektur der von uns betrachteten Systeme bildet. Anschließend haben wir *algebraische Petrinetze* als Formalismus zur Modellierung von Services mit Daten eingeführt, die sich insbesondere zur Darstellung von Kontroll- und Datenfluss von Systemen eignen. Zusätzlich haben wir die Syntax und Semantik von *CTL* präsentiert. Diese Logik nutzen wir im weiteren Verlauf als Grundlage zur Spezifikation von datenabhängigen Verhaltenseigenschaften. Abschließend haben wir ein Beispiel, bestehend aus einer Pumpe und eines Schalters, vorgestellt, auf das wir im Verlauf dieser Arbeit wiederholt verweisen.

# 3 Verwandte Arbeiten

Aufgrund der stetig steigenden Komplexität von Software-Systemen findet das Paradigma der serviceorientierten Architekturen immer mehr Anwendung. Damit einher geht auch eine verstärkte Forschung im Bereich automatisierter Service-Komposition. In diesem Kapitel diskutieren wir verwandte Arbeiten im Hinblick auf die Kriterien, die wir in Abschnitt 1.2.2 in Form von Forschungsfragen bereits erwähnt haben. Wir zeigen, dass bisher kein Ansatz existiert, der sich mit der automatisierten Controller-Synthese für Services mit Daten derart beschäftigt, dass komplexe, datenabhängige Eigenschaften sichergestellt werden. Dabei fordern wir von den existierenden Ansätzen die Handhabung von asynchroner Kommunikation sowie die generische und formale Modellierung und Analyse von Daten und Datenabhängigkeiten.

Im folgenden Abschnitt diskutieren wir zunächst verwandte Arbeiten im Bereich der automatisierten Service-Komposition (Abschnitt 3.1) aus verschiedenen Blickwinkeln.

Zusätzlich betrachten wir existierende Arbeiten, die sich mit dem Zusammenhang zwischen temporaler Logik und formalen Modellen beschäftigen (Abschnitt 3.2). Wir zeigen damit, dass aktuell keine Ansätze existieren, mit deren Hilfe wir CTL-Formeln in algebraische Petrinetze übersetzen können, was eine der Grundideen unseres Ansatzes ist. Abschließend diskutieren wir Möglichkeiten, Services zu modellieren und erläutern unseren Entschluss algebraische Petrinetze zur Repräsentation von Service-Verhalten zu nutzen (Abschnitt 3.3).

## 3.1 Automatisierte Service-Komposition

Der Forschungsbereich rund um automatisierte Service-Komposition ist bereits gut untersucht. Es existiert eine Vielzahl an Ansätzen, die in Überblickspapieren wie z.B. [RS05, DS05, BP10, DM12, SMKF12] zusammengefasst wurden. Der Fokus liegt dabei auf unterschiedlichsten Gesichtspunkten. Da in dieser Arbeit das Hauptaugenmerk auf medizinischen Geräten und deren Komposition liegt, betrachten wir den Spezialfall, dass die zu komponierenden Services gegeben sind und die Interoperabilität mittels einer weiteren Komponente realisiert wird. Daher betrachten wir in diesem Abschnitt nur Arbeiten, die sich

mit der Komposition von Services durch einen Controller bzw. Mediator beschäftigen.

### 3.1.1 Komposition mit Datenbehandlung auf Protokollebene

Bei der Komposition verschiedener, unabhängig voneinander entwickelter Komponenten treten unterschiedliche Probleme auf. Das in der Literatur am häufigsten betrachtete Problem sind sichtbar werdende Interface-Inkompatibilitäten. In diesem Abschnitt stellen wir exemplarisch einige dieser Ansätze aus verschiedensten Anwendungsbereichen vor.

#### Komponenten-Basierte Software Entwicklung

Die Kapselung von Funktionalität in einzelne Komponenten, die je nach Bedarf miteinander komponiert werden können, um eine komplexere Funktionalität zu erreichen, ist der grundlegende Gedanke serviceorientierter Architekturen. Neben anderen Charakteristiken ist dies auch die Grundidee der Komponenten-basierten Software Entwicklung (engl.: *Component-Based Software Engineering* (*CBSE*)) [HC01].

In dieser Domäne ist die Komponenten-Adaption ein kritischer Part, weswegen viele Ansätze zur Synthese von Adaptern bzw. Controllern entworfen wurden.

Yellin and Storm [YS97] entwickelten beispielsweise einen Controller-Synthese Prozess basierend auf endlichen Automaten. Dieser Ansatz behandelt allerdings nur die Protokollebene und betrachtet nur erlaubte Typen der ausgetauschten Nachrichten. Sie abstrahieren vom datenabhängigen Verhalten und können nur Deadlock-Freiheit als Eigenschaft des komponierten Systems, bestehend aus den Komponenten und dem Adapter, garantieren.

Bracciali et al. [BBC02, BBC05] erweiterten diesen Blick auf die Verhaltensebene. Sie präsentieren einen auf dem $\pi$-Kalkül basierenden Controller-Synthese-Prozess. Als Grundidee für diesen Ansatz entwickelten sie Interaktionsmuster [BBT01], die generisch Kommunikationskontexte in Form von Kanalnamen definieren. Diese sind nach außen sichtbar und ermöglichen die Interaktion mit der Komponente. Die Kontexte werden zur Definition des Datenaustausches genutzt. Darauf aufbauend können Paarungsregeln spezifiziert werden, die angeben, welche Kanäle untereinander verbunden werden sollen. Basierend auf diesen Regeln wird mit ihrem Ansatz ein Controller synthetisiert. Dabei abstrahieren Bracciali et al. jedoch von datenabhängigem Verhalten und stellen nur Deadlockfreiheit als Kompositionseigenschaft sicher.

Als zusätzliche Erweiterung entwickelten Canal und Poizat [CPS08] einen Algorithmus, der in der Lage ist, Paarungen von Kanälen zu erkennen, die ein Fehlverhalten hervorrufen und dieses zu behandeln. Ihr Ansatz basiert auf einem gelabelten Transitionssystem und synchronen Vektoren. Diese Vektoren

definieren die Kommunikation zwischen Komponenten durch die Angabe der Events, die die Ausführung der jeweiligen Komponenten forcieren. Mit ihrem Ansatz ist es möglich, Interface-Inkompatibilitäten, wie unterschiedliche Eventnamen, inkompatible Typen oder fehlende Empfängerkanäle zu behandeln und so die Deadlock-Freiheit der Komposition zu garantieren. Sie betrachten jedoch Daten weder generisch noch formal, sodass ihr Ansatz nicht für weitere, möglicher Weise datenabhängige Anforderungen genutzt werden kann.

Es existieren noch weitere Ansätze im Bereich der automatisierten Controller-Synthese auf die wir hier nicht weiter eingehen wollen (z.B. [Reu01, SR02, PV02]). Dennoch ist, nach unserem Wissen, keine der existierenden Arbeiten dazu geeignet, datenabhängige Eigenschaften während des Controller-Synthese-Prozesses zu garantieren.

#### Workflows

Wie in beispielsweise [YPVdH02] gezeigt wurde, existiert eine Beziehung zwischen dem SOA-Paradigma und Geschäftsprozessen. Letztere werden e-Services genannt und vorrangig als Workflows modelliert. Zwischen der Komposition von e-Services und dem klassischen Workflow-Management existieren jedoch signifikante Differenzen. Um diese Differenzen zu bewältigen haben Mecella et al. [MPP02] ein Framework entwickelt, mit dem eine Komposition von e-Services hinsichtlich der Koordination und Manipulation von ausgetauschten Nachrichten ermöglicht wird. In ihrer Arbeit realisieren sie die Komposition von verschiedenen e-Services mit sogenannten Orchestrierungs-Netzen, die auf (gefärbte) Petrinetze zurückzuführen sind. Auch wenn dieser Ansatz vielversprechend erscheint, so betrachtet er weder die Daten der ausgetauschten Nachrichten noch behandelt er Verhaltenseigenschaften des Systems mit Ausnahme der korrekten Weiterleitung von Nachrichten.

Schmidt präsentiert in [Sch05] einen Ansatz zur Komposition von verschiedenen Workflows basierend auf der Überprüfung ihrer Bedienbarkeit. Dafür erweiterte er Workflow-Netze, die eine Spezialisierung von Petrinetzen darstellen, um Interfaces, die zur Modellierung der (asynchronen) Kommunikation genutzt werden können. Basierend auf diesen offenen Workflow-Netzen wird ein Controller in Form eines Automaten synthetisiert. Dies geschieht durch die Berechnung von Ausführungsschritten, die das komponierte System vollziehen kann. Der dadurch resultierende Controller kann als Ausführungspartner betrachtet werden und koordiniert die Reihenfolge der ausgetauschten Nachrichten. Daten und datenabhängiges Verhalten werden auch hier nicht betrachtet.

#### Web-Services

Die Idee Bedienbarkeit durch die Synthese eines Kommunikationspartners zu überprüfen, findet nicht nur im Bereich der Workflows sondern auch bei Web-Services Anwendung [AMSW09, GMW12]. In diesen Ansätzen wird basie-

rend auf Services und einer Menge von elementaren Aktivitäten ein Controller synthetisiert, sodass das komponierte System Deadlock-frei interagieren kann. Hierbei ist das Verhalten der Services mit Hilfe von Petrinetzen modelliert. Bei den elementaren Aktivitäten, die unterstützt werden, handelt es sich um das Erzeugen, Kopieren, Löschen, Aufspalten und Verschmelzen von ausgetauschten Nachrichten. Damit können eine asynchrone Kommunikation gewährleistet und auftretende Interface-Inkompatibilitäten behandelt werden.

Ein weiterer regelbasierter Synthese-Ansatz für Web-Services wurde in [WDOV08] veröffentlicht. Basierend auf endlichen Automaten definiert dieser Ansatz einen Prozess zur Synthese eines Adapters, der eine Deadlock- und Nachrichtenverlust-freie Komposition von Web-Services sicherstellt. Zur Modellierung des Verhaltens des Adapters haben die Autoren eine operationale Semantik entwickelt. Diese umfasst einen Anpassungszyklus, einen Auswahlprozess von Paarungsregeln und eine Anwendungsreihenfolge dieser Regeln. Damit können sie auftretende Interface-Inkompatibilitäten beheben und Nachrichten koordinieren.

Nezhad et al. betrachten in ihrer Arbeit [MNBM+07] ebenfalls Interface-Inkompatibilitäten und Nachrichtenkoordination. Sie stellen einen auf endlichen Automaten beruhenden Ansatz vor, mit dem Fehlpaarungen von Kanälen auf Protokollebene automatisch identifiziert werden können. Zudem kann damit ein Adapter generiert werden um diese Fehlpaarungen zu behandeln. Zusätzlich präsentieren sie einen Formalismus, genannt mismatch tree, der genutzt wird, um Deadlock-Situationen zu erkennen und um Vorschläge zu unterbreiten, wie diese Situationen umgangen werden können, z.B. durch eine Neuordnung der ausgetauschten Nachrichten.

Dumas et al. betrachten in [DSW06] neben anderen Interface-Anpassungen wie Speicherung, Abfangen und Veränderung, ebenfalls Neuanordnungen von Nachrichten. Um einen Mediator zu synthetisieren haben sie eine Algebra bestehend aus sechs Operationen definiert, die die Interface-Anpassungen ermöglichen. Zusätzlich stellen sie eine graphische Repräsentation zur Verfügung, die eine Spezifikation von geforderten Interfaces mit Hilfe von algebraischen Ausdrücken erlaubt und diese mit den zur Verfügung stehenden Interfaces in Verbindung setzt. Die algebraischen Ausdrücke dürfen dabei jedoch nur die von ihnen in der Algebra definierten Operatoren beinhalten.

All diese Ansätze bieten keine formale Datenbehandlung, weswegen Services mit Daten nicht modelliert und datenabhängige Verhaltenseigenschaften nicht sichergestellt werden können.

### 3.1.2 Komposition unter Beachtung Datenabhängigen Verhaltens

Bisher haben wir exemplarisch Ansätze diskutiert, die sich mit Interface-Anpassungen im Sinne der Modifikation, Neuanordnung, Typänderung und Koordination von Nachrichten ohne Datenbehandlung beschäftigen.

In diesem Abschnitt präsentieren wir die wichtigsten Ansätze, die die Komposition von Services mit Daten untersuchen.

Bernardi et al. stellen in ihrer Arbeit [BCDG+05a] ein Framework vor, das colombo genannt wird. Damit ist es möglich einen Controller unter Beachtung datenabhängigem Verhaltens zu synthetisieren. Die Daten können dabei unendlich viele Werte annehmen. Das Verhalten der zu komponierenden Web-Services ist hierbei als mit Guards versehendem Automaten modelliert. Ihr Synthese-Prozess stellt sicher, dass das komponierte System, bestehend aus den Web-Services und dem Controller, das Verhalten eines vordefinierten Zielservices simuliert. Dieser Zielservice spezifiziert das gewünschte Verhalten eines zusammengesetzten Services, das mit demselben Service-Nutzer interagiert wie das mit dem Controller komponierte System. Um dies für alle potentiellen Service-Nutzer sicherzustellen, haben sie zunächst ein Weltmodell eingeführt, das als Datenbankschema verstanden werden kann. Zusätzlich erlauben sie die Spezifikation von Constraints, die die Art und Berechtigung des Zugriffes auf das Datenbankschema festlegen. Das heißt, sollte ein Web-Services auf einen speziellen Eintrag in dem Datenbankschema zugreifen wollen, so muss er hinsichtlich der Constraints den korrekten Wert senden. Das Senden von Nachrichten sowie ihre Auswirkungen auf das Verhalten der Services und des Datenbankschemas ist durch atomare Prozesse realisiert. Diese atomaren Prozesse bestehen aus einer Eingabe- und Ausgabesignatur sowie einer bedingten Auswirkung. Eingabe- und Ausgabe Signaturen sind Mengen von getypten Variablen. Bei bedingten Auswirkungen handelt es sich um Mengen von Paaren, die die Ausgabe des Prozesses beschreiben, welche erwartet wird, wenn die Eingabe die entsprechende Bedingung erfüllt. Abhängig von diesen bedingten Auswirkungen, wird ein Service aus einer gegebenen Menge von verfügbaren Services ausgewählt und mit dem betrachteten Service komponiert. Da der Service-Nutzer nicht-deterministisch Nachrichten lesen kann, ist jeder Service mit einem lokalen Speicher versehen. Die bedingte Verzweigung, die der Controller aufweist, ist abhängig von den Werten der Variablen im lokalen Speicher und dem Zustand des Weltschemas. Dadurch wird ein unendliches Transitionssystem konstruiert, da die Werte aus einem unendlichen Wertebereich gewählt werden können. Um dies zu handhaben, stellen sie eine Abstraktionstechnik bereit, die es ermöglicht, mit einer endlichen Menge symbolischer Variablen statt mit unendlich vielen konkreten Werten zu arbeiten. Dazu werden Klassen von Werten mit gleichen Auswirkungen gebildet. Dies führt zu einem symbolischen Ausführungsbaum mit einer beschränkten Anzahl von Verzweigungen. Abschließend stellen sie die Verbindung zwischen dem symbolischen Ausführungsbaum und der *Propositional-Dynamischen Logik (engl. propositional dynamic logic)* [HKT00, BCDG+05b] vor. Sie nutzen diese Logik um alle Verhaltensmöglichkeiten der Services und ihrer Kommunikation zu spezifizieren. Diese Kommunikation ist durch die ausgetauschten Nachrichten, ihre Werte und ihre Reihenfolge definiert und entspricht genau dem Verhalten des gewünschten Controllers. Dadurch und aufgrund der Nähe zum symbolischen Ausführungsbaum, könnte der Controller direkt aus den Formeln extrahiert werden. Dies kann jedoch nur realisiert werden, wenn die Formel erfüllbar ist. Um die Erfüllbarkeit einer solchen Formel zu überprüfen, muss ein System konstruiert und dahingehend analysiert werden, ob die Negation der Formel

verletzt wird [FL79]. Da wir einen Controller direkt aus den gegebenen Formeln synthetisieren und dieser per-Konstruktion korrekt ist, benötigt unsere Berechnung unter den gleichen Annahmen (begrenzte Anzahl von Nachrichten im lokalen Speicher und begrenzte Anzahl symbolischer Variablen) nie länger als ihre Berechnung, sondern ist in den meisten Fällen schneller. Des Weiteren erlaubt der Einsatz der Propositional-Dynamischen Logik nur zustandsbasierte Bedingungen, während wir auch pfadbasierte Bedingungen unterstützen. Dies erreichen wir durch die Nutzung einer Untermenge von CTL.

Wie wir bereits in Abschnitt 3.1.1 erwähnt haben, könnte die Idee der Partnersynthese zur Generierung eines Controllers genutzt werden. Dazu würde man die Vereinigung der zu komponierenden Services als einen Service betrachten und einen Partner synthetisieren, der mit diesem Deadlock-frei kommunizieren kann. Bisher haben wir nur Ansätze zur Partnersynthese betrachtet, die vollständig von Daten und datenabhängigem Verhalten abstrahieren. Das Thema Daten tritt mittlerweile jedoch immer mehr in den Fokus.

So entwickelten Lohmann und Wolf [LW11] einen Ansatz zur überprüfung der Bedienbarkeit eines Services mit Daten durch Partnersynthese. Basierend auf algebraischen Petrinetzen synthetisieren sie dabei einen symbolischen Partner. Dazu konstruieren sie zunächst einen *symbolischen Erreichbarkeitsgraphen* (*SRG*) des Netzes, der das gesamte mögliche Serviceverhalten abbildet. Während dieser Konstruktion können Relationen zwischen Variablen auftreten, die den Service daran hindern einen Endzustand zu erreichen. Diese Relationen werden als Constraints gespeichert. Darauf basierend werden alle Pfade aus dem SRG eliminiert, die zu einem Deadlock führen. Die übrig gebliebenen Pfade im SRG zusammen mit den gespeicherten Constraints beschreiben das valide Verhalten und bilden somit den symbolischen Kommunikationspartner des betrachteten Services. Letzterer gilt als bedienbar, wenn es eine Interpretation der zum Netz gehörigen algebraischen Spezifikation gibt, sodass die Constraints erfüllt sind.

Zwei sehr ähnliche Ansätz wurden zur gleichen Zeit von Wagner entwickelt, die auf Services mit Daten ausgelegt sind und Deadlockfreiheit garantieren [Wag11, Wag12]. Beide Ansätze basieren auf *gefärbten Petrinetzen* (engl.: *Colored Petri Nets* (*CPNs*)) und konzentrieren sich auf das Erkennen und Eliminieren von Deadlock-Situationen, die durch datenabhängige Bedingungen hervorgerufen werden und den Service an einer weiteren Ausführung hindern. Wie auch Lohmann und Wolf konstruiert er einen *symbolischen Erreichbarkeitsgraphen (SRG)* der zur Erkennung von Deadlocks genutzt wird, die im Design des Geschäftsprozesses auftreten können [Wag11]. Pfade auf denen ein Deadlock gefunden wurde, werden durch ihn hinsichtlich möglicher Wertebeschränkungen analysiert. Dies resultiert in Vorschlägen zur Reparatur des Prozesses unter Angabe notwendiger Guards und weiterer Aktivitäten, genauer gesagt Transitionen. Zusätzlich dazu kann die Deadlock-Analyse genutzt werden um Bedienbarkeit eines gegebenen Services zu entscheiden [Wag12].

Obwohl die Kombination der Ansätze von Wagner und Lohmann/Wolf sehr vielversprechend hinsichtlich der Synthese eines korrekten Controllers für Services mit Daten in unserem Problemsetting wirkt, unterstützen sie nicht die

Möglichkeit andere Verhaltenseigenschaften als Deadlock-Freiheit sicherzustellen. Nichts desto trotz gäbe es die Option zunächst einen Partner zu synthetisieren, diesen mit den Services zu komponieren und anschließend die gegebenen Verhaltenseigenschaften formal zu verifizieren. Im Vergleich zu unserem Ansatz, der per-Konstruktion korrekt ist, würde dies jedoch in den meisten Fällen einen viel höheren und nie einen geringeren Berechnungsaufwand bedürfen. Im Gegensatz zu den eben vorgestellten Ansätzen synthetisieren wir einen Controller direkt aus den gegebenen, sicherzustellenden Eigenschaften. Somit müssen wir nicht das gesamte System hinsichtlich dieser Eigenschaften überprüfen, sondern analysieren schon während der Konstruktion die Teilsysteme auf Widersprüche.

Vor Kurzem stellten Belkhir et al. [BCR15] einen Ansatz zur Berechnung eines Mediators vor, der die Kommunikation zwischen verschiedenen, verfügbaren Services und eines gegebenen Service-Nutzers ermöglicht. In ihrer Arbeit fokussieren sie sich auf die Komposition von datenbehafteten Services. Diese Komposition ist abhängig von den ausgetauschten Datenwerten, die aus einem unendlich großen Wertebereich gewählt werden können. Um diese unendliche Dimension zu handhaben, definieren sie eine neue Struktur, die parametrisierte Automaten genannt wird. Dies ist eine Erweiterung von endlichen Automaten, bei denen die Transitionen mit Zeichen, Variablen oder Guards beschriftet sind. Hierbei können Variablen mit Werten aus einem unendlichen Wertebereich, also aus einer unendlichen Menge von Zeichen, belegt werden. Guards hingegen können Gleichungen und Ungleichungen über der Menge der Zeichen sein. Diese Automaten werden von Belkhir et al. zur Spezifikation des Verhaltens der Services und des Service-Nutzers angewendet. Die Grundidee ihres Ansatzes ist es, eine Gewinnstrategie zu berechnen, die eine korrekte Kommunikation zwischen den Services und dem Service-Nutzer realisieren kann. Daraus leiten sie anschließend den Mediator ab. Dazu bilden sie zunächst das asynchrone Produkt der Services. Darauf und auf dem parametrisierten Automaten, der das Verhalten des Service-Nutzers repräsentiert, aufbauend, wird ein symbolisches Simulationsspiel berechnet. Dieses Spiel wird anschließend durch die Instanziierung der Variablen konkretisiert. Die Instanziierung wird durch die Zuweisung von Werten aus einem endlichen Wertebereich an Variablen realisiert. Der endliche Wertebereich wurde zuvor aus dem unendlichen Wertebereich berechnet. Basierend auf dem so konkretisierten Simulationsspiel wird eine Gewinnstrategie für den Widerleger bestimmt. Das bedeutet, dass sie Annehmen, dass die Kommunikation schief läuft, z.B. dass Nachrichten an den falschen Service gesendet oder Constraints verletzt werden. Sie berechnen anschließend eine Gewinnstrategie für den Spieler, der diese Annahme nicht erfüllt. Wird eine solche Strategie gefunden, so impliziert dies, dass es eine Transitionsreihenfolge gibt, bei der die Kommunikation korrekt funktioniert. Diese Reihenfolge beschreibt das Verhalten des gesuchten Mediators. Obwohl sie in ihrem Ansatz eine symbolische Datenbehandlung definieren, kann nur die korrekte, möglicher Weise datenabhängige Weiterleitung von Nachrichten hinsichtlich gegebener Constraints sichergestellt werden. Im Gegensatz dazu können wir durch die Nutzung von CTL mehr (datenabhängige) Verhaltensanforderungen garantieren. Belkhir et al. geben in ihrer Arbeit jedoch an, dass es durch kleine Anpassungen ihres Algorithmus' möglich ist, alle potentiellen

Mediatoren zu berechnen. Dies könnte man nutzen um zu analysieren, ob mindestens einer der Mediatoren alle datenabhängigen Eigenschaften erfüllt. Dies wäre der gesuchte Controller. Jedoch würde dieser Ansatz einen sehr hohen Berechnungsaufwand benötigen, der weit über dem unseres Ansatzes liegt.

So weit wir wissen, existiert kein Ansatz, der alle unsere Anforderungen erfüllt. Insbesondere wird ein *datenabhängiger* Controller-Synthese-Prozess, durch den komplexe (datenabhängige) Eigenschaften sichergestellt werden können, nicht durch existierende Arbeiten zur Verfügung gestellt. Die wenigen in der Literatur zu findenden Ansätze, die Anpassungspotential haben, würden zu viel Berechnungsaufwand mit sich bringen.

## 3.2 Temporal-Logische Formeln als Formale Modelle

Im vorherigen Abschnitt haben wir bestehende Ansätze diskutiert, die sich mit der Komposition von Services mit Daten beschäftigen. Da die Grundidee unseres Ansatzes darin besteht, einen Controller direkt aus den gegebenen Verhaltensanforderungen und den gegebenen Services zu synthetisieren, stellen wir in diesem Abschnitt existierende Ansätze vor, die sich mit der Konstruktion formaler Modelle, wie Automaten und Petri-Netze, aus einer Formel befassen.

In diesem Bereich existieren einige Forschungsarbeiten, die sich hauptsächlich mit der *Lineare Temporale Logik* (*LTL*) Synthese beschäftigen. Die Grundidee ist es, aus einer logischen Spezifikation, die als LTL-Formel gegeben ist, automatisiert Systementwürfe zu konstruieren, wie beispielsweise faire diskrete Systeme [KV00], reaktive Designs [PPS06] oder Hardware-Designs [BGJ$^+$07].

Bei den Ansätzen, die sich mit LTL-Synthese beschäftigen, wird die LTL-Spezifikation verwendet, um einen Büchi-Automaten zu konstruieren (z.B. [DGV99, SB00]). Dieser Automat repräsentiert alle Modelle, die die entsprechenden LTL-Formeln erfüllen.

Wie beispielsweise in [SB00] beschrieben, wird eine LTL-Formel $\varphi$ zunächst in eine positive Normalform überführt. Danach werden iterativ für jedes atomare Element der Formel Zustände konstruiert, die in der positiven Normalform angegeben ist. Die Zustände werden dann mit der atomaren Proposition (oder ihrer Negation) gelabelt. Abschließend werden die Übergänge in Übereinstimmung mit der Semantik der temporalen Operatoren definiert, die in der Formel auftreten.

Dieser Ansatz wurde mit der Zeit optimiert, um die Anzahl der Zustände in dem resultierenden Automaten zu reduzieren, beispielsweise durch Umstellung der Formel [GO01, BKŘS12]. Außerdem wurde der Ansatz auf alle LTL-Formeln [JB06], auf *Property Specification Language* (*PSL*), eine Logik, die LTL enthält [BGJ$^+$07] und auf das $\mu$-Kalkül [AMM14] erweitert.

Obwohl dieser Bereich der Forschung gut untersucht ist, sind diese bestehenden Ansätze kaum an unsere Bedürfnisse anpassbar. Durch die von uns zugelassene Spezifikation des Systems mittels CTL-Formeln wird eine weitere Dimension in den Synthese-Prozess integriert. Mit den bestehenden Ansätzen ist es nicht möglich, existenzielle und universelle Temporale-Pfad-Operatoren darzustellen. Zudem eignen sich Büchi-Automaten nicht zur Darstellung von Daten und deren Modifikation. Somit wäre es sehr schwierig, falls es möglich ist, die oben aufgeführten Ansätze anzupassen.

Da CTL-Formeln nicht nur über einen Pfad in einem System argumentieren, sondern auch Aussagen über sich verzweigende Pfade erlauben, ist es nicht möglich, die Ansätze, die für LTL entwickelt wurden, zu verwenden. In diesem Zusammenhang haben Kupferman et al. [KVW00] einen Ansatz entwickelt, der es ermöglicht, CTL-Formeln in einen schwach-alternierenden Automaten (engl.: Weak Alternating Automaton) zu übersetzen. Sie geben an, dass sich dieses Verfahren zur Verifikation eines Systems bezüglich CTL-Formeln eignet. Der Grund dafür ist, dass diese Übersetzung in linearer Zeit durchgeführt werden kann. Zur Verifikation übersetzt man das Komplement der Formel in einen solchen Automaten, bildet den Produktautomaten und löst das nicht-Leerheits-Problem. Ohne an dieser Stelle ins Detail zu gehen, ist dieser Ansatz für unsere Zwecke nicht anpassbar, da es extrem schwierig wäre, verschiedene Datentypen und datenabhängige Propositionen zu integrieren. Außerdem ist, erlauben wir, dass mehrere Services eine Reihe von Nachrichten verschicken. Würden wir den Ansatz von Kupfermann adaptieren, so müssten wir eine große Menge an Automaten konstruieren und darüber den Produktautomaten bilden. Dies würde den Berechnungsaufwand des Controller-Synthese-Prozesses immens erhöhen. Dieser Punkt wird ausführlicher in Abschnitt 3.3 diskutiert.

Soweit wir wissen, gibt es bisher keinen Ansatz, der die Übersetzung von CTL-Formeln in (High-Level) Petri-Netze ermöglicht. Jedoch ist eine derartige Übersetzung für unser Synthese-Verfahren notwendig, da das Verfahren in einem APN resultieren muss, der den Controller repräsentiert. Denn nur so ist die Komposition mit den Service-Netzen möglich.

Wir wollen nicht verschweigen, dass es eine (nur lose verbundene) Alternative zu einer direkten Übersetzung von CTL-Formeln in formale Modelle gibt. Diese ist die Nutzung von Planungsansätzen [PBT14, SG13]. Ihre Anwendung führt zu Plänen, die die Ausführungsreihenfolge und die Bedingungen der Übergänge beschreiben, sodass die Formel in dem betrachteten System erfüllt ist. Da diese Pläne als Transitionssysteme betrachtet werden können, ist eine Übersetzung in jedes formale Modell möglich, dessen Semantik auf Transitionssystemen definiert ist. Da wir jedoch Model-Checking-Techniken aufgrund des hohen Rechenaufwand umgehen wollen, finden diese Arbeiten in unserem Ansatz keine Beachtung.

## 3.3 Modellierung von Services

Es gibt eine Vielzahl formaler Modelle, die verwendet werden können, um Service- Verhalten zu modellieren, wie z.B. in [Gro14, Jen91, Rei91, Rei10] dargestellt wird. Um Services mit Daten und deren Interaktion adäquat zu modellieren, ist es notwendig, dass der Formalismus der Wahl ausdrucksstark genug ist. Darüber hinaus ist es, aufgrund der beabsichtigten Verwendung unseres Ansatzes im Bereich der Medizin wünschenswert, dass der verwendete Formalismus leicht verständlich und intuitiv zu bedienen ist. Dies erhöht die Akzeptanz unseres Ansatzes. Eine weitere Notwendigkeit ist die Unterstützung der formalen Datenbehandlung. Da wir die Nutzbarkeit unseres Ansatzes weiter erhöhen wollen, ist es wünschenswert, dass der gewählte Formalismus eine kompakte Darstellung ermöglicht.

Zusammenzufassend, muss der mögliche Formalismus also die folgenden Kriterien erfüllen.

1. Große Ausdrucksstärke
2. Unterstützung verteilter Systeme
3. Intuitive Verständlichkeit
4. Datenrepräsentation
5. Kompaktheit

Die folgenden Formalismen sind sehr vielversprechend für die Erfüllung dieser Kriterien.

1. *Business Process Management Notation* (*BPMN*)[Gro14],
2. $\pi$-Kalkül [MPW92, Mil99],
3. Eingabe/Ausgabe-Automaten (engl.: I/O Automata) [GL00]
4. High-Level Petrinetze [Jen91, Rei91, Rei10]

Obwohl BPMN [Gro14] der meist akzeptierte Formalismus zur Modellierung von Geschäftsprozessen ist, ist er weder gut für verteilte Systeme noch für die Beschreibung von Geräteverhalten, die gerade im medizinischen Bereich wichtig ist, geeignet. Insbesondere ist es sehr schwierig, BPMN hinsichtlich unterschiedlicher Nachrichten mit verschiedenen Typen anzupassen. Aus diesen Gründen haben wir uns gegen die Anwendung dieses Formalismus' entschieden.

Das $\pi$-Kalkül [MPW92], als Verallgemeinerung der herkömmlichen Prozess-Algebren, ist besser geeignet, um verteilte Systeme zu modellieren. Jedoch weist dieser Formalismus einen erheblichen Nachteil auf. Mit dem $\pi$-Kalkül können Prozesse nur abstrakt beschrieben werden [VdA05]. Für Benutzer ohne Wissen auf dem Gebiet der formalen Verifikation ist es oft schwierig, diese Art von Modellen zu verstehen.

Im Gegensatz dazu, bieten Eingabe/Ausgabe-Automaten und High-Level Petrinetze eine grafische Darstellung, die leicht zu verstehen ist. Gleichzeitig sind beide Formalismen in der Lage Daten angemessen zu repräsentieren. CPN [Jen91] als eine Form der High-Level Petrinetze bietet dabei nur eine spezifische Instanziierung der Daten. Da wir darauf abzielen, Daten generisch zu behandeln, sodass unser Ansatz für alle Datentypen anwendbar ist, sind CPNe für unseren Ansatz nicht geeignet. Darüber hinaus haben Eingabe/Ausgabe-Automaten im Vergleich zu APN [Rei91], den signifikanten Nachteil, dass eine Behandlung mehrerer Nachrichten eine Einführung mehrerer Automaten bedarf. Für diese muss das Produkt berechnet werden, um über das gemeinsame Verhalten Aussagen treffen zu können. Diese Situation ist viel einfacher und übersichtlicher zu handhaben, wenn Petrinetze verwendet werden, da hierbei einfach eine neue Marke für jede Nachricht erzeugt werden kann.

Zusammenfassend können wir sagen, dass sich algebraische Petrinetze für unseren Ansatz und den vorgesehenen Einsatzbereich am besten eignen, um das Verhalten der Services und deren Interaktion zu modellieren. Grund dafür sind ihre große Ausdrucksstärke, die Fähigkeit Prozesse und Geräteverhalten zu modellieren und die Möglichkeit verteilte Systeme zu repräsentieren. Gleichzeitig sind sie einfach zu verstehen, ermöglichen eine generische Datenbehandlung und bieten eine (grafische) Darstellung, die kompakter ist als die der Eingabe/Ausgabe-Automaten.

## 3.4 Zusammenfassung

In diesem Kapitel haben wir existierende Arbeiten diskutiert, die in Zusammenhang mit unserem Ansatz stehen. Dafür haben wir zunächst Lösungen für die automatisierte Service-Komposition vorgestellt, die sich mit dem Austausch von Daten beschäftigen. Insbesondere in Abschnitt 3.1.2 haben wir gezeigt, dass es derzeit keinen Ansatz gibt, der Daten und datenabhängiges Verhalten angemessen berücksichtigt bzw. der (datenabhängige) Eigenschaften ohne Model-Checking sicherstellen kann. Model-Checking ist hinsichtlich der Berechnungszeiten sehr ineffizient, sodass der Berechnungsaufwand weit über dem unseres Ansatzes liegen würde.

Im Anschluß daran haben wir Möglichkeiten diskutiert, temporal-logische Formeln, deren Ausdrucksstärke der von uns definierten Teilmenge von CTL-Formeln entspricht, als operationale Modelle darzustellen. Insbesondere haben wir gezeigt, dass zwar eine Reihe von Ansätzen existiert, die sich mit der Synthese von Automaten aus LTL -Formeln beschäftigen, aber nur wenige, die eine Lösung für CTL-Formeln bieten. Die Diskussion führte zu der Schlussfolgerung, dass es zur Zeit keinen Ansatz gibt, der die Übersetzung von CTL-Formeln in (High-Level) Petrinetze ermöglicht.

In Abschnitt 3.3 haben wir Möglichkeiten der Service-Modellierung diskutiert. Wir haben erklärt, warum der potentielle Formalismus die Kriterien *große Ausdruckskraft, die Eignung für verteilte Systeme, intuitive Verständlichkeit, Datendarstellung und Kompaktheit* erfüllen muss. Auf dieser Basis

haben wir BPMN, $\pi$-Kalkül, Eingabe/Ausgabe-Automaten und High-Level Petrinetze (CPNs und APNs) verglichen. Dies führte zu der Beobachtung, dass APNs der am besten geeignete Formalismus für unsere Kriterien und damit für unseren Ansatz ist.

# Teil 2
# Grundlegende Formalismen

# 4 Definition von RCTL-Formeln und RCTL-Netzen

## 4.1 Eingeschränkte CTL (RCTL)

Wie bereits in vorherigen Kapiteln erwähnt, findet CTL sehr große Akzeptanz bei der Spezifikation und Verifikation von Verhaltenseigenschaften verteilter Systeme. Wir nutzen dies, beschränken uns jedoch dabei auf eine bestimmte Untermenge, mit der Auswirkungen auf zukünftige Zustände des Systems spezifiziert werden, die eintreten sobald sich das System in einem gewissen Zustand befindet. Diese eingeschränkte Untermenge von CTL bezeichnen wir im weiteren Verlauf als *RCTL*.

### 4.1.1 Syntax von RCTL

Für die Spezifikation datenabhängiger Verhaltenseigenschaften benötigen wir zunächst die Möglichkeit, Beschränkungen der Datenwertebereiche, den Datenfluss und Datenabhängigkeiten auf Ebene atomarer Propositionen auszudrücken.

**Definition 4.1** (Menge Atomarer Propositionen)**.**
*Sei $SP_R = (\Sigma, E, R)$ eine relationale algebraische Spezifikation mit Sortenmenge $S$ und Variablenmenge $X$. Sei $AN$ ein algebraisches Petrinetz über $SP_R$ mit Platzmenge $P$.*
*Sei $p, q \in P, \sim \in \{<, >, \leq, \geq, =, \neq\} \subseteq R$ eine 2-stellige Relation. Für $s \in S, te_1, te_2 \in T_{\Sigma,s}(X)$, und $te \in T_{\Sigma,s}$, ist die **Menge atomarer Propositionen AP** über $P, S$ und $X$ definiert als*

$$AP := p.te_1 \mid p.te_1 \wedge (te_1 \sim te) \mid p.te_1 \wedge q.te_2 \wedge (te_1 \sim te_2)$$

**Bemerkung 4.2.**

***p.te*$_1$** *bedeutet, dass Platz $p$ mit dem Term $te_1$ markiert ist. $te_1$ kann eine Konstante, eine Variable zur Sorte $s$ oder eine Operation zur Sorte $s$ sein.*

$\boldsymbol{p.te_1 \wedge (te_1 \sim te)}$ *bedeutet, dass p mit Term $te_1$ markiert ist und $te_1$ in Relation zu te bzgl. des Vergleichsoperators $\sim$ steht. te kann eine Konstante oder eine Operation basierend auf einer Konstanten sein. Das heißt te enthält keine Variable.*

$\boldsymbol{p.te_1 \wedge q.te_2 \wedge (te_1 \sim te_2)}$ *bedeutet, dass wenn zwei Terme, $te_1$ und $te_2$, miteinander verglichen werden, müssen sie an einen Platz, p bzw. q, gebunden werden. Beide Terme können Variablen enthalten.*

Mit diesen atomaren Propositionen können wir nun die Menge eingeschränkter CTL-Formeln definieren, die wir zur Spezifikation datenabhängiger Verhaltenseigenschaften nutzen.

**Definition 4.3** (RCTL-Propositionen)**.**
*Sei $SP_R = (\Sigma, E, R)$ eine relationale algebraische Spezifikation mit Sortenmenge S und Variablenmenge X. Sei AN ein algebraisches Petrinetz über $SP_R$ mit Platzmenge P.*
*Sei AP die Menge atomarer Propositionen über P, S und X. Die* ***Menge von RCTL-Propositionen*** `PROP` *über AP ist wie folgt induktiv definiert.*

1. *Wenn $a \in AP$ gilt, dann gilt $a \in$* `PROP`*.*
2. *Wenn $A_1 \in$* `PROP` *und $A_2 \in$* `PROP` *gilt, dann gilt $A_1 \wedge A_2 \in$* `PROP`

**Definition 4.4** (Rechte-RCTL-Formeln)**.**
*Sei* `PROP` *eine Menge von RCTL-Propositionen. Die* ***Menge rechter-RCTL-Formeln*** `RIGHT` *über* `PROP` *ist wie folgt definiert.*

1. *Wenn $A_R \in$* `PROP` *gilt, dann gilt*
   $EXA_R, AXA_R, EFA_R, AFA_R, EGA_R, AGA_R \in$ `RIGHT`*.*
2. *Wenn $\Psi \in$* `RIGHT` *gilt, dann gilt*
   $EX\Psi, AX\Psi, EF\Psi, AF\Psi, EG\Psi, AG\Psi \in$ `RIGHT`*.*
3. *Wenn $\Psi_1 \in$* `RIGHT` *und $\Psi_2 \in$* `RIGHT` *gilt, dann gilt $\Psi_1 \wedge \Psi_2 \in$* `RIGHT`*.*

**Definition 4.5** (RCTL-Formeln)**.**
*Sei* `PROP` *eine Menge von RCTL-Propositionen. Sei $A_L \in$* `PROP` *und sei $\Psi \in$* `RIGHT` *eine rechte-RCTL-Formel über* `PROP`*.*

*Die* ***Menge von RCTL -Formeln $\Phi$*** *über* `PROP` *ist wie folgt definiert.*

$$\Phi := TRUE \mid A_L \rightarrow \Psi \tag{4.1}$$

Die spezielle Form der RCTL-Formeln spielt in Kapitel 5.2 eine entscheidende Rolle und wird dort näher erläutert.

### 4.1.2 Semantik

Da RCTL eine Teilmenge von CTL ist, entspricht ihre Semantik die der Semantik von CTL (Def. 2.68 auf Seite 47).

Wir wollen die Definition an dieser Stelle kurz wiederholen. Die Semantik von CTL ist durch den Operator $\models$ über ein LTS $TS = (St, s_0, R, L)$ definiert. Für RCTL-Formeln $\varphi \in \Phi$ bleibt die Definition dieses Operators noch gültig. Da wir jedoch keine Negation und Disjunktion unterstützen, können wir nicht auf die in Notation 2.70 eingeführte Abkürzung $A_L \rightarrow \Psi \equiv \neg A_L \vee \Psi$ zurückgreifen. Aus diesem Grund definieren wir im Folgenden zusätzlich die Semantik der Implikation.

$$\forall s \in St.(TS, s \models A_L \rightarrow \Psi \text{ iff } TS, s \models A_L \text{ impliziert } TS, s \models \Psi)$$

## 4.2 RCTL-Netz

Unser vorrangiges Ziel ist es, RCTL-Formeln und APNs derart anzupassen, dass die beiden Formalismen zusammengeführt werden können.

### 4.2.1 Aufbau eines RCTL-Netzes

Wie im vorherigen Abschnitt erläutert, sollte die Ausdrucksstärke von CTL so weit wie möglich beibehalten werden, ohne komplexe Änderungen an den APNs vorzunehmen. Aus diesem Grund haben wir RCTL definiert. RCTL umfasst jedoch temporale-Pfad Operatoren, die nicht direkt durch Elemente der APNs dargestellt werden können. Daher erweitern wir APNs durch eine Funktion, die jeder Transition einen CTL-Operator (oder $\tau$) zuordnet. Ein solches Label beschreibt, wann (im nächsten Zustand, irgendwo in der Zukunft, in jedem zukünftigen Zustand, ..) und wo (immer oder irgendwann) das System einen Zustand erreicht. Ein solches mit einem Label versehenes Netz bezeichnen wir als *RCTL-Netz*.

**Definition 4.6** (RCTL-Netz)**.**
*Sei $SP_R = (\Sigma, E, R)$ eine relationale algebraische Spezifikation. Sei $AN$ ein algebraisches Petrinetz über $SP_R$ mit Transitionsmenge $T$,*
*$K := \{AX, EX, AF, EF, AG, EG\}$ und $L : T \rightarrow K \cup \{\tau\}$ eine Labeling-Funktion. Das Tupel*

$$EN := (AN, L)$$

*wird **RCTL-Netz** über $SP_R$ genannt.*

Die Semantik eines RCTL-Netzes ist in der selben Weise definiert wie die Semantik eines APNs (Abschnitt 2.2.3).

Einzig die Definition des Erreichbarkeitsgraphen ändert sich.

**Definition 4.7** (*Symbolischer Erreichbarkeitsgraph* (*SRG*) eines RCTL-Netzes)**.**

*Sei $SP_R = (\Sigma, E, R)$ eine relationale algebraische Spezifikation mit Variablenmenge $X$. Sei $EN := (AN, L)$ ein RCTL-Netz über $SP_R$ mit Platzmenge $P$, Transitionsmenge $T$ und initialer Markierung $M_0$. Sei $R_{EN}(M_0) := \{M(p) | p \in P \wedge M(p) \in T_\Sigma^+(X) \wedge M$ ist erreichbar von $M_0\}$ die Erreichbarkeitsmenge bzgl. $M_0$ und sei $G$ die Menge der $SP_R$-Guards. Der* ***symbolische Erreichbarkeitsgraph*** *des Netzes $EN$ bzgl. der initialen Markierung $M_0$ ist ein kantenbeschrifteter gerichteter Graph über $T \times L \times G$ mit initialem Knoten $M_0$.*

*Er ist wie folgt definiert.*

$$SRG_{EN}(M_0) = \{R_{EN}(M_0), Ed, M_0, l\} \text{ mit}$$
$$\forall M \in R_{EN}(M_0).\ (M \xrightarrow{t, L(t), guard(t)} M' \implies$$
$$((M, M') \in Ed \wedge l((M, M')) = (t, L(t), guard(t)))$$

### 4.2.2 Komposition von RCTL-Netzen

Nachdem wir das Konzept der RCTL-Netze eingeführt haben, beschäftigen wir uns in diesem Abschnitt mit deren Komposition. Dabei betrachten wir zwei mögliche Konstellationen.

#### (A) Sequentielle Komposition

Sollten die betrachteten Netze die spezielle Struktur aufweisen, dass mindestens ein Finalplatz des einen Netzes mit einem Initialplatz, so erfolgt die Komposition nachfolgendem Schema.

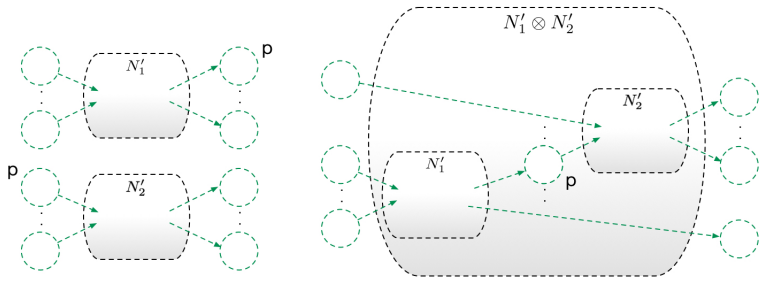
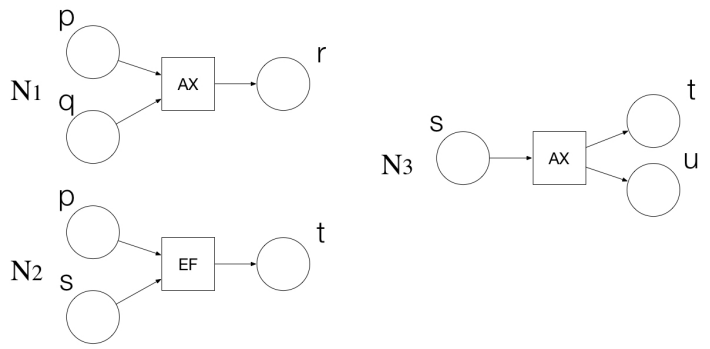



**Abbildung 4.1:** Sequentielle Komposition

#### (B) Verzweigende Komposition

Für alle Fälle, bei denen die sequentielle Komposition nicht anwendbar ist, erfolgt die Komposition, wie im folgenden Verlauf beschrieben wird. Wir betrachten dazu das folgende Beispiel.

**Beispiel 4.1.** Die folgenden Netze sollen mit Hilfe unseres Algorithmus' komponiert werden.

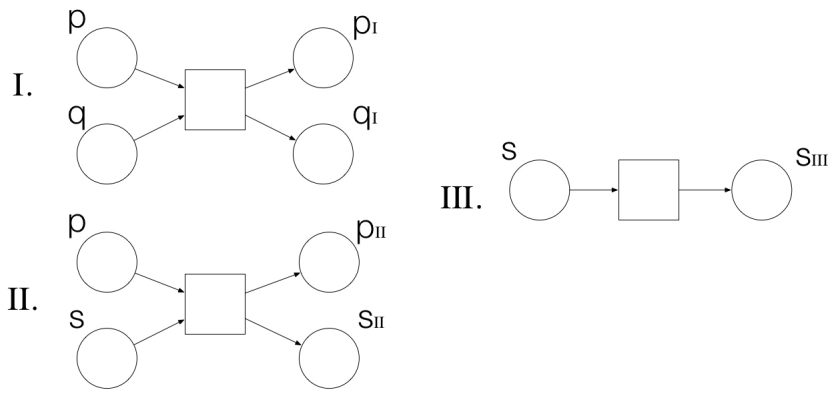



### (B.1.) Links-Komposition

Wir betrachten zunächst die Komposition der Initialplätze. Dabei wollen wir für jeden mehrfach auftauchenden Platz eine Marke duplizieren. Um das zu realisieren, replizieren wir einen Platz entsprechend der Anzahl seiner Vorkommen in der Menge der RCTL-Netze. Dazu werden die Initilaplätze der betrachteten Netze kopiert und mit einem sie eindeutig identifizierenden Index versehen. Die Initialplätze und die so konstruierten Plätze werden durch eine Transition verbunden. Das so entstehende Netz nennen wir im Folgenden **Netzausschnitt**. Dies wird im folgenden Beispiel veranschaulicht.

**Beispiel 4.2.** (Fortsetzung von Beispiel 4.1)

Im Folgenden werden die zu den Netzen $N_1, N_2$ und $N_3$ gehörigen Netzausschnitte dargestellt.

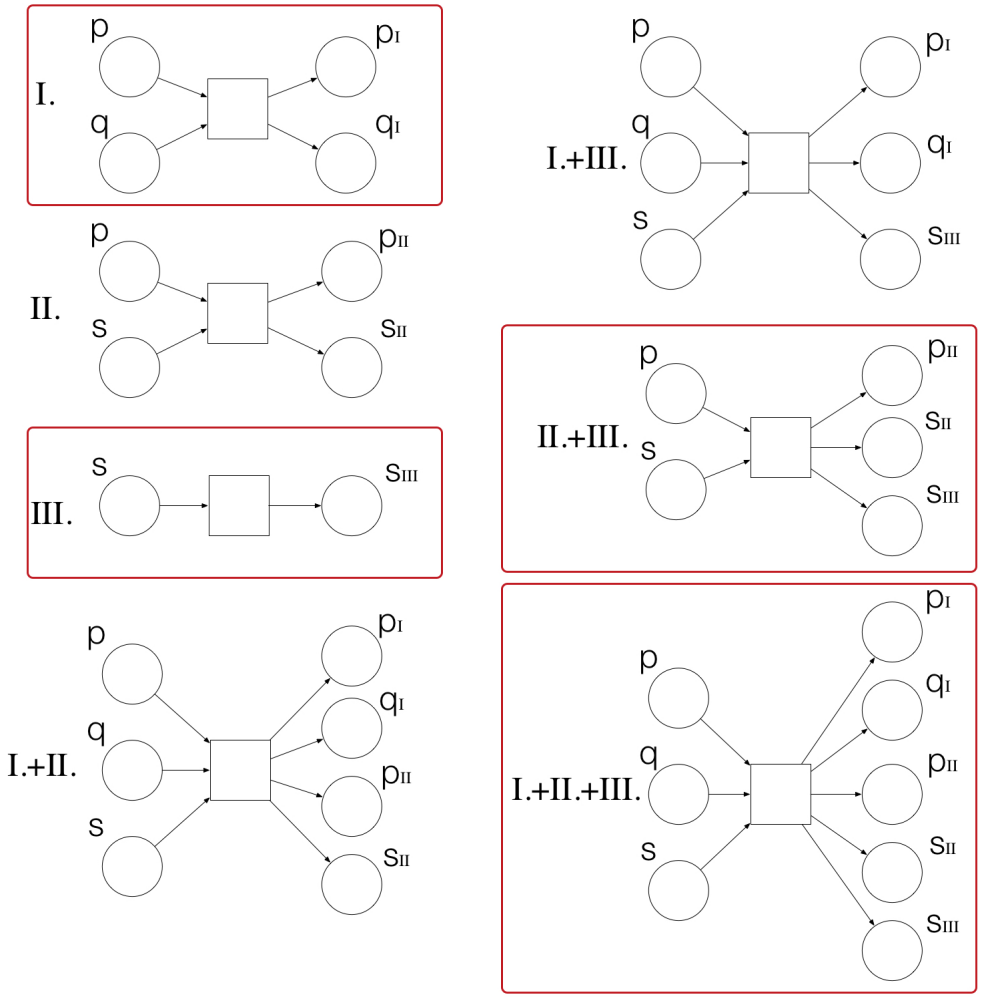


### Schritt 1: Bestimmung der zu komponierenden Netzausschnitte

Aus theoretischer Sicht müssten wir alle Permutationen potentieller Platzreplikationen betrachten. Jedoch entstehen dadurch unnötige Redundanzen. Wir nutzen daher nur die Permutationen, die nicht von einer anderen abgedeckt werden.

Dies wird in folgendem Beispiel veranschaulicht.

**Beispiel 4.3.** (Fortsetzung von Beispiel 4.2)




Wie in dieser Abbildung ersichtlich wird, wird „II.“ durch „II.+III.“ und „I.+II.“ sowie „I.+III.“ durch „I.+II.+III.“ abgedeckt.

Die nicht-umrandeten Netzausschnitte sind redundant und können daher vernachlässigt werden.

**Definition 4.8** (Redundanter Netzausschnitt)**.**
*Seien $NA_1$ und $NA_2$ zwei Netzausschnitte mit den gleichen Initialplätzen. Ist die Menge der Finalplätze in $NA_1$ eine Teilmenge der Finalplätze von $NA_2$, so wird $NA_1$* ***redundanter Netzausschnitt*** *genannt.*

### Schritt 2: Zusammenführung notwendiger Netzausschnitte

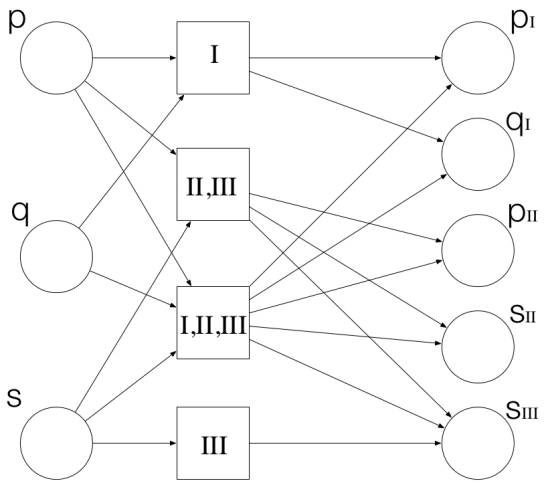
**Definition 4.9** (Notwendiger Netzausschnitt)**.**
*Sei* $NA := \{NA_1, NA_2, ..\}$ *eine Menge von Netzausschnitten. Ist* $NA_i$ *für kein* $NA_j$ *mit* $i, j \in \mathbb{N} \wedge i \neq j$ *ein redundanter Netzausschnitt, so wird* $NA_i$ ***notwendiger Netzausschnitt genannt****.*

*Die Menge der notwendigen Netzausschnitte bezeichnen wir mit* $NA^+$*.*

Die notwendigen Netzausschnitte werden nun komponiert, indem Plätze mit gleicher ID verschmolzen werden. Dies ist in folgendem Beispiel veranschaulicht.

**Beispiel 4.4.** (Fortsetzung von Beispiel 4.3)

Bei den in Beispiel 4.3 rot umrandeten Netzabschnitten handelt es sich um notwendige Netzanschnitte. Diese werden nun komponiert, indem die entsprechenden Plätze verschmolzen werden. Dies resultiert in folgendem Netz.




### Zusammenfassung der Links-Komposition

Mit Hilfe der Links-Komposition wie wir sie beschrieben haben, erhalten wir einen Netzausschnitt, der alle Netzausschnitte abdeckt, die sich aus den betrachteten Netzen ergeben. Dieser Schritt dient der Replikation von Plätzen und damit verbunden dem Duplizieren von Marken. Dies stellt sicher, dass jede Permutation von Anfangsmarkierungen für die zu komponierenden Netze abgedeckt wird.

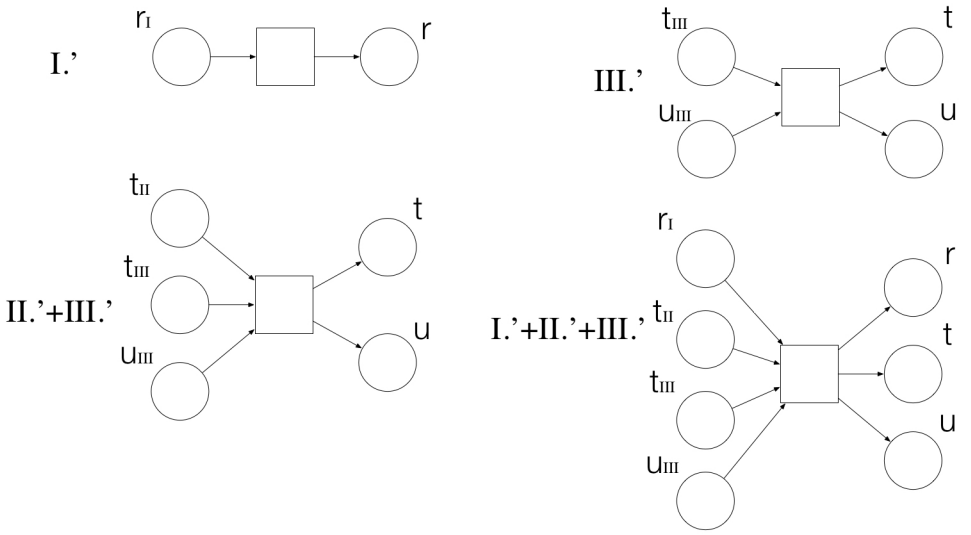
Darauf basierend wird nun die Komposition der Finalplätze betrachtet.

### (B.2.) Rechts-Komposition

In Analogie zu Beispiel 4.2 konstruieren wir zunächst die notwendigen rechten Netzausschnitte für die Finalplätze. Dazu replizieren wir diese und versehen sie mit einem eindeutigen Index. Die Replikate werden durch eine Transition mit den entsprechenden Finalplätzen verbunden. Die *notwendigen rechten* Netzausschnitte werden dabei durch die notwendigen Netzausschnitte der Links-Komposition bestimmt.

**Beispiel 4.5.** (Notwendige Rechte Netzausschnitte)

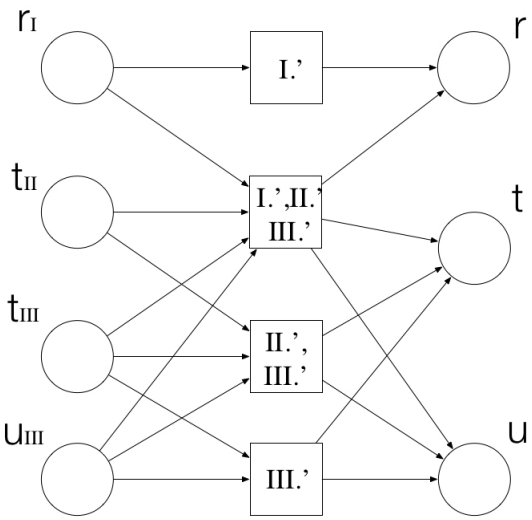
Basierend auf den Netzen aus Beispiel 4.1 und den notwendigen Netzausschnitten der Links-Komposition (Rot-umrandete Netzausschnitte in Beispiel 4.3), ergeben sich die folgenden für die Rechts-Komposition notwendigen Netzausschnitte.




Im zweiten Schritt werden die Netzausschnitte komponiert, indem Plätze mit gleichem Identifizierer verschmolzen werden.

**Beispiel 4.6.** (Fortsetzung von Beispiel 4.5)

Die Rechts-Komposition (der Netzausschnitte) führt zu folgendem Netzausschnitt.

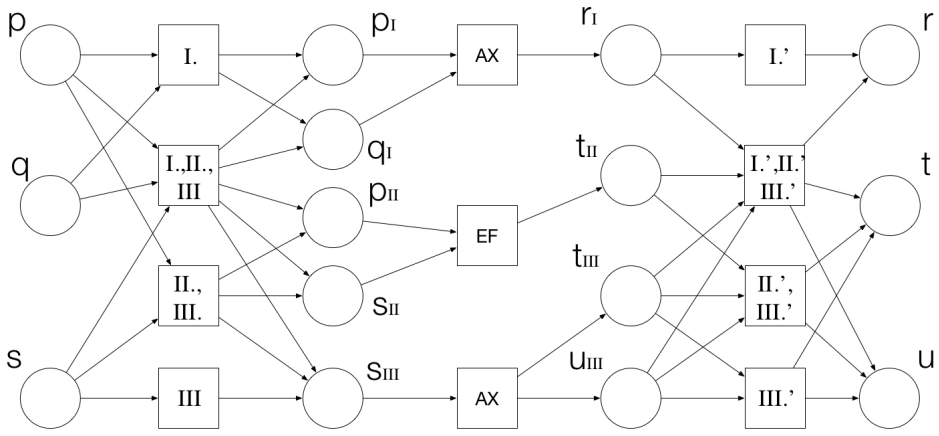



### (B.3.) Komposition

Im letzten Schritt der verzweigenden Komposition werden die Netzausschnitte komponiert, indem die original Netze mit den entsprechenden Plätzen verbunden werden.

**Beispiel 4.7.** (Resultat der Komposition der Netze aus Beispiel 4.1)

Die Komposition der Netzausschnitte mit den originalen Netzen führt in unserem Beispiel zu folgendem Netz.



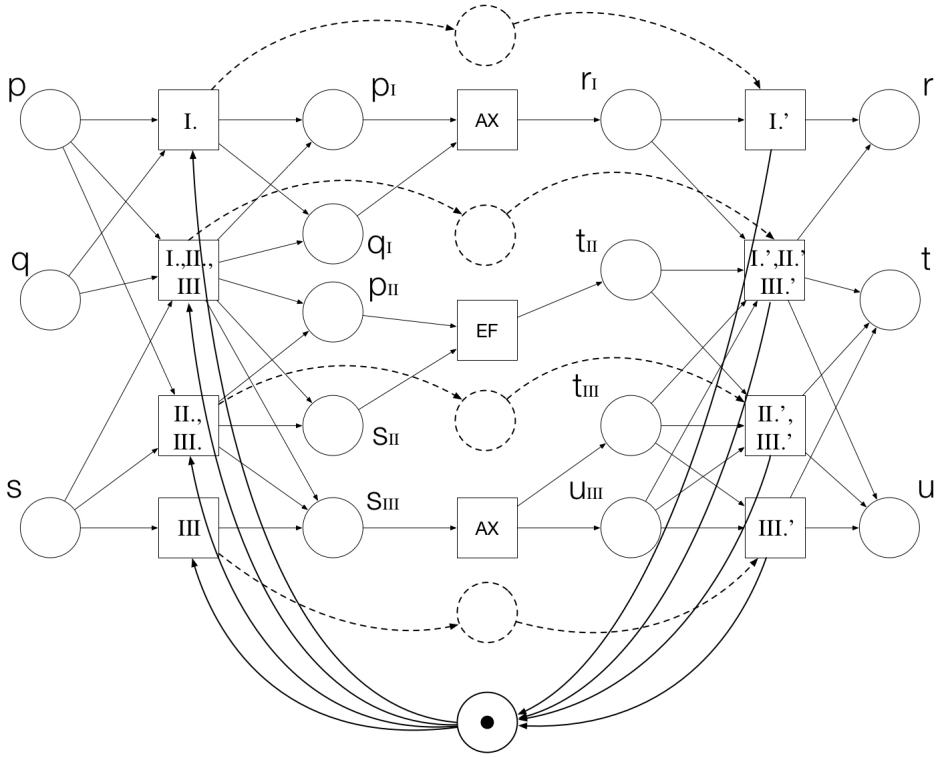

Um sicher zu stellen, dass die richtigen Transitionen schalten und der Zeitpunkt der Ankunft der Marken auf den Plätzen keine Rolle spielt, nutzen wir das Konzept der Runden.

Eine Runde bestimmt dabei den Zeitpunkt an dem eine der Transitionen schalten kann, die mit mindestens einem der Initialplätze verbunden ist. Sie

endet, wenn mindestens einer der Finalplätze markiert wird. Zu diesem Moment befinden sich Marken höchstens auf den Initial- oder Finalplätzen. Alle anderen Plätze sind unmarkiert. Runden werden durch die Einführung eines neuen Platzes realisiert. Dies wird in folgendem Beispiel veranschaulicht.

**Beispiel 4.8.** (Fortsetzung von Beispiel 4.7 )

Die Realisierung des Rundenkonzeptes führt zu folgendem Netz.




Die gestrichelten Plätze stellen sicher, dass die richtigen Transitionen schalten und somit sich nach Abschluss einer Runde keine Marken mehr auf internen Plätzen (nicht-initlal und nicht-final) befinden.

# 5 Operationen auf RCTL-Formeln und RCTL-Netzen

In diesem Kapitel stellen wir die von uns entwickelten Algorithmen vor, mit denen es möglich ist, RCTL-Formeln und APNs zusammen zu bringen.

Unsere Algorithmen ermöglichen eine Abbildung

1. von einem APN $N$ in eine Menge $\Psi$ von RCTL-Formeln (Abschnitt 5.1), sodass durch $\Psi$ das beobachtbare Verhalten von $N$ vollständig spezifiziert wird.
2. von einer RCTL-Formel $\varphi$ in ein RCTL-Netz $N_\varphi$ (Abschnitt 5.2), sodass $\varphi$ in $N_\varphi$ gilt.
3. von einem RCTL-Netz $C$ in ein APN $C'$ (Abschnitt 5.3), sodass $C'$ das Verhalten von $C$ simuliert.

## 5.1 Extraktion von RCTL-Formeln aus einem APN

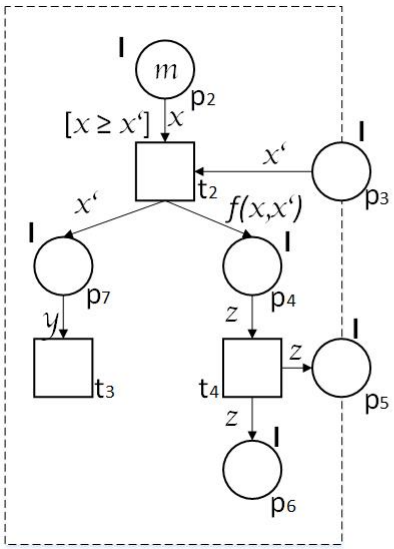
In diesem Kapitel stellen wir einen Algorithmus zur Extraktion einer Menge $\Psi$ von RCTL-Formeln aus einem gegebenen RCTL-Netz $N$ vor (Abschnitt 5.1.1). Dieser arbeitet in zwei Schritten. Im ersten Schritt wird sämtliches Verhalten aus $N$ eliminiert, welches keinen Einfluss auf die Kommunikation, also keine Verbindung zu dem Interface von $N$ hat. Im zweiten Schritt ermitteln wir Beschränkungen der Daten-Wertebereiche an dem Interface und extrahieren davon ausgehend RCTL-Formeln, die das beobachtbare Verhalten von $N$ spezifizieren. Abschließend diskutieren wir die Korrektheit unseres Algorithmus' (Abschnitt 5.1.2).

### 5.1.1 Extraktion Beobachtbaren Verhaltens

Verschiedene Services weisen auch verschiedene Komplexitäten auf. Besonders für aufwendige Aufgaben können die APNs, die das Service-Verhalten model-

lieren, sehr viele Plätze und Transitionen beinhalten. In Kombination mit Daten mit potentiell unendlichen Wertebereichen laufen wir bei der Analyse der Services direkt in das Zustandsraum-Explosions-Problem. Für unsere Zwecke spielt das vollständige Verhalten der Services nur eine Nebenrolle. Wir sind vorrangig an dem Verhalten der Services an den Interfaces interessiert. Das bedeutet, wir beschränken uns auf jenes Verhalten, das einen Einfluss auf die ausgetauschten Daten an den Interface-Ports hat. Um dies auszunutzen, haben wir einen Algorithmus entwickelt, der automatisiert das an den Interfaces beobachtbare Verhalten in Form von RCTL-Formeln extrahiert.

Um eine Intuition dafür zu bekommen, nutzen wir in diesem Abschnitt das folgende Beispiel (Abb. 5.1).




$SP_{\text{Pump}} = (\Sigma_{\text{Pump}}, E_{\text{Pump}})$

$\boldsymbol{\Sigma_{\text{Pump}}}=$

| | |
|---|---|
| **sorts:** | $I$ |
| **vars:** | $x, x', z, y \in X_I$ |
| **operations:** | $m : \rightarrow I$ |
| | $f : I \times I \rightarrow I$ |

$\boldsymbol{E_{\text{Pump}}} = z = f(x,x')$

$y = x'$

$x \geq x'$

**Abbildung 5.1:** Auszug aus dem laufenden Beispiel

## Überblick

Der Algorithmus für die Extraktion beobachtbaren Verhaltens ist in Alg. 1 dargestellt. Als Eingabe erfordert er ein azyklisches APN, das das Verhalten eines Services modelliert. Aus diesem Netz wird das Verhalten entfernt, das keinen Einfluss auf die Kommunikation (Zeile 2) hat. Danach wird ein Vorbereitungsschritt durchgeführt (Zeile 4), bei dem Abhängigkeiten zwischen Variablen bestimmt und Guards zu den Transitionen hin verschoben werden, die mit den Interface-Plätzen verbunden sind. Aus diesem so reduzierten Netz werden RCTL-Formeln extrahiert, die das Verhalten zwischen Initial- und Interface-Platz, sowie zwischen Interface- und Final-Platz spezifizieren (Zeile 6). Die im Vorbereitungsschritt berechneten Resultate werden beim Aufbau der RCTL-Formeln genutzt, die die *Menge der Eigenschaften* $\Psi$ bilden. Diese Menge wird als Ausgabe des Algorithmus' zurückgegeben.

Im Folgenden erläutern wir die einzelnen Schritte detaillierter.

**Algorithmus 1 :** APN2RCTL($AN_S$)

**Eingabe** : $AN_S$ - das APN des Service $S$
**Ausgabe** : $\Psi$ - Menge der RCTL-Formeln

```
1 % Phase Eins
2 AN'_S = LöscheUnnötigePfade(AN_S) % Alg. 2
3 % Vorbereitungsschritt
4 AN'_S = Vorbereitung(AN'_S) % Alg. 3
5 % Phase Zwei
6 Ψ = ExtrahiereFormeln(AN'_S) % Alg. 4
7 return Ψ
```

### Phase 1 - Lösche unnötige Pfade

Basierend auf dem Service-APN wird ein Analyseprozess durchgeführt. Durch diesen werden kausale Zusammenhänge zwischen den Interface-Plätzen oder zwischen je einem Initial-Platz und einem Interface-Platz ermittelt. Dazu wird die mengentheoretische Darstellung des APNs genutzt um Plätze, Transitionen und Kanten zu ermitteln, die in einer Verbindung zu mindestens einem der Interface-Plätze stehen. Diese Elemente bilden das reduzierte Netz.

Wie in Algorithmus 2 auf Seite 78 dargestellt ist, beginnt die Ermittlung der Elemente mit den Interface- und Final-Plätzen (Zeile 1). Dabei werden nur Final-Plätze betrachtet, die in einem kausalen Zusammenhang mit mindestens einem Interface-Platz stehen. Diese werden einem leeren Netz $AN'$ hinzugefügt.

$AN'$ wird nun nach und nach um die in die Kommunikation involvierten Elemente erweitert. Dieser Prozess wird solange durchgeführt, wie neue Elemente zu dem Netz hinzukommen. Dafür wird das Flag `änderung` genutzt (Zeile 4).

Im ersten Schritt dieses sich wiederholenden Prozesses werden alle Transitionen zu $AN'$ hinzugefügt, die durch eine Kante mit den Plätzen aus $AN'$ verbunden sind (Zeilen 5-6). Hierbei bestimmt die Art des Platzes die Richtung der betrachteten Kante. Falls Platz $p$ ein Interface-Eingangsplatz ist, suchen wir die Transitionen $t$, für die eine Kante $(p, t)$ Teil des originalen Netzes ist (Zeilen 7-11). Möglicherweise ist eine Transition bereits Teil von $AN'$, die entsprechende Kante jedoch nicht. Daher suchen wir speziell nach solchen Situationen und fügen diese Kante hinzu (Zeilen 12-13). Obwohl dadurch $AN'$ verändert wird, hat dies keine Auswirkung auf weitere Iterationen. Somit bleibt das Flag `änderung` auf `FALSE`.

Im Falle, dass der betrachtete Platz $p$ kein Interface-Eingangsplatz ist, suchen wir Transitionen $t$, für die $(t, p)$ eine Kante im original Netz ist (Zeilen 14-20). Diese Transitionen und die entsprechenden Kanten werden zu $AN'$ hinzugefügt.

Für die in $AN'$ befindlichen Transitionen $t$ fügen wir nun alle Plätze $p$ hinzu, für die $(p, t)$ eine Kante im Originalnetz ist (Zeilen 21-28).

Danach startet eine neue Iteration bis keine Plätze und keine Transitionen mehr zu $AN'$ hinzugefügt werden.

Der Algorithmus terminiert schlussendlich durch die Rückgabe des reduzierten Netzes (Zeile 29).

Während dieses Extraktionsschrittes werden die Kantenbeschriftungen und die algebraische Spezifikation bei dem Aufbau des reduzierten Netzes mit übernommen werden, auch wenn das nicht explizit im Algorithmus aufgeführt ist.

**Algorithmus 2 :** LöscheUnnötigePfade($AN$)

**Eingabe** : $AN_S$ - das APN des Service $S$.
**Ausgabe** : $AN'_S$ - das reduzierte ***apn!*** (***apn!***)

```
1  füge Interface- und Final-Plätze von AN_S zu AN'_S hinzu
2  änderung = TRUE
3  while änderung do
4      änderung = FALSE
5      for alle Plätze p in P^{AN'_S} do
6          for alle Transitionen t in T^{AN_S} do
7              if p ist ein Interface-Eingangsplatz then
8                  if (t ∉ T^{AN'_S}) ∧ ((p,t) ∈ F^{AN_S}) then
9                      füge t zu T^{AN'_S} hinzu
10                     füge Kante (p,t) zu F^{AN'_S} hinzu
11                     änderung = TRUE
12                 if (t ∈ T^{AN'_S}) ∧ ((p,t) ∈ F^{AN_S}) ∧ ((p,t) ∉ F^{AN'_S}) then
13                     füge Kante (p,t) zu F^{AN'_S} hinzu
14             else
15                 if (t ∉ T^{AN'_S}) ∧ ((t,p) ∈ F^{AN_S}) then
16                     füge t zu T^{AN'_S} hinzu
17                     füge Kante (t,p) zu F^{AN'_S} hinzu
18                     änderung = TRUE
19                 if (t ∈ T^{AN'_S}) ∧ ((t,p) ∈ F^{AN_S}) ∧ ((t,p) ∉ F^{AN'_S}) then
20                     füge Kante (t,p) zu F^{AN'_S} hinzu
21     for alle Transitionen t in T^{AN'_S} do
22         for alle Plätze p in P^{AN_S} do
23             if (p ∉ P^{AN'_S}) ∧ ((p,t) ∈ F^{AN_S}) then
24                 füge p zu P^{AN'_S} hinzu
25                 füge Kante (p,t) zu F^{AN'_S}
26                 änderung = TRUE
27             if (p ∈ P^{AN'_S}) ∧ ((p,t) ∈ F^{AN_S}) ∧ ((p,t) ∉ F^{AN'_S}) then
28                 add arc (p,t) to F^{AN'_S}
29 return AN'_S
```

Um eine bessere Intuition dafür zu bekommen, betrachten wir das folgende Beispiel.

**Beispiel 5.1.** (Phase 1 - Fortführung des Beispiels aus Abb. 5.1)

Angenommen, das Netz in Abbildung 5.1 dient als Eingabe für Algorithmus 2. Wie wir bereits erwähnt haben, nutzen wir im Folgenden die mengentheoretische Darstellung. Um die Übersichtlichkeit zu wahren, listen wir nur die Mengen der Plätze, der Transitionen und die Flussrelation auf.

$$\begin{aligned} AN_S &= (P, T, F, ..) \\ P &= \{p_2, p_3, p_4, p_5, p_6, p_7\} \\ P_{in} = \{p_3\}, P_{out} &= \{p_5\}, P_{final} = \{p_6\} \\ T &= \{t_2, t_3, t_4\} \\ F &= \{(p_2, t_2), (p_3, t_2), (t_2, p_7), (t_2, p_4) \\ &\quad (p_7, t_3), (p_4, t_4), (t_4, p_5), (t_4, p_6)\} \end{aligned}$$

Die *Initialisierung* führt zu folgendem Netz.

$$\begin{aligned} AN'_S &= (P', T', F', ..) \\ P' = \{\mathbf{p_3}, \mathbf{p_5}, \mathbf{p_6}\}, T' &= \emptyset, F' = \emptyset \end{aligned}$$

Das Durchlaufen der `while`-Schleife resultiert in:

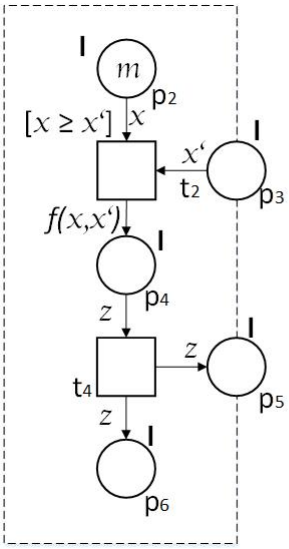
$$\begin{aligned} AN'_S &= (P', T', F', ..) \\ P' = \{p_2, p_3, \mathbf{p_4}, p_5, p_6, \}, T' &= \{\mathbf{t_2}, \mathbf{t_4}\} \\ F' = \{(\mathbf{p_3}, \mathbf{t_2}), (\mathbf{t_4}, \mathbf{p_5}), (\mathbf{t_4}, \mathbf{p_6}), (\mathbf{p_2}, \mathbf{t_2}), (\mathbf{p_4}, \mathbf{t_4})\} \end{aligned}$$

Da Plätze und Transitionen zu dem reduzierten Netz hinzu gefügt wurden, wurde das Flag `änderung` auf `TRUE` gesetzt. Somit findet ein zweiter Durchlauf der `while`-Schleife statt. Während dieses zweiten Durchlaufs ändert sich nur $F'$, wodurch das Flag `FALSE` bleibt. Der Algorithmus terminiert mit dem Zurückgeben des folgenden Netzes.

$$\begin{aligned} AN'_S &= (P', T', F', ..) \\ P' = \{p_2, p_3, p_4, p_5, p_6, \}, T' &= \{t_2, t_4\} \\ F' = \{(p_3, t_2), (t_4, p_5), (t_4, p_6), (p_2, t_2), (p_4, t_4), (\mathbf{t_2}, \mathbf{p_4})\} \end{aligned}$$

Dies kann graphisch wie folgt dargestellt werden.

$AN'_S$




$SP'_{\text{Pump}} = (\Sigma'_{\text{Pump}}, E'_{\text{Pump}})$

$\mathbf{\Sigma'_{Pump}}=$

| | |
|---|---|
| **sorts:** | $I$ |
| **vars:** | $x, x', z \in X_I$ |
| **operations:** | $m : \rightarrow I$ |
| | $f : I \times I \rightarrow I$ |

$\boldsymbol{E'_{\text{Pump}}} = z = f(x,x')$
$x \geq x'$

### Korrektheit und Terminierung der ersten Phase

**Theorem 5.1.**
*Algorithmus 2 auf Seite 78 terminiert.*

*Beweis.*
Der Algorithmus 2 terminiert spätestens dann, wenn alle Elemente des original Netzes $AN$ zum neuen Netz hinzugefügt wurden. Sowohl die Menge der Plätze als auch die Menge der Transitionen von $AN$ sind endlich. Somit terminiert der Algorithmus. □

Um die Korrektheit zu beweisen benötigen wir die folgende Definition.

**Definition 5.2.** *(Weg in einem APN)*
*Sei $AN_S$ ein algebraisches Petrinetz. Ein* ***Weg*** $\mathcal{W}$ *in $AN_S$ ist eine Sequenz von Kanten $(x, y) \in P \times T \cup T \times P$ mit der folgenden Struktur $\mathcal{W}_{AN_S} = (p_0, t_0)(t_0, p_1)...(t_i, p_j), i, j \in \mathbb{N}$.*

**Theorem 5.3.**
*Sei $AN_S$ ein algebraisches Petrinetz und sei $AN'_S$ das reduzierte APN, das aus der Anwendung von Algorithmus 2 resultiert. Für alle Wege $\mathcal{W}$ in $AN_S$, die von einem Initalplatz zu einem Interface-Platz oder von einem Interface-Eingangsplatz zu einem Final-Platz führen, gilt:*

$$\forall f \in F^{AN_S}.\ (f \in \mathcal{W}_{AN_S} \Leftrightarrow f \in F^{AN'_S})$$

*Beweis.*
„⇒“
Für den Beweis des **ersten Falles** sei $\mathcal{W}_{AN_S} = (p_0, t_0)(t_0, p_1)...(t_i, p_j), i, j \in \mathbb{N}$ ein Weg in $AN_S$. Angenommen es sei mindestens ein Interface-Platz Teil eines Tupels auf dem Weg. Wir zeigen, dass für ein beliebiges Element $f = (p, t)$ oder $f = (t, p)$ des Weges $\mathcal{W}_{AN_S}$ folgendes gilt:

$$f \in \mathcal{W}_{AN_S} \implies f \in F^{AN'_S}$$

Der Beweis erfolgt durch Widerspruch.

Sei $f = (p, t) \in \mathcal{W}_{AN_S}$. Es sei angemerkt, dass aufgrund der Definition algebraischer Petrinetze in diesem Fall $p$ weder ein Interface-Ausgangs- noch ein Final-Platz sein kann.

**Annahme:** $f \notin F^{AN'_S}$.

Wenn $p$ ein Interface-Eingangsplatz ist, dann muss $p$ nach Algorithmus 2 Zeile 1 auch Teil von $AN'_S$ sein. Da $p$ in $AN'_S$ ist, müssen durch die Zeilen 7-13 in Algorithmus 2 auch $t$ und $(p, t)$ Teil von $AN'_S$ sein.

Sei $p$ ein anderer beliebiger Platz ist, der in einer kausalen Abhängigkeit mit einem Interface-Platz steht. In diesem Fall ist die Transition, die direkt mit eine Interface-Platz verbunden ist, Teil von $AN'_S$. Beginnend von dieser Transition sucht der Algorithmus in umgekehrter Reihenfolge und fügt alle Plätze hinzu, die mit dieser Transition verbunden sind. Für diese Plätze werden die mit ihnen verbundenen Transitionen hinzugefügt, usw. Das bedeutet, an einem bestimmten Punkt wird $t$ zu $AN'_S$ hinzugefügt, da sie in kausaler Abhängigkeit mit einem Interface-Platz steht, da laut Annahme $f = (p, t) \in \mathcal{W}_{AN_S}$ gilt. Da $t$ Teil von $AN'_S$ ist, werden auch $p$ und $(p, t)$ zu $AN'_S$ hinzugefügt (Alg. 2 Zeilen 21-28).

Somit ist es für eine Kante $(p, t) \in \mathcal{W}_{AN_S}$ nicht möglich, nicht Teil von $AN'_S$ zu sein. Dies ist ein Widerspruch zur Annahme, daher muss

$$(p, t) \in \mathcal{W}_{AN_S} \implies (p, t) \in F^{AN'_S}$$

gelten.

Die Argumentation des Beweises der Aussage

$$(t, p) \in \mathcal{W}_{AN_S} \implies (t, p) \in F^{AN'_S}$$

ist dem obigen Beweis sehr ähnlich. Wir verzichten daher darauf diesen Fall hier direkt anzugeben.

Mit diesen Resultaten gilt:

$$f \in \mathcal{W}_{AN_S} \Rightarrow f \in F^{AN'_S}$$

„⇐“
Für den **zweiten Teil** des Beweises sei $f \in F^{AN'_S}$.

**Annahme:** Es existiert kein Weg $\mathcal{W}_{AN_S} = (p_0, t_0)(t_0, p_1)...(t_i, p_j), i, j \in \mathbb{N}$, sodass $f \in \mathcal{W}_{AN_S}$.

Wenn $f \in F^{AN'_S}$, dann muss Algorithmus 2 entsprechend ein Weg $\mathcal{W}_{AN_S}$ in $AN_S$ existieren, der $f$ enthält, denn nur dadurch kann $f$ Teil von $AN'_S$ sein. Dies ist ein Widerspruch zur Annahme.

Somit muss gelten:

$$f \in \sigma_{AN_S} \Leftarrow f \in F^{AN'_S}$$

□

### Vorbereitungsschritt

Nach dem wir das Verhalten des Services soweit reduziert haben, dass nur noch die für die Kommunikation wichtigen Elemente enthalten sind, führen wir nun einen weiteren Analyseschritt aus, der als Vorbereitung für die zweite Phase dient. Die grundlegende Idee dabei ist, die Informationen zusammenzutragen, die die Datenbereiche der ausgetauschten Nachrichten einschränken.

Um genauer zu sein: Der Vorbereitungsschritt basiert auf dem durch die erste Phase reduzierten Netz (Alg. 2). Dieses Netz besteht nur noch aus den Komponenten, die in die Kommunikation verwickelt sind. In diesem Netz können jedoch noch verschiedene Guards und indirekte Abhängigkeiten von Variablen auftreten. Um diese zu behandeln, haben wir einen Vorbereitungsschritt definiert, der solche Informationen identifizieren und benutzen kann. Dabei werden Variablen ersetzt und Guards zu den Transitionen verschoben, die direkt mit Interface-Plätzen verbunden sind. Eine Vorversion dessen wurde bereits in [Hap15] vorgestellt.

Wie in Algorithmus 3 auf Seite 83 skizziert ist, erfolgt der Vorbereitungsschritt auf zwei Ebenen für alle Wege in dem betrachteten Netz.

Auf der ersten Ebene wird die erste Kantenbeschriftung, die aus einer Variablen besteht als Referenz-Variable genutzt. Alle anderen Kanten werden hinsichtlich der Abhängigkeiten von dieser Referenzvariablen untersucht. Dazu wird die Menge der Gleichungen in der algebraischen Spezifikation mit einbezogen (Zeile 3). Sollte eine Abhängigkeit gefunden werden, werden die Vorkommen der abhäbgigen Variable durch die Referenzvariable ersetzt (Zeile 4). Dies wird solange durchgeführt, bis auf dem Weg keine Änderung mehr auftritt.

Der zweite Schritt erfolgt auf dem selben Weg. In diesem Schritt werden Guards und ihre Zugehörigkeit analysiert. Wenn der Guard von einer eingehenden Nachricht abhängt, so wird er an die Transition verschoben, die mit

dem Interface-Platz verbunden ist, über den die Nachricht ankommt. Das selbe passiert, wenn der Guard keinen Einfluss auf eine ausgehende Nachricht hat (Zeilen 6-7). In dem Fall, dass der Guard eine ausgehende Nachricht beeinflusst, wird er an die Transition verschoben, die mit dem Interface-Ausgangsplatz verbunden ist, über den die Nachricht gesendet werden soll (Zeilen 8-9).

Der Vorbereitungsschritt erleichtert die Extraktion von RCTL-Formeln, die in der zweiten Phase stattfindet.

**Algorithmus 3 :** Vorbereitung($AN_S$)

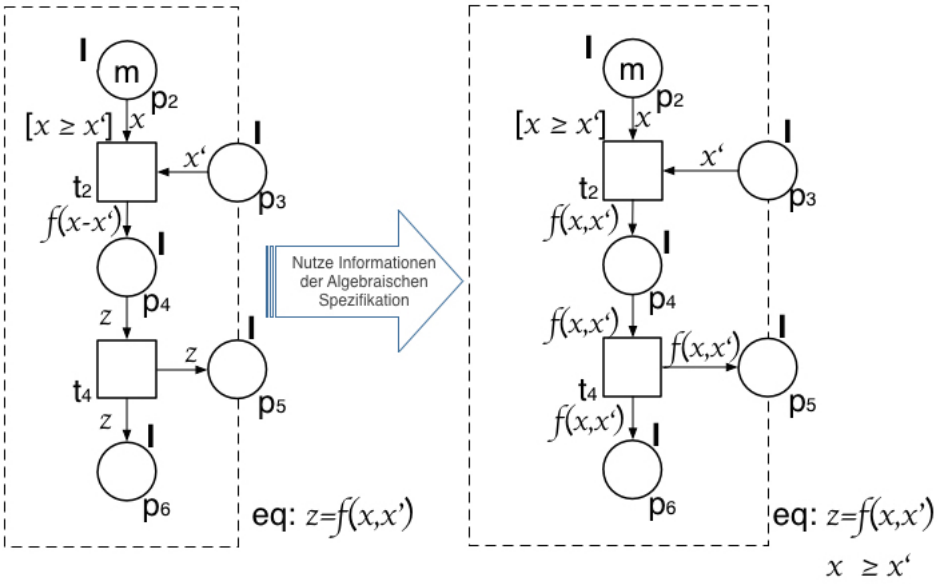
**Eingabe** : $AN_S$: das reduzierte APN des Service $S$
**Ausgabe** : $AN'_S$: das APN des Service $S$ mit verschobenen Guards und ausgenutzten Abhängigkeiten

```
1  for alle Wege W_{AN_S} in AN_S do
2      for alle Kantenbeschriftungen λ und Guards do
3          auf Abhängigkeiten prüfen unter Berücksichtigung der
           Gleichungsmenge
4          substituiere abhängige Variable oder Term durch die
           Referenzvariable oder -term
5      for alle Guards do
6          if Guard ist anhängig von einer Variable eines
           Interface-Eingangsplatzes auf dem Weg oder hat keine
           Verbindung zu einem Interface-Ausgangsplatz then
7              verschiebe Guard zur Transition, die mit dem
               Interface-Eingangsplatz verbunden ist, von dem der Guard
               abhängig ist
8          else
9              verschiebe Guard zur Transition, die mit einem
               Interface-Ausgangsplatz verbunden ist.
10 return AN'_S
```

Um eine bessere Intuition für diesen Schritt zu bekommen betrachten wir das folgende Beispiel, das eine Fortsetzung von Beispiel 5.1 ist.

**Beispiel 5.2.** (Vorbereitungsschritt)

Basierend auf dem aus Beispiel 5.1 resultierenden Netz wird in diesem Beispiel der Vorbereitungsschritt durchgeführt.




Wie in der oberen Abbildung dargestellt ist, beginnt der Algorithmus mit dem in der ersten Phase reduzierten Netz. Wir nutzen die Informationen aus der algebraischen Spezifikation. Für dieses Beispiel ist dies insbesondere die Gleichung $z = f(x, x')$. Damit wird die Kantenbeschriftung $z$ zu $f(x, x')$ geändert. Da der Guard direkt von der Eingabevariable abhängt und er sich bereits an der richtigen Transition befindet, ist ein Verschieben des Guards nicht mehr nötig. Wir speichern jedoch die Information des Guards in der algebraischen Spezifikation ($x \geq x'$).

### Phase 2 - Extraktion von RCTL-Formeln

Nachdem wir das Service-Verhalten auf das für die Kommunikation verantwortliche Verhalten reduziert und in dem Vorbereitungsschritt Informationen bzgl. der ausgetauschten Daten zusammengetragen haben, können wir nun RCTL-Formeln extrahieren, die dies spezifizieren.

Unser Algorithmus (Alg. 4) nutzt das aus Algorithmus 3 resultierende, reduzierte Netz $AN'_S$ als Eingabe. Darauf basierend, wird der SRG von $AN'_S$ beginnend mit der Start-Markierung berechnet (Zeile 1). Die Start-Markierung ist dabei die initiale Markierung zusammen mit einer Markierung der Interface-Eingangsplätze (Zeile 2).

Für jeden Interface-Ausgangsplatz $p_{out}$ wird eine RCTL-Formel konstruiert. Dazu betrachten wir alle Pfade, auf denen der jeweilige Interface-Ausgangsplatz erreichbar ist (Zeilen 3-4).

Bei der Konstruktion der Formel wird die initiale Markierung (der Form $p_{init}.variable$) mit den Markierungen der Interface-Eingangsplätze konjugiert, die in einer kausalen Relation zu $p_{out}$ stehen (Zeilen 5-7). Um die Vorbedin-

**Algorithmus 4 :** ExtrahiereFormeln($AN'_S$)

**Eingabe** : $AN'_S$: das aus Algorithmus 3 resultierende APN
**Ausgabe** : $R$: Menge von RCTL-Formeln

```
1  Konstruiere den symbolischen Erreichbarkeitsgraphen
   SRG_AN'(Start-Markierung) von AN'_S
2  Start-Markierung = Initiale Markierung ∪ Markierung der
   Interface-Eingangsplätze
3  for alle Interface-Ausgangsplätze p_out do
4      finde Pfade σ, die von der Start-Markierung zu einer Markierung von
       p_out führen
5      r = Initial-Markierung
6      for alle Interface-Eingangsplätze p_in, die in der Start-Markierung
       von σ markiert sind und wo von p_out abhängig ist do
7          r++ = ∧p_in.λ((p_in, t))
8      for alle Guards auf dem Pfad σ do
9          r++ = ∧ Guard
10     r ++ = →
11     if p_out auf allen Pfaden im original Netz AN_S erreichbar ist then
12         r ++= AF
13     else
14         r ++ = EF
15     r ++ = p_out.λ((t, p_out))
16     füge req zu R hinzu
17 for alle Pfade σ auf denen kein Interface-Ausgangsplatz erreichbar ist
   oder auf dem ein Interface-Eingangsplatz noch markiert ist, nachdem
   ein Interface-Ausgangsplatz erreicht wurde do
18     r = Initial-Markierung
19     for alle Interface-Eingangsplätze, die in der Start-Markierung von σ
       markiert sind do
20         r++ = ∧p_in.λ((p_in, t))
21     for alle Guards auf dem Pfad σ do
22         r++ = ∧ Guard
23     r ++ = →
24     if p_final auf allen Pfaden des originalen Netzen AN_S erreichbar ist
       then
25         r ++= AF
26     else
27         r ++ = EF
28     r ++ = p_final
29     füge req zu R hinzu
30 return R
```

gung, also die linke Seite der Formel, zur Erreichung von $p_{out}$ zu vervollständigen, werden zudem alle Guards verundet, die sich an den Transitionen der

Interface-Plätze befinden (Zeilen 8-9). Die linke Seite der Formel wird mit einer Implikation ($\rightarrow$) abgeschlossen (Zeile 10).

Abhängig davon, ob $p_{out}$ im originalen Netz auf allen Pfaden im SRG oder auf mindestens einem Pfad erreichbar ist, wird der temporale-Pfad Operator $AF$ oder $EF$ der Formel hinzugefügt (Zeilen 11-14).

Die so entstandene Formel wird nun einer Menge $R$ hinzugefügt (Zeilen 15-16).

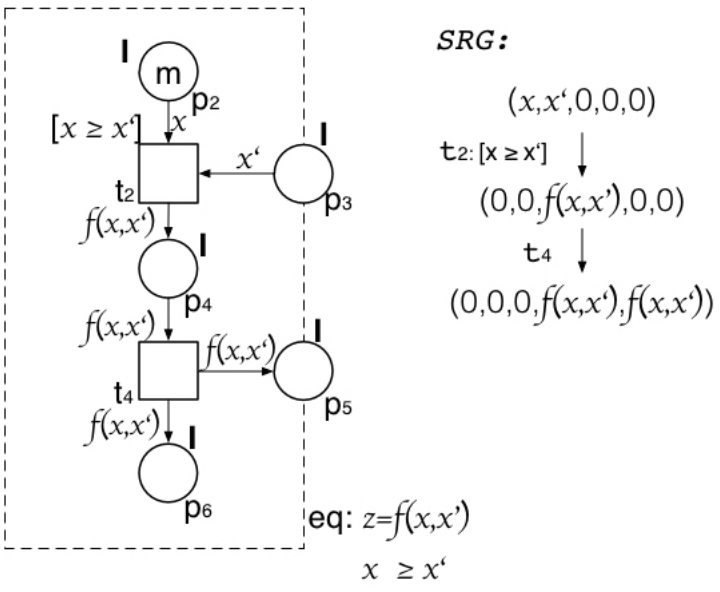
Nachdem wir alle Formeln für die Interface-Ausgangsplätze extrahiert haben, analysieren wir jene Interface-Eingangsplätze, die unabhängig von allen Interface-Ausgangsplätzen sind. In diesem Fall berechnen wir eine Formel für einen Final-Platz (Zeilen 17-29). Dies geschieht analog zum vorher betrachteten Fall.

Der Algorithmus terminiert anschließend mit der Rückgabe der Formelmenge $\Psi$.

Um eine bessere Intuition dafür zu bekommen, betrachten wir das folgende Beispiel.

**Beispiel 5.3.** (Phase 2 )

Basierend auf dem aus dem Vorbereitungsschritt resultierenden Netz (Beispiel 5.2), wird die zweite Phase ausgeführt. Dazu wird der SRG berechnet, was in der folgenden Abbildung dargestellt ist.




Die Anwendung unseres Algorithmus' (Alg. 4) führt zu der folgenden Formel.

$$p_2.x \wedge p_3.x' \wedge [x \geq x'] \rightarrow AFp_5.f(x, x') \tag{5.1}$$

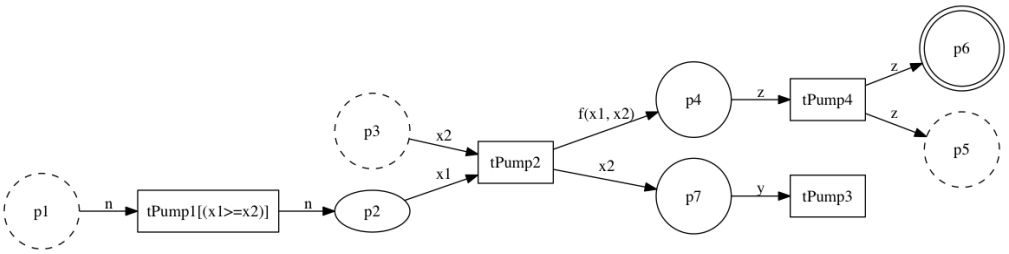
Wir erhalten hierbei nur eine Formel, da der SRG nur einen Pfad aufweist. Des weiteren sei darauf hingewiesen, dass der temporale Pfad-Operator ($AF$) aus der Tatsache resultiert, dass der Final-Platz $p_6$ im originalen Netz $AN_S$ auf allen Pfaden irgendwann erreicht wird.

**Beispiel 5.4. (Laufendes Beispiel - Extraktion beobachtbaren Verhaltens)**

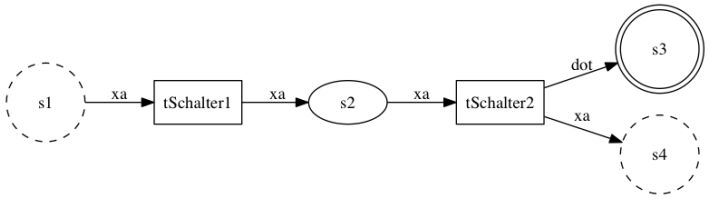
Wir haben den oben beschriebenen Algorithmus zur Extraktion beobachtbaren Verhaltens implementiert. Das Resultat der automatischen Berechnung ist unten zu sehen.

Als Eingabe werden die folgenden Nezte betrachtet, die mit Hilfe von Dot(Graphviz) [GKN15] visualisiert sind. Diese Netze wurden automatisch aus Textdateien erstellt, die die mengentheoretischen Darstellungen beinhalten. In dieser Repräsentation werden initial Plätze als Ovale, Interfache-Plätze gestrichelt und Final-Plätze mit Doppelkreisen dargestellt. Aufgrund der eingeschränkten Möglichkeiten von Dot (Graphviz) können nicht alle Elemente der Netze visualisiert werden. Die fehlenden Elemente, wie die algebraischen Spezifikationen und die Beschriftung der Plätze mit Sortensymbolen, sind dennoch Teil der internen Repräsentation.

Pumpe:




Schalter:




Basierend auf diesen Netzen wird Algorithmus 1 ausgeführt. Dies resultiert in den folgenden, automatisch berechneten zwei RCTL-Formeln.

$$p_1.n \wedge p_3.x_2 \wedge (n \geq x_2) \rightarrow AF(p5.f(n, x_2))$$
$$s_1.x_a \rightarrow AF(s_4.x_a)$$

## 5.1.2 Korrektheit der Extraktion Beobachtbaren Verhaltens

Im vorangegangen Abschnitt haben wir unseren Algorithmus zur Extraktion von RCTL-Formeln aus einem APN vorgestellt. In diesem Abschnitt diskutieren wir dessen Korrektheit. Dafür betrachten wir zunächst die Terminierung. Anschließend überprüfen wir die Korrektheit hinsichtlich der Vollständigkeit des Algorithmus'. Vollständigkeit bedeutet in unserem Fall, dass wir weder falsche Formeln extrahieren, noch dass für die Kommunikation wichtiges Verhalten nicht durch eine Formel abgebildet wird.

### Terminierung

Um Aussagen über die Terminierung von Algorithmus 1 zu treffen, überprüfen wir die Terminierung der drei Phasen.

Die Terminierung von *Phase 1* wurde bereits im vorherigen Abschnitt diskutiert (Theorem 5.1 ).

Während des *Vorbereitungsschrittes*, der nach der ersten Phase ausgeführt wird, werden alle Wege in dem betrachteten Netz durchlaufen (siehe Alg. 3). Da es nur eine endliche Anzahl von Wegen in dem Netz gibt, terminiert der Algorithmus.

In der *zweiten Phase* des Extraktionsprozesses forciert Algorithmus 4 die Konstruktion des SRG. Da die Datenwerte nicht direkt betrachtet sondern als Variablen repräsentiert werden und die Netze maximal eine Marke pro Platz aufweisen, benötigt die Konstruktion des SRGs für ein endliches Netz endliche Zeit. Dies wird auch durch den anschließenden Analyseschritt nicht geändert, da jeder der Pfad maximal $n$-mal betrachtet wird, wobei $n$ die Anzahl der Interface-Ausgangsplätze und per Definition endlich ist.

Da alle Phasen der Extraktion beobachtbaren Verhaltens in endlicher Zeit zu einem Resultat kommen, terminiert Algorithmus 1 ebenfalls.

### Korrektheit

Im Folgenden diskutieren wir die Korrektheit von Algorithmus 1. Dafür zeigen wir, dass wir weder falsche Formeln extrahieren, noch dass für die Kommunikation wichtiges Verhalten nicht durch eine Formel abgebildet wird.

### Annahme: Alle Extrahierten Formeln sind Korrekt

Um zu beweisen, dass wir nur korrekte Formeln extrahieren, zeigen wir, dass jede der extrahierten Formeln in dem als Eingabe genutzten Netz gilt. Dies wird durch folgendes Theorem zusammengefasst.

**Theorem 5.4.**
*Sei $AN_S$ ein APN. Sei $r \in \Psi$ eine Eigenschaft, die aus der Anwendung von Algorithmus 1 (APN2RCTL($AN_S$)) resultiert. Dann gilt:*

$$AN_S \models r$$

*Beweis.*

Der Beweis erfolgt durch Widerspruch.

Da nur die Phase zwei (Alg. 4) einen direkten Einfluss auf die Extraktion der RCTL-Formeln hat, betrachten wir auch nur diese in diesem Beweis. Sei im folgenden $AN_S$ ein APN und $\Psi = \text{APN2RCTL}(AN_S)$.

**Annahme:** Es existiert eine Formel $r \in \Psi$, sodass $AN_S \not\models r$.

Während des Beweises hat $r$ die folgende Struktur

$$\text{Start-Markierung } (\wedge \text{ Guards}) \rightarrow AF(\text{oder } EF) \text{ End-Markierung.}$$

Um zu zeigen, dass $AN_S \not\models r$ gilt, nehmen wir an, dass die Vorbedingung (Start-Markierung ($\wedge$ Guards)) in $AN_S$ gilt. Wenn $AN_S \not\models r$ gelten soll, darf die Nachbedingung von $r$ in $AN_S$ nicht gelten. Da die Final-Markierung entweder ein Interface-Ausgangsplatz oder ein Final-Platz ist, der sich auch so in $AN_S$ wiederfindet, hängt der Wahrheitswert von $r$ von der richtigen Wahl des temporalen-Pfad Operators ab.

Da wir jedoch im Ausgangsnetz $AN_S$ überprüfen, ob die End-Markierung auf allen Pfaden (im Falle von $AF$) oder auf mindestens einem Pfad ($EF$) erreicht wird, wählen wir immer den richtigen temporalen-Pfad Operator. Somit gilt die extrahierte Nachbedingung von $r$ im Netz $AN_S$ ebenfalls.

Das ist ein Widerspruch zu der Annahme, dass $AN_S \not\models r$ gilt. Da wir eine beliebige Anforderung gewählt haben, muss die Annahme für alle $r \in \Psi$ falsch sein.

Daher muss Theorem 5.4 wahr sein.

□

### Annahme: Kein Verlust von Informationen

Im Folgenden diskutieren wir, dass durch unseren Algorithmus alle für die Kommunikation wichtigen Informationen durch die extrahierten Formeln spezifiziert werden.

Der Beweis besteht aus drei Teilen.

1. Zunächst argumentieren wir, dass alle Elemente des Netzes $AN_S$ extrahiert werden, die in die Kommunikation involviert sind.
2. Danach zeigen wir, dass die Substitution von Variablen und Termen auf den extrahierten Pfaden keine Informationen zerstört.
3. Abschließend zeigen wir, dass die Guards auf den extrahierten Pfaden nicht verloren gehen oder signifikant verändert werden.

### Teil 1

Nach Theorem 5.3 gilt, dass alle Wege in $AN_S$, die in die Kommunikation involviert sind im reduzierten Netz $AN'_S$ erhalten bleiben. Da wir in Algorithmus 4 über alle Pfade in $AN'_S$ iterieren und je eine Formel extrahieren, gehen keine für die Kommunikation wichtigen Informationen verloren.

### Teil 2

Im Teil 1 haben wir argumentiert, dass keine wichtigen Informationen verloren gehen. Wir zeigen jetzt, dass während der Substitution von Variablen und Termen auf den extrahierten Pfaden, keine Informationen verloren gehen. Dies ist in folgendem Theorem zusammengefasst.

**Theorem 5.5.**
*Sei $AN_S$ ein reduziertes Service-APN mit Kantenmenge $F^{AN}$, Kanten-Beschriftungsfunktion $\lambda^{AN}$ und Gleichungsmenge $EQ^{AN}$.*
*Sei $AN'_S$ = Vorbereitung($AN_S$) das Service-APN nach der Substitution von Variablen oder Termen. Seien $X$ bzw. $Y$ die Variablenmengen die in $AN_S$ bzw. in $AN'_S$ auftauchen, wobei $Y \subseteq X$ gilt. Seien $t_X \in T_\Sigma(X)$ bzw. $t_Y \in T_\Sigma(Y)$ Terme, die von Variablen aus $X$ bzw. $Y$ abhängig sind. Sei $\delta : X \to Y$ eine Substitutionsfunktion.*

*Dann gilt:*

$$\forall t_X \text{ in } AN_S.\ \forall t_Y \text{ in } AN'_S.$$
$$\exists f \in F^{AN_S} \vee e \in EQ^{AN_S}.(t_Y = \delta(t_X) \in \lambda^{AN_S}(f) \vee e \equiv t_Y = \delta(t_X))$$

*Beweis.*
Entsprechend Algorithmus 3 werden alle Kantenbeschriftungen und Guards hinsichtlich ihrer Abhängigkeit von Termen untersucht. Nur in dem Fall, dass eine Abhängigkeit vorliegt, werden die Gleichungen der algebraischen Spezifikation zur Substitution genutzt. Das bedeutet, ein Term $t_X$ wird durch einen Term $t_Y$ substituiert, wenn in $AN_S$ eine Kantenbeschriftung $\lambda^{AN_S}(f)$ oder eine Gleichung $e$ existiert, sodass $t_X = t_Y$ gilt. Somit muss Theorem 5.5 richtig sein. □

### Teil 3

Im letzten Teil zeigen wir, dass alle auftretenden Guards weder entfernt noch

signifikant verändert werden. „Nicht signifikant verändert“bedeutet in diesem Kontext, dass die Terme, die in den Guards auftreten, nur durch andere Terme substituiert werden. Dass die Substitution keinen negativen Einfluss hat, haben wir bereits in Teil 2 argumentiert. Wir argumentieren daher im Folgenden nur, dass alle Guards aus dem original Netz erhalten bleiben. Das ist im folgenden Theorem zusammengefasst.

**Theorem 5.6.** *Sei $AN_S$ das reduzierte Service-APN nach der Substitution von Variablen oder Termen mit Platzmenge $P$ und Transitionsmenge $T$. Sei $AN'_S$ das APN nach dem Verschieben der Guards.*
*Sei $\mathcal{W}_{AN_S} = ((p_0, t_0), (t_0, p_1), ..., (t_i, p_j))$, mit $p_k \in P, t_l \in T$ für $k, l \in [0, i], i \in \mathbb{N}$ ein Weg, der von einem Initial-Platz zu mindestens einem Interface-Platz in $AN_S$ führt. Dann gilt:*

$$\forall \mathcal{W}_{AN_S}.\forall \text{ Guards } g_X \text{ in } AN_S.$$
$$g_X \text{ ist ein Guard auf dem Weg } \mathcal{W}_{AN_S} \implies g_X \text{ in } AN'_S.$$

*Beweis.*
Entsprechend Algorithmus 3 Zeilen 5-9 iterieren wir über alle Wege $\mathcal{W}_{AN_S}$ und alle Guards auf jedem solcher Wege. Die entdeckten Guards werden nur zu den Interface-Eingangs- oder Interface-Ausgangsplätzen hin verschoben. Das bedeutet alle Guards bleiben erhalten. Daher muss Theorem 5.6 wahr sein. □

### 5.1.3 Zusammenfassung

In diesem Kapitel haben wir einen Algorithmus zur Extraktion beobachtbaren Verhaltens definiert und dessen Korrektheit bewiesen. Der Algorithmus besteht aus zwei Phasen und einem Vorbereitungsschritt. Sein Ziel ist die Extraktion einer Menge von RCTL-Formeln aus einem APN, welches das Verhalten eines Services modelliert.

In der ersten Phase werden aus einem gegebenen APN die Elemente gelöscht, die keinen Einfluss auf die Kommunikation haben. In dem daraus resultierenden Netz werden in einem Vorbereitungsschritt Variablen und Terme soweit wie möglich ersetzt, um deren Anzahl zu reduzieren. Dazu werden Gleichungen aus der algebraischen Spezifikation genutzt. Zusätzlich werden in diesem Schritt auch Guards an Transitionen verschoben, die mit den Interface-Plätzen verbunden sind. Das resultierende Netz wird als Eingabe für die zweite Phase genutzt. In dieser Phase werden die Formeln extrahiert, die das Kommunikationsverhalten des Services spezifizieren.

## 5.2 Übersetzung einer RCTL-Formel in ein RCTL-Netz

Im letzten Abschnitt haben wir einen Algorithmus vorgestellt, mit dessen Hilfe eine Menge $R$ von RCTL-Formeln aus einem gegebenen APN $N$ ermittelt werden kann, sodass $R$ das beobachtbare Verhalten von $N$ beschreibt.

In diesem Abschnitt stellen wir nun einen Algorithmus zur Übersetzung von RCTL-Formeln in RCTL-Netze vor und ebnen damit den Weg zur Zusammenführung der beiden Formalismen.

Basierend auf der Syntax und Semantik von RCTL (siehe Abschnitt 4.1) und RCTL-Netzen (siehe Abschnitt 4.2), definieren wir dazu Übersetzungsregeln, diskutieren deren Korrektheit und zeigen potentielle Optimierungsmöglichkeiten.

Die Übersetzung einer RCTL-Formel $\varphi$ in ein RCTL-Netz $N_\varphi$ ist dabei so definiert, dass $\varphi$ in $N_\varphi$ gültig ist.

**Notation 5.7.**
*Sei $EN := (AN, L)$ ein RCTL-Netz. Sei $\Phi$ eine Menge von RCTL-Formeln und sei $\varphi \in \Phi$ eine RCTL-Formel. Sei $M$ eine Markierung in $EN$.*

*Dann wird die Aussage „$\varphi$ ist **gültig in $M$**“ mit $(EN, M) \models \varphi$ bezeichnet.*

**Notation 5.8.**
*Sei $EN := (AN, L)$ ein RCTL-Netz und sei $\varphi \in \Phi$ eine RCTL-Formel.*

*Dann ist $\varphi$ **in $EN$ gültig**, geschrieben als $EN \models \varphi$, genau dann, wenn für alle erreichbaren Markierungen $M \in R_{EN}(M_0)$ gilt: $(EN, M) \models \varphi$.*

Wie in Definition 4.5 auf Seite 66 deutlich wird, ist eine RCTL-Formel $\varphi \in \Phi$ entweder `TRUE` oder folgt dem Aufbau $\varphi = A_L \rightarrow \Psi$. Für $\varphi = TRUE$ erzeugen wir ein leeres Netz, was für die Übersetzungsregeln uninteressant ist und somit hier keine weitere Beachtung findet. Wir betrachten daher im Folgenden nur Formeln der Form $\varphi = A_L \rightarrow \Psi$. Durch die Implikation auf oberster Ebene wird die Formel in zwei Teilformeln unterteilt. Die linke Seite der Formel beschreibt dabei den Zustand, in dem sich das System aktuell befindet. Die rechte Seite beschreibt zukünftige Zustände, in denen sich das System befinden wird, wenn der Zustand eintritt, den die linke Seite der Formel spezifiziert.

### 5.2.1 Links-Übersetzung

**Definition 5.9** (Links-Übersetzung)**.**
*Sei $AP$ die Menge atomarer Propositionen von RCTL. Sei* `PROP` *die Menge von RCTL-Propositionen über $AP$. Die **Links-Übersetzung** ist eine Funktion $Trans_L$ : `PROP` $\rightarrow RCTL - Netze$. Ihre Definition folgt den in den Tabellen 5.1 und 5.2 angegebenenx Regeln.*

### Übersetzungsregeln der Linken Seite

Tabelle 5.1 veranschaulicht unsere Übersetzungsregeln für die atomaren Propositionen ($a_L \in AP$). Sei im Folgenden $\sim \in \{<, >, \leq, \geq, =, \neq\}$.

**Tabelle 5.1:** $Trans_L(a_L)$

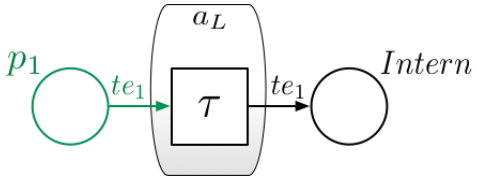
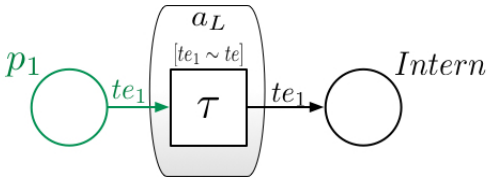
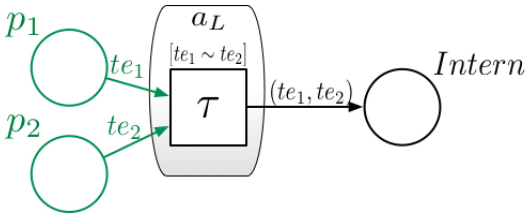
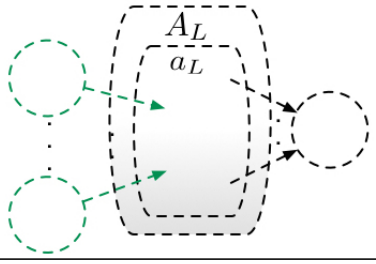
| | Atomare Proposition | RCTL-Netz |
|---|---|---|
| 1 | $a_L = p_1.te_1$ |   |
| 2 | $a_L = p_1.te_1 \wedge (te_1 \sim te)$ |   |
| 3 | $a_L = p_1.te_1 \wedge p_2.te_2 \wedge (te_1 \sim te_2)$ |   |

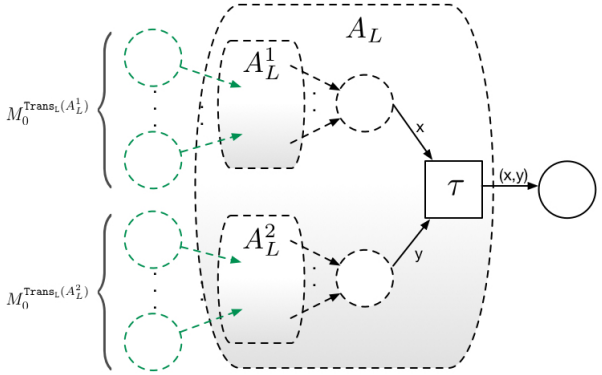
Tabelle 5.2 veranschaulicht unsere Übersetzungsregeln für die RCTL-Propositionen ($A_L \in PROP$).

**Tabelle 5.2:** $Trans_L(A_L)$

| | $A_L$ | RCTL-Netz |
|---|---|---|
| 4 | $A_L = a_L$ |   |

Fortsetzung auf der nächsten Seite

**Tabelle 5.2** – Fortsetzung der vorherigen Seite

| | $A_L$ | RCTL-Netz |
|---|---|---|
| 5 | $A_L = A_L^1 \wedge A_L^2$ |  |



### Korrektheit

Um die Korrektheit unserer Übersetzungsregeln zu beweisen, benötigen wir die folgenden Definitionen.

**Definition 5.10** (Links-Initiale Markierung)**.**
*Sei $A_L \in$ PROP eine RCTL-Proposition und sei $\mathtt{Trans_L}(A_L)$ das aus der Links-Übersetzung von $A_L$ resultierende RCTL-Netz.*

*Eine **links-initiale Markierung von** $\mathtt{Trans_L}(\boldsymbol{A_L})$, die mit $M_0^{\mathtt{Trans_L}(A_L)}$ bezeichnet wird, ist wie folgt definiert.*

1. *Jeder Platz mit einem leeren Vorbereich ist mit exakt einem Term markiert.*
2. *Alle anderen Plätze sind unmarkiert.*

**Lemma 5.11.**
*Sei $A_L \in$ PROP eine RCTL-Proposition und sei $\mathtt{Trans_L}(A_L)$ das aus der Links-Übersetzung von $A_L$ resultierende RCTL-Netz.*

*Dann besitzt $\mathtt{Trans_L}(A_L)$ genau einen Platz mit einem leeren Nachbereich Im Folgenden nennen wir diesen Platz **Finalplatz** von $\mathtt{Trans_L}(A_L)$.*

*Beweis.*
Trivialer Weise folgt die Beobachtung, dass $\mathtt{Trans_L}(A_L)$ genau einen Finalplatz besitzt, direkt aus den Übersetzungsregeln (Regeln 1-5 in Tab. 5.1 und Tab. 5.2). □

**Definition 5.12** (Links-Finale Markierung)**.**
*Sei $A_L \in$ PROP eine RCTL-Proposition und sei $\mathtt{Trans_L}(A_L)$ das aus der Links-Übersetzung von $A_L$ resultierende RCTL-Netz.*

*Eine **links-finale Markierung von** $\texttt{Trans}_\texttt{L}(A_L)$ $M_\Omega^{\texttt{Trans}_\texttt{L}(A_L)}$ ist wie folgt definiert.*

1. *Der Finalplatz ist mit genau einem Term markiert.*
2. *Alle anderen Plätze sind unmarkiert.*

**Theorem 5.13.**
*Sei $A_L \in \texttt{PROP}$ eine RCTL-Proposition. Sei $\texttt{Trans}_\texttt{L}(A_L)$ das aus der Links-Übersetzung von $A_L$ resultierende RCTL-Netz. Sei $\texttt{Trans}_\texttt{L}(A_L)$ initial mit der links-initialen Markierung $M_0^{\texttt{Trans}_\texttt{L}(A_L)}$ markiert.*

*Dann gilt:*

*$A_L$ ist gültig in $M_0^{\texttt{Trans}_\texttt{L}(A_L)}$ genau dann, wenn eine links-finale Markierung $M_\Omega^{\texttt{Trans}_\texttt{L}(A_L)}$ von $\texttt{Trans}_\texttt{L}(A_L)$ von $M_0^{\texttt{Trans}_\texttt{L}(A_L)}$ aus immer erreichbar ist.*

**Bemerkung 5.14.**
*Die Aussage, dass eine Markierung $M_\Omega$ von einer Markierung $M_0$ aus immer erreichbar ist, bedeutet, dass $M_\Omega$ für jede denkbare Belegung in $M_0$ und jeden potentiellen Ausführungsverlauf erreichbar ist.*

*Beweis.*
Sei $AP$ die Menge atomarer Propositionen von RCTL. Sei $A_L \in \texttt{PROP}$ eine RCTL-Proposition über $AP$ und sei $\texttt{Trans}_\texttt{L}(A_L)$ das aus der Links-Übersetzung von $A_L$ resultierende RCTL-Netz. Sei $\texttt{Trans}_\texttt{L}(A_L)$ initial mit einer links-initialen Markierung $M_0^{\texttt{Trans}_\texttt{L}(A_L)}$ markiert.

Der Beweis erfolgt mittels struktureller Induktion über den Aufbau der Formeln $A_L$. Dies beinhaltet:

Induktionsanfang: Wenn $A_L$ eine atomare Proposition ist, also $A_L = a_L$, dann gilt:
$(\texttt{Trans}_\texttt{L}(a_L), M_0^{\texttt{Trans}_\texttt{L}(a_L)}) \models a_L \Leftrightarrow$ eine links-finale Markierung $M_\Omega^{\texttt{Trans}_\texttt{L}(a_L)}$ von $\texttt{Trans}_\texttt{L}(a_L)$ ist immer erreichbar.

Induktionsvoraussetzung: Für alle $A_L$ gilt:
$(\texttt{Trans}_\texttt{L}(A_L), M_0^{\texttt{Trans}_\texttt{L}(A_L)}) \models A_L \Leftrightarrow$ eine links-finale Markierung $M_\Omega^{\texttt{Trans}_\texttt{L}(A_L)}$ von $\texttt{Trans}_\texttt{L}(A_L)$ ist immer erreichbar.

Induktionsschritt:
Für $A_L = A_L^1 \wedge A_L^2$ gilt:
$(\texttt{Trans}_\texttt{L}(A_L^1 \wedge A_L^2), M_0^{\texttt{Trans}_\texttt{L}(A_L^1 \wedge A_L^2)}) \models A_L^1 \wedge A_L^2 \Leftrightarrow$ eine links-finale Markierung $M_\Omega^{\texttt{Trans}_\texttt{L}(A_L^1 \wedge A_L^2)}$ von $\texttt{Trans}_\texttt{L}(A_L)$ ist immer erreichbar.

### Induktionsanfang

Fall 1: $a_L = p_1.te_1$
Sei $M_0^{\texttt{Trans}_\texttt{L}(a_L)} = (te_1, \vartheta)$. Entsprechend der Übersetzungsregel 1 (Tab.

5.1) und der Definition 4.7, hat der symbolische Erreichbarkeitsgraph von $\texttt{Trans}_\texttt{L}(a_L)$ die folgende Struktur:

$$SRG_{\texttt{Trans}_\texttt{L}(a_L)}(M_0^{\texttt{Trans}_\texttt{L}(a_L)}) = \{ \\ \{(te_1, \vartheta), (\vartheta, te_1)\}, \\ \{((te_1, \vartheta), (\vartheta, te_1))\}, \\ (te_1, \vartheta), \\ l((te_1, \vartheta), (\vartheta, te_1)) = (t, \tau, \texttt{TRUE})\}$$

**⇒:** Angenommen, dass $(\texttt{Trans}_\texttt{L}(a_L), M_0^{\texttt{Trans}_\texttt{L}(a_L)}) \models a_L$ gilt. Dann kann die $\tau$-Transition schalten. Dies führt zu einer links-finalen Markierung $M_\Omega^{\texttt{Trans}_\texttt{L}(a_L)} = (\vartheta, te_1)$. Da der $SRG$ nur einen Pfad aufweist, ist eine linke-finale Markierung immer erreichbar.

**⇐:** Angenommen, die links-finale Markierung ist immer erreichbar. Dazu muss die $\tau-$Transition geschaltet haben. Das kann nur der Fall sein, wenn $\texttt{Trans}_\texttt{L}(\texttt{a}_\texttt{L})$ initial mit $M_0^{\texttt{Trans}_\texttt{L}(a_L)} = (te_1, \vartheta)$ markiert ist.

Das bedeutet, dass $a_L$ in $\texttt{Trans}_\texttt{L}(a_L)$ für die links-initiale Markierung $M_0^{\texttt{Trans}_\texttt{L}(a_L)}$ gültig ist.

Fall 2: $a_L = p_1.te_1 \wedge (te_1 \sim te)$
Sei $M_0^{\texttt{Trans}_\texttt{L}(a_L)} = (te_1, \vartheta)$.

Entsprechend der Übersetzungsregel 2 (Tab. 5.1) und der Definition 4.7, hat der symbolische Erreichbarkeitsgraph von $\texttt{Trans}_\texttt{L}(a_L)$ die folgende Struktur:

$$SRG_{\texttt{Trans}_\texttt{L}(a_L)}(M_0^{\texttt{Trans}_\texttt{L}(a_L)}) = \{ \\ \{(te_1, \vartheta), (\vartheta, te_1)\}, \\ \{((te_1, \vartheta), (\vartheta, te_1))\}, \\ (te_1, \vartheta), \\ l((te_1, \vartheta), (\vartheta, te_1)) = (t, \tau, \{(te_1 \sim te)\})\}$$

**⇒:** Angenommen, $(\texttt{Trans}_\texttt{L}(a_L), M_0^{\texttt{Trans}_\texttt{L}(a_L)}) \models a_L$ gilt, was auch beinhaltet, dass der Guard $(te_1 \sim te)$ zu `TRUE` ausgewertet wird. In diesem Fall schaltet die $\tau$-Transition, was zu einer links-finalen Markierung $M_\Omega^{\texttt{Trans}_\texttt{L}(a_L)} = (\vartheta, te_1)$ führt. Da der $SRG$ nur einen Pfad aufweist, ist eine links-finale Markierung immer erreichbar ist.

**⇐:** Angenommen, die links-finale Markierung ist immer erreichbar. Das bedeutet, die $\tau-$Transition hat geschaltet. Dies kann nur der Fall sein, wenn der Guard zu `TRUE` ausgewertet wird und $\texttt{Trans}_\texttt{L}(\texttt{a}_\texttt{L})$ initial mit $M_0^{\texttt{Trans}_\texttt{L}(a_L)} = (te_1, \vartheta)$. markiert ist.

Somit ist $a_L$ in $\texttt{Trans}_\texttt{L}(a_L)$ gültig für die links-initiale Markierung $M_0^{\texttt{Trans}_\texttt{L}(a_L)}$.

Fall 3: $a_L = p_1.te_1 \wedge p2.te_2 \wedge (te_1 \sim te_2)$
Entsprechend der Übersetzungsregel 3 (Tab. 5.1) und der Definition 4.7, hat der symbolische Erreichbarkeitsgraph von $\texttt{Trans}_\texttt{L}(a_L)$ die folgende Struktur:

$$SRG_{\mathtt{Trans_L}(a_L)}(M_0^{\mathtt{Trans_L}(a_L)}) = \{$$
$$\{(te_1, te_2, \vartheta), (\vartheta, \vartheta, (te_1, te_2))\},$$
$$\{((te_1, te_2, \vartheta), (\vartheta, \vartheta, (te_1, te_2)))\},$$
$$(te_1, te_2, \vartheta),$$
$$l((te_1, te_2, \vartheta), (\vartheta, \vartheta, (te_1, te_2))) = (t, \tau, \{(te_1 \sim te_2)\})\}$$

**⇒:** Angenommen $(\mathtt{Trans_L}(a_L), M_0^{\mathtt{Trans_L}(a_L)}) \models a_L$ gilt was auch beinhaltet, dass der Guard $(te_1 \sim te_2)$ zu `TRUE` ausgewertet wird. In diesem Fall schaltet die $\tau$-Transition, was zu einer links-finalen Markierung $M_\Omega^{\mathtt{Trans_L}(a_L)} = (\vartheta, \vartheta, (te_1, te_2))$ führt. Da der $SRG$ nur einen Pfad aufweist, ist eine links-finale Markierung immer erreichbar ist.

**⇐:** Angenommen, die links-finale Markierung ist immer erreichbar. Das bedeutet, die $\tau-$Transition hat geschaltet. Dies kann nur der Fall sein, wenn der Guard zu `TRUE` ausgewertet wird und $\mathtt{Trans_L}(\mathtt{a_L})$ initial mit $M_0^{\mathtt{Trans_L}(a_L)} = (te_1, te_2, \vartheta).$ markiert ist.

Somit ist $a_L$ in $\mathtt{Trans_L}(a_L)$ gültig für die links-initiale Markierung $M_0^{\mathtt{Trans_L}(a_L)}$.

**Resultat:** Für die Basis-Fälle gilt:

$$(\mathtt{Trans_L}(a_L), M_0^{\mathtt{Trans_L}(a_L)}) \models a_L \Leftrightarrow \text{eine links-finale Markierung}$$
$$M_\Omega^{\mathtt{Trans_L}(a_L)} \text{ von } \mathtt{Trans_L}(A_L) \text{ ist immer erreichbar.}$$

### Induktionsschritt

$A_L = A_L^1 \wedge A_L^2$:

**⇒:** Angenommen, $(\mathtt{Trans_L}(A_L^1 \wedge A_L^2), M_0^{\mathtt{Trans_L}(A_L^1 \wedge A_L^2)}) \models A_L^1 \wedge A_L^2$ gilt. Entsprechend der Übersetzungsregel 10 (Tab. 5.2) und der Induktionsvoraussetzung, sind die beiden internen Plätze mit genau einem Term markiert. In diesem Zustand kann die $\tau-$Transition schalten und somit ist eine links-finale Markierung $M_\Omega^{\mathtt{Trans_L}(A_L^1 \wedge A_L^2)}$ von $\mathtt{Trans_L}(A_L^1 \wedge A_L^2)$ immer erreichbar.

**⇐:** Angenommen, die links-finale Markierung ist immer erreichbar. Entsprechend der Übersetzungsregel 10 (Tab. 5.2), bedeutet das, die $\tau-$Transition hat geschaltet. Dies kann nur der Fall sein, wenn die zwei internen Plätze markiert waren. Somit muss nach der Induktionsvoraussetzung gelten:

$$(\mathtt{Trans_L}(A_L^1), M_0^{\mathtt{Trans_L}(A_L^1)}) \models A_L^1 \text{ und } (\mathtt{Trans_L}(A_L^2), M_0^{\mathtt{Trans_L}(A_L^2)}) \models A_L^2$$

Daher gilt:

$$(\mathtt{Trans_L}(A_L^1 \wedge A_L^2), M_0^{\mathtt{Trans_L}(A_L^1 \wedge A_L^2)}) \models A_L^1 \wedge A_L^2.$$

**Resultat:** Für $A_L = A_L^1 \wedge A_L^2$ gilt:

$$(\mathtt{Trans_L}(A_L), M_0^{\mathtt{Trans_L}(A_L)}) \models A_L \Leftrightarrow \text{eine links-finale Markierung}$$
$$M_\Omega^{\mathtt{Trans_L}(A_L)} \text{ von } \mathtt{Trans_L}(A_L) \text{ ist immer erreichbar.}$$

□

### 5.2.2 Rechts-Übersetzung

**Definition 5.15** (Rechts-Übersetzung)**.**
*Sei AP die Menge atomarer Propositionen von RCTL. Sei* PROP *die Menge von RCTL-Propositionen über AP und sei* RIGHT *die Menge rechter-RCTL-Formeln über* PROP*. Die* ***Rechts-Übersetzung*** *ist eine Funktion* $Trans_R$ : PROP $\cup$ RIGHT $\rightarrow$ *RCTL − Netze. Ihre Definition folgt den, in den Tabellen 5.3, 5.4 und 5.5 angegebenen, Regeln.*

#### Übersetzungsregeln der Rechten Seite

Tabelle 5.3 definiert die Übersetzung atomarer Propositionen der rechten Seite ($a_R \in AP$). Sei im Folgenden $\sim \in \{<, >, \leq, \geq, =, \neq\}$.

**Tabelle 5.3:** $Trans_R(a_R)$

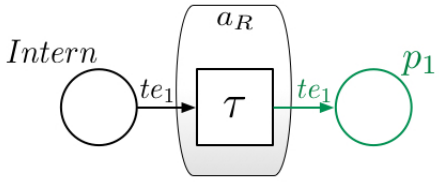
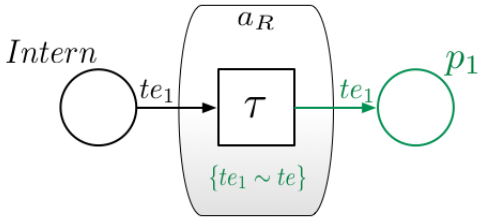
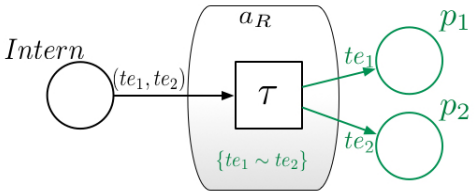
| | Atomare Proposition | RCTL-Netz |
|---|---|---|
| 6 | $a_R = p_1.te_1$ |  |
| 7 | $a_R = p_1.te_1 \wedge (te_1 \sim te)$ |  |
| 8 | $a_R = p_1.te_1 \wedge p_2.te_2 \wedge (te_1 \sim te_2)$ |  |

Tabelle 5.5 definiert die Übersetzung von RCTL-Propositionen der rechten Seite ($A_R \in$ PROP).

**Tabelle 5.4:** $Trans_R(A_R)$

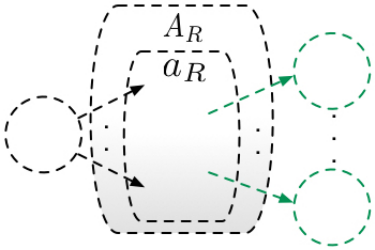
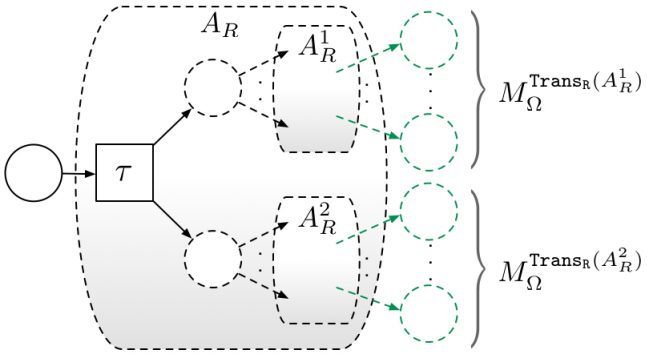
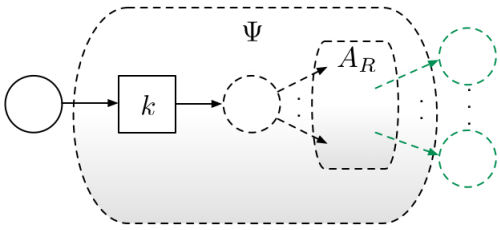
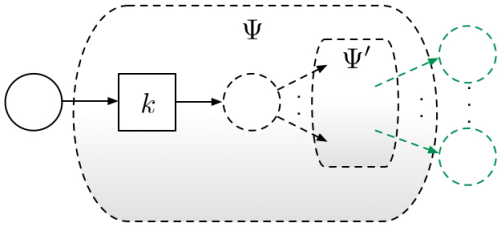
| | $A_R$ | RCTL-Netz |
|---|---|---|
| 9 | $A_R = a_R$ |  |
| 10 | $A_R = A_R^1 \wedge A_R^2$ |  |

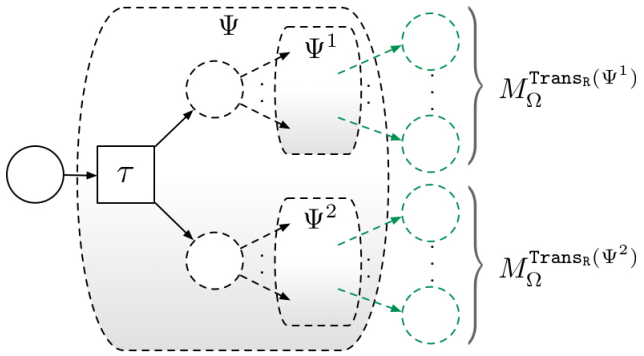
Tabelle 5.4 definiert die Übersetzung rechter-RCTL-Formeln über $PROP$ der rechten Seite ($\Psi \in$ `RIGHT`). Sei im folgenden $k \in K := \{AX, EX, AF, EF, AG, EG\}$.

**Tabelle 5.5:** $Trans_R(\Psi)$

| | $\Psi$ | RCTL-Netz |
|---|---|---|
| 11 | $\Psi = k\ A_R$ |  |
| 12 | $\Psi = k\ \Psi'$ |  |

Fortsetzung auf der nächsten Seite

**Tabelle 5.5** – Fortsetzung der vorherigen Seite

| | $\Psi$ | RCTL-Netz |
|---|---|---|
| 13 | $\Psi = \Psi^1 \wedge \Psi^2$ |  |



### Korrektheit

Um die Korrektheit unserer Transformation zu beweisen, benötigen wir die folgenden Definitionen.

**Lemma 5.16.**
*Sei* $\Psi \in$ `RIGHT` *eine rechte-RCTL-Formel und sei* $\mathtt{Trans_R}(\Psi)$ *das aus der Rechts-Übersetzung von* $\Psi$ *resultierende RCTL-Netz.*

*Dann besitzt* $\mathtt{Trans_R}(\Psi)$ *genau einen Platz mit leerem Vorbereich. Diesen bezeichnen wir im Folgenden als* ***Initialplatz*** *von* $\mathtt{Trans_R}(\Psi)$.

*Beweis.*
Trivialer Weise folgt die Beobachtung, dass $\mathtt{Trans_R}(\Psi)$ genau einen Initialplatz besitzt, aus den Übersetzungsregeln (Regeln 6-13, die in den Tabellen 5.3, 5.4 und 5.5 definiert sind). □

**Definition 5.17** (Rechts-Initiale Markierung)**.**
*Sei* $\Psi \in$ `RIGHT` *eine rechte-RCTL-Formel und sei* $\mathtt{Trans_R}(\Psi)$ *das aus der Rechts-Übersetzung von* $\Psi$ *resultierende RCTL-Netz.*

*Eine* ***rechts-initiale Markierung von*** $\mathtt{Trans_R}(\Psi)$ $M_0^{\mathtt{Trans_R}(\Psi)}$ *ist wie folgt definiert.*

1. *Der Initialplatz ist mit genau einem Term markiert.*
2. *Alle anderen Plätze sind unmarkiert.*

**Definition 5.18** (Rechts-Finale Markierung)**.**
*Sei* $\Psi \in$ `RIGHT` *eine rechte-RCTL-Formel und sei* $\mathtt{Trans_R}(\Psi)$ *das aus der Rechts-Übersetzung von* $\Psi$ *resultierende RCTL-Netz.*

*Eine* ***rechts-finale Markierung von*** $\mathtt{Trans_R}(\Psi)$ $M_\Omega^{\mathtt{Trans_R}(\Psi)}$ *ist wie folgt definiert.*

1. *Jeder Platz mit einem leeren Nachbereich ist mit genau einem Term markiert.*
2. *Alle anderen Plätze sind unmarkiert.*

**Theorem 5.19.**
*Sei AP die Menge atomarer Propositionen von RCTL und sei* $A_R \in \texttt{PROP}$ *eine RCTL-Proposition über AP. Sei* $\texttt{Trans}_\texttt{R}(A_R)$ *das aus der Rechts-Übersetzung von* $A_R$ *resultierende RCTL-Netz.*

*Dann gilt:*

$A_R$ *ist gültig in* $M_\Omega^{\texttt{Trans}_\texttt{R}(A_R)}$ *genau dann, wenn* $\texttt{Trans}_\texttt{R}(A_R)$ *ist initial mit einer rechts-initialen Markierung* $M_0^{\texttt{Trans}_\texttt{R}(A_R)}$ *markiert.*

*Beweis.*
Sei $AP$ die Menge atomarer Propositionen von RCTL und sei $A_R \in \texttt{PROP}$ eine RCTL-Proposition über $AP$. Sei $\texttt{Trans}_\texttt{R}(A_R)$ das aus der Rechts-Übersetzung von $A_R$ resultierende RCTL-Netz.

Der Beweis erfolgt durch strukturelle Induktion über den Aufbau der Formeln $A_R$. Dies beinhaltet:

Induktionsanfang:
Wenn $A_R$ eine atomare Proposition ist, also $A_R = a_R$, dann gilt:
$(\texttt{Trans}_\texttt{R}(a_R), M_\Omega^{\texttt{Trans}_\texttt{R}(a_R)}) \models a_R \Leftrightarrow$
$\texttt{Trans}_\texttt{R}(a_R)$ ist initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_\texttt{R}(a_R)}$ markiert

Induktionsvoraussetzung:
Für alle $A_R$ gilt:
$(\texttt{Trans}_\texttt{R}(A_R), M_\Omega^{\texttt{Trans}_\texttt{R}(A_R)}) \models A_R \Leftrightarrow$
$\texttt{Trans}_\texttt{R}(A_R)$ ist initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_\texttt{R}(A_R)}$ markiert

Induktionsschritt:
Für $A_R = A_R^1 \wedge A_R^2$ gilt:
$(\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2), M_\Omega^{\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2)}) \models A_R^1 \wedge A_R^2 \Leftrightarrow$
$\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2)$ ist initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2)}$ markiert

### Induktionsanfang

Fall 1: $a_R = p_1.te_1$
**⇒:** Gelte $(\texttt{Trans}_\texttt{R}(a_R), M_\Omega^{\texttt{Trans}_\texttt{R}(a_R)}) \models a_R$. Entsprechend der Übersetzungsregel 6 (Tab. 5.3), bedeutet dies, dass die $\tau$-Transition geschaltet hat. Das kann nur der Fall sein, wenn der Platz *Intern* initial mit genau

einem Term markiert war. Das bedeutet, $\texttt{Trans}_\texttt{R}(a_R)$ ist zu diesem Zeitpunkt initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_\texttt{R}(a_R)}$ markiert.

**$\Leftarrow$:** Angenommen $\texttt{Trans}_\texttt{R}(a_R)$ ist initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_\texttt{R}(a_R)}$ markiert. In diesem Fall schaltet die $\tau-$ Transition und Platz $p_1$ ist mit genau einem Term $te_1$ markiert. Das bedeutet, $\texttt{Trans}_\texttt{R}(a_R)$ ist mit einer rechts-finalen Markierung $M_\Omega^{\texttt{Trans}_\texttt{R}(a_R)}$ markiert und es gilt:
$(\texttt{Trans}_\texttt{R}(a_R), M_\Omega^{\texttt{Trans}_\texttt{R}(a_R)}) \models a_R$.

Fall 2: $a_R = p_1.te_1 \wedge (te_1 \sim te)$

**$\Rightarrow$:** Gelte $(\texttt{Trans}_\texttt{R}(a_R), M_\Omega^{\texttt{Trans}_\texttt{R}(a_R)}) \models a_R$. Entsprechend der Übersetzungsregel 7 (Tab. 5.3), bedeutet dies, dass die $\tau$-Transition geschaltet hat und eine Relation $te_1 \sim te$ gespeichert wurde. Das kann nur der Fall sein, wenn der Platz $Intern$ initial mit genau einem Term markiert war. Das bedeutet, $\texttt{Trans}_\texttt{R}(a_R)$ ist zu diesem Zeitpunkt initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_\texttt{R}(a_R)}$ markiert.

**$\Leftarrow$:** Angenommen $\texttt{Trans}_\texttt{R}(a_R)$ ist initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_\texttt{R}(a_R)}$ markiert. In diesem Fall schaltet die $\tau-$ Transition und Platz $p_1$ ist mit genau einem Term $te_1$ markiert. Dabei wird eine Relation $\{te_1 \sim te\}$ gespeichert, die die Menge der möglichen Interpretationen einschränkt, denn es sind nur solche Interpretationen erlaubt, die die Relation erfüllen. Das bedeutet, $\texttt{Trans}_\texttt{R}(a_R)$ ist mit einer rechts-finalen Markierung $M_\Omega^{\texttt{Trans}_\texttt{R}(a_R)}$ markiert und es gilt:
$(\texttt{Trans}_\texttt{R}(a_R), M_\Omega^{\texttt{Trans}_\texttt{R}(a_R)}) \models a_R$.

Fall 3: $a_R = p_1.te_1 \wedge p2.te_2 \wedge (te_1 \sim te_2)$

**$\Rightarrow$:** Gelte $\texttt{Trans}_\texttt{R}(a_R) \models_{M_\Omega^{\texttt{Trans}_\texttt{R}(a_R)}} a_R$. Entsprechend der Übersetzungsregel 8 (Tab. 5.3), bedeutet dies, dass die $\tau$-Transition geschaltet hat und eine Relation $te_1 \sim te_2$ gespeichert wurde. Das kann nur der Fall sein, wenn der Platz $Intern$ initial mit genau einem Term markiert war. Das bedeutet, $\texttt{Trans}_\texttt{R}(a_R)$ ist zu diesem Zeitpunkt initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_\texttt{R}(a_R)}$ markiert.

**$\Leftarrow$:** Angenommen $\texttt{Trans}_\texttt{R}(a_R)$ ist initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_\texttt{R}(a_R)}$ markiert. In diesem Fall schaltet die $\tau-$ Transition und die Plätze $p_1$ und $p_2$ sind mit genau einem Term $te_1$ und $te_2$ markiert. Dabei wird eine Relation $\{te_1 \sim te_2\}$ gespeichert, die die Menge der möglichen Interpretationen einschränkt, denn es sind nur solche Interpretationen erlaubt, die die Relation erfüllen. Das bedeutet, $\texttt{Trans}_\texttt{R}(a_R)$ ist mit einer rechts-finalen Markierung $M_\Omega^{\texttt{Trans}_\texttt{R}(a_R)}$ markiert und es gilt:
$(\texttt{Trans}_\texttt{R}(a_R), M_\Omega^{\texttt{Trans}_\texttt{R}(a_R)}) \models a_R$.

**Resultat:** Für die Basis-Fälle gilt:

$$(\texttt{Trans}_\texttt{R}(a_R), M_\Omega^{\texttt{Trans}_\texttt{R}(a_R)}) \models a_R \Leftrightarrow$$
$$\texttt{Trans}_\texttt{R}(a_R) \text{ ist initial mit einer rechts-initialen Markierung } M_0^{\texttt{Trans}_\texttt{R}(a_R)} \text{ markiert.}$$

#### Induktionsschritt

$A_R = A_R^1 \wedge A_R^2$:

**$\Rightarrow$:** Gelte $(\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2), M_\Omega^{\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2)}) \models A_R^1 \wedge A_R^2$. Entsprechend der Übersetzungsregel 10 (Tab. 5.4), impliziert das:
$(\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2), M_\Omega^{\texttt{Trans}_\texttt{R}(A_R^1)}) \models A_R^1$ und
$\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2) \models_{M_\Omega^{\texttt{Trans}_\texttt{R}(A_R^2)}} A_R^2$. Aus der Induktionsvoraussetzung folgt, dass beide internen Plätze irgendwann mit genau einem Term markiert sind. Dies kann nur der Fall sein, wenn die $\tau-$Transition geschaltet hat. Dazu muss $\texttt{Trans}(A_R^1 \wedge A_R^2)$ initial mit einer rechts-initialen Markierung markiert sein.

**$\Leftarrow$:** Angenommen, $\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2)$ ist initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2)}$ markiert. Entsprechend der Übersetzungsregel 10 (Tab. 5.4), schaltet die $\tau-$Transition und jeder interne Platz ist mit genau einem Term markiert. Diese Markierung ist eine rechts-initiale Markierung für $\texttt{Trans}(A_R^1)$ und $\texttt{Trans}(A_R^2)$. Aus der Induktionsvoraussetzung folgt, dass $M_\Omega^{\texttt{Trans}_\texttt{R}}(A_R^1)$ und $M_\Omega^{\texttt{Trans}_\texttt{R}}(A_R^2)$ erreicht werden und es gilt:
$(\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2), M_\Omega^{\texttt{Trans}_\texttt{R}(A_R^1)}) \models A_R^1$ und
$(\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2), M_\Omega^{\texttt{Trans}_\texttt{R}(A_R^2)}) \models A_R^2$.

Somit gilt: $(\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2), M_\Omega^{\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2)}) \models A_R^1 \wedge A_R^2$

**Resultat:** Für $A_R = A_R^1 \wedge A_R^2$ gilt:

$$(\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2), M_\Omega^{\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2)}) \models A_R^1 \wedge A_R^2 \Leftrightarrow$$
$$\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2) \text{ ist initial mit einer rechts-initialen Markierung}$$
$$M_0^{\texttt{Trans}_\texttt{R}(A_R^1 \wedge A_R^2)} \text{ markiert.}$$

□

**Theorem 5.20.**
*Sei AP die Menge atomarer Propositionen von RCTL und sei* PROP *die Menge von RCTL-Propositionen über AP. Sei* $\Psi \in \texttt{RIGHT}$ *eine rechte-RCTL-Formel über* PROP. *Sei* $\texttt{Trans}_\texttt{R}(\Psi)$ *das aus der Rechts-Übersetzung von* $\Psi$ *resultierende RCTL-Netz.*

*Dann gilt:*

$$\Psi \textit{ ist in } M_\Omega^{\texttt{Trans}_\texttt{R}(\Psi)} \textit{ gültig genau dann, wenn}$$
$$\texttt{Trans}_\texttt{R}(\Psi) \textit{ initial mit einer rechts-initialen Markierung } M_0^{\texttt{Trans}_\texttt{R}(\Psi)} \textit{ markiert ist.}$$

*Beweis.*
Sei $AP$ die Menge atomarer Propositionen von RCTL und sei PROP die Menge von RCTL-Propositionen über $AP$. Sei $\Psi \in \texttt{RIGHT}$ eine rechte-RCTL-Formel

über PROP. Sei $\texttt{Trans}_{\texttt{R}}(\Psi)$ das aus der Rechts-Übersetzung von $\Psi$ resultierende RCTL-Netz. Sei $k \in K := \{AX, EX, AF, EF, AG, EG\}$ ein temporaler-Pfad Operator.

Der Beweis erfolgt mittels struktureller Induktion über den Aufbau der Formeln $\Psi$. Die beinhaltet:

Induktionsvoraussetzung: Für $\Psi = k\ A_R$ gilt:
$(\texttt{Trans}_{\texttt{R}}(k\ A_R), M_\Omega^{\texttt{Trans}_{\texttt{R}}(k\ A_R)}) \models k\ A_R \Leftrightarrow \texttt{Trans}_{\texttt{R}}(k\ A_R)$ ist initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_{\texttt{R}}(k\ A_R)}$ markiert

Induktionsvoraussetzung: Für alle $\Psi$ gilt:
$(\texttt{Trans}_{\texttt{R}}(\Psi), M_\Omega^{\texttt{Trans}_{\texttt{R}}(\Psi)}) \models \Psi \Leftrightarrow \texttt{Trans}_{\texttt{R}}(\Psi)$ ist initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_{\texttt{R}}(\Psi)}$ markiert

Induktionsschritt:

i) Für $\Psi = k\ \Psi'$ gilt:
$(\texttt{Trans}_{\texttt{R}}(k\ \Psi'), M_\Omega^{\texttt{Trans}_{\texttt{R}}(k\ \Psi')}) \models k\ \Psi' \Leftrightarrow$
$\texttt{Trans}_{\texttt{R}}(k\ \Psi')$ ist initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_{\texttt{R}}(k\ \Psi')}$

ii) Für $\Psi = \Psi^1 \wedge \Psi^2$ gilt:
$(\texttt{Trans}_{\texttt{R}}(\Psi^1 \wedge \Psi^2), M_\Omega^{\texttt{Trans}_{\texttt{R}}(\Psi^1 \wedge \Psi^2)}) \models \Psi^1 \wedge \Psi^2 \Leftrightarrow$
$\texttt{Trans}_{\texttt{R}}(\Psi^1 \wedge \Psi^2)$ ist initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_{\texttt{R}}(\Psi^1 \wedge \Psi^2)}$ markiert

### Induktionsanfang

**$\Rightarrow$:** Gelte $(\texttt{Trans}_{\texttt{R}}(k\ A_R), M_\Omega^{\texttt{Trans}_{\texttt{R}}(k\ A'_R)}) \models k\ A_R$. Entsprechend der Übersetzungsregel 11 (Tab. 5.5), ist $M_\Omega^{\texttt{Trans}_{\texttt{R}}(k\ A_R)}$ eine rechts-finale Markierung von $\texttt{Trans}_{\texttt{R}}(A_R)$. In diesem Fall folgt aus Theorem 5.19, dass der interne Platz irgendwann mit genau einem Term markiert ist. Dies ist eine rechts-initiale Markierung von $\texttt{Trans}_{\texttt{R}}(A_R)$. Damit dies eintreten kann, muss die $k$- Transition geschaltet haben. Dazu muss der Initialplatz initial mit genau einem Term markiert sein. Somit gilt: $\texttt{Trans}_{\texttt{R}}(k\ A_R)$ ist initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_{\texttt{R}}(k\ A_R)}$ markiert.

**$\Leftarrow$:** Angenommen, $\texttt{Trans}_{\texttt{R}}(k\ A_R)$ ist initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_{\texttt{R}}(k\ A_R)}$ markiert. Entsprechend der Übersetzungsregel 11 (Tab. 5.5) schaltet die $k-$Transition und der interne Platz wird mit genau einem Term belegt. Da diese Markierung eine rechts-initiale Markierung von $\texttt{Trans}_{\texttt{R}}(A_R)$ ist, folgt aus Theorem 5.19, dass eine rechts-finale Markierung $M_\Omega^{\texttt{Trans}_{\texttt{R}}(A_R)}$ immer erreicht wird und $(\texttt{Trans}_{\texttt{R}}(A_R), M_\Omega^{\texttt{Trans}_{\texttt{R}}(A_R)}) \models A_R$ gilt.

Konsequenter Weise ist $\texttt{Trans}_{\texttt{R}}(k\ A_R)$ mit einer rechts-finalen Markierung $M_\Omega^{\texttt{Trans}_{\texttt{R}}(k\ A_R)}$ markiert und es gilt:
$(\texttt{Trans}_{\texttt{R}}(k\ A_R), M_\Omega^{\texttt{Trans}_{\texttt{R}}(k\ A_R)}) \models k\ A_R$.

**Induktionsschritt**

i) $\Psi = k\ \Psi'$

**⇒:** Gelte $(\mathtt{Trans}_{\mathtt{R}}(k\ \Psi'), M_{\Omega}^{\mathtt{Trans}_{\mathtt{R}}(k\ \Psi')}) \models k\ \Psi'$. Entsprechend der Übersetzungsregel 12 (Tab. 5.5), ist $M_{\Omega}^{\mathtt{Trans}_{\mathtt{R}}(k\ \Psi')}$ eine rechts-finale Markierung von $\mathtt{Trans}_{\mathtt{R}}(\Psi')$. In diesem Fall folgt aus Induktionsvoraussetzung, dass der interne Platz irgendwann mit genau einem Term markiert ist. Dies ist eine rechts-initiale Markierung von $\mathtt{Trans}_{\mathtt{R}}(\Psi')$. Damit dies eintreten kann, muss die $k$- Transition geschaltet haben. Dazu muss der Initialplatz initial mit genau einem Term markiert sein. Somit gilt: $\mathtt{Trans}_{\mathtt{R}}(k\ \Psi')$ ist initial mit einer rechts-initialen Markierung $M_0^{\mathtt{Trans}_{\mathtt{R}}(k\ \Psi')}$ markiert.

**⇐:** Angenommen, $\mathtt{Trans}_{\mathtt{R}}(k\ \Psi')$ ist initial mit einer rechts-initialen Markierung $M_0^{\mathtt{Trans}_{\mathtt{R}}(k\ \Psi')}$ markiert. Entsprechend der Übersetzungsregel 12 (Tab. 5.5) schaltet die $k-$Transition und der interne Platz wird mit genau einem Term belegt. Da diese Markierung eine rechts-initiale Markierung von $\mathtt{Trans}_{\mathtt{R}}(\Psi')$ ist, folgt aus der Induktionsvoraussetzung, dass eine rechts-finale Markierung $M_{\Omega}^{\mathtt{Trans}_{\mathtt{R}}(\Psi')}$ immer erreicht wird und $(\mathtt{Trans}_{\mathtt{R}}(\Psi'), M_{\Omega}^{\mathtt{Trans}_{\mathtt{R}}(\Psi')}) \models \Psi'$ gilt.

Konsequenter Weise ist $\mathtt{Trans}_{\mathtt{R}}(k\ \Psi')$ mit einer rechts-finalen Markierung $M_{\Omega}^{\mathtt{Trans}_{\mathtt{R}}(k\ \Psi')}$ markiert und es gilt:
$(\mathtt{Trans}_{\mathtt{R}}(k\ \Psi'), M_{\Omega}^{\mathtt{Trans}_{\mathtt{R}}(k\ \Psi')}) \models k\ \Psi'$.

ii) $\Psi = \Psi^1 \wedge \Psi^2$

**⇒:** Gelte $\mathtt{Trans}_{\mathtt{R}}(\Psi^1 \wedge \Psi^2) \models_{M_{\Omega}^{\mathtt{Trans}_{\mathtt{R}}(\Psi^1 \wedge \Psi^2)}} \Psi^1 \wedge \Psi^2$. Entsprechend der Übersetzungsregel 13 (Tab. 5.5), ist $M_{\Omega}^{\mathtt{Trans}_{\mathtt{R}}(\Psi^1 \wedge \Psi^2)}$ eine Kombination aus rechts-finalen Markierungen von $\mathtt{Trans}_{\mathtt{R}}(\Psi^1)$ und $\mathtt{Trans}_{\mathtt{R}}(\Psi^2)$. Wenn $\mathtt{Trans}_{\mathtt{R}}(\Psi^)$ und $\mathtt{Trans}_{\mathtt{R}}(\Psi^2)$ je mit einer rechts-finalen Markierung markiert sind, so folgt aus der Induktionsvoraussetzung, dass die beiden internen Plätze irgendwann mit exakt einem Term markiert waren. Das bedeutet, die $\tau$-Transition hat im Vorfeld geschaltet. Dies kann nur der Fall sein, wenn der Initialplatz initial mit genau einem Term markiert war. Somit ist $\mathtt{Trans}_{\mathtt{R}}(\Psi^1 \wedge \Psi^2)$ initial mit einer rechts-initialen Markierung $M_0^{\mathtt{Trans}_{\mathtt{R}}(\Psi^1 \wedge \Psi^2)}$ markiert.

**⇐:** Angenommen $\mathtt{Trans}_{\mathtt{R}}(\Psi^1 \wedge \Psi^2)$ ist initial mit einer rechts-initialen Markierung $M_0^{\mathtt{Trans}_{\mathtt{R}}(\Psi^1 \wedge \Psi^2)}$ markiert. Nach Übersetzungsregel 13 (Tab. 5.5) schaltet die $\tau-$Transition und beide internen Plätze sind daraufhin mit genau einem Term markiert. Da diese Markierung je eine rechts-initiale Markierung von $\mathtt{Trans}_{\mathtt{R}}(\Psi^1)$ und von $\mathtt{Trans}_{\mathtt{R}}(\Psi^2)$ ist, folgt aus der Induktionsvoraussetzung, dass jeweils eine rechts-finale Markierung erreichbar ist und dass gilt:

$(\texttt{Trans}_{\texttt{R}}(\Psi^1), M_\Omega^{\texttt{Trans}_{\texttt{R}}(\Psi^1)}) \models \Psi^1$ und
$(\texttt{Trans}_{\texttt{R}}(\Psi^2), M_\Omega^{\texttt{Trans}_{\texttt{R}}(\Psi^2)}) \models \Psi^2$.

Das bedeutet, $\texttt{Trans}_{\texttt{R}}(\Psi^1 \wedge \Psi^2)$ ist mit einer rechts-finalen Markierung $M_\Omega^{\texttt{Trans}_{\texttt{R}}(\Psi^1 \wedge \Psi^2)}$ markiert und es gilt:
$(\texttt{Trans}_{\texttt{R}}(\Psi^1 \wedge \Psi^2), M_\Omega^{\texttt{Trans}_{\texttt{R}}(\Psi^1 \wedge \Psi^2)}) \models \Psi^1 \wedge \Psi^2$.

**Resultat:** Für $\Psi = k\ \Psi'$ und $\Psi = \Psi^1 \wedge \Psi^2$ gilt:

$$(\texttt{Trans}_{\texttt{R}}(\Psi), M_\Omega^{\texttt{Trans}_{\texttt{R}}(\Psi)}) \models \Psi \Leftrightarrow$$
$\texttt{Trans}_{\texttt{R}}(\Psi)$ ist initial mit einer rechts-initialen Markierung $M_0^{\texttt{Trans}_{\texttt{R}}(\Psi)}$ markiert.

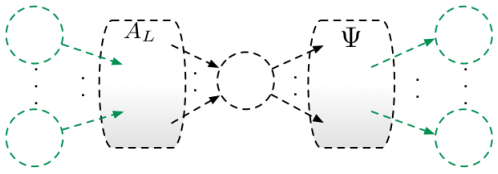
□

### 5.2.3 Übersetzung von RCTL - Formeln

**Definition 5.21** (RCTL-Übersetzung)**.**
*Sei AP die Menge atomarer Propositionen von RCTL und sei* $\Phi$ *die Menge von RCTL-Formeln über AP. Die* ***RCTL-Übersetzung*** *ist eine Funktion* $\texttt{Trans} : \Phi \rightarrow RCTL - Netze$. *Ihre Definition folgt den in Tabelle 5.6 angegebenen Regeln.*

#### Übersetzungsregeln

**Tabelle 5.6:** *Trans*($\varphi$)

| | RCTL | RCTL-Netz |
|---|---|---|
| 14 | $\varphi = TRUE$ | leeres RCTL-Netz |
| 15 | $\varphi = A_L \rightarrow \Psi$ |  |



#### Korrektheit

**Theorem 5.22.**
*Sei* $\varphi \in \Phi$ *eine RCTL-Formel und sei* $\texttt{Trans}(\varphi)$ *das aus der RCTL-Übersetzung von* $\varphi$ *resultierende RCTL-Netz.*

*Es gilt:*

$$\texttt{Trans}(\varphi) \models \varphi.$$

*Beweis.*

$\varphi = TRUE$:
Entsprechend der Übersetzungsregel 14 (Tab. 5.6), wird ein leeres RCTL-Netz produziert, wenn $\varphi = TRUE$.

In diesem trivialen Fall ist offensichtlich, dass

$$\texttt{Trans}(\varphi) \models \varphi$$

gilt, da die Menge erreichbarer Markierungen von $\texttt{Trans}(\varphi)$ leer ist.

$\varphi = A_L \rightarrow \Psi$:
Um zu beweisen, dass

$$\texttt{Trans}(A_L \rightarrow \Psi) \models A_L \rightarrow \Psi$$

gilt, müssen wir zeigen, dass folgendes gilt.

$$\texttt{Trans}(A_L) \models A_L \text{ impliziert } \texttt{Trans}(\Psi) \models \Psi.$$

Das bedeutet, immer wenn $\texttt{Trans}(A_L) \models A_L$ gilt, dann gilt auch $\texttt{Trans}(\Psi) \models \Psi$.

Entsprechend der Übersetzungsregel 15 (Tab. 5.6) gilt:

$$\texttt{Trans}(\varphi) = \texttt{Trans}_{\texttt{L}}(A_L) \,\hat{\otimes}\, \texttt{Trans}_{\texttt{R}}(\Psi),$$

Hierbei verschmilzt der Operator $\hat{\otimes}$ den Finalplatz von $\texttt{Trans}_{\texttt{L}}(A_L)$ mit dem Initialplatz von $\texttt{Trans}_{\texttt{R}}(\Psi)$.

Angenommen $\texttt{Trans}_{\texttt{L}}(A_L) \models A_L$ gilt. Dann folgt aus Theorem 5.13, dass der Finalplatz von $\texttt{Trans}_{\texttt{L}}(A_L)$ irgendwann mit genau einem Term markiert ist. Da dieser Platz gleichzeitig der Initialplatz von $\texttt{Trans}_{\texttt{R}}(\Psi)$ ist und es sich hierbei um eine rechts-initiale Markierung von $\texttt{Trans}_{\texttt{R}}(\Psi)$ handelt, folgt aus Theorem 5.20, dass $\texttt{Trans}_{\texttt{R}}(\Psi) \models \Psi$ gilt.

Somit gilt für $\varphi = A_L \rightarrow \Psi$:

$$\texttt{Trans}(\varphi) \models \varphi$$

□

### Terminierung

Die Übersetzung erfolgt rekursiv. Der Algorithmus terminiert, wenn die atomaren Propositionen übersetzt werden. Für jedes Element der Formel wird genau eine Regel genutzt. Somit ist der Aufwand des Übersetzungsprozesses linear in der Anzahl der Elemente der Formel. Das bedeutet, für eine Formel mit einer endlichen Menge an Elementen terminiert der Übersetzungsprozess .

### Beispiel

Um eine Intuition für den Übersetzungsprozess zu bekommen, betrachten wir Beispiel 5.5.

**Beispiel 5.5.** (Übersetzung)

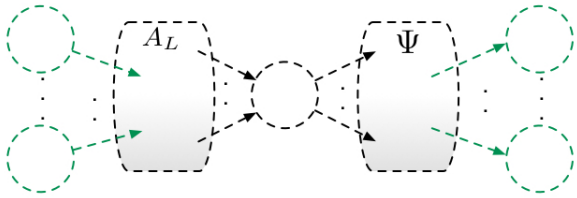
Angenommen wir wollen die folgende RCTL-Formel übersetzen:

$$\varphi = s_4.x_a \wedge (p_2.x \wedge p_1.y) \rightarrow AF(AXp_3.x' \wedge AF(p_5.z \wedge (z \geq zero)))$$

**Schritt 1:**

Wende **Regel 15** (Tab. 5.6) für :
$A_L = s_4.x_a \wedge (p_2.x \wedge p_1.y)$ und
$\Psi = AF(AXp_3.x' \wedge AF(p_5.z \wedge (z \geq zero))$

$N_\varphi$ :

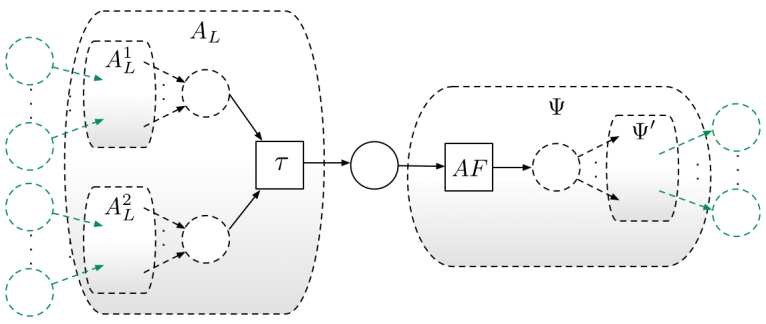



**Schritt 2:**

Wende **Regel 5** (Tab. 5.2) an auf:
$A_L = A_L^1 \wedge A_L^2$ mit
$A_L^1 = s_4.x_a$ und $A_L^2 = (p_2.x \wedge p_1.y)$

Wende **Regel 12** (Tab. 5.5) an auf:
$\Psi = k\ \Psi'$ mit
$k = AF$ und $\Psi' = (AXp_3.x' \wedge AF(p_5.z \wedge (z \geq zero))$

$N_\varphi$ :

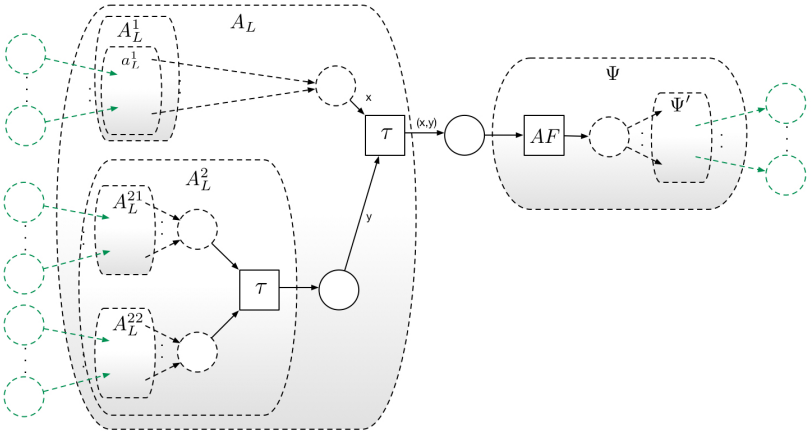


**Schritt 3:**

Wende **Regel 4** (Tab. 5.2) an auf:
$A_L^1 = a_L = s_4.x_a$

Wende **Regel 5** (Tab. 5.2) an auf
$A_L^2 = A_L^{2,1} \wedge A_L^{2,2}$ mit
$A_L^{2,1} = p_2.x$ und $A_L^{2,2} = p_1.y$

$N_\varphi$ :

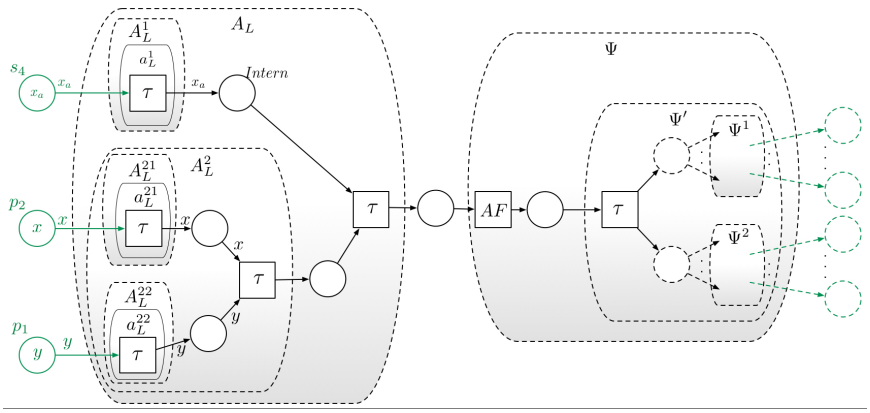



**Schritt 4:**

Wende **Regel 1** (Tab. 5.1) an auf:
$a_L = s_4.x_a$

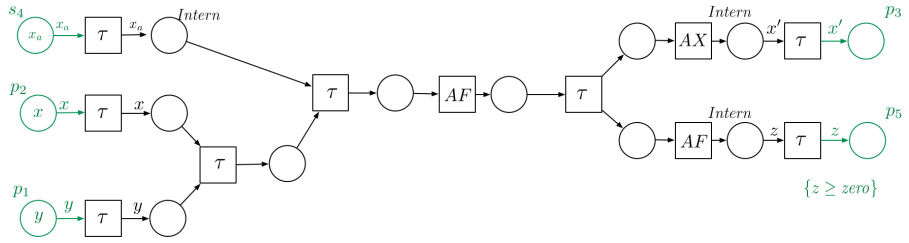
Wende **Regel 13** (Tab. 5.5) an auf :
$\Psi' = \Psi'_1 \wedge \Psi'_2$ mit
$\Psi'_1 = AXp_3.x'$ und $\Psi'_2 = AF(p_5.z \wedge (z \geq zero))$

$N_\varphi$ :




...

**Resultat:**
$N_\varphi$




## 5.2.4 Optimierungsmöglichkeiten

Bei der Definition der Übersetzungsregeln haben wir uns auf eine breite und allgemein gültige Anwendbarkeit konzentriert. Aus diesem Grund werden an einigen Stellen größere Netze als notwendig erzeugt. Es existieren daher Optimierungsmöglichkeiten, die die Komplexität einiger Netze verringern können. Die Optimierungen können dabei zum einen während der Übersetzung und zum anderen im Nachhinein durchgeführt werden.

### Optimierung während der Übersetzung

Das Ziel der Optimierung während der Übersetzung ist es, die Konstruktion unnötiger $\tau$-Transitionen und interner Plätze soweit wie möglich zu vermeiden. Dafür nutzen wir Zusammenhänge zwischen den temporalen-Pfad Operatoren.

Denn für $k_1, k_2 \in \{AG, AX\}$ gilt basierend auf Definition 2.68:
$k_1 = k_2 \implies k_1(\Psi_1 \wedge \Psi_2) = k_1\Psi_1 \wedge k_2\Psi_2$.

Dies kann genutzt werden, um $k_1(\Psi_1 \wedge \Psi_2)$ durch $(k_1\Psi_1 \wedge k_1\Psi_2)$ zu ersetzen und damit die Anzahl der konstruierten Plätze zu reduzieren. Das erhöht die Übersichtlichkeit.

### Optimierung nach der Übersetzung

Das Ziel bei der Optimierung nach durchgeführter Übersetzung besteht darin, unnötige $\tau$-Transitionen zu entfernen. Eine $\tau$-Transition ist unnötig, wenn sie direkt vor oder nach einer Transition auftritt, die mit einem temporalen Pfad-Operator gelabelt ist, und diese Transition genau einen Platz im Vor- oder Nachbereich aufweist. In diesem Fall kann die $\tau$-Transition entfernt werden. Dies wird mit Hilfe von Algorithmus 5 auf Seite 111 realisiert.

Die Hauptidee dieses Algorithmus' ist die Entfernung von $\tau$-Transitionen durch Umbiegen von Kanten, die mit internen Plätzen verbunden sind. Da-

**Algorithmus 5 :** Optimierung - Entfernung von $\tau$-Transitionen

**Eingabe** : Das zu optimierende RCTL-Netz $N$
**Ausgabe** : Optimiertes RCTL-Netz

```
1  for alle τ-Transitionen t, die mit genau einem internen Platz p
   verbunden, also für die (t,p) ∈ F^N gilt, und p nur eine eingehende
   Kante besitzt do
2      % umbiegen von Kanten und löschen der unnötigen
       τ-Transitionen
3      for alle p' ∈ •t do
4          % bei Verzweigungen, die in dem internen Platz
           beginnen
5          for alle t' mit (p,t') ∈ F^N do
6              füge (p',t') mit der Kantenbeschriftung von (p',t) zu F^N
               hinzu
7              füge den Guard von t zu t' hinzu
8              entferne (p',t) und (p,t') aus N
9      entferne t, p und (p,t) aus N
10 for alle internen Plätze p, die mit genau einer τ-Transition t verbunden
   sind, also für die (p,t) ∈ F^N gilt, und für die t nur einen eingehende
   Kante besitzt do
11     %umbiegen von Kanten und löschen der unnötigen
       τ-Transitionen
12     for alle t' mit (t',p) ∈ F^N do
13         % bei Verzweigungen, die in der τ-Transition beginnen
14         for alle p' mit (t,p') ∈ F^N do
15             füge (t',p') mit Kantenbeschriftung von (t,p') zu F^N hinzu
16             füge Guard von t zu t' hinzu
17             entferne (t',p) und (t,p') aus N
18     entferne t, p und (p,t) aus N
19 return N
```

bei betrachten wir zwei Fälle. Im ersten Fall (Zeile 1-9 in Alg. 5) werden alle auftretenden Verbindungen von $\tau$-Transitionen mit einem internen Platz behandelt. Hierbei werden die Input-Plätze $p'$ einer $\tau$-Transition $t$ mit allen Output-Transitionen $t'$ des internen Platzes $p$ verbunden.

Der zweite Fall ist ab Zeile 9 in Alg. 5 dargestellt. Im zweiten Fall (Zeile 10-18 in Alg. 5) betrachten wir jeden internen Platz $p$ i der mit genau einer $\tau$-Transition verbunden ist. Für diesen werden alle Input-Transitionen $t'$ des internen Platzes mit den Output-Plätzen der $\tau$-Transition verbunden.

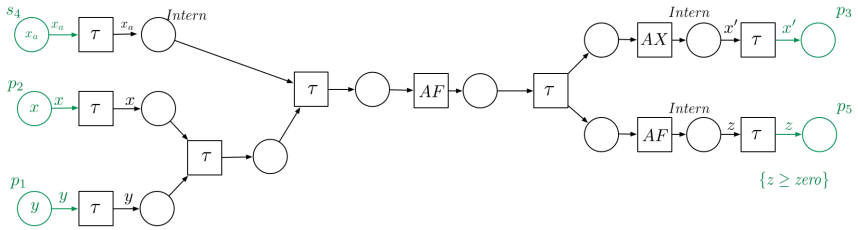
In beiden Fällen werden die behandelten $\tau$-Transitionen und internen Plätze aus dem RCTL-Netz $N$ entfernt, bevor es zurückgegeben wird.

Dieses Vorgehen veranschaulicht das folgenden Beispiel.

**Beispiel 5.6.** (Fortsetzung von Beispiel 5.5 von Seite 108)

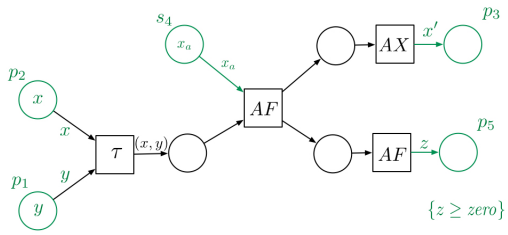
Als kurze Erinnerung zeigen wir im folgenden noch einmal das RCTL-Netz, das aus der Übersetzung in Beispiel 5.5 von Seite 108 resultierte.

$N_\varphi$ :




Auf dieses Netz können wir die *Optimierung nach der Übersetzung* (Alg. 5) anwenden. Die führt zu dem folgenden, verkleinerten Netz.

$N'_\varphi$ :



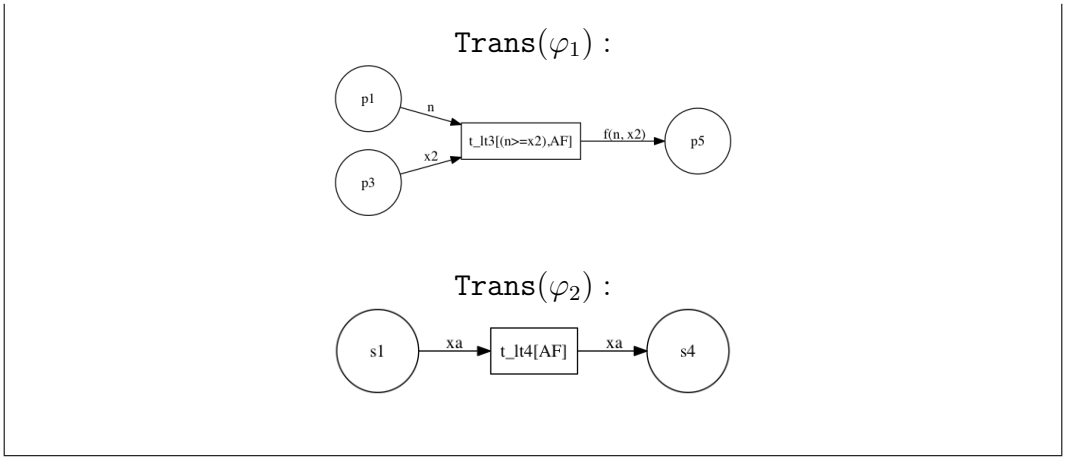

Wir haben die Übersetzung und die Optimierung implementiert und auf unser laufendes Beispiel angewandt. Das folgende Beispiel zeigt die automatisch erzeugten RCTL-Netze zweier Formeln.

**Beispiel 5.7. (Laufendes Beispiel)**

Gegeben seien die beiden Formeln:

$$\varphi_1 = p_1.n \wedge p_3.x_2 \wedge (n \geq x_2) \rightarrow AF(p5.f(n, x_2))$$
$$\varphi_2 = s_1.x_a \rightarrow AF(s_4.x_a)$$

Die Anwendung unserer Übersetzungsfunktion führt zu den folgenden zwei RCTL-Netzen. Die Grafiken wurden hierbei mit Dot(Graphviz) [GKN15] erzeugt.




## 5.3 Zurückführung auf ein APN

Da das Hauptziel unseres Ansatzes die automatische Synthese eines Controllers ist, der die korrekte Interoperabilität zwei gegebener Services gewährleistet, sollten der Controller und die Services mit dem selben Formalismus dargestellt werden. Aus diesem Grund stellen wir in Abschnitt 5.3.1 einen Algorithmus vor, mit dessen Hilfe wir ein gegebenes RCTL-Netz auf ein APN zurückführen können. Anschließend diskutieren wir dessen Korrektheit.

### 5.3.1 Zurückführung eines RCTL-Netzes auf ein APN

Bei der Zurückführung eines RCTL-Netzes $C$ auf ein APN $C'$ müssen wir sicher stellen, dass das Verhalten von $C$ durch $C'$ simuliert werden kann. Da in den Labels der Transitionen ein spezielles Verhalten kodiert ist, genügt es hierbei nicht, nur die Label zu streichen.

**Algorithmus 6 :** RCTLN2APN($C$)

**Eingabe** : RCTL-Netz $C$
**Ausgabe** : APN $C'$, das das Verhalten von $C$ simuliert

```
1 C' = C
2 for alle Transitionen t ∈ C mit Label L(t) = AG oder L(t) = EG do
3 |  C' = EntferneLabelAGundEG(C) %Alg.7
4 for alle Transitionen t ∈ C mit Label L(t) = AX oder L(t) = EX do
5 |  C' = EntferneLabelAXundEX(C') %Alg.9
6 for alle Transitionen t ∈ C mit Label L(t) = AF oder L(t) = EF do
7 |  C' = EntferneLabelAFundEF(C') %Alg.10
8 return C'
```

Unterschiedliche Arten von Labels werden auf unterschiedliche Weise entfernt. Wie in Algorithmus 6 dargestellt ist, beginnen wir den Prozess mit der

Entfernung der Label $AG$ und $EG$, da diese die restriktivsten sind. Anschließend werden die Label $AX$ und $EX$ behandelt. Zum Schluss entfernen wir noch die Label $AF$ und $EF$. In den folgenden Unterabschnitten stellen wir die einzelnen Algorithmen detaillierter vor.

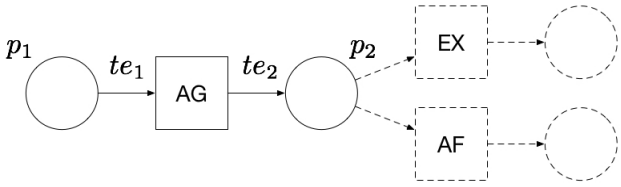
## Entfernung der Labels $AG$ und $EG$

Bevor wir unseren Algorithmus zur Entfernung der Labels $AG$ und $EG$ vorstellen, geben wir eine Intuition für die zu Grunde liegende Idee mittels folgenden Beispiels.

**Beispiel 5.8.**

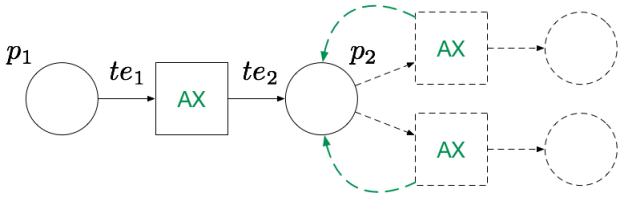
Gegeben sei des folgende RCTL-Netz.

$C$ :




Die Entfernung der Labels $AG$ bzw. $EG$ ist eher eine Substitution der temporalen Pfad-Operatoren. Diese Labels werden in den Transitionen zunächst durch $AX$ bzw. $EX$ ersetzt. Danach werden alle Labels der Transitionen aktualisiert, die in unserem Beispiel mit $p_2$ verbunden sind. Wenn es sich bei dem originalen Label um $AG$ handelt, so werden alle Pfad-Operatoren in den verbundenen Transitionen durch $A$ substituiert. Somit wird aus dem Label $EX$ das Label $AX$. Zusätzlich werden die Vorkommen der temporalen Operatoren $F$ durch $X$ ersetzt. Dadurch wird aus dem Label $AF$ das Label $AX$. Abschließend werden neue Kanten von den angepassten Transitionen zu $p_2$ erzeugt. Dies führt zu dem folgenden Netz $C'$, das das Verhalten von $C$ simuliert.

$C'$ :




Der folgende Algorithmus fasst dieses Vorgehen zusammen.

Die Labels werden wie folgt angepasst.

**Algorithmus 7 :** EntferneLabels$AG$und$EG$

**Eingabe** : RCTL-Netz $C$
**Ausgabe** : RCTL-Netz $C'$, das das Verhalten von $C$ simuliert ohne Labels $AG$ und $EG$ .

```
1  C' = kopiere(C)
2  for alle Transitionen t mit Label (L) AG oder EG do
3      if L(t) = AG then
4          substituiere Label von t durch AX
5      else
6          substituiere Label von t durch EX
7      for alle p ∈ t• do
8          for alle Transitionen t' mit (p, t') ∈ F^C do
9              anpassenDerLabels(t', L(t)) % wende Algorithm 8 an
10             füge (t', p) zu F^C' hinzu % falls (t', p) ∉ F^C'
11             füge entsprechende Kantenbeschriftung hinzu
12 return C'
```

**Algorithmus 8 :** anpassenDerLabels(t,l)

**Eingabe** : Transition $t'$, dessen Label angepasst werden soll, Label $l$, das als Referenz-Label dient.
**Ausgabe** : RCTL-Netz $C'$ mit angepassten Labels.

```
1  C' ← globaler Zugriff auf das Netz
2  if l = AG then
3      if L(t) = Ek mit k ∈ {X, F, G} then
4          L(t) = Ak
5  if L(t) = oF mit o ∈ {A, E} then
6      L(t) = oX
7  for all p' ∈ t• do
8      for alle t'' mit (p', t'') ∈ F^C' do
9          anpassenDerLabels(t'', l)
10 return C'
```

Der Algorithmus 7 überprüft alle Transitionen hinsichtlich eines Labels $AG$ bzw. $EG$. Ein solches Label wird durch das Label $AX$ (wenn das Originallabel $AG$ ist) bzw. $EX$ (sonst) (Zeilen 3-6). Anschließend werden die Labels aller nachfolgenden Transitionen durch die Anwendung von Algorithmus 8 angepasst.

Zusätzlich werden neue Kanten mit Beschriftungen erzeugt und dem Netz hinzugefügt (Zeilen 8-13).

Das Basiskonzept des Label-Anpassunsalgorithmus' (Alg. 8) ist eine Tiefensuche, bei der die Anpassungen stets stattfinden, bevor weiter in die Tiefe gegangen wird. Der Algorithmus passt zunächst die Pfad-Operatoren an, falls

das Referenz-Label einen Allquantor aufweist (Zeilen 2-4). Im Anschluss daran werden Transitionen hinsichtlich des temporalen Operators $F$ analysiert. Wird ein solcher gefunden, ändert der Algorithmus ihn in den Operator $X$ (Zeilen 5-6). Abschließend wird für alle Ausgangsplätze der betrachteten Transition der Algorithmus (Alg. 8) rekursiv aufgerufen (Zeilen 10-12).

Da wir die Labels $AG$ bzw. $EG$ nur mit $AX$ bzw. $EX$ substituieren, sind wir an diesem Punkt noch nicht fertig, sondern müssen die Behandlung dieser Labels ebenfalls durchführen.

## Entfernung der Labels $AX$ und $EX$

Wenn wir die Labels $AX$ und $EX$ entfernen wollen, müssen wir sicher stellen, dass alle mit $X$ gelabelten, aktiven Transitionen zur selben Zeit schalten. Aus diesem Grund markieren wir diese Transitionen und löschen die Label. Dies wird in Algorithmus 9 veranschaulicht.

**Algorithmus 9 :** Entferne Labels $AX$ und $EX$

**Eingabe** : RCTL-Netz $C$
**Ausgabe** : RCTL-Netz $C'$, ohne $AX$ und $EX$ Labels

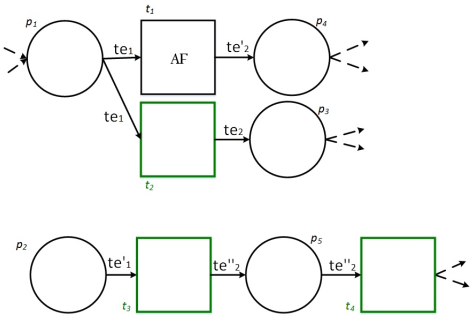
```
1 for alle Transitionen t mit Label (l) AX oder EX do
2 |   markiere Transition
3 |   lösche Label aus t
4 return C'
```

An dieser Stelle verzichten wir darauf einen expliziten Algorithmus zur gleichzeitigen Ausführung der markierten Transitionen zu präsentieren. Die Grundidee unserer Implementierung beruht darauf, dass wir für jede mögliche Kombination der markierten Transitionen eine neue Transition erzeugen und entsprechende Kanten erzeugen. Diese Idee wird in folgendem Beispiel veranschaulicht.

**Beispiel 5.9.** (Entfernung der Labels $AX$ und $EX$ )

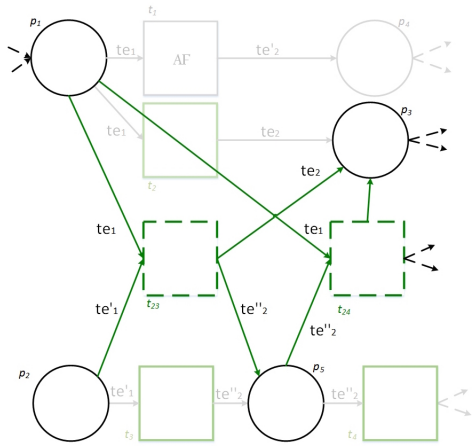
Angenommen das folgende Netz ist das Resultat von Algorithmus 9. Der Algorithmus entfernt die Labels $AX$ und $EX$ und markiert die entsprechenden Transitionen (im Beispielnetz grün dargestellt).

$C$ :




Im nächsten Schritt fügen wir für jede Permutation der grünen Transitionen eine neue Transition und die entsprechenden Kanten hinzu. Im Beispiel wird demnach eine neue Transition $t_{23}$ für die Kombination der Transitionen $t_2$ und $t_3$ sowie eine neue Transition $t_{24}$ für die Kombination der Transitionen $t_2$ und $t_4$ erstellt und mit den entsprechenden Plätzen über Kanten verbunden. Es werden dabei auch Inhibitor-Kanten erzeugt (beispielsweise von $p_1$ zu $t_3$ sowie $p_2$ zu $t_2$) um die Ausführung der richtigen Transition zu forcieren. Diese Kanten werden aber aufgrund der Übersichtlichkeit nicht mit dargestellt.

$C'$ :




Die auf diese Weise konstruierten Transitionen repräsentieren die gemeinsame Ausführung der markierten Transitionen.

Nachdem wir die Labels $AG, EG, AX$ und $EX$ entfernt haben, betrachten wir nun die Labels $AF$ und $EF$.

## Entfernung der Labels $AF$ und $EF$

Um die Label $AF$ und $EF$ zu entfernen, müssen wir sicherstellen, dass die markierten Transitionen immer zuerst schalten, da diese eine höhere Priorität aufweisen. Wie in Algorithmus 10 dargestellt ist, lösen wir dieses Problem durch die Hinzunahme von Inhibitor-Kanten ausgehend von allen Eingangsplätzen der höher priorisierten Transition. Eine solche Kante wird nicht erzeugt, wenn der Eingangsplatz der höher priorisierten Transition auch ein Eingangsplatz von der $AF/EF$- Transition ($t$) ist. Zusätzlich stellen wir sicher, dass $t$ irgendwann schaltet. Dafür analysieren wir die markierten Transitionen hinsichtlich existierender Schleifen. Falls wir eine solche Schleife finden, forcieren wir eine gemeinsame Ausführung von $t$ und der in die Schleife involvierten Transitionen durch die Hinzunahme einer gemeinsamen Transition. Dies geschieht analog zu dem in Beispiel 5.9 beschriebenen Vorgehen.

**Algorithmus 10 :** EntferneLabels$AF$und$EF$

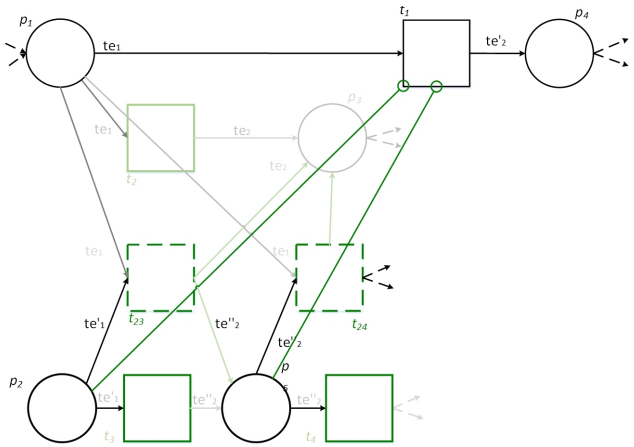
**Eingabe** : RCTL-Netz $C$, das aus Algorithmus 9 resultiert
**Ausgabe** : RCTL-Netz $C'$ mit entfernten Labels.

```
1  for alle Transitionen t mit Label (l) AF oder EF do
2      for alle markierten Transitionen t_green do
3          suche Schleifen
4          if Schleifen gefunden then
5              for alle in die Schleife involvierten Transitionen t_loop do
6                  behandle t als markierte Transition
7                  füge neue Transition tt_loop hinzu, die eine gemeinsame
                   Ausführung von t und t_loop forciert
8          else
9              for alle p ∈ •t_green ∧ !p ∈ •t do
10                 füge Inhibitor-Kanten von t_green zu t hinzu
11     entferne Label von t
12 return C'
```

Das Ergebnis dieses Algorithmus' ist in folgendem Beispiel dargestellt.

**Beispiel 5.10.** (Entfernung der Labels $AF$ und $EF$ )

Auf das in Beispiel 5.9 resultierende Netz $C$ wenden wir nun Algorithmus 10 an und entfernen damit das Label $AF$.




Die Transition $t_2$ wurde ausgegraut , da ihr einziger Eingangsplatz auch ein Eingangsplatz von $t_1$ ist und somit dafür keine Inhibitor-Kante hinzugefügt werden muss.

## Zusammenfassung des Entfernungsschrittes

In diesem Abschnitt haben wir einen Algorithmus vorgestellt, der die Entfernung der CTL-Label und somit die Zurückführung eines RCTL-Netzes auf ein APN erlaubt. Das bietet den großen Vorteil, dass wir existierende Tools zur Repräsentation und Analyse der APNs nutzen können.

Der Algorithmus selbst differenziert zwischen der Art der Labels, da das Verhalten erhalten bleiben muss, das in den Labels kodiert ist. Für jeden möglichen CTL-Operator haben wir einen eigenen Teilalgorithmus vorgestellt, der die entsprechenden Charakteristiken behandelt.

Zu dem Zeitpunkt an dem wir die Labels löschen, unterscheiden wir nicht mehr zwischen den einzelnen Pfad-Operatoren, sondern behandeln alle als Allquantoren. Dies ist zulässig, da Allquantoren eine Überapproximation der Existenzquantoren sind.

Zusätzlich dazu nutzen wir Inhibitor-Kanten um eine Art Ordnung auf den Schaltsequenzen zu definieren. Damit können wir sicherstellen, dass zunächst die Transitionen schalten, die im RCTL-Netz nicht mit $AF/EF$ gelabelt sind. Weitere Möglichkeiten, Transitionen zu priorisieren ohne den Formalismus zu signifikant zu ändern, sind derzeit nicht bekannt.

### 5.3.2 Korrektheit

Im Folgenden diskutieren wir die Korrektheit unseres Algorithmus'. Korrekt bedeutet in unserem Sinne, dass eine Formel, die im RCTL-Netz gilt, auch in

dem APN gilt, das aus unserem Algorithmus resultiert. Wir zeigen, dass das folgende Theorem gilt.

**Theorem 5.23.**
*Sei $C$ ein RCTL-Netz und sei $C'$ das APN, das aus der Anwendung von Algorithmus 6 auf $C$ resultiert. Sei $\varphi \in \Phi$ eine RCTL-Formel. Dann gilt:*

$$(C \models \varphi) \implies (C' \models \varphi)$$

*Beweis.*
Angenommen $\varphi$ ist gültig in $C$. Wenn wir zeigen wollen, dass in diesem Fall $\varphi$ auch gültig in $C'$ ist, müssen wir zeigen, dass das Verhalten von $C$ durch $C'$ simuliert wird. Das bedeutet wir können für alle Transitionen $t$, die mit einem temporalen Pfad-Operator in $C$ gelabelt sind, zeigen, dass deren Semantik in $C'$ erhalten bleibt.

Im Folgenden diskutieren wir die verschiedenen Arten der Labels, die eine Transition $t$ aufweisen kann und zeigen damit, dass unser Algorithmus (Alg. 6) korrekt ist.

#### Labels AG/EG

Angenommen $C$ besitzt ein Subnetz, in dem Plätze $p_i, i \in \{1, .., n\}$ Eingangs-Plätze von $t$ und $q_j, j \in \{1, .., r\}$ Ausgangs-Plätze von $t$ sind. Zudem sei $L(t) = AG$. Wenn alle $p_i$ markiert sind, ist $t$ aktiviert und muss im nächsten Schritt schalten, da die Semantik von $AG$ das erfordert. Nachdem $t$ geschaltet hat, müssen auf allen Pfaden alle $q_j$ markiert bleiben.

Wenn wir das Label von $t$ entfernen, können drei Fehler auftauchen. Erstens wäre es möglich, dass ein Pfad existiert, auf dem nicht alle $q_j$ markiert sind. Zweiten wäre es möglich, dass es eine Transition gibt, die Marken von einem der Ausgangs-Plätze $q_j$ konsumiert ohne sie diesen erneut zu markieren. Drittens könnte es passieren, dass $t$ nicht im nächsten Schritt schaltet, da eine andere Transition ebenfalls aktiviert ist und zuerst schaltet.

Der erste Fall kann nur dann eintreten, wenn eine der auf $t$ folgenden Transitionen nicht sicherstellt, dass alle Pfade involviert sind. Wir behandeln diesen Fall durch die Anpassung der Labels aller nachfolgenden Transitionen. Dies geschieht durch unseren Algorithmus 8, der eine Tiefensuche darstellt und alle Existenzquantoren durch Allquantoren ersetzt. Somit gilt, dass die Labels der nachfolgenden Transitionen sicherstellen, dass die Markierung der $q_j$ auf allen Pfaden gilt, da die Substitution mit dem Allquantor eine Überapproximation darstellt.

Der zweite Fall kann eintreten, wenn eine Transition eine Marke von einem Platz $q_j$ konsumiert, die eigentlich erhalten bleiben muss. Um dies zu handhaben, fügen wir eine Rückwärtskante von jeder Transition $t'$ für die $(q_j, t') \in F^C$ zurück zu $q_j$ hinzu. Dadurch wird eine erneute Markierung von $q_j$ forciert, sobald $t'$ schaltet.

Da unser Algorithmus 7 das Label $AG$ durch $AX$ und $EG$ durch $EX$ ersetzt, kann es nicht passieren, dass die Transition $t$ nicht im nächsten Schritt schaltet. Dies wird durch den Algorithmus 9 sichergestellt.

#### Labels AX/EX

Im Falle, dass eine aktive Transition mit $AX$ oder $EX$ gelabelt ist, müssen wir sicherstellen, dass diese im nächsten Schritt schaltet und somit alle Ausgangs-Plätze der Transition markiert werden. Durch unseren Algorithmus werden die Labels entfernt und die entsprechenden Transitionen markiert. Danach wird für jede Permutation der markierten Transitionen eine neue produziert, die eine gleichzeitige Ausführung der in der Permutation involvierten Transitionen ermöglicht. Somit stellen wir sicher, dass alle Ausgangs-Plätze aller aktiven, markierten Transitionen durch einen Schaltschritt markiert werden.

Es ist jedoch möglich, dass es zusätzlich eine aktive Transition gibt, die mit $AF$ oder $EF$ gelabelt ist. Wenn eine dieser Transitionen vor einer $AX/EX$-Transition schaltet, würde dies einen Widerspruch zur Semantik der $AX/EX$-Label darstellen. Im Folgenden diskutieren wir, dass diese Situation nicht eintreten kann, da dieser Fall in Algorithmus 10 behandelt wird.

#### Labels AF/EF

Bis zu diesem Punkt haben wir die Korrektheit unseres Algorithmus' hinsichtlich der Behandlung der Label $AG/EG$ und $AX/EX$ diskutiert. Es gibt jedoch zwei Möglichkeiten, die eintreten und dabei unseren Algorithmus inkorrekt werden lassen können. Erstens könnte eine $AF/EF$-Transition schalten bevor eine aktive Transition (die zuvor mal eine $AX/EX$-Transition war) schaltet. Zweitens könnte es sein, dass eine $AF/EF$-Transition niemals schaltet.

Da wir Inhibitor-Kanten ausgehend von den Eingangs-Plätzen der markierten Transitionen zu den $AF/EF$-Transitionen hinzufügen, können die beiden Transitionsarten nicht gleichzeitig aktiviert sein. Genauer gesagt kann eine $AF/EF$-Transition nicht schalten sobald eine $AX/EX$-Transition aktiviert ist. Somit kann es nicht passieren, dass eine $AF/EF$-Transition schaltet, bevor es eine aktivierte $AX/EX$-Transition tut.

Für den zweiten Fall sei angemerkt, dass eine $AF/EF$-Transition niemals schalten kann, falls eine Schleife um eine markierte Transition existiert. Diese Schleife wird durch unseren Algorithmus entdeckt und forciert dabei die Behandlung einer $AF/EF$-Transition als markierte ($AX/EX$-) Transition. Entsprechend des $AX/EX$-Falles wird eine neue Transition erzeugt und hinzugefügt, die die gemeinsame Ausführung der in die Schleife involvierten Transitionen und der $AF/EF$-Transition sicherstellt. Dadurch kann es nicht passieren, dass eine $AF/EF$-Transition niemals schaltet.

#### Zusammenfassung

Da bei den Teilalgorithmen die Semantik der Label des RCTL-Netzes $C$ im APN $C'$ erhalten bleibt, muss Theorem 5.23 wahr sein. □

# Teil 3
# Controller-Synthese für Services mit Daten

# 6 Controller Synthese

## 6.1 Grundlegende Idee

Um automatisiert einen Controller für Services mit Daten zu synthetisieren und dabei spezifizierte Verhaltenseigenschaften sicherzustellen, haben wir einen fünfstufigen Ansatz entwickelt [BTGGB14, BTGB14, BTG14, BT14], der in Abbildung 6.1 visualisiert ist.

Dieser Ansatz kombiniert das durch ein APN modellierte Service-Verhalten mit den durch RCTL spezifizierten Verhaltenseigenschaften. Basierend auf diesen Eingaben, führen wir zunächst die *Extraktion beobachtbaren Verhaltens* durch. Dabei extrahieren wir eine Menge von RCTL-Formeln aus jedem der Service-Netze. Diese Formeln spezifizieren das Kommunikationsverhalten des jeweiligen Services. Sie werden zusammen mit den Verhaltenseigenschaften als Eingabe für den *Übersetzungsschritt* genutzt. In diesem wird jede RCTL-Formel in ein RCTL-Netz übersetzt. Diese einzelnen Netze werden im *Kompositionsschritt* derart kombiniert, dass in dem resultierenden Netz alle Verhaltenseigenschaften gelten. Das resultierende Netz wird als Eingabe des *Extraktionsschrittes* genutzt. Dabei werden alle Elemente, die für den Controller notwendig sind, aus dem Netz extrahiert und anschließend das resultierende RCTL-Netz auf ein APN zurückgeführt. Das so entstehende APN ist per Konstruktion korrekt. Es ist somit möglich, den Controller mit den Services zu komponieren und ihre Interoperabilität zu gewährleisten.

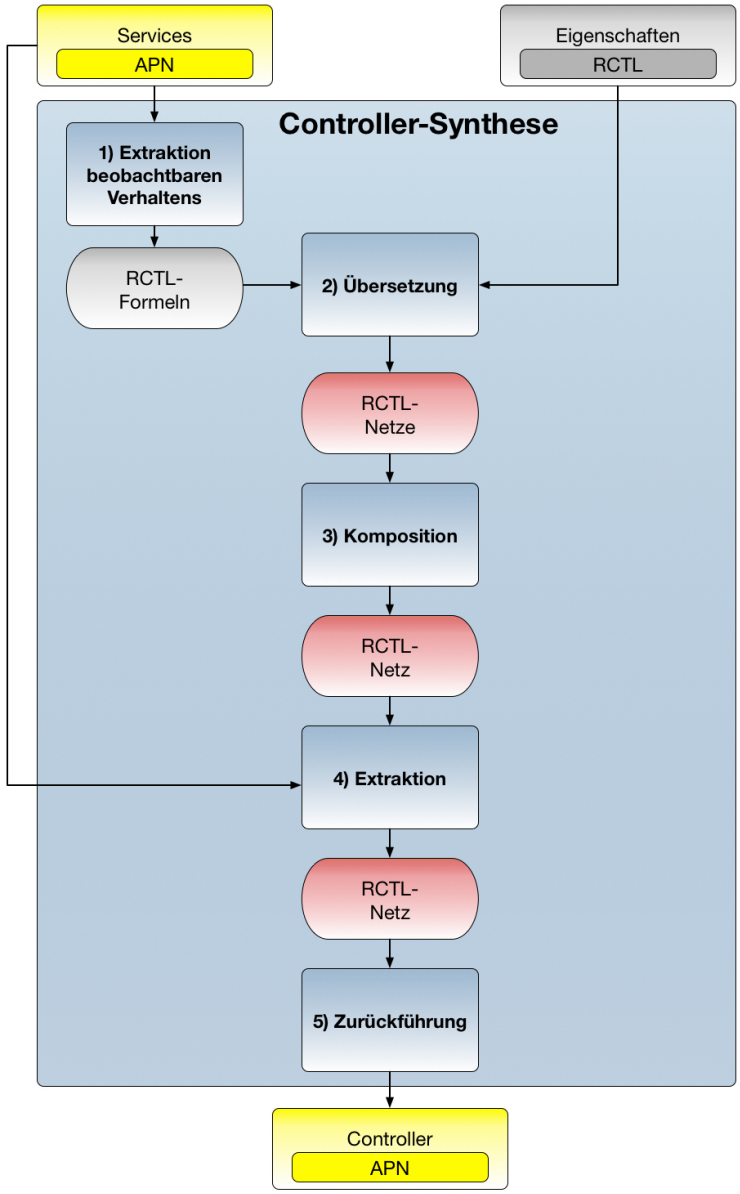



**Abbildung 6.1:** Grundlegende Idee des Controller-Synthese Prozesses

## 6.2 Phasen des Synthese Prozesses

Um ein besseres Verständis für unseren Ansatz zu erhalten, gehen wir im Folgenden auf die einzelnen Phasen etwas genauer ein und wenden sie auf unser laufendes Beispiel an.

**Beispiel 6.1.** (**Laufendes Beispiel** - Eingaben)

Nehmen wir an, dies sind die Eingaben für unseren Controller-Synthese Prozess:

**Service Verhalten:**

Wir nutzen die APN-Repräsentation aus dem in Abschnitt 2.4 vorgestellten laufenden Beispiel (siehe Abbildung 2.2 auf Seite 49).

**Eigenschaften:** Wir wollen einen Controller so synthetisieren, dass die folgenden Eigenschaften gelten:

$\boldsymbol{\varphi_1 = s_4.x \rightarrow AF(p_3.x')}$
Wann immer der Schalter einen Wert an die Umgebung sendet ($s_4.x$), soll dieser Wert immer irgendwann ($AF$) als Eingabewert für die Pumpe genutzt werden ($p_3.x'$).

$\boldsymbol{\varphi_2 = p_5.z \wedge z \leq Const_{limit} \rightarrow AX(C_{intern}.err)}$
Sollte die noch im Reservoir vorhandene Menge ($z$) des Medikamentes kleiner oder gleich einer vordefinierten Reservemenge ($Const_{limit}$) sein, muss der Controller unverzüglich eine Fehlernachricht erzeugen ($AX(C_{intern}.err)$).

$\boldsymbol{\varphi_3 = (C_{intern}.err) \rightarrow AF(AX(p_1.z_{new}) \wedge AF(s_1.z_{new}))}$
Wann immer eine Fehlernachricht vom Controller erzeugt wurde ($C_{intern}.err$), soll das Reservoir der Pumpe aufgefüllt ($AX(p_1.z_{new})$) und der Schalter irgendwann zurückgesetzt werden ($AF(s_1.z_{new})$).

### 6.2.1 Extraktion Beobachtbaren Verhaltens

Im ersten Schritt unseres Prozesses extrahieren wir für jeden Service das beobachtbare Verhalten, also das Verhalten an den Interface-Ports. Dabei wird für jedes algebraische Petrinetz, das das Verhalten eines Services modelliert, eine Menge von RCTL-Formeln extrahiert. Dies haben wir bereits in Abschnitt 5.1.1 erläutert und die Ergebnisse der Anwendung auf unser laufendes Beispiel gezeigt (siehe Beispiel 4 auf Seite 87).

In unserem Beispiel nutzen wir exemplarisch das APN ($\mathbf{N}$), das das Verhalten er Pumpe unseres laufenden Beispiels (Abbildung 2.2a auf Seite 49) modelliert. Dieses Netz ist ganz links in Abbildung 6.2 dargestellt.

Für ein solches Netz folgen nun zwei Reduktionsphasen um die entsprechenden RCTL-Formeln zu extrahieren.

**Phase1 :** In der ersten Phase wird nur das in die Kommunikation involvierte Verhalten übernommen. Für unser Beispiel bedeutet das, dass Teile des Netzes entfernt werden. Präziser gesagt, werden der Platz $p_7$ und die Transition $t_5$ sowie die entsprechenden Kanten entfernt. Zusätzlich dazu werden in einem Vorbereitungsschritt die Gleichungen der algebraischen Spezifikation dazu genutzt, Variablen und Terme zu substituieren. Zudem

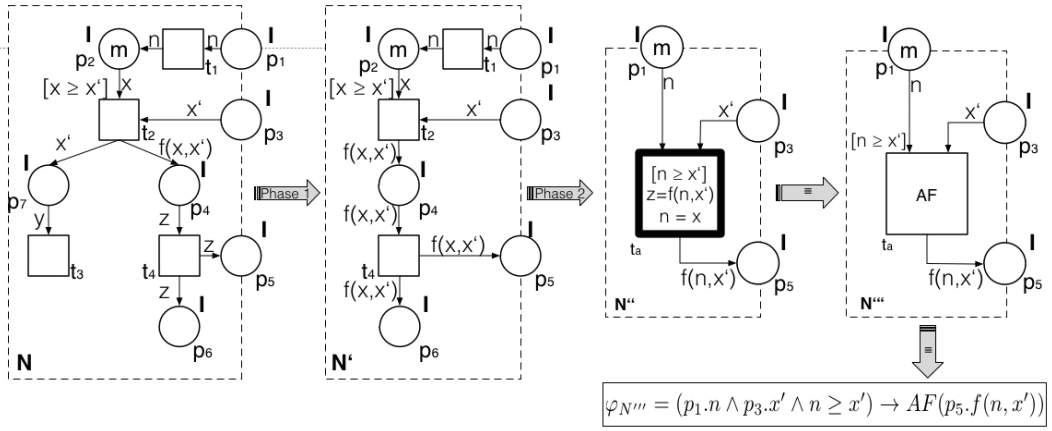



**Abbildung 6.2:** Extraktion beobachtbaren Verhaltens

werden Guards an die mit Interface-Plätzen verbundenen Transitionen verschoben. Das Resultat dieser Phase wird in Abbildung 6.2 durch das Netz $N'$ dargestellt.

**Phase 2:** In der zweiten Phase der Extraktion beobachtbaren Verhaltens, analysieren wir das aus Phase 1 resultierende Netz hinsichtlich Einschränkungen der Datenwertbereiche an den Interface-Ports. Dazu führen wir gedanklich einen weiteren Abstraktionsschritt durch, in dem interne Plätze und Transitionen zu einer einzigen Transition zusammengefügt werden. Diese Transition kann als eine hierarchische Transition angesehen werden, wie sie für gefärbte Petrinetze durch Jensen [Jen97] eingeführt wurden. Das bedeutet die hierarchische Transition kann verborgenes Verhalten aufweisen, was dazu führt, dass interne Schritte zwischen dem Konsum der Marken auf den Eingangsplätzen und dem Produzieren von Marken auf den Ausgangsplätzen vollzogen werden. Datenabhängigkeiten, die während des Abstraktionsschrittes auftreten, müssen extrahiert und gespeichert werden.

Um eine Intutition für die zweite Phase zu geben, betrachten wir erneut unser Beispiel. In dem Netz **N'** aus Abbildung 6.2 werden alle internen Zustände und Transitionen zu einer (hierarchischen) Transition zusammengefasst. Dabei werden Datenabhängigkeiten wie der Guard an Transition $t_2$ sowie die Definitionen von $z$ und $n$ intern gespeichert. Diese Informationen finden sich später in der extrahierten Formel wieder und schränken die Wertebereiche der über $p_5$ ausgetauschten Daten ein. Um das in der hierarchischen Transition verborgene Verhalten dennoch zu repräsentieren, wird die Transition mit einem temporalen Pfad-Operator gelabelt. Dabei ist das Label abhängig davon, ob die gespeicherten Datenabhängigkeiten immer (**AF**) einen Einfluss auf die gesendeten Nachrichten haben oder nur in mindestens einem Fall (**EF**). Das Resultat der zweiten Phase angewendet auf unser Beispiel ist in dem Netz ***N'''*** in Abbildung 6.2 repräsentiert. Daraus ergibt sich die folgende RCTL-Formel

$$APN2RCTL(N''') := (p_1.n \wedge p_3.x' \wedge n \geq x') \rightarrow AFp_5.f(n, x') \quad (6.1)$$

Diese Gleichung beschreibt die folgenden Zusammenhänge.

*Wann immer die Menge (n) des Medikamentes im Reservoir ($p_1$) der Pumpe größer ist als die gefragte Menge ($x'$), injiziert die Pumpe diese und sendet eine Nachricht ($p_5.f(n, x')$) an die Umgebung, die über die Restmenge im Reservoir informiert.*

### 6.2.2 Übersetzung

Die aus dem vorherigen Schritt resultierenden RCTL-Formeln werden zusammen mit den Eigenschaften als Eingabe für die Übersetzung genutzt. In diesem Schritt wird jede RCTL-Formel $\varphi$ in ein RCTL-Netz $N_\varphi$ übersetzt, so dass $\varphi$ in $N_\varphi$ erfüllt ist.

Die Anwendung unserer in Kapitel 5.2 vorgestellten ÃIJbersetzungs- und Optimierungsregeln liefert für die Formel $\varphi_2$ aus unserem laufenden Beispiel (siehe Beispiel 6.1) das folgende Netz $N_{\varphi_2} = Trans(\varphi_2)$, das in Abbildung 6.3.

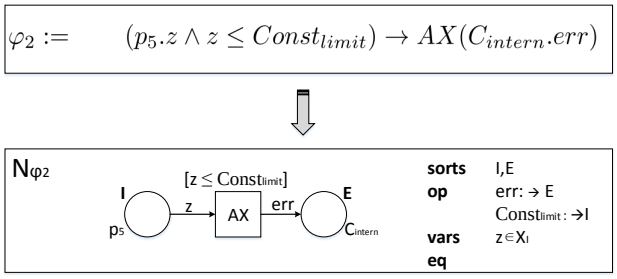



**Abbildung 6.3:** Übersetzung einer RCTL-Formel in ein RCTL-Netz

Dabei werden atomare Elemente der Formel in Elemente des RCTL-Netzes übersetzt. Für das Beispiel bedeutet das: $p_5.z$ wird in einen Platz mit Identifikator $p_5$ und eine Kante mit der Beschriftung $z$ übersetzt. Propositionen auf der linken Seite der Implikation, die einen Vergleichsoperator beinhalten, werden in Guards übersetzt, wie man am Beispiel $z \leq Const_{limit}$ sehen kannst. Variablen (wie z.B. $z$) und Konstanten (wie $err$ und $Const_{limit}$) werden in der algebraischen Spezifikation als Variablen (**vars**) und Operationen (**op**) gespeichert. Zusätzlich werden CTL-spezifische Operatoren, wie $AX$, als Transitionslabel in das RCTL-Netz übernommen.

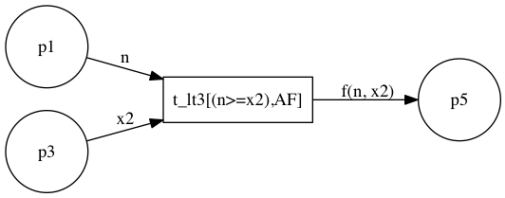
**Beispiel 6.2. (Laufendes Beispiel)**

Aus der Extraktion beobachtbaren Verhaltens ergeben sich für die Pumpe und den Schalter die folgenden RCTL-Formeln (siehe Beispiel 4 auf Seite 87).

$$\varphi_{Pumpe} = p_1.n \wedge p_3.x_2 \wedge (n \geq x_2) \rightarrow AF(p5.f(n, x_2))$$
$$\varphi_{Schalter} = s_1.x_a \rightarrow AF(s_4.x_a)$$

Wendet man darauf unsere Übersetzung an, so erhält man die folgenden RCTL-Netze.

$Trans(\varphi_{Pumpe})$:




$Trans(\varphi_{Schalter})$:

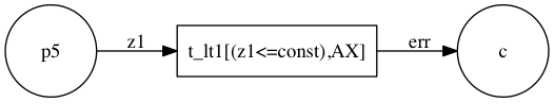

Zusätzlich haben wir zu Beginn dieses Kapitels in Beispiel 6.1 die folgenden Eigenschaften spezifiziert.

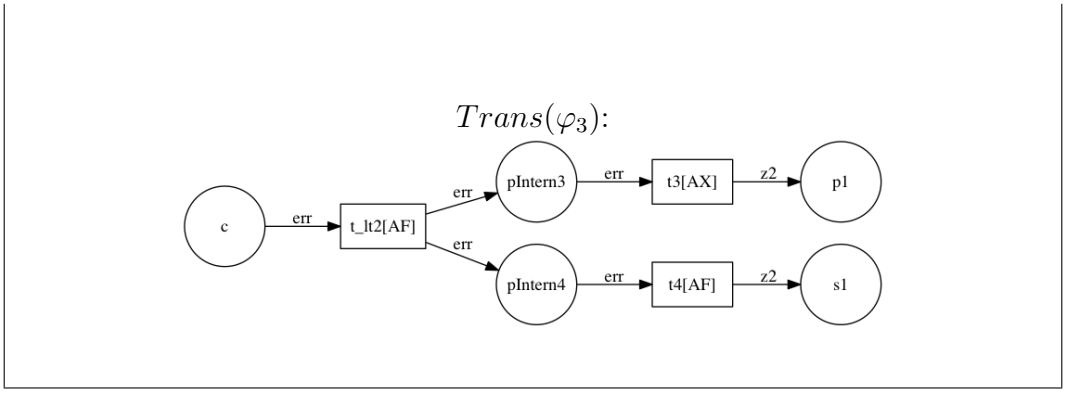
1. $\varphi_1 = s_4.x \rightarrow AF(p_3.x')$
2. $\varphi_2 = p_5.z \wedge z \leq Const_{limit} \rightarrow AX(C_{intern}.err)$
3. $\varphi_3 = (C_{intern}.err) \rightarrow AF(AX(p_1.z_{new}) \wedge AF(s_1.z_{new}))$

Wendet man unsere Übersetzung auf diese Eigenschaften an, so erhält man die folgenden RCTL-Netze.

$Trans(\varphi_1)$:


$Trans(\varphi_2)$:






### 6.2.3 Komposition

Nach der Übersetzung aller Eigenschaften werden die resultierenden Netze $(Trans(\varphi_{Pumpe}), Trans(\varphi_{Schalter}), Trans(\varphi_1), Trans(\varphi_2), Trans(\varphi_3))$ komponiert. Die Komposition ist dabei so gestaltet, dass alle Eigenschaften $(\varphi_{Pumpe}, \varphi_{Schalter}, \varphi_1, \varphi_2, \varphi_3)$ in dem komponierten Netz gelten.

Um dies umzusetzen, nutzen wir die in Kapitel 4.2.2 eingeführte Komposition von RCTL-Netzen. Für diese Komposition gilt das folgende Theorem:

**Theorem 6.1.**
*Seien $\varphi_1, \varphi_2 \in \Phi$ zwei RCTL-Formeln und seien $N_1, N_2$ zwei RCTL-Netze, sodass $N_i = Trans(\varphi_i), i \in \{1, 2\}$. Dann gilt:*

$$N_1 \otimes N_2 \models \varphi_1 \wedge \varphi_2$$

*Beweis.* Seien $\varphi_1, \varphi_2 \in \Phi$ zwei RCTL-Formeln und seien $N_1, N_2$ zwei RCTL-Netze, sodass $N_i = Trans(\varphi_i), i \in \{1, 2\}$.

Dann gilt bei Verwendung unseres Kompositionsalgorithmus (siehe Kapitel 4.2.2, Seite 68 ff.) :

$$N_1 \otimes N_2 \models \varphi_1 \wedge \varphi_2 \ .$$

#### Fall 1: Keine gemeinsamen Plätze

Wenn $N_1$ und $N_2$ keine gemeinsamen Plätze haben, wird mit unserem Kompositionsalgorithmus ein neues Netz $N'$ erzeugt, dass $N_1$ und $N_2$ als nebenläufige Netze beinhaltet. Da nach Theorem 5.22 $N_i \models \varphi_i, i \in \{1, 2\}$ gilt und $N_1$ und $N_2$ in $N'$ unverändert enthalten sind, gilt:

$$N_1 \otimes N_2 \models \varphi_1 \wedge \varphi_2$$

#### Fall 2: Sequentielle Komposition

Im Falle der sequentiellen Komposition wird ein neues Netz $N'$ erzeugt, in

dem die Finalplätze aus $N_1$ mit den Initialplätzen aus $N_2$ (oder andersherum) miteinander verschmolzen sind.

Da nach Voraussetzung $N_1 \models \varphi_1$ und $N_2 \models \varphi_2$ gilt, sind die Finalplätze von $N_1$ irgendwann markiert. Durch das Verschmelzen der Plätze werden darauffolgend auch die Finalplätze von $N_2$ in $N'$ belegt sein.

Daher gilt

$$N_1 \otimes N_2 \models \varphi_1 \wedge \varphi_2$$

**Fall 3: Verzweigende Komposition**
Die Argumentation erfolgt mittels Widerspruch. Dazu nehmen wir an, dass $N_1 \models \varphi_1$ und $N_2 \models \varphi_2$, jedoch $N_1 \otimes N_2 \not\models \varphi_1 \wedge \varphi_2$ gilt.

Das bedeutet, dass

$$N_1 \otimes N_2 \not\models \varphi_1$$
$$\text{oder}$$
$$N_1 \otimes N_2 \not\models \varphi_2.$$

gelten muss.

Dies kann nur eintreten, wenn die Ausführung eines der Netze die Ausführung des anderen Netzes verhindert.

Wir wissen, dass $N_1 \models \varphi_1$ und $N_2 \models \varphi_2$ gilt. In dem von uns definierten Kompositionsalgorithmus bleiben die Netze strukturell erhalten. Die gemeinsamen Plätze werden dupliziert, sodass eine Markierung vervielfältigt werden kann. Das führt dazu, dass keine um Token konkurrierende Situation eintritt.

Unter den gegebenen Voraussetzungen kann die Annahme nicht eintreten. Somit muss gelten:

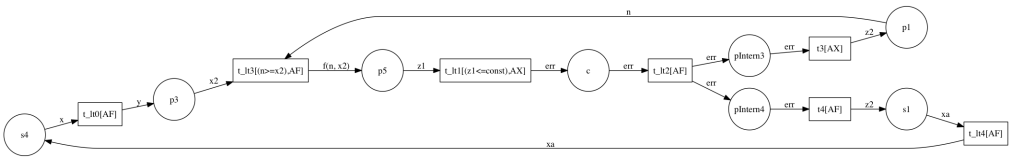
$$N_1 \otimes N_2 \models \varphi_1 \wedge \varphi_2.$$

□

Um eine Intuition für diesen Schritt zu bekommen, betrachten wir unser laufendes Beispiel.

**Beispiel 6.3. (Laufendes Beispiel - Komposition)**

Die aus dem Übersetzungschritt resultierenden RCTL-Netze (siehe Beispiel 6.2 auf Seite 130) werden als Eingaben für den *Kompositionsschritt* genutzt. Dieser folgt den Regeln, die in Kapitel 4.2.2 eingeführt wurden. In unserem Beispiel resultiert dieser Schritt in folgendem RCTL-Netz, das mittels Dot(Graphviz) visualisiert ist.




Die algebraische Spezifikation ist nicht Teil der graphischen Repräsentation, wird jedoch in einer Zwischendarstellung gespeichert und kann als Shell-Ausgabe betrachtet werden. Dies ist in der folgenden Abbildung dargestellt.

```
COMPOSITION2 -> algSpec:
SORTS:
D
T
I

 VARIABLES:
x_
y_
xa_D
x1_I
x2_I
n_I
z_I
y_I
err_

 OPERATIONS:
v:  -> D
 dot:  -> T
m:  -> I
 f: I I  -> I

 EQUATIONS:
 z==f(x1, x2)
 y==x2
 n==x1
```

## 6.2.4 Extraktion

Das aus der Komposition resultierende RCTL-Netz wird zusammen mit den original Service-Netzen in diesem Schritt genutzt um das RCTL-Netz zu extrahieren, das das Verhalten des gewünschten Controllers modelliert.

Die Notwendigkeit für diesen Schritt begründet sich in der Möglichkeit, dass das komponierte Netz Elemente aus den originalen Service-Netzen aufweist, was nicht Teil des Controllers sein darf. Daher extrahieren wir die Elemente aus dem komponierten Netz, die nicht in den Service-Netzen enthalten sind. Hierbei bilden die Interface-Plätze die einzige Ausnahme. Diese werden initial in das Controller-Netz übernommen, da sie die Schnittstellen für den Controller zu den Services darstellen. Ausgehend von diesen extrahiert Algorithmus 11 die Elemente des gewünschten Controllers.

**Algorithmus 11 :** Extract($EN^{Comp}, AN_{S_1}, AN_{S_2}$)

**Eingabe** :
$EN^{comp} = (P^{comp}, T^{comp}, F^{comp}, h^{comp}, M_0^{comp}, \lambda^{comp}, E^{comp}, L^{comp})$ das aus der Komposition resultierende RCTL-Netz
$AN_{S_1} = (P^{S_1}, T^{S_1}, F^{S_1}, h^{S_1}, M_0^{S_1}, \lambda^{S_1}, E^{S_1})$ das APN von Service $S_1$
$AN_{S_2} = (P^{S_2}, T^{S_2}, F^{S_2}, h^{S_2}, M_0^{S_2}, \lambda^{S_2}, E^{S_2})$ das APN von Service $S_2$
**Ausgabe** : Controller $C$ als RCTL-Netz

```
1  % Intitialisierung
2  C ← leeres RCTL-Netz
3  P_o^C ← P_i^{s1} ∪ P_i^{s2}
4  P_i^C ← P_o^{s1} ∪ P_o^{s2}
5  P^C ← P_o^C ∪ P_i^C
6  repeat ← TRUE
7  while repeat do
8      repeat ← FALSE
9      for f = (p,t) ∨ f = (t,p) ∈ F^comp do
10         if (p ∈ P^C ∧ t ∉ (T^C ∪ T^{S1} ∪ T^{S2}) then
11             T^C ← T^C ∪ {t}
12             F^C ← F^C ∪ {f}
13             repeat = TRUE
14         if t ∈ T^C ∧ p ∉ (P^C ∪ P^{S1} ∪ P^{S2}) then
15             P^C ← P^C ∪ {p}
16             F^C ← F^C ∪ {f}
17             repeat = TRUE
18         if p ∈ P^C ∧ t ∈ T^C ∧ f ∉ F^C then
19             F^C ← F^C ∪ {f}
20             repeat = TRUE
21         if p ∉ (P^C ∪ P^{S1} ∪ P^{S2}) ∧ t ∉ (T^C ∪ T^{S1} ∪ T^{S2}) ∧ f ∉ F^C then
22             P^C ← P^C ∪ {p}
23             T^C ← T^C ∪ {t}
24             F^C ← F^C ∪ {f}
25             repeat = TRUE
26 return C
```

### Algorithmus

Ausgehend von dem Netz $EN^{comp}$, das aus der Komposition resultiert, wird durch den Algorithmus ein RCTL-Netz erzeugt, dass das Verhalten des Controllers simuliert.

Der Algorithmus beginnt mit einer Initialisierungsphase. In dieser Phase wird ein leeres RCTL-Netz $C$ erzeugt (Zeile 2). Hierbei bedeutet leer, dass alle Mengen sowie die algebraische Spezifikation leer sind. Im nächsten Schritt werden die Interface-Plätze der Services $C$ hinzugefügt (Zeilen 3 - 5).

Nach der Initialisierung iterieren wir über die Kanten des Ausgangsnetzes und identifizieren die Plätze, Transitionen und Kanten, die zum Controller gehören sollen.

Solange eine Erweiterung des Controller-RCTL-Netzes stattfindet (zu erkennen an dem Flag *repeat*), durchlaufen wir alle Elemente aus $F^{comp}$ (Zeile 9). Für jede Kante $f \in F^{comp}$ mit $f = (p, t)$ or $f = (t, p)$ überprüfen wir ob folgendes gilt:

1. Platz $p$ ist bereits Teil von $C$ und die mit ihm verbundenen Transitionen sind weder bereits in $C$ noch in einem der Services vorhanden (Zeile 10). In diesem Fall fügen wir die entsprechende Transition $t$ zu der Menge der Transitionen des Controllers ($T^C$) und die Kante $f$ zur Flussrelation $F^C$ von $C$ hinzu (Zeilen 10-13).

2. Transition $t$ ist bereits in $C$ enthalten und einer der mit ihr verbundenen Plätze $p$ ist weder ein Element eines der Services noch ist $p$ in der Menge der Plätze von C (Zeile 14). In diesem Fall fügen wir $p$ zur Menge der der Plätze des Controllers ($P^C$) hinzu und erweitern $F^C$ um die Kante $f$ (Zeile 14-17).

3. Platz $p$ und Transition $t$ sind bereits in $C$ enthalten, die Kante zwischen ihnen jedoch nicht. In diesem Fall fügen wir die Kante $f$ zur Flussrelation $F^C$ von $C$ hinzu (Zeile 18-20).

4. Weder Platz $p$ noch Transition $t$ noch die Kante $f$ zwischen $p$ und $t$ sind bisher Teil von $C$, und keins davon ist Teil eines Services. In diesem Fall werden $p$,$t$ und $f$ zu $C$ hinzugefügt (Zeilen 21-24)

Wenn $C$ keine Änderungen mehr aufweist, wird es zurückgegeben und der Algorithmus terminiert (Zeile 26).

Es gelten die folgenden Definitionen:

1. Die Funktion $h^C$ , die jedem Platz aus $C$ ein Sortensymbol zuweist, ist definiert als $h^C = h^{comp}|_{p \in P^C}$.

2. Für die initiale Markierung gilt: $M_0^C(p) = \emptyset$ für alle $p \in P^C$.

3. Die Funktion $\lambda^C$, die jeder Kante aus $C$ eine Beschriftung zuweist, ist definiert als $\lambda^C = \lambda^{comp}|_{p \in P^C, t \in T^C}$.

4. Die Menge der Gleichungen aus $C$ ist definiert als $E^C = E^{comp}$. Das bedeutet alle Gleichungen bleiben bei der Extraktion erhalten.

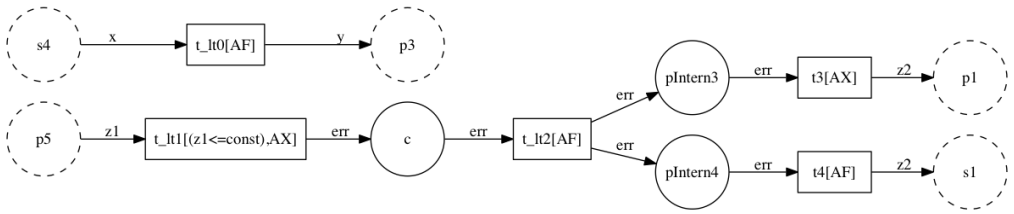
5. Die Funktion, die jeder Transition aus $C$ einen temporalen Pfad-Operator oder $\tau$ zuweist ist definiert als $L^C = L^{comp}|_{t \in T^C}$.

Hierbei bedeutet $A|_B$, dass die Definition von $A$ auf die Elemente von $B$ eingeschränkt wird.

**Beispiel 6.4. (Laufendes Beispiel)**

Sei $EN^{comp}$ das komponierte RCTL-Netz aus Beispiel 6.3 von Seite 132 . Die Anwendung von Algorithmus 11 auf $EN^{comp}$ resultiert in dem folgenden RCTL-Netz.

Extract($EN^{comp}$):




**Theorem 6.2.**
*Seien $S_1$ und $S_2$ zwei Service-APNs und sei $EN^{comp}$ das aus dem Kompositionsschritt resultierende RCTL-Netz. Dann beinhaltet das aus Algortithmus 11 resultierende Netz $C = Extract(EN^{comp}, S_1, S_2)$ die Interface-Plätze von $S_1$ und $S_2$ sowie alle anderen Elemente aus $EN^{comp}$, die weder Teil von $S_1$ noch von $S_2$ sind.*

*Beweis.*
Sei im Folgenden $S_i, i \in \{1, 2\}$ ein APN mit Transitionsmenge $T^{S_i}$, Platzmenge $P^{S_i}$ und Flussrelation $F^{S_i}$. Sei $EN^{comp}$ das aus dem Kompositionsschritt resultierende RCTL-Netz mit Transitionsmenge $T^{comp}$, Platzmenge $P^{comp}$ und Flussrelation $F^{comp}$. Sei $C = Extract(EN^{comp}, S_1, S_2)$ ein RCTL-Netz mit Transitionsmenge $T^C$, Platzmenge $P^C$ und Flussrelation $F^C$.

Der Beweis erfolgt durch Widerspruch.

Dafür nehmen wir an, dass es Elemente in $EN^{comp}$ gibt, die weder in $S_1$ noch in $S_2$ noch in $C$ enthalten sind.

$\exists f = (p, t) \in F^{comp} \vee f = (t, p) \in F^{comp}$, sodass nach der Rückgabe von $C$ gilt:

$$\begin{aligned}((p \in P^{comp} \setminus (P^{S_1} \cup P^{S_2})) \wedge p \notin P^C) \vee \\ ((t \in T^{comp} \setminus (T^{S_1} \cup T^{S_2})) \wedge t \notin T^C) \vee \\ ((f \in F^{comp} \setminus (F^{S_1} \cup F^{S_2})) \wedge f \notin F^C)\end{aligned} \tag{6.2}$$

Durch die Definition der RCTL-Formeln und des Übersetzungsschrittes ist ein Auftreten von unverbundenen Plätzen oder Transitionen nicht möglich. Daher ist es ausreichend hier nur über Kanten zu argumentieren.

Um unser Annahme zu überprüfen, betrachten wir die folgenden Fälle für eine beliebige Kante $f = (p, t) \in F^{comp} \vee f = (t, p) \in F^{comp}$.

1. Sei $(p \in P^{comp} \setminus (P^{S_1} \cup P^{S_2})) \wedge p \notin P^C$. Dann können die folgenden Fälle eintreten.

   1.1. $t \in T^C$: An diesem Punkt würde der zweite Fall im Algorithmus (Zeile 14) angewendet und $p$ würde zu $P^C$ hinzugefügt werden. Dies ist ein Widerspruch zu unserer Annahme.

   1.2. $t \notin T^C$: In diesem Fall gibt es zwei weitere Möglichkeiten.

      1.2.1. $f \notin F^C$ : An diesem Punkt würde der vierte Fall im Algorithmus (Zeile 21) angewendet und $p$ würde zu $P^C$ hinzugefügt werden. Dies ist ein Widerspruch zu unserer Annahme.

      1.2.2. $f \in F^C$ : Dieser Fall kann nicht eintreten da die Definition von $F$ erfordert, dass falls $f = (p, t) \vee f = (t, p) \in F^C$ gilt, so muss auch $p \in P^C$ und $t \in T^C$ gelten. Dies ist ein Widerspruch zu unserer Annahme.

   Das bedeutet, immer wenn $p \in P^{comp} \setminus (P^{S_1} \cup P^{S_2})$ gilt, so ist $p \in P^C$. Dies ist ein Widerspruch zur Annahme 6.2.

2. Gelte $(t \in T^{comp} \setminus (T^{S_1} \cup T^{S_2})) \wedge t \notin T^C$.
   Dann könnten die folgenden Fälle eintreten.

   2.1. $p \in P^C$: An diesem Punkt würde der erste Fall im Algorithmus (Zeile 10) angewendet und $t$ würde zu $T^C$ hinzugefügt werden. Dies ist ein Widerspruch zu unserer Annahme.

   2.2. $p \notin P^C$: In diesem Fall gibt es zwei weitere Möglichkeiten.

      2.2.1. $f \notin F^C$ :An diesem Punkt würde der vierte Fall im Algorithmus (Zeile 21) angewendet und $t$ würde zu $T^C$ hinzugefügt werden. Dies ist ein Widerspruch zu unserer Annahme.

      2.2.2. $f \in F^C$ : Dieser Fall kann nicht eintreten da die Definition von $F$ erfordert, dass falls $f = (p, t) \vee f = (t, p) \in F^C$ gilt, so muss auch $p \in P^C$ und $t \in T^C$ gelten. Dies ist ein Widerspruch zu unserer Annahme.

   Das bedeutet, immer wenn $(t \in T^{comp} \setminus (T^{S_1} \cup T^{S_2}))$ gilt, so ist $t \in T^C$. Dies ist ein Widerspruch zur Annahme 6.2.

3. Gelte $(f \in F^{comp} \setminus (F^{S_1} \cup F^{S_2})) \wedge f \notin F^C$.
   Dann könnten die folgenden Fälle eintreten.

   3.1. $p \in P^C \wedge t \in T^C$: An diesem Punkt würde der dritte Fall im Algorithmus (Zeile 18) angewendet und $f$ würde zu $F^C$ hinzugefügt werden. Dies ist ein Widerspruch zu unserer Annahme.

   3.2. $p \in P^C \wedge t \notin T^C$: In diesem Fall gibt es zwei weitere Möglichkeiten.

      3.2.1. $t \in (T^{S_1} \cup T^{S_2})$ : In diesem Fall, müsste $p$ ein Interface-Platz sein, d.h. es müsste $p \in (P^{S_1} \cup P^{S_2})$ gelten. Damit wäre

$f \in (F^{S_1} \cup F^{S_2})$, was bedeutet, dass $f$ nicht Teil des Controllers sein darf.

3.2.2. $t \notin (T^{S_1} \cup T^{S_2})$ : An diesem Punkt würde der erste Fall im Algorithmus (Zeile 10) angewendet und $t$ und $f$ würden zu $T^C$ und $F^C$ hinzugefügt werden. Dies ist ein Widerspruch zu unserer Annahme.

3.3. $p \notin P^C \wedge t \in T^C$: In diesem Fall gibt es zwei weitere Möglichkeiten.

3.3.1. $p \in (P^{S_1} \cup P^{S_2})$ : Wenn $p$ und $t$ mit einander verbunden sind, muss $t \in (T^{S_1} \cup T^{S_2})$ und, daher auch $f \in (F^{S_1} \cup F^{S_2})$ gelten. Somit darf $f$ nicht Teil des Controllers sein.

3.3.2. $p \notin (P^{S_1} \cup P^{S_2})$ : An diesem Punkt würde der zweite Fall im Algorithmus (Zeile 14) angewendet und $f$ würde zu $F^C$ hinzugefügt werden. Dies ist ein Widerspruch zu unserer Annahme.

3.4. $p \notin P^C \wedge t \notin T^C$: An diesem Punkt würde der vierte Fall im Algorithmus (Zeile 21) angewendet und $p, t$ und $f$ würden zu $P^C, T^C$ und $F^C$ hinzugefügt werden. Dies ist ein Widerspruch zu unserer Annahme.

Das bedeutet, immer wenn $f \in F^{comp} \setminus (F^{S_1} \cup F^{S_2})$ gilt, so ist $f \in F^C$. Dies ist ein Widerspruch zur Annahme 6.2.

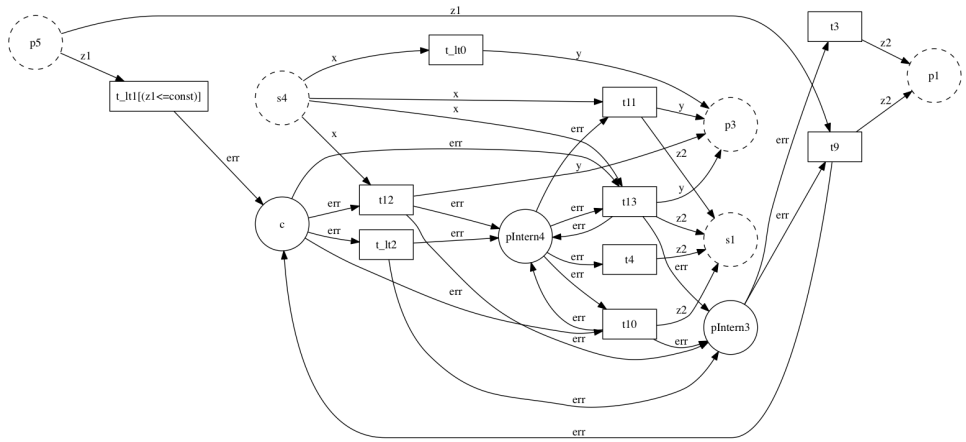
Da wir über eine beliebige Kante $f \in F^{comp}$ argumentiert haben, gilt der Widerspruch für alle $f$. Somit kann Annahme 6.2 nicht stimmen. Unser Algorithmus extrahiert demnach das Verhalten aus dem komponierten Netz, das in keinem der Service-Netze auftritt und somit den Controller modelliert. □

### 6.2.5 Zurückführung eines RCTL-Netzes auf ein APN

Der Controller liegt nach dem Extraktionsschritt als RCTL-Netz $C$ vor. Um ihn jedoch adäquat mit den Service-Netzen komponieren zu können, überführen wir im letzten Schritt unseres Ansatzes $C$ noch in ein APN. Den Algorithmus dafür haben wir bereits in Kapitel 5.3 vorgestellt. Wir gehen daher an dieser Stelle nicht in Detail, sondern geben nur eine Intuition durch die Betrachtung unseres laufenden Beispiels.

**Beispiel 6.5. (Laufendes Beispiel)**

Auf das aus dem Extraktionsschritt resultierte RCTL-Netz (siehe Beispiel 6.4 auf Seite 135) wird der Zurückführungsschritt angewendet. Dies führt zu folgendem APN, das das Verhalten des gewünschten Controllers repräsentiert.




Dieser Controller ist hinsichtlich der spezifizierten Eigenschaften korrekt und kann über die Interface-Plätze (in der Abbildung gestrichelt dargestellt) mit den Services komponiert werden.

## 6.3 Korrektheit

In diesem Kapitel haben wir unseren Ansatz zur Controller-Synthese für Services mit Daten vorgestellt. Dieser Ansatz ist so definiert, dass das folgende Theorem gilt.

**Theorem 6.3.**
*Seien $S_1, S_2$ zwei APNs und sei $\Phi = \{\varphi_1, \ldots, \varphi_n\}, n \in \mathbb{N}$ eine Menge von Anforderungen als RCTL-Formeln. Wenn unser Synthese-Prozess $Synth(S_1, S_2, \Phi)$ in einem APN $C$ resultiert, dann gilt:*

$$S_1 \otimes C \otimes S_2 \models \varphi_1 \wedge \cdots \wedge \varphi_n$$

*Beweis.*
Angenommen es gilt $\Phi = \{\varphi_1, \ldots, \varphi_n\}, n \in \mathbb{N}$ und $Synth(S_1, S_2, \Phi) = C$.

Unserem Ansatz folgend gilt:

$$Synth(S_1, S_2, \Phi) = C =$$
$$RCTLN2APN(Extract(\otimes(Trans(APN2RCTL(\{S_1, S_2\}) \cup \Phi))))$$
$$\text{mit } APN2RCTL(\{S_1, S_2\}) = APN2RCTL(S_1) \cup APN2RCTL(S_2)$$
$$\text{und } Trans(\{\phi_1, \ldots, \phi_k\}) := \{Trans(\phi_1), \ldots, Trans(\phi_k)\}$$

1. Aus den Theoremen 5.4 (Seite 89), 5.5 (Seite 90) und 5.6 (Seite 91) ergibt sich, dass das Verhalten der beiden Services an den Interfaces vollständig spezifiziert ist und dass:

   1.1. für alle $q \in APN2RCTL(S_1)$ gilt $S_1 \models q$ und

   1.2. für alle $r \in APN2RCTL(S_2)$ gilt $S_2 \models r$.

2. Nach Theorem 5.22 (Seite 106) gilt für alle RCTL-Formeln $\varphi$: $Trans(\varphi) \models \varphi$. Somit gilt :

   2.1. für alle $\varphi \in \Phi$ gilt $Trans(\varphi) \models \varphi$,

   2.2. für alle $q \in APN2RCTL(S_1)$ gilt $Trans(q) \models q$ und

   2.3. für alle $r \in APN2RCTL(S_2)$ gilt $Trans(r) \models r$.

3. Nach Theorem 6.1 (Seite 131) gilt für RCTL-Netze $N_i$ und RCTL-Formeln $\varphi_i$ mit $N_i \models \varphi_i, i \in [1,n]$:
   $\bigotimes\{N_i | i \in [1,n]\} \models \bigwedge\{\varphi_i | i \in [1,n]\}$.
   Somit gilt für $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ gilt $APN2RCTL(S_1) = \{q_1, \ldots, q_k\}$ und für $APN2RCTL(S_2) = \{r_1, \ldots, r_l\}$:
   $\otimes(Trans(APN2RCTL(S_1) \cup APN2RCTL(S_2) \cup \Phi)) \models$
   $q_1 \wedge \cdots \wedge q_k \wedge r_1 \wedge \cdots \wedge r_l \wedge \varphi_1 \wedge \cdots \wedge \varphi_n$.

4. Aus Theorem 6.2 ergibt sich für
   $C' = Extract(\otimes(Trans(APN2RCTL(S_1) \cup APN2RCTL(S_2) \cup \Phi)), S_1, S_2)$,
   dass sämtliches für den Controller wichtiges und nicht in $S_1$ und $S_2$ vorhandenes Verhalten durch $C'$ abgedeckt ist.

5. Abschließend folgt aus Theorem 5.23 (Seite 120), dass für alle RCTL-Formeln $\varphi$, RCTL-Netze $C'$ und für $C = RCTLN2APN(C')$ gilt:
   $(C' \models \varphi) \implies (C \models \varphi)$.

Da nach der Argumentation in 1.) das gesamte Interface-Verhalten der Services in dem Synthese-Prozess beachtet wird und das in den Services vorkommende Verhalten nach 4.) aus dem Controller entfernt wird, kann durch die Zurückführung eines RCTL-Netzes in ein APN das resultierende Netz mit den Services komponiert werden. Dabei gehen nach 5.) keine Informationen verloren und es gilt für
$C = RCTLN2APN(Extract(\otimes(Trans(APN2RCTL(\{S_1, S_2\}) \cup \Phi))))$:

$$S_1 \otimes C \otimes S_2 \models \varphi_1 \wedge \cdots \wedge \varphi_n$$

□

Somit arbeitet der synthetisierte Controller hinsichtlich der Verhaltenseigenschaften für jede Interpretation korrekt, die die algebraische Spezifikation erfüllt.

**Bemerkung 6.4.**
*Wir haben in Theorem 6.3 die Klammerung weggelassen, da für unseren synthetisierten Controller das folgende Theorem gilt.*

**Theorem 6.5.**
*Seien $S_1, S_2$ zwei APNs und sei $\Phi = \{\varphi_1, \ldots, \varphi_n\}, n \in \mathbb{N}$ eine Menge von Anforderungen als RCTL-Formeln. Sei $C = Synth(S_1, S_2, \Phi)$. Dann gilt:*

$$(S_1 \otimes C) \otimes S_2 = S_1 \otimes (C \otimes S_2)$$

*Beweis.* Seien $S_1$ und $S_2$ zwei APNs mit den Interface-Eingangsplätzen $P_i^{S_1}$ und $P_i^{S_2}$, sowie den Interface-Ausgangsplätzen $P_o^{S_1}$ und $P_o^{S_2}$. Sei $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ eine Menge von Anforderungen als RCTL-Formeln und $C = Synth(S_1, S_2, \Phi)$ der aus unserem Synthese Prozess resultierende Controller mit den Interface-Eingangsplätzen $P_i^C$ und den Interface-Ausgangsplätzen $P_o^C$. Im folgenden nehmen wir an, dass $S_1, S_2$ und $C$ paarweise komponierbar sind (vgl. Definition 2.12 auf Seite 24)).

Um zu zeigen, dass $(S_1 \otimes C) \otimes S_2 = S_1 \otimes (C \otimes S_2)$ gilt, zeigen wir, dass $P_i^{(S_1 \otimes C) \otimes S_2} = P_i^{S_1 \otimes (C \otimes S_2)}$ und $P_o^{(S_1 \otimes C) \otimes S_2} = P_o^{S_1 \otimes (C \otimes S_2)}$ gilt. Dies genügt, da die Komposition der anderen Netzelemente bereits als assoziativ bekannt sind.

Durch Anwendung von Algorithmus 11 (Seite 134) gilt $P_i^C = P_o^{S_1} \cup P_o^{S_2}$ und $P_o^C = P_i^{S_1} \cup P_i^{S_2}$.

Sei $N = S_1 \otimes C$, dann gilt:

$$\begin{aligned} P_i^N &= P_i^{S_1 \otimes C} \\ &= (P_i^{S_1} \cup P_i^C) \setminus ((P_i^{S_1} \cap P_o^C) \cup (P_i^C \cap P_o^{S_1})) \text{ (nach Def. 2.12, S.24)} \\ &= (P_i^{S_1} \cup (P_o^{S_1} \cup P_o^{S_2})) \setminus ((P_i^{S_1} \cap (P_i^{S_1} \cup P_i^{S_2})) \cup ((P_o^{S_1} \cup P_o^{S_2}) \cap P_o^{S_1})) \\ &= (P_i^{S_1} \cup (P_o^{S_1} \cup P_o^{S_2})) \setminus (P_i^{S_1} \cup P_o^{S_1}) = P_o^{S_2} \end{aligned}$$

$$\begin{aligned} P_o^N = P_o^{S_1 \otimes C} &= (P_o^{S_1} \cup P_o^C) \setminus ((P_o^{S_1} \cap P_i^C) \cup (P_o^C \cap P_i^{S_1})) \\ &= (P_o^{S_1} \cup (P_i^{S_1} \cup P_i^{S_2})) \setminus ((P_o^{S_1} \cap (P_o^{S_1} \cup P_o^{S_2})) \cup ((P_i^{S_1} \cup P_i^{S_2}) \cap P_o^{S_1})) \\ &= (P_o^{S_1} \cup (P_i^{S_1} \cup P_i^{S_2})) \setminus (P_o^{S_1} \cup P_i^{S_1}) = P_i^{S_2} \end{aligned}$$

Damit gilt :

$$\begin{aligned} P_i^{N \otimes S_2} = P_i^{(S_1 \otimes C) \otimes S_2} &= (P_i^N \cup P_i^{S_2}) \setminus ((P_i^N \cap P_o^{S_2}) \cup (P_i^{S_2} \cap P_o^N)) \\ &= (P_o^{S_2} \cup P_i^{S_2}) \setminus ((P_o^{S_2} \cap P_o^{S_2}) \cup (P_i^{S_2} \cap P_i^{S_2})) = \emptyset \end{aligned}$$

und

$$P_o^{N\otimes S_2} = P_o^{(S_1\otimes C)\otimes S_2} = (P_o^N \cup P_o^{S_2}) \setminus ((P_o^N \cap P_i^{S_2}) \cup (P_o^{S_2} \cap P_i^N)) \\ = (P_i^{S_2} \cup P_o^{S_2}) \setminus ((P_i^{S_2} \cap P_i^{S_2}) \cup (P_o^{S_2} \cap P_o^{S_2})) = \emptyset$$

Sei $N = C \otimes S_2$, dann gilt:

$$P_i^N = P_i^{C\otimes S_2} = (P_i^C \cup P_i^{S_2}) \setminus ((P_i^C \cap P_o^{S_2}) \cup (P_i^{S_2} \cap P_o^C)) \\ = ((P_o^{S_1} \cup P_o^{S_2}) \cup P_i^{S_2}) \setminus (((P_o^{S_1} \cup P_o^{S_2}) \cap P_o^{S_2}) \cup (P_i^{S_2} \cap (P_i^{S_1} \cup P_i^{S_2}))) \\ = ((P_o^{S_1} \cup P_o^{S_2}) \cup P_i^{S_2}) \setminus (P_o^{S_2} \cup P_i^{S_2}) = P_o^{S_1}$$

$$P_o^N = P_o^{C\otimes S_2} = (P_o^C \cup P_o^{S_2}) \setminus ((P_o^C \cap P_i^{S_2}) \cup (P_o^{S_2} \cap P_i^C)) \\ = ((P_i^{S_1} \cup P_i^{S_2}) \cup P_o^{S_2}) \setminus (((P_i^{S_1} \cup P_i^{S_2}) \cap P_i^{S_2}) \cup (P_o^{S_2} \cap (P_o^{S_1} \cup P_o^{S_2}))) \\ = ((P_i^{S_1} \cup P_i^{S_2}) \cup P_o^{S_2}) \setminus (P_i^{S_2} \cup P_o^{S_2}) = P_i^{S_1}$$

Damit gilt :

$$P_i^{S_1\otimes N} = P_i^{S_1\otimes(C\otimes S_2)} = (P_i^{S_1} \cup P_i^N) \setminus ((P_i^{S_1} \cap P_o^N) \cup (P_i^N \cap P_o^{S_1})) \\ = (P_i^{S_1} \cup P_o^{S_1}) \setminus ((P_i^{S_1} \cap P_i^{S_1}) \cup (P_o^{S_1} \cap P_o^{S_1})) = \emptyset$$

$$P_o^{S_1\otimes N} = P_o^{S_1\otimes(C\otimes S_2)} = (P_o^{S_1} \cup P_o^N) \setminus ((P_o^{S_1} \cap P_i^N) \cup (P_o^N \cap P_i^{S_1})) \\ = (P_o^{S_1} \cup P_i^{S_1}) \setminus ((P_o^{S_1} \cap P_o^{S_1}) \cup (P_i^{S_1} \cap P_i^{S_1})) = \emptyset$$

Offensichtlich gilt daher:

$$P_i^{(S_1\otimes C)\otimes S_2} = P_i^{S_1\otimes(C\otimes S_2)} \text{ und } P_o^{(S_1\otimes C)\otimes S_2} = P_o^{S_1\otimes(C\otimes S_2)}$$

Damit gilt:

$$(S_1 \otimes C) \otimes S_2 = S_1 \otimes (C \otimes S_2)$$

□

# 7 Implementierung und Fallstudien

In den vorherigen Abschnitten haben wir die theoretischen Aspekte unseres Controller-Synthese Prozesses vorgestellt. In diesem Kapitel betrachten wir nun dessen praktische Anwendung. Dies umfasst eine Präsentation der Hauptaspekte unserer Implementierung (Abschnitt 7.1) und die Evaluierung unseres Ansatzes (Abschnitte 7.2, 7.3 und 7.4).

## 7.1 Implementierung

In diesem Abschnitt stellen wir die Grundstruktur unserer Implementierung vor (siehe Abbildung 7.1).

Als Eingabe werden zwei Arten von Textdateien verlangt. Zum einen ist dies jeweils eine Textdatei, die eine mengentheoretische Beschreibung eines Petrinetzes enthält. Zum anderen handelt es sich um eine Textdatei, die aus allen sicherzustellenden Verhaltenseigenschaften als Formeln besteht.

Diese Eingaben werden mittels unseres Petrinetz-Parsers und unseres Formel-Parsers, der auf [Kol14] aufbaut, zunächst in eine Zwischendarstellung überführt. Auf die in der Zwischenrepräsentation gespeicherten Petrinetze wird unser Algorithmus zur Extraktion beobachtbaren Verhaltens (Abschnitt 5.1) angewendet. Dies resultiert in der Zwischenrepräsentation einer RCTL-Formel. Diese Zwischenrepräsentation ist komponentenbasiert. Das bedeutet für jeden Teil der Formel existiert ein eigenes Objekt. Die Gesamtformel ist eine Kombination aus solchen Objekten. Um eine Intuition dafür zu bekommen, betrachten wir das folgende Beispiel (Beispiel 7.1).

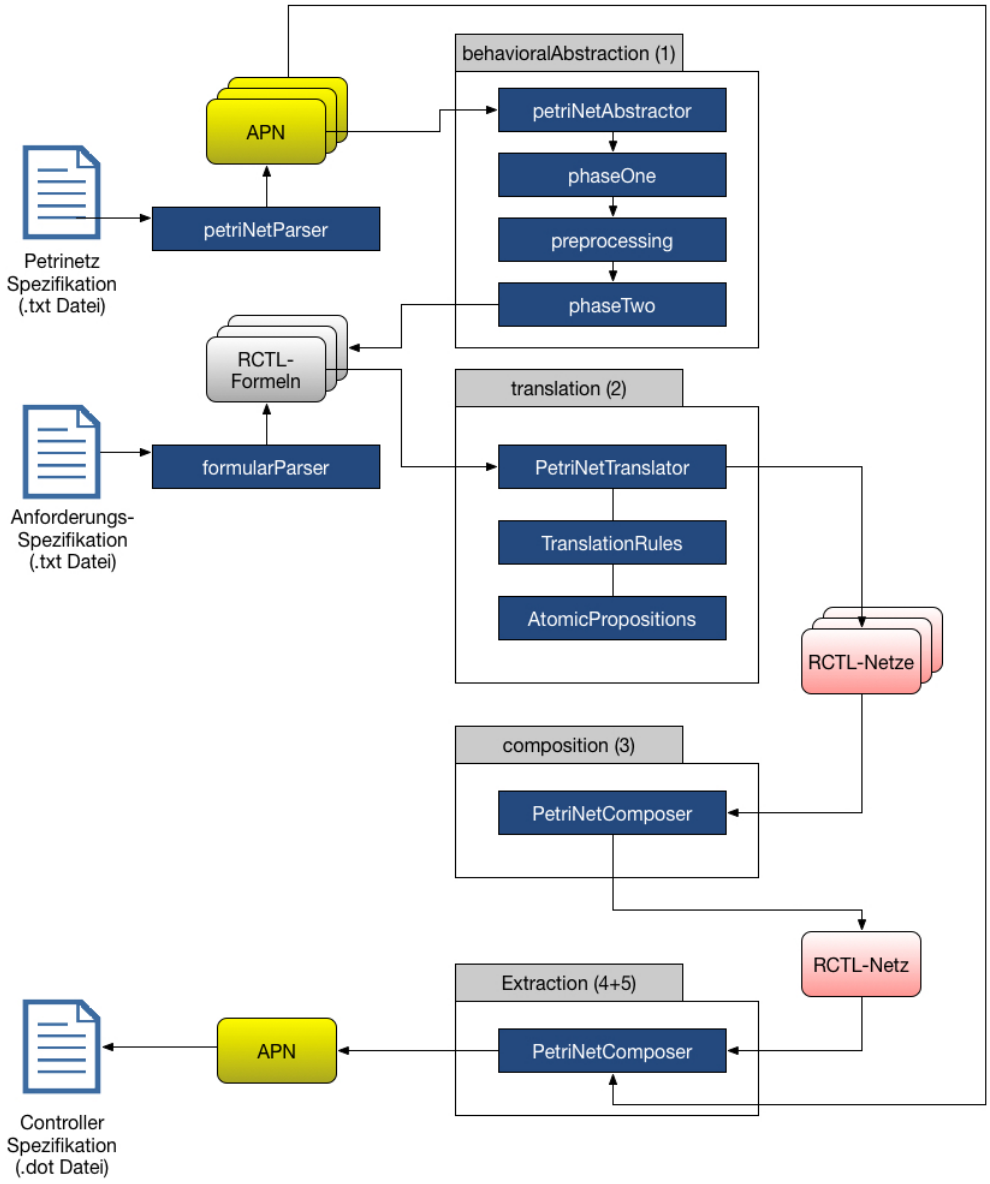



**Abbildung 7.1:** Struktur der Implementierung

**Beispiel 7.1.** (Zwischendarstellung einer RCTL-Formel)

Sei $\varphi = p_1.x \rightarrow AX(p_2.y \wedge p_3.z)$ eine RCTL-Formel. Dann hat ihre Zwischendarstellung die folgende Struktur.

$IR(\varphi) :=$
**Implication(AtomProp($p_1.x$), CTLFormula( $AX$,**
**Conjunction(AtomProp($p_2.y$), AtomProp($p_3.z$))))**

Die RCTL-Formeln in der Zwischendarstellung werden als Eingabe für den Übersetzungsschritt aus dem Paket *translation* genutzt. Dessen Implementierung folgt den Regeln, die in Abschnitt 6.2.2 vorgestellt wurden. Dies resultiert in einer Menge von RCTL-Netzen, die als Eingabe für den Kompositionsschritt genutzt wird. Dieser ist in dem Paket *PetriNetComposer* realisiert. Das Paket beinhaltet unter anderem eine Prioritäten-Warteschlange für die zu komponierenden RCTL-Netze. In dieser Warteschlange werden die RCTL-Netz in aufsteigender Reihenfolge hinsichtlich der Anzahl ihrer Plätze sortiert. Aus dieser wird jeweils das kleinste Netz entfernt und mit dem aktuell komponierten Netz verschmolzen. Sollten zwei oder mehr Netze die gleiche (kleinste) Anzahl an Plätzen haben, wird das erste Netz aus der Warteschlange entnommen. Dadurch ist der Kompositionsschritt nicht-deterministisch sondern hängt von der Reihenfolge der hinzugefügten Netze (gleicher Größe) ab.

Das aus dem *PetriNetComposer* resultierende *RCTL-Netz* wird an den *PetriNetzExtractor* als Eingabe weiter gegeben. Dieser extrahiert unter Hinzunahme der Informationen aus den originalen Service-Netzen die Elemente des Controllers und führt das RCTL-Netz zurück auf ein *APN*. Die Implementierung folgt dabei den Beschreibungen, die in Kapitel 6.2.4 und Kapitel 6.2.5 gegeben wurden.

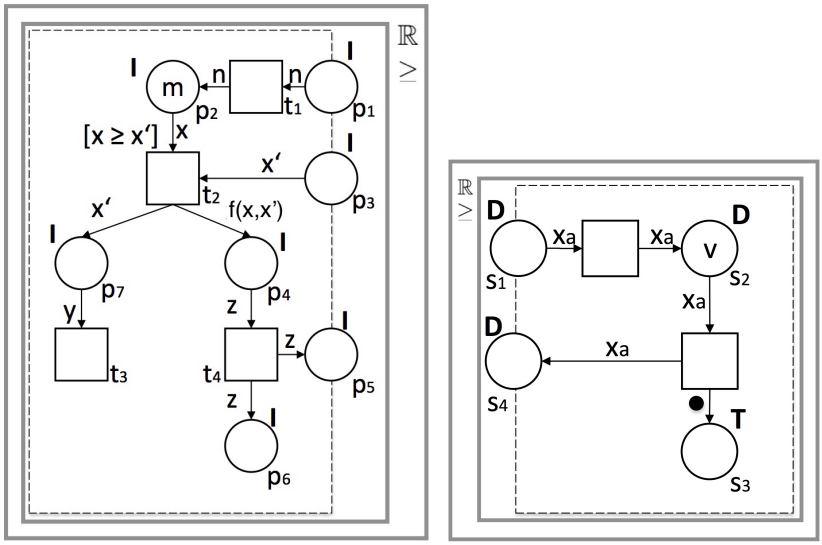
Zur Repräsentation des synthetisierten Controllers wird dieser in eine Textdatei geschrieben, die mittels *dot* (Graphviz) [GKN15] interpretiert und visualisiert werden kann.

Wir haben unsere Implementierung anhand von drei Fallstudien evaluiert, um Aussagen über die praktische Anwendbarkeit unseres Ansatzes treffen zu können. In allen Fällen haben wir unsere Tests auf einem MacBook Air mit einem 1,6 Ghz Intel® Core ™ i5-Prozessor und 4GB RAM unter Mac OS X El Capitan durchgeführt. Wir haben zur Bestimmung der Laufzeiten jedes Schrittes unseres Prozesses die Java-eigene Funktion 'System.currentTimeMillis();' genutzt. Um den Fehler zu minimieren, der aus äußeren Umständen, wie z.B. CPU Auslastung resultiert, haben wir jeden Test 10 mal durchgeführt und die Ergebnisse gemittelt. Das Programm sowie Nutzungshinweise sind unter [BT17] zu finden.

In den folgenden Abschnitten stellen wir die drei Fallstudien, die Berechnungszeiten und den jeweils konstruierten Controller vor. Als erste Fallstudie nutzen wir unser laufendes Beispiel, das wir bereits in Abschnitt 2.4 eingeführt haben. Zusätzlich nutzen wir zur Evaluierung unsere Ansatzes zwei weitere Fallstudien aus dem medizinischen Bereich. Grob gesagt sind dies die künstliche Bauchspeicheldrüse und ein Apnoe-Erkennungssystem. Auch wenn wir die Anwendbarkeit unseres Controller-Synthese Prozesses nur an der Sicherstellung einer zuverlässigen Interoperabilität medizinischer Geräte demonstrieren, sind andere Einsatzbereiche durchaus auch denkbar. Dies umfasst alle Domänen, in denen Komponenten mittels eines Controllers kombiniert werden sollen.

## 7.2 Fallstudie 1: Generische Pumpe (Laufendes Beispiel)

Das laufende Beispiel, das uns als erste Fallstudie dient, haben wir bereits in Abschnitt 2.4 (Abbildung 2.3 auf Seite 49) eingeführt. Daher fassen wir hier nur die Hauptaspekte noch einmal kurz zusammen und zeigen die APNs an dieser Stelle erneut.




**(a)** Pumpe **(b)** Schalter

**Abbildung 7.2:** Laufendes Beispiel - Verhalten als APN

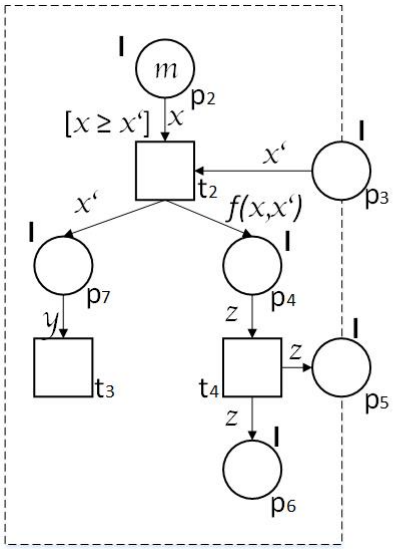
Bei dem betrachteten System handelt es sich um eine Pumpe und einen Schalter. Die Pumpe stellt zwei Eingangskanäle zur Verfügung. Einer davon dient dazu, Informationen über den aktuellen Füllstand der Pumpe zu erhalten. Über den anderen Eingangskanal kann der Pumpe mitgeteilt werden, wie viel des Medikamentes injiziert werden soll. Zusätzlich dazu bietet die Pumpe einen Ausgangskanal über den der aktualisierte Füllstand an die Umgebung gesendet wird. Der Schalter sendet nur ein konstanten Wert über seinen Ausgangskanal und kann durch eine Nachricht über den Eingangskanal zurückgesetzt werden.

**Ziel** Der gewünschte Controller sollte in der Lage sein, den Schalter zurückzusetzen, Nachrichten von dem Schalter an die Pumpe weiterzuleiten und die Pumpe über ihren aktuellen Füllstand zu informieren.

### Eingaben

Um die praktische Anwendbarkeit unseres Prozesses zu evaluieren, haben wir verschiedene Situationen für unser laufendes Beispiel analysiert. Zum einen reduzieren wir die Komplexität der betrachteten Services, während die Anzahl

der Eigenschaften konstant bleibt. Dazu nutzen wir das in Abschnitt 5.1 vorgestellte, reduzierte Service-Netz der Pumpe, das wir an dieser Stelle noch einmal wiederholen.




$SP_{\text{Pump}} = (\Sigma_{\text{Pump}}, E_{\text{Pump}})$

$\Sigma_{\text{Pump}}=$

| | |
|---|---|
| **sorts:** | $I$ |
| **vars:** | $x, x', z, y \in X_I$ |
| **operations:** | $m : \rightarrow I$ |
| | $f : I \times I \rightarrow I$ |

$E_{\text{Pump}} = z = f(x,x')$

$y = x'$

$x \geq x'$

**Abbildung 7.3:** Auszug aus dem laufenden Beispiel

Zum anderen nutzen wir die originalen Services und verdoppeln die Anzahl der Eigenschaften. Für alle Szenarien dokumentieren wir den Zeitverbrauch für alle Schritte unseres Prozesses. Dadurch lassen sich die zeitintensivsten Teilprozesse identifizieren.

Zur Evaluation haben wir die folgenden Eigenschaften genutzt.

Zum einen betrachten wir die Eigenschaften, die wir in Abschnitt 2.4 fordern und die wie folgt dargestellt werden.

1. $\varphi_1 = s_4.x \rightarrow AF(p_3.x')$
2. $\varphi_2 = p_5.z \wedge z \leq Const_{limit} \rightarrow AX(C_{intern}.err)$
3. $\varphi_3 = (C_{intern}.err) \rightarrow AF(AX(p_1.z_{new}) \wedge AF(s_1.z_{new}))$

Zusätzlich haben wir die folgenden Eigenschaften zur Auswertung genutzt.

4. $\varphi_4 = p_5.z \wedge z \geq Const_{limit} \rightarrow AX(s_1.z)$
   Das bedeutet, wenn sich genügend Medizin im Reservoir der Pumpe befindet, soll der Schalter direkt zurückgesetzt werden.
5. $p_1.n \wedge p_3.x2 \wedge n \geq x2 \rightarrow AF(s_4.x_3)$
   Das bedeutet, wenn beide Interface-Eingangskanäle der Pumpe mit $n$ und $x2$ markiert sind und $n \geq x2$ gilt, so erreicht der Schalter einen Zustand, in dem er eine Nachricht über seinen Interface Ausgangskanal sendet.
6. $s_1.x_3 \rightarrow AF(s_4.x_3 \wedge p_3.x_4)$
   Dies bedeutet, wann immer der Schalter ein Signal über den Eingangskanal erhält, wird er irgendwann diesen Wert über den Ausgangsport

senden und die Pumpe eine Information erhalten, wieviel des Medikamentes injiziert werden soll.

### Resultat

Um die Skalierbarkeit unseres Prozesses auswerten zu können, analysieren wir die folgenden Kombinationen.

**A** originale Service-Netze (Abb. 7.2 ) und Eigenschaften 1.-3.

**B** originales Netz des Schalters (Abb. 7.2b), reduziertes Netz der Pumpe (Abb. 7.4 ) und (leicht angepasste) Eigenschaften 1.-3.

**C** originale Service-Netze (Abb. 7.2 ) und Eigenschaften 1.-6.

Für diese Kombinationen ergaben sich die folgenden (gemittelten) Messungen.

| | Schritt 1 | Schritt 2 | Schritt 3 | Schritt 4 | Schritt 5 | Gesamt |
|---|---|---|---|---|---|---|
| **A** | 35 ms | 44 ms | 15 ms | 9 ms | 12 ms | 229 ms |
| **B** | 30 ms | 51 ms | 13 ms | 9 ms | 25 ms | 236 ms |
| **C** | 34 ms | 78 ms | 20 ms | 13 ms | 84 ms | 348 ms |

**Tabelle 7.1:** Zeitverbrauch des Laufenden Beispiels

Der Synthese-Prozess für die Kombination **A** resultiert in folgendem Controller. Diesen haben wir bereits in Abschnitt 6.2.5 dargestellt. Der Übersichtlichkeit halber verzichten wir an dieser Stelle auf die Visualisierung der Controller für die Kombinationen **B** und **C**.

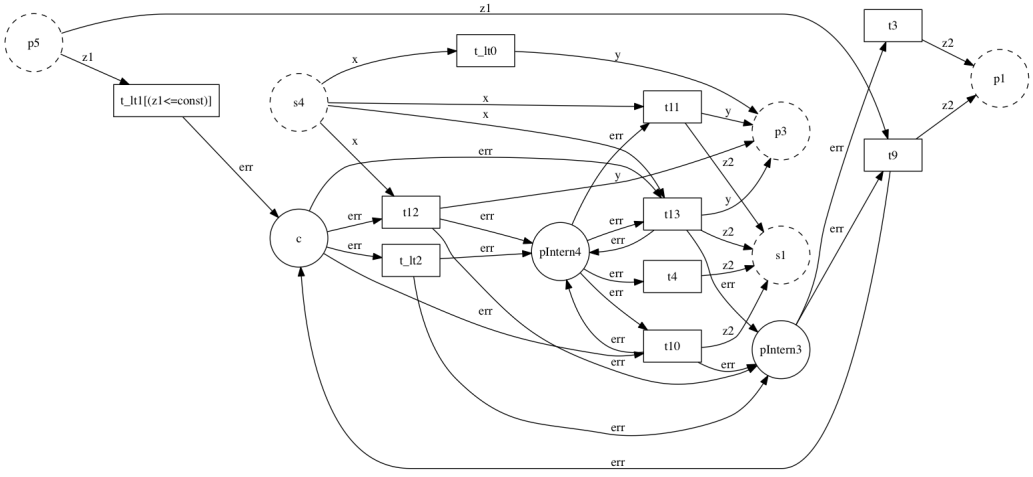



**Abbildung 7.4:** Synthetisierter Controller für Fallstudie 1 (**A**)

### Diskussion

Wie aus Tabelle 7.1 hervorgeht, hängt der Berechnungsaufwand dieser Fallstudie eher von der Anzahl der Anforderungen als von der Komplexität der Service-Netze ab. Das begründet sich in der Tatsache, dass die Service-Netze keinen Einfluss auf den Zurückführungsschritt haben, der den zeitintensivsten Schritt unseres Prozesses darstellt. Denn bereits während der Extraktion (Schritt 4 unseres Synthese Prozesses) werden alle Elemente der Service-Netze aus dem Controller-RCTL-Netz entfernt. Somit haben sie bei der Zurückführung des RCTL-Netzes auf ein APN keinen Einfluss mehr. Im Gegensatz dazu werden bei der Komposition umso mehr Transitionen erzeugt, je mehr Eigenschaften sicherzustellen sind. Basierend darauf wird annähernd die Potenzmenge der Transitionen erzeugt und somit wächst die Komplexität des Controller-APNs exponentiell. Für alle betrachteten Kombinationen konnte je ein Controller in akzeptabler Zeit erzeugt werden.

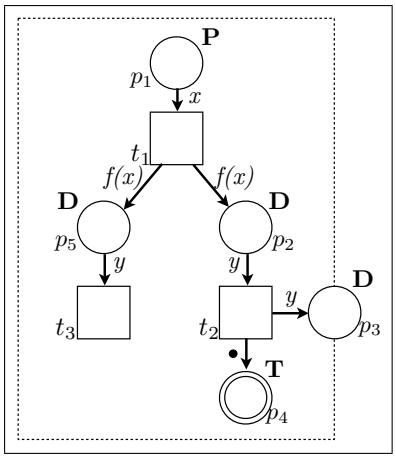
## 7.3 Fallstudie 2: Künstliche Bauchspeicheldrüse



Heutzutage wächst die Zahl an Menschen mit chronischen Krankheiten weltweit. Ein Beispiel dafür ist Typ-1-Diabetes [EDH11]. Dabei handelt es sich um eine unheilbare Erkrankung, die hauptsächlich durch das Zusammenspiel von Gendefekten, autoimmun Erkrankungen und Virusinfektionen hervorgerufen wird. Betroffene Menschen können Insulin nicht selbst produzieren, sondern müssen es sich ihr Leben lang injizieren. Das Risiko einer falschen Medikation ist besonders für Kinder und behinderte Menschen sehr hoch, was schnell sehr kritisch für die Gesundheit werden kann. Da dieses Problem bereits bekannt ist, haben einige Unternehmen, wie z.B. MiniMed® ein neues, geschlossenes System entwickelt, das *künstliche Bauchspeicheldrüse* genannt wird. Dieses System erhöht die Lebensqualität der Betroffenen durch die Übernahme der Insulin-Dosierung. Es besteht aus einem Glukose-Sensor und einer Insulin-Pumpe. Für das kontinuierliche Messen können verschiedene Arten von Sensoren genutzt werden. In der Charité und somit in unserem Fallbeispiel betrachten wir elektro-chemische Sensoren, deren Funktionalität [Wan08] im Folgenden erklärt wird.

**Glukose-Sensor** Abbildung 7.5 zeigt ein algebraisches Petrinetz, das das Verhalten eines Glukose-Sensors modelliert. Die enzymatische Komponente des Sensors reagiert mit dem Blutzucker. Abhängig von der Blutzucker-Konzentration wird dabei ein geringer Stromfluss ($x$) freigesetzt. Dieser Stromfluss wird dann mittels eines Analog-Digital-Konvertierers digitalisiert ($f(x)$). Das digitale Signal ($y$) wird zum einen intern gespeichert und zum anderen über das Interface an die Umgebung gesendet. Dies kann dann auf einem Display betrachtet werden. Der Nutzer muss darauf aufbauend entscheiden, ob eine In-

jektion notwendig ist oder nicht. Der Hauptnachteil eines solchen Sensors ist, dass er als Fremdkörper erkannt wird und daher alle 4-7 Tage ersetzt werden muss.




**(a)** Petrinetz

$$
\begin{aligned}
&\text{SPEC-GS} = (S_{GS}, OP_{GS}, X_{GS}, EQ_{GS}) \\
&\quad S_{GS} = \{P, D, T\} \\
&\quad OP_{GS} = \{\cdot, f\} \\
&\qquad\qquad \cdot :\rightarrow T \\
&\qquad\qquad f : P \rightarrow D \\
&\quad X_{GS} = \{X_P, X_D\} \\
&\qquad\quad X_P = \{x\} \\
&\qquad\quad X_D = \{y\} \\
&\quad E_{GS} = \emptyset
\end{aligned}
$$

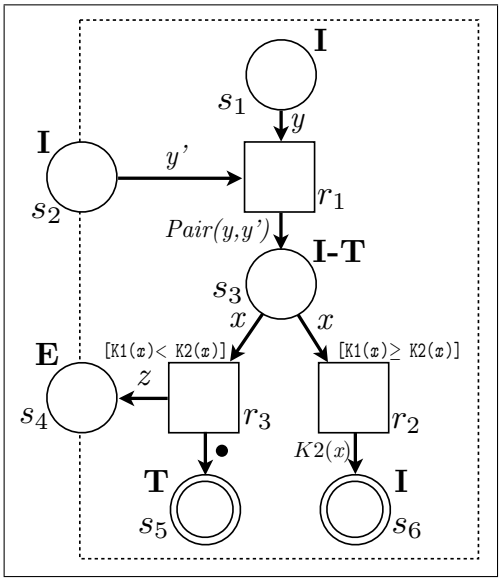
**(b)** Algebraische Spezifikation

**Abbildung 7.5:** Algebraisches Petrinetz eines Glukose-Sensors ©2014 ICST

**Insulin-Pumpe** Zusätzlich zum Glukose-Sensor haben wir basierend auf textuellen Beschreibungen [Bor15, Inc] das Verhalten einer Insulin-Pumpe modelliert. Dieses ist in Abbildung 7.6 dargestellt. Im Allgemeinen besteht eine Insulin-Pumpe aus einem Reservoir, einem Katheter, einem Füllstandsmesser und der Pumpe selbst.

Das Reservoir kann 150-300 Einheiten Insulin beinhalten. Die Menge an Insulin, die noch im Reservoir enthalten ist ($y$) wird durch den Füllstandsmesser bestimmt. Aktuell reagiert eine Insulin-Pumpe nicht auf die Signale des Glukose-Sensors. Sie kann nur periodisch eine feste Menge an Insulin über den Katheter injizieren. Die Insulin-Menge ($y'$) kann über den Eingangskanal kommuniziert werden. Innerhalb der Insulin-Pumpe wird $y$ mit der noch verfügbaren Menge ($y$) verglichen. Falls nicht mehr genug Insulin verfügbar sein sollte, so sendet die Insulin-Pumpe eine Warnung ($z$) an die Umgebung. Ansonsten injiziert die Pumpe die gewünschte Insulin-Menge ($K2(x)$) über den Katheter.

**Ziel** Durch die Entwicklung als geschlossenes System existiert eine signifikante Herstellerabhängigkeit. Somit ist es unser Ziel, unseren Ansatz zur automatisierten Synthese eines Controllers basierend auf gegebenen Eigenschaften zu nutzen, um die Interoperabilität eines Glukose-Sensors und einer Insulin-Pumpe sicherzustellen. Dadurch kann die Herstellerabhängigkeit reduziert werden.




**(a)** Petrinetz

$$SPEC - IP = (S_{IP}, OP_{IP}, X_{IP}, EQ_{IP})$$
$$S_{IP} = \{I, I-T, T, E\}$$
$$OP_{IP} = \{z, \cdot, Pair, K_1, K_2\}$$
$$z :\rightarrow E$$
$$\cdot :\rightarrow T$$
$$Pair : I \times I \rightarrow I - T$$
$$K_1, K_2 : I - T \rightarrow I$$
$$X_{IP} = \{X_I, X_{I-T}, X_T, X_E\}$$
$$X_I = \{y, y'\}$$
$$X_{I-T} = \{x\}$$
$$X_T = \emptyset$$
$$X_E = \emptyset$$
$$EQ_{IP} = \{e_1, e_2\}$$
$$e_1 \triangleq (K_1(Pair(y, y')) = y)$$
$$e_2 \triangleq (K_2(Pair(y, y')) = y')$$

**(b)** Algebraische Spezifikation

**Abbildung 7.6:** Algebraisches Petrinetz einer Insulin-Pumpe ©2014 ICST

### Eingaben

Für die *künstliche Bauchspeicheldrüse* (Services in Abbildungen 7.5a) und 7.6a) betrachten wir die folgenden Eigenschaften.

1. $\varphi_1 = p3.x \rightarrow AF(s2.y)$
   Diese Formel besagt, dass die Glukose-Konzentration, die durch den Glukose-Sensor gemessen wurde, als Eingabe für die Insulin-Pumpe genutzt wird.

2. $\varphi_2 = p3.x \wedge (x > gmax) \rightarrow AF(s2.g(x))$
   Diese Formel besagt, dass falls die Glukose-Konzentration $x$ zu hoch ist (die Schranke $gmax$ überschritten wird), die Insulin-Pumpe eine gewisse Menge an Insulin ($g(x)$) irgendwann injizieren muss.

3. $\varphi_3 = s4.z \rightarrow AX(c3.err)$
   Diese Eigenschaft stellt im Falle einer Fehlerbenachrichtigung sicher, dass der Controller diese durch bspw. das Senden eines Alarms weiterleitet.

### Resultat

Der Zeitverbrauch für diese Fallstudie ist in Tabelle 7.2 aufgeführt.

Der Berechnungsaufwand dieser Fallstudie ist vergleichbar zu dem Zeitverbrauch des laufenden Beispiels, da beide annähernd die gleiche Komplexität aufweisen. Unser Prozess resultiert in folgendem, automatisch synthetisierten Controller (Abb. 7.7).

| Schritt 1 | Schritt 2 | Schritt 3 | Schritt 4 | Schritt 5 | Gesamt |
|---|---|---|---|---|---|
| 45 ms | 49 ms | 21 ms | 10 ms | 17 ms | 259 ms |

**Tabelle 7.2:** Zeitverbrauch der Künstlichen Bauchspeicheldrüse

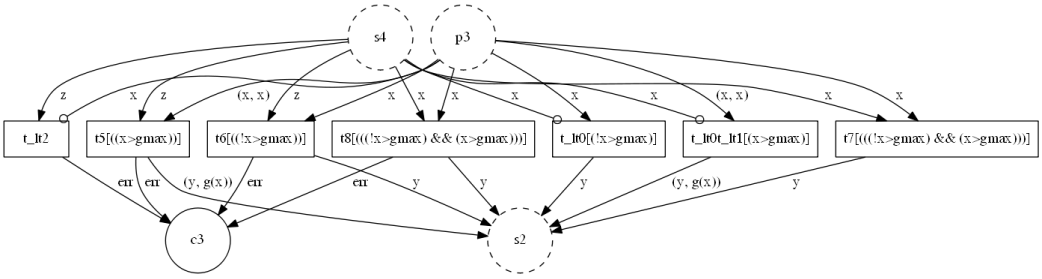



**Abbildung 7.7:** Synthetisierter Controller für die Künstliche Bauchspeicheldrüse (bestehend aus Glukose Sensor und Insulin-Pumpe)

## 7.4 Fallstudie 3: Apnoe-Erkennungssystem

*Copyright Notice: We have published the textual content of this section in [BTGGB14] ©2014 IEEE*

Plötzlicher Kindstod ist einer der häufigsten Todesursachen im Kleinkindalter. Das Risiko dafür ist bei Frühgeborenen und während des Schlafes am höchsten. Obwohl die tatsächlichen Ursachen bisher noch unbekannt sind, versterben die Kinder schlussendlich an einer Sauerstoffunterversorgung hervorgerufen durch Atemunregelmäßigkeiten, wie z.B. Apnoes [ASM08].

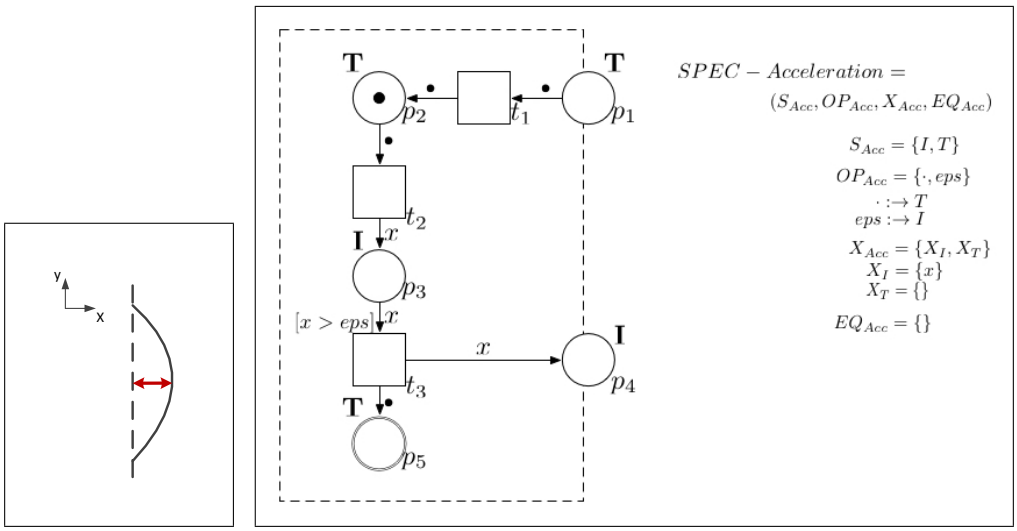
Um Apnoes und deren Ursachen zu erkennen, können verschiedene Sensoren genutzt werden. Jeder dieser Sensoren hat eine hohe Fehlalarmrate. Um das Risiko von Fehlalarmen zu reduzieren und zu behandeln, nutzen wir eine Kombination partiell redundanter Sensoren. Das von uns modellierte System besteht aus einem Bewegungssensor, der die Atemfrequenz bestimmt, einem Lichtsensor, um die Sauerstoffsättigung im Blut zu bestimmen und einem EKG-Sensor, zur Feststellung der Herzrate. Da der Pulsmesser aufgrund seines Aufbaus auch dazu in der Lage ist die Herzrate zu bestimmen, weisen die Sensoren eine gewisse Redundanz auf und reduzieren so die (bedingte) Wahrscheinlichkeit von Fehlalarmen [MBC12]. Ein Auszug aus diesem Überwachungssystem, bestehend aus einem Bewegungs- und einem EKG-Sensor wurde im Rahmen einer Masterarbeit als Andorid Applikation für Shimmer Sensoren [Shi] in Kooperation mit dem Master Studenten Marcus Pannwitz [Pan14] implementiert.

Unser System zeichnet dabei nicht nur Vital-Parameter von Kleinkindern auf, sondern reagiert mit Alarmen auf drastische Änderungen der Werte. Wir unterscheiden zwischen zwei verschiedenen Alarmarten: einem akustischen Alarm und einen Notruf. Der Ton soll dabei Pfleger oder Eltern warnen und die Kleinkinder zum weiteratmen animieren. Der Notruf hingegen soll automatisch pro-

fessionelle Hilfe verständigen. Im restlichen Teil dieses Abschnittes führen wir die wichtigsten Teile unseres Assistenzsystems ein.

**Bewegungs-Sensor** Der Bewegungs-Sensor, dessen Verhalten in Abbildung 7.8 dargestellt ist, wird zur Bestimmung der Atemfrequenz genutzt. Er wird durch einen Gurt an der Brust angebracht und misst den Ausschlag des Sensors, der durch die Bewegung des Brustkorbes beim Ein- und Ausatmen forciert wird. Durch diese Messungen erhalten wir Informationen über die Tiefe und Frequenz der Atmung. Sie werden periodisch hervorgerufen durch ein Signal der Umgebung (über den Kanal $p_1$). Die Transition $t_2$ zwingt den Sensor zu einer Messung des aktuellen Deltas, d.h. des Ausschlags, der auf der X-Achse sichtbar ist. Falls das Delta nicht zu klein ist, also größer als ein Epsilon (Guard an der Transition $t_3$), erkennt der Sensor dies als Ausschlag und sendet es als Signal an die Umgebung (durch den Interface-Kanal $p_4$).

Würde man nur den Bewegungs-Sensor als einzigen Anhaltspunkt für Atemunregelmäßigkeiten nutzen, würden viele Fehlalarme entstehen. Gründe dafür können zum Beispiel ein verrutschter Brustgurt oder ein sich stark bewegender Patient sein. Wegen diesen Ungenauigkeiten, nutzen wir zusätzlich einen zweiten Sensor, der die Zuverlässigkeit steigert: der Lichtsensor.




**(a)** Funktionalität **(b)** Algebraisches Petrinetz

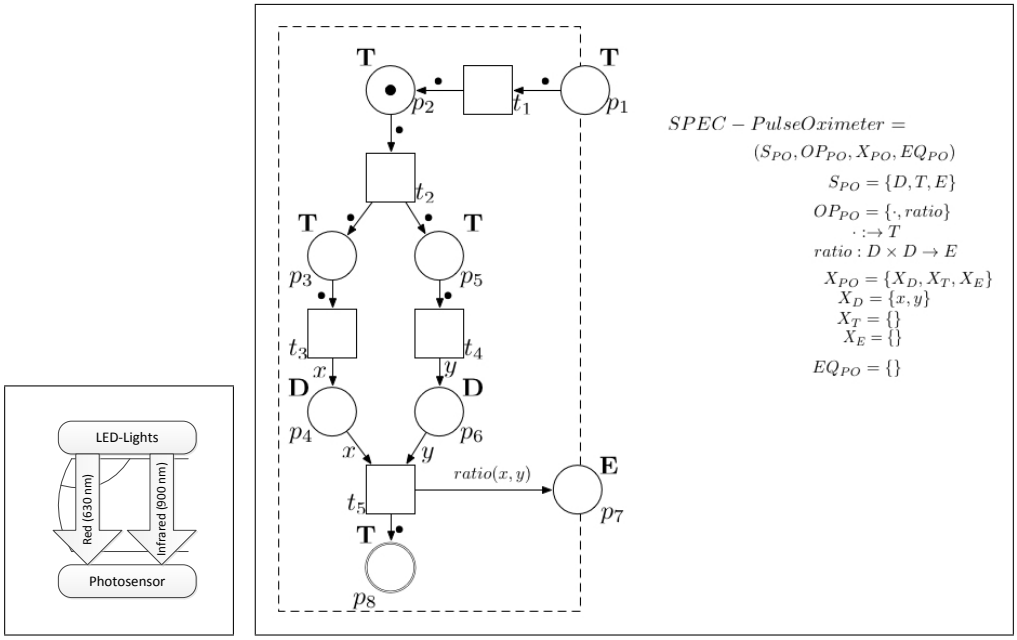
**Abbildung 7.8:** Bewegungssensor

**Lichtsensor** Eine Reduktion der Atemfrequenz führt zu einem Absinken der Sauerstoffkonzentration im Blut. Außerdem existiert ein Zusammenhang mit der Hämoglobin- und der Sauerstoffkonzentration des Blutes. Dies wird sich bei Lichtsensoren [SH87] zu Nutze gemacht, um die Sauerstoffsättigung ($SpO_2$) des Patienten zu bestimmen.

Dazu wird der Lichtsensor, dessen Funktionalität in Abbildung 7.9 dargestellt ist, an die Fingerspitze, das Ohrläppchen oder bei Neugeborenen an

den Fuß angebracht. Der eine Teil des Sensors besteht aus zwei Lichtquellen, deren ausgesendetes Licht teilweise vom Hämoglobin im Blut absorbiert wird. Dies wird durch den zweiten Teil des Lichtsensors wahrgenommen, indem die Intensität des nicht absorbierten Lichtes bestimmt wird. Die Hämoglobin-Konzentration wird aus den gemessenen Parametern unter Beachtung der folgenden Faktoren berechnet.

Zum einen existiert ein signifikanter Unterschied zwischen der Hämoglobin-Konzentration des arteriellen und des venösen Blutes. Zum anderen bewegt sich das Blut durch die Gefäße in Wellen. Um dennoch einen genauen Messwert zu erhalten, wird Licht in kurzen Abständen mit zwei verschiedenen Wellenlängen (900nm und 630nm) mit einer gewissen Intensität ausgestrahlt. Das Infrarotlicht (900 nm) wird dabei hauptsächlich durch das arterielle und das rote Licht durch das venöse Blut absorbiert. Die Hämoglobin-Konzentration und somit die Sauerstoffsättigung wird dann durch den Mittelwert der auftretenden Amplituden (in Abb. 7.9 $x$ und $y$), die Berechnung der Absorbtionsraten ($\text{ratio}(x, y)$) und die Überführung dieser Raten in $SpO_2$ durch Nutzung existierender Tabellen berechnet.



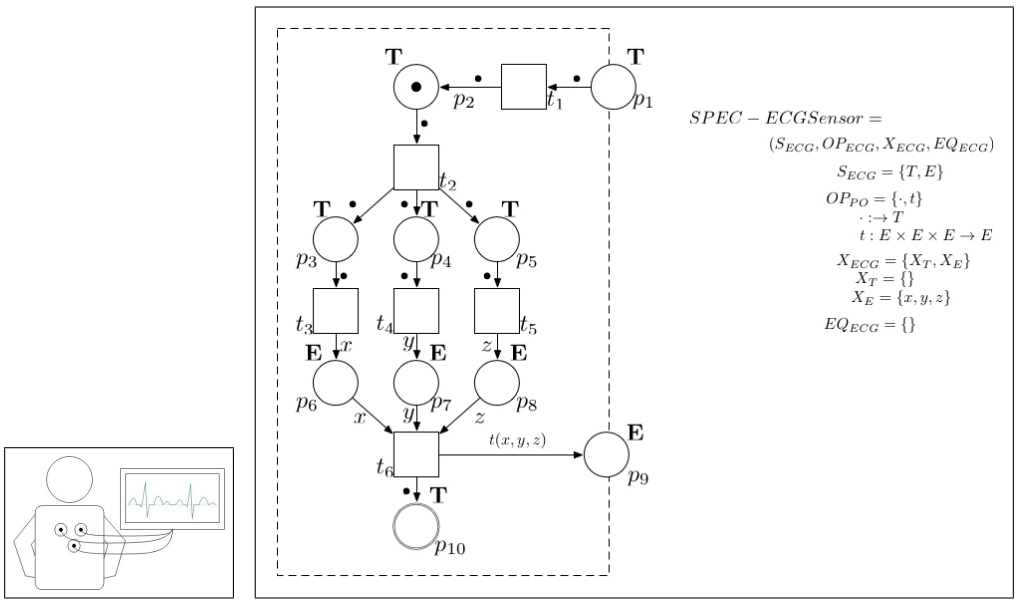

**(a)** Funktionalität **(b)** Algebraisches Petrinetz

**Abbildung 7.9:** Lichtsensor

**EKG-Sensor** Neben der Sauerstoffunterversorgung ist eine sinkende Herzrate ein weiterer Indikator für Apnoen. Um diesen Effekt zu erkennen ist eine zuverlässige Messung der Herzrate notwendig. Obwohl es an diesem Punkt schon möglich ist, die Herzrate mittels der Messergebnisse der beiden oben beschriebenen Sensoren zu berechnen, wäre dies bestenfalls eine gute Schätzung. Daher und um Fehlalarme zu vermeiden, nutzen wir in unserem System zusätzlich ein *Elektrokardiogramm (EKG)*. Dieses bestimmt die Herzrate durch

die Messung und Auswertung elektrischer Aktivität, die durch die Kontraktion des Herzmuskels entsteht.

Das Verhalten eines EKG-Sensors wird in Abbildung 7.10 dargestellt. Die Messungen erfolgen durch drei Elektroden, die an der Brust des Patienten angebracht werden. Jede dieser Elektroden misst ein Signal. In Abbildung 7.10b wird dies durch die Variablen $x, y, z$ dargestellt, denen Werte beim Schalten der Transitionen $t_3, t_4, t_5$ zugewiesen werden. Diese drei Signale werden dann durch die Operation $t$ kombiniert und das Ergebnis der Umgebung durch den Ausgangskanal $p_9$ mitgeteilt. Die Periodizität der Messungen wurde durch die Hinzunahme eines Eingangskanals $p_1$ modelliert, durch den der Sensor neu gestartet werden kann.




**(a)** Überblick **(b)** Algebraisches Petrinetz

**Abbildung 7.10:** EKG-Sensor

Durch die Nutzung der *drei* Sensoren haben wir ein Apnoe-Erkennungssystem modelliert, das eine geringe Anzahl an Fehlalarmen aufweist. Dies wird insbesondere dadurch erreicht, dass wir Probleme mit den Sensoren mit berücksichtigen.

Dazu haben wir ein farbkodiertes Alarm-System definiert, welches wir im Folgenden vorstellen.

**Farbkodiertes Alarm-System** In vielen Fällen ruft die Auswertung, der durch die Sensoren gemessenen Daten, Fehlalarme hervor. Dies kann beispielsweise durch defekte Sensoren oder Ausreißer geschehen. Inspiriert durch die Triage, einem Priorisierungskonzept bei Notfallszenarien, haben wir eine Einstufung unterschiedlicher Alarmebenen für unser Apnoe-Erkennungssystem entwickelt. Dabei werden drei Ebenen unterschieden.

**Ebene 1 - Grüner Alarm** Wenn die Auswertung der Daten für maximal einen Sensor ergibt, dass eine gegebene Grenze überschritten wird, ohne dabei die anderen Sensoren zu beeinflussen, interpretieren wir dies als einen Fehler im Sensor. Diese Annahme ist durch die Redundanz und die vorliegende Korrelation der Sensoren in unserem Beispiel legitim. Es ist jedoch auch möglich, dass eine Verzögerung zwischen Anomalien von verschiedenen Sensoren auftritt. Somit betrachten wir zusätzlich die Resultate der darauf folgenden Messungen. Falls die Werte des entsprechenden Sensors in der zweiten Messung sich wieder im Normalbereich befinden, nehmen wir an, dass es sich um einen Ausreißer handelte. Sollte die Grenze erneut überschritten werden und keine anderen Sensoren auf Fehler hinweisende Werte anzeigen, senden wir ein visuelles Signal. Dies soll Betreuer, Eltern des Patienten oder andere Verantwortliche dazu veranlassen, den auffälligen Sensor zu überprüfen.

**Ebene 2 - Orangener Alarm** Wenn zwei der Sensoren unerwartete Werte aufweisen, senden wir einen orangenen Alarm in Form eines akustischen Signals. Dadurch sollen Betreuer oder Eltern unmittelbar informiert werden. Zudem kann ein Kind dazu angeregt werden, die Atmung wieder aufzunehmen. Zusätzlich überprüfen wir welche der Sensoren involviert sind. Der Grund dafür ist, dass die Atmung und Herzfrequenz einer schlafenden Person natürlicher Weise geringer ist als die einer wachen Person. Sollte sich die Sauerstoffversorgung in den erwarteten Grenzen befinden, wird kein Alarm ausgelöst.

**Ebene 3 - Roter Alarm** Auf der dritten Ebene senden wir einen roten Alarm, wenn alle drei Sensoren unübliche Werte in kurzen Abständen aufweisen. Dieser Alarm beinhaltet ein akustisches Signal um Betreuer oder Eltern zu informieren, sowie die Absetzung eines Notrufes um professionelle Hilfe herbeizurufen.

**Ziel** In diesem Fallbeispiel wollen wir unseren Controller-Synthese Prozess nutzen, sodass für das oben beschriebenen Apnoe-Erkennungssystem folgende Anforderungen erfüllt sind.

1. Alle Sensoren müssen mit einer gemeinsamen Komponente kommunizieren.
2. Die gemeinsame Komponente muss Nachrichten erzeugen können.
3. Der Typ der erzeugten Nachricht muss von den Messwerten aller drei Sensoren abhängen und dem farbkodierten Alarm-System entsprechen.

### Eingaben

Für unser Apnoe-Erkennungssystem (Services der Abbildungen 7.8b, 7.9b und 7.10b) stellen wir die folgenden Eigenschaften sicher. Wir geben zunächst

die formale Darstellung und im späteren Verlauf die dazugehörige textuelle Beschreibung an.

1a. $(p_4^{Acc}.x \wedge x > Acc_{Max} \wedge p_7^{PO}.y \wedge y <= PO_{Max} \wedge p_9^{ECG}.z \wedge z <= ECG_{Max}) \rightarrow (AX(c1.green) \wedge AF(p_1^{Acc}.x2 \wedge p_1^{PO}.y2 \wedge p_1^{ECG}.z2))$

1b. $(p_4^{Acc}.x \wedge x <= Acc_{Max} \wedge p_7^{PO}.y \wedge y > PO_{Max} \wedge p_9^{ECG}.z \wedge z <= ECG_{Max}) \rightarrow (AX(c1.green) \wedge AF(p_1^{Acc}.x2 \wedge p_1^{PO}.y2 \wedge p_1^{ECG}.z2))$

1c. $(p_4^{Acc}.x \wedge x <= Acc_{Max} \wedge p_7^{PO}.y \wedge y <= PO_{Max} \wedge p_9^{ECG}.z \wedge z > ECG_{Max}) \rightarrow (AX(c1.green) \wedge AF(p_1^{Acc}.x2 \wedge p_1^{PO}.y2 \wedge p_1^{ECG}.z2))$

2a. $(p_4^{Acc}.x \wedge x > Acc_{Max} \wedge p_7^{PO}.y \wedge y > PO_{Max} \wedge p_9^{ECG}.z \wedge z <= ECG_{Max}) \rightarrow (AX(c1.orange) \wedge AF(p_1^{Acc}.x2 \wedge p_1^{PO}.y2 \wedge p_1^{ECG}.z2))$

2b. $(p_4^{Acc}.x \wedge x > Acc_{Max} \wedge p_7^{PO}.y \wedge y <= PO_{Max} \wedge p_9^{ECG}.z \wedge z > ECG_{Max}) \rightarrow AF(p_1^{Acc}.x2 \wedge p_1^{PO}.y2 \wedge p_1^{ECG}.z2)$

2c. $(p_4^{Acc}.x \wedge x <= Acc_{Max} \wedge p_7^{PO}.y \wedge y > PO_{Max} \wedge p_9^{ECG}.z \wedge z > ECG_{Max}) \rightarrow (AX(c1.orange) \wedge AF(p_1^{Acc}.x2 \wedge p_1^{PO}.y2 \wedge p_1^{ECG}.z2))$

3 $(p_4^{Acc}.x \wedge x > Acc_{Max} \wedge p_7^{PO}.y \wedge y > PO_{Max} \wedge p_9^{ECG}.z \wedge z > ECG_{Max}) \rightarrow (AX(c1.red) \wedge AF(p_1^{Acc}.x2 \wedge p_1^{PO}.y2 \wedge p_1^{ECG}.z2))$

Diese Eigenschaften repräsentieren das von den gesendeten Daten abhängige, farbkodierte Alarm-System.

Die Eigenschaften 1a.-1c. definieren hierbei die erste Ebene. Wenn die Auswertung der Sensordaten zeigt, dass nur ein Schwellwert überschritten wird, produziert der Controller einen grünen Alarm.

Die Eigenschaften 2a.-2c. spezifizieren die zweite Ebene des farbkodierten Alarm-Systems. Sollten zwei Schwellwerte überschritten werden, so erzeugt der Controller einen orangenen Alarm. Eigenschaft 2b. sorgt für die Einhaltung der oben beschriebenen Zusammenhänge zwischen Atmung, Herzfrequenz und Sauerstoffsättigung.

Eigenschaft 3 beschreibt die dritte Ebene und spezifiziert, dass ein roter Alarm gesendet wird, wenn alle drei Sensoren auffällige Werte anzeigen.

Die Sensoren werden nach dem möglichen Senden eines Alarms durch Nachrichten auf den Eingangsplätzen neu gestartet.

### Resultat

Um unseren Controller-Synthese Prozess auf die drei gegebenen Services anwenden zu können, fassen wir den Bewegungssensor und den Lichtsensor als *einen* Service auf. Die Anwendung unseres Prozesses auf die Services (Bewegungs-Licht-sensor und EKG-Sensor) und den beschriebenen Eigenschaften resultiert in folgenden Berechnungszeiten.

| Schritt 1 | Schritt 2 | Schritt 3 | Schritt 4 | Schritt 5 | Gesamt |
|---|---|---|---|---|---|
| 41 ms | 95 ms | 1104 ms | 294 ms | 2155333 ms | 2156985 ms |

**Tabelle 7.3:** Zeitverbrauch des Apnoe-Erkennungs-System

Wie in Tabelle 7.3 deutlich wird, ist auch hier die Zurückführung eines RCTL-Netzes auf ein APN der berechnungsintensivste Part unseres Ansatzes. Grund dafür ist die Berechnung der Transitions-Potenzmenge. Daher wird auch deutlich, dass der Prozess mit steigender Anzahl an Eigenschaften immer komplexer wird.

Der synthetisierte Controller dieser Fallstudie besteht aus 32.951 Transitionen, 17 Plätzen und 737.794 Kanten. Er ist somit zu komplex um ihn hier abzubilden.

Um dennoch einen groben Überblick zu liefern, stellen wir im Folgenden das synthetisierte RCTL-Netz (komplexitttsbedingt nur schematisch) dar.

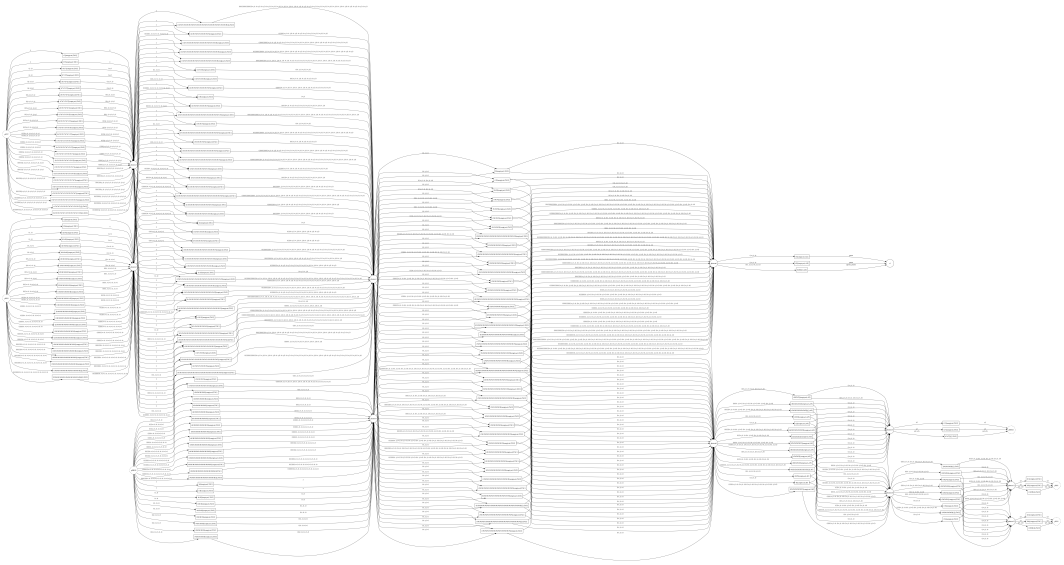


**Abbildung 7.11:** Schematische Darstellung des synthetisierten Controllers (RCTL-Netz) für das Apnoe-Erkennungssystem

Bei der Eingabe von acht Verhaltenseigenschaften war auch nach 48 Stunden die Berechnung des Controllers noch nicht abgeschlossen, sodass wir den Prozess manuell abbrachen.

## 7.5 Zusammenfassung

In diesem Kapitel haben wir die Hauptbestandteile unserer Implementierung vorgestellt. Anschließend haben wir unser laufendes Beispiel noch einmal kurz zusammengefasst und unsere zwei medizinischen Fallstudien eingeführt. Diese

haben wir zur Evaluierung unseres Prozesses genutzt und dessen Zeitverbrauch angegeben. In der Diskussion haben wir den Zurückführungsschritt als den aufwändigsten Schritt identifiziert. Um die Skalierbarkeit unseres Prozesses einschätzen zu können, haben wir sowohl die Anzahl der Eigenschaften als auch die Komplexität der Services verändert.

Im Allgemeinen wächst der Berechnungsaufwand exponentiell mit steigender Anzahl und Komplexität von Eigenschaften und Services. Die hauptverantwortlichen Schritte unseres Synthese Prozesses sind dabei die Komposition und die Zurückführung eines RCTL-Netzes auf ein APN. Die Komplexität des Zurückführungsschrittes steigt mit steigender Zahl an Transition innerhalb des extrahierten Controllers. Der Grund dafür besteht darin, dass im schlimmsten Fall für jede Permutation von Transitionen eine neue erzeugt wird.

Für unsere Fallstudien konnten wir jedoch jeweils einen Controller in akzeptabler Zeit synthetisieren, was auf die praktische Anwendbarkeit unseres Ansatzes hindeutet.

# 8 Zusammenfassung & Ausblick

In diesem finalen Kapitel fassen wir die Resultate dieser Arbeit noch einmal zusammen (Abschnitt 8.1) und diskutieren sie hinsichtlich der Beantwortung der Forschungsfragen, die zu Beginn eingeführt wurden (Abschnitt 8.2). Abschließend geben wir noch einen Ausblick auf zukünftige Arbeiten (Abschnitt 8.3).

## 8.1 Resultate

Durch die steigende Nachfrage nach computergestützten Systemen wachsender Komplexität, gewinnen Strategien immer mehr an Bedeutung, die Wiederverwendbarkeit und Wartbarkeit zunehmend unterstützen. Aus diesem Trend entwickelte sich das Paradigma der serviceorientierten Architekturen, bei dem Funktionalitäten als Services gekapselt und bereitgestellt werden, die dann wiederverwendet und neu kombiniert werden können. Damit einhergehend entstanden verschiedene neue Forschungsfelder. Eines der größten ist dabei das Gebiet der Service-Komposition. Dazu werden sogenannte Controller konstruiert, die mit den zu komponierenden Services interagieren und den Nachrichtenaustausch zwischen ihnen koordinieren. Aktuell existierende Controller-Synthese-Verfahren bieten jedoch weder eine formale Datenbehandlung noch eine Berücksichtigung datenabhängigen Verhaltens, wodurch hoher manueller Aufwand notwendig wird und den Prozess damit fehleranfällig und zeitaufwändig macht. Gerade in sicherheitskritischen Bereichen, wie der Medizintechnik, ist Fehleranfälligkeit jedoch nicht tolerierbar.

Wir haben daher in dieser Arbeit einen Ansatz zur automatisierten Controller-Synthese für Services mit Daten vorgestellt. Dessen Ziel ist es, für zwei gegebene Services mit Daten $S_1, S_2$ und einer Menge datenabhängiger Verhaltenseigenschaften $\Phi = \{\varphi_1, \varphi_2, \dots\}$, einen Controller $C$ derart zu synthetisieren, dass $\varphi_1 \wedge \varphi_2 \wedge \dots$ in dem komponierten System $S_1 \otimes C \otimes S_2$ gilt und die Komposition somit korrekt ist. Dabei synthetisieren wir $C$ direkt aus $S_1, S_2$ und $\Phi$.

Um in diesen Prozess eine formale Datenbehandlung zu integrieren, haben wir angenommen, dass die Verhaltenseigenschaften als RCTL-Formeln $\varphi_1, \varphi_2, \dots$ und das Verhalten der Services als APNs $S_1, S_2$ gegeben sind.

Auf diesen Annahmen aufbauend führen wir unseren fünfschrittigen Prozess durch.

Im ersten Schritt verkleinern wir zunächst die Service-Modelle, sodass nur noch das für die Kommunikation wichtige Verhalten erhalten bleibt. Diesen Schritt bezeichnen wir als *Extraktion beobachtbaren Verhaltens*. Wir haben dafür einen Algorithmus definiert, der aus einem APN $S$ ein Menge von RCTL-Formeln $\Psi_S = \{\psi_S^1, \psi_S^2, \dots\}$ extrahiert, die das Verhalten von $S$ an den Interface-Ports spezifizieren.

Im zweiten Schritt übersetzen wir diese RCTL-Formeln ( $\psi_{S_1}^1, \psi_{S_1}^2, \dots,$ $\psi_{S_2}^1, \psi_{S_2}^2, \dots$ ) sowie die Verhaltenseigenschaften $(\varphi_1, \varphi_2, \dots)$ in algebraische Petrinetze $(N_{\psi_{S_1}^1}, N_{\psi_{S_1}^2}, \dots, N_{\psi_{S_2}^1}, N_{\psi_{S_2}^2}, \dots, N_{\varphi_1}, N_{\varphi_2}, \dots)$. Um die Semantik der Formeln erhalten zu können, haben wir die originalen algebraischen Petrinetze um eine Funktion erweitert, die jeder Transition einen temporalen-Pfad Operator (oder im Falle, dass dieser nicht notwendig ist $\tau$) zuweist. Die so konstruierten Netze bezeichnen wir als *RCTL-Netze*.

Im dritten Schritt komponieren wir die aus der Übersetzung resultierenden RCTL-Netze $(N_{\psi_{S_1}^1}, N_{\psi_{S_1}^2}, \dots, N_{\psi_{S_2}^1}, N_{\psi_{S_2}^2}, \dots, N_{\varphi_1}, N_{\varphi_2}, \dots)$. Unser Kompositionsalgorithmus ist dabei so definiert, dass $\bigwedge\{\psi_{S_1}^1, \psi_{S_1}^2, \dots, \psi_{S_2}^1, \psi_{S_2}^2, \dots,$ $\varphi_1, \varphi_2, \dots\}$ in $\bigotimes\{N_{\psi_{S_1}^1}, N_{\psi_{S_1}^2}, \dots, N_{\psi_{S_2}^1}, N_{\psi_{S_2}^2}, \dots, N_{\varphi_1}, N_{\varphi_2}, \dots\}$ gilt. Als Resultat erhalten wir ein RCTL-Netz, das Service-spezifisches Verhalten aufweisen kann. Dies ist für den gesuchten Controller nachteilig, da dieser nur das Kommunikationsverhalten zwischen den Services abbilden soll. Um dies zu behandeln, haben wir für den vierten Schritt unseres Prozesses einen Extraktionsalgorithmus entwickelt, der das Controller-Verhalten separiert.

Im fünften und letzten Schritt nutzen wir das aus der Extraktion resultierende RCTL-Netz und überführen dies zurück in ein APN. Bei diesem APN handelt es sich um den gesuchten Controller, der mit den originalen Service-Modellen komponiert werden kann.

Da alle Stufen automatisiert und korrekt arbeiten, erfüllt das komponierte System (bestehend aus den Services und dem Controller) alle spezifizierten Eigenschaften per Konstruktion. Darauf aufbauend arbeitet der Controller korrekt für jede Instantiierung, die eine $SP_R$-Algebra der algebraischen Spezifikation des komponierten Systems darstellt.

Um unseren Prozess beispielhaft zu demonstrieren, haben wir diesen in Form eines Prototypen implementiert und anhand von drei Fallbeispielen aus dem medizinischen Bereich (Interoperabilität einer Pumpe und eines Schalters, einer künstlichen Bauchspeicheldrüse und eines Apnoe-Erkennungssystem) evaluiert. Die Ergebnisse zeigen, dass die Berechnungszeiten abhängig von der Anzahl und Struktur der Verhaltenseigenschaften sind, wir für die Fallstudien jedoch akzeptable Zeiten erhalten. Diese Zeiten sind besser als die zu erwartenden Berechnungszeiten beim Model-Checking.

## 8.2 Diskussion

In Abschnitt 1.2.2 haben wir mittels Forschungsfragen Kriterien aufgestellt, die unser Ansatz zur automatischen Controller-Synthese für Services mit Daten erfüllen soll. In diesem Abschnitt diskutieren wir die Resultate dieser Arbeit und erläutern inwiefern wir diese Kriterien erfüllt haben.

1. **Asynchrone Kommunikation**

   Generell sollten Ansätze zur Controller-Synthese mit asynchroner Kommunikation umgehen können um bei praktischen Problemen anwendbar zu sein. In unserem Ansatz stellen wir dies durch die Nutzung von *algebraischen Petrinetzen (APNs)* sicher. Sie stellen die Möglichkeit bereit, Nachrichten auf den Interface-Ports eines Services zu speichern, die jederzeit von einem anderen Service empfangen werden können.

2. **Spezifikation komplexer, datenabhängiger Eigenschaften**

   Durch unsere Definition einer speziellen Menge von atomaren Propositionen, sind wir in der Lage auch datenabhängige Eigenschaften zu spezifizieren. Die von uns definierte und auf den eben genannten atomaren Propositionen aufbauende Logik RCTL ermöglicht darüber hinaus temporale Pfad-Operatoren beliebig zu schachteln. Somit können wir auch komplexere, datenabhängige Verhaltenseigenschaften spezifizieren.

3. **Handhabung datenabhängigen Verhaltens**

   In allen Schritten unseres Synthese-Prozesses werden Datenabhängigkeiten beachtet.

   Genauer gesagt werden zunächst während der *Extraktion beobachtbaren Verhaltens* alle für die Kommunikation wichtigen Pfade und Datenabhängigkeiten mit Hilfe von RCTL-Formeln spezifiziert. Diese bleiben auch bei der *Übersetzung* von RCTL-Formeln in RCTL-Netze erhalten. Bei der *Komposition* der entstandenen RCTL-Netze werden sie zudem dazu genutzt, sich widersprechende Eigenschaften zu erkennen. Sollten diese nicht existieren, so entsteht der Controller durch den *Extraktionsschritt* und die anschließende Zurückführung eines RCTL-Netzes auf ein APN. In beiden Schritten bleiben die vorhandenen Datenabhängigkeiten ebenfalls erhalten.

   Somit ist unser Synthese-Prozess in der Lage datenabhängiges Verhalten zu Erkennen und adäquat zu handhaben.

4. **Korrektheit-per-Konstruktion**

   Unser Synthese-Prozess ist eine Hintereinanderausführung von fünf Schritten. Diese sind: *Extraktion beobachtbaren Verhaltens, Übersetzung von RCTL-Formeln in RCTL-Netze, Komposition von RCTL-Netzen, Extraktion des Controllers und Zurückführung eines RCTL-Netzes auf ein APN.* Jeder dieser Schritte nutzt das Resultat des vorangegangenen Schrittes als Eingabe. Da alle dieser einzelnen Schritte korrekt arbeiten, re-

sultiert der gesamte Prozess in einem Controller, der per Konstruktion korrekt ist.

5. **Automatisierte Synthese**

   Alle Schritte in unserem Controller-Synthese Prozess haben wir in Form eines Prototypen implementiert, so dass sie voll-automatisch arbeiten. Um ein Indiz für die Anwendbarkeit zu liefern, haben wir unseren Prototypen auf drei medizinische Fallstudien angewendet und die Berechnungszeiten gemessen.

## 8.3 Ausblick

Mit unserem Ansatz ist es möglich einen Controller zu synthetisieren, der garantiert, dass das komponierte System (bestehend aus Services und Controller) die gewünschten Verhaltenseigenschaften erfüllt. Aktuell nehmen wir an, dass diese Eigenschaften als RCTL-Formeln und das Verhalten der Services als APNs gegeben sind. Diese und andere von uns getroffenen Einschränkungen bieten großes Potential unseren Ansatz weiter zu verbessern und seine Praktikabilität zu erhöhen. In diesem Abschnitt diskutieren wir zum einen wie man die Wahrscheinlichkeit erhöht, dass ein Controller mit vertretbarem Aufwand synthetisiert werden kann, zum anderen, wie die Ausdrucksstärke der Formel verbessert werden kann und wie existierende Ansätze mit unserem kombiniert werden können.

### 8.3.1 Priorisierung der Eigenschaften

Es ist möglich, dass sich Eigenschaften durch eine fehlerhafte Spezifikation widersprechen. Dies kann beispielsweise bei der Sicherstellung von *Quality of Service* (*QoS*)-Kriterien wie z.B. maximale Anzahl von Zuständen, Antwortzeiten, etc. der Fall sein. Durch Modifikationen könnte unser Ansatz dazu genutzt werden, zu erkennen, dass ein QoS-Kriterium nicht erfüllbar ist, wenn die notwendigen Eigenschaften kombiniert werden. Eine Vorarbeit dazu wurde bereits in [Kab15] Unser Synthese-Prozess könnte dahingehend erweitert werden, Gegenbeispiele automatisiert zu ermitteln, die zu widersprüchlichen Eigenschaften zu erkennen und in geeigneter Weise dem Nutzer für eine manuelle Unterstützung zur Verfügung zu stellen.

Die Wahrscheinlichkeit für ein solch „negatives“ Ergebnis könnte in diesem Fall durch die Einführung von Prioritäten hinsichtlich der Verhaltenseigenschaften verringert werden. Dies würde eine Unterscheidung zwischen notwendigen und optionalen Eigenschaften ermöglichen. Damit könnte zunächst ein Controller synthetisiert werden, der zumindest die wichtigsten Eigenschaften gewährleistet. Dazu könnten vorrangig die notwendigsten Eigenschaften komponiert und so ein erster Entwurf eines Controllers bereit gestellt werden. Der Nutzer kann diesen Schritt für Schritt um weniger wichtige Eigenschaften erweitern und widersprüchliche Eigenschaften ausschließen.

### 8.3.2 Verbesserung der Ausdrucksstärke von RCTL

Es gibt einige Möglichkeiten die Spezifikation der Eigenschaften zu verbessern. In diesem Abschnitt betrachten wir die beiden erfolgversprechendsten.

#### Spezifikation kapizitätsgebundener Eigenschaften

Aktuell erlauben wir die Definition von Kapazitäten nur in den Service-Modellen. Diese Informationen werden während der *Extraktion beobachtbaren Verhaltens* vernachlässigt, da derzeit keine Kapazitätseinschränkungen durch Formeln spezifiziert werden können. Es wäre daher sehr vorteilhaft, die Definition von RCTL durch neue atomare Propositionen zu erweitern und die einzelnen Schritte unseres Synthese Prozesses darauf anzupassen.

#### Erweiterung auf CTL

Wenn wir die Ausdrucksstärke der RCTL-Formeln erhöhen wollen, so ist die Integration der Negation am erfolgversprechendsten. Damit wäre es möglich, das Erreichen eines spezifischen Zustandes zu verhindern, sobald ein anderer eingetreten ist. Nehmen wir beispielsweise an, dass zwei Pumpen koordiniert werden müssen, wobei nur eine davon injizieren darf. Das bedeutet, sobald eine der beiden aktiv ist, muss die andere inaktiv bleiben. Dies wird in der folgenden Eigenschaft spezifiziert.

$$Pump1inject.x \rightarrow AG(\neg Pump2inject.y). \tag{8.1}$$

Für die folgenden Erklärungen nehmen wir an, dass die generelle Struktur der RCTL-Formeln erhalten bleibt. Das bedeutet, auf der höchsten Ebene der Formel befindet sich eine Implikation. Die Integration der Negation auf der linken Seite der Implikation kann durch Transformation jeder Formel in eine (disjunktive oder konjunktive) Normalform und durch die Nutzung von Inhibitorkanten realisiert werden. Wesentlich schwerer ist die Integration der Negation auf der rechten Seite der Implikation. Mit den vorhandenen Mitteln ist es nicht möglich sicherzustellen, dass ein bestimmter Platz nicht markiert werden darf. Daher muss man die Syntax und Semantik der RCTL-Netze bspw. durch eine neue Kanten- und Transitionsart erweitern, wobei existierende Analysemethoden gültig bleiben sollten.

Eine weitere Verbesserung würde die Integration des temporalen Pfad-Operators $E(\varphi U \psi)$ $(EU)$ bringen. Damit wäre es möglich sicherzustellen, dass ein Zustand so lange gilt, bis ein anderer Zustand erreicht wird. Auch dies würde sich insbesondere im medizinischen Bereich als vorteilhaft erweisen. Als Beispiel dafür nehmen wir an, dass eine Pumpe mit einem synthetisierten Controller verbunden ist. Möglicherweise ist es notwendig sicherzustellen, dass sobald die Pumpe leer ist, der Controller so lange periodisch wiederkehrend eine

Alarmnachricht sendet, bis die Pumpe wieder aufgefüllt ist. Dies kann durch die folgende Formel spezifiziert werden.

$$\begin{aligned}(fillLevelPump.x \wedge (x == 0)) \rightarrow \\ AG(\text{cSendAlarm.y U } (fillLevelPump.x \wedge (x > 0)))\end{aligned} \tag{8.2}$$

Für diesen Fall sollte die Menge möglicher Transitionslabel um den Operator *EU* erweitert werden. Dies müsste dann auch im Kompositionsschritt beachtet werden.

Wenn die Negation und der Until-Operator ($EU$) integriert sind, bilden die zugelassen Operatoren ($a \in AP, \wedge, \neg, EF, EX, EG, EU$) eine Junktorbasis und somit würde CTL als Ganzes unterstützt werden.

### 8.3.3 Kombination mit existierenden Arbeiten

Da unser Controller-Synthese Prozess APNs und CTL-Formeln als Eingabe erfordert und in einem APN resultiert, ist dessen industrielle Anwendbarkeit und die Kombination mit existierenden Frameworks eine höchst interessante Frage. In diesem Abschnitt präsentieren wir existierende Ansätze, die mit unserem Prozess kombiniert werden können. Dadurch werden die Anforderungen an die Eingaben unseres Prozesses gelockert und somit dessen Nutzbarkeit und Akzeptanz erhöht.

Bevor wir jedoch ins Detail gehen, wollen wir noch einmal darauf hinweisen, dass wir insbesondere jene Anforderungen betrachten, die in der sicherheitskritischen Domäne der medizinischen Geräte auftreten. Dadurch können wir die Vorteile nutzen, die die Medizinprodukte-Richtlinien [Eur07] aufweisen. Diese wurden vom europäischen Parlament verabschiedet, um eine korrekte und sichere Funktionalität von medizinischen Geräten sicherzustellen. Ihre deutsche Umsetzung, das Medizinprodukte-Gesetz [Ger11], fordert von den Herstellern solcher Geräte eine vollständige Qualitätssicherung. Dies umfasst eine Dokumentation der gesamten Funktionalität jedes Gerätes und eine Beschreibung des Kommunikationsverhaltens. Daher ist es legitim anzunehmen, dass eine textuelle Beschreibung für jedes Gerät verfügbar ist.

Um von dieser Situation zu profitieren und die Annahmen an die Eingaben unseres Prozesses zu lockern, können wir eine Kette von Ansätzen nutzen, die eine strukturierte, textuelle Beschreibung in eine sehr spezielle Form der *Unified Modeling Language* (*UML*) [DB09] übersetzt. Das Ergebnis wird anschließend in ein high-level Petrinetz [BP01] überführt. Desweiteren wäre es mit einer Reihe von Anpassungen möglich, die geforderten RCTL-Formeln durch die Übersetzung natürlicher Sprache in temporale Logik [NF96] automatisch zu erhalten. Die angegebenen Ansätze können zwar einzelne Worte erkennen und in Verbindung setzen, jedoch können sie bisher nicht adäquat mit der semantischen Bedeutung der Worte sowie mit dem sie umgebenen Kontext umgehen.

Dieses Problem müsste bei einer gewünschten Anwendung im Vorfeld behoben werden.

Um unsere Resultate von der Modell- hin zur Code-Ebene zu transformieren und somit industrielle Anwendbarkeit zu erreichen, können wir mit Hilfe komplexerer Anpassungen einen Ansatz nutzen, der eine spezielle Art von high-level Petrinetzen in *Web Service - Business Process Execution Language* (*WS-BPEL*)-Code übersetzt [LVdA06].

WS-BPEL ist eine XML-basierte, oft genutzte Sprache zur Modellierung von Geschäftsprozessen, deren Aktivitäten als Web-Services bereitgestellt werden. Mit WS-BPEL-Engines wird eine Ausführung von Instanzen solcher Geschäftsprozesse ermöglicht. Damit hat dieser Ansatz das Potential die Ausführung unseres synthetisierten Controllers sicherzustellen und somit ein lauffähiges System, bestehend aus dem Controller und den medizinischen Geräten, zur Verfügung zu stellen.

Zusammenfassend kann gesagt werden, dass eine angemessene Anpassung dieser und ähnlicher Ansätze in Kombination mit unserem Synthese-Prozess das Potential hat, voll-automatisch Controller zu synthetisieren, die hinsichtlich datenabhängiger Verhaltenseigenschaften korrekt sind. Damit wird eine sichere und zuverlässige Interoperabilität medizinischer Geräte nicht nur auf der Modell- sondern auch auf der Anwendungsebene erreicht.

# Abbildungsverzeichnis

# Liste der Algorithmen

# Tabellenverzeichnis

# Akronymverzeichnis

**APN** algebraisches Petrinetz . . . . . 7

**BPMN** Business Process Management Notation . . . . . 60

**CBSE** Component-Based Software Engineering . . . . . 52

**CPN** Colored Petri Net . . . . . 56

**CTL** Computation Tree Logic . . . . . 7

**LTL** Lineare Temporale Logik . . . . . 58

**LTS** (Knoten-)gelabeltes Transitionssystem . . . . . 46

**PSL** Property Specification Language . . . . . 58

**QoS** Quality of Service . . . . . 164

**RCTL** Restricted Computation Tree Logic . . . . . 7

**SOA** serviceorientierte Architektur . . . . . 3

# Symbolverzeichnis

# Literaturverzeichnis


[Abe90] Dirk Abel. *Petri-Netze für Ingenieure*. Springer, 1990.

[AMM14] Benjamin Aminof, Fabio Mogavero, and Aniello Murano. Synthesis of hierarchical systems. *Science of Computer Programming*, 83:56–79, 2014.

[AMSW09] WilM.P. Aalst, ArjanJ. Mooij, Christian Stahl, and Karsten Wolf. Service interaction: Patterns, formalization, and analysis. In Marco Bernardo, Luca Padovani, and Gianluigi Zavattaro, editors, *Formal Methods for Web Services*, volume 5569 of *Lecture Notes in Computer Science*, pages 42–88. Springer Berlin Heidelberg, 2009.

[ASM08] Jalal M Abu-Shaweesh and Richard J Martin. Neonatal apnea: what's new? *Pediatric pulmonology*, 43(10):937–944, 2008.

[BBC02] Andrea Bracciali, Antonio Brogi, and Carlos Canal. Adapting components with mismatching behaviours. In Judith Bishop, editor, *Component Deployment*, volume 2370 of *Lecture Notes in Computer Science*, pages 185–199. Springer Berlin Heidelberg, 2002.

[BBC05] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45 – 54, 2005.

[BBT01] Andrea Bracciali, Antonio Brogi, and Franco Turini. Coordinating interaction patterns. In *Proceedings of the 2001 ACM symposium on Applied computing*, SAC '01, pages 159–165, New York, NY, USA, 2001. ACM.

[BCDG+05a] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Ri-

chard Hull, and Massimo Mecella. Automatic composition of transition-based semantic web services with messaging. In *Proceedings of the 31st international conference on Very large data bases*, pages 613–624. VLDB Endowment, 2005.

[BCDG+05b] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Richard Hull, and Massimo Mecella. On-line appendix to the paper „automatic composition of transition-based semantic web services with messaging“. Technical report, http://www.dis.uniroma1.it/∼mecella/publications/eService/AppendixVLDB2005.pdf, Last Checked: 26.02.2017, 2005.

[BCR15] Walid Belkhir, Yannick Chevalier, and Michael Rusinowitch. Parametrized automata simulation and application to service composition. *Journal of Symbolic Computation*, 69:40–60, 2015.

[BGJ+07] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Interactive presentation: Automatic hardware synthesis from specifications: a case study. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1188–1193. EDA Consortium, 2007.

[BKŘS12] Tomáš Babiak, Mojmír Křetínskỳ, Vojtěch Řehák, and Jan Strejček. Ltl to büchi automata translation: Fast and more deterministic. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 95–109. Springer, 2012.

[Bor15] Valentin Borcea. Insulin pumpen-therapie. http://www.diabetes-gelsenkirchen.de/praxis_neustr/?dir=insulinpumpen-therapie, Last Checked: 26.02.2017, 2015.

[BP01] Luciano Baresi and Mauro Pezzè. On formalizing uml with high-level petri nets. pages 276–304. Springer-Verlag, 2001.

[BP10] George Baryannis and Dimitris Plexousakis. Automated web service composition: State of the art and research challenges. *ICS-FORTH, Tech. Rep*, 409, 2010.

[BT17] Franziska Bathelt-Tok. Controller-synthesis tool, 2017. `https://www.dropbox.com/sh/fm2qkf3gczs2z5h/AABbGCJzNi7g9cH5D5-Hsa7Ua?dl=0`.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Program-*

*ming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[CHT03] Kishore Channabasavaiah, Kerrie Holley, and Edward Tuggle. Migrating to a service-oriented architecture. *IBM DeveloperWorks*, 16, 2003.

[CPS08] C. Canal, P. Poizat, and G. Salaun. Model-based adaptation of behavioral mismatching components. *Software Engineering, IEEE Transactions on*, 34(4):546 –563, july-aug. 2008.

[DB09] D.K. Deeptimahanti and M.A Babar. An automated tool for generating uml models from natural language requirements. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 680–682, Nov 2009.

[DGV99] Marco Daniele, Fausto Giunchiglia, and Moshe Y Vardi. Improved automata generation for linear temporal logic. In *Computer Aided Verification*, pages 249–260. Springer, 1999.

[DM12] Kristian Duske and Richard Müller. A survey on approaches for timed services. In *ZEUS*, pages 66–73. Citeseer, 2012.

[DS05] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International journal of web and grid services*, 1(1):1–30, 2005.

[DSW06] Marlon Dumas, Murray Spork, and Kenneth Wang. *Adapt or perish: Algebra and visual notation for service interface adaptation*. Springer, 2006.

[EDH11] Daniela Elleri, David Dunger, and Roman Hovorka. Closed-loop insulin delivery for treatment of type 1 diabetes. *BMC medicine*, 9(1):120, 2011.

[EMC+01] H Ehrig, B Mahr, F Cornelius, M Große-Rhode, and P Zeitz. *Mathematisch-strukturelle Grundlagen der Informatik*. Springer DE, 2001.

[Eur07] European Parliament. Medical device directive. http://ec.europa.eu/growth/single-market/european-standards/harmonised-standards/medical-devices/index_en.htm, Last Checked: 26.02.2017, 2007.

[FL79] Michael J Fischer and Richard E Ladner. Propositional dynamic logic of regular programs. *Journal of computer and system sciences*, 18(2):194–211, 1979.

[Ger11] German Government. Medical devices act. http://www.bundesgesundheitsministerium.de/themen/gesundheitswesen/medizinprodukte/definition-und-wirtschaftliche-bedeutung.html, Last Checked: 26.02.2017, 2011.

[GKN15] Emden R. Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot. http://www.graphviz.org/pdf/dotguide.pdf, Last Checked: 26.02.2017, 2015.

[GL00] Stephen J Garland and Nancy A Lynch. Using i/o automata for developing distributed systems. *Foundations of Component-Based Systems*, 13:285312, 2000.

[GMW12] Christian Gierds, Arjan J. Mooij, and Karsten Wolf. Reducing adapter synthesis to controller synthesis. *IEEE T. Services Computing*, 5(1):72–85, 2012.

[GO01] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In *Computer Aided Verification*, pages 53–65. Springer, 2001.

[Gro14] Object Management Group. Business process modeling notation (bpmn) specification. http://www.bpmn.org/, Last Checked: 26.02.2017, January 2014.

[HC01] George T. Heineman and William T. Councill. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic logic*, volume 206. MIT press Cambridge, 2000.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[Inc] LifeScan Inc. Insulin therapy, 2015. http://www.onetouch.com/articles/insulintherapy, Last Checked: 20.01.2017.

[JB06] Barbara Jobstmann and Roderick Bloem. Optimizations for ltl synthesis. In *Formal Methods in Computer Aided Design, 2006. FMCAD'06*, pages 117–124. IEEE, 2006.

[Jen91] Kurt Jensen. Coloured petri nets: A high level language for system design and analysis. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 342–416. Springer-Verlag Berlin, Heidelberg, 1991.

[Jen97] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer-Verlag, 1997.

[KV00] Orna Kupfermant and Moshe Y Vardit. Synthesis with incomplete information. In *Advances in Temporal Logic*, pages 109–127. Springer-Verlag, 2000.

[KVW00] Orna Kupferman, Moshe Y Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM (JACM)*, 47(2):312–360, 2000.

[LVdA06] Kristian Bisgaard Lassen and Wil MP Van der Aalst. *WorkflowNet2BPEL4WS: A tool for translating unstructured workflow processes to readable BPEL*. Springer, 2006.

[LW11] Niels Lohmann and Karsten Wolf. Data under control. In *Proceedings of the 18th German Workshop on Algorithms and Tools for Petri Nets (AWPN 2011)*, pages 34–40, Hagen, Germany, 2011.

[MBC12] Violeta Monasterio, Fred Burgess, and Gari D Clifford. Robust classification of neonatal apnoea-related desaturations. *Physiological measurement*, 33(9):1503, 2012.

[Mel10] Ingo Melzer. *Service-orientierte Architekturen mit Web Services: Konzepte-Standards-Praxis*. Springer-Verlag, 2010.

[Mil99] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[MNBM+07] Hamid Reza Motahari Nezhad, Boualem Benatallah, Axel Martens, Francisco Curbera, and Fabio Casati. Semi-automated adaptation of service interactions. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 993–1002, New York, NY, USA, 2007. ACM.

[MPP02] Massimo Mecella, Francesco Parisi Presicce, and Barbara Pernici. Modeling e-service orchestration through petri nets. In *Technologies for E-Services*, pages 38–47. Springer, 2002.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and computation*, 100(1):1–40, 1992.

[NF96] Rani Nelken and Nissim Francez. *Automatic translation of natural language system specifications into temporal logic*, pages 360–371. Springer - Verlag, Berlin, Heidelberg, 1996.

[PBT14] Marco Pistore, Renato Bettin, and Paolo Traverso. Symbolic techniques for planning with extended goals in non-deterministic domains. In *Sixth European Conference on Planning*, 2014.

[Pet62] Carl Adam Petri. Kommunikation mit automaten. *Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr*, 2, 1962.

[PPS06] Nir Piterman, Amir Pnueli, and Yaniv SaŠar. Synthesis of reactive (1) designs. In *Verification, Model Checking, and Abstract Interpretation*, pages 364–380. Springer, 2006.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *Software Engineering, IEEE Transactions on*, 28(11):1056–1076, 2002.

[PVDH07] Mike P Papazoglou and Willem-Jan Van Den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB journal*, 16(3):389–415, 2007.

[Rei91] W. Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80:1–34, 1991.

[Rei10] Wolfang Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 15 July 2010. 248 pages; ISBN 978-3-8348-1290-2.

[Reu01] Ralf H Reussner. Adapting components and predicting architectural properties with parameterised contracts. *Tagungsband des Arbeitstreffens der GI Fachgruppen*, 2(4):33–43, 2001.

[RS05] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In *Semantic Web Services and Web Process Composition*, pages 43–54. Springer, 2005.

[SB00] Fabio Somenzi and Roderick Bloem. Efficient büchi automata from ltl formulae. In *Computer Aided Verification*, pages 248–263. Springer, 2000.

[Sch96] Karsten Schmidt. *Symbolische Analysemethoden für algebraische Petri-Netze*. PhD thesis, Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät II, 1996.

[Sch05] Karsten Schmidt. Controllability of open workflow nets. In *EMISA*, volume 75, pages 236–249, 2005.

[SG13] Daniel Stohr and Sabine Glesner. Planning in real-time domains with timed ctl goals via symbolic model checking. In *Theoretical Aspects of Software Engineering (TASE), 2013 International Symposium on*, pages 7–14. IEEE, 2013.

[SH87] John W Severinghaus and Yoshiyuki Honda. History of blood gas analysis. vii. pulse oximetry. *Journal of clinical monitoring*, 3(2):135–138, 1987.

[Shi] 2015 Shimmer. http://www.shimmersensing.com/, last checked: 26.02.2017.

[SMKF12] Yang Syu, Shang-Pin Ma, Jong-Yin Kuo, and Yong-Yi FanJiang. A survey on automated service composition methods and related techniques. In *Services Computing (SCC), 2012 IEEE Ninth International Conference on*, pages 290–297. IEEE, 2012.

[SR02] Heinz W Schmidt and Ralf H Reussner. Generating adapters for concurrent component protocol synchronisation. In *Formal Methods for Open Object-Based Distributed Systems V*, pages 213–229. Springer, 2002.

[Vau87] Jacques Vautherin. Parallel systems specifications with coloured petri nets and algebraic specifications. In *Advances in Petri Nets 1987*, pages 293–308. Springer, 1987.

[VdA05] Wil MP Van der Aalst. Pi calculus versus petri nets: Let us eat Şhumble pieŤ rather than further inflate the Şpi hypeŤ. *BPTrends*, 3(5):1–11, 2005.

[Wag11] Christoph Wagner. A data-centric approach to deadlock elimination in business processes. *Services und ihre Komposition*, 2011.

[Wag12] Christoph Wagner. Partner synthesis for data-dependent services. In *ZEUS*, pages 17–24. Citeseer, 2012.

[Wan08] Joseph Wang. Electrochemical glucose biosensors. *Chemical reviews*, 108(2):814–825, 2008.

[WDOV08] Kenneth Wang, Marlon Dumas, Chun Ouyang, and Julien Vayssiere. The service adaptation machine. In *IEEE Sixth European Conference on Web Services, 2008. (ECOWS'08)*, pages 145–154. IEEE, 2008.

[Wol09] Karsten Wolf. Does my service have partners? In *Transactions on Petri Nets and Other Models of Concurrency II*, pages 152–171. Springer, 2009.

[YPVdH02] Jian Yang, Mike P Papazoglou, and W-J Van den Heuvel. Tackling the challenges of service composition in e-marketplaces. In *Research Issues in Data Engineering: Engineering E-Commerce/E-Business Systems, 2002. RIDE-2EC 2002. Proceedings. Twelfth International Workshop on*, pages 125–133. IEEE, 2002.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, March 1997.

# Veröffentlichungen von Franziska Bathelt-Tok


[BT14] Franziska Bathelt-Tok. Safe and reliable interoperability of medical devices using data-dependent controller synthesis. In *The IEEE Intelligent Informatics Bulletin*, volume 15, pages 24–25, 2014.

[BTG14] Franziska Bathelt-Tok and Sabine Glesner. Towards the automated synthesis of data dependent service controllers. In *Service-Oriented Computing–ICSOC 2013 Workshops*, pages 528–534. Springer, 2014.

[BTGB14] Franziska Bathelt-Tok, Sabine Glesner, and Oliver Blankenstein. Data-dependent controller synthesis to enable reliable and safe interoperability of medical devices. In *Proceedings of the 8th International Conference on Pervasive Computing Technologies for Healthcare*, pages 162–165. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014.

[BTGGB14] Franziska Bathelt-Tok, Helena Gruhn, Sabine Glesner, and Oliver Blankenstein. Towards the development of smart and reliable health assistance networks exemplified by an apnea detection system. In *Healthcare Informatics (ICHI), 2014 IEEE International Conference on*, pages 226–231. IEEE, 2014.

# Bachelor- & Masterarbeiten betreut von Franziska Bathelt-Tok

[Hap15] Ronny Hapke. Deadlock freedom of data-dependent services. Master's thesis, Technische Universität Berlin, 2015.

[Kab15] Dennie Kabul. Subnetzerkennung bei algebraischen Petrinetzen. Master's thesis, Technische Universität Berlin, 2015.

[Kol14] Thomas Kolb. Entwicklung eines Parsers zum Einlesen von CTL-Termen und Umwandlung in Petri-Netze. Bachelor's Thesis, Technische Universität Berlin, 2014.

[Pan14] Marcus Pannwitz. Entwicklung einer Smartphone-App zur drahtlosen, sicheren und zuverlässigen Überwachung von Patienten über Shimmer-Sensoren. Master's thesis, Technische Universität Berlin, 2014.