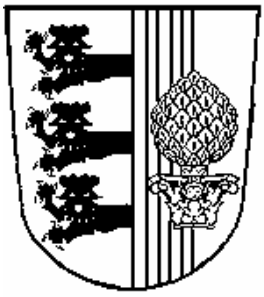
Universität Augsburg
Institut für Informatik



# Zeitbeschriftete Petri-Netze und FastAsy

**Elmar Bihler**

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

**2008**

Erstgutachter: Prof. Dr. Walter Vogler
Zweitgutachter: Prof. Dr. Bernhard Möller

Tag der mündlichen Prüfung: 19. Dezember 2008

# Inhaltsverzeichnis

# Einleitung

Betrachten wir komplexe natürliche Systeme der wirklichen Welt, so stossen wir immer wieder auf zwei Eigenschaften: Solche Systeme arbeiten verteilt, also an mehreren Stellen gleichzeitig, und ihre Teilsysteme arbeiten jeweils in ihrer eigenen Zeit, also ohne ihre Geschwindigkeiten aufeinander abzustimmen. Eine Eigenschaft des Menschen wiederum scheint es zu sein, nach effizienter Gestaltung der Systeme, die ihn umgeben, zu streben. Zusammengenommen haben wir so bereits die drei zentralen Begriffe der vorliegenden Arbeit vor uns: Parallelität, Asynchronizität und Effizienz.

Erstaunlich mutet es an, dass vom Menschen entworfene Computersysteme den ersten beiden Begriffen traditionell eher wenig Rechnung tragen. Beim klassischen Typ eines Computers bearbeiten getaktete Prozessoren sequentielle Programme. Belohnt wurde die Informatik dafür mit mathematischen Begrifflichkeiten, mit denen die Effizienz solcher Systeme sehr gut beherrscht werden konnte.

Die theoretische Informatik betrachtet natürlich lange schon auch Modelle von Systemen, die aus asynchron arbeitenden parallelen Komponenten bestehen, aber nur langsam werden reale technische Systeme auf immer feinerer Granularität auch derart entworfen.

Die Betrachtung ihrer Effizienz nämlich stellt uns vor Probleme: Natürlich können im konkreten Fall Geschwindigkeiten gemessen und mit den Mitteln der Statistik teilweise auch aussagekräftig aufbereitet werden. Mit der Erhöhung des Abstraktionsgrades aber schwindet unsere Vorstellung davon, was Effizienz in solchen Systemen überhaupt bedeutet.

Während die Untersuchung sequentieller Algorithmen den *worst case* betrachten kann, verliert dieser Begriff im asynchronen Fall naiv betrachtet erst einmal seine Bedeutung: Da sich Komponenten beliebig Zeit lassen können, kann ein System beliebig langsam arbeiten. Erst ein durch [LF81] angeregtes erweitertes Verständnis von Asynchronizität bereitet hier den Weg zu sinnvollen Betrachtungen.

Als formales Modell unserer Systeme wählen wir in dieser Arbeit sogenannte Petri-Netze, eine graphische Sprache, welche gleichermaßen intuitiv wie auch exakten Beweisen zugänglich ist. Frühe Arbeiten haben bereits aufgezeigt, wie Zeitbetrachtungen in derart modellierten Systemen aussehen können: In [Ram74] und [MF76] wurde begonnen, mittels gedachter Stopuhren, die mit den Komponenten von Petri-Netzen assoziiert waren, zeitliche Abläufe zu erfassen. Die Art, wie dies genau geschieht, wurde in den folgenden Jahren in den verschiedensten Richtungen erweitert und variiert; [Bow00] etwa gibt einen Überblick über gebräuchliche Varianten.

Um nun die Effizienz paralleler zeitlicher Abläufe bewerten zu können, kombinieren [Vog95a, Vog95b, Vog95c] obige Ansätze mit Testing-Szenarios, wie sie in [DNH84] vorgestellt wurden, und erweitern diese um eine obere Zeitschranke. [JV95] zeigt dann ein Diskretisierungsresultat, um Abläufe aus kontinuierlichen Zeitschritten trotzdem diskret darstellen zu können. [Vog96] kombiniert wiederum die dortigen Ergebnisse mit einer vermutlich auf [Ric85] und [Dur85] zurückgehenden Erweiterung der Petri-Netz-Sprache zur Modellierung von Nebenbedingungen, den sogenannten Lesekanten oder auch *read-arcs*. In [BV98] haben wir den Prototypen eines Tools vorgestellt, das auf der Grundlage obiger Arbeiten vollautomatische Effizienzvergleiche durchführen kann. In [Bih98], [BV04] wurde, allerdings ausgehend von der Variante ohne Lesekanten, das Modell dahingehend erweitert,

dass Intervalle auf den Kanten von Netzgraphen eine feinere Kontrolle der zeitlichen Abläufe ermöglichen. Es tauchte damit erstmals die Frage auf, wie Abläufe einzuordnen sind, in denen auf Grund der Spezifikation keine Zeit vergeht oder nicht einmal vergehen kann. Weiter blieb zunächst die Frage offen, inwieweit das erweiterte Modell bezüglich der Ausdrucksmächtigkeit in der Lage ist, Lesekanten zu ersetzen.

Im Rahmen dieser Arbeit werden wir zunächst in Kapitel 1 ein verallgemeinertes Netzmodell vorstellen, welches sowohl Lesekanten als auch zeitliche Kontrolle über Kantenintervalle beinhaltet. Zusätzlich werden als weiteres Instrument der Zeitsteuerung noch sogenannte *lazy-arcs* eingeführt, die es lokal erlauben, eine Grundannahme der vorangegangenen Arbeiten außer Kraft zu setzen, nämlich dass jeder Verarbeitungsschritt in endlicher Zeit „dringend“ wird, also nicht mehr zurückgestellt werden kann. Während letztere Erweiterung, wie wir zeigen werden, die Ausdrucksmächtigkeit nicht erhöht, ist sie dennoch geeignet, die Zustandsmenge eines untersuchten Systems herabzusetzen.

Die zentralen Resultate der früheren Arbeiten werden nun hier auch für das erweiterte Modell hergeleitet. Bei der Behandlung zeitlicher Anomalien (d.h. verschieden „bösartiger“ Ausprägungen von Abläufen ohne Fortschritt der Zeit) muss man feststellen, dass die Einführung von Lesekanten die Situation gegenüber [Bih98] beträchtlich schwieriger werden lässt. Die auftretenden Anomalien werden hier nun feiner kategorisiert und anders als in der vorigen Arbeit systematisch aufeinander abgestützt. Es wird untersucht, welche Anomalien mit unserem Effizienzbegriff vereinbar sind und welche als Spezifikationsfehler betrachtet werden müssen. Die diesbezügliche Untersuchung endet mit einem statisch am Netzgraphen ablesbaren Kriterium für die Abwesenheit verbotener Anomalien, was angesichts der stark dynamisch orientierten Definitionen der Anomalien bemerkenswert ist.

In Kapitel 2 stellen wir darauf aufbauend einen theoretischen Ansatz vor, den Erreichbarkeitsgraphen unserer Systeme, welchen Tools wie FastAsy (s.u.) erzeugen müssen, beträchtlich zu verkleinern. Dabei setzen wir nicht wie andere Ansätze an der Zustandsmenge an, sondern machen uns die Beobachtung zu Nutze, dass die Übergänge gewissen Regelmäßigkeiten unterworfen sind. Wir werden nicht nur zeigen, wie man diese Redundanzen ausfaktorisieren kann, sondern auch, wie die Verfahren aus dem ersten Kapitel anzupassen sind, damit sie direkt auf dem verkleinerten Erreichbarkeitsgraphen arbeiten können. Für sämtliche Verfahren wird deren Korrektheit gezeigt.

Als letztes theoretisches Kapitel beschäftigt sich Kapitel 3 mit der noch aus früheren Arbeiten offenen Frage, inwiefern Lesekanten durch die erweiterte zeitliche Kontrolle durch Intervalle ersetzt werden können. Abhängig vom Grad der Äquivalenz (Simulation, RT-Gleichheit, Bisimulation) und der Erhaltung eventueller Unterklassen von Netzen kommen wir zu verschiedenen Transformationen, für die wir jeweils ihre spezifischen Eigenschaften zeigen, so dass die Frage umfassend beantwortet wird.

Der zweite große Teil der Arbeit widmet sich dem Unterfangen, ausgehend von dem in [BV98] vorgestellten Prototyp ein flexibles Tool zu bauen, mit welchem auf lange Sicht Neuerungen im theoretischen Modell abgebildet werden können. Durch diesen Anspruch erhielt das vorgestellte Softwareprojekt *FastAsy* mehr und mehr den Charakter eines Frameworks, so dass eine ausführliche Dokumentation nicht nur des implementierungs-

technischen Hintergrunds sondern vor allem auch der Entwurfsphilosphie angebracht erschien. Die Arbeit legt großen Wert darauf, dem Leser die eingeflossenen Entscheidungen von der Wahl der Werkzeuge über den Entwurf bis hin zu erwähnenswerten Einzelheiten der Umsetzung detailliert zugänglich zu machen. Die damit verbundene Hoffnung ist es, einerseits die Nachhaltigkeit des Tools zu stärken, indem spätere Weiterentwicklungen im Sinne des ursprünglichen Entwurfs stattfinden, und andererseits auch Impulse für andere wissenschaftliche Softwareprojekte zu geben.

So stellen wir nach einer grundsätzlichen Aufnahme der Anforderungen in Kapitel 4 und einer Übersicht über ältere Codebases von FastAsy in Kapitel 5 dann in Kapitel 6 die verwendeten Bibliotheken und Entwicklungstools vor. Von besonderem Interesse dürften hier die Ausführungen über die verschiedenen Teile der sehr umfassenden Bibliothek Boost (siehe [BCPL]) sein, welche das Verständnis der Sprache C++ in der Community in den letzten Jahren regelrecht revolutioniert und auch nachweisbar Einfluß auf die Weiterentwicklung der Sprache genommen hat. Wir gehen auch auf Werkzeuge zur Versionierung von Quelltext, zur maschinengestützten Dokumentation des Quelltexts und zur Visualisierung von Graphen ein.

Anschließend schlägt eine Schilderung der verwendeten Verfahren in Kapitel 7 die Brücke zwischen den theoretischen Ausführungen von Teil I und der Umsetzung als Computerprogramm. Das entsprechende Kapitel ist genügend in sich abgeschlossen, um von jemanden, der nur an der Anwendnung oder aber Weiterentwicklung von FastAsy interessiert ist, weitgehend auch ohne die zu Grunde liegenden theoretischen Ausführungen aus Teil I verstanden zu werden, ebenso wie das in Kapitel 8 folgende Anwenderhandbuch.

Mit Kapitel 9 widmet sich der Kernabschnitt des zweiten Teils dann verschiedenen Aspekten der Umsetzung aus softwaretechnischer Sicht. Dabei werden zunächst kurz einige moderne Methoden vorgestellt, die im Entwicklungsprozess zum Einsatz kamen. Dazu zählen Entwurfsmuster, Unit Testing, Refactoring und Laufzeit-Zusicherungen.

Anschließen geben wir einen Überblick über die Architektur von FastAsy, welche einem gelockerten Schichtenkonzept folgt. Mit der zentralen Schicht beschreiben wir dann den wahrscheinlich wichtigsten Beitrag der Architektur: Diese sogenannte *Bausteine*-Schicht sitzt zwischen den Fachfassaden, welche jeweils spezielle Typen von Anfragen des Benutzers aus der GUI in tiefere Schichten weiterleiten, und der Schicht der Quellsysteme, welche die Einzelschritte untersuchter Systeme generiert. Somit ist diese Schicht für das ganze Spektrum der fachlichen Algorithmen zuständig. Zu diesem Zweck unterteilt sich die Schicht in eine erweiterbare Menge von Bausteinen, welche jeweils verschiedene Verarbeitungsoperationen übernehmen. Die einzelnen Bausteine sind nun weder an feste Typen gebunden (was durch generische Programmierung erreicht wird) noch in ihrer genauen Reihenfolge festgelegt. Einige Bausteine sind darüber hinaus noch bezüglich ihrer genauen Funktionsweise parametrisierbar. Bemerkenswert ist die Art der Datenübergabe zwischen den Bausteinen, denn sie erfolgt nicht durch Weitergabe von fertigen Zwischenergebnissen, sondern durch eine Art *pull*-Verfahren, bei dem jeder Baustein seinen Nachbarn nur nach genau der Information fragt, die er zur Beantwortung einer Anfrage, die er gerade selbst vorliegen hat, benötigt. Durch Kombination mit speziellen Bausteinen, die sofort den Ergebnisraum ihres Nachbarn vollständig untersuchen und zwischenspeichern, ergibt sich eine sehr feingranulare Kontrolle über die Zwischenergebnisse, die im Speicher materialisiert werden und die, die *on-*

*the-fly* berechnet werden. Damit ist insbesondere auch ein gezielter Trade-Off zwischen Zeit- und Speicherbedarf möglich.

Es folgen, eine Ebene detailierter als die Architektur, ausführliche Betrachtungen des Entwurfs der verschiedenen Packages von FastAsy. Diese folgen jeweils einem einheitlichen dreiteiligen Schema: Erstens werden die spezifischen Anforderungen an das Package aufgeführt. In einem zweiten Schritt wird auf dieser Grundlage ein Klassenverbund für das Package ausgearbeitet und es werden die Anforderungen in Form von Verantwortlichkeiten auf die Klassen verteilt. Im dritten Schritt werden die Ergebnisse als Klassendiagramm dargestellt, zudem dokumentieren ausgewählte Sequenzdiagramme die Kollaborations-mechanismen zwischen den Klassen.

Als Beleg für die weitreichende Flexibilität unseres Entwurfs wird in Kapitel 10 in einem kurzen „Developers' Tutorial" unserem Tool ein Modul hinzugefügt, welches das Einlesen von Ausgaben des PAFAS-Tools [PFS] und damit das Verarbeiten von Systemen erlaubt, welche als Prozessalgebren statt als Petri-Netze formuliert sind.

Einige Ausblicke und Gedanken zur Weiterentwicklung von FastAsy in Kapitel 11 runden den zweiten Teil der Arbeit ab.

# Teil I Theoretische Überlegungen

# Kapitel 1 Das Netzmodell und grundlegende Resultate

## *1.1 Petri-Netze*

Wir definieren zunächst einen grundlegenden Typ von Petri-Netzen, welcher noch keine Information über zeitliches Verhalten trägt und normalerweise auch ohne Berücksichtigung von zeitlichem Verhalten interpretiert wird:

Ein *Petri-Netz* N ist ein Tupel $(S,T,F,M_N)$ und besteht aus einer Menge T von *Transitionen*, einer Menge S von *Stellen*, einer *Flussrelation* $F \subseteq S\times T \cup T\times S$, so dass S, T, F einen gerichteten, bipartiten Netzgraph $(S\cup T, F)$ definieren, und einer *Anfangsmarkierung* $M_N: S\to\mathbb{N}_0$. Kantengewichte werden in dieser Arbeit keine Rolle spielen. (Man lasse sich nicht davon beirren, dass im Teil II in 8.1.2 formale Kantengewichte verwendet werden, um in einem Dateiformat Informationen über den Kantentypus zu übertragen.)

Die Zustände eines Petri-Netzes beschreiben *Markierungen* $M: S\to\mathbb{N}_0$. Entsprechend ist die Anfangsmarkierung diejenige, mit der wir das System starten.

In der graphischen Repräsentation eines Petri-Netzes stellen wir Transitionen als Rechtecke und Stellen als Kreise dar. Marken werden im Inneren der Stellen und meist als Punkte notiert:

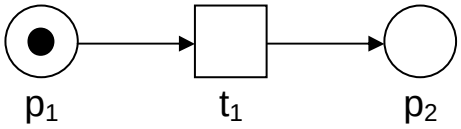



**Abbildung 1: Ein Petri-Netz**

Weiter ist zu einer Transition t der *Vorbereich* ${}^\bullet t := \{s\in S \mid (s,t)\in F\}$ und der *Nachbereich* $t^\bullet := \{s\in S \mid (t,s)\in F\}$ definiert, analog der Vor- und Nachbereich einer Stelle s.

Ist eine Transition unter einer Markierung M *aktiviert* (d.h. $\forall s\in {}^\bullet t: M(s)>0$), so kann sie *schalten* (geschrieben als $M[t\rangle M'$). Sei der *Effekt* $\Delta t(s)$ definiert durch:
$\Delta t(s) = -1$ für $s\in {}^\bullet t\setminus t^\bullet$, $\Delta t(s) = 1$ für $s\in t^\bullet\setminus {}^\bullet t$ und $\Delta t(s) = 0$ sonst. Dann gilt $M' := M + \Delta t$.

Wir nennen eine Markierung M *erreichbar*, falls es ein $n\in\mathbb{N}_0$ und für $i=1,...,n$ Transitionen $t_i$ und Markierungen $M_i$ gibt, so dass gilt

$$M_N\,[t_1\rangle\, M_2\,[t_2\rangle\, M_1 \ldots [t_n\rangle\, M_n \text{ und } M = M_n.$$

Wir können den Begriff des Schaltens nun auf ganze Sequenzen von Transitionen erweitern, indem wir für Transitionen $t_i$ und Markierungen $M_i$ statt

$$M_1\,[t_1\rangle\, M_2\,[t_2\rangle\, M_1 \ldots [t_n\rangle\, M_n$$

schreiben

$$M_1\,[t_1\, t2 \ldots t_n\rangle\, M_n.$$

Ist dabei $M_1 = M_N$, so nennen wir $w := t_1t_2 \ldots t_n$ eine *Schaltfolge* von N. Wir schreiben für die Menge aller Schaltfolgen (*firing sequences*) von N auch FS(N).

Notiert man die Schaltfolgen von N als gerichteten Graphen mit Markierungen M als Knoten und mit durch t beschrifteten Kanten von M nach M' für Übergänge $M[t\rangle M'$, so erhält man den *Erreichbarkeitsgraphen* (siehe Beispiel nach 1.3.1.3). Wir werden diesen Begriff allerdings später etwas allgemeiner verwenden, da wir auch abweichende Verhaltens-beschreibungen – etwa mit Zeit versehene – dergestalt notieren wollen.

Sei Σ nun ein endliches Alphabet von Zeichen, sogenannten *Aktionen*. Ein *beschriftetes Petri-Netz* $(S,T,F,M_N,l)$ besteht dann aus einem Petri-Netz $(S,T,F,M_N)$ und einer *Transitionsbeschriftung* $l:T\to\Sigma\cup\{\lambda\}$, wobei die mit dem leeren Wort λ beschrifteten Transitionen *unsichtbare* oder auch *interne* Transitionen heißen. Wenn wir *l* in natürlicher Weise auf Sequenzen $w\in T^*$ erweitern, erhalten wir für Schaltfolgen w sogenannte *Aktionsschaltfolgen* $l(w)\in\Sigma^*$. Wir schreiben $M[l(w)\rangle\rangle M'$, wenn $M[w\rangle M'$. Man beachte, dass darin die internen Transitionen nicht mehr zu sehen sind, da die λs bei der Konkatenation verschwinden. Wir nennen $L(N) := \{l(w) \mid w\in FS(N)\}$ die *Sprache* von N.

Wie man sofort sieht, ist die Menge der erreichbaren Markierungen im Allgemeinen unendlich. Es gibt einen Typ von Petri-Netzen, der dies verbietet, indem für Stellen sogenannte *Kapazitäten* festgelegt werden und Transitionen nur noch schalten können, wenn sie diese nicht verletzen. Wir gehen hier jedoch einen anderen Weg und verlangen von zu untersuchenden Petri-Netzen, dass ihre Abläufe selbst es gar nicht zulassen, dass beliebig viele Marken auf einer Stelle zu liegen kommen. Anders ausgedrückt existiert für jedes Petri-Netz eine Oberschranke n, so dass unter keiner erreichbaren Markierung auf einer Stelle mehr als n Marken liegen. Man spricht von *beschränkten* Petri-Netzen, wenn solch ein n für jedes Petri-Netz existiert und von *n-beschränkten* Petri-Netzen, wenn dieses n universal ist. Jedoch unterscheidet sich die Ausdrucksmächtigkeit für verschiedene n *nicht*, so dass wir uns in weiten Teilen auf *1-beschränkte* (sog. *sichere*) Petri-Netze beschränken werden. In sicheren Netzen notieren wir Markierungen statt als Funktionen auch einfach als die markierten Teilmengen von S, d.h. $s\in M :\Leftrightarrow \{s\in S \mid M(s)=1\}$.

Wir werden im Weiteren oft von *Netzen* statt von Petri-Netzen sprechen, da im Verlauf der Arbeit keine Verwechslungsgefahr besteht. Wir verwenden, dem gängigen Sprachgebrauch folgend, beide Begriffe auch dann, wenn wir die technischen Einzelheiten der Definition variieren. Insbesondere kommen durch Einführung von Lesekanten und Zeit noch weitere Freiheitsgrade in der Definition hinzu. Weiter sprechen wir von *Netzklasse*, um genau einen solchen in allen Einzelheiten definierten Typus zu bezeichnen. Obwohl dies oft der Fall ist, meinen wir mit dem Begriff *Netzklasse* nicht, dass für zwei gegebene Netzklassen automatisch eine gemeinsame Oberklasse existiert – bestimmte Erweiterungen können durchaus unvereinbar sein.

## *1.2 Lesekanten*

Die Netzklassen, die wir in dieser Arbeit behandeln, werden zumeist mit sogenannten *Lesekanten* (oder *Read Arcs*) versehen sein. Dabei handelt es sich um einen weiteren Kantentypus, der eine Stelle und eine Transition verbindet. Die Vorstellung hierbei ist, dass für das Schalten der Transition zwar eine Marke auf der Stelle vorhanden sein muss, diese jedoch im Verlauf des Schaltens nicht entfernt wird. Betrachtet man das Verhalten des Netzes allein durch Schaltfolgen, so hat eine Lesekante freilich dieselben Auswirkungen wie eine *Schlinge* (d.h. zwei entgegengesetzte Kanten zwischen Stelle und Transition). Bezieht man sich in seinen Betrachtungen jedoch auf Nebenläufigkeit, Fairness oder bestimmte Zeitbegriffe, so spielen die Lesekanten auf einmal eine wichtige Rolle. In der Tat zeigt z.B. [Vog96], dass eine bestimmte Formulierung des MUTEX-Problems mit Petri-Netzen ohne Lesekanten nicht lösbar ist, wohl aber mit ihnen. Die Idee, Nebenbedingungen mit besonders

ausgezeichneten Kanten zu modellieren, ist indes schon älter und geht unseres Wissens auf [Ric85] und [Dur85] zurück.

Lesekanten notieren wir wie im folgenden Beispiel mit einer Kante ohne Pfeile. (Die Kante mit doppelten Pfeilenden ist lediglich eine abkürzende Schreibweise für zwei entgegengesetzte Kanten.)

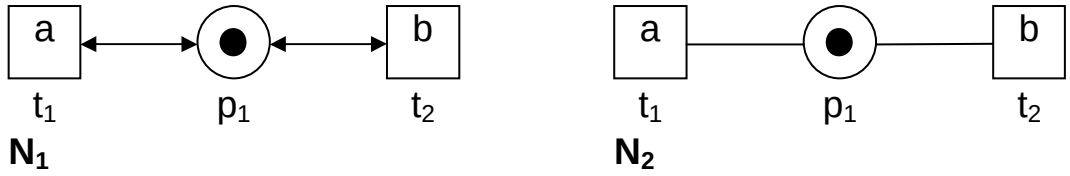



**Abbildung 2: Netze ohne und mit Lesekanten**

Betrachtet man im Beispiel lediglich Schaltfolgen, erhält man für beide Netze:

$$( t_1 + t_2 )^*$$

Betrachtet man die Mengen von Transitionen, die gemeinsam nebenläufig schalten können, so kann man sich vorstellen, dass in $N_2$ $\{t_1,t_2\}$ schalten kann, wohingegen dies in $N_1$ nicht möglich ist, da nur eine Marke vorhanden ist und jede der Transitionen diese wegnehmen müsste, um schalten zu können.

Ähnlich verhält es sich, wenn man sog. *schwache Fairness* zugrunde legt (d.h. eine Aktion muss irgendwann ausgeführt werden, wenn sie ab einem gewissen Zeitpunkt immer möglich ist): Während der Ablauf $t_1^{\omega}$ in $N_1$ die schwache Fairness erfüllt (schließlich wird $t_2$ von jedem $t_1$ immer wieder deaktiviert), ist dies in $N_2$ nicht der Fall, da $t_2$ aktiviert bleibt.

Ohne späteren Betrachtungen vorgreifen zu wollen, kann man sich hier schon vorstellen, dass es auch bei der Betrachtung von zeitlicher Effizienz eine Rolle spielen wird, ob etwa $t_2$ immer wieder deaktiviert wird, oder „lange Zeit“ aktiviert ist.

Die Modellierung von Lesekanten kann folgendermaßen geschehen:

#### *1.2.1.1 Definition*

Ein Petri-Netz mit Lesekanten N ist ein Tupel $(S,T,F,R,M_N)$. Dabei ist $(S,T,F,M_N)$ ein Petri-Netz und $R \subset S \times T$ die *Lesemenge*, wobei gilt $R \cap F = \varnothing$ und $R^{-1} \cap F = \varnothing$. Für $t \in T$ nennen wir $t^{\wedge} := \{s \in S \mid (s,t) \in R\}$ den *Lesebereich* von t.

Wir nennen weiter ${}^{\blacksquare}t := \{s \in S \mid (s,t) \in R \cup F\}$ den *erweiterten Vorbereich* und $t^{\blacksquare} := \{s \in S \mid (t,s) \in R^{-1} \cup F\}$ den *erweiterten Nachbereich*.

Eine Transition t mit inzidenten Lesekanten kann nur schalten, falls zusätzlich zu $\forall s \in {}^{\bullet}t: M(s) > 0$ auch gilt $\forall s \in t^{\wedge}: M(s) > 0$, mit den obigen Begriffen gilt also generell:

$$M[t\rangle M' \quad \Leftrightarrow \quad \forall s \in {}^{\blacksquare}t: M(s) > 0 \quad \wedge \quad \forall s \in S: M'(s) = M(s) + \Delta t(s)$$

(Der *Effekt* einer Transition $\Delta t(s)$ ändert sich durch Einführung von Lesekanten nicht.)

□

## *1.3 Transitionssysteme und Automaten*

Bevor wir uns in Abschnitt 1.4 damit beschäftigen, wie wir Zeit in Petri-Netze einbringen können, möchten wir kurz die Notation zu Transitionssystemen und Automaten klären, da uns

diese im folgenden als Werkzeug bei weiteren Betrachtungen dienen werden. Grundlagen zu diesen beiden Konzepten müssen im Rahmen dieser Arbeit allerdings vorausgesetzt werden.

### *1.3.1.1 Definition*

Ein *beschriftetes Transitionssystem mit Anfangszustand* (im folgenden auch kurz *Transitionssystem* genannt) ist ein Tupel $(Q, \Lambda, \delta, q_0)$, wobei Q eine Menge an Zuständen, $\Lambda$ ein endliches Aktionsalphabet, $\delta \subseteq Q \times (\Lambda \cup \{\lambda\}) \times Q$ die Menge der *Zustandsübergänge* und $q_0$ der Startzustand ist.

□

Bemerkung: Die wenig übliche Benennung des Alphabets von Transitionssystemen mit $\Lambda$ statt $\Sigma$ rührt daher, dass wir unter $\Sigma$ ja schon das Aktionsalphabet unserer Petri-Netze verstehen und beide (wie wir etwa in 1.4.6 sehen werden), zwar nicht unabhängig voneinander, aber trotzdem verschieden sind.

### *1.3.1.2 Definition*

Ein *endlicher Automat* (im folgenden auch kurz *Automat* genannt) ist ein Tupel $(Q, \Lambda, \delta, q_0, Q_F)$, wobei Q eine endliche Menge an Zuständen, $\Lambda$ ein endliches Aktionsalphabet, $\delta \subseteq Q \times (\Lambda \cup \{\lambda\}) \times Q$ die (unmittelbare) Zustandsübergangsrelation, $q_0$ der Startzustand und $Q_F \subseteq Q$ eine Menge von Endzuständen.

□

In dieser Arbeit wird für vorkommende Automaten ausnahmslos gelten, dass $Q_F = Q$, so dass wir $Q_F$ meist nicht gesondert angeben werden. Insofern ist ein endlicher Automat für uns einfach ein Transitionssystem mit endlicher Zustandsmenge und endlich vielen Übergängen. Des weiteren lassen sich beide Konzepte natürlich auch als gerichtete Graphen auffassen.

### *1.3.1.3 Definition*

Ein Automat bzw. Transitionssystem heißt:

- *buchstabierend*, wenn $\nexists (q, \lambda, q') \in \delta$
- *deterministisch*, wenn er buchstabierend ist und $\nexists (q, a, q') \in \delta, (q, a, q'') \in \delta$ mit $q' \neq q''$.

Für $(q, a, q') \in \delta$ schreiben wir auch $q \xrightarrow{a} q'$. Für $q, q' \in Q$ und $w \in \Lambda^*$ schreiben wir $q \overset{w}{\twoheadrightarrow} q'$, wenn $n \in \mathbb{N}$ existiert und $a_1,\ldots,a_n \in \Lambda \cup \{\lambda\}$, $q_1,\ldots,q_{n+1} \in Q$ mit $w = a_1 \ldots a_n$ und:

$$q = q_1 \xrightarrow{a1} q_2 \ \ldots \ q_n \xrightarrow{an} q_{n+1} = q' .$$

Weiter schreiben wir auch $\delta(q, a)$ bzw. $\delta(q)$ und meinen damit $\{ q' \mid (q, a, q') \in \delta \}$ bzw. $\{ (q', a) \mid (q, a, q') \in \delta \}$.

□

Man bemerke, dass $\lambda$-Übergänge in w nicht mehr sichtbar sind und deswegen für $a \in \Lambda$ sich $\xrightarrow{a}$ und $\overset{a}{\twoheadrightarrow}$ in ihrer Bedeutung unterscheiden. Man bemerke weiter, dass $n \in \mathbb{N}_0$ und deshalb immer $q \overset{\lambda}{\twoheadrightarrow} q$ gilt, selbst wenn $(q, \lambda, q) \notin \delta$ .

Ein Beispiel für Transitionssysteme stellt der *beschriftete Erreichbarkeitsgraph* eines über $\Sigma$ beschrifteten Petri-Netzes $(S,T,F,M_N)$ dar, und zwar vermöge:

- $Q := \{ M \mid \exists w \in T^*: M_N[w\rangle M \}$

- $\Lambda := \Sigma$
- $(M,a,M') \in \delta :\Leftrightarrow \exists t \in T: M[t\rangle M' \wedge a = l(t)$
- $q_0 = M_N$ .

Ist das Netz sicher, so handelt es sich gleichzeitig um einen endlichen Automaten, da die Anzahl der Zustände durch $2^{|S|}$ nach oben beschränkt ist und die Anzahl der Übergänge durch $2^{|S|} * 2^{|S|} * |l(T)|$ .

Es sei wie schon oben nochmals darauf hingewiesen, dass wir in dieser Arbeit nicht allgemein $\Lambda = \Sigma$ annehmen können.

Mit Sprache und Sprachgleichheit zusammenhängende Begriffe für die Automaten und Transitionssysteme seien wie üblich definiert. Zusätzlich benötigen wir aber noch die Begriffe Simulation und Bisimulation:

### *1.3.1.4 Definiton*

Seien $TS^1$, $TS^2$ zwei Transitionssysteme. Eine Relation $S \subseteq Q^1 \times Q^2$ heißt (*schwache*) *Simulation*, wenn:

1) $(q_0^1, q_0^2) \in S$ und
2) $\forall (q^1, q^2) \in S, a \in \Lambda \cup \{\lambda\}$ *mit* $q^1 \overset{a}{\twoheadrightarrow} q^1'$ gilt:
   $\exists q^2' \in Q^2: q^2 \overset{a}{\twoheadrightarrow} q^2' \wedge (q^1', q^2') \in S$ .

Wir sagen, $TS^2$ kann $TS^1$ (schwach) simulieren, wenn eine solche Simulation existiert. Wir nennen S eine *starke Simulation*, wenn es eine schwache Simulation zwischen $TS^1_{[\tau \text{ for } \lambda]}$ und $TS^2_{[\tau \text{ for } \lambda]}$ ist. (Dabei bedeutet [τ for λ] einfach, dass alle Vorkommen von λ durch ein neues Zeichen τ ausgetauscht werden und somit unsichtbare Übergänge sichtbar werden.)

□

Offenbar fallen für buchstabierende Transitionssysteme starke und schwache Simulation zusammen. Außerdem können wir in diesem Fall im Punkt 2 der Definition beide Male $\overset{a}{\twoheadrightarrow}$ äquivalent dazu durch $\overset{a}{\rightarrow}$ ersetzen. Allgemeiner gilt jedoch:

### *1.3.1.5 Satz*

Seien $TS^1$, $TS^2$ zwei Transitionssysteme. Eine Relation $S \subseteq Q^1 \times Q^2$ ist genau dann eine (*schwache*) *Simulation*, wenn:

1) $(q_0^1, q_0^2) \in S$ und
2) $\forall (q^1, q^2) \in S, a \in \Lambda \cup \{\lambda\}$ *mit* $q^1 \overset{a}{\rightarrow} q^1'$ gilt:
   $\exists q^2' \in Q^2: q^2 \overset{a}{\twoheadrightarrow} q^2' \wedge (q^1', q^2') \in S$ .

**Beweis:**

Wenn S eine Simulation ist, gelten offensichtlich 1) und 2).

Sei andersherum S eine Relation, für die 1) und 2) gilt. Es seien weiter $(q^1, q^2) \in S$ und $a \in \Lambda \cup \{\lambda\}$ mit $q^1 \overset{a}{\twoheadrightarrow} q^1'$ . Dann existieren nach Definition 1.3.1.3 von $\twoheadrightarrow$ entsprechende Einzelschritte. Anwendung von 2) auf alle Einzelschritte liefert schließlich den rechten Teil von 2) in 1.3.1.4.

□

Immer wenn wir daran interessiert sind, die Existenz einer Simulation zu zeigen, werden wir bevorzugt diese äquivalente Charakterisierung verwenden, da dort der Nachweis von 2) leichter zu führen ist.

Wir werden den Begriff der Simulation im Teil II in Abschnitt 7.3 benötigen, wo wir folgenden Satz aus der Automatentheorie verwenden:

### *1.3.1.6 Satz*

Seien $A_1$, $A_2$ Automaten und $A_2$ deterministisch. Dann sind äquivalent:

1) Die Sprache von $A_1$ ist in der von $A_2$ enthalten.
2) $A_2$ kann $A_1$ simulieren.

□

Ohne die Voraussetzung, dass $A_2$ deterministisch ist, gilt übrigens immer noch 2) $\Rightarrow$ 1), aber nicht mehr die Umkehrung. Die folgenden zwei Definitionen sagen uns zusammen mit den anschließenden Sätzen jedoch, dass wir uns zu jedem Automaten immer einen sprachgleichen deterministischen Automaten verschaffen können.

### *1.3.1.7 Definition*

Für einen Automaten $A = (Q, \Lambda, \delta, q_0, F)$ sei der *Potenzautomat* $PA := \wp(A)$ gegeben durch $(Q_{PA}, \Lambda, \delta_{PA}, \{q_0\}, F_{PA})$, wobei $Q_{PA} := \wp(Q)$, $F_{PA} := \{ P \in \wp(Q) \mid P \cap F \neq \emptyset \}$ und $\delta_{PA}$ für ein $P \in Q_{PA}$ definiert ist durch:

$$\delta_{PA}(P) := \{ (x, \cup_{p \in P} \delta(p, x)) \mid x \in \Lambda \cup \{\lambda\}, \exists p \in P, p' \in Q: (p, x, p') \in \delta \}$$

(Man bedenke, dass für eine Anwendung innerhalb eines Tools natürlich die von $\{q_0\}$ aus erreichbare Teilmenge von $\wp(Q)$ ausreicht.)

□

Bemerkung: Im Unterschied zu ebenfalls gängigen alternativen Definitionen des Potenzautomaten ist unsere Version evtl. nicht *vollständig*, da keine Übergänge der Form $(P, x, \emptyset)$ eingefügt werden.

### *1.3.1.8 Satz*

Sei A ein buchstabierender Automat und PA := $\wp$(A) der Potenzautomat.
Dann ist *PA* deterministisch und $L(A) = L(PA)$.

**Beweis:**

bekannt.

□

Man bemerkt, dass die Konstruktion des Potenzautomaten nur für buchstabierende Automaten sinnvoll ist: Selbst wenn man aus dem Potenzautomaten eines nicht-buchstabierenden Automaten die λ-Übergänge (wie im folgenden definiert) eliminiert, lässt ihn das i.A. nicht-deterministisch zurück.

### *1.3.1.9 Definition*

Für einen Automaten $A = (Q, \Lambda, \delta_A, q_0, F)$ sei seine *Externalisierung* $EA := Ext(A)$ gegeben durch $(Q, \Lambda, \delta_{EA}, q_0, F_{EA})$, wobei für $q, q' \in Q$ und $a \in \Lambda \cup \{\lambda\}$:

$$(q, a, q') \in \delta_{EA}(q, a) \;:\Leftrightarrow\; q \overset{a}{\rightarrow\!\!\!\!\rightarrow}_A q' \;\wedge\; a \neq \lambda \quad \text{und}$$

$$F_{EA} = F \cup \{\, q \in Q \mid \exists q' \in F: q \overset{\lambda}{\twoheadrightarrow}_A q' \,\}\,.$$

□

### *1.3.1.10 Satz*

Sei A ein Automat und EA := Ext(A) seine Externalisierung. Dann ist EA buchstabierend und *L(A) = L(EA)*.

**Beweis;**

bekannt.

□

Zusammen mit 1.3.1.6 können wir also für zwei Automaten $A_1$ und $A_2$ die Sprachinklusion $L(A_1) \subseteq L(A_2)$ entscheiden, indem wir versuchen, $A_1$ mit $\wp(Ext(A_2))$ zu simulieren. (Oder alternativ $\wp(Ext(A_1))$ mit $\wp(Ext(A_2))$, wenn sowieso beide Potenzautomaten vorliegen, weil man z.B. beide Richtungen der Inklusion entscheiden will.)

Nach der Bemerkung nach 1.3.1.6 ist gegenseitige Simulation ein hinreichendes Kriterium für Sprachgleichheit. Mit der Bisimulation existiert aber noch ein wesentlich stärkerer Begriff für Verhaltensgleichheit:

### *1.3.1.11 Definition*

Seien $TS^1$, $TS^2$ zwei Transitionssysteme. Eine Relation $B \subseteq Q^1 \times Q^2$ heißt (*schwache*) *Bisimulation*, wenn:

1) $(q_0^1, q_0^2) \in B$ ,
2) $\forall (q^1, q^2) \in B, a \in \Lambda \cup \{\lambda\}$ *mit* $q^1 \overset{a}{\twoheadrightarrow} q^1'$ gilt:
   $\exists q^2' \in Q^2: q^2 \overset{a}{\twoheadrightarrow} q^2' \wedge (q^1', q^2') \in B$ und
3) $\forall (q^1, q^2) \in B, a \in \Lambda \cup \{\lambda\}$ *mit* $q^2 \overset{a}{\twoheadrightarrow} q^2'$ gilt:
   $\exists q^1' \in Q^1: q^1 \overset{a}{\twoheadrightarrow} q^1' \wedge (q^1', q^2') \in B$ .

Wir sagen, $TS^2$ und $TS^1$ sind (schwach) bisimular, wenn eine solche Bisimulation existiert.

Wir nennen B eine *starke Bisimulation*, wenn es eine schwache Bisimulation zwischen $TS^1_{[\tau \text{ for } \lambda]}$ und $TS^2_{[\tau \text{ for } \lambda]}$ ist.

□

Bisimulation verlangt also, dass an jeder Stelle eines gemeinsamen Ablaufs eines der beiden Systeme die Führung übernehmen darf und das andere alle Aktionen nachbilden können muss.

Auch hier fallen für buchstabierende Transitionssysteme wieder die starke und schwache Version zusammen und wir können in den Punkten 2 und 3 der Definition $\overset{a}{\twoheadrightarrow}$ durch $\overset{a}{\rightarrow}$ ersetzen. Ebenso gilt:

### *1.3.1.12 Satz*

Seien $TS^1$, $TS^2$ zwei Transitionssysteme. Eine Relation $B \subseteq Q^1 \times Q^2$ ist genau dann eine (*schwache*) *Bisimulation*, wenn:

1) $(q_0^1, q_0^2) \in B$ ,

2) $\forall (q^1, q^2) \in B, a \in \Lambda \cup \{\lambda\}$ *mit* $q^1 \xrightarrow{a} q^{1'}$ gilt:
$\exists q^{2'} \in Q^2: q^2 \overset{a}{\twoheadrightarrow} q^{2'} \wedge (q^{1'}, q^{2'}) \in B$ und

3) $\forall (q^1, q^2) \in B, a \in \Lambda \cup \{\lambda\}$ *mit* $q^2 \xrightarrow{a} q^{2'}$ gilt:
$\exists q^{1'} \in Q^1: q^1 \overset{a}{\twoheadrightarrow} q^{1'} \wedge (q^{1'}, q^{2'}) \in B$ .

**Beweis:**

wie 1.3.1.5.

□

Spricht man bei Petri-Netzen von Bisimularität, so meint man üblicherweise die Bisimularität ihrer beschrifteten Erreichbarkeitsgraphen. Wir wollen aber den Begriff des Erreichbarkeitsgraphen für jede auf einer Schaltregel basierenden Semantik in der offensichtlichen Weise verallgemeinert verstehen und damit auch den Begriff der Bisimulation. Wir werden an den jeweiligen Stellen angeben, auf welche Semantik sich die Bisimulation bezieht.

Wir geben nun noch einige bekannte Aussagen über Simulationen und Bisimulationen an, die wir im Verlauf dieser Arbeit benötigen werden:

### *1.3.1.13 Satz*

Seien $TS^1$, $TS^2$ zwei Transitionssysteme und die Relation $B \subseteq Q^1 \times Q^2$ eine (schwache) Bisimulation. Dann ist $B^{-1}$ ebenfalls eine (schwache) Bisimulation und B und $B^{-1}$ sind (schwache) Simulationen.

□

### *1.3.1.14 Satz*

Seien $TS^1$, $TS^2$, $TS^3$ Transitionssysteme und die Relationen $S_1 \subseteq Q^1 \times Q^2$ und $S_2 \subseteq Q^2 \times Q^3$ (schwache) Simulationen. Dann ist $S := S_1 \circ S_2$ eine (schwache) Simulation.

□

### *1.3.1.15 Korollar*

Seien $TS^1$, $TS^2$, $TS^3$ Transitionssysteme und die Relationen $B_1 \subseteq Q^1 \times Q^2$ und $B_2 \subseteq Q^2 \times Q^3$ (schwache) Bisimulationen. Dann ist $B := B_1 \circ B_2$ eine (schwache) Bisimulation.

□

### *1.3.1.16 Korollar*

Seien $TS^1$, $TS^2$, $TS^3$ Transitionssysteme, die Relation $B \subseteq Q^1 \times Q^2$ eine (schwache) Bisimulation und $S \subseteq Q^2 \times Q^3$ eine (schwache) Simulation. Dann ist $B \circ S$ eine (schwache) Simulation.

□

### *1.3.1.17 Korollar*

Seien $TS^1$, $TS^2$, $TS^3$ Transitionssysteme, die Relation $S \subseteq Q^1 \times Q^2$ eine (schwache) Simulation und $B \subseteq Q^2 \times Q^3$ eine (schwache) Bisimulation. Dann ist $S \circ B$ eine (schwache) Simulation.

□

## *1.4 PTT-Netze*

### 1.4.1 Zeit in Petri-Netzen

Um Netze „mit Zeit zu versehen“ hat man verschiedene Alternativen zu erwägen:

- Welche Vorgänge brauchen Zeit (Aktivierung, Schalten)?
- Welche Entitäten zählen Zeit (Stellen, Transitionen, Marken)?
- Wird eine Dauer durch einen festen Wert, eine Obergrenze oder ein Intervall angegeben?
- Was geschieht nach Ablauf einer Frist?
- Was sind die Wertebereiche für Zeitangaben?

(Für eine systematische Zusammenstellung gebräuchlicher Kombinationsmöglichkeiten siehe wie erwähnt [Bow00].)

Beim ersten Punkt sprechen wir davon, dass die *Aktivierung* Zeit braucht, wenn nach Ankunft der Marken Zeit vergeht, bis eine Transition aktiviert ist (die Marken bleiben also während dieser Zeit erhalten). Vergeht hingegen während des *Schaltens* Zeit, so werden die Marken von der Transition entfernt, und erst nach einer bestimmten Zeitspanne erscheinen Marken im Nachbereich. Man sieht bereits an der Existenz eines derartigen Zwischenzustands, dass es technisch aufwändiger ist, Zeit während des Schaltens vergehen zu lassen. Das Verhältnis der beiden Ansätze wurde schon in [JV95] eingehend untersucht. Wir entscheiden uns im folgenden für die Variante, in der nur während der Aktivierung Zeit vergeht, das Schalten jedoch ein Punktereignis darstellt.

Würden nun z.B. die Transitionen die Zeit zählen, so verginge nach Eintreffen der letzten Marke im Vorbereich eine gewisse, üblicherweise durch eine Annotation der Transition angegebene Zeit und anschließend würde die Transition schalten. Der Zeitpunkt des Eintreffens früherer Marken wäre also irrelevant. Wir entscheiden uns für einen flexibleren Ansatz: Jede Marke soll eine eigene Uhr besitzen, die zum Zeitpunkt der Entstehung der Marke zu zählen anfängt. Diese Uhren vergleicht man mit Intervallen an den Kanten des Netzes, die von Stellen zu Transitionen führen. Eine Marke kann zum Schalten der Transition beitragen, wenn ihre Uhr die untere Intervallgrenze überschritten hat. In der Formalisierung für sichere Netze werden wir ausnutzen, dass auf einer Stelle nur eine einzige Marke liegen kann und wir damit die Uhr auch an die Stelle anheften können. Dies hat den Vorteil, dass die Menge der Uhren für jedes Netz feststeht. Für unmarkierte Stellen werden wir die Uhr auf $-\infty$ setzen, was weitere technische Vorteile mit sich bringt (s.u.).

Es gibt nun Ansätze, in denen die Marke nach Überschreiten der oberen Intervallgrenze nutzlos wird (sog. *weak timing*) und solche, die von Abläufen verlangen, dass dies nicht geschieht (*strong timing*). Die erste Interpretation wird beispielsweise im Multimedia-Bereich eingesetzt, wo etwa Teile des Multimedia-Stroms unbrauchbar werden, wenn sie zu spät erscheinen (siehe z.B. [SDS94]). Unsere Vorstellung entspricht dagegen dem letzteren Ansatz, wobei wir sehen werden, dass wir obige Aussage ändern zu „…dass dies nicht *ohne Grund* geschieht.“ Erstens soll nämlich ein System, welches über synchrone Kommunikation mit dem Netz verbunden wird, in der Lage sein, Transitionen zu verzögern. Zweitens aber soll eine *schnelle* Marke durchaus auf eine langsamere auf einer anderen Stelle des Vorbereichs warten dürfen. Wir verlangen also nicht, dass sich beim Schalten alle Uhren beteiligter

Marken innerhalb ihres Intervalls befinden. Lediglich die letzte Marke, die ihre obere Intervallschranke erreicht, darf diese nicht mehr überschreiten.

Letztlich spielen auch die Wertebereiche der verschiedenen Zeitangaben eine Rolle. Wir halten diesbezüglich zunächst fest, dass unsere intuitive Vorstellung von Zeit einfach eine kontinuierliche ist. Eine Einschränkung auf diskrete Zeit scheint im Allgemeinen nicht überzeugend. Auch wir gehen deshalb von kontinuierlicher Zeit aus. Wir werden aber die theoretische Rechtfertigung dafür sehen, mit diskreter Zeit zu rechnen und damit trotzdem Aussagen über das kontinuierliche System zu machen. An einer anderen Stelle jedoch werden wir uns wirklich einschränken müssen: Die zeitlichen *Grenzen*, welche wir als Teil der Systemspezifikation angeben, dürfen tatsächlich nicht kontinuierlich gewählt werden, damit alle Beweise funktionieren. Da Petri-Netze ohne Zeit jedoch auch mit einer endlichen Spezifikation auskommen müssen, scheint diese Beschränkung nicht unnatürlich, zumal man sich ja auf Grund der endlichen Anzahl von Zeitgrenzen jeder reellwertigen Spezifikation mit beliebiger Genauigkeit annähern könnte: $\mathbb{Q}$ liegt dicht in $\mathbb{R}$, anschließend skaliert man geeignet und kann sich also sogar auf Grenzen aus $\mathbb{N}_0$ einschränken, was wir im Weiteren tun wollen.

## 1.4.2 Formalisierung

Wir wollen die oben skizzierte Vorstellung nun formalisieren. Gegenüber der in [Bih98] und [BV04] vorgestellten Netzklasse wurden die im folgenden definierten Netze um zwei Aspekte erweitert: Zum einen wurden die aus anderen Varianten bekannten Lesekanten miteinbezogen, und zum anderen erlauben wir die Erweiterung des Wertebereichs der oberen Intervallgrenze ins Unendliche. (Weitergehende Überlegungen zu letzterem finden sich in 1.4.6.)

Die Ergebnisse sind zumeist einfache Verallgemeinerungen der Aussagen aus [Bih98] und [BV04], so dass wir Beweise oft nur verknappt angeben oder (wo die Übertragung wörtlich ist) weglassen werden. Gravierende Unterschiede finden sich dann allerdings ab Abschnitt 1.4.8.

### *1.4.2.1 Definition*

Ein *PTT* (*place-transition arc timed*) *-Netz* N ist ein Tupel $(S,T,F,R,M_N,l,lb,ub)$ und besteht aus einem sicheren, beschrifteten Petri-Netz mit Lesekanten $(S,T,F,R,M_N,l)$ und zusätzlich den Funktionen $lb: (F \cup R) \cap (S \times T) \to \mathbb{N}_0$ und $ub: (F \cup R) \cap (S \times T) \to \mathbb{N}_0 \cup \{\infty\}$mit $lb \leq ub$. Diese Funktionen geben für jede Kante (und Lesekante) von einer Stelle zu einer Transition die beiden Intervallgrenzen an.

Eine Kante (s,t) nennen wir *zero-arc,* falls $ub(s,t) = 0$, wir nennen sie *lazy-arc,* falls $ub(s,t) = \infty$.

Ein PTT-Netz mit lb = 0 nennen wir *asynchron*.

□

Tatsächlich entsprechen Netze mit lb = 0 auch unserem intuitiven Verständnis von Asynchronizität, wenn wir es (zurückgehend auf [LF81]) geeignet formulieren: Ein asynchrones System ist dadurch charakterisiert, dass die Verhältnisse der Geschwindigkeiten der einzelnen Komponenten nicht festgelegt sind. Dies trifft sowohl dann zu, wenn Aktionen

der Komponenten, wie gebräuchlich, beliebig *lang* dauern dürfen, als auch wenn sie, wie in unserem Fall, beliebig *schnell* sein dürfen.

Da für jede endliche Familie von Netzen $N_i$ auch die Menge der Intervallgrenzen

$$\bigcup_i ( lb_i((F_i \cup R_i) \cap (S_i \times T_i)) \cup ub_i((F_i \cup R_i) \cap (S_i \times T_i)) )$$

endlich ist, stellt die Beschränkung auf $\mathbb{N}_0$ statt $\mathbb{Q}_0^+$ bei den Intervallgrenzen auch für Vergleiche zwischen endlich vielen Netzen noch keine Einschränkung dar.

### *1.4.2.2 Definition*

Eine mit kontinuierlicher Zeit behaftete Markierung *CID* (*continuously-timed instantaneous description*) eines Netzes ist eine Funktion $\rho: S \to \mathbb{R}_0^+ \cup \{-\infty\}$.

Es besteht keine technische Notwendigkeit, zusätzlich zur CID noch die Markierung festzuhalten, denn aus einer CID ρ ergibt sich die korrespondierende gewöhnliche Markierung $M_\rho$ wie folgt:

$$M_\rho := \{ s \in S \mid \rho(s) \neq -\infty \}.$$

Die Anfangs-CID $CID_N$ (oder auch $\rho_N$) eines Netzes ergibt sich aus der Anfangsmarkierung $M_N$ als:

$$CID_N(s) := \begin{cases} 0 & \text{für } s \in M_N \\ -\infty & \text{sonst.} \end{cases}$$

□

In der grafischen Repräsentation geben wir lb und ub an, indem wir die Kanten mit dem entsprechenden Intervall [lb,ub] beschriften. Intervalle [0,1] werden wir in der Regel nicht explizit darstellen, da sie als Standardverhalten gelten dürfen in dem Sinne, dass sie in früheren Arbeiten die einzige und implizit angenommene Möglichkeit waren.

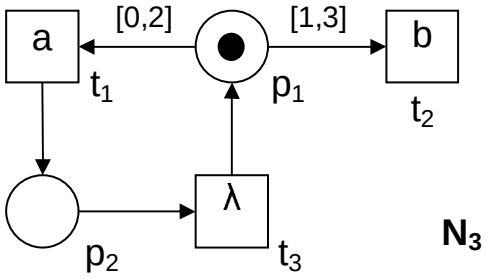



**Abbildung 3: Ein PTT-Netz**

Alle Begriffe normaler beschrifteter Petri-Netze übertragen sich auf ein PTT-Netz *(S,T,F,$M_N$,l,lb,ub)*, indem wir nur *(S,T,F,$M_N$,l)* betrachten.

### *1.4.2.3 Definition*

Wir sagen, ein $t \in T$ ist unter einer CID ρ *zeit-aktiviert*, falls gilt

$$\forall s \in {}^\bullet t : \rho(s) \geq lb(s,t)$$

Weiter sagen wir, $t \in T$ ist unter ρ *dringend*, falls

$$\forall s \in {}^\bullet t : \rho(s) \geq ub(s,t)$$

Wir schreiben

$$E(\rho) := \{ t \in T \mid \forall s \in {}^\bullet t : \rho(s) \geq lb(s,t) \}$$

für die Menge der unter ρ zeit-aktivierten und

$$U(\rho) := \{ t \in T \mid \forall s \in {}^\bullet t : \rho(s) \geq ub(s,t) \}$$

für die Menge der unter ρ dringenden Transitionen.

Eine Aktion a ist *zeit-aktiviert* bzw. *dringend*, wenn eine Transition t mit $l(t) = a$ existiert, die zeit-aktiviert bzw. dringend ist.

Außerdem seien definiert:

- $Res_{lb}(\rho,t) := max \{ lb(s,t) - \rho(s) \mid s \in {}^\bullet t \} \cup \{-\infty\}$
  (die restliche Zeit bis zur Zeit-Aktivierung von t. Ein negativer Wert ist dabei ausdrücklich zulässig und bedeutet, dass t bereits aktiviert ist.)
- $Res_{ub}(\rho,t) := max \{ ub(s,t) - \rho(s) \mid s \in {}^\bullet t \} \cup \{-\infty\}$
  (die restliche Zeit, bis t dringend wird)
- $Res_{lb}(\rho) := min \{ Res_{lb}(\rho,t) \mid t \in T \wedge \exists s \in {}^\bullet t : \rho(s) < lb(s,t) \} \cup \{\infty\}$
  (die restliche Zeit bis zur *nächsten* Zeit-Aktivierung)
- $Res_{ub}(\rho) := min \{ Res_{ub}(\rho,t) \mid t \in T \wedge \exists s \in {}^\bullet t : \rho(s) < ub(s,t) \} \cup \{\infty\}$
  (die restliche Zeit, bis die *nächste* Transition dringend wird)

(Dabei wird $-\infty$ aus formalen Gründen benötigt, um den, wie wir in 1.4.9 sehen werden, pathologischen Fall ${}^\bullet t = \emptyset$ abzufangen und $\infty$ für den Fall, dass keine Transitionen mehr allein durch Vergehen von Zeit neu zeit-aktiviert bzw. dringend werden können.)

□

### *1.4.2.4 Lemma*

Eine Transition $t \in T$ ist für eine CID ρ unter $M_\rho$ aktiviert, wenn sie unter ρ zeit-aktiviert ist.

□

Die Umkehrung ist jedoch nicht unbedingt der Fall: In obigem Beispiel sind $t_1$ und $t_2$ von Beginn an aktiviert, jedoch ist nur $t_1$ zeit-aktiviert. In asynchronen PTT-Netzen allerdings fallen die Begriffe aktiviert und zeit-aktiviert tatsächlich zusammen.

Da der ursprüngliche Begriff der Aktiviertheit für PTT-Netze kaum sinnvolle Information enthält (d.h. der Zustand, in dem eine Transition aktiviert, aber nicht zeit-aktiviert ist, aus Sicht einer intuitiven Interpretation nicht von dem zu unterscheiden ist, in dem die Transition gar nicht aktiviert ist), sagen wir auch, eine Transition t ist *aktiviert unter einer CID ρ,* wenn sie zeit-aktiviert unter ρ ist.

Den Begriff des *Schaltens* erweitern wir wie folgt auf zeitbehaftete Markierungen, wobei wir nun statt Transitionen auch *Zeitschritte* schalten lassen können:

### *1.4.2.5 Definition*

Für CIDs ρ, ρ' und eine unter ρ zeit-aktivierte Transition t schreiben wir $\rho [t\rangle_c \rho'$, wenn gilt:

$\rho'(s) = -\infty \qquad \forall s \in {}^\bullet t \setminus t^\bullet$
$\rho'(s) = 0 \qquad \forall s \in t^\bullet$
$\rho'(s) = \rho(s) \qquad sonst$

(Man beachte, dass Lesekanten durch den letzten Fall abgedeckt sind.)
Für CIDs ρ, ρ' und einen Zeitschritt $\varepsilon \in \mathbb{R}^+$ schreiben wir $\rho [\varepsilon\rangle_c \rho'$, wenn gilt:

$\rho' = \rho + \varepsilon \qquad und$
$\forall t \in T\ \exists s \in {}^\blacksquare t: \rho'(s) \leq ub(s,t)$

(Man sieht hier, warum es geschickt ist, unmarkierten Stellen in der Formalisierung den Uhrenwert -∞ zuzuweisen, denn es erspart eine Fallunterscheidung beim Vergehen der Zeitspanne ε. Man beachte auch, dass ein Zeitschritt nur erlaubt ist, wenn alle Uhrenwerte „korrekt behandelt" werden, d.h. wenn keine Transition dringend ist.)

Wir ziehen diese Schritte zunächst wieder auf die Ebene der Sequenzen (*continuous firing sequences*)

$CFS(N) := \{w \mid CID_N[w\rangle_c\}$

und dann auf die Sprache (*continuous language*)

$CL(N) = \{v \mid CID_N[v\rangle_c\} := \{l(w) \mid CID_N[w\rangle_c\}$

hoch. (Hierzu sei *l* als die Identität auf $\mathbb{R}$ erweitert.)
Wir nennen $\{\rho \mid CID_N[w\rangle_c \rho\}$ die Menge der *c-erreichbaren* CIDs.
Wir schreiben $\rho[\rangle_c$ für $\{w \mid \rho[w\rangle_c\}$ und $\rho[\rangle\rangle_c$ für $\{w \mid \rho[w\rangle\rangle_c\}$.

Für $w \in CFS(N)$ bezeichne α(w) die Projektion auf die Transitionen und ζ(w) die Summe der Zeitschritte, entsprechend seien für $l(w) \in CL(N)$ definiert $\alpha(l(w)) := l(\alpha(w))$ und $\zeta(l(w)) := \zeta(w)$.

□

### *1.4.2.6 Lemma*

Für CIDs ρ und ρ', eine Transition $t \in T$ und einen Zeitschritt $\varepsilon \in \mathbb{R}^+$ mit $\rho [\varepsilon\rangle_c \rho'$ gilt:

*t ist unter ρ aktiviert* $\Rightarrow$ *t ist unter ρ' aktiviert.*

Außerdem gilt $M_\rho = M_{\rho'}$ .

**Beweis**
Klar mit der Definition von $\rho [\varepsilon\rangle_c \rho'$ und 1.4.2.3.

□

### *1.4.2.7 Lemma*

Sei $w \in CFS(N)$, dann ist α(w) in FS(N).

**Beweis**
Es gilt für CIDs ρ, ρ', $t \in T$ offensichtlich immer $\rho[t\rangle_c\rho' \Rightarrow M_\rho[t\rangle M_{\rho'}$. Zusammen mit 1.4.2.6 folgt die Behauptung.

□

### 1.4.3 Modulare Konstruktion

Es existieren mehrere Konzepte, aus gegebenen Netzen modular komplexere Systeme aufzubauen, so z.B. Stellenverschmelzung, hierarchische Netze oder Synchronisation bestimmter Aktionen. Wir beschränken uns in dieser Arbeit auf letzteren Ansatz. Er erscheint im Kontext von PTT-Netzen besonders geeignet, da sich, wie wir später sehen werden, das zeitliche Verhalten eines zusammengesetzten Systems auf elegante Art ermitteln lässt.

Die angesprochene *Parallelkomposition* für PTT-Netze folgt der üblichen Konstruktion für Petri-Netze, bei der eine Menge *synchronisierter Aktionen* angegeben wird. Die Vorstellung dabei ist, dass die Zusammensetzung zweier Systeme eine solche Aktion ausführt, indem jedes einzelne System die Aktion ausführt. Man beachte dabei aber, dass die Beschriftung von Transition ja nicht injektiv sein muss und folglich jedes System im Allgemeinen mehrere Möglichkeiten zur Durchführung einer Aktion hat. Folgerichtig existieren im Gesamtsystem kombinatorisch viele Möglichkeiten für eine synchronisierte Aktion. Der Vollständigkeit halber sei noch erwähnt, dass Aktionen, die nicht (explizit) synchronisiert werden, nach wie vor unter voller Kontrolle der Teilsysteme stehen. Insofern betrachten wir es also sozusagen als Zufall, wenn in verschiedenen Teilsystemen Aktionen gleichen Namens vorkommen.

Es stellt sich als nächstes natürlich die Frage nach dem zeitlichen Verhalten synchronisierter Aktionen. Um diese Frage zu beantworten, begeben wir uns aber auf die Ebene der Transitionen: Jede Möglichkeit des Gesamtsystems, eine Aktion auszuführen, besteht (s.u.) aus einer Transition, die aus der Verschmelzung zweier Transitionen hervorgeht, welche wiederum eine jeweilige Möglichkeit der einzelnen Teilsysteme darstellen, die Aktion auszuführen. Aus unserer Vorstellung gemeinsamer Abläufe geht nun hervor, dass die verschmolzene Transition *aktiviert* sein soll, wenn dies beide Einzeltransitionen sind. Es liegt daher nahe festzulegen, dass die verschmolzene Transition auch dann *dringend* sein soll, wenn dies für die einzelnen Transitionen der Fall ist. Mit anderen Worten ist das Gesamtsystem gezwungen, auf den langsamsten Synchronisationspartner zu warten, weil es nicht beschleunigend in den zeitlichen Ablauf des Partners eingreifen kann – für die allermeisten technischen Systeme eine realistische Annahme.

Damit formalisieren wir die Parallelkomposition für PTT-Netze wie folgt:

#### *1.4.3.1 Definition*

Seien $N_i = (S_i,T_i,F_i,R_i,M_{Ni},l_i,lb_i,ub_i)$ für $i \in \{1,2\}$ zwei PTT-Netze, sei $A \subseteq \Sigma$ die *Synchronisationsmenge*, d.h. die Menge der synchronisierten Aktionen. Dann ist die *Parallelkomposition* $N = N_1 \,||_A\, N_2$ dieser Netze definiert als:

$$
\begin{aligned}
S := \quad & \{ (s_1,*) \mid s_1 \in S_1 \} \cup \{ (*,s_2) \mid s_2 \in S_2 \} \\
T := \quad & \{ (t_1,*) \mid t_1 \in T_1 \wedge l_1(t_1) \notin A \} \cup \\
& \{ (*,t_2) \mid t_2 \in T_2 \wedge l_2(t_2) \notin A \} \cup \\
& \{ (t_1,t_2) \mid t_1 \in T_1 \wedge t_2 \in T_2 \wedge l_1(t_1) = l_2(t_2) \in A \} \\
F := \quad & \{ ((s_1,*),(t_1,t_2)) \mid (s_1,t_1) \in F_1 , (t_1,t_2) \in T \} \cup \\
& \{ ((*,s_2),(t_1,t_2)) \mid (s_2,t_2) \in F_2 , (t_1,t_2) \in T \} \cup \\
& \{ ((t_1,t_2),(s_1,*)) \mid (t_1,s_1) \in F_1 , (t_1,t_2) \in T \} \cup \\
& \{ ((t_1,t_2),(*,s_2)) \mid (t_2,s_2) \in F_2 , (t_1,t_2) \in T \}
\end{aligned}
$$

$$
\begin{array}{lll}
R := & \{ ((s_1,*),(t_1,t_2)) \mid (s_1,t_1) \in R_1 , (t_1,t_2) \in T \} \cup & \\
 & \{ ((*,s_2),(t_1,t_2)) \mid (s_2,t_2) \in R_2 , (t_1,t_2) \in T \} & \\
l(t_1,t_2) := & l_1(t_1) \quad \textit{falls } t_1 \in T_1 , & \\
 & l_2(t_2) \quad \textit{sonst} & \\
M_N := & M_{N1} \times \{*\} \cup \{*\} \times M_{N2} & \\
lb((s_1,s_2),(t_1,t_2)) := & lb_1(s_1,t_1) & \textit{falls } s_1 \in S_1 \\
 & lb_2(s_2,t_2) & \textit{sonst} \\
ub((s_1,s_2),(t_1,t_2)) := & ub_1(s_1,t_1) & \textit{falls } s_1 \in S_1 \\
 & ub_2(s_2,t_2) & \textit{sonst}
\end{array}
$$

□

Man beachte, dass mit $(t_1,t_2) \in T$ für $t_i$ auch * möglich ist. Weiter sind die Funktionen *lb* und *ub* genau dann undefiniert, wenn die rechte Seite undefiniert ist.

Außerdem sei noch darauf hingewiesen, dass laut dieser Definition eine Aktion $a \in A$, die zwar in $l(T_1)$, aber nicht in $l(T_2)$ ist, niemals schalten kann. Ein fehlender Synchronisationspartner befreit nämlich eine Transition *nicht* von der Synchronisationspflicht.

## 1.4.4 Zeitliche Effizienz

Mit Hilfe der Parallelkomposition sind wir nun in der Lage, einen Effizienzbegriff für PTT-Netze zu formulieren. Wir betrachten dazu nämlich nicht Abläufe des untersuchten Systems, sondern die Frage, welche Performance ein System aus der Sicht eines Clients hat. Die Menge der Clients wird dabei wiederum durch PTT-Netze beschrieben.

Die Idee, Systeme durch ihre Kommunkation mit weiteren Systeme untersuchen zu lassen, ist als *Testing* bekannt geworden und geht unseres Wissens auf [DNH84] zurück und wurde auch schon im Kontext zeitlicher Effizienz verwendet, etwa in [Vog95c], [Vog96], [Bih98]. Wir stellen den Ansatz hier noch einmal speziell für PTT-Netze vor.

### *1.4.4.1 Definition*

Es sei ω eine spezielle, reservierte Aktion, welche den Erfolg eines Tests anzeigt. Ein PTT-Netz bezeichnen wir daher als *testbar*, wenn gilt $\omega \notin l(T)$.

Ein *c-Zeit-Test* ist ein Paar (O,D), wobei O ein *Testnetz* und $D \in \mathbb{R}_0^+$ die *Testdauer* ist. Ein testbares Netz N *c-erfüllt* den Test (O,D), wenn nach Ablauf von Zeit D in jedem Fall ein Erfolg eintritt, d.h.:

$$\forall w \in CL( N \mid\mid_{\Sigma-\{\omega\}} O ): \quad \zeta(w) > D \Rightarrow \omega \in w .$$

Wir schreiben dafür auch $N\ must_c\ (O,D)$.

Weiter nennen wir für testbare PTT-Netze $N_1$, $N_2$ das Netz $N_1$ die *c-schnellere Implementierung*, wenn für alle Tests (O,D) gilt, dass $N_2\ must_c\ (O,D) \Rightarrow N_1\ must_c\ (O,D)$. In diesem Fall schreiben wir auch $N_1 \sqsupseteq_c N_2$.

□

Man beachte, dass für $(O,D_1)$ und $(O,D_2)$ mit $D_1 \le D_2$ gilt:

$$N\ c\text{-erfüllt}\ (O,D_1) \ \Rightarrow\ N\ c\text{-erfüllt}\ (O,D_2).$$

Der Begriff *c-schnellere Implementierung* ist also insofern präzise, als $N_1$ gegenüber $N_2$ sowohl solche Tests mit anderem Testnetz O wie auch solche mit kleinerer Zeitschranke D

zusätzlich bestehen kann. Trotzdem werden wir später oft mit derselben Bedeutung nur noch vom *c-schnelleren* Netz sprechen, da der Vergleich zeitlicher Effizienz unser Hauptaugenmerk darstellt.

Ein technisches Hindernis an dieser Stelle besteht nun in der Tatsache, dass wir es durch die Modellierung reeller Zeitschritte mit einer unüberschaubaren Menge von Sequenzen in CFS(N) zu tun haben. Andererseits ist unser Verständnis von Zeit inhärent kontinuierlich. Glücklicherweise existiert eine theoretische Rechtfertigung, ohne Beschränkung der Allgemeinheit mit diskreter Zeit zu arbeiten (wie auch für ähnliche Ansätze, z.B. bei [Pop91], wo allerdings beliebig gerundet werden kann; dahingegen müssen wir für den Umgang mit Tests die Dauer einer Sequenz immer *verlängern*): Wir stellen fest, dass wenn wir den Rahmen des Testens wie im folgenden für diskrete Zeit definieren, die Begriffe der schnelleren Implementierung für den diskreten und den kontinuierlichen Fall zusammenfallen.

### *1.4.4.2 Definition*

Für CIDs ρ, ρ' und eine zeit-aktivierte Transition t gelte

$$\rho[t\rangle\rho' :\Leftrightarrow \quad \rho[t\rangle_c\rho'$$

Für CIDs ρ, ρ' und einen Zeitschritt $\varepsilon\in\mathbb{R}^+$ gelte

$$\rho[\varepsilon\rangle\rho' :\Leftrightarrow \quad \rho[\varepsilon\rangle_c\rho' \wedge \varepsilon\in\mathbb{N}$$

Wir ziehen wie oben diese Schritte auf Sequenzen (*discrete firing sequences*)

$$DFS(N) := \{w \mid CID_N[w\rangle\}$$

und die Sprache (*discrete language*)

$$DL(N) := \{l(w) \mid CID_N[w\rangle\rangle\}$$

hoch.

Wir nennen $\{\rho \mid CID_N[w\rangle\rho\}$ die Menge der d-*erreichbaren* CIDs. Da für alle d-erreichbaren CIDs ρ offensichtlich gilt, dass $\rho(S) \subseteq \mathbb{N}_0$, bezeichnen wir diese auch mit *ID* und die Anfangs-CID mit $ID_N$.

Damit sind α(w) und ζ(w) auch für $w\in$ DFS(N) bzw. $w\in$ DL(N) definiert. Auch die Begriffe aus 1.4.4.1 übertragen sich analog: Ein *(d-)Zeit-Test* (oder im folgenden einfach *Test*) ist ein c-Zeit-Test mit $D\in\mathbb{N}_0$, ein testbares Netz *(d-)erfüllt* einen solchen Test (*N must (O,D) )*, wenn $\forall w\in DL(\, N \,||_{\Sigma-\{\omega\}}\, O\,): \zeta(w) > D \Rightarrow \omega\in w$.

Wir nennen das Netz $N_1$ die *(d-)schnellere Implementierung*, wenn für alle Tests (O,D) gilt, dass $N_2$ *must (O,D)* $\Rightarrow N_1$ *must (O,D)* und schreiben dafür auch $N_1 \sqsupseteq N_2$.

□

### *1.4.4.3 Lemma*

In einem Netz N existiert für jedes $w\in$ CFS(N) ein $v\in$ DFS(N), so dass gilt

$$\alpha(v) = \alpha(w) \text{ und } \zeta(v) - \zeta(w) \in [0, 1[\ .$$

**Beweis**

Per Induktion über |w|. Dazu seien $\rho_v$ und $\rho_w$ CIDs, so dass $CID_N[v\rangle_c\rho_v$ und $CID_N[w\rangle_c\rho_w$. (Damit ist $\rho_v$ natürlich auch eine ID und es gilt $\rho_N[v\rangle\rho_v$.)

Es sei $d_{vw} := \zeta(v) - \zeta(w)$, also die Zeit, um die wir in v aufgerundet haben und $M_v := \{ s \in S \mid \rho_v(s) \neq -\infty \}$, analog sei $M_w$ definiert. Wir verschärfen mit diesen Notationen die Induktionsbehauptung zu:

1. $\alpha(v) = \alpha(w)$
2. $d_{vw} \in [0, 1[$
3. $M_v = M_w$
4. $\forall s \in M_v : \rho_w(s) - \rho_v(s) + d_{vw} \in [0, 1[$.
   (Für $s \notin M_v$ gilt offensichtlich $\rho_w(s) = \rho_v(s) = -\infty$.)

Für den Induktionsschritt sei $w = w'x$. Die Induktionsvoraussetzung verschafft uns ein v' zu w' mit den obigen Eigenschaften. Ist $x \in \Sigma$, so sei $x' = x$. Ist $x \in \mathbb{R}^+$, so sei $x' = \lceil x - d_{v'w'} \rceil$. Wir setzen $v := v'x'$ für $x'>0$ und $v := v'$ sonst. Für beide Fälle prüft man nun, dass einerseits x' nach v' schalten kann und andererseits w und v zusammen die Behauptungen wieder erfüllen.

□

### *1.4.4.4 Satz*

Die Relationen $\sqsupseteq_c$ und $\sqsupseteq$ fallen zusammen.

**Beweis**

„$\sqsupseteq_c \subseteq \sqsupseteq$“:

Seien $N_1 \sqsupseteq_c N_2$ Netze und (O,D) ein d-Zeit-Test mit $\neg N_1$ *must (O,D)*. Damit ist (O,D) natürlich auch ein c-Zeit-Test und $\neg N_1$ *must$_c$ (O,D)*, folglich $\neg N_2$ *must$_c$ (O,D)*, genauer $\exists w \in CL( N ||_{\Sigma-\{\omega\}} O ): \zeta(w) > D \wedge \omega \notin w$. Vermöge des zu w gehörigen v aus 1.4.4.3 gilt damit auch $\neg N_2$ *must (O,D)*.

„$\sqsupseteq_c \supseteq \sqsupseteq$“:

Seien $N_1 \sqsupseteq N_2$ Netze und (O,D) ein c-Zeit-Test mit $\neg N_1$ *must$_c$ (O,D)*. (O,D) ist nicht unbedingt ein d-Zeit-Test, so dass wir zunächst $D' := \lfloor D \rfloor$ setzen. Wie oben gilt mit 1.4.4.3 dann auch $\neg N_1$ *must (O,D')* und weiter nach Voraussetzung $\neg N_2$ *must (O,D')* vermöge eines $v \in DL(N_2)$ mit $\zeta(v) > D'$. Da $\zeta(v) \in \mathbb{N}_0$, gilt aber auch $\zeta(v) \geq D'+1 > D$ und somit $\neg N_2$ *must$_c$ (O,D)*.

□

Da die in DL bzw. DFS vorkommenden Zeitschritte alle aus $\mathbb{N}_0$ sind, verwenden wir auch eine alternative Notation, bei der wir alle Zeitschritte n mit $n \geq 1$ als $\sigma^n$ schreiben. Wir nennen in einem Ablauf die Teilsequenz vor dem ersten bzw. zwischen je zwei $\sigma$'s eine *Runde*. Betrachten wir nochmal 1.4.2.5, so sehen wir, dass wir genau dann in die nächste Runde voranschreiten dürfen, wenn keine Transition dringend ist.

Betrachten wir nun kurz informell zwei unterschiedliche Situationen: Im ersten Fall ist in einem Netz $N_1$ eine interne Transition $t_1$ dringend. Gemäß unserer Synchronisationsregeln existiert keine Parallelkomposition mit $N_1$, in welcher $t_1$ *nicht* dringend ist. Egal, in welche Umgebung $N_1$ eingebettet wird, wir werden an diesem Punkt immer $(t_1,*)$ schalten oder deaktivieren müssen, bevor der Zeitschritt $\sigma$ erfolgt.

Haben wir jedoch in einem Netz $N_2$ ein mit einer Aktion a beschriftetes ebenfalls dringendes $t_2$, so können wir $N_2$ über die Synchronisationsmenge {a} mit einer Umgebung

synchronisieren, für welche a *nicht* dringend ist. Somit ist in der Parallelkomposition jedes $(t_2,t)$ ebenfalls nicht dringend. Damit kann in Abläufen der Parallelkomposition ein Zeitschritt $\sigma$ stehen, wo dies in der Projektion auf $N_2$ nicht möglich wäre. Es existieren also Abläufe, deren Projektionen wir in den Abläufen von $N_2$ gar nicht sehen können.

Mit anderen Worten werden wir, wenn wir die Kompositionalität der zeitlichen Semantik anstreben, DL noch verfeinern müssen. Damit wird dann auch eine Charakterisierung der $\sqsupseteq$-Relation möglich, die wesentlich besser handzuhaben ist als die Definition über *alle* Tests.

## 1.4.5 Charakterisierung zeitlicher Effizienz

Führen wir obigen Gedanken weiter, so müssen wir in Abläufen eines zu untersuchenden Netzes N an den Stellen, an denen in Synchronisation mit einer beliebigen Umgebung ein Zeitschritt stehen *könnte*, dies vermerken. Dabei müssen wir angeben, welche Aktionen in N eigentlich dringend sind und nur über den Synchronisationspartner verzögert werden können, und welche schon in N nicht dringend sind. Wir halten nochmal fest, dass dringende interne Transitionen in jeder Umgebung dringend bleiben; über sie müssen wir also gar nichts vermerken.

Ist eine Transition (bzw. Aktion) unter einer ID nicht dringend, so sagen wir auch, sie darf *verweigert* werden. Dieser Begriff stammt daher, dass im klassischen Testen (nach [DNH84]) analog eine *failure semantics* genannte Charakterisierung Verwendung findet, welche am Ende jeden Ablaufs zusätzlich vermerkt, welche Aktionen (im herkömmlichen Sinne, in diesem Kontext wird ja eben nicht über Zeit geredet) verweigert werden können.

Wir erlauben nun also statt Zeitschritten $\sigma$ alle *potentiellen Zeitschritte* und notieren diese als sogenannte Verweigerungsmengen $X\subseteq\Sigma$. Dabei stellen die vollen Verweigerungsmengen $\Sigma$ genau die alten Zeitschritte $\sigma$ dar, so dass die im folgenden definierten Semantiken Verfeinerungen von DFS bzw. DL darstellen.

### *1.4.5.1 Definition*

Seien N ein Netz und $\rho$ und $\rho$' IDs von N. Für eine Transition $t\in T$ schreiben wir $\rho[t\rangle_r \rho'$, wenn gilt $\rho[t\rangle\ \rho'$. Für eine Menge $X\subseteq\Sigma$ schreiben wir $\rho[X\rangle_r\ \rho'$, wenn gilt:

$$\forall t\in T\ (\ \forall s\in {}^\bullet t\colon \rho(s) \geq ub(s,t)\ ) \Rightarrow l(t)\notin X \cup \{\lambda\} \quad \textit{und}$$
$$\rho' = \rho + 1\ .$$

Die entsprechenden Schaltfolgen bilden die Menge *RFS(N)* der *Verweigerungs-schaltfolgen* (*refusal firing sequences*). Ersetzen wir in *RFS(N)* alle Transitionen *t* durch ihre Aktionsbeschriftungen *l(t)*, so erhalten wir die Menge *RT(N)* der *Verweigerungsabläufe* oder *refusal traces*.

Sei $ll\colon T\cup\wp(\Sigma) \to \Sigma\cup\{\lambda\}\cup\wp\{\Sigma\}$ dadurch definiert, dass wir *l* auf $\wp(\Sigma\}$ durch die Identität fortsetzen. Dann schreiben wir analog zu weiter oben $\rho[ll(w)\ \rangle\rangle_r\ \rho'$, wenn $\rho[w\rangle_r\ \rho'$ gilt. Weiter sei $\alpha$ auch auf RFS (bzw. RT) als Projektion auf die Transitionen (bzw. Aktionen) und $\zeta$ als die Anzahl der Verweigerungsmengen in einem Wort definiert.

Wir sagen, $\rho$ sei *r-erreichbar* (oder *erreichbar*, wenn klar ist, dass wir uns auf Verweigerungsabläufe beziehen), wenn $w\in RFS(N)$ existiert, so dass $\rho_N[w\rangle_r\ \rho$.

□

### *1.4.5.2 Lemma*

Sei N ein Netz, $w \in (T \cup \{\sigma\})^*$ und v entstehe aus w, indem man alle σ durch Σ austauscht. Dann gilt: $w \in DFS(N) \Leftrightarrow v \in RFS(N)$.

□

Halten wir noch einige weitere Eigenschaften der RT-Semantik fest:

### *1.4.5.3 Lemma*

Für Netze N, $N_1$, $N_2$ gilt:

- $RT(N_1) \subseteq RT(N_2) \Rightarrow DL(N_1) \subseteq DL(N_2)$
- $\forall X \subseteq \Sigma, Y \subseteq X: vXw \in RT(N) \Rightarrow vYw \in RT(N)$
- $\forall X, Z \subseteq \Sigma, Z \cap l(T) = \emptyset: vXw \in RT(N) \Rightarrow v(X \cup Z)w \in RT(N)$ .

□

Stellen wir uns aber nun der bereits weiter oben angedeuteten Frage, ob die RT-Semantik *kompositional* ist, d.h. ob wir allein aus den RT-Semantiken zweier Netze auf die RT-Semantik einer Parallelkomposition dieser Netze schließen können. Dazu definieren wir zunächst eine Familie von Operatoren || auf Wörtern und zeigen anschließend, dass diese, angewandt auf die Wörter der jeweiligen RT-Semantiken, genau das gewünschte leisten.

### *1.4.5.4 Definiton*

Seien $u, v, w \in (\Sigma \cup \wp(\Sigma))^*$ und $A \subseteq \Sigma$. Wir schreiben $w = u \,||_A\, v$, wenn ein n existiert mit $u = u_1 \ldots u_n$, $v = v_1 \ldots v_n$ und $w = w_1 \ldots w_n$, so dass für jedes i=1…n einer der folgenden Fälle gilt:

1. $u_i = v_i = w_i \in A$
2. $u_i = w_i \in (\Sigma - A) \wedge v_i = \lambda$
3. $v_i = w_i \in (\Sigma - A) \wedge u_i = \lambda$
4. $u_i, v_i, w_i \subseteq \Sigma \wedge w_i \subseteq ( (u_i \cup v_i) \cap A ) \cup (u_i \cap v_i)$ .

□

Blicken wir nochmals auf die Kompositionsvorschrift in 1.4.3.1: Wie man sieht, stammt jede Stelle der Komposition jeweils aus genau einem der Teilnetze. Damit können wir jede ID des Gesamtsystems aufspalten in zwei IDs der Einzelsysteme. Es liegt nun nahe, sich zu fragen, ob für jedes RT-Wort des Gesamtsystems Abläufe in den isolierten Teilnetzen existieren, so dass die IDs nach jedem Schritt derart korrespondieren. Die zentrale Beobachtung ist nun, dass einerseits eine nicht synchronisierte Transition genau dann im Einzelsystem zeit-aktiviert bzw. dringend ist, wenn sie es in der Komposition ist und andererseits eine durch Synchronisation verschmolzene Transition genau dann zeit-aktiviert bzw. dringend ist, wenn es die beiden Einzeltransitionen in den isolierten Netzen sind. Durch Induktion und geeignete Fallunterscheidung wird dann etwa in [BV04] gezeigt, dass 1.4.5.4 genau die Projektion eines Ablaufs des Gesamtsystems auf die Einzelsysteme beschreibt, und dass umgekehrt auf diese Weise aus zwei „passenden“ Einzelabläufen auch ein Ablauf der Komposition gewonnen werden kann, formal:

### *1.4.5.5 Satz*

Für Netze $N_1$, $N_2$ und eine Synchronisationsmenge $A \subseteq \Sigma$ gilt:

$$RT( N_1 ||_A N_2 ) = \bigcup \{ u ||_A v \mid u \in RT(N_1) \wedge v \in RT(N_2) \} .$$

□

Der Beweis erfolgt in [Bih98] in beiden Richtungen zunächst über Fallunterscheidungen und dann über Induktion. Man vergewissert sich schnell, dass dies nach der Einführung von Lesekanten und *lazy-arcs* noch völlig analog funktioniert.

Eine unmittelbare Folgerung aus dem Satz ist:

### *1.4.5.6 Korollar*

Für Netze N, $N_1$, $N_2$ und eine Synchronisationsmenge $A \subseteq \Sigma$ gilt:

$$RT(N_1) \subseteq RT(N_2) \Rightarrow RT(N_1 ||_A N) \subseteq RT(N_2 ||_A N) .$$

□

Bevor wir nun zum Hauptsatz des Kapitels kommen, müssen wir zunächst noch eine Forderung an ein vernünftiges Verhalten der zeitlichen Abläufe in den untersuchten Netzen stellen. Inwieweit diese Forderungen einschränkend sind und wie wir solche Netze besser fassbar charakterisieren können, werden wir dann in 1.4.6 detailiert untersuchen.

### *1.4.5.7 Definition*

Wir nennen ein Netz N *zeit-real*, wenn für alle $w \in RFS(N)$ gilt, dass es Verlängerungen w' gibt mit $w \cdot w' \in RFS(N)$ und $last(w') \subseteq \Sigma$.

### *1.4.5.8 Lemma*

Sei N ein Netz, dann sind äquivalent:

N ist zeit-real $\Leftrightarrow$ $\forall w \in RFS(N), D \in \mathbb{N}\ \exists w' \in (\Sigma \cup \wp(\Sigma))^*$ mit $w \cdot w' \in RFS(N) \wedge \zeta(w \cdot w') > D$ .

□

Wir können an dieser Stelle bereits sehen, warum nicht zeit-reale Netze ein Problem für unsere Zeit-Tests darstellen: Abläufe, die den Test wegen eines fehlenden ω nicht (rechtzeitig) bestehen, werden im Testszenario nicht wahrgenommen, wenn sie sich nicht über die Zeitschranke D hinaus verlängern lassen. Wir werden wie gesagt in 1.4.9 sehen, dass solche Netze pathologisch sind.

### *1.4.5.9 Satz*

Seien $N_1$, $N_2$ testbare und zeit-reale Netze. Dann gilt:

$$N_1 \sqsupseteq N_2 \Leftrightarrow RT(N_1) \subseteq RT(N_2) .$$

**Beweis:**

„$\Leftarrow$“:

Sei (O,D) ein d-Zeit-Test, den $N_1$ nicht besteht vermöge eines $w \in DL(N_1||O)$. Dann ist nach 1.4.5.3 und 1.4.5.6 auch $w \in DL(N_2||O)$ und $N_2$ besteht (O,D) ebenfalls nicht.

„$\Rightarrow$“:

Wir konstruieren hierfür für vorgegebenes $w \in (\Sigma \cup \wp(\Sigma))^*$ einen d-Zeit-Test (O,D), welchen genau diejenigen Netze bestehen sollen, für welche $w \in RT(N)$ *nicht* gilt. Solche Tests wurden bereits in [Bih98] und [BV04] vorgestellt. Gilt dann $N_1 \sqsupseteq N_2$ und wählen wir ein $w \in RT(N_1)$, so besteht $N_1$ den entsprechenden Test nicht, nach Definition $\sqsupseteq$ also auch $N_2$, und nach Konstruktion gilt folglich $w \in RT(N_2)$.

Die in obigen Arbeiten verwendeten Testnetze können wir in unserem Kontext wörtlich übernehmen. Wir müssen aber noch argumentieren, warum die Erweiterung um Lesekanten und *lazy-arcs* die Familie von Tests nicht verändert: Der Beweis des entsprechenden Theorems folgert zunächst, wie eine Verweigerungsfolge ohne $\omega$ im Testnetz selbst auszusehen hat. Um nun zu zeigen, dass dadurch einerseits für das getestete Netz zwingend w in der RT-Semantik enthalten sein muss und andererseits der Test dann auch fehlschlägt, bemüht der Beweis lediglich die Regeln der Synchronisation nach 1.4.5.5 (und nicht etwa die Definition auf den Netzgraphen nach 1.4.3.1).

Es sei darauf hingewiesen, dass die verwendeten Testnetze zeit-real und asynchron (und natürlich ohne Lesekanten oder *lazy-arcs*) sind und sich weiter sogar auf die Kantenintervalle [0,0] und [0,1] beschränken.

□

## 1.4.6 Charakterisierung durch endliche Transitionssysteme

Im Gegensatz zu der schwer greifbaren Definition aus 1.4.4, welche eine unendliche Anzahl von Testnetzen bemüht, haben wir im letzten Abschnitt eine sehr viel übersichtlichere Charakterisierung kennengelernt. Wollen wir $RT(N_1) \subseteq RT(N_2)$ jedoch mit Hilfe eines Tools prüfen, so müssen wir dazu immer noch die möglicherweise unendliche Anzahl von IDs in den Griff bekommen.

Dazu führen wir uns kurz vor Augen, dass das genaue Alter einer Marke auf einer Stelle s nur solange eine Rolle spielt, solange gilt: $\exists t \in s^{\bullet}: \rho(s) < ub(s,t) < \infty$. (Man beachte gegenüber früheren Arbeiten, dass natürlich der Fall $ub(s,t) = \infty$ gesondert berücksichtigt werden muss.) Erreicht oder überschreitet eine Marke das Alter

$$\rho_{max}(s) := max( \{ub(s, t) \mid t \in s^{\bullet}, ub(s, t) < \infty\} \cup \{0\} ) ,$$

so kann sie einfach als „alt“ gelten.

Wir erhalten so für ein Netz N eine Projektion PR von IDs auf IDs mit endlichem Wertebereich:

$$PR: ( \mathbb{N}_0 \cup \{\infty\} )^S \to \{ f \in \mathbb{N}_0^S \mid f(s) \leq \rho_{max}(s) \}, \text{ wobei}$$
$$PR( \rho )(s) := min( \rho(s), \rho_{max}(s) ) .$$

Außerdem modifiziert man die r-Schaltregel aus 1.4.5.1 wie folgt zu einer neuen Schaltregel $[\rangle_{rmax}$:

Sei N ein Netz und ρ und ρ’ IDs von N. Für eine Transition $t \in T$ schreiben wir $\rho[t\rangle_{rmax} \rho'$, wenn gilt $\rho[t\rangle \rho'$. Für eine Menge $X \subseteq \Sigma$ schreiben wir $\rho[X\rangle_{rmax} \rho'$, wenn gilt:

$$\forall t \in T ( \forall s \in {}^{\bullet}t: \rho(s) \geq ub(s,t) \vee \rho(s) = \rho_{max}(s) ) \Rightarrow l(t) \notin X \cup \{\lambda\} \quad \text{und}$$
$$\forall s \in S: \rho'(s) = min( \rho(s) + 1, \rho_{max}(s) ) .$$

Man beweist nun ganz einfach, dass die Transitionssysteme $TR_r(N)$ und $TR_{rmax}(N)$ , die $[\rangle_r$ und $[\rangle_{rmax}$ von der Start-ID aus induktiv definieren, vermöge *PR* stark bisimular sind.

Wie man am Wertebereich der IDs sieht, sind die $TR_{rmax}$ (im Gegensatz zu den $TR_r$) nicht nur Transitionssysteme, sondern nach 1.3.1.2 endliche Automaten. (Wir werden in Kapitel 2 noch sehen, wie wir diese effizient repräsentieren können.)

Nach 1.3.1.16 und 1.3.1.17 gilt nun:

$$TR_r(N_1) \text{ kann } TR_r(N_2) \text{ simulieren} \Leftrightarrow TR_{rmax}(N_1) \text{ kann } TR_{rmax}(N_2) \text{ simulieren}$$

und natürlich umgekehrt.

Nun ist die Sprache der $TR_r$ und $TR_{rmax}$ aber natürlich noch nicht RT, sondern RFS. Wir können aus beiden Transitionssysteme konstruieren, deren Sprache RT ist, indem wir in allen Kantenbeschriftungen Transitionen *t* durch ihre Aktionsbeschriftungen *l(t)* ersetzen und dabei die Verweigerungsmengen unverändert lassen. Mit der in 1.4.5.1 definierten Funktion *ll* können wir aus den Zustandsübergangsrelationen $\delta_r$ und $\delta_{rmax}$ von $TR_r$ und $TR_{rmax}$ auch konstruieren: Es seien $\delta_{lr}$ bzw. $\delta_{lrmax}$ die jeweils kleinsten Transitionssysteme mit

$$(p, x, q) \in \delta_r \Rightarrow (p, ll(x), q) \in \delta_{lr} \quad \text{bzw.}$$
$$(p, x, q) \in \delta_{rmax} \Rightarrow (p, ll(x), q) \in \delta_{lrmax} .$$

Es seien dann zwei neue Transitionssysteme $TR_{lr}$ und $TR_{lrmax}$ definiert, die jeweils durch Austausch der Zustandsübergangsrelationen aus $TR_r$ und $TR_{rmax}$ gegen $\delta_{lr}$ und $\delta_{lrmax}$ hervorgehen. Aus denselben Gründen wie oben gilt dann:

$$TR_{lr}(N_1) \text{ kann } TR_{lr}(N_2) \text{ simulieren} \Leftrightarrow TR_{lrmax}(N_1) \text{ kann } TR_{lrmax}(N_2) \text{ simulieren} .$$

Wir werden vor allem im Teil II der Arbeit, wo konkret über das Tool FastAsy gesprochen wird, nicht immer genau zwischen den verschiedenen Varianten unterscheiden und für die verschiedenen vorkommenden Transitionssysteme (sowie für Zwischenstufen mit gewisser praktischer Relevanz, die aber an dieser Stelle nicht erwähnt werden) den allgemeinen Begriff *r-Erreichbarkeitsgraph* verwenden.

Die folgenden Tabellen geben einen Überblick über die wichtigsten Transitionssysteme, mit denen wir RT/RFS charakterisieren:

| | $\mathbf{TR_r}$ | $\mathbf{TR_{rmax}}$ | $\mathbf{TR_{lr}}$ | $\mathbf{TR_{lrmax}}$ |
|---|---|---|---|---|
| *Sprache* | RFS | RFS | RT | RT |
| *endlicher Zustandsraum* | nein | ja | nein | ja |
| *buchstabierend* | ja | ja | nein | nein |
| *deterministisch* | ja | ja | nein | nein |

**Tabelle 1: Charakteristische Transitionssysteme 1**

| | $\mathbf{Ext(\ TR_{lr}\ )}$ | $\mathbf{Ext(\ TR_{lrmax}\ )}$ | $\mathbf{\wp(\ Ext(\ TR_{lrmax}\ ))}$ |
|---|---|---|---|
| *Sprache* | RT | RT | RT |
| *endlicher Zustandsraum* | nein | ja | ja |
| *buchstabierend* | ja | ja | ja |
| *deterministisch* | nein | nein | ja |

**Tabelle 2: Charakteristische Transitionssysteme 2**

Zusammenfassend dürfen wir mit 1.3.1.6 und 1.3.1.8 also, um $RT(N_1) \subseteq RT(N_2)$ zu entscheiden, prüfen ob $\wp(\ Ext(\ TR_{lrmax}(N_2)))$ das Transitionssystem $\wp(\ Ext(\ TR_{lrmax}(N_1)))$ simulieren kann.

Damit haben wir den letzten Schritt beisammen, um die $\sqsupseteq$-Relation vollautomatisch von einem Tool prüfen zu lassen, wie wir es in Teil II vorstellen.

## 1.4.7 Anmerkung zum Alphabet

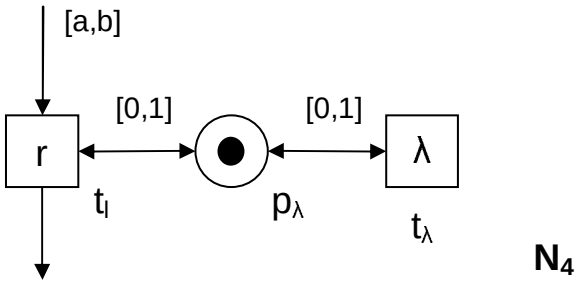
Wir sind bisher stillschweigend davon ausgegangen, dass zu vergleichende Netze über dasselbe Alphabet $\Sigma$ verfügen. Diese Annahme ist in der Tat auch in den allermeisten Fällen gegeben, da wir bei einem Effizienzvergleich ja in erster Linie daran interessiert sind, funktional *gleiche* Netze einander gegenüberzustellen.

Wie in 1.4.4.1 bereits dargestellt, ist es aber keine zwingende Voraussetzung für den Vergleich, dass für zwei beschriftete Netze $N_1$ und $N_2$ gilt $l(T_1) = l(T_2)$. Die Rechtfertigung, trotzdem von einem für alle betrachteten Netze universellen Alphabet auszugehen, ist die Folgende: Wollen wir eine endliche Menge von Netzen $N_1,\ldots,N_n$ vergleichen, so sei das universelle Alphabet $\Sigma := \Sigma_1 \cup \ldots \cup \Sigma_n$ . Da die einzelnen $\Sigma_i$ endlich waren, ist es auch $\Sigma$ noch. Dadurch steigt zwar für ein Netz $N_i$ mit $\Sigma_i \subset \Sigma$ die Anzahl der mit Verweigerungsmengen beschrifteten Kanten in $TR_r(\ N_i\ )$ bzw. $TR_{rmax}(\ N_i\ )$ an, jedoch nur in der in 1.4.5.3 angegebenen kontrollierten Form. Modulo dieser Gesetzmäßigkeit bleiben die RT-Erreichbarkeitsgraphen aber bei Vergrößerung des Alphabets isomorph.

## 1.4.8 Anmerkungen zu lazy-arcs

Allen in [BV98], [Bih98] und [BV04] vorgestellten Netzklassen ist gemeinsam, dass jede Transition nach einer festen Zeit schalten oder deaktiviert werden muss. In [BV98] haben wir aber z.B. einen Fall, in dem dies nicht erwünscht ist: Dort wird das Verhalten verschiedener MUTEX-Scheduler aus der Sicht eines einzelnen Benutzers untersucht. Für alle weiteren

Benutzer wird ein Standardverhalten angenommen. Es ist nun unnatürlich, einem solchen Standardbenutzer zu unterstellen, er würde zwangsläufig innerhalb einer bestimmten Zeit Interesse signalisieren, den kritischen Bereich zu betreten (was innerhalb der untersuchten Systeme durch Ausführen einer Aktion *r* modelliert war). Deshalb wurde in [BV98] zu einem Modellierungstrick gegriffen, den wir hier verallgemeinert und übertragen auf PTT-Netze




darstellen:

**Abbildung 4: Modellierung einer nicht-dringenden Transition ohne lazy-arcs**

Die neue Transition $t_\lambda$ sorgt dafür, dass das Alter der Marke auf $p_\lambda$ immer wieder (und ohne sichtbare Aktion) auf 0 gesetzt werden kann. Damit können wir t jederzeit ohne Auswirkungen auf das restliche Netz zu einer nicht-dringenden Transition machen, wenn wir Zeit vergehen lassen wollen.

Dieser wiederkehrende Modellierungstrick (im Software-Engineering würde man hier von einem *Idiom* sprechen) birgt allerdings zwei Nachteile:

- Er verschleiert jemandem, der ihn nicht kennt, die Absicht der Modellierung. (Siehe auch Absatz 9.3.2.1 im Teil II, wo für dieses Phänomen in ganz anderem Zusammenhang der Begriff *semantische Entfernung* eingeführt wird.)
- Bei der automatisierten Untersuchung solcher Systeme fällt negativ ins Gewicht, dass sowohl der benötigte Speicherplatz einer einzelnen ID wächst wie auch der gesamte Zustandsraum ansteigt.

Dem zweiten Punkt konnten wir freilich in *FastAsy* schon früher dadurch begegnen, dass wir Netze auf das „wörtliche“ Vorkommen dieses Musters (bzw. dessen Entsprechung in der dort verwendeten Netzklasse) untersucht haben und beim Aufbau des Erreichbarkeitsgraphen unterstellt haben, $t_\lambda$ würde stets sofort nach Aktivierung schalten. Eine solche Sonderbehandlung erschien jedoch nicht sonderlich befriedigend, da sie einerseits nicht orthogonal zu anderen Freiheitsgraden des Tools zu implementieren war und andererseits schon bei leichten Abweichungen vom Muster die Optimierungsfunktion nicht mehr griff.

In Folge dessen stellte sich die Frage, ob man nicht generell die Netzklasse derart erweitern sollte, dass einzelnen Transitionen ermöglicht würde, beliebig lange zu warten. (Vgl. dazu die Kanten mit expliziter Kontrolle der *fairness* in [KW97], mit denen sich die gleichen Effekte erzielen lassen, allerdings dort ohne Zeit.)

Obwohl es sich dabei offensichtlich um eine Eigenschaft der Transitionen handelt, fällt auf, dass man ausgehend von der Netzklasse in [BV04] das gewünschte Resultat auf elegante Art erhält, indem man den Wertebereich der oberen Intervallgrenzen von $\mathbb{N}_0$ auf $\mathbb{N}_0 \cup \{\infty\}$

abändert. Insbesondere vergewissert man sich, dass damit das Gros der Beweise wörtlich weiter gilt und alle anderen auf offensichtliche Weise angepasst werden können. Der Netzausschnitt aus Abbildung 4 stellt sich damit wie folgt dar:

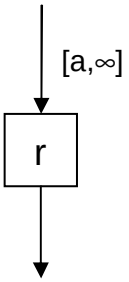



**Abbildung 5: Modellierung einer nicht-dringenden Transition mit lazy-arcs**

Man sieht natürlich, dass es das Verhalten einer Transition nicht beeinflusst, wenn weitere lazy-arcs hinzukommen. In der Tat sind bereits bei einer Transition mit *einem* eingehenden lazy-arc die Obergrenzen aller Intervalle auf anderen inzidenten Kanten unerheblich. Deswegen können wir auch von einer *lazy-transition* sprechen.

Aus praktischen Erwägungen heraus ist es sogar sinnvoll, bei einer lazy-transition $t_l$ grundsätzlich für alle Stellen s im Vorbereich $ub(s, t_l)$ auf ∞ zu setzen, denn so sinkt u.U. $\rho_{max}(s)$ für einige Stellen s ab, was den im Tool generierten Erreichbarkeitsgraph verkleinert.

Auch bezüglich der Parallelkomposition verhalten sich lazy-transitions mit der Modellierung über lazy-arcs automatisch so, wie man erwarten wird: Wird eine Transition mit einer lazy-transition synchronisiert, so ist die resultierende Transition selbst wieder *lazy*, was genau unserer Vorstellung entspricht, dass das schnellere System auf seinen langsameren Synchronisationspartner warten muss.

Es sei noch darauf hingewiesen, dass lazy-arcs die Anzahl der Zustände *nicht* in der Weise wie unsere erste Modellierung vergrößern. Dies bedeutet selbstverständlich nicht, dass sich nicht der Zustandsraum vergrößern kann, wenn wir eine normale Transition *lazy* machen (schließlich kann dadurch zusätzliches Verhalten möglich werden). Bei der ersten Version wird jedoch unnötigerweise noch zusätzlich zwischen verschiedenen Werten für das Alter von $p_\lambda$ unterschieden.

Wir haben in 1.4.6 schon gesehen, warum lazy-arcs den Zustandsraum im Tool immer noch endlich lassen.

Wir wollen nun kurz argumentieren, warum lazy-arcs die Ausdrucksmächtigkeit der Netzklasse bzgl. der in dieser Arbeit relevanten Semantiken nicht verändert. Dazu zeigen wir, dass die Konstruktion aus obigem $N_4$ in der Tat jeden lazy-arc ersetzen kann, so dass beide Varianten schwach bisimular sind im Sinne von 1.3.1.11. (Ein offensichtliches Beispiel, warum *starke* Bisumularität im Allgemeinen nicht gelten kann, wäre ein Netz mit lazy-arcs, aber mit $\lambda \notin l(T)$).

### *1.4.8.1 Satz*

Sei N ein PTT-Netz. Dann gibt es ein PTT-Netz ohne lazy-arcs N', so dass die Semantiken L, CL, DL, RT von N und N' schwach bisimular sind.

**Beweis:**

Für die Sprache L gilt die Aussage offensichtlich, da Zeitinformationen in der Definition gar nicht beachtet werden.

Wir beweisen die Aussage im folgenden für RT. Damit folgt die Aussage schon für DL. Der Beweis für CL ist völlig analog.

Sei also N ein PTT-Netz, das o.B.d.A. über einen einzelnen lazy-arc $(s_l, t_l)$ verfügt. (Wir werden diesen eliminieren, ohne weitere hinzuzufügen, so dass wir mehrere lazy-arcs einfach sukzessive behandeln können.)

Dann gehe N' aus N hervor, indem wir:

- Eine neue Stelle $p_\lambda$ und eine neue Transition $t_\lambda$ einfügen,
- Kanten $(t_\lambda, p_\lambda)$, $(p_\lambda, t_\lambda)$, $(t_1, p_\lambda)$, $(p_\lambda, t_1)$ einfügen,
- $(p_\lambda, t_\lambda)$ mit [0,1] und $(p_\lambda, t_l)$ mit [0,1] beschriften und
- $p_\lambda$ in $M_{N'}$ markieren.

Es seien TR und TR' die Transitionssysteme, die durch $M_N$ bzw. $M_{N'}$ und die $[\rangle\rangle_r$-Schaltregel aus 1.4.5.1 induktiv definiert sind. Zustände der Transitionssysteme sind also genau die r-erreichbaren IDs der Netze, die Alphabete sind gegeben durch $\Lambda = l(T) \cup \wp(\Sigma)$ und $\Lambda' = l(T') \cup \wp(\Sigma)$.

Es sei eine Relation $B \subseteq Q \times Q'$ wie folgt definiert: $(\rho, \rho') \in B \Leftrightarrow \rho(s) = \rho'(s)$ *für* $s \in S$. Offenbar gilt für alle $\rho' \in Q'$: $\rho'(p_\lambda) \in \{0,1\}$. Das Paar der Startzustände ist per Definition in B, so dass von 1.3.1.12 nur noch 2) und 3) zu zeigen bleibt. Seien also $(\rho, \rho') \in B$.

*„2)"*:

Es gelte $\rho \xrightarrow{\varepsilon} \tau$ für eine ID $\tau$ und ein $\varepsilon \in \Lambda \cup \{\lambda\}$.

Falls $\varepsilon \in l(T)$, so gilt, da wir ja die zu Grunde liegenden Transitionen (es können ja auch interne beteiligt sein) auch in N' schalten können, offensichtlich für $\tau'$ mit $\tau'|_S = \tau|_S$ und $\tau'(p_\lambda) \in \{0,1\}$, dass $\rho' \xrightarrow{\varepsilon} \!\!\!\!\rightarrow \tau'$, und nach Definition auch $(\tau, \tau') \in B$.

Falls $\varepsilon \subseteq \Sigma$, so können wir alle internen Transitionen, die an dem Schritt beteiligt sein könnten, wie im ersten Fall durch sich selbst simulieren; wir dürfen also o.B.d.A. annehmen, dass an der ε unterliegenden RFS-Schaltfolge gar keine Transitionen beteiligt waren und folglich gilt $\tau = \rho + 1$. Gilt nun $\rho'(p_\lambda) = 1$, so können wir ε nicht sofort verweigern und müssen zunächst $t_\lambda$ schalten, womit wir zu ρ'' gelangen, für das gilt $\rho''|_S = \rho|_S$ und $\rho''(p_\lambda) = 0$. (Andernfalls sei einfach $\rho'' = \rho$.)

Von ρ'' aus können wir nun in jedem Fall in N' ebenfalls ε verweigern und es gilt $\rho'' \xrightarrow{\varepsilon} \tau'$ mit $\tau' = \rho'' + 1$. Nach Definition von ρ'' gilt jetzt $\tau'(p_\lambda) = 1$. Setzen wir die Schaltfolge nun zusammen, gilt insgesamt $\rho' \xrightarrow{\varepsilon} \!\!\!\!\rightarrow \tau''$ und $(\tau, \tau'') \in B$.

*„3)"*:

Es gelte $\rho' \xrightarrow{\varepsilon} \tau'$ für eine ID $\tau'$ und ein $\varepsilon \in \Lambda'$.

Für $\varepsilon \in l(T')$ simulieren wir jedes $t \in T$ durch sich selbst, $t_\lambda$ hingegen wird einfach durch das leere Wort simuliert.

Für $\varepsilon \subseteq \Sigma$ nehmen wir wie in 2) o.B.d.A an, dass in TR das ε isoliert steht. Jetzt bewirkt genau $ub(s_1, t_1)=\infty$, dass $t_1$ in N' niemals dringend ist und wir also ε gleichermaßen in N verweigern dürfen. Wie man sofort sieht, ist dann für τ mit $\rho \overset{\varepsilon}{\twoheadrightarrow} \tau$ auch wieder $(\tau, \tau') \in B$.

□

Bemerkung: Wie man am Beweis sieht, gilt sogar noch eine etwas stärkere Äquivalenz zwischen N und N' als Bisimulation, da ja jedes Schalten einer Transition t aus N durch Schalten der gleichen Transition in N' (ggf. unter Hinzunahme von $t_\lambda$) und mit Ausnahme von $t_\lambda$ auch andersherum simuliert wird. Genauer gesagt gilt Bisimularität also auch noch, wenn wir $\forall t \in T: l(t) = l'(t) := t$ und $l'(t_\lambda) := \lambda$ setzen.

Man könnte nun durch die Einführung der lazy-arcs versucht sein, wieder mit dem traditionelleren Begriff von Asynchronizität zu operieren, nämlich dem, bei dem sich *alle* Transitionen beliebig Zeit lassen dürfen. Wir erinnern uns aber daran, dass die RT-Semantik ja eine *worst-case* Semantik darstellt, und der *worst-case* in diesem traditionellen Szenario ist nach wie vor der Fall, dass alle Transitionen immer weiter warten.

### 1.4.9 Anomalien nicht zeit-realer Netze

Wie angekündigt, wollen wir nun untersuchen, wie nicht zeit-reale Netze einzuordnen sind. Prinzipiell handelt es sich dabei ja um ein Artefakt der Modellierung, welches man sich primär dadurch einhandelt, dass man 0 als obere Zeitschranke *ub* zulässt und somit eine Reaktion in *Nullzeit* verlangt: Für Transitionen t kann damit z.B. auch gelten $ub(\blacksquare t)=0$, womit eine solche Transition in jedem Fall vor dem Vergehen von Zeit schalten muss, wenn sie nicht deaktiviert wird. (Genauer betrachtet kann auch noch das gänzliche Fehlen eines nicht-erweiterten Vorbereichs die Ursache sein, dass ein Netz nicht zeit-real ist, s.u.)

Obwohl es nun so erscheinen mag, als ob die Aktivierung einer Transition mit $ub(\blacksquare t)=0$ das Vergehen von Zeit verhindert, mache man sich bewusst, dass die duale Sichtweise, nämlich die, dass vor dem Ablaufen von Zeit einfach noch bestimmte Dinge zu geschehen haben, weit weniger unplausibel ist. Wir verbieten ja nicht in einem System das Vergehen von Zeit, sondern nur das Aufschreiben eines bestimmten Ablaufs, welchen das System eben nicht produzieren kann.

Die Situation wird allerdings dann trotzdem heikel, wenn eine unendliche Folge solcher Transitionen aktiviert wird. Dann nämlich hat man unendliche Abläufe, in denen *nie mehr* Zeit vergeht. Anders gesagt kann die Frage, welchen Zustand ein System zu einem bestimmten Zeitpunkt hat, für keinen Zeitpunkt nach dem Eintreten einer solchen Situation mehr beantwortet werden. Sind alle diese Transitionen nun auch noch intern, kann nicht einmal mehr ein Synchronisationspartner (insbesondere auch kein Testnetz) dieses Verhalten entschärfen. (Im Gegenteil wird der Synchronisationspartner sogar mit dem Phänomen „infiziert".)

Wie wir gesehen haben, stellt dies ein Problem für das Testen dar, weil solche Netze die Testumgebung „betrügen" können: Wenn gar keine Abläufe mit Dauer >D vorhanden sind, werden *alle* Tests (O,D) automatisch bestanden. Netze mit solchem Verhalten können wir deshalb mit unserem Ansatz nicht behandeln. Wir betrachten sie als Spezifizierungsfehler – dazu müssen wir aber natürlich genau umreißen, welches Verhalten wir verbieten wollen.

In [Bih98] wird unter anderem auch ein statisch prüfbares, notwendiges Kriterium für nicht zeit-reale Netze hergeleitet, nämlich das Vorhandensein eines Kreises im Netzgraphen, längs dem alle Transitionen intern sind mit $ub({}^{\bullet}t)=0$. Durch die Erweiterung auf PTT-Netze verändern sich allerdings einige der Resultate. Außerdem wurden in der genannten Arbeit Transitionen ohne Vorbereich kategorisch ausgeschlossen, wohingegen wir das entsprechende Verbot hier in diesen Zusammenhang explizit rechtfertigen werden. Weiter werden wir gegenüber [Bih98] die auftretenden Anomalien systematischer klassifizieren. Dazu zunächst einige Definitionen:

### *1.4.9.1 Definition*

Sei N ein Netz.

- Für eine CID ρ ist die Menge der unendlichen kontinuierlichen Schaltfolgen ab ρ definiert durch:
  $CFS^{\omega}(\rho) := \{ w \in (T \cup \mathbb{R}^+)^{\omega} \mid$ *für jedes endliche Präfix v von w gilt* $\rho[v\rangle_c \}$,
  analog definiert man $DFS^{\omega}(\rho)$, $RFS^{\omega}(\rho)$, $DT^{\omega}(\rho)$, $RT^{\omega}(\rho)$.
  ζ und α seien auf unendlichen Folgen in der offensichtlichen Weise definiert.
- Eine Transition t heißt *Prezero-Transition*, wenn $\forall s \in {}^{\bullet}t: ub(s,t) = 0$, sie heißt *Zero-Transition*, wenn $\forall s \in t^{\bullet}: ub(s,t) = 0$ und ansonsten *Nonzero-Transition*.
  Es seien *PZ(N)*, *Z(N)* und *NZ(N)* die Mengen der Prezero-, Zero- und Nonzero-Transitionen von N.
- Eine Transition t heißt *isoliert*, wenn ${}^{\bullet}t = t^{\bullet} = \emptyset$, sie heißt *autark*, wenn ${}^{\bullet}t = \emptyset$ und *präautark*, wenn ${}^{\circ}t = \emptyset$.
  (Damit ist jede autarke Transition natürlich eine Zero-Transition und jede präautarke eine Prezero-Transition. In sicheren Netzen ist jede autarke Transition auch isoliert und für jede nicht-tote, präautarke Transition gilt $t^{\bullet} = \emptyset$.)
- Wir sagen für $w \in CFS(N)$ bzw. für eine CID $\rho_0$ und $w \in CFS^{\omega}(\rho_0)$, eine CID *ρ tritt längs w auf*, wenn ein Präfix v von w existiert, so daß ρ die CID nach v ist, analog für IDs ρ und $w \in DFS(N)$ bzw. $w \in RFS(N)$.
- Ein CID ρ von N ist tot, wenn $\forall t \in T: \exists s \in {}^{\bullet}t: \rho(s) = -\infty$.
  (Man beachte, daß dies im Gegensatz zu Standard-Petri-Netzen nicht äquivalent dazu ist, dass unter ρ keine Transition schalten kann, denn in nicht-asynchronen PTT-Netzen kann das Vergehen von Zeit wieder Transitionen aktivieren.)
- Wir nennen für $\varepsilon \in \mathbb{R}^+$ und eine CID ρ' mit $\rho'[\varepsilon\rangle_c$ den Zeitschritt ε *maximal*, wenn $\varepsilon = Res_{ub}(\rho) < \infty$ oder wenn $\varepsilon = 1$ und $Res_{ub}(\rho) = \infty$. (Dabei bedeutet der zweite Fall natürlich schon, dass ρ tot ist.)

□

Präautarke Transitionen sind deshalb unschön, weil sie sich, wenn sie aktiviert werden, nicht selbst deaktivieren und so unendlich oft hintereinander schalten können (sogar ohne Vergehen von Zeit, selbst wenn die inzidenten Lesekanten mit $ub > 0$ beschriftet sind).

Wir wollen zunächst zwei Sätze über maximale Zeitschritte beweisen. Betrachten wir die Dauer unendlicher Abläufe, sind maximale Zeitschritte deswegen wichtig, weil in asynchronen Netzen natürlich eigentlich *nie* Zeit vergehen muss – alle Abläufe können ebensogut in Zeit 0 stattfinden. Die Untersuchung von Abläufen mit maximalen Zeitschritten garantiert uns, dass wir niemals „böswillig“ keine Zeit mehr vergehen lassen.

### *1.4.9.2 Satz*

Es sei N ein Netz und $w \in CFS^{\omega}(CID_N)$, so dass w nur maximale Zeitschritte beinhaltet. Dann gilt für jede Stelle s, dass die Uhrenwerte auf s alle aus $\mathbb{N}_0 \cup \{-\infty\}$ sind.

**Beweis:**

Die Uhrenwerte kommen nur durch Addition, Vorzeichenwechsel, Maximums- und Minimumsbildung aus den natürlichen Zahlen $CID_N(s)$ und weiter 0 und $-\infty$ zustande. Sie sind zusätzlich per Definition entweder gleich $-\infty$ oder aber $\geq 0$.

□

Es ließen sich also auch dadurch endliche Transitionssysteme aus der CFS-Semantik gewinnen, dass man nur solche Zeitschritte berücksichtigt, die maximal sind. Dieser Ansatz wird indes in dieser Arbeit nicht weiter verfolgt. Weit wichtiger für uns ist folgende Aussage:

### *1.4.9.3 Satz*

Es sei N ein Netz, $\rho$ eine CID und $w \in CFS^{\omega}(\rho)$, so dass w unendlich viele maximale (insbesondere unendlich viele) Zeitschritte beinhaltet. Dann gilt $\zeta(w) = \infty$.

**Beweis:**

Wir zeigen, dass in einem endlichen Präfix von w Zeit 1 vergeht. Dann trifft auf den Rest von w wiederum die Voraussetzung zu und wir sind fertig.

Sei $\varepsilon$ ein maximaler Zeitschritt in w und $\rho$, $\rho'$ die CIDs vor und nach $\varepsilon$. Dann ist $\varepsilon$ entweder $\geq 1$ (und es gilt schon die Behauptung) oder nach Definition gilt:

$$\exists t \in T,\ s \in {}^{\bullet}t:\ \varepsilon = ub(s,t) - \rho(s) > 0.$$

Nach Definition der Schaltregel gilt damit:

$$\rho'(s) = ub(s,t).$$

Da $\varepsilon > 0$, ist damit auch $ub(s,t) > 0$ und nach Definition sogar $ub(s,t) \in \mathbb{N}$. Da wir unendlich viele maximale Zeitschritte, aber nur endlich viele Stellen und Transitionen haben, können wir obige s und t so wählen, dass für unendlich viele $\rho_i$ nach maximalen Zeitschritten längs w gilt $\rho_i(s) = ub(s,t) \in \mathbb{N}$. Betrachen wir für beliebiges k nun $\rho_k$ und $\rho_{k+1}$: Da zwischen $\rho_k$ und $\rho_{k+1}$ mindestens ein Zeitschritt (nämlich der vor $\rho_{k+1}$) steht, muss die Marke von s zwischenzeitlich verschwunden sein, also zwischen beiden ein $\rho_0$ auftreten mit $\rho_0(s) = 0$. Damit muss die Summe der Zeitschritte zwischen $\rho_0$ und $\rho_{k+1}$ aber mindestens ub(s,t), also $\geq 1$ sein.

□

Bemerkung: Wählt man die Zeitschritte nicht maximal, können selbst unendlich viele Zeitschritte nur endliche Zeit dauern, indem man die Zeitschritte etwa als $\varepsilon_n := 1/2^n$ wählt. Die Aussage dieses Satzes ist die, dass man selbst unter der wenig restriktiven Annahme, dass man nur *immer irgendwann wieder* die zur Verfügung stehende Zeit auch aufbraucht, vom Netz nicht in eine solche konvergierende unendliche Reihe von Zeitschritten (eine sog. *Zeno-Folge*) gezwungen werden kann.

Dahingegen kann ein Netz wie gesagt sehr wohl den Fall erzwingen, dass gar kein Zeitschritt mehr stattfindet. Wir wollen kurz einen Zusatz zu unserem Diskretisierungsresulat formulieren, welcher uns sagt, dass wir diese Phänomäne auch in DFS bzw. RFS studieren

können. Dazu sei für ein $w \in CFS(N)$ die Sequenz $Discr(w) \in DFS(N)$ durch die Konstruktion im Beweis von 1.4.4.3 gegeben und erfüllt damit insbesondere $\alpha( Discr(w) ) = \alpha(w)$ und $\zeta( Discr(w) ) - \zeta(w) \in [0, 1[$. (Wohlgemerkt ist $Discr(w)$ eindeutig definiert, wir behaupten jedoch nicht, dass es die einzige Sequenz mit dieser Eigenschaft für w ist.)

Damit gilt folgendes Lemma:

### *1.4.9.4 Lemma*

Seien für ein Netz N die Sequenzen w, w'∈CFS(N) gegeben, so dass w' eine Verlängerung von w ist. Dann ist *Discr(w')* auch eine Verlängerung von *Discr(w)*.

**Beweis:**

Die Konstruktion von v im Beweis von 1.4.4.3 erfolgt induktiv.

□

### *1.4.9.5 Korollar*

Sei $w \in CFS^{\omega}(N)$ und $\zeta(w) < \infty$. Dann existiert ein $v \in DFS^{\omega}(N)$ mit $\zeta(v) < \infty$ und ein $u \in RFS^{\omega}(N)$ mit $\zeta(u) < \infty$.

**Beweis:**

Nach 1.4.9.4 existiert ein v, so dass jedes Präfix v' von v ein Präfix von *Discr(w')* für ein Präfix w' von w ist. Wegen der Präfixabgeschlossenheit von DFS ist v' dann auch in *DFS(N)*.

(Betrachtet man den Beweis von 1.4.4.3 noch genauer, so stellt man fest, dass sogar jedes v' selbst die Form *Discr(w')* hat, doch dies wird gar nicht benötigt.)

Das zugehörige u liefert uns wiederum eine Argumentation über die Präfixe zusammen mit Lemma 1.4.5.2.

□

Wir wollen nun die Anomalien näher benennen:

### *1.4.9.6 Definition*

Sei N ein Netz und ρ eine ID von N. Wir nennen für ein $w \in RFS^{\omega}(\rho)$ das Paar $(\rho,w)$

- einen *refusal-timestop*, wenn nach keinem endlichen Präfix von w eine Verweigerungsmenge folgen kann.
- einen *timestop*, wenn nach keinem endlichen Präfix von w ein Zeitschritt folgen (also Σ verweigert werden) kann.

In beiden Fällen ist offensichtlich $\zeta(w)=0$. Wir sagen, ρ hat einen timestop, wenn ein w existiert, so dass $(\rho,w)$ ein timestop ist.

Wir nennen weiter einen timestop $(\rho,w)$

- *intern*, wenn $l(w)=\lambda$
- *sichtbar*, wenn $\alpha(l(w))$ unendlich lang ist
- *erreichbar*, wenn ρ d-erreichbar
- *potentiell*, wenn ρ r-erreichbar
- *hart*, wenn alle t in w dringend sind
- *quasi-hart*, wenn alle t in w dringend sind oder mit dringenden in Konflikt stehen.
- *semi-hart*, wenn w unendlich viele dringende Transitionen enthält

- *semi-quasi-hart*, wenn für unendlich viele t in w gilt, dass sie dringend sind oder mit dringenden in Konflikt stehen.

Wir nennen schließlich eine CID ρ einen *timelock*, wenn für alle $w \in RFS^{\omega}(\rho)$ gilt, dass *(ρ,w)* timestop ist. In natürlicher Weise sind dann refusal-timelock und weiter interner, sichtbarer, erreichbarer, usw… timelock definiert.

□

Es ist natürlich nur im Hinblick auf technische Aspekte sinnvoll, von timestops zu reden, die nicht einmal potentiell sind (deren ID also nicht einmal r-erreichbar ist).

Bemerkung zu [Bih98]: Unsere harten timestops sind den *Σ-timestops* aus der früheren Arbeit vergleichbar, unsere refusal-timelocks entsprechen genau den dortigen *internal-timestops*. Gegenüber der früheren Arbeit stützen sich nun timelocks auf timestops ab, was wesentlich befriedigender erscheint. In [Bih98] standen sich Σ-timestops und internal-timestops noch isoliert gegenüber.

Die Motivation für die Definition eines timestop ist die, dass wir damit einen Ablauf vor uns haben, der unendlich viele Transitionen in Nullzeit schaltet und zwar eben nicht „böswillig“, denn es besteht ja nach keiner Transition Gelegenheit, Zeit vergehen zu lassen. (Oder anders ausgedrückt: Es handelt sich nicht nur um die schnelle Variante eines Ablaufs, in dem genauso gut auch Zeit vergehen könnte.) Dabei ist es aber natürlich so, dass eine andere Auswahl von Transitionen (im Gegensatz zum timelock) dazu führen kann, dass auch wieder Zeit vergehen darf.

In [Bih98] werden nur Sequenzen dringender Transitionen untersucht. Die Idee dahinter war, dass das Schalten einer nicht-dringenden Transition sozusagen bereits als „Böswilligkeit“ ausgelegt wird.

Betrachten wir dazu folgendes Beispiel mit der Start-ID als ρ:

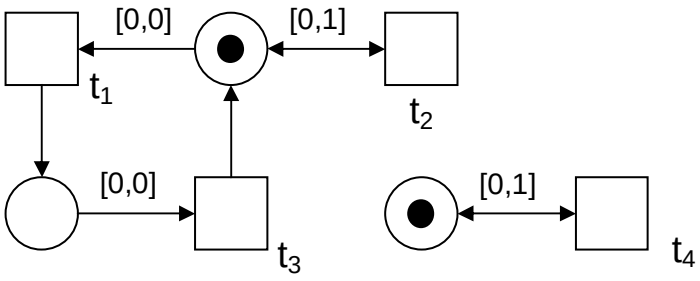



**Abbildung 6: timestops**

Nach [Bih98] wäre in Abbildung 6 das Paar $(\rho, (t_1t_3)^{\omega})$ ein timestop, aber $(\rho, t_2^{\omega})$ nicht – denn $t_2$ selbst ist ja in diesem Ablauf nie dringend. Intuitiver ist es aber wohl doch, auch Sequenzen mit $t_2$ als „schlimme“ timestops aufzufassen, schließlich muss $t_2$ einfach sofort schalten, wenn es nicht deaktiviert werden will. Der Begriff quasi-hart umfasst nun genau solche Situationen – während $(\rho, (t_1t_3)^{\omega})$ ein harter timestop ist, ist $(\rho, t_2^{\omega})$ quasi-hart (aber nicht semi-hart).

In $(\rho, (t_1t_3t_4)^{\omega})$ wiederum schalten zwar immer wieder dringende Transitionen, aber mit $t_4$ eben auch ab und zu nicht-dringende. (D.h. das System „tut meistens sein bestes“.) Somit ist dies ein semi-harter, aber kein quasi-harter timestop.

Mit $(\rho, (t_2t_4)^\omega)$ haben wir schließlich noch einen Vertreter der semi-quasi-harten timestops: Obwohl $t_4$ völlig isoliert steht, kommt mit $t_2$ zumindest unendlich oft eine Transition vor, die mit einer dringenden in Konflikt steht.

Sogar das Paar $(\rho, (t_4)^\omega)$ ist zwar noch ein timestop, aber nicht einmal mehr semi-quasi-hart. In dieser Sequenz ignorieren wir also „grundlos“ (d.h. ohne Konflikt) fortwährend dringende Transitionen zugunsten einer Transition, welche eigentlich noch warten könnte. Trotzdem gilt natürlich auch noch für $t_4$, dass es entweder schnell schalten muss oder gar nicht.

Setzte man das Intervall bei $t_1$ auf [0,1], so gäbe es gar keine timestops mehr.

Der Effekt, den $t_2$ und $t_3$ auf $t_4$ ausüben, kann natürlich auch in einer Parallelkomposition auftreten: Stellen wir uns ein beliebiges Netz N vor und synchronisieren es über der Menge ∅ mit dem Netz aus Abbildung 7, ist bereits die Start-ID ein timelock (und für den Fall $l(t_0)=\lambda$ sogar ein refusal-timelock), und für alle Abläufe $w \in \alpha(DFS(N))$ ist $(CID_N, w)$ ein timestop, obwohl $t_0$ darin nie vorkommt. (Wir sagen auch, dass timestops unter bestimmten Umständen in der Parallelkomposition „infektiös“ sein können.)

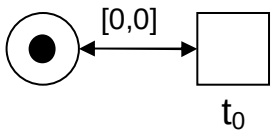



**Abbildung 7: Ein „infektiöses“ Petri-Netz**

Ebensogut kann aber ein vorhandener timestop durch Parallelkomposition verschwinden, weil eine mit einer Aktion aus der Synchronisationsmenge beschriftete Transition durch ihren Synchronisationspartner bedingt auf einmal warten oder nicht mehr schalten darf. Man betrachte etwa die Parallelkomposition des Netzes von Abbildung 7 (mit $l(t_0)\neq\lambda$ als Teil der Synchronisationsmenge) und einem weiteren Netz, in dem keine Transition mit $l(t_0)$ beschriftet ist.

Zu den vorgestellten Varianten zeitlicher Anomalien ist abschließend wohl anzumerken, dass man sich damit abfinden muss, dass das abstrakte Netzmodell keine Antwort darauf geben kann, welche Arten von timestops/timelocks Spezifizierungsfehler darstellen und welche verwendet werden dürfen. Obige Klassifikation kann aber Hinweise geben, welche Gedanken man sich in einem Anwendungsfall machen muss. Eine genauere Untersuchung der Beziehungen zwischen den verschiedenen Varianten wäre dafür mit Sicherheit interessant, im Rahmen dieser Arbeit wollen wir aber nur kurz folgende Implikationen zwischen timelocks und timestops zeigen:

### *1.4.9.7 Lemma*

Sei für ein Netz N die ID ρ ein timelock.Dann existiert eine Sequenz $w \in RFS^\omega(\rho)$, so dass $(\rho,w)$ ein harter timestop ist.

Ist ρ ein refusal-timelock, so existiert sogar ein w, so dass $(\rho,w)$ ein harter *interner* refusal-timestop ist.

**Beweis:**

Jede von ρ erreichbare ID ist offensichtlich selbst wieder ein (refusal-)timelock. Weiter ist unter jedem timelock natürlich zumindest eine Transition dringend, ansonsten könnte ja ein Zeitschritt schalten. Wir konstruieren also sukzessive w, indem wir immer eine dringende Transition schalten. Nach Definition eines (refusal-)timelocks ist $(\rho,w)$ dann ein (refusal-)timestop, und nach Konstruktion ist w auch hart.

Ist ρ ein refusal-timelock, so müssen offenbar sogar jeweils interne Transititonen dringend sein.

□

Wir wollen nun die Frage beantworten, welche Art von timestops mit unseren Effizienztests verträglich ist. (Das Kriterium für diese Veträglichkeit haben wir ja in 1.4.5.7 bereits mit dem Begriff *zeit-real* eingeführt.)

### *1.4.9.8 Satz*

Es sind äquivalent:

N ist zeit-real $\Leftrightarrow$ N hat keinen potentiellen refusal-timelock

**Beweis:**

‚$\Rightarrow$':

Nehmen wir im Widerspruch zur Aussage an, ein zeit-reales Netz N hätte einen potentiellen refusal-timelock ρ. Dann existiert w, so dass ρ vermöge w r-erreichbar ist. Da N zeit-real, existiert weiter ein w', so dass $w\cdot w'\in RFS(N)$ und $last(w')\subseteq\Sigma$. Nun ist w·w' aber ein endliches Präfix einer unendlichen Folge w'' aus $RFS^{\omega}(\rho)$, denn jede Folge aus RFS kann offensichtlich beliebig verlängert werden. Somit ist $(\rho,w'')$ kein refusal-timestop und weiter ρ kein refusal-timelock.

‚$\Leftarrow$':

Wenn N nicht zeit-real ist, so existiert ein $w\in RFS(N)$, so dass für alle Verlängerungen w' mit $w\cdot w'\in RFS(N)$ gilt, dass $last(w') \in T$. Anders gesagt existiert ein r-erreichbares ρ, so dass für alle $w'\in RFS(\rho)$ gilt $w'\in T^*$. Damit ist aber ρ nach Definition von $RFS^{\omega}$ schon ein refusal-timelock.

□

Im Prinzip kann ein Tool durch Aufbau des r-Erreichbarkeitsgraphen testen, ob ein Netz zeit-real ist. Wir sind aber natürlich auch an einem statisch prüfbaren Kriterium dafür interessiert. Wir werden uns dabei mit einem hinreichenden Kriterium begnügen müssen, welches jedoch keine unzumutbaren Einschränkungen fordert. Zunächst wollen wir uns aber noch kurz den autarken Transitionen zuwenden:

### *1.4.9.9 Satz*

Sei N ein Netz mit einer autarken Transition t. Dann ist jede ID ρ von N ein timelock. Gilt l(t)=λ, ist jedes ρ sogar ein refusal-timelock.

**Beweis:**

Nach 1.4.2.3 ist t unter jedem ρ dringend.

□

Es ist also weder verwunderlich noch eine echte Einschränkung (man beachte, dass in unserem Setting autarke Transitionen ja sogar automatisch isoliert sind), wenn man autarke Transitionen verbietet. [Bih98], [BV98], [BV04] tun dies ja wie gesagt als General-voraussetzung. Im vorliegenden Werk arbeiten wir hingegen in voller Allgemeinheit. Man beachte aber, dass die zentrale Aussage 1.4.5.9 immer zeit-reale Netze voraussetzt.

Auch präautarke Transitionen sind als potentiell problematisch zu betrachten: Sie sind wie autarke Transitionen zu sich selbst nebenläufig, und in sicheren Netzen sind sie, wie bereits erwähnt, entweder tot oder ohne Nachbereich und damit auch nur sehr eingeschränkt sinnvoll. Ist eine ID erreichbar, in der eine präautarke Transition dringend ist, so hat diese ID bereits einen harten timestop. Im Gegensatz zu autarken Transitionen haben präautarke Transitionen allerdings nicht automatisch einen timelock zur Folge.

Wie wir in 1.4.9.11, 1.4.9.13 und 1.4.9.14 sehen werden, hat es trotzdem große Vorteile für die Abschätzbarkeit des Verhaltens, wenn wir auch präautarke Transitionen ausschließen. (D.h. man sollte bevorzugt mit Netzen modellieren, die keine präautarken Transitionen besitzen.)

### *1.4.9.10 Definition*

Sei N ein Netz und $K = t_1, s_1, \ldots, t_{n-1}, s_{n-1}, t_n$ mit $t_1 = t_n$ und $n>1$. Wir nennen K:

- einen *gerichteten geschlossenen Kantenzug*, falls
  $\forall i<n: (t_i, s_i) \in F \cup R^{-1}$ und $(s_i, t_{i+1}) \in F \cup R$
- einen *gerichteten geschlossenen F-Kantenzug*, falls
  $\forall i<n: (t_i, s_i) \in F$ und $(s_i, t_{i+1}) \in F$
- einen *gerichteten Kreis*, falls K ein gerichteter geschlossener Kantenzug und
  $\forall i,j<n$ gilt, dass $t_i \neq t_j, s_i \neq s_j$
- einen *gerichteten F-Kreis*, falls K ein gerichteter geschlossener F-Kantenzug und
  $\forall i,j<n$ gilt, dass $t_i \neq t_j, s_i \neq s_j$ .

□

Bemerkungen: Da im Kontext von Netzgraphen nicht gerichtete Kantenzüge keine Rolle spielen, werden wir das Wort „gerichtet“ auch weglassen. Weiter beachte man, dass aus der Existenz eines geschlossenen (F-)Kantenzugs $t_1, \ldots, t_n$ sofort die Existenz eines (F-)Kreises $t_1, \ldots, t_n$ folgt.

### *1.4.9.11 Lemma*

Sei N ein Netz ohne präautarke Transitionen. Dann gilt:

1) Ist N asynchron und läßt sich eine Transition t mindestens zweimal schalten, so existiert in N ein Kreis, der t enthält.
2) Läßt sich t unendlich oft schalten, so existiert sogar ein F-Kreis.

**Bemerkung**: Wir haben außerdem die dringende Vermutung, dass noch eine weitere Aussage gilt, die wir bis jetzt jedoch noch nicht vollständig beweisen konnten. Sie wird im weiteren auch nirgends verwendet, wir wollen sie jedoch trotzdem als Hypothese angeben:

3) Ist N asynchron und läßt sich t unendlich oft schalten, so existiert sogar ein F-Kreis, der t enthält.

**Beweis:**

*„1“*:

Da N asynchron ist, dürfen wir die kausale Argumentation traditioneller Petri-Netze bemühen: Da t nicht präautark ist, deaktiviert es sich beim Schalten selbst. Wäre t nun nicht unter seinen eigenen Ursachen, könnte man auch ohne das Schalten von t weitere Marken in den Vorbereich von t legen und N wäre nicht sicher.

(In der Tat könnte man für diesen ersten Teil die Voraussetzung „präautark“ noch zu „autark“ aufweichen, da eine präautarke Transition t nach Definition bereits einen Kreis, der t enthält, impliziert.)

*„2“*:

Es sei w eine unendliche Sequenz, in der eine Transition $t_0 = t$ unendlich oft schaltet. Ferner sei $T^1$ die Menge der Transitionen, von denen t im Verlauf von v zumindest einmal eine Marke auf eine Stelle aus $^\bullet t_0$ erhält. Da $t_0$ nicht präautark ist, gilt $T^1 \neq \varnothing$. Weiter muss, da $T^1$ endlich ist und $t_0$ unendlich oft aktiviert wird, eine Transition $t_1$ in $T^1$ existieren, von der $t_0$ unendlich oft eine Marke in $^\bullet t_0$ erhält. Damit muss auch $t_1$ unendlich oft schalten. Zu $t_1$ konstruieren wir ebenso $T^2$, und sukzessive unendlich viele $T^n$ und $t_n$ . Wegen des endlichen Vorrats an Transitionen müssen aber $i,j$ existieren mit $i \neq j$ und $t_i = t_j$ .

□

Es ist sehr illustrativ, sich an dieser Stelle einige Gegenbeispiele anzusehen, um zu überprüfen, ob die Voraussetzungen in 1.4.9.11 tatsächlich alle nötig sind.

Betrachten wir zunächst in Abbildung 8 eine Famile asynchroner Netze $N_n$, in denen sich eine Transition t sogar jeweils n+1-mal schalten lässt, die aber keinen F-Kreis enthalten:

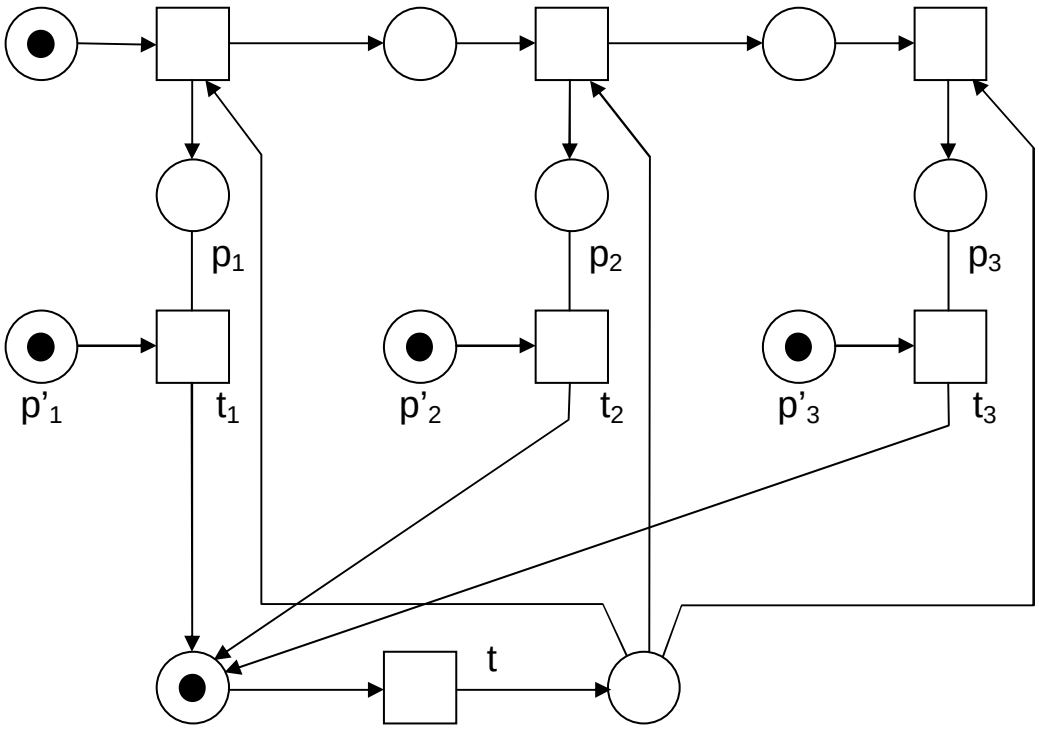



**Abbildung 8: $N_n$ für n=3**

Wie wir sehen, benutzen die $N_n$ den nur endlichen Zustandsraum, um über zusätzliche Stellen $p'_i$ den Lesekanten $(p_i, t_i)$ die Semantik einer normalen Kante (d.h. mit Verbrauch der Marke im Vorbereich) zu geben. Dieser Trick kann jedoch nur für eine jeweils gegebene Zahl an Wiederholungen verwendet werden, denn auf die $p'_i$ können keine Marken mehr gelegt werden – ansonsten hätten wir doch wieder einen F-Kreis.

In Abbildung 9 zeigen wir ein nicht-asynchrones Netz, welches seine Taktung zur Steuerung von Verhalten einsetzt und insofern t zweimal schalten kann, ohne einen Kreis zu beinhalten. Man sieht, wie vorsichtig man in Folge dessen sein muss, wenn man kausale Argumentationen aus den klassischen Petri-Netzen auf nicht-asynchrone Netze übertragen will:

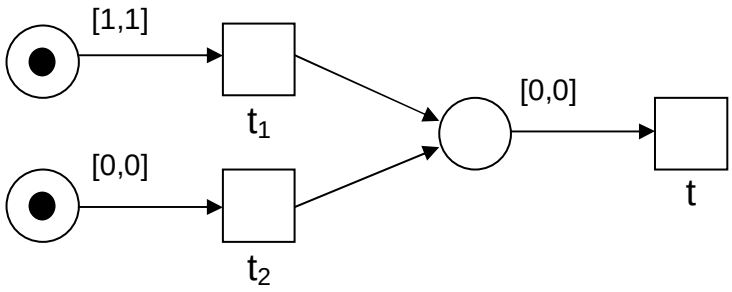



**Abbildung 9: Ein getaktetes Petri-Netz zu 1)**

Mit dem gleichen Ansatz können wir auch ein getaktetes Netz konstruieren, das 3) verletzen würde. Dies gelingt sogar in einer Version mit $t^\bullet=\varnothing$ (dargestellt in Abbildung 10 ohne den gestrichelten Teil) und einer Version mit $t^\bullet\neq\varnothing$ (mit dem gestrichelten Teil).

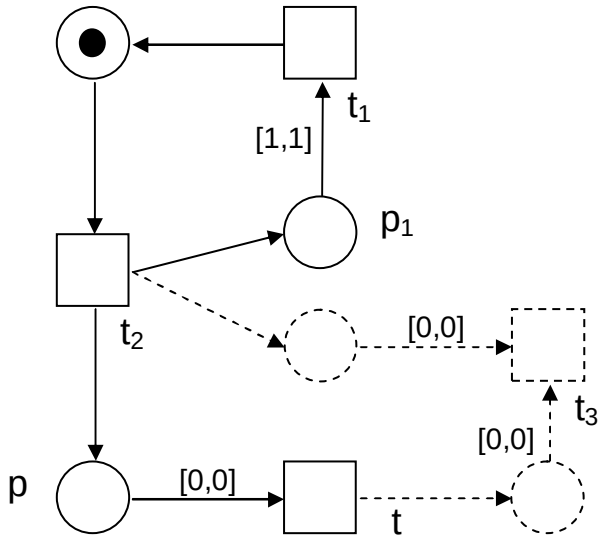



**Abbildung 10: Ein getaktetes Petri-Netz zu 3)**

Hier existiert zwar mit $t_1$ und $t_2$ ein F-Kreis, doch er benötigt keine „Rückmeldung“ von t, da auf Grund der Intervalle auf (p, t) und $(p_1, t_1)$ jedes Mal die Transition t schalten muss, bevor wieder $t_1$ schalten kann. Im gestrichelten Teil ist dann $t_3$ noch dafür verantwortlich, die durch t entstandene Marke schnell genug zu vernichten.

Es mag nun allerdings überraschend sein, dass wir trotz der Beispiele in Abbildung 9 und Abbildung 10 die Voraussetzung, dass N asynchron ist, in 1.4.9.11 *nicht* benötigen. Dies liegt daran, dass beide Netze nicht sicher im Sinne der Definition sind – obwohl in beiden zu keiner Zeit eine Transition schalten kann, deren Nachbereich nicht leer ist. Dazu rufe man sich ins Gedächtnis, dass der Begriff „sicher" über Abläufe aus FS definiert ist und daher die Intervalle gar nicht berücksichtigt.

Bemerkung: Man könnte natürlich den etwas schwächeren Begriff *r-sicher* definieren, indem man nur mehr fordert, dass in r-sicheren Netzen keine ID in RFS erreichbar ist, für die eine zeit-aktivierte Transition t eine markierte Stelle in $t^\bullet \setminus {}^\bullet t$ hat. (Offenbar wären sichere Netze auch r-sicher.) Dann wäre die Voraussetzung „asynchron" tatsächlich nötig. So aber können wir folgendes Korollar formulieren:

### *1.4.9.12 Korollar*

Sei N ein Netz ohne präautarke Transitionen. Dann gilt:

1) Läßt sich eine Transition t mindestens zweimal schalten, so existiert in N ein Kreis, der t enthält.

**Bemerkung**: Unter der Voraussetzung, dass die Hypothese 1.4.9.11 gilt, gilt dann auch:

2) Läßt sich t unendlich oft schalten, so existiert sogar ein F-Kreis, der t enthält.

**Beweis:**

Wir führen beide Aussagen wie folgt auf 1.4.9.11 zurück: Wir konstruieren das asynchrone Netz N', indem wir alle unteren Intervallschranken auf 0 setzen. Läßt sich t in N zweimal bzw. unendlich oft schalten, dann nach 1.4.2.7 in *FS(N)* und damit nach Konstruktion auch in *RFS(N')*. Weiter ist N' auch sicher, da *FS(N) = FS(N')*. Damit existiert in N' ein Kreis bzw. ein F-Kreis, welcher t enthält, und nach Konstruktion von N' existiert dieser auch in N.

□

Nun wollen wir aber Punkt 2) aus 1.4.9.11 verwenden, um folgende Aussage zu zeigen:

### *1.4.9.13 Satz*

Sei N ein Netz ohne präautarke Transitionen, dessen Netzgraph keine F-Kreise enthält, auf denen alle Transitionen intern und prezero sind. Dann hat in N keine ID einen harten refusal-timestop.

**Beweis:**

Sei im Widerspruch dazu $(\rho,w)$ ein harter refusal-timestop. Wir zerteilen w zunächst wie folgt in $w_1 \cdot w_2$: In $w_1$ sollen alle Transitionen enthalten sein, die nur endlich oft schalten und mindestens einmal jede, die in w überhaupt schaltet. Da w unendlich lang ist, ist diese Zerteilung möglich und $w_2$ ist offenbar immer noch unendlich lang.

Wir wollen die Transitionen in $w_2$ dann $T_w$ nennen. Ist $T_w$ leer, so besteht $w_2$ nur aus Verweigerungsmengen und wir erhalten einen Widerspruch. Sei $\rho'$ die ID nach $w_1$, also $\rho[w_1\rangle_r \rho'$. Dann gilt für alle von $\rho'$ erreichbaren IDs $\rho''$ und alle Transitionen $t \in T_w$, dass $\rho''({}^\bullet t) \in \{0,-\infty\}$.

Zwischenbemerkung: Man beachte, dass wir über $\rho''(t^)$ keine solche Aussage machen können, weswegen wir die Aussage des Satzes gegenüber [Bih98] auf prezero Transitionen abschwächen müssen.

Nach dem bereits gesagten müssen alle Transitionen in $T_w$ prezero sein, da sie nach Definition eines harten timestops dringend sind. Weiter enthält $T_w$ ja nur Transitionen, welche in $w_2$ unendlich oft schalten. Sei $t_\omega$ eine solche Transition.

Wir konstruieren nun N' aus N, indem wir $\rho'$ als Start-ID verwenden und alle Transitionen $T\backslash T_w$ aus N entfernen. Offenbar kann $w_2$ in N' schalten. Weiter entstehen in N' keine präautarken Transitionen und N' ist sicher.

Nun dürfen wir 1.4.9.11, 2) auf $t_\omega$ anwenden und erhalten einen F-Kreis in N'. Da N' nur prezero Transitionen enthält, besteht der F-Kreis ausschließlich aus prezero Transitionen. Nach Konstruktion war dieser F-Kreis aber auch in N vorhanden und wir erhalten einen Widerspruch.

□

### *1.4.9.14 Korollar*

Sei N ein Netz ohne präautarke Transitionen, dessen Netzgraph keine gerichteten F-Kreise enthält, auf denen alle Transitionen prezero sind. Dann ist N zeit-real.

**Beweis:**

Klar mit 1.4.9.7 und 1.4.9.8.

□

Bemerkung: Obwohl dieses Kriterium nur hinreichend ist, hat es doch den Vorteil, dass es erhalten bleibt, wenn man $N_1$ und $N_2$ über beliebige Aktionsmengen synchronisiert. Dies ist vor allem deshalb tatsächlich wichtig, weil im Allgemeinen die Synchronisation zweier zeit-realer Netze $N_1$ und $N_2$ *nicht* zeit-real sein muss.

Dies kann z.B. wie in der folgenden Abbildung 11 dadurch geschehen, dass die Transitionen, die ein Teilnetz aus einem timestop befreien könnten, in der Synchronisation nicht mehr schalten können.

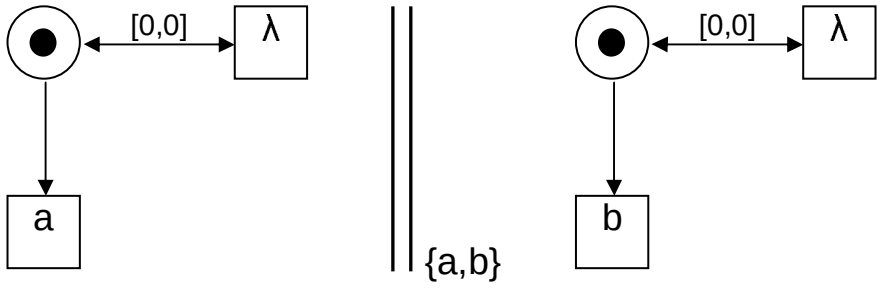



**Abbildung 11: Timelock durch Synchronisation**

Beide Teilnetze haben zwar erreichbare, harte timestops, sind aber zeit-real, da keine timelocks existieren. In der Parallelkomposition über *{a,b}* sind jedoch alle sichtbaren Transitionen tot, so dass die internen Transitionen nicht mehr aus dem timelock befreit werden können.

# Kapitel 2 Implizite Verweigerungsmengen

## *2.1 Motivation und Grundidee*

Wie wir in 1.4.6 gesehen haben, können wir durch ein Tool die RT-Semantiken (bzw. dazu äquivalente Transitionssysteme) zu vergleichender Netze berechnen. Wir sind nun natürlich daran interessiert, die entstehenden Transitionssysteme in möglichst kompakter Weise zu speichern. Wie im zweiten Teil der Arbeit in Kapitel 7 dargelegt wird, folgen auf die Berechnung des Transitionssystems jedoch noch mehrere komplexe Verarbeitungsschritte (vor allem die Konstruktion des Potenzautomaten und eben die Simulation), so dass eine kompakte Darstellung insbesondere dann von großem Wert wäre, wenn diese Schritte ebenfalls noch auf einer (ggf. abweichenden) kompakten Darstellung arbeiten könnten. Wie wir im folgenden sehen werden, ist dies möglich.

Grundlage des weiteren ist die Feststellung, dass in Transitionssystemen, die Verweigerungsmengen enthalten (siehe Tabelle 1 und Tabelle 2 in 1.4.6), die Beschriftungen der Kanten eine spezielle Struktur besitzen.

Genauer sind in $TR_r$, $TR_{rmax}$, $TR_{lr}$, $TR_{lrmax}$, $Ext( TR_{lr} )$ und $Ext( TR_{lrmax} )$ die Verweigerungsmengen auf Kanten, die vom selben Zustand ausgehen:

- halbgeordnet
- nach unten abgeschlossen
- unter Vereinigung abgeschlossen

Außerdem führen alle solche Kanten aus einem gegebenen Zustand nur in einen einzigen Zielzustand. (Daraus resultieren auch besondere Eigenschaften für $\wp( Ext( TR_{lrmax} ))$, die aber etwas tieferliegend sind.)

In der naiven Darstellung wird diese Struktur der Verweigerungsmengen nicht ausgenutzt. Schon die Anzahl der Kanten zwischen zwei Zuständen liegt damit in der Größenordnung von $O( 2^n )$, wenn n die Anzahl sichtbarer Aktionen ist.

Wir werden im folgenden einen Ansatz vorstellen, der obige Feststellungen verwendet, um Redundanzen in der Darstellung zu beseitigen, indem Kanten, auf deren Existenz implizit geschlossen werden kann, nicht mehr dargestellt werden.

Diese nahe liegende Idee führt jedoch zu Problemen, wenn wir, wie eingangs angedeutet, auch in späteren Verarbeitungsschritten solche Kanten implizit lassen wollen: Die naive Anwendung der in 1.3.1.7 vorgestellten Konstruktion des Potenzautomaten auf die komprimierte Darstellung (siehe 2.2.4, Abbildung 13) führt dazu, dass es aus einem Zustand mehrere gleichbeschriftete implizite Kanten in verschiedene Zustände geben kann. Wir benötigen aber *deterministische* Versionen der Automaten, wenn die Inklusion zweier Semantiken mittels wechselseitiger Simulation geprüft werden soll (siehe 1.3.1.6).

Wir werden deshalb unsere Potenzautomaten derart modifizieren müssen, dass sie zusätzliche Kanten enthalten (aber dennoch wesentlich weniger als die klassischen Potenzautomaten, in denen *alle* Kanten explizit sind). Weiter werden wir ihre Konstruktion direkt auf unserer komprimierten Darstellung spezifizieren, so dass keine weiteren (und evtl. größeren) Zwischenobjekte anfallen. Schließlich werden wir auch noch zeigen, wie wir den

Begriff der Simulation anpassen können, um so letzten Endes eine durchgängige Verarbeitung der komprimierten Darstellung zu erhalten, wie sie dann in unserem Tool FastAsy auch umgesetzt ist.

## *2.2 Ausnutzung von Abschlusseigenschaften*

### 2.2.1 Grundlegende Definitionen

In 1.3 wurden bereits die wichtigsten Begriffe zu Automaten eingeführt. In Ergänzung dazu seien für das gesamte Kapitel 2 noch einige Konventionen eingeführt:

Automaten werden wir mit großen lateinischen Buchstaben A, B, C, … benennen. Da Start- und Endzustände in unseren Betrachtungen nicht besonders ausgezeichnet sind (Startzustände bleiben in allen Konstruktionen erhalten und es gibt *nur* Endzustände), dürfen wir einen Automaten A einfach mit seiner Zustandsmenge und seiner Übergangsrelation $(Q_A, \delta_A)$ identifizieren. Wann immer der Kontext nicht sowieso klar ist, gehören $Q_A$ und $\delta_A$ automatisch zu A, $Q_B$ und $\delta_B$ zu B, usw.

Das Alphabet $\Lambda$ werden wir im folgenden als für alle Automaten universell ansehen. Zwar hängt $\Lambda$ von den Alphabeten betrachteter Petri-Netze ab, doch wenn diese nicht übereinstimmen, können wir, wie in 1.4.7 argumentiert wird, problemlos deren Vereinigung betrachten.

#### *2.2.1.1 Definition*

Wir nennen $\Lambda$ ein *Ref-Alphabet*, wenn es folgende Form hat: Es existiert eine endliche Menge *Act* von Aktionen und eine endliche Menge von Verweigerungsaktionen *Refact*, so dass $\Lambda = Act \cup \wp(Refact)$. Wir schreiben auch $Ref := \wp(Refact)$.

□

Wir stellen uns bei der Untersuchung eines Petri-Netzes unter *Refact* das Alphabet $\Sigma$ des Netzes vor, unter *Act* je nach Zusammenhang (siehe 2.2.1.5) die Menge der Transitionen T selbst oder ebenfalls $\Sigma$. Man beachte, dass auch in letzterem Fall $a \in Act$ und $\{a\} \in Ref$ dabei nicht identifiziert werden.

#### *2.2.1.2 Definition*

Für Zustände $p, q \in Q$, eine Aktion $a \in \Lambda$ und eine Übergangsrelation $\delta$ seien folgende Schreibweisen definiert:

- $Acts(q) := \{ a \in Act \mid \delta(q, a) \neq \emptyset \}$
- $Acts(q, p) := \{ a \in Act \mid (q, a, p) \in \delta \}$
- $Refs(q) := \{ a \in Ref \mid \delta(q, a) \neq \emptyset \}$
- $Refs(q, p) := \{ a \in Ref \mid (q, a, p) \in \delta \}$.

□

#### *2.2.1.3 Definition*

Sei A ein endlicher Automat.

- A heißt *nach unten abgeschlossen*, wenn:
  $\forall q, q' \in Q, \forall X, Y \in Ref: (q, X, q') \in \delta \wedge Y \subseteq X \Rightarrow (q, Y, q') \in \delta$
- A heißt *unter Vereinigung abgeschlossen*, wenn:
  $\forall q, q' \in Q, \forall X, Y \in Ref: (q, X, q') \in \delta \wedge (q, Y, q') \in \delta \Rightarrow (q, X \cup Y, q') \in \delta$

- A heißt *abgeschlossen*, wenn A unter Vereinigung und nach unten abgeschlossen ist.
- A heißt *Ref-kohärent*, wenn:
  $\forall q, q', q'' \in Q, \forall X, Y \in Ref: (q, X, q') \in \delta \wedge (q, Y, q'') \in \delta \Rightarrow q' = q''$
- A heißt *Ref-unär*, wenn A Ref-kohärent ist und zusätzlich gilt:
  $\forall q \in Q: |Refs(q)| \leq 1$

□

#### *2.2.1.4 Definition*

Wenn A unter Vereinigung abgeschlossen ist, ist für $p,q \in Q$ durch den Ausdruck $max(Refs(q,p))$ eine eindeutige Menge aus Ref definiert, falls $Refs(q,p) \neq \emptyset$. In diesem Fall schreiben wir dafür auch $Maxref(q, p)$.

Per Konvention schreiben wir weiter $Maxref(q, p) = \perp$, falls $Refs(q,p) = \emptyset$.

Ist A zusätzlich Ref-kohärent, so dürfen wir auch $Maxref(q)$ schreiben und meinen damit die eindeutige Menge $max(Refs(q))$ bzw. eben $\perp$.

□

Bemerkung: Da Funktionen wie *Refs*, *Acts* und *Maxref* natürlich von der Zustandsübergangsrelation des jeweiligen Automaten abhängen, werden wir zur Vermeidung von Zweideutigkeiten die Funktionsnamen gelegentlich mit dem Namen des Automaten indizieren.

#### *2.2.1.5 Satz*

Sei N ein PTT-Netz. Dann gilt:

- $\Sigma \cup \wp(\Sigma)$ und $T \cup \wp(\Sigma)$ sind Ref-Alphabete.
- $TR_{lrmax}(N)$ und $TR_{rmax}(N)$ sind abgeschlossen und Ref-kohärent.
- $TR_{rmax}(N)$ ist dabei noch buchstabierend.
- $Ext(TR_{lrmax}(N))$ ist buchstabierend, abgeschlossen und Ref-kohärent.

□

Dieser Satz zeigt uns zunächst die Anwendbarkeit des in diesem Kapitel gesagten auf verschiedene Varianten von r-Erreichbarkeitsgraphen. Obwohl die weiteren Ausführungen allgemein sind, können wir uns dabei im Rahmen dieser Arbeit immer einen dieser Anwendungsfälle vorstellen.

Im folgenden sei $\Lambda$ immer ein Ref-Alphabet.

### 2.2.2 NdeA-Kompression

Wir definieren zunächst zwei Funktionen *Cmpr* und *Decmpr* auf Automaten, die das Eliminieren von implizierbaren Kanten bzw. deren Rekonstruktion bewirken sollen.

#### *2.2.2.1 Definition*

Es sei A ein Automat, zu dem wir wie folgt einen weiteren Automaten C konstruieren:

Es sei $Q_C = Q_A$ .

Es sei $\delta_C \subseteq Q_A \times \Lambda \cup \{\lambda\} \times Q_A$ definiert durch:

i) Für $a \in Act \cup \{\lambda\}$ und $p, q \in Q_A$ gilt:
$(p, a, q) \in \delta_C :\Leftrightarrow (p, a, q) \in \delta_A$

ii) Für $X \in Ref$ und $p, q \in Q_A$ gilt:
$(p, X, q) \in \delta_C \;:\Leftrightarrow\; [(p, X, q) \in \delta_A \wedge (\forall Y \in Ref: (p, Y, q) \in \delta_A \Rightarrow Y \subseteq X)]$

Wenn A unter Vereinigung abgeschlossen ist, so können wir ii) außerdem mit der Notation aus 2.2.1.4 wie folgt schreiben:

iib) Für $X \in Ref$ und $p, q \in Q$ gilt:
$(p, X, q) \in Cmpr(\delta_A) \;\Leftrightarrow\; Maxref_A(p, q) = X \neq \perp$.

Wir schreiben $\delta_C = Cmpr(\delta_A)$ bzw. $C = Cmpr(A)$ und nennen C die *Kompression* von A.

□

Bemerkung: Wir haben *Cmpr()* auf allgemeinen Automaten definiert und Sätze wie 2.2.2.3 oder 2.2.3.1 gelten in der Tat allgemein. Wie wir in 2.2.2.4 sehen werden, führt der Begriff der Kompression aber nur für abgeschlossene Automaten auf das beabsichtigte Resultat.

### *2.2.2.2 Definition*

Es sei C ein Automat, zu dem wir wie folgt einen weiteren Automaten D konstruieren:
Es sei $Q_D = Q_C$ und es sei $\delta_D \subseteq Q_C \times \Lambda \cup \{\lambda\} \times Q_C$ definiert durch:

i) Für $a \in Act$ und $p, q \in Q_C$ gilt:
$(p, a, q) \in \delta_D \;:\Leftrightarrow\; (p, a, q) \in \delta_C$

ii) Für $Y \in Ref$ und $p, q \in Q_C$ gilt:
$(p, Y, q) \in \delta_D \;\Leftrightarrow\; [\exists X \in Ref: (p, X, q) \in \delta_C \wedge Y \subseteq X]$

Wir schreiben $\delta_D = Decmpr(\delta_C)$ bzw. $D = Decmpr(C)$ und nennen D die *Dekompression* von C.

□

Wie wir sehen, tasten beide Konstruktionen den Anteil der mit Aktionen aus *Act* beschrifteten Übergänge nicht an.

### *2.2.2.3 Lemma*

Die Kompression C eines Ref-kohärenten Automaten A ist Ref-unär. Insbesondere ist C selbst wieder Ref-kohärent.

**Beweis:**

C ist Ref-kohärent wegen $\delta_C \subseteq \delta_A$. Nach 2.2.2.1 ii) kommt für eine Kante auch nur ein einziger Kandidat in Frage.

□

Bemerkung: Wir werden diese Tatsache in 2.2.4.4 verwenden, um den Aufwand für die Konstruktion des Potenzautomaten drastisch zu reduzieren.

### *2.2.2.4 Satz*

Sei C die Kompression eines Automaten A, dann gilt:

i) Ist A unter Vereinigung abgeschlossen, so gilt
$\forall (p, Y, q) \in \delta_A \; \exists X \in Ref: X \supseteq Y \wedge (p, X, q) \in \delta_C$.

ii) Ist A nach unten abgeschlossen, so gilt
$\forall (p, X, q) \in \delta_C, Y \subseteq X: (p, Y, q) \in \delta_A$.

iii) Ist A also abgeschlossen, so gilt
$A = Decmp( Cmpr( A ) )$.

**Beweis:**

„*i)*“:

Da A unter Vereinigung abgeschlossen ist, existiert zunächst $X \in Ref$ mit $(p, X, q) \in \delta_A$, so dass für beliebige $(p, Y, q) \in \delta_A$ gilt: $X \supseteq Y$. Nach 2.2.2.1 ii) ist dann aber auch $(p, X, q) \in \delta_C$.

„*ii)*“:

Bezüglich der zweiten Aussage gilt zunächst wegen $\delta_C \subseteq \delta_A$, dass $(p, X, q) \in \delta_A$. Nach Definition der Abgeschlossenheit nach unten ist dann jedes $Y \subseteq X$ Element von $Refs_A(p, q)$.

„*iii)*“:

Aus i) und 2.2.2.2 ii) folgt sofort, dass $\delta_A \subseteq Decmp( \delta_C )$. Aus 2.2.2.2 ii) und ii) folgt $\delta_A \supseteq Decmp( \delta_C )$.

□

Wie wir sehen, können wir also einen abgeschlossenen Automaten über *Cmpr* reduzieren und über *Decmp* verlustfrei wieder rekonstruieren.

## 2.2.3 Späte Elimination interner Übergänge

Obwohl $Ext( TR_{lrmax}( N ))$ buchstabierend ist, werden wir in einem Tool, das für Netze einen r- Erreichbarkeitsgraphen per *token game* berechnet, im Allgemeinen nicht direkt eine buchstabierende Variante, sondern zunächst $TR_{lrmax}( N )$ erhalten. Im Anschluß an 1.3.1.8 wurde aber bereits bemerkt, dass die Konstruktion des Potenzautomaten im Allgemeinen nur auf buchstabierenden Automaten zu einem deterministischen Automaten führt. Da wir ja als Endergebnis an einem deterministischen Automaten interessiert sind, resultiert zunächst für das folgende, dass wir wie in 1.3.1.9 beschrieben aus einem nicht-buchstabierenden Automaten A die internen Übergänge eliminieren müssten, bevor wir ihn komprimieren dürfen. Da wir aber, wie wir sehen werden, die Kompression der nicht-buchstabierenden Version sehr effizient direkt aus dem *token game* gewinnen können, wäre dies ein Umweg. Der folgende Satz zeigt aber, dass wir auch erst aus der so erhaltenen Kompression die λ-Übergänge eliminieren können:

### *2.2.3.1 Satz*

Für einen Automaten A kommutieren Kompression und Bildung der Externalisierung:

$$Cmpr(EA( A )) = EA(Cmpr( A )) .$$

**Beweis:**

Offensichtlich weicht höchstens die Zustandsübergangsrelation der beiden Automaten ab. Für den Beweis benennen wir die verschiedenen Relationen wie folgt: $\delta_{CE}$ sei diejenige von $Cmpr(EA( A ))$, $\delta_E$ die von $EA( A )$, $\delta_{EC}$ die von $EA(Cmpr( A ))$ und schließlich $\delta_C$ die von $Cmpr( A )$. Wir zeigen nun $\delta_{CE} = \delta_{EC}$ :

‘$\subseteq$’:

Sei $(p, \varepsilon, q) \in \delta_{CE}$. Nach Konstruktion 2.2.2.1 gilt sofort auch $(p, \varepsilon, q) \in \delta_E$ und nach 1.3.1.9 existieren Zustände $p'$, $q'$ mit $p \overset{\lambda}{\rightarrow\!\!\!\!\rightarrow} p' \overset{\varepsilon}{\rightarrow} q' \overset{\lambda}{\rightarrow\!\!\!\!\rightarrow} q$ in A, außerdem folgt $\varepsilon \neq \lambda$.

Falls nun $a:=\varepsilon\in Act$, so gilt $p \overset{\lambda}{\twoheadrightarrow} p' \overset{a}{\rightarrow} q' \overset{\lambda}{\twoheadrightarrow} q$ automatisch auch in *Cmpr(A)*. Falls $X:=\varepsilon\in Ref$, so gilt nach 2.2.2.1 ja zusätzlich $\forall Y\in Ref: (p, Y, q) \in \delta_E \Rightarrow Y \subseteq X$. Nach Konstruktion der Externalisierung gilt damit aber auch $\forall Y\in Ref: (p', Y, q') \in \delta_A \Rightarrow Y \subseteq X$.

Damit gilt in beiden Fällen $p' \overset{\varepsilon}{\rightarrow} q'$ in Cmpr(A), und $p \overset{\lambda}{\twoheadrightarrow} p'$ und $q' \overset{\lambda}{\twoheadrightarrow} q$ gelten dort sowieso offensichtlich. Wieder nach 1.3.1.9 gilt also auch $(p, \varepsilon, q) \in \delta_{EC}$.

‚⊇':

zeigt man analog.

□

Laut diesem Satz dürfen wir also Kompression und Elimination der λ-Kanten in beliebiger Reihenfolge durchführen bzw., wie dies in unserem Tool der Fall ist, die Externalisierung auf einem r-Erreichbarkeitsgraphen mit implizit gelassenen Verweigerungsmengen (genauer: auf $Cmpr(\ TR_{lrmax}\ )$ )bilden, aber so tun, als ob es die Kompression der Externalisierung wäre.

Mit diesem Resultat werden wir o.B.d.A. im Rest von Kapitel 2 davon ausgehen, dass wir nur buchstabierende Automaten behandeln. (Man beachte auch, dass etwa $TR_{rmax}$, für welches das folgende ebenfalls anwendbar ist, nach Konstruktion bereits buchstabierend ist.)

## 2.2.4 Kompression des Potenzautomaten

Mit der NdeA-Kompression Cmpr und ihrer Umkehrung Decmp haben wir bereits die Möglichkeit geschaffen, die r-Erreichbarkeitsgraphen kompakter zu speichern. Wir können daraus jedoch wie bereits angedeutet noch weitaus größere Vorteile bzgl. Speicherbedarf und Verarbeitungsgeschwindigkeit gewinnen, wenn wir die Konstruktion des Potzenautomaten derart auf der komprimierten Version durchführen können, dass eine komprimierte Version des Potenzautomaten entsteht. Genau das werden wir im folgenden zeigen, wobei allerdings der Kompressionsbegriff beim Potenzautomaten von 2.2.2.1 abweicht. Dies ist jedoch nicht weiter verwunderlich, da natürlich die Konstruktion in 1.3.1.7 die Abgeschlossenheit nicht erhalten kann:

Man betrachte dazu das Beispiel in Abbildung 12: Der Multizustand *Q2* besteht aus einer Kombination zweier Einzelzustände *p21* und *p22*. Nun genügen die Kanten von *p21* nach *p31* und *p22* nach *p32* zwar für sich gesehen der Abgeschlossenheitseigenschaft; durch die Kombination in *Q2* jedoch geht die Eigenschaft in *P(A)* verloren, da über die mit *{b}* und *{}* beschrifteten Kanten ja *p31* und *p32* gleichzeitig erreicht werden können und nach Konstruktion diese Kanten in einen neuen Multizustand *Q31+32* laufen. Damit werden die Verweigerungsmengen *{b}* und *{}* gewissermaßen aus der Abgeschlossenheit herausgerissen.

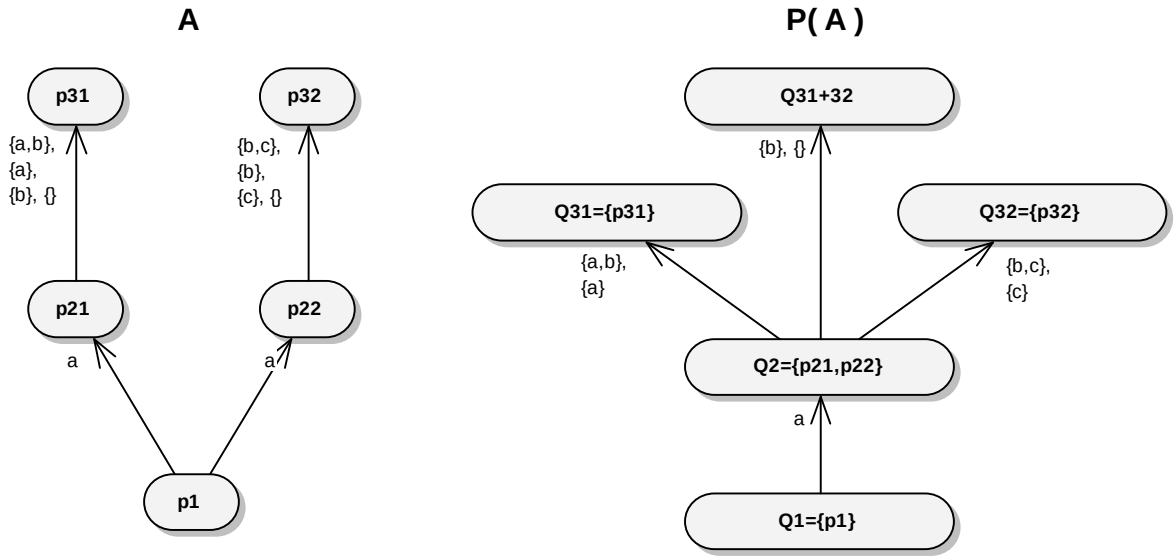



**Abbildung 12: Fehlerhafte Konstruktion I**

Wir wollen nun zunächst zeigen, dass dies nicht völlig willkürlich erfolgt und sich eine abgeschwächte Form der Abgeschlossenheit auch im Potenzautomaten noch erhält. Es stellt sich heraus, dass sich obiges Beispiel tatsächlich verallgemeinern lässt und wir zwar unter den Kanten zwischen zwei fest gewählten Zuständen keine Abgeschlossenheit nach unten mehr haben, aber immerhin noch, wenn wir stattdessen für einen Zustand alle seine ausgehenden Kanten betrachten.

### *2.2.4.1 Definition*

Sei PA ein endlicher Automat. PA heißt *schwach nach unten abgeschlossen*, wenn:

$$\forall P \in Q_{PA},\ \forall X, Y \in Ref:\quad X \in Refs_{PA}(P) \wedge Y \subseteq X \Rightarrow Y \in Refs_{PA}(P)\ .$$

□

### *2.2.4.2 Lemma*

Der Potenzautomat PA eines nach unten abgeschlossenen Automaten A ist schwach nach unten abgeschlossen.

**Beweis:**

Seien $P, P', X, Y$ fest gewählt mit $X \in Refs_{PA}(P, P')$ und $Y \subseteq X$. Dann existieren nach Definition die A-Zustände $p \in P$ und $p' \in P'$ mit $X \in Refs_A(p, p')$. Da A nach unten abgeschlossen ist, gilt aber auch $Y \in Refs_A(p, p')$. Mit anderen Worten ist $p' \in \delta_A(p, Y)$ und somit existiert ein P'' mit $(P, Y, P'') \in \delta_{PA}$ und $p' \in P''$, insbesondere also $Y \in Refs_{PA}(P)$.

□

Da obiger Automat PA nach Konstruktion deterministisch ist, gilt offenbar sogar:

$$\forall P, P' \in Q_{PA},\ \forall X, Y \in Ref:\quad (P, X, P') \in \delta_{PA} \wedge Y \subseteq X$$
$$\Rightarrow\ \exists_1 P'' \in Q_{PA}: Y \in Refs_{PA}(P, P'')\ .$$

Weiter gilt noch $P' \subseteq P''$, denn wegen der Abgeschlossenheit von A nach unten:

$$\forall p \in P\ \forall Y \subseteq X: \delta_A(p, X) \subseteq \delta_A(p, Y)\ .$$

Wir nennen die Tatsache, dass *kleinere* Verweigerungsmengen immer auf *größere* (oder gleichgroße) Zustände führen, *Kontravarianz*.

Die Aussage 2.2.4.2 bezog sich nun auf den klassischen Potenzautomaten. Wir wollen die Potenzautomatenkonstruktion aber ja verwenden, um aus einem komprimierten NdeA einen in gewissem Sinne äquivalenten komprimierten DeA zu konstruieren. Wie wir gleich sehen werden, müssen wir dazu von der klassischen Konstruktion des Potenzautomaten abweichen. Dabei wird aber die Beobachtung aus obigem Lemmas eine Schlüsselrolle bei der Modifikation der Potenzautomatenkonstruktion spielen.

Unser nächstes Ziel ist es also, einen modifizierten Potenzautomatenbegriff $\wp_{\text{mod}}$ und eine weitere Dekompressionsoperation *P-Decmp* einzuführen, so dass folgendes gilt:

$$P\text{-}Decmp(\ \wp_{mod}(\ Cmpr(\ \delta_A\ )\ )\ ) = \wp(\ \delta_A\ )\ .$$

Um die Verständlichkeit des entsprechenden Beweises zu fördern, werden wir die in den folgenden Definitionen und Sätzen vorkommenden Automaten gleich so benennen, wie die jeweiligen Zwischenobjekte später im Beweis des Theorems heißen:

$PA := \wp(\ A\ )$
$C = Cmpr(\ A\ )$
$PM = \wp_{mod}(\ C\ )$
$D = P\text{-}Decmp(\ C\ )$

Überlegen wir nun zunächst, warum die übliche Vorgehensweise nach 1.3.1.7 auf die Kompression eines (abgeschlossenen) Automaten angewendet kein korrektes Ergebnis liefert:

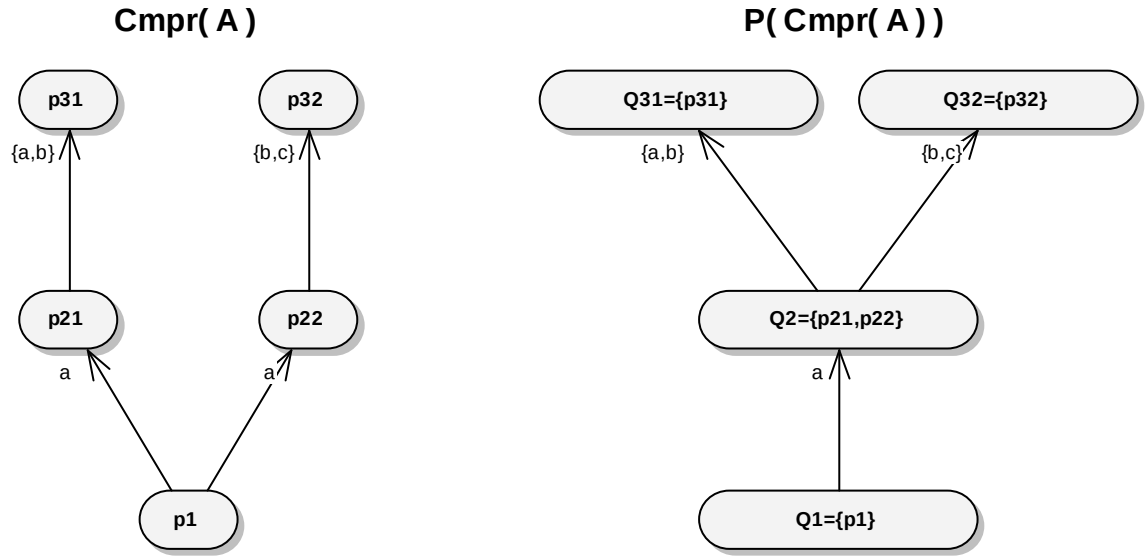



**Abbildung 13: Fehlerhafte Konstruktion II**

Wie in Abbildung 13 dargestellt, enthält $P := \wp(\ Cmpr(\ A\ )\ )$ zwar aus dem kombinierten Zustand *Q2* heraus die Übergänge für *{a,b}* und *{b,c}*, jedoch nicht den weiteren Übergang *{b}* in einen Zustand *{p31,p32}*, den wir zumindest erwarten würden. Es ist natürlich an diesem Punkt noch nicht ganz klar, wie wir überhaupt den entstehenden

Potenzautomaten zu interpretieren haben. Wir können jedoch schon sehen, dass wir weder mit der klassischen Interpretation noch mit *Decmp* arbeiten werden können. (In ersterem Fall würden wir die impliziten Mengen verlieren, in letzterem Fall wäre *P* nicht-deterministisch.)

Die Idee ist nun, die Konstruktion derart zu modifizieren, dass man jeweils die ausgehenden Kanten aller Einzelzustände eines Multizustands gemeinsam betrachtet. Zusätzlich zu allen deren mengenwertigen Kantenbeschriftungen nehmen wir auch Kanten für alle Schnitte dieser Mengen auf. Das Ziel dieser neuen Kanten bestimmen wir genau wie im klassischen Algorithmus.

Die Interpretation (welche formal in der Dekompressionsoperation ihren Ausdruck findet) werden wir dann derart anpassen, dass wir für eine auszuführende Aktion in Gestalt einer Verweigerungsmenge jeweils die speziellste passende Kante (also die kleinste dieser Schnittmengen, die die auszuführende Verweigerungsmenge noch enthält) als die gültige betrachten.

### *2.2.4.3 Definition*

Für einen Automaten C sei der *modifizierte Potenzautomat* PM := $\wp_{\text{mod}}$( C ) definiert wie in 1.3.1.7, außer dass wir $\delta_{PM}$ (für den Ref-Anteil) abweichend definieren:

Für $P \in \wp(Q)$ sei zunächst

$$\mathfrak{X} := \cup_{o \in P} Refs_C(o),$$
$$\mathfrak{A} := \cup_{o \in P} Acts_C(o),$$

weiter

$$\mathfrak{X}^{\cap} := \{ X_1 \cap ... \cap X_n \mid n \geq 1, X_i \in \mathfrak{X} \}$$

und damit

$$\delta_{PM}(P) := \{ (a, P') \mid a \in \mathfrak{A}, P' = \cup_{o \in P} \delta_C(o, a) \neq \emptyset \}$$
$$\cup \{ (X, P') \mid X \in \mathfrak{X}^{\cap}, P' = \cup_{o \in P} \cup_{M \in Ref:\, M \supseteq X} \delta_C(o, M) \neq \emptyset \}.$$

□

Bemerkung: Genau genommen steckt in der Vereinigung über alle $M \in Ref: M \supseteq X$ eine Art implizite Dekompression von C. Allerdings wird C später laut 2.2.2.3 Ref-unär sein, sodass die Vereinigung unnötig ist und wir stattdessen für jedes o genau die Menge angeben können, die für den Übergang aus o verantwortlich ist:

### *2.2.4.4 Lemma*

Falls C Ref-unär, gilt mit den Notationen aus 2.2.4.3 für festes *P* und ein $X \in Ref$:

Sei $O(P,X) = \{ o \in P \mid \exists M \in Refs_C(o): M \supseteq X \}$. Dann existiert für jedes $o \in O(P,X)$ genau ein $M \in Refs_C(o)$ mit $M \supseteq X$, welches wir im folgenden mit $X^o$ bezeichnen wollen. Insbesondere ist $X^o \in \mathfrak{X}$ und es gilt $X^o \supseteq X$.

Weiter gilt:

$$\delta_{PM}(P) := \{ (a, P') \mid a \in \mathfrak{A}, P' = \cup_{o \in P} \delta_C(o, a) \neq \emptyset \}$$
$$\cup \{ (X, P') \mid X \in \mathfrak{X}^{\cap}, P' = \cup_{o \in O(P,X)} \delta_C(o, X^o) \neq \emptyset \}.$$

**Beweis:**

*O(P,X)* stellt also die Menge der Einzelzustände von P dar, von denen Kanten ausgehen, die für eine gegebene Verweigerungsmenge X überhaupt als Übergänge in Frage kommen. Für jeden von diesen Zuständen ist (da Teil eines Ref-unären Automaten) klar, dass nur eine einzige ausgehende Kante mit einer Menge beschriftet ist und nach Konstruktion von *O(P, X)* existiert dieses $X^o$ auch wirklich und ist Obermenge von *X*.

Die restliche Aussage folgt dann durch Einsetzen.

□

### *2.2.4.5 Lemma*

Mit den Voraussetzungen und Notationen aus 2.2.4.4 und wiederum für festes P und beliebiges $X \in \mathfrak{X}^{\cap}$ gilt:

$$X = \bigcap_{o \in O(P, X)} X^o .$$

**Beweis:**

‚$\subseteq$‘:

klar, da für alle $o \in O(P, X)$ gilt $X \subseteq X^o$.

‚$\supseteq$‘:

Da $X \in \mathfrak{X}^{\cap}$, hat es für geeignetes n und geeignete $X_1,...,X_n \in \mathfrak{X}$ die Form $X = X_1 \cap ... \cap X_n$. Für jedes $i=1..n$ aber existiert ein $o \in P$ mit $Refs_C(o) = \{ X_i \}$. Da $X_i \supseteq X$, existieren sogar $o \in O(P, X)$ mit dieser Eigenschaft. Im Ganzen also: $\forall i=1..n \; \exists o \in O(P, X): X_i = X^o$, und es folgt die Behauptung.

□

### *2.2.4.6 Lemma*

Für einen Automaten *C* und $PM := \wp_{mod}(C)$ gilt: PM ist deterministisch.

**Beweis:**

klar nach 2.2.4.3.

□

Obige Aussage alleine ist aber noch nicht von direktem Nutzen für uns, denn im modifizierten Potenzautomaten bleiben ja, ähnlich wie bei der Kompression eines NdeA, noch Kanten implizit. Wir benötigen also nun die Vorschrift, die die impliziten Kanten wieder zum Vorschein bringt, und müssen dann zeigen, dass PM auch mit diesen Kanten noch deterministisch ist.

### *2.2.4.7 Definition*

Für einen endlichen Automaten PM definieren wir den Automat $D := P\text{-}Decmp(PM)$ und seine Zustandsübergangsrelation $\delta_D$ wie folgt:

Es gelte $Q_D := Q_{PM}$ und für $P \in Q_D$ sei

$$\begin{aligned} \delta_D(P) := & \{ (a, P') \mid a \in Acts_{PM}(P, P') \} \\ & \cup \{ (Y, P') \mid Y \in Ref \\ & \quad \wedge (\exists X \in Refs_{PM}(P, P'): Y \subseteq X) \\ & \quad \wedge (\forall Z \in Refs_{PM}(P): Y \subseteq Z \Rightarrow X \subseteq Z) \} . \end{aligned}$$

Wir nennen D eine *P-Dekompression* von PM.

□

Wir stellen also in D genau die in 2.2.4.2 am klassischen Potenzautomaten bemerkte schwache Abschlusseigenschaft nach unten her, indem wir für eine Ref-Menge Y, für die in PM keine Kante vorhanden ist, aber eine für eine Obermenge X von Y, eine Kante zum Zielzustand der X-Kante hinzufügen. Um bei mehreren Kandidaten für X das Ziel dieser neuen Kante eindeutig zu bestimmen, verlangen wir außerdem, dass es nicht eine beliebige Obermenge X von Y ist, sondern die kleinste. Diese existiert und ist eindeutig, da alle Kandidaten ja aus einem für P festen $\mathcal{X}^{\cap}$ stammen (siehe 2.2.4.3). (Die oben erwähnte Kontravarianz sagt uns, dass wir mit dieser kleinsten Obermenge in den größten Multizustand laufen.)

Wir bemerken zunächst, dass die P-Dekompression ihren Namen zu Recht trägt in folgendem Sinne:

### *2.2.4.8 Lemma*

Sei PM ein endlicher Automat und D *:= P-Decmp( PM )* seine P-Dekompression. Dann gilt:

$$\delta_{PM} \subseteq \delta_D \, .$$

**Beweis:**

Offenbar ist in Definition 2.2.4.7 jedes X mit $X \in Refs_{PM}( P, P' )$ auch selbst wieder enthalten.

□

### *2.2.4.9 Lemma*

Für einen deterministischen Automaten PM ist *D: = P-Decmp( PM )* deterministisch.

**Beweis:**

Für $\delta_D( P, a )$ mit $a \in Acts_D( P )$ ist der Fall sowieso klar. Offenbar ist weiter für festes P und Y die Wahl von X in 2.2.4.7, falls es existiert, eindeutig. Weil PM deterministisch ist, besteht aber auch bezüglich P‘ keine Wahlmöglichkeit mehr und *P-Decmp( PM )* ist somit selbst deterministisch.

□

### *2.2.4.10 Theorem*

Für einen abgeschlossenen und Ref-kohärenten Automaten A gilt:

$$P\text{-}Decmp( \wp_{mod}( Cmpr( \delta_A ) ) ) = \wp( \delta_A ) \, .$$

**Beweis:**

Wir wollen zunächst die Zwischenergebnisse wie angekündigt wie folgt benennen:

PA := $\wp$( A )
C = Cmpr( A )
PM = $\wp_{mod}$( C )
D = P-Decmp( C )

Für die Zustandsräume gilt dann:

$Q_{PA} = Q_D = Q_{PM} = \wp( Q_A )$ *und* $Q_C = Q_A$ .

Für mit $a \in Act$ beschriftete Übergänge ist die Behauptung offensichtlich, da sie durch *P-Decmp* und *Cmpr* unverändert bleiben und ihre Behandlung in $\wp$ und $\wp_{mod}$ identisch ist. Wir beschränken uns also auf die Untersuchung von Übergängen mit Beschriftungen aus *Ref*.

Nach 2.2.2.3 ist zunächst C Ref-unär, nach 2.2.4.6 ist PM und nach 2.2.4.9 weiter D deterministisch. Im folgenden Beweis werden wir außerdem die Notationen $\mathcal{X}^*$ und $\mathcal{X}^\cap$ aus 2.2.4.3 sowie *O(P, X)* und $X^o$ aus 2.2.4.4 verwenden.

Wir werden für die $\subseteq$-Richtung ausgehend von einem festen Übergang in D seine „Ursachen“ durch PM und C verfolgen, um schließlich zu sehen, was wir daraus an Kenntnis über korrespondierende Übergänge in A erlangen können. Damit werden wir zeigen, dass der entsprechende Übergang auch in PA existiert.

Umgekehrt werden wir anschließend für einen Übergang aus PA die entsprechenden einzelnen Übergänge in A betrachten und aus diesen Übergänge in C und dann in PM ableiten, bis wir sehen werden, dass der ursprüngliche Übergang aus PA auch in D existieren muss.

Diese einzelnen Beweisschritte werden wir im folgenden zum klareren Verständnis auch explizit kennzeichnen.

‚$\subseteq$‘:

Sei $P, P' \in Q_{PM}$ und $Y \in Ref$ mit $( P, Y, P' ) \in \delta_D$.

*Von D nach PM*:

Nach Definition von D gelten also:

$\exists X \in Refs_{PM}( P, P' ): Y \subseteq X$ und
$\forall Z \in Refs_{PM}( P ): Y \subseteq Z \Rightarrow X \subseteq Z$ . (*)

Wegen der zweiten Bedingung ist dieses X auch schon eindeutig bestimmt. (Außerdem werden wir diese Minimalität von X am Ende des Beweises nochmals benötigen.)

Wir erhalten also:

$( P, X, P' ) \in \delta_{PM}$ .

*Von PM nach C*:

Es sei nun wie gehabt $\mathcal{X}^* := \cup_{p \in P} Refs_C( p )$, weiter $\mathcal{X}^\cap := \{ X_1 \cap ... \cap X_n \mid n \geq 1, X_i \in \mathcal{X}^* \}$. Nach der Konstruktion von PM in 2.2.4.3 muss gelten $X \in \mathcal{X}^\cap$. Es sei $O := O( P, X )$, d.h. die Menge der Einzelzustände aus P, die zu dem X-Übergang beitragen. Nach Lemma 2.2.4.4 können wir dann schreiben:

$$\cup_{o \in O}\ \delta_C( o, X^o ) = P' .$$

*Von C nach A*:

Damit wissen wir wegen $\delta_C \subseteq \delta_A$ auch schon

$$\cup_{o \in O}\ \delta_A( o, X^o ) \supseteq P' .$$

Nun gilt nach Konstruktion von O, dass $\forall o \in O: \delta_C( o, X^o ) \neq \varnothing$ und damit wegen $\delta_C \subseteq \delta_A$ auch $\forall o \in O: \delta_A( o, X^o ) \neq \varnothing$. Damit und aus der Ref-Kohärenz von A erhalten wir aus der obigen Zeile dann sogar

$$\cup_{o \in O}\ \delta_A( o, X^o ) = P' .$$

Dieses Ergebnis ist aber leider noch in zweifacher Hinsicht von C abhängig, nämlich durch O und durch $X^o$. Beide Abhängigkeiten können wir jedoch eliminieren:

Für die $X^o$ gilt ja nach deren Konstruktion in 2.2.4.4, dass $X \subseteq X^o$. Wegen der Abgeschlossenheit von A nach unten folgt also:

$$\cup_{o \in O}\ \delta_A( o, X ) \supseteq P' .$$

Es folgt mit der bereits oben erwähnten Tatsache, dass $\forall o \in O: \delta_A( o, X^o ) \neq \varnothing$ und der Ref-Kohärenz von A dann:

$$\cup_{o \in O}\ \delta_A( o, X ) = P' .$$

Nun gilt ja aber nicht nur $C = Cmpr( A )$, sondern nach 2.2.2.4 auch $A = Decmp( C )$, und damit für O laut 2.2.2.2:

$$O = \{ o \in P \mid \exists M \in Refs_C( o ): M \supseteq X \} = \{ o \in P \mid \delta_A( o, X ) \neq \varnothing \} .$$

Damit können wir nun also ohne Abhängigkeiten von C schreiben:

$$\cup_{p \in P}\ \delta_A( p, X ) = P' .$$

*Von A nach PA*:

Mit der obigen Formulierung ist nach 1.3.1.7 dieser Schritt natürlich trivial, wir erhalten einfach:

$$\delta_{PA}( P, X ) = P' .$$

Allerdings sind wir damit noch nicht am Ende, denn wir erinnern uns, dass wir eigentlich ja $\delta_{PA}( P, Y ) = P'$ zeigen wollen. Immerhin liefert uns aber die Kontravarianz (siehe Bemerkung nach 2.2.4.2) schon:

$$\delta_{PA}( P, Y ) \supseteq \delta_{PA}( P, X ) = P' .$$

Es bleibt also z.Z., dass $\delta_{PA}( P, X ) \supseteq \delta_{PA}( P, Y )$, oder mit anderen Worten, dass in A nicht mit Y weitere Übergänge möglich wären, die mit dem größeren X nicht erlaubt sind.

Nehmen wir also im Widerspruch dazu an, es gäbe $p \in P$, so dass $\delta_A( p, X ) \neq \delta_A( p, Y )$. Wegen der Ref-Kohärenz und Abgeschlossenheit von A hieße das auch schon

$$\exists p' \in Q_A: \delta_A( p, Y ) = \{ p' \} \wedge \delta_A( p, X ) = \varnothing .$$

(Genauer muss natürlich sogar $p \in P \setminus O$ gelten, was aber im folgenden keine Rolle spielt.) Dann existiert laut 2.2.1.4 eine Menge $Y' := Maxrefs_A( p, p' )$, und diese hat nach ihrer Definition die Eigenschaften $Y \subseteq Y'$ und $\delta_A( p, Y' ) = \{ p' \}$. Nach 2.2.2.1 gilt dann aber auch in C, dass

$$\delta_C( p, Y' ) = \{ p' \} .$$

Wegen $Y' \in Refs_C( p )$ gilt dann aber $Y' \in \mathfrak{X}$, folglich nach der Konstruktion von PM in 2.2.4.3 auch $Y' \in Refs_{PM}( P )$, und wegen der Bedingung (*) von oben schließlich

$$X \subseteq Y' .$$

Aus der Abgeschlossenheit von A nach unten folgt damit letztendlich

$$p' \in \delta_A( p, X )$$

und wir haben $\delta_A( p, X ) = \emptyset$ widerlegt.

‚$\supseteq$':

Sei $P, P' \in Q_{PA}$ und $Y \in Ref$ mit $( P, Y, P' ) \in \delta_{PA}$.

*Von PA nach A*:

Es gilt nach Definition 1.3.1.7:

$$P' = \{ q \in Q_A \mid \exists p \in P: ( p, Y, q ) \in \delta_A \} .$$

*Von A nach C*:

Wenden wir auf obige Zeile nun 2.2.2.4 an, so liefert die Aussage i) des Satzes genau ‚$\subseteq$' und Aussage ii) weiter ‚$\supseteq$' für folgende Gleichheit:

$$P' = \{ q \in Q_C \mid \exists p \in P, M \in Refs_C( p, q ): M \supseteq Y \} . \quad (**)$$

*Von C nach PM*:

Es sei nun

$$X := min \{ M \in Refs_{PM}( P ) \mid M \supseteq Y \} ,$$

d.h. die minimale Obermenge von Y, über die wir in PM den Zustand P verlassen können. Nach Konstruktion von PM ist die Menge $Refs_{PM}( P )$ nichts anderes als das $\mathfrak{X}^{\cap}$ aus 2.2.4.3. Damit können wir $X$ aber auch schreiben als:

$$X = \bigcap_{M \in \mathfrak{X} \text{ mit } M \supseteq Y} M ,$$

und wir sehen, dass $X$ wohldefiniert ist, falls in $\mathfrak{X}$ wenigstens ein solches $M \supseteq Y$ existiert. Dies ist aber nach (**) klar, da ansonsten $P' = \emptyset$ wäre, was laut der Nachbemerkung zu 1.3.1.7 nicht möglich ist. Dieses X hat nun folgende Eigenschaft:

$$\forall p \in P, M \in Refs_C( p ) : M \supseteq Y \Rightarrow M \supseteq X .$$

Damit können wir statt (**) auch schreiben:

$$P' = \cup_{p \in P} \{ q \in Q_C \mid M \in Refs_C( p, q ): M \supseteq X \} .$$

Wir formen dann weiter um:

$= \cup_{p \in P} \{ q \in Q_C \mid \exists M \in Ref: (p, M, q) \in \delta_C \wedge M \supseteq X \}$
$= \cup_{p \in P} \cup_{M \in Ref:\ M \supseteq X} \delta_C(p, M)$ .

Nun ist ja aber nach Konstruktion $X \in \mathcal{X}$, weiter wie oben bemerkt $P' \neq \emptyset$ und nach Definition 2.2.4.3 erhalten wir:

$(P, X, P') \in \delta_{PM}$ .

*Von PM nach D*:

Wir sehen nun schnell, dass unser X genau so gewählt ist, dass es die Rolle der Menge X in Definition 2.2.4.7 erfüllt, und es folgt schließlich:

$(P, Y, P') \in \delta_D$ .

□

### *2.2.4.11 Korollar*

Seien $A_1$, $A_2$ abgeschlossene und Ref-kohärente Automaten, und für $i=1,2$ seien zwei weitere Automaten $D_i := P\text{-}Decmp( \wp_{mod}( Cmpr( A_i )))$ definiert.

Dann sind äquivalent:

1) Die Sprache von $A_1$ ist in der von $A_2$ enthalten.
2) $D_2$ kann $D_1$ simulieren.

**Beweis:**

klar aus obigem zusammen mit 1.3.1.6.

□

## 2.2.5 Späte P-Dekompression

Korollar 2.2.4.11 scheint nun nahezulegen, vor der Simulation die P-Dekompression auf die modifizierten Potenzautomaten anzuwenden. In der Anwendung innerhalb von FastAsy wollen wir aber selbst diesen Schritt noch eliminieren, um ganz ohne Dekompression auszukommen. Dazu benötigen wir eine Simulationsrelation, welche sozusagen die P-Dekompression enthält. Wir generalisieren zunächst die Simulationsvorschrift, so dass der simulierende Automat nicht notwendigerweise über eine gleichbeschriftete Kante laufen muss. (Dabei stützen wir unsere Definition auf die alternative Formulierung einer Simulation aus 1.3.1.5.)

### *2.2.5.1 Definiton*

Seien $TS^1$, $TS^2$ zwei Transitionssysteme und es sei eine *Auswahlvorschrift*

$f: (\Lambda \cup \{\lambda\}) \times 2^{\Lambda} \to \Lambda \cup \{\lambda\} \cup \{\perp\}$

mit

$f(x, M) \in M \cup \{\perp\}$ und
$f(\lambda, M) = \lambda$

gegeben, welche für eine zu simulierende Übergangsbeschriftung und eine Menge möglicher Beschriftungen diejenige auswählt, mit deren Hilfe der Übergang simuliert werden soll.

Eine Relation $S \subseteq Q^1 \times Q^2$ heißt nun *generalisierte* f-*Simulation*, wenn:

1) $(q_0^1, q_0^2) \in S$ und
2) $\forall (q^1, q^2) \in S, a \in \Lambda \cup \{\lambda\}$ *mit* $q^1 \xrightarrow{a} q^1'$ gilt:
   $\exists q^2' \in Q^2: q^2 \xrightarrow{b} q^2'$
   $\wedge\ b = f( a, \{ c \in \Lambda \mid \delta_2( q^2, c ) \neq \varnothing \} ) \neq \perp$
   $\wedge\ (q^1', q^2') \in S$ .

Wir sagen, $TS^2$ kann $TS^1$ f-simulieren, wenn eine solche generalisierte f-Simulation existiert.

□

Die Auswahlvorschrift bekommt also eine zu simulierende Aktion und die Menge der zur Verfügung stehenden Aktionen übergeben und wählt daraus diejenige Aktion *b* aus, mit welcher die ursprüngliche Aktion *a* simuliert wird. Die Auswahlvorschrift kann auch ein Fehlerzeichen $\perp$ liefern, in welchem Fall keine Simulation existiert.

Es sei noch bemerkt, dass in unserem Kontext die Auswahlvorschrift auch auf $f: (\Lambda \times 2^{\Lambda}) \rightarrow \Lambda \cup \{\perp\}$ eingeschränkt werden kann, da wir ja mit buchstabierenden Automaten arbeiten. Weiter ergibt sich für deterministisches $TS^2$ ebenso wie bei der klassischen Simulation der Zielzustand von $TS^2$ deterministisch.

Bevor wir nun dazu übergehen, die konkrete Auswahlfunktion für unseren Anwendungsfall zu konstruieren, wollen wir noch argumentieren, warum man die P-Dekompression (mit Gewinn für die Effizienz) auch für das *simulierte* System eliminieren kann.

Bei der Prüfung, ob eine Simulation existiert, betrachten wir ja jede ausgehende Kante im simulierten System, in obigem Anwendungsfall also in $D_1$. Führen wir uns die Situation jedoch noch einmal genau vor Augen, so bemerken wir, dass es auch ausreichend ist, nur jede ausgehende Kante in $PM_1 := \wp_{mod}( Cmpr( A_1 ))$ zu betrachten:

Wir benötigen zunächst ein technisches Lemma:

### *2.2.5.2 Lemma*

Sei A ein buchstabierender Automat und $PA := \wp( A )$. Es sei $Q_0$ der Startzustand von PA, $R_0 \in Q_{PA}$ mit $R_0 \supseteq Q_0$ und PA' gehe aus PA hervor, indem wir $R_0$ als Startzustand setzen.

Dann kann PA' den Automat PA simulieren.

**Beweis:**

Klar aus der Konstruktion des Potenzautomaten durch Induktion.

□

### *2.2.5.3 Korollar*

Seien $A_1$, $A_2$ abgeschlossene und Ref-kohärente Automaten, für $i=1,2$ seien zwei weitere Automaten $D_i := P\text{-}Decmp( \wp_{mod}( Cmpr( A_i )))$ definiert und zusätzlich sei $PM_1 := \wp_{mod}( Cmpr( A_1 ))$. Dann sind äquivalent:

1) $D_2$ kann $D_1$ simulieren.
2) $D_2$ kann $PM_1$ simulieren.

**Beweis:**

*‚1) ⇒ 2)'*:

einfache Folgerung aus 2.2.4.8.

*‚2) ⇒ 1)'*:

Sei *( P, Y, P' )* ein Übergang in $D_1$. Dann existiert laut 2.2.4.7 ein Übergang *( P, X, P' )* mit einem $X \supseteq Y$ in $PM_1$. Nach Voraussetzung kann $D_2$ diesen simulieren, für gewähltes *Q* etwa mit einem Übergang *( Q, X, Q' )*. Wegen der Kontravarianz (siehe Bemerkung nach 2.2.4.2) existiert nun in $D_2$ ein Übergang *( Q, Y, Q'' )* mit $Q'' \supseteq Q'$. Nach 2.2.4.10 ist $D_2$ ja identisch mit einem Potenzautomaten, und mit Lemma 2.2.5.2 folgt so schnell die Behauptung.

□

Wir werden nun eine Auswahlfunktion konstruieren, die genau der Dekompression aus 2.2.4.7 entspricht:

### *2.2.5.4 Satz*

Seien $A_1$, $A_2$ Automaten. Es sei weiter durch $f_D$ eine Auswahlvorschrift gegeben:

- Für $\varepsilon = a \in Acts$ sei $f_D( a, M ) := a$ , falls $a \in M$
- für $\varepsilon = Y \in Refs$ sei $f_D( Y, M ) := X$ , falls ein $X \in M$ mit den Eigenschaften existiert: $Y \subseteq X \ \wedge \ ( \forall Z \in M: Y \subseteq Z \Rightarrow X \subseteq Z )$ ,
- $f_D( \varepsilon, M ) = \perp$ sonst.

Dann sind äquivalent:

1) *P-Decmp( $A_2$ )* kann $A_1$ simulieren.
2) $A_2$ kann $A_1$ $f_D$-simulieren.

**Beweis:**

Klar nach Konstruktion.

□

Zusammenfassend erhalten wir mit dem obigen $f_D$ nun denjenigen Sachverhalt, der letzten Endes in FastAsy Anwendung findet:

### *2.2.5.5 Korollar*

Seien $A_1$, $A_2$ abgeschlossene und Ref-kohärente Automaten, und für $i=1,2$ seien zwei weitere Automaten $PM_i := \wp_{mod}( Cmpr( A_i ))$ definiert.

Dann sind äquivalent:

1) Die Sprache von $A_1$ ist in der von $A_2$ enthalten.
2) $PM_2$ kann $PM_1$ $f_D$-simulieren.

**Beweis:**

Aus 2.2.5.3 wissen wir zunächst, dass 1) äquivalent ist zu:

*$D_2$ kann $D_1$ simulieren.*

Nach 2.2.5.3 ist wiederum dazu äquivalent:

*$D_2$ kann $PM_1$ simulieren.*

Nun ist aber $D_2$ nichts anderes als *P-Decmp( $PM_2$ )*, so dass wir äquivalent schreiben können:

*P-Decmp( $PM_2$ ) kann $PM_1$ simulieren.*

Die Anwendung von 2.2.5.4 schließlich liefert direkt das Ergebnis.

□

## 2.3 Ergebnis

Fassen wir nochmals zusammen, was die Resultate aus Kapitel 2 für die Verfahrensweise in unserem Tool gebracht hat: Statt aus einem Netz das komplette Transitionssystem $TR_{lrmax}$ aufzubauen, konstruieren wir lediglich *Cmpr( $TR_{lrmax}$ )*.

Die Kompression müssen wir dabei noch nicht einmal wirklich durchführen, denn nach 1.4.5.1 ist es in der Tat wesentlich einfacher, direkt die maximale Verweigerungsmenge zu konstruieren.

Eigentlich würden wir ja nun *Cmpr( Ext( $TR_{lrmax}$ ))* für die Weiterverarbeitung benötigen, aber wie wir in 2.2.3 gesehen haben, dürfen wir stattdessen auch einfach *Ext( Cmpr( $TR_{lrmax}$ ))* berechnen.

Daraus berechnen wir nun in einem weiteren Schritt *PM :=* $\wp_{mod}$*( Ext( Cmpr( $TR_{lrmax}$ ))*. Dies führen wir für alle zu vergleichenden Netze durch. (Im Rahmen dieser Arbeit wird in Teil II , 7.2 noch argumentiert, warum dies günstig ist.)

Anschließend können wir dann die schneller-als-Relation zwischen zwei Netzen dadurch entscheiden, dass wir wechselseitige $f_D$-Simulationen zwischen den verschiedenen PM durchführen.

Anhang C zeigt, welche praktischen Ergebnisse die mit Hilfe unserer Resultate implementierte Optimierung in FastAsy liefert: Schon bei Netzen mit wenigen sichtbaren Aktionen (*mutex3* und *mutex4* in verschiedenen Varianten mit jeweils 3 Aktionen) erhöht sich die Geschwindigkeit teilweise bis um Faktor 4. Steigt die Zahl der sichtbaren Aktionen dann an (auf 6 bei *mutex1* und *mutex2* für 2 Benutzer, auf 9 für 3 Benutzer), übersteigt der Zugewinn an Effizienz teilweise schon Faktor 1000.

# Kapitel 3 Substitution von Read-Arcs

## *3.1 Einleitung*

In der Urversion unserer zeitlichen Semantik, wie sie etwa in [BV98] beschrieben wird und die auf [Vog95c] bzw. in der Version mit Lesekanten auf [Vog96] zurückgeht, existierten noch keine expliziten Angaben von Zeitintervallen in den Netzen. Stattdessen wurden Transitionen so behandelt, als wären alle eingehenden Kanten (und Lesekanten) implizit mit [0,1] beschriftet. (Insofern hat es in diesen Versionen auch ausgereicht, statt dem Alter von Marken auf den Stellen einfach die Menge dringender Transitionen in der Markierung mitzuführen.) In [Vog96] wird gezeigt, dass Lesekanten die Ausdrucksmächtigkeit solcher Netze vergrößert, wenn schwach faire oder zeitbehaftete Semantiken betrachtet werden.

Nachdem mit [Bih98] dann eine Netzvariante mit expliziten Zeitintervallen vorgestellt wurde, stellte sich natürlich die Frage, wie sich die Ausdrucksmächtigkeiten von Lesekanten und Zeitintervallen verhalten. Noch in [Bih98] selbst wurde eine Version eines Mutex-Netzes dargestellt, das ohne Lesekanten und nur mit [0,0]- und [0,1]-Kanten auskommt. Dies ist deshalb von besonderem Interesse, weil [Vog96] zeigt, dass klassische Petri-Netze zur Lösung des Mutex-Problems (nach dem dort angegebenen Verständnis) nicht ausreichen.

Da wir im Rahmen dieser Arbeit ein Netzmodell verwenden, welches sowohl Lesekanten als auch Intervalle zulässt, wollen wir im folgenden etwas allgemeiner die Frage untersuchen, inwiefern Read-Arcs in PTT-Netzen durch Konstruktionen mit Zeitintervallen ersetzt werden können.

Genauer betrachten wir folgenden allgemeinen Netzausschnitt um eine Lesekante herum:

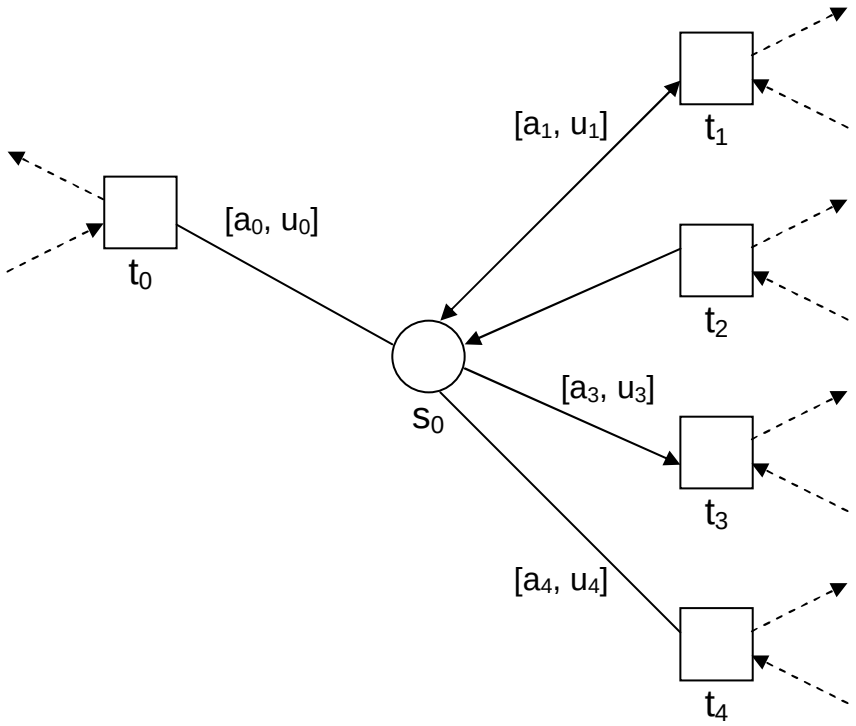



**Abbildung 14: Zu transformierender Netzausschnitt**

Wir wollen diesen Ausschnitt dann derart transformieren, dass die Lesekante $(t_0, s_0)$ verschwindet. Wenn wir dann noch weiter zeigen, dass Voraussetzungen, die wir ggf. treffen

müssen, nach der Transformation erhalten bleiben und außerdem keine weiteren Lesekanten hinzukommen, können wir so sukzessive alle Lesekanten entfernen.

Die betrachteten Äquivalenzen werden Bisimularität auf $TR_r$ und $TR_{lr}$ sowie Sprachgleichheit von $TR_{lr}$ sein (siehe 1.4.6). Letzteres ist natürlich nichts anderes als die Gleichheit von RT und somit in unserem Zusammenhang die wichtigste Äquivalenz.

Außerdem wird es für Tranformationen ein zentrales Kriterium sein, ob sie die Klasse asynchroner Netze erhalten. Denn obwohl wir im Rahmen dieser Arbeit auch nicht-asynchrone Netze behandeln können, sehen wir die asynchronen Netze immer als den wirklich interessanten Fall an. (Nicht-asynchrone Netze sind schließlich einfach getaktete Netze, die man in Folge auch mit einem einfacheren Ansatz untersuchen könnte.)

Bevor wir uns nun der ersten Konstruktion zuwenden, wollen wir noch eine alternative Schreibweise für Vor-, Nach- und Lesebereiche einführen. Wenn wir nämlich, wie im folgenden immer wieder, von mehreren Netzen sprechen, die teilweise gleichbenannte Transitionen und Stellen enthalten, sind die „$t^\bullet$“-Schreibweisen nicht mehr klar definiert. Stattdessen verwenden wir eine mehr algebraische Notation, wie sie etwa [Sta90] gebraucht:

#### *3.1.1.1 Definition*

Für Mengen M, N, eine Relation $R \subseteq M \times N$ und Elemente $m \in M$, $n \in N$ sei

$$m\,R := \{ x \in N \mid (m, x) \in R \} \quad \text{und}$$
$$R\,n := \{ x \in M \mid (x, n) \in R \} .$$

□

Außerdem gelte die übliche Konvention, dass ornamentierte Bezeichner wie etwa F', R', usw. die Bestandteile des Netzes N' sind, auch wenn dies nicht eigens erwähnt wird. Damit können wir beispielsweise die u.U. verschiedenen Nachbereiche einer Transition t in N und N' jeweils eindeutig mit $t\,F$ und $t\,F'$ bezeichnen.

## *3.2 Substitutionskonstruktionen*

### 3.2.1 Erste Substitution

Von der ersten Transformation, die wir vorstellen, wird sich zeigen, dass sie nur unter einschränkenden Bedingungen korrekt ist. Ihre Betrachtung lohnt trotzdem, denn erstens ist die Verwandschaft mit dem ursprünglichen Netzgraphen ausgesprochen stark (was auch zu einer entsprechend starken Äquivalenz, nämlich der Bisimularität auf $TR_r$, führen wird), und zweitens sind genau die Situationen, in denen die Konstruktion versagt, sehr illustrativ.

Die Idee bei dieser ersten Transformation ist es, die Stelle $s_0$ zu duplizieren, so dass die neue Stelle $s_0$' das Timing von $t_0$ übernimmt, während $s_0$ weiterhin die anderen adjazenten Transitionen $t_1$-$t_4$ kontrolliert:

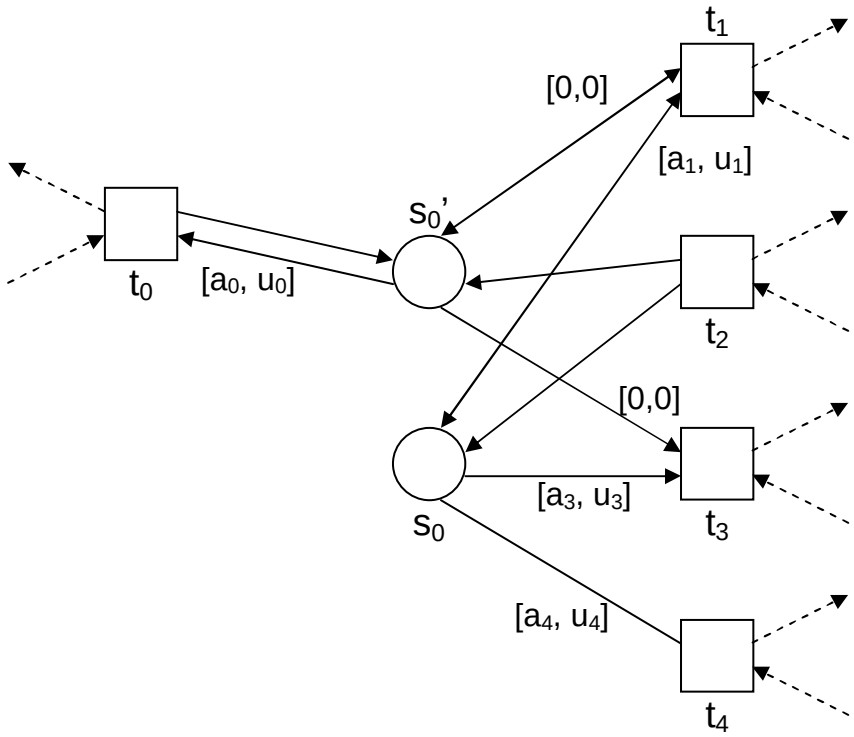



**Abbildung 15: Transform$_1$**

Die Idee, dass Lesekanten durch Aufspaltung einer Stelle in gewissem Sinne, allerdings ohne Berücksichtigung von Zeit, nachgebildet werden können, ist bekannt und findet sich z.B. schon in [MR95]. In [VSY98] werden, ebenfalls für Netze ohne Zeit, solche Aufspaltungen benutzt, um die Komplexität von Unfoldings herabzusetzen.

Formaler definieren wir nun die Transformation wie folgt:

### *3.2.1.1 Definition*

Für ein Netz $N = (S, T, F, R, M_N, l, lb, ub)$, eine Stelle $s_0 \in S$ und eine Transition $t_0 \in T$ mit $(s_0, t_0) \in R$ und $s_0' \notin S$ sei $N' = Transform_1( N, (s_0, t_0) )$ wie folgt definiert:

$$T' = T$$
$$l' = l$$
$$S' = S \cup \{s_0'\}$$
$$M_N' = M_N \cup \{s_0' \mid s_0 \in M_N \}$$
$$R' = R \setminus \{ s_0, t_0) \}$$

(d.h. Weglassen der fraglichen Lesekante)

$$F' = F \cup \{ (t_0, s_0'), (s_0', t_0) \} \cup \{ (t, s_0') \mid t \in F\, s_0 \} \cup \{ (s_0', t) \mid t \in s_0\, F \}$$

(d.h. $t_0$ wird *nur*, alle anderen zu $s_0$ F-adjazenten Transitionen werden *zusätzlich* mit $s_0'$ verbunden.)

$$lb' |_{F \cap F'} = lb |_{F \cap F'}$$
$$lb' |_{R \cap R'} = lb |_{R \cap R'}$$
$$lb'( s_0', t ) = 0 \quad \forall t \in s_0\, F$$
$$lb'( s_0', t_0 ) = lb( s_0, t_0 )$$
$$ub' |_{F \cap F'} = ub |_{F \cap F'}$$

$ub' |_{R \cap R'} = ub |_{R \cap R'}$
$ub'( s_0', t ) = 0 \quad \forall t \in s_0 \, F$
$ub'( s_0', t_0 ) = ub( s_0, t_0 )$

(d.h. die neuen Kanten werden alle mit *[0,0]* beschriftet, mit Ausnahme von $(s_0', t_0)$, welche die Inschrift der früheren Lesekante trägt.)

□

Für den Rest von 3.2.1 seien N und N' fest wie in 3.2.1.1.

Wir wollen nun eine Relation B auf den IDs der Netze definieren, die die Grundlage unserer späteren Bisimulationsrelation sein wird.

### *3.2.1.2 Definition*

Für N, N' und beliebige IDs ρ von N und ρ' von N' sei nun folgende Relation B definiert:

$\rho \, B \, \rho' \quad :\Leftrightarrow I) \wedge II) \wedge III)$

wobei:

$I) \quad :\Leftrightarrow \quad \forall s \in (S \cap S'): \rho'( s ) = \rho( s )$
$II) \quad :\Leftrightarrow \quad \rho'( s_0' ) \leq \rho( s_0 )$
$III) \quad :\Leftrightarrow \quad ( \rho'( s_0' ) = -\infty \Leftrightarrow \rho( s_0 ) = -\infty )$ .

□

Mit anderen Worten stehen zwei IDs vermöge B genau dann in Relation, wenn:
- sie auf allen Stellen bis auf $s_0'$ übereinstimmen,
- das Alter der Marke auf $s_0'$ in N' dasjenige von $s_0$ in N nicht übersteigt
- auf $s_0'$ in N' genau dann eine Marke liegt, wenn auf $s_0$ in N eine liegt.

Man sieht zunächst natürlich sofort, dass für $N' = Transform_1( N, (s_0, t_0) )$ nach Konstruktion gilt $\rho_N \, B \, \rho_{N'}$. Um zu untersuchen, wann B sich aber auch weiter erhält, betrachten wir zunächst die Beziehung von $s_0$ und $s_0'$ in dem Netz, das durch $Transform_1$ gewonnen wird:

### *3.2.1.3 Lemma*

Seien $\rho_1$, $\rho_2$ Markierungen von N'.

Falls $\rho_1( s_0' ) \leq \rho_1( s_0 )$ und $( \rho_1( s_0' ) = -\infty \Leftrightarrow \rho_1( s_0 ) = -\infty )$ erfüllt sind, dann gelten:

$i) \quad \forall t \in T: \rho_1[t\rangle\rho_2 \quad \Rightarrow \quad \rho_2( s_0' ) \leq \rho_2( s_0 )$
$ii) \quad \forall X \in \Sigma: \rho_1[X\rangle\rho_2 \quad \Rightarrow \quad \rho_2( s_0' ) \leq \rho_2( s_0 )$ .
$iii) \quad \forall t \in T: \rho_1[t\rangle\rho_2 \quad \Rightarrow \quad ( \rho_2( s_0' ) = -\infty \Leftrightarrow \rho_2( s_0 ) = -\infty )$
$iv) \quad \forall X \in \Sigma: \rho_1[X\rangle\rho_2 \quad \Rightarrow \quad ( \rho_2( s_0' ) = -\infty \Leftrightarrow \rho_2( s_0 ) = -\infty )$ .

Falls $\rho_1( s_0' ) = \rho_1( s_0 )$, so gelten auch:

$v) \quad \forall t \in T \setminus \{t_0\}: \rho_1[t\rangle\rho_2 \quad \Rightarrow \quad \rho_2( s_0' ) = \rho_2( s_0 )$
$vi) \quad \forall X \in \Sigma: \rho_1[X\rangle\rho_2 \quad \Rightarrow \quad \rho_2( s_0' ) = \rho_2( s_0 )$ .

**Beweis:**

*‚i), iii), v)'*:

Hierfür unterscheiden wir nach den Adjazenzen von t zu $s_0$ und $s_0$' in N'. Falls t gar nicht adjazent zu $s_0$ oder $s_0$' ist, gelten i), iii) und v) offensichtlich, da die Uhren auf $s_0$ und $s_0$' in diesen Fällen unverändert bleiben. Ansonsten unterscheiden wir die in Abbildung 15 dargestellten Fälle:

t ist vom Typ $t_1$:

Wegen $\rho_1[t\rangle_r$ muss gelten $\rho_1(s_0), \rho_1(s_0') > -\infty$, und nach Schaltregel gilt $\rho_2(s_0) = \rho_2(s_0') = 0$.

t ist vom Typ $t_2$:

Aufgrund der Sicherheit muss gelten $\rho_1(s_0) = \rho_1(s_0') = -\infty$, und nach Schaltregel gilt $\rho_2(s_0) = \rho_2(s_0') = 0$.

t ist vom Typ $t_3$:

Nach Schaltregel $\rho_2(s_0) = \rho_2(s_0') = -\infty$.

t ist vom Typ $t_4$:

Dann ist t nach Konstruktion nicht adjazent zu $s_0$'. Also gilt $\rho_1(s_0') = \rho_2(s_0')$. Nach Schaltregel aber auch $\rho_1(s_0) = \rho_2(s_0)$.

(Für v) geht hier noch die stärkere Voraussetzung ein.)

$t = t_0$:

Da $\rho_1[t\rangle_r$, ist $\rho_1(s_0') > -\infty$, nach Voraussetzung also $\rho_1(s_0) > -\infty$. Außerdem $\rho_2(s_0') = 0$ und natürlich $\rho_2(s_0) = \rho_1(s_0) \geq 0$.

(Für v) ist der Fall $t = t_0$ ja ausdrücklich ausgenommen, so dass dieses Ergebnis auch schon ausreicht.)

*‚ii), iv), vi)'*:

klar nach Schaltregel, da das Voranschreiten von Zeit alle Stellen gleichermaßen betrifft.

□

Mit anderen Worten bleiben folgende Eigenschaften in N' invariant:

- Es liegen entweder auf $s_0$ und $s_0$' eine Marke oder auf keiner der beiden Stellen.
- Eine Marke auf $s_0$ ist immer mindestens so alt wie eine auf $s_0$'.

Man sieht leicht, dass beide Eigenschaften nach Konstruktion in der Startmarkierung gelten, damit gelten sie also in jeder erreichbaren Markierung.

Diese Beziehung zwischen $s_0$ und $s_0$' in N' verwenden wir nun, um zu zeigen, dass die Relation B zwischen den IDs von N und N' erhalten bleibt, *falls* beide gemeinsam einen Schritt in RFS schalten können.

### *3.2.1.4 Lemma*

Für Markierungen $\rho_1$, $\rho_2$ von N und $\rho_1$', $\rho_2$' von N' mit $\rho_1 \, B \, \rho_1'$ gilt:

*i)* $\quad \forall t \in T: \rho_1[t\rangle\rho_2 \wedge \rho_1'[t\rangle\rho_2' \quad \Rightarrow \quad \rho_2 \, B \, \rho_2'$

*ii)* $\forall X \subset \Sigma: \rho_1[X\rangle\rho_2 \wedge \rho_1'[X\rangle\rho_2' \Rightarrow \rho_2 \, B \, \rho_2'$ .

**Beweis:**

‚i)':

Bem.: Große römische Ziffern in diesem Beweis beziehen sich auf die Konjunktion aus der Definition von B in 3.2.1.2.

Es sei $S_{inv} := S\backslash\{s_0\} = S'\backslash\{s_0, s_0'\}$. Dann gilt nach Konstruktion von N':

$F \cap ( S_{inv} \times T ) = F' \cap ( S_{inv} \times T )$
$F \cap ( T \times S_{inv} ) = F' \cap ( T \times S_{inv} )$
$R \cap ( S_{inv} \times T ) = R' \cap ( S_{inv} \times T )$
$lb| S_{inv} \times T = lb'| S_{inv} \times T$
$ub| S_{inv} \times T = ub'| S_{inv} \times T$ .

Insbesondere gilt für beliebige $\rho_1$ und $\rho_1'$ mit $\rho_1| S_{inv} = \rho_1'| S_{inv}$ auch:

$\forall t \in T: \rho_1[t\rangle\rho_2 \wedge \rho_1'[t\rangle\rho_2' \Rightarrow \rho_2| S_{inv} = \rho_2'| S_{inv}$.

Da aber aus $\rho_1 \, B \, \rho_2$ nach Definition schon $\rho_1| S_{inv} = \rho_1'| S_{inv}$ folgt, müssen wir in den folgenden Fällen nur noch $S_{var} := \{s_0, s_0'\}$ untersuchen. Es genügt also mit anderen Worten, statt I) zu zeigen:

*IV)* $\rho_2'( s_0 ) = \rho_2( s_0 )$

Dies zeigt man aber leicht durch Unterscheidung derselben Fälle wie im Beweis zu 3.2.1.3.

Unter der Voraussetzung IV) lassen sich dann II) und III) umformen zu:

*V)* $\rho_2'( s_0' ) \leq \rho_2'( s_0 )$
*VI)* $( \rho_2'( s_0' ) = -\infty \Leftrightarrow \rho_2'( s_0 ) = -\infty )$ .

Sowohl V) als auch VI) sprechen jetzt nur mehr über N', so dass wir 3.2.1.3 anwenden können.

*‚ii)'*:

klar nach Schaltregel.

□

Nun bliebe noch zu zeigen, dass für Markierungen, die vermöge B in Relation stehen, tatsächlich alle Schritte von N auch für N' möglich sind und umgekehrt. Interessanterweise wird im Beweis des Theorems die Voraussetzung $\rho \, B \, \rho'$ aber gar nicht ausreichen, da wir über die Vergangenheit von ρ bzw. ρ' argumentieren und dazu voraussetzen müssen, dass in beiden Netzen eine *gemeinsame* Vergangenheit existiert.

### *3.2.1.5 Definition*

Seien N, N' zwei Netze mit $T = T'$. Zwei Markierungen ρ von N und ρ' von N' heißen *RFS-gemeinsam erreichbar*, wenn ein $w \in RFS(N) \cap RFS(N')$ existiert mit $\rho_N[w\rangle_r \rho$ und $\rho_{N'}[w\rangle_r \rho'$.

Wir scheiben dafür auch $\rho \, /\!\!/^{RFS} \, \rho'$.

□

### *3.2.1.6 Definiton*

Die Relation $B_{\|}$ auf den Markierungen von N und N' sei gegeben durch:

$$B_{//} := B \cap //_{RFS} \quad .$$

□

Offensichtlich gilt die Aussage 3.2.1.4 auch für $B_{\|}$.

Wir haben bereits angedeutet, dass im allgemeinen Fall B und $B_{\|}$ leider keine Bisimulation sein müssen. Vergleichen wir nun nochmals Abbildung 14 mit Abbildung 15, so sehen wir auch bereits das mögliche Problem: In N' deaktiviert sich $t_0$ beim Schalten auf jeden Fall für $a_0$ Runden, in N jedoch nicht unbedingt. Genauer hängt es nun am Verhalten der Umgebung, ob diese Tatsache die Bisimulation zerstört. Wir werden am entsprechenden Beweis noch genau ablesen können, wie diese Anforderung aussieht.

Ein hinreichendes, statisch und lokal prüfbares Kriterium dafür, dass dieser Unterschied keine Rolle spielt, ist jedoch folgendes:

### *3.2.1.7 Definition*

Für eine Stelle $s' \in {}^\wedge t$ heißt eine Transition t *wohlgesteuert bezüglich s'*, wenn gilt:

*i)* $\exists s \in {}^\bullet t: lb(s, t) \geq lb(s', t)$

*ii)* $\exists s \in {}^\bullet t: ub(s, t) \geq ub(s', t)$ .

Eine Transition t heißt wohlgesteuert, wenn sie bezüglich aller Stellen ihres Lesebereichs wohlgesteuert ist. Ein Netz heißt wohlgesteuert, wenn alle Transitionen wohlgesteuert sind.

□

Bei wohlgesteuerten Transitionen hat die Lesekante nur dann Einfluss auf das zeitliche Verhalten, wenn die entsprechende Marke ‚spät' kommt. Kommen nämlich beispielsweise alle Marken gleichzeitig an, so ist es ja nach i) der Definition eine nicht-Lesekante, die bei der Aktivierung „das letzte Wort hat" und nach ii) ebenfalls bei der Frage, wann t dringend wird. (Wir sagen dann, diese Kante *kontrolliert* die Aktivierung bzw. die Dringlichkeit.) Ist weiter direkt nach dem Schalten t nicht deaktiviert, haben auf jeden Fall wieder echte Kanten die zeitliche Kontrolle. Bei nicht wohlgesteuerten Transitionen trifft dies jedoch nicht unbedingt zu.

Folgendes Beispiel mag den Sachverhalt einfach veranschaulichen:

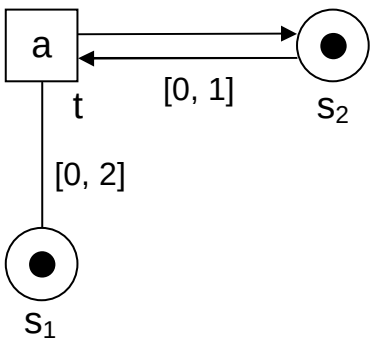



**Abbildung 16: Fehlende Wohlgesteuertheit**

Wie man sieht, ist t nicht wohlgesteuert. In den ersten beiden Runden kann immer a verweigert werden, und zwar völlig unabhängig davon, wie oft es bereits geschaltet hat (denn die Lesekante hat „das letzte Wort“). Nach Ablauf dieser Zeit aber signalisiert die Lesekante grundsätzlich Dringlichkeit, und in Folge dessen kann nie mehr zweimal hintereinander a verweigert werden.

Auf dem Netz in Abbildung 16 bewirkt *Transform1* nun, dass die Lesekante durch eine Schlinge ersetzt wird. Offensichtlich wird dadurch die Ausführung von $\Sigma\,\Sigma\,a\,\Sigma\,\Sigma$ möglich, was vorher nicht der Fall war.

Allgemeiner werden wir aber in 3.2.2.1 sehen, dass es überhaupt kein Netz ohne Lesekanten und mit t als einziger Transition geben kann, welches die gleichen Verweigerungsfolgen wie das Netz in Abbildung 16 hat. (Aus dieser Beobachtung wird sich später die Idee für eine Substitution für nicht-wohlgesteuerte Netze ergeben, bei der t für die verschiedenen Situationen vor und nach Zeit 2 dupliziert wird.)

Nun wollen wir aber zunächst für wohlgesteuertes $t_0$ das noch fehlende Lemma zeigen:

### *3.2.1.8 Lemma*

Sei wie oben $N' = Transform_1(N, (s_0, t_0))$ und zusätzlich $t_0$ wohlgesteuert bezüglich $s_0$. Es seien ρ bzw. ρ’ Markierungen von N bzw. N’ mit $\rho\ B_{/\!/}\ \rho'$. Dann gilt:

*i)* $E(\rho) = E'(\rho')$
*ii)* $U(\rho) = U'(\rho')$ .

**Beweis:**

*‚i)’*:

Wir zeigen für ein fest gewähltes t:

*(*)* $\forall s \in (F \cup R)\ t: \rho(s) \geq lb(s, t) \quad \Leftrightarrow \quad \forall s \in (F' \cup R')\ t: \rho'(s) \geq lb'(s, t)$ .

Wir unterscheiden nun für t folgende Fälle:

$t \neq t_0,\ t \notin s_0\ F$:

Dann gilt nach Konstruktion

$$(F \cup R)\ t = (F' \cup R')\ t$$

und

$$lb|\ (F \cup R)\ t = lb'|\ (F \cup R)\ t \quad .$$

Wegen $\rho\ B\ \rho'$ wissen wir auch noch

$$\rho|\ (F \cup R)\ t = \rho'|\ (F \cup R)\ t \quad .$$

Damit gilt offensichtlich (*).

$t \in s_0\ F$:

Nach Konstruktion gilt dann

$$(F' \cup R')\ t = ((F \cup R)\ t) \cup \{ s_0' \}$$

und

$lb'(s_0', t) = 0$ .

Mit den Argumenten aus dem ersten Fall ist ‚⇐' aus (*) klar.
Da weiter zumindest $s_0 \in (F \cup R)\, t$, folgt aus der linken Seite von (*) insbesondere

$\rho(s_0) \geq lb(s_0, t) \geq 0$ .

Wegen $\rho\ B_{/\!/}\ \rho'$ gilt damit $\rho'(s_0) \geq 0$ und nach 3.2.1.3 auch $\rho'(s_0') \geq 0$. Also gilt

$\rho'(s_0') \geq 0 = lb'(s_0', t)$

und somit ‚⇒' in (*).

<u>t = $t_0$</u>:

Es sei $S_\cap := ((F \cup R)\, t_0) \cap ((F' \cup R')\, t_0$ .
Dann gelten nach Konstruktion zunächst:

$S_\cap = (F \cup R)\, t_0 \setminus \{s_0\} = (F' \cup R')\, t_0 \setminus \{s_0'\}$
$lb|\, S_\cap = lb'|\, S_\cap$
$lb(s_0, t_0) = lb'(s_0', t_0)$.

Nach $\rho\ B\ \rho'$ gilt weiter:

$\rho|\, S_\cap = \rho'|\, S_\cap$ .

Zusammen mit 3.2.1.3 und der rechten Seite von (*) erhält man dann:

$\rho(s_0) = \rho'(s_0) \geq \rho'(s_0') \geq lb'(s_0', t_0) = lb(s_0, t_0)$ .

Damit ist die Richtung ‚⇐' in (*) schon klar.

Nehmen wir nun an, dass ‚⇒' nicht gelte:

$\forall s \in S_\cap \cup \{s_0\} : \rho(s) \geq lb(s, t_0) \wedge$
$\exists s \in S_\cap \cup \{s_0'\} : \rho'(s) < lb'(s, t_0)$ .

Für alle $s \in S_\cap$ folgt ja aber aus der ersten Zeile und $\rho\ B\ \rho'$:

$\rho'(s) = \rho(s) \geq lb(s, t_0) = lb'(s, t_0)$ .

Somit kann es sich bei dem s aus der zweiten Zeile nur noch um $s_0$' handeln.
Unsere Widerspruchsannahme lautet also:

*W1* $\forall s \in S_\cap : \rho(s) \geq lb(s, t_0) \wedge$
*W2* $\rho'(s_0') < lb'(s_0', t_0)$ .

(Dies ist nun genau die Formalisierung dessen, was wir in der Bemerkung vor 3.2.1.7 bereits vermutet haben.)

Da $s_0$ in N markiert ist, wissen wir aus $\rho\ B\ \rho'$ , dass auch $s_0$' markiert ist.

Die Idee ist nun zu zeigen, dass W2 nur eintreten kann, wenn $t_0$ selbst ‚kürzlich' geschaltet und damit das Alter der Marke auf $s_0$' in N' zurückgesetzt hat, denn bei der Lesekante in N tritt dieser Effekt ja nicht auf. (Hier kommt dann auch zum Tragen, dass beide Netze eine gemeinsame Vergangenheit in RFS haben.)

Genau die Eigenschaft der Wohlgesteuertheit besagt dann, dass $t_0$ in N' dabei auch mindestens eine andere Stelle zurückgesetzt hat und auf die Marke dort mindestens ebenso lange warten muss. Auf diese Marke muss $t_0$ aber auch in N warten, so dass W1 nicht gelten kann.

Wir formalisieren dies wie folgt:

Sei dazu w eine Sequenz, vermöge der ρ und ρ' RFS-gemeinsam erreichbar sind. Wir zerlegen w nun wie folgt in drei Teile $w_1 \cdot \ldots \cdot w_2 \cdot \ldots \cdot w_3$:

$w_1$ sei das kürzeste Präfix von w, für das gilt:

$$\forall t_{in} \in F\ s_0 \colon t_{in} \notin w_2 \wedge t_{in} \notin w_3 \quad .$$

Mit anderen Worten: Genau in $last(w_1)$ wird das letzte Mal in w eine Marke auf $s_0$ gelegt oder $w_1$ ist leer und $s_0$ ist schon unter $\rho_N$ markiert.

Damit gilt für die Zustände $\rho_1$ und $\rho_1$' nach dem Ausführen von $w_1$ aber auf jeden Fall:

$$\rho_1(s_0) = \rho_1'(s_0') = 0 \quad .$$

$w_3$ sei nun weiter das längste Suffix von w mit $t_0 \notin w_3$.

(Nach W2, 3.2.1.3 v) und vi), und Konstruktion von $w_1$ enthält $w_2 \cdot w_3$ in jedem Fall mindestens ein $t_0$. Damit sind $w_2$ und $w_3$ wohldefiniert und $w_2$ nicht leer.)

Genau in $last(w_2)$ schaltet also das letzte Mal $t_0$, womit wir über den Zustand $\rho_2$' von N' nach Ausführung von $w_1 \cdot w_2$ wissen, dass $\rho_2'(s_0') = 0$.

Nun sei $n := \zeta(w_3)$. Dann gilt:

$$\rho'(s_0') = \rho_2'(s_0') + n = n \quad ,$$

und wir erhalten aus W2:

$$lb(s_0, t_0) = lb'(s_0', t_0) > n \quad .$$

Da $t_0$ in N wohlgesteuert ist bezüglich $s_0$, erhalten wir weiter:

$$\exists s_w \in F\ t_0 \colon lb(s_w, t_0) \geq lb(s_0, t_0) > n \quad .$$

Es sei nun $\rho_2$ der Zustand von N nach Ausführung von $w_1 \cdot w_2$. Nach Konstruktion von $w_2$ wissen wir, dass

$$\rho_2(s_w) = \rho_2'(s_w) \leq 0 \quad ,$$

denn $t_0$ hat ja am Ende von $w_2$ gerade geschaltet.

Nun kann aber wegen $\zeta(w_3) = n$ auch in N auf $s_w$ keine Marke liegen, die älter als n ist. Damit erhalten wir aber

$$\rho(s_w) \leq n < lb(s_w, t_0) \quad ,$$

was im Widerspruch zu W1 steht.

*,ii)'*:

wörtlich derselbe Beweis wie in i) mit *ub()* statt *lb()*.

□

Bemerkung: Wie gesagt ist die Wohlgesteuertheit lediglich ein hinreichendes Kriterium. Am letzten Abschnitt des obigen Beweises sieht man recht gut, worauf es eigentlich ankommt: Dass nämlich nicht eine Marke auf $s_0$ in N zwischen zwei Vorkommen von $t_0$ in einer Schaltfolge liegen bleibt und dort soweit altert, dass in N' (wo das Alter von $s_0$' durch $t_0$ ja zwangsläufig auf 0 gesetzt wird) die Transition $t_0$ nicht rechtzeitig aktiviert oder dringend wird.

In Abbildung 17 etwa sehen wir den Fall eines nicht-wohlgesteuerten Netzes, für das *Transform*$_1$ trotzdem ein bisimulares Netz liefern würde. Die Umgebung von $t_0$ im Netzgraphen stellt nämlich sicher, dass zwischen zwei Vorkommen von $t_0$ jeweils $t_1$ schalten muss. Dadurch wird die kritische Marke auf $s_0$ immer wieder zurückgesetzt, und die Uhren von $s_0$ in N und von $s_0$' in N' fangen gleichzeitig zu laufen an.

Wenn wir in der Abbildung jedoch die Schleife zwischen $s_0$ und $t_1$ eliminieren, so schlägt *Transform*$_1$ fehl: In N ist die Schaltfolge

$$\{a\}\, t_0\, t_1\, \{a\}\, \{a\}$$

nicht möglich, in $N' = Transform_1(N, (s_0, t_0))$ wäre sie es hingegen schon, da dort $t_0$ die Uhr auf der neu hinzugekommenen Stelle auf 0 zurücksetzen würde.

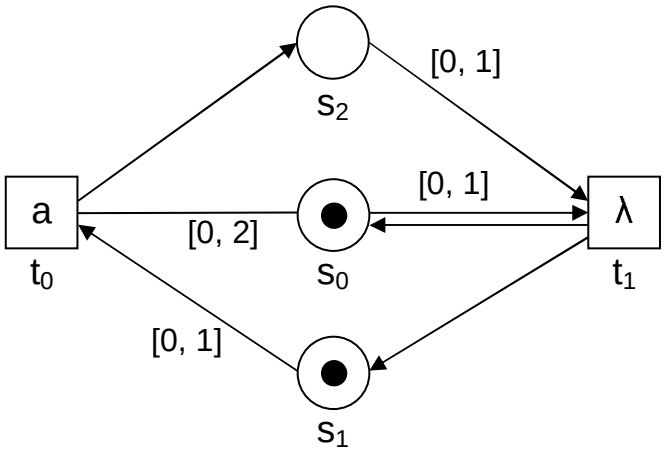



**Abbildung 17: Nicht-wohlgesteuertes $t_0$**

Trotzdem lässt sich das Kriterium der Wohlgesteuertheit oft erfüllen, insbesondere z.B. ist die Einbettung der Netzklasse aus [Vog96] in unsere PTT-Netze wohlgesteuert.

### *3.2.1.9 Korollar*

Seien *N* und $N' = Transform_1(N, (s_0, t_0))$ Netze wie oben und zusätzlich $t_0$ wohlgesteuert bezüglich $s_0$. Dann sind $TR_r(N)$ und $TR_r(N')$ bisimular vermöge $B_{\|}$.

**Beweis:**

Sofort aus 3.2.1.4 und 3.2.1.8 mit Induktion über die Länge der Schaltfolge. (Man beachte, dass die möglichen Verweigerungsmengen genau über die Menge der dringenden Transitionen definiert sind.)

□

Um nun unsere Hauptaussage beweisen zu können, müssen wir aber noch prüfen, ob *Transform*$_1$ auch die Wohlgesteuertheit intakt lässt, so dass wir die Konstruktion überhaupt mehrfach anwenden dürfen:

### *3.2.1.10 Lemma*

Seien $N$ und $N' = Transform_1( N, ( s_0, t_0 ))$ Netze wie oben und zusätzlich N wohlgesteuert. Dann ist auch N' wohlgesteuert.

**Beweis:**

Die Anwendung von $Transform_1$ führt erstens keine neuen Lesekanten oder ändert die Intervallbeschriftung auf vorhandenen. Zweitens werden den Vorbereichen der Transitionen höchstens Stellen hinzugefügt.

□

### *3.2.1.11 Theorem*

Für jedes wohlgesteuerte Netz N gibt es ein N' ohne Lesekanten, so dass $TR_r( N )$ und $TR_r( N' )$ bisimular sind.

**Beweis:**

Durch wiederholte Anwendung von $Transform_1$ können iterativ alle Lesekanten entfernt werden. Nach 3.2.1.10 bleiben dabei die Voraussetzungen erhalten.

□

Mit diesem Theorem können wir nun auch die Frage beantworten, die dieses Kapitel eigentlich motiviert hat:

### *3.2.1.12 Korollar*

Für jedes Netz N mit $lb \equiv 0, ub \equiv 1$ und Lesekanten (also die Netzklasse aus [Vog96]) existiert ein Netz N' mit $lb' \equiv 0, ub' \rightarrow \{0,1\}$ und *ohne* Lesekanten, so dass $TR_r( N )$ und $TR_r( N' )$ bisimular sind.

**Beweis:**

N ist wegen $lb \equiv 0, ub \equiv 1$ bereits wohlgesteuert. Anwendung von 3.2.1.11 entfernt die Lesekanten. $lb' \equiv 0, ub' \rightarrow \{0,1\}$ ist klar nach Konstruktion.

□

## 3.2.2 Grenzen bisimularer Konstruktionen

In 3.2.1 haben wir mit $Transform_1$ eine einfache Substitutionskonstruktion kennengelernt, welche darüber hinaus zu einer sehr starken Übereinstimmung zwischen dem ursprünglichen und dem transformierten Netz, nämlich einer Bisimulation auf $TR_r$ führt. Wir werden in diesem Abschnitt nun sehen, dass sich Konstruktionen mit solch starker Verwandschaft leider nicht allgemein finden lassen.

Weiter werden wir sehen, dass wir selbst die schwächere Bisimulation auf $TR_{lr}$ nicht allgemein erreichen können, wenn wir eine Transformation verlangen, die zumindest die sehr wichtige Klasse der asynchronen Netze erhält.

Betrachten wir noch einmal das Gegenbeispiel in Abbildung 16 aus 3.2.1. Die charakteristischen Elemente aus RT sind dabei von folgender Form (es seien dabei immer nur maximale Verweigerungsmengen angegeben, siehe Kapitel 2):

$$a^*\{a\}\{a\}\varnothing^* \; ( \; a \; \{a\}\varnothing^* \; )^*$$

Genauer ausgedrückt:

$$\begin{array}{ll} i) & \forall w \in RT: \zeta(w) = 0 \Rightarrow w\{a\}\{a\} \in RT \\ ii) & \forall w \in RT: \zeta(w) \geq 1 \Rightarrow w\{a\}\{a\} \notin RT \quad . \end{array}$$

Wir haben uns bereits überzeugt, dass $Transform_1$ schon bei diesem Minimalbeispiel versagt. (Im wesentlichen wird dabei die Lesekante durch eine Schlinge ersetzt, so dass natürlich ii) nicht mehr gilt.)

Allgemeiner können wir aber feststellen:

### *3.2.2.1 Lemma*

Sei N das in Abbildung 16 angegebene Netz und N' ein beliebiges Netz mit $T' = T$ und $R' = \varnothing$. Dann erfüllt $RT(N')$ nicht i) $\wedge$ ii).

**Beweis**:

Da $T' = \{t\}$, müssen alle Stellen in ${}^\bullet t$ markiert sein, sonst wäre N' tot und somit vermöge $w=\{a\}$ die Bedingung ii) verletzt. Da nur $t$ selbst Stellen markieren kann, folgt aus dem gleichen Grund schon ${}^\bullet t \subseteq t^\bullet$. (Da N' sicher ist, muss sogar ${}^\bullet t = t^\bullet$ gelten.) Aus i) folgt dann aber mit $w = \lambda$, dass $\exists s \in {}^\bullet t: u'(s, t) \geq 2$. Damit ist aber ii) etwa für $\{a\}\{a\}a\{a\}\{a\}$ offenbar verletzt, denn nach dem Schalten von $t$ wird die Stelle $s$ neu markiert, also wird $t$ frühestens nach zwei Runden wieder dringend.

□

### *3.2.2.2 Korollar*

Sei N das in Abbildung 16 angegebene Netz. Dann existiert kein Netz N' mit $R'=\varnothing$, so dass $TR_r(N)$ bisimular zu $TR_r(N')$ ist.

**Beweis:**

Angenommen, N' wäre im Widerspruch dazu solch ein Netz. Dann wären in N' alle Transitionen bis auf eine Transition t tot. Aus N' konstruieren wir nun N'', indem wir alle

toten Transitionen aus N' entfernen. Dann ist $TR_r( N'' )$ offenbar bisimular zu $TR_r( N' )$ und somit auch zu $TR_r( N )$. Weiter ist T'' = T, und wir führen die Annahme mit 3.2.2.1 zum Widerspruch.

□

Nun ist die Forderung nach bisimularen $TR_r$ natürlich schon deswegen sehr hart, da sie, wie man oben sieht, im wesentlichen die Gleichheit der Transitionsmengen voraussetzt. Betrachtet man stattdessen nur mehr Bisimularität auf den $TR_{lr}$, so fällt diese Einschränkung weg. Um zu zeigen, dass wir, zumindest für den Fall der asynchronen Netze, auch dafür kein zu obigem N äquivalentes Netz finden können, benötigen wir zunächst eine Hilfsaussage:

### *3.2.2.3 Lemma*

Sei N das in Abbildung 16 angegebene Netz und N' ein dazu $TR_{lr}$-bisimulares, asynchrones IT-Netz. Dann existiert ein asynchrones Netz N'' mit $TR_{lr}( N'' ) = TR_{lr}( N )$, in dem alle Transitionen lebendig sind. Gilt dabei $R'=\varnothing$, so auch $R''=\varnothing$.

**Beweis:**

*Schritt 1*:

Wir konstruieren in N' eine Schaltfolge $\alpha \in T^*$ mit $l(\alpha) \in a^*$, nach der alle Transitionen in N' entweder lebendig oder tot sind:

Wir wählen ein *t* aus, das weder lebendig noch tot ist und ‚töten' es im folgenden: Da *t* nicht lebendig ist, existiert eine Schaltfolge $\alpha_1'$ in $RFS( N' )$, nach der *t* nie wieder aktiviert wird. Da N' asynchron ist, ist erstens auch die Schaltfolge $\alpha_1 := \alpha_1' \cap T^*$ aktiviert und zweitens *t* auch nach $\alpha_1$ tot.

Wir wiederholen diesen Schritt, bis auf keine Transition mehr die Voraussetzung zutrifft und erhalten das gewünschte $\alpha$ durch Konkatenation aller $\alpha_i$.

*Schritt 2*:

Es sei N'' wie N' bis auf $\rho_{N''}$, genauer sei $\rho_{N'}[\alpha\rangle\rho_{N''}$. ($\rho_{N''}$ entspricht einer gültiger Anfangsmarkierung, da in α keine Zeitschritte enthalten sind und insofern $\rho_{N''}( S ) \subseteq \{0,-\infty\}$).

Dann sind N' und N'' $TR_{lr}$-bisimular, insbesondere N und N'':

In N bleibt nämlich bei jedem Schalten von a die Anfangs-ID bestehen, so dass das Paar $(\rho_N, \rho_{N''})$ Element der Bisimulation von N und N' ist. Also ist durch die gleiche Relation und das Startelement $(\rho_N, \rho_{N''})$ auch eine Bisimulation zwischen $TR_{lr}( N )$ und $TR_{lr}( N'' )$ definiert.

*Schritt 3*:

Durch Entfernen aller toten Transitionen bleibt N'' natürlich bisimular zu N. Weiter sehen wir sofort, dass durch die Umformung von N' bzw. N'' auch keine Lesekanten entstehen können.

□

Damit können wir nun zeigen:

### *3.2.2.4 Satz*

Sei N das in Abbildung 16 angegebene Netz. Falls ein Netz N' mit $TR_{lr}( N' ) = TR_{lr}( N )$ und $R'=\varnothing$ existiert, so ist N' nicht asynchron.

**Beweis:**

Nehmen wir an, dass N' im Widerspruch zur Behauptung asynchron ist. Dann dürfen wir nach 3.2.2.3 für N' o.B.d.A. annehmen, dass es nur lebendige Transitionen enthält.

Man bemerkt, dass es in 1-sicheren Netzen mit ausschließlich lebendigen Transitionen keine Stellen geben kann, die Transitionen im Vor-, aber keine in ihrem Nachbereich haben oder umgekehrt. Völlig isolierte Stellen ändern aber offensichtlich sowieso nichts am Verhalten von N', so dass wir zusammenfassend o.B.d.A. annehmen dürfen:

$$S' = {}^\bullet T' = T'^\bullet \quad .$$

Anwendung der Schaltregel liefert für $TR_{lr}( N )$ zunächst einmal ein Transitionssystem, welches im wesentlichen (d.h. bis auf Bisimularität) wie folgt aussieht:

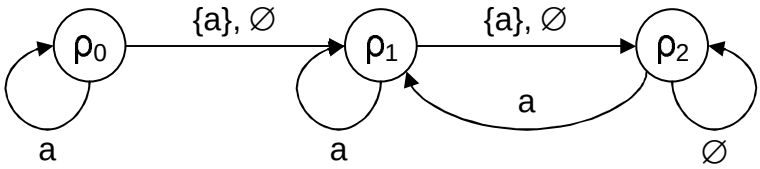



**Abbildung 18: $TR_N$**

(Das Transitionssystem $TR_N$ in Abbildung 18 geht aus dem zu $TR_{lr}( N )$ bisimularen $TR_{lrmax}( N )$ hervor, indem man drei weitere bisimulare Zustände zusammenfasst. Dadurch ist aber natürlich auch $TR_N$ zu $TR_{lr}( N )$ bisimular.)

Es enthalte nun für $i=0..2$ die Menge $R_i$ genau diejenigen Zuständen von N', die bisimular mit $\rho_i$ sind. Sei $\rho_1$' ein beliebiger Zustand aus $R_1$ und $w \in RT( N' )$ eine Sequenz mit

$$\rho_{N'} [w\rangle\rangle_r \rho_1' \quad .$$

Nun konstruieren wir als Fortsetzung von $w$ eine Sequenz w' ohne Zeitschritte, so dass für ein $\rho_1'' \in R_1$ mit $\rho_1' [w'\rangle\rangle_r \rho_1''$ gilt:

$$\rho_1''( S' ) \subseteq \{-\infty, 0\} \quad .$$

Eine solche Fortsetzung existiert, denn alle Transitionen sind lebendig, also gibt es zumindest ein w', in dem alle Transitionen einmal schalten. Da N' asynchron, braucht dieses w' auch keine Zeitschritte zu enthalten. Da $R'=\emptyset$, werden alle Stellen durch w' mindestens einmal geleert.

Sei nun w'' diejenige Sequenz, die aus w durch auslassen der Zeitschritte hervorgeht. Da N' asynchron ist, gibt es ein $\rho_1'''$ mit

$$\rho_{N'} [w''\cdot w'\rangle\rangle_r \rho_1''' \quad ,$$

und $\rho_1'''$ stimmt in der Lage der Marken mit $\rho_1''$ überein. Da in $w''\cdot w'$ aber keine Zeit vergeht, gilt auch

$$\rho_1'''( S' ) \subseteq \{-\infty, 0\}$$

und damit

$$\rho_1'' = \rho_1''' \quad .$$

Daraus folgt nun sofort, dass $\rho_0$ bisimular zu $\rho_1$ sein muss, was aber offensichtlich falsch ist, denn wie man in Abbildung 18 sieht, ist von $\rho_0$ aus die Folge $\{a\}\{a\}$ möglich, von $\rho_1$ aus nicht.

□

### 3.2.3 Zweite Substitution

Nachdem wir in 3.2.2 gesehen haben, dass wir im allgemeinen Fall selbst Bisimularität auf $TR_{lr}$ nicht unter Erhalt der Asynchronizität erreichen können, wenden wir uns nun der Sprachgleichheit von $TR_{lr}$ (mit anderen Worten der Gleichheit von RT) zu.

Wir betrachten wiederum einen Netzausschnitt wie in Abbildung 14, wobei es sich jetzt um ein rein asynchrones Netz handeln soll.

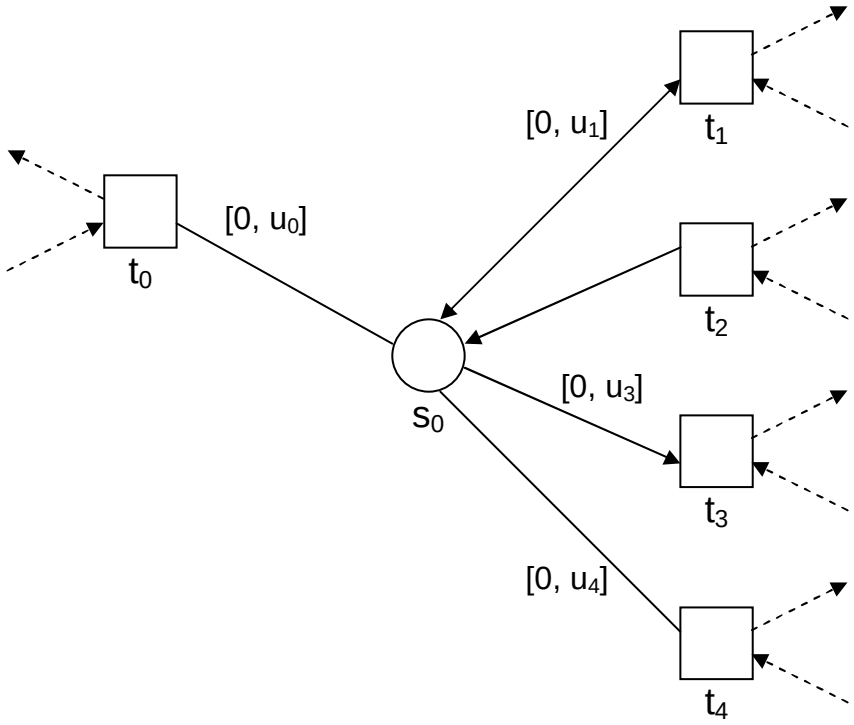



**Abbildung 19: Zu transformierender asynchroner Netzausschnitt**

Wie wir aus der vorangegangenen Diskussion wissen, stellen bestimmte $t_0$ ein Problem dar, da eine solche Transition sozusagen zwei Modi kennt: Einen, in dem das Alter der Marke auf $s_0$ noch Aktivierung bzw. Dringlichkeit beeinflusst und einen, in dem ein Voranschreiten des Alters der Marke keine Rolle mehr spielt. Dabei kann $t_0$ durch eigenes Schalten nie vom zweiten in den ersten Modus zurückgelangen. (Im asynchronen Fall beschränken sich die Überlegungen auf die Dringlichkeit.)

Im Gegensatz zu *Transform*$_1$ werden wir nun die Umgebung von $t_0$ duplizieren und damit dann beide Modi explizit nachbilden. Das Umschalten zwischen den beiden Versionen soll mittels einer internen Transition geschehen.

Betrachten wir eine Skizze des derart umgeformten Netzausschnitts. Der Übersichtlichkeit halber stellen wir das Netz in mehreren Teilen dar, die noch durch Stellenverschmelzung zusammengefügt werden müssen, hier zunächst die Umgebung von $t_0$:

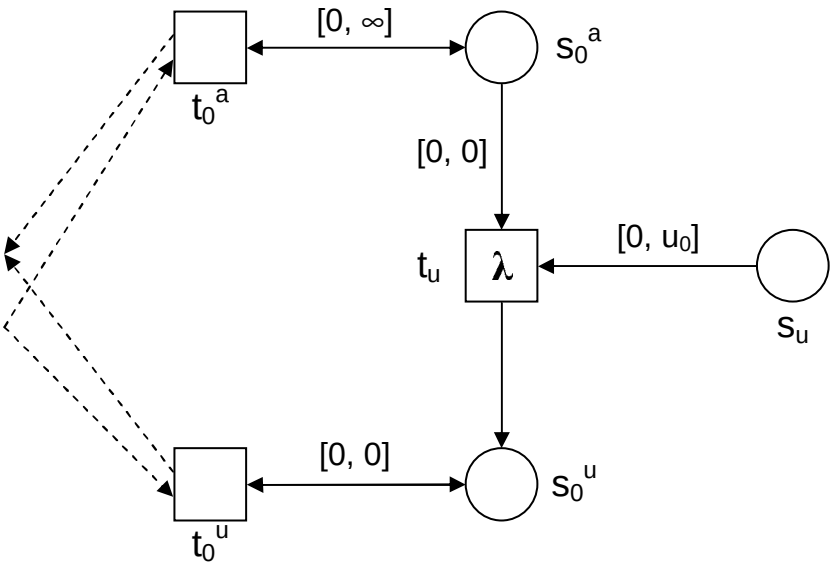



**Abbildung 20: Transform$_2$, Umgebung von $t_0$**

In unserem transformierten Netz wird:

- immer nur entweder $s_0^a$ oder $s_0^u$ markiert sein,
- $s_u$ immer genau dann markiert sein, wenn $s_0^a$ markiert ist,
- $s_0^a$ oder $s_0^u$ genau dann markiert sein, wenn $s_0$ markiert ist.

Dabei soll $t_0^a$ die Rolle von $t_0$ übernehmen für den Fall, dass die Marke auf $s_0$ jünger ist als die obere Intervallgrenze $u_0$ und $t_0^u$ für den Fall, dass sie älter ist. Für beide Fälle existieren mit $s_0^a$ und $s_0^u$ getrennte Kopien von $s_0$, welche aber konzeptionell keine Zeitinformation mehr tragen, sondern lediglich der Steuerung des Kontrollflusses dienen. Die Stelle $s_0$ selbst wird auch beibehalten (siehe Abbildung 21 - Abbildung 24) und, wie schon weiter oben, verwendet, um die anderen adjazenten Transitionen zu steuern. Sie darf natürlich wiederum von $t_0^a$ und $t_0^u$ nicht beeinflusst werden. Die $s_u$ und $t_u$ schließlich dienen dazu, zwischen $t_0^a$ und $t_0^u$ umzuschalten. Durch die obere Intervallgrenze $u_0$ auf $(s_u, t_u)$ findet dieses Umschalten (spätestens) dann statt, wenn im ursprünglichen Netz die Lesekante $(s_0, t_0)$ Dringlichkeit signalisiert. Da wir die Asynchronizität des ursprünglichen Netzes erhalten wollen, müssen wir allerdings in Kauf nehmen, dass $t_u$ zu früh schalten kann. Dies beeinträchtigt nicht die wechselseitige Simulation, insbesondere auch nicht die RT-Äquivalenz, wohl aber die Bisimulation.

Bemerkung: Offenbar lässt sich die Konstruktion bei Verzicht auf Asynchronizität leicht derart verändern, dass sie bisimular wird. (Man setze einfach $lb(s_u, t_u) := u_0$.) Allerdings ist es bei nicht-asynchronen Netzen für die Iteration der Konstruktion natürlich nötig, allgemein untere Intervallgrenzen ungleich 0 zuzulassen, was zu weiteren Komplikationen führt. Da unser Hauptaugenmerk aber sowieso nicht auf den getakteten Netzen liegt, wird eine solche Konstruktion im Rahmen dieser Arbeit zwar in 3.2.4 angegeben, aber nicht mehr eingehend untersucht. (Geht man allerdings von einem asynchronen Ursprungsnetz aus, lässt sich die Transformation auch unverändert durchführen, siehe ebenfalls 3.2.4.)

Betrachten wir aber nun die verschiedenen Fälle zu $s_0$ adjazenter Transitionen von $t_1$ bis $t_4$ und deren Behandlung durch *Transform$_2$*. Auch diese müssen (mit Ausnahme von $t_2$ und $t_4$)

für beide Modi von $t_0$ repliziert werden, um jeweils für beide Fälle $s_0^a$ und $s_0^u$ korrekt zu behandeln:

Für $t_1$, welches im Vor- und Nachbereich von $s_0$ ist, müssen beide Versionen jeweils Marken auf $s_0^a$, $s_0$ und $s_u$ legen. Sie unterscheiden sich jedoch darin, dass $t_1^a$ die Marken aus Stellen im Vorbereich von $t_0^a$ nimmt und $t_1^u$ von Stellen im Vorbereich von $t_0^u$. Von $s_u$ kann nur $t_1^a$ eine Marke nehmen, denn wenn $t_1^u$ aktiviert ist, liegt dort nie eine Marke.

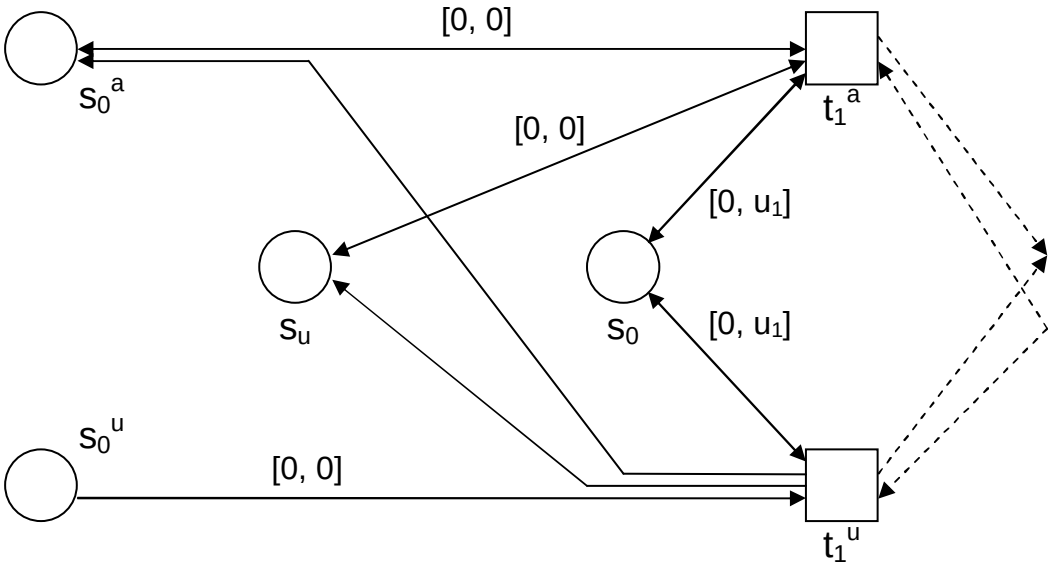



**Abbildung 21: Transform$_2$, Umgebung von $t_1$**

Die Fälle $t_2$ und $t_3$ sind im wesentlichen jeweils nur die entsprechenden Teile aus obigem Netz, die für Vor- und Nachbereich zuständig sind. Wir wollen sie der Vollständigkeit halber trotzdem kurz aufführen.

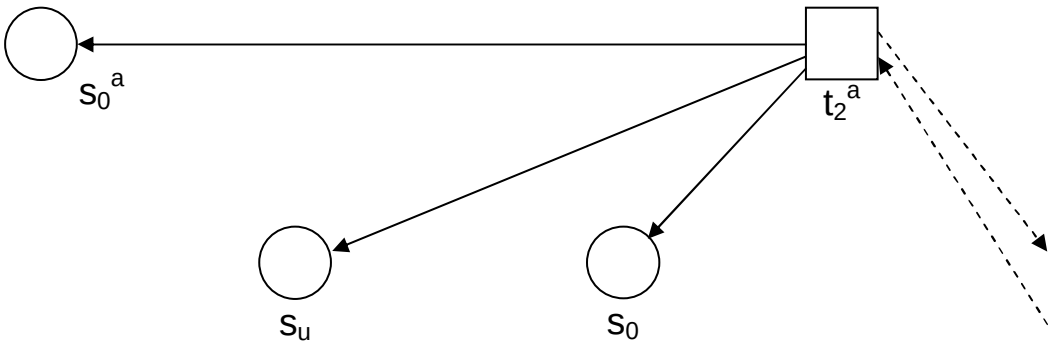



**Abbildung 22: Transform$_2$, Umgebung von $t_2$**

Man beachte für $t_2$, dass kein $t_2^u$ benötigt wird, weil sowieso keine Verbindung zu $s_0^u$ existiert, denn eine frische Marke wird ja immer auf $s_0^a$ gelegt.

Für $t_3$ hingegen ist es nötig, sowohl das Konsumieren einer Marke auf $s_0^a$ als auch einer auf $s_0^u$ vorzusehen:

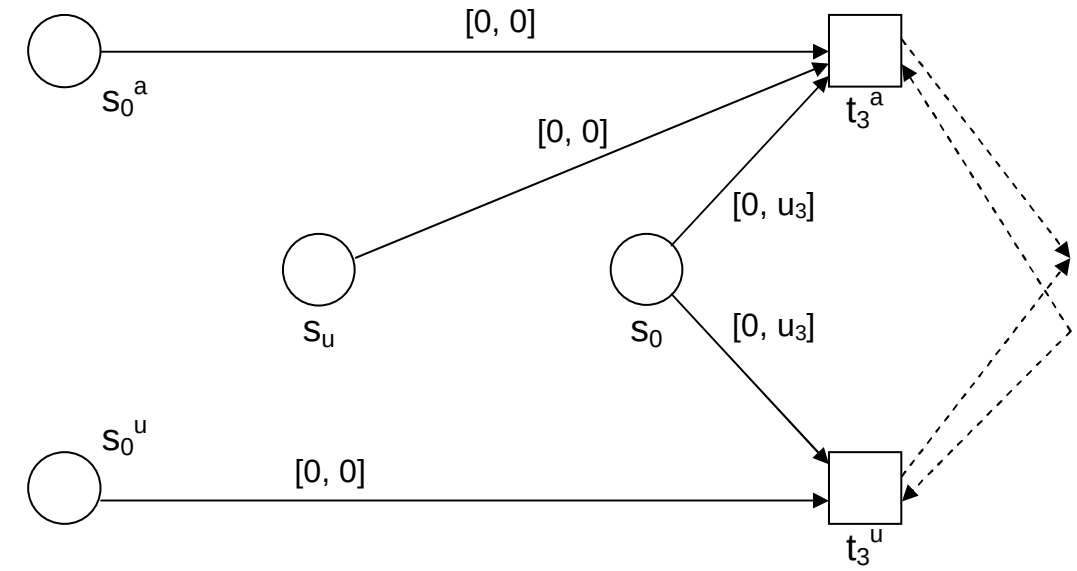



**Abbildung 23: Transform$_2$, Umgebung von $t_3$**

Der letzte Fall schließlich gestaltet sich am übersichtlichsten, denn da $t_4$ die Marke auf $s_0$ im ursprünglichen Netz auf keinen Fall verändert, reicht es auch im transformierten Netz, lediglich das Vorhandensein der Marke auf $s_0$ zu prüfen. Dafür reicht aber natürlich eine einzige Transition aus, so dass wir $t_4$ effektiv unberührt lassen:

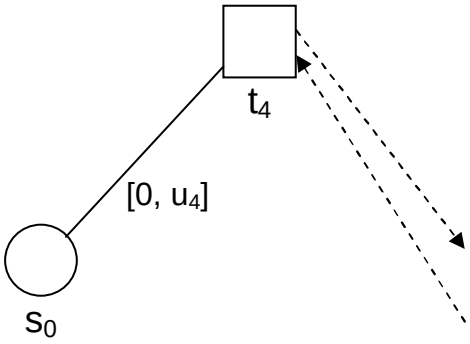



**Abbildung 24: Transform$_2$, Umgebung von $t_4$**

Um die Korrektheit der Transformation zu beweisen, sei im folgenden noch die formale Definition angegeben.

### *3.2.3.1 Definition*

Für ein Netz $N = (S, T, F, R, l, M_N, lb, ub)$ mit $lb \equiv 0$, eine Stelle $s_0$ und eine Transition $t_0$ mit $(s_0, t_0) \in R$ sei $N' = Transform_2(N, (s_0, t_0))$ wie folgt definiert:

$$T^{\sim} = s_0 (F \cup F^{-1}) \cup \{ t_0 \}$$
$$T^a = \{ t^a \mid t \in T^{\sim} \}$$
$$T^u = \{ t^u \mid t \in s_0 F \} \cup \{ t_0^u \}$$
$$T' = (T \setminus T^{\sim}) \cup T^a \cup T^u \cup \{ t_u \}$$

(Die Vereinigungen seien alle disjunkt. Man beachte, dass Transitionen $\neq t_0$, die mit $s_0$ über eine Lesekante verbunden sind, nicht verändert werden.)

$$
\begin{array}{lll}
l'(t') & := & l(t') \quad \text{für } t' \in T \setminus \tilde{T} \\
 & := & l(t) \quad \text{für } t' = t^a \vee t' = t^u \text{ mit } t' \in \tilde{T} \\
 & := & \lambda \quad \text{für } t' = t_u
\end{array}
$$

(Die neuen Transitionen $t^a$ und $t^u$ werden also so beschriftet wie die Transitionen t, die sie ersetzen. Die neue Transition $t_u$ ist intern.)

$$
\begin{array}{ll}
S' & := S \cup \{ s_0^a, s_0^u, s_u \} \\
M_N' & := M_N \cup \{ s_0^a, s_u \mid s_0 \in M_N \} \\
R' & := R \setminus \{ (t_0, s_0) \}
\end{array}
$$

(d.h. Weglassen der zu ersetzenden Lesekante)

$$
\begin{array}{ll}
F' = & ( F \cap ( (T' \times S) \cup (S \times T') ) ) \\
\cup & \{ (t^a, s_0), (t^a, s_0^a), (t^a, s_u) \mid (t, s_0) \in F \wedge t \neq t_0 \} \\
\cup & \{ (t^u, s_0), (t^u, s_0^a), (t^u, s_u) \mid (t, s_0) \in F \cap F^{-1} \wedge t \neq t_0 \} \\
\cup & \{ (s_0, t^a), (s_0, t^u), (s_0^a, t^a), (s_0^u, t^u), (s_u, t^a) \mid (s_0, t) \in F \wedge t \neq t_0 \} \\
\cup & \{ (t_0^a, s_0^a), (s_0^a, t_0^a), (t_0^u, s_0^u), (s_0^u, t_0^u) \} \\
\cup & \{ (s_u, t_u), (s_0^a, t_u), (t_u, s_0^u) \}
\end{array}
$$

(d.h. Verbinden aller $t^a$ und $t^u$ mit $s_0$ wie die früheren t's, zusätzlich legen alle im Vorbereich von $s_0$ auch Marken auf $s_0^a$ und $s_u$. Weiter nehmen alle $t^a$ im Nachbereich von $s_0$ auch Marken aus $s_0^a$ und $s_u$, alle $t^u$ im Nachbereich von $s_0$ aus $s_0^u$. Zusätzlich werden $s_0^a$ und $s_u$ der Vorbereich von $t_u$ und $s_0^u$ der Nachbereich.)

$$
\begin{array}{lllll}
lb' \equiv 0 & & & & \\
ub'|_{F' \cap F} & := ub|_{F' \cap F} & & & \\
ub'(s_0, t^a) & = ub'(s_0, t^u) & := ub(s_0, t) & & \text{für } (s_0, t) \in F \wedge t \neq t_0 \\
ub'(s_0^a, t^a) & = ub'(s_0^u, t^u) = ub'(s^u, t^a) & := 0 & & \text{für } (s_0, t) \in F \wedge t \neq t_0 \\
ub'(s_0^a, t_0^a) & := \infty & (*) & & \\
ub'(s_0^u, t_0^u) & := 0 & & & \\
ub'(s_u, t_u) & := ub(s_0, t_0) & & &
\end{array}
$$

(d.h. Zeitsteuerung aller $t^a$ und $t^u$ nur durch $s_0$ genau wie die früheren t's, $t_0^a$ wird nie dringend, $t_0^u$ kann von $s_0(^u)$ nie aufgehalten werden, und $t_u$ wird nach der Zeit dringend, nach der in N $t_0$ dringend wird.)

□

Bemerkung: Will man durch *Transform*$_2$ die Klasse der Netze ohne lazy-arcs erhalten, so kann man bei (*) auch

$$ub'(s_0^a, t_0^a) := ub(s_u, t_u) + 1$$

setzen. Denn jede Transition, die eine Marke auf $s_u$ legt, legt auch eine auf $s_0^a$, so dass für zwei Marken auf $s_u$ und auf $s_0^a$ diejenige auf $s_0^a$ nie älter sein kann. Das folgende Lemma zeigt noch, dass entweder beide oder keine der beiden Stellen markiert sind. Betrachten wir dann Abbildung 20, so sehen wir, dass $t_u$ immer $t_0^a$ deaktivieren muss, bevor $t_0^a$ dringend werden kann. (Insbesondere ist $t_u$ intern und kann somit auch in RFS nicht aufgehalten werden.)

### *3.2.3.2 Lemma*

Seien N und N' wie oben und $u_0 := ub(s_0, t_0)$. Dann ist für jede erreichbare Markierung ρ' von N' genau eine der folgenden Aussagen wahr:

$$
\begin{array}{ll}
(Z1) & \rho'(s_0) = \rho'(s_u) = \rho'(s_0^a) = \rho'(s_0^u) = -\infty \\
(Z2) & -\infty < \rho'(s_0) \le u_0 \\
 & \wedge\, \rho'(s_u) = \rho'(s_0) \\
 & \wedge\, -\infty \neq \rho'(s_0^a) \le \rho'(s_u) \\
 & \wedge\, \rho'(s_0^u) = -\infty \\
(Z3) & \rho'(s_0) \neq -\infty \\
 & \wedge\, \rho'(s_u) = -\infty \\
 & \wedge\, \rho'(s_0^a) = -\infty \\
 & \wedge\, \rho'(s_0^u) \neq -\infty
\end{array}
$$

**Beweis:**

Die Fälle sind offensichtlich paarweise disjunkt. Man führt nun für jeden der Fälle Fallunterscheidungen durch und zeigt durch die Schaltregel, dass nach jedem Schalten in RFS wieder einer der drei Fälle vorliegt. Nach Definition 3.2.3.1 gilt für $\rho_N$ entweder (Z1) oder (Z2). Der Rest folgt durch Induktion.

□

Für das folgende mag es hilfreich sein, sich kurz die intuitive Bedeutung der drei Fälle vor Augen zu führen:

- (Z1) gilt, wenn $s_0$ nicht markiert ist. Damit ist auch keine der neuen Stellen $s_u$, $s_0^a$ oder $s_0^u$ markiert.
- (Z2) gilt, wenn $s_0$ markiert ist und $t_0$ im nicht-dringenden Modus simuliert wird (d.h. ihre Funktion von $t_0^a$ übernommen wird). Dann ist neben $s_0$ auch $s_0^a$ markiert und außerdem $s_u$, welches die Zeit zählt, zu der $t_0$ (spätestens) in den dringenden Modus wechseln muss.
- In (Z3) hat die Simulation von $t_0$ dann (durch Schalten von $t_u$) in den dringenden Modus gewechselt. Jetzt sind $s_u$ und $s_0^a$ unmarkiert, $s_0$ und $s_0^u$ hingegen markiert. Das Alter der Marke auf $s_0^u$ ist dabei unerheblich, da $lb(s_0^u, .) \equiv ub(s_0^u, .) \equiv 0$.

Wir werden nun Simulationsrelationen in beide Richtungen aufstellen. Dadurch, dass wir die Invarianten (Z1)-(Z3) bereits allgemein gezeigt haben, fallen diese sehr einfach aus.

### *3.2.3.3 Definition*

Es seien N und N' die Netze von oben und $u_0 := u(s_0, t_0)$.

Für zwei Markierungen ρ und ρ' sei die folgende Relation $B_\rightarrow$ definiert:

$$\rho\, B_\rightarrow \rho' \Leftrightarrow (B1) \wedge (B2)$$

wobei

$$
\begin{array}{ll}
(B1) & \forall s \in S: \rho(s) = \rho'(s) \\
(B2) & \rho(s_0) \le u_0 \;\Leftrightarrow\; \rho'(s_u) = \rho(s_0) \quad .
\end{array}
$$

□

### *3.2.3.4 Lemma*

Es seien N und N' wie oben. Dann ist $B_{\rightarrow}$ eine Simulation von $TR_{lr}( N )$ nach $TR_{lr}( N' )$.

**Beweis:**

Wir verwenden für den Beweis die Charakterisierung einer Simulation nach 1.3.1.5.

Seien also $\rho_1, \rho_1', \rho_2$ erreichbare Markierungen (und damit Zustände in $TR_{lr}( N )$ bzw. $TR_{lr}( N' )$) mit $\rho_1 \, B_{\rightarrow} \rho_1'$. Es sei $\varepsilon \in \Lambda \cup \{\lambda\} = \Sigma \cup \wp(\Sigma) \cup \{\lambda\}$. Dann ist zu zeigen:

$$\rho_1 \xrightarrow{\varepsilon} \rho_2 \;\Rightarrow\; \exists \rho_2' : \rho_1' \xrightarrow{\varepsilon} \rho_2' \;\wedge\; \rho_2 \, B_{\rightarrow} \rho_2' \quad .$$

Wir unterscheiden folgende Fälle:

$\varepsilon \in \Sigma \cup \{\lambda\}$:

Dann können wir aufgrund der Definition von $TR_{lr}$ in 1.4.6 eine Transition t wählen, so dass $l(t) = \varepsilon$ und $\rho_1 [t\rangle_r \rho_2$.

Wir unterscheiden damit weiter:

$t \notin (F \cup F^{-1})s_0$:

Dann gilt nach Konstruktion $t \in T'$, $t \notin (F' \cup F'^{-1})(S' \setminus S)$. Somit folgen $\rho_1' [t\rangle_r$ und weiter aus (B1) und (B2) für $\rho_1$ und $\rho_2$ sofort wieder (B1) und (B2) für $\rho_1'$ und $\rho_2'$.

(Man beachte, dass dieser Fall auch die übrigen mit $s_0$ verbundenen Lesekanten enthält.)

$t \in (F \cup F^{-1})s_0 \wedge t \neq t_0$:

Nach 3.2.3.2 befinden wir uns in einem der Fälle (Z1)-(Z3):

Für (Z1) gilt wegen (B1) insbesondere $\rho_1( s_0 ) = -\infty$, damit muss offenbar $t \in (F \setminus F^{-1})s_0$ gelten (siehe Abbildung 22). Damit ist in N' das korrespondierende $t^a$ aktiviert, nach dem Schalten sind wir in (Z2), und (B1) und (B2) gelten wie oben. Nach Konstruktion gilt offenbar auch $l(t^a) = \varepsilon$.

Für (Z2) liefert (B1) $-\infty < \rho_1( s_0 ) \leq u_0$. Wie man sieht, kommen für t die Fälle $t \in (F^{-1} \setminus F)s_0$ (siehe Abbildung 23) und $t \in (F \cap F^{-1})s_0$ (Abbildung 21) in Frage. Wir können in N' in beiden Fällen wieder das korrespondierende $t^a$ schalten, wobei $l(t^a) = \varepsilon$. Man prüft mit der Schaltregel jeweils leicht nach, dass (B1) und (B2) noch gelten.

Im letzten Fall (Z3) erhalten wir aus (B1) $\rho_1( s_0 ) > u_0$. Es kommen die gleichen Transitionen t wie im Fall (Z2) in Frage, nur ist für diese statt $t^a$ nun $t^u$ mit $l(t^u) = \varepsilon$ aktiviert. Man prüft wiederum leicht (B1) und (B2) für die verschiedenen Fälle.

$t = t_0$:

Dann ist in N die Stelle $s_0$ markiert und wir befinden uns in (Z2) oder (Z3). Für (Z2) kann $t_0^a$ schalten und man prüft wiederum (B1) und (B2) nach Schaltregel. Für (Z3) ebenso mit $t_0^u$.

$\varepsilon = X \subset \Sigma$:

Wir zeigen für diesen Fall zunächst, dass für jede dringende Transition $t \neq t_u$ in N' eine gleichbeschriftete und ebenfalls dringende Transition in N existiert.

Falls $t \notin (F' \cup F'^{-1}) s_0$ und $t \notin (t_0^a, t_0^u, t_u)$, so gilt ja auch $t \in T$, und nach Konstruktion und (B1) ist t dringend in N gdw. es in N' dringend ist.

Falls $t \in (F' \cup F'^{-1}) s_0$, so existiert eine Transition $r \in T$ mit $t = r^a$ oder $t = r^u$. Falls $r \notin s_0 F$, so auch $t \notin s_0 F'$, und nach Konstruktion und (B1) ist r dringend in N. Falls $r \in s_0 F$, so $t \in s_0 F'$ und es folgt aus der Tatsache, dass t dringend ist, schon $\rho_1'( s_0 ) \geq ub'( s_0, t )$. Dann gilt aber nach Konstruktion und (B1) auch $\rho_1( s_0 ) \geq ub'( s_0, t ) = ub( s_0, r )$ und r ist in N ebenfalls dringend.

Für $t = t_0^a$ sehen wir direkt, dass es nie dringend werden kann. Für $t = t_0^u$ folgt aus der Aktivierung mit 3.2.3.2, dass wir uns in (Z3) befinden und weiter mit (B1), (B2), dass $\rho_1( s_0 ) > u_0$, also $t_0$ dringend in N.

Solange $t_u$ also nicht dringend ist, können wir auch in N' den Schritt X direkt ausführen, weil dort nicht mehr Aktionen dringend sein können als in N. Offenbar gilt danach wieder (B1) und die rechte Seite von (B2) ist wahr. Die linke Seite muss aber auch wahr sein, sonst wäre $t_u$ ja doch dringend gewesen.

Das einzige Problem könnte also ein dringendes $t_u$ darstellen. In diesem Fall ist $\rho'( s_u ) \geq u_0$, und da $t_u$ intern ist sogar $\rho'( s_u ) = u_0$. Dann wissen wir aber aus (Z2) und (B1), dass $\rho( s_0 ) = u_0$. Wir simulieren nun den X-Schritt aus N, indem wir in N' die Sequenz $t_u X$ ausführen, was nach Schaltregel möglich ist. Nach deren Ausführung sind ferner auch beide Seiten von (B2) falsch, womit (B2) gilt. Offensichtlich gilt auch (B1).

□

### *3.2.3.5 Definition*

Es seien die Netze N und N' wie oben.

Für zwei Markierungen ρ von N und ρ' von N' sei die folgende Relation $B_{\leftarrow}$ definiert:

$$\rho' \, B_{\leftarrow} \rho \quad \Leftrightarrow \quad (B3)$$

wobei:

$$(B3) \qquad \forall s \in S: \rho( s ) = \rho'( s ) \quad .$$

□

### *3.2.3.6 Lemma*

Es seien N und N' wie oben. Dann ist $B_{\leftarrow}$ eine Simulation von $TR_{lr}( N' )$ nach $TR_{lr}( N )$.

**Beweis:**

Wie oben verwenden wir 1.3.1.5. Seien dazu $\rho_1, \rho_2, \rho_2'$ erreichbare Markierungen (und damit Zustände in $TR_{lr}( N )$ bzw. $TR_{lr}( N' )$)mit $\rho_1' \, B_{\leftarrow} \rho_1$. Es sei $\varepsilon \in \Lambda \cup \{\lambda\} = \Sigma \cup \wp(\Sigma) \cup \{\lambda\}$. Dann ist zu zeigen:

$$\rho_1' \xrightarrow{\varepsilon} \rho_2' \quad \Rightarrow \quad \exists \rho_2: \rho_1 \overset{\varepsilon}{\twoheadrightarrow} \rho_2 \quad \wedge \quad \rho_2' \, B_{\leftarrow} \rho_2 \, .$$

Wir unterscheiden wieder folgende Fälle:

<u>$\varepsilon \in \Sigma \cup \{\lambda\}$</u>:

Dann können wir aufgrund der Definition von $TR_{lr}$ in 1.4.6 eine Transition t in N' wählen, so dass $l'(t') = \varepsilon$ und $\rho_1'[t\rangle_r \rho_2'$.

Für $t \notin \{t_0^a, t_0^u, t_u\}$ unterscheidet man wieder die üblichen Fälle gemäß Konstruktion und zeigt jeweils aus (Z1-Z3) und (B3) einfach, dass die korrespondierende Transition in N aktiviert ist und (B3) nach dem Schalten erhalten bleibt.

Für $t' \in \{t_0^a, t_0^u\}$ wissen wir aus 3.2.3.2 und (B3), dass $\rho_1(s_0) > -\infty$, somit ist $t_0$ in N aktiviert. Die Schaltregel liefert uns dann für $\rho_1[t_0\rangle_r\rho_2$ die Gleichung $\rho_1(s_0) = \rho_2(s_0)$ und somit $\rho_2' B_{\leftarrow} \rho_2$.

Den Fall $t=t_u$ simulieren wir in N dadurch, dass wir gar nichts tun. Offenbar gilt damit weiterhin (B3).

$\varepsilon = X \subset \Sigma$:

Wir simulieren die Verweigerung von X in N' einfach durch Verweigern derselben Menge in N. Dazu vergewissern wir uns, dass für jede Transition t, die in N dringend ist, eine gleichbeschriftete Transition in N' dringend ist. Für $t \neq t_0$ ist dies aufgrund der Konstruktion und (B3) trivial. Für $t=t_0$ kommt von den Fällen in 3.2.3.2 nur (Z3) in Frage. ( (Z2) scheidet aus, weil dort das interne $t_u$ dringend wäre und N' keinen Schritt X durchführen könnte.) Für (Z3) aber ist $t_0^u$ dringend, weil $s_0^u$ markiert ist.

Nach Schaltregel gilt natürlich auch $\rho_2' B_{\leftarrow} \rho_2$.

□

Bemerkung: Das verwunderliche an obigem Beweis mag vielleicht sein, dass ein vorzeitiges Schalten von $t_u$ (d.h. bevor es dringend wird) kein Problem darstellt. Dies liegt aber einfach daran, dass zwar $t_0^a$ dadurch deaktiviert wird, $t_0^u$ aber die Rolle von $t_0^a$ problemlos übernehmen kann. Ein Problem könnte nur bei den Verweigerungsmengen auftauchen (denn $l'(t_0^a) = l'(t_0^u) = l(t_0)$ kann dadurch nicht mehr verweigert werden). Dies tritt aber auch nicht auf, da ja N' bei der Simulation „den Ton angibt" und insofern die fehlende Verweigerungsmenge nicht ins Gewicht fällt (siehe auch Bemerkung nach 3.2.3.7).

### *3.2.3.7 Korollar*

Die Netze N und N' wie oben sind RT-äquivalent.

**Beweis**:

3.2.3.4 und 3.2.3.6, zusammen mit der Bemerkung im Anschluss an 1.3.1.6.

□

Genauer betrachtet geht die Beziehung der beiden Netze sogar noch über RT-Äquivalenz und wechselseitige Simulation hinaus: Wir stellen nämlich fest, dass die Bedingung B3 aus 3.2.3.5 genau dem B1 aus 3.2.3.3 entspricht. Damit liegt der Fall vor, dass $B_{\leftarrow}^{-1} \subseteq B_{\rightarrow}$, d.h. die Rückrichtung der wechselseitigen Simulation ist in der Vorwärtsrichtung enthalten. (Eine solche Situation wird unter anderem in [AFGI03] unter der Bezeichnung *2-nested-simulation* allgemeiner untersucht.)

Informell sind wir damit noch einen Schritt näher an der Bisimulation als bei einer bloßen wechselseitigen Simulation, da nur noch einer der Simulationspartner aus dem gemeinsamen Verhalten ausbrechen kann – im vorliegenden Fall nämlich das transformierte Netz N', welches durch verfrühtes Schalten von $t_u$ die Bedingung B2 zerstören kann.

In einer wirklichen Bisimulation müsste N' nach einem frühen Schalten von $t_u$ (d.h. in einer Situation mit $\rho(s_0) \leq u_0$) immer noch alle RT-Folgen ausführen können, die N

ausführen kann. N kann aber unter Umständen jetzt noch die Aktion $l(t_0)$ verweigern, N' dagegen nicht mehr, da $t_0^u$ nach Schalten von $t_u$ auf jeden Fall dringend ist.

#### *3.2.3.8 Lemma*

Für ein asynchrones Netz N ist *Transform₂( N )* ebenfalls asynchron.

**Beweis:**

klar nach Konstruktion.

□

#### *3.2.3.9 Theorem*

Für jedes asynchrone Netz N gibt es ein asynchrones N' ohne Lesekanten, so dass *RT( N ) = RT( N' )*.

**Beweis:**

Durch wiederholte Anwendung von *Transform₂* können iterativ alle Lesekanten entfernt werden. Nach 3.2.3.8 bleiben dabei die Voraussetzungen erhalten.

□

### 3.2.4 Verallgemeinerung von Transform₂

Obwohl synchrone Netze, wie wir bereits bemerkt haben, nicht von besonderem Interesse für uns sind, wollen wir die zentralen Gedanken zur Verallgemeinerung von *Transform₂* noch skizzieren:

Wie im Anschluss an 3.2.3.6 bemerkt, können wir mit *Transform₂* im allgemeinen keine Bisimulation auf r-Erreichbarkeitsgraphen erreichen. Wir haben jedoch schon darauf hingewiesen, dass man das verfrühte Schalten von $t_u$ leicht verhindern kann, wenn man *Transform₂'* wie *Transform₂* definiert bis auf:

$$lb( s_u, t_u ) := ub( s_u, t_u ) = ub( s_0, t_0 ) \quad .$$

Damit verliert man allerdings das für die Iteration der Substitution wichtige Lemma 3.2.3.8. Will man das Hauptresultat für diese neue Konstruktion retten, müsste man sich also automatisch damit auseinandersetzen, ob für allgemeine (nicht notwendigerweise asynchrone) Netze eine ähnliche Konstruktion existiert. Diese existiert auch tatsächlich und wird im Mittelpunkt dieses Abschnitts stehen. Bevor wir diese Verallgemeinerung nun vorstellen, wollen wir allerdings noch mit einer anderen Überlegung zeigen, dass wir sie für den Fall, in dem wir von einem asynchronen Netz ausgehen, tatsächlich auch *Transform₂'* iterieren können:

Die in einem Iterationsschritt eingeführte Stelle $s_u$ zeichnet sich nämlich dadurch aus, dass sie keine inzidenten Lesekanten besitzt, ebenso die neue Transition $t_u$. Damit ist *Transform₂'* auch noch auf solchen Netzen wohldefiniert, die durch *Transform₂'* aus asynchronen Netzen entstehen. Setzen wir dieses Argument durch Induktion fort, so sehen wir, dass wir durch wiederholte Anwendung von *Transform₂'* ebenso wie mit *Transform₂* alle Lesekanten entfernen können und diesmal ein sogar bisimulares, allerdings nicht mehr asynchrones Netz erhalten.

Wenden wir uns aber nun dem allgemeinen Fall zu, indem wir zu Beginn kein asynchrones Netz voraussetzen. Wir befinden uns also wieder im Fall von Abbildung 14. Wir verändern die Transformation der Umgebung von $t_0$ wie folgt:

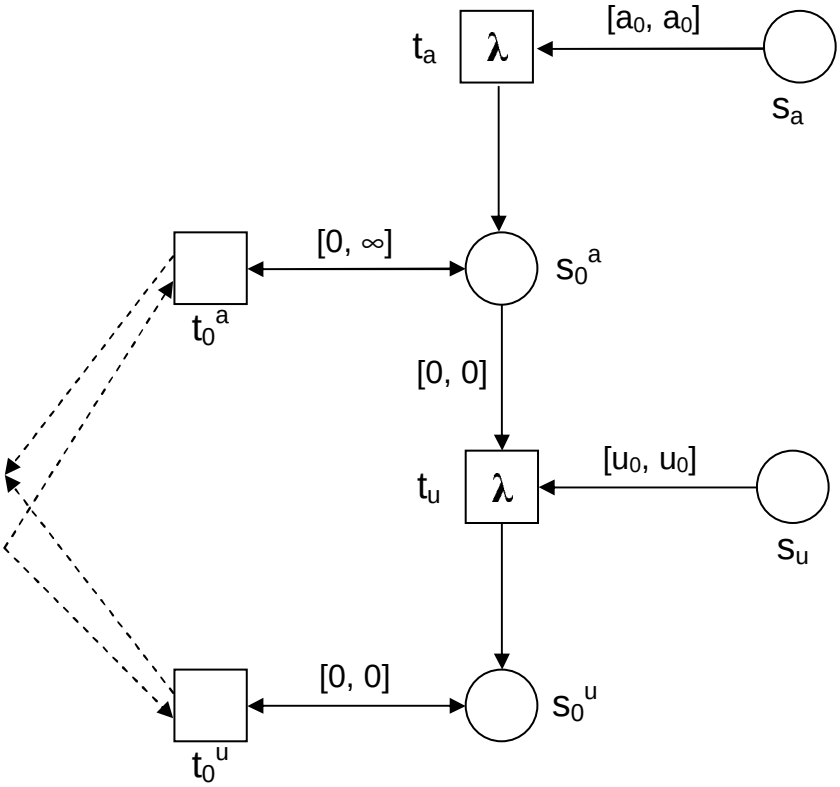



**Abbildung 25: Transform$_3$, Umgebung von $t_0$**

Dabei haben wir einerseits die Aktivierungsgrenze von $t_u$ wie oben geschildert auf $u_0$ angehoben, andererseits aber auch eine neue Transition $t_a$ eingeführt, welche wie $t_0$ funktioniert, nur mit dem Unterschied, dass sie für die Aktivierung zuständig ist statt für die Dringlichkeit. Aus den zwei Schaltmodi von Transform$_2$ sind nun also drei geworden.

Dem müssen natürlich auch die Transformationen der anderen zu $s_0$ adjazenten Transitionen Rechnung tragen, hier am Beispiel von $t_1$:

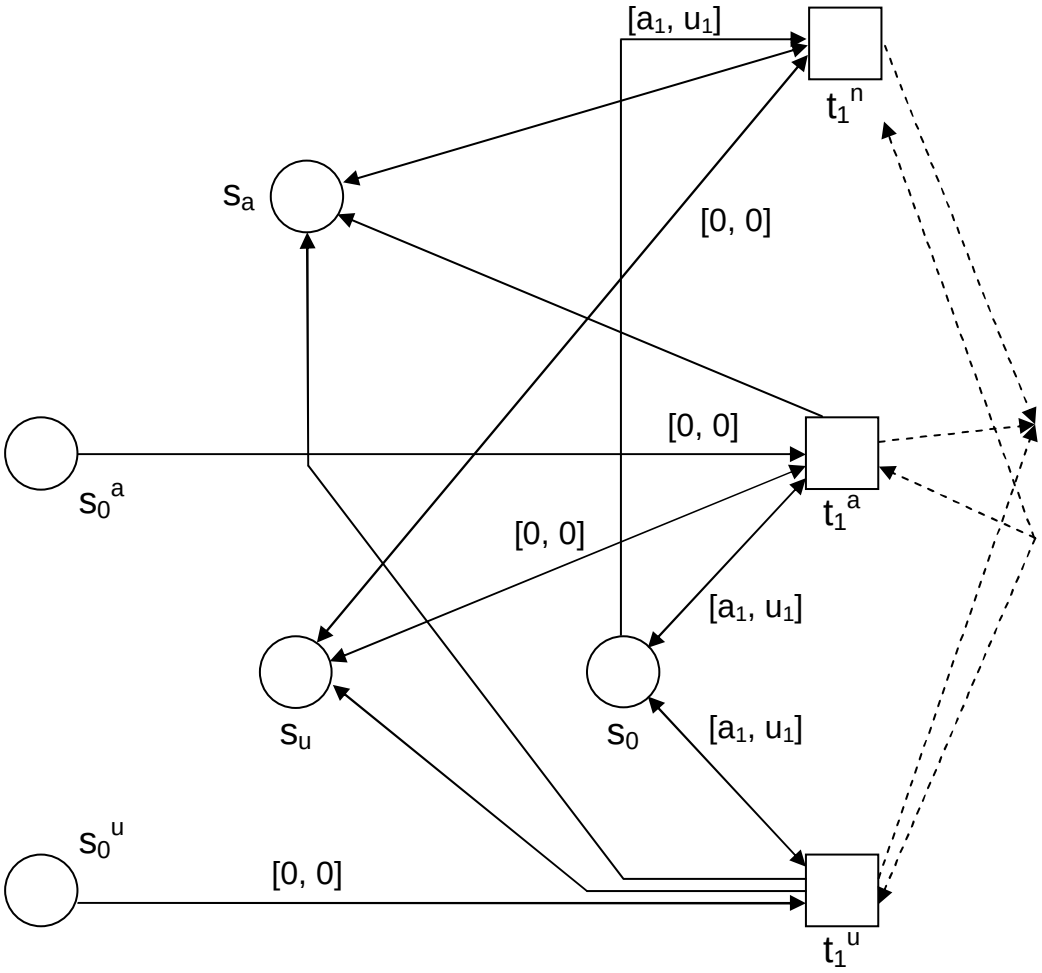



**Abbildung 26: Transform$_3$, Umgebung von $t_1$**

Dabei ersetzt $t_1^n$ die Transition $t_1$ im ersten Fall, in dem $t_0$ noch gar nicht aktiviert ist, $t_1^a$ wie gehabt für den Fall, dass $t_0$ aktiviert, aber noch nicht dringend ist und $t_1^u$ für den Fall, dass $t_0$ dringend ist. Für jedes Schalten von $t_1$ in N werden dabei in N' alle Stellen in Abbildung 25 neu initialisiert. Die Fälle $t_2$ und $t_3$ vereinfachen sich wieder ähnlich wie bei *Transform$_2$*, und der Fall der Lesekante *( $s_0$, $t_4$ )* bleibt sogar wörtlich erhalten. Wir wollen diese Konstruktion *Transform$_3$* nennen.

*Transform$_3$* führt nun tatsächlich einerseits auf Netze, deren zugeordnete $TR_{lr}$ bisimular sind und ist andererseits auch ohne Rücksicht auf Asynchronizität anwendbar. Allerdings ist die für uns zentrale Klasse der asynchronen Netze nicht unter dieser Konstruktion stabil. Die Korrektheit von *Transform$_3$* werden wir nicht mehr im Detail beweisen, die Ideen dazu folgen den weiter oben in diesem Kapitel vorgestellten.

## *3.3 Ergebnis*

Wir haben in Kapitel 3 nun mehrere Substitutionskonstruktionen kennengelernt, welche es alle unter ihren jeweiligen Voraussetzungen erlauben, Netze mit Lesekanten in solche ohne Lesekanten mit in gewissem Sinne gleichem Verhalten zu transformieren. Es hat sich aber gezeigt, dass keine Transformation gleichzeitig allen Wünschen gerecht werden kann.

Abschließend wollen wir deshalb die verschiedenen Voraussetzungen, die Eigenschaften und die jeweiligen Äquivalenzbegriffe zusammenfassen:

**Transform$_1$**

- setzt wohlgesteuerte, aber nicht notwendigerweise asynchrone Netze voraus,
- erhält Asynchronizität,
- $TR_r$-bisimular, insbesondere wird T erhalten.

**Transform$_2$**

- setzt asynchrone Netze voraus,
- erhält Asynchronizität,
- *2-nested-simulation* auf $TR_{lr}$, insbesondere RT-äquivalent.

**Transform$_2$'**

- setzt asynchrone Netze voraus,
- erhält *nicht* die Asynchronizität,
- $TR_{lr}$-bisimular, jedoch nicht $TR_r$-bisimular.

**Transform$_3$**

- beliebige PTT-Netze,
- erhält *nicht* die Asynchronizität,
- $TR_{lr}$-bisimular, jedoch nicht $TR_r$-bisimular.

Mit dieser Übersicht endet der theoretisch orientierte erste Teil dieser Arbeit. Im nun folgenden zweiten Teil stellen wir Aufbau und Arbeitsweise eines Tools/Frameworks vor, welches auf der vorgestellten Klasse der PTT-Netze operiert.

# Teil II Softwaretechnik/FastAsy

# Kapitel 4 Problemstellung

Ausgehend von der sehr konkreten Problemstellung in [Bih98] (siehe aber auch [BV98]), für eine fest gegebene Charakterisierung zeitlicher Effizienz in Form spezieller Refusal-Traces Vergleiche zwischen zwei Petrinetzen durchzuführen, haben sich die Anforderungen an unser Tool *FastAsy* durch Anwendung auf verschiedene Probleme und durch Anregungen aus theoretischen Überlegungen, wie sie unter anderem im ersten Teil der vorliegenden Arbeit dargestellt werden, erheblich ausgeweitet:

- Wir möchten verschiedene Klassen von Petrinetzen mit FastAsy behandeln. Abseits von der theoretischen Vorarbeit, die jeweils nötig ist, um die Gültigkeit unseres Ansatzes für neue Netzklassen zu zeigen, soll uns das Framework von FastAsy möglichst viel Arbeit abnehmen, wenn es darum geht, neue Netzversionen zu integrieren. Auch die Möglichkeit, Systeme zu betrachten, die in einem ganz anderen Formalismus notiert sind, wollen wir uns offen halten. Wir denken dabei insbesondere an Prozessalgebren.
- Ebenso möchten wir uns nicht mehr auf konkrete Definitionen des (angereicherten) Erreichbarkeitsgraphen oder der Simulationsrelation beschränken. Es soll möglich sein, Ideen aus theoretischen Arbeiten auch in dieser Hinsicht schnell in FastAsy umzusetzen.
- Damit zusammenhängend möchten wir die angestrebte Modularität auch nutzen, um konzeptionelle Optimierungen an den verwendeten Verfahren mit möglichst nur lokalen Änderungen zu implementieren.
- Betrachtet man den Code der alten FastAsy-Versionen, so fällt auf, dass an verschiedenen Stellen immer wieder der iterative Rahmen einer Induktionsvorschrift implementiert wird. (Vereinfacht gesagt werden an mehreren Stellen Varianten von Tiefen-/Breitensuche implementiert, die sich im wesentlichen nur in der Bestimmung der Nachfolgeknoten unterscheiden.) Diese Beobachtung würden wir gerne ausnutzen, indem wir das Design derart verändern, dass Erweiterungen nur mehr die Induktionsvorschrift implementieren müssen und die Iteration durch das Framework durchgeführt wird.

Unsere Bestrebung ist es, FastAsy zu einem universell erweiterbaren Werkzeug im Bereich simulationsbasierter Effizienzvergleiche zu machen. Unter dem Anwender stellen wir uns jemanden vor, der mit der Theorie vertraut, wahrscheinlich sogar mit deren Weiterentwicklung beschäftigt ist, und FastAsy einsetzen möchte, um anhand von Beispielen neue Ideen schnell und bequem im Hinblick auf Anwendbarkeit und intuitiv einsichtige Ergebnisse testen zu können. Wenn einige Softwareprozesse vorschlagen, sich am Beginn der Entwicklung eines Softwaresystems eine übergreifende *Metapher* (es sei am Rande die Bemerkung gestattet, dass „Allegorie“ wahrscheinlich der bessere Begriff gewesen wäre) zurechtzulegen und sich im weiteren von dieser leiten zu lassen (siehe z.B. [BK99]), so wäre dies im Falle von FastAsy wohl das Bild eines Experimentierkastens: In diesem wollen wir Bauteile auswechseln und auf verschiedene Weisen miteinander verdrahten können, und für Versuchsaufbauten, die sich bewähren, möchten wir diese hinter einer Frontplatte mit den wichtigsten Schaltern und Reglern darauf verbergen.

Mit der Schaffung klarer Strukturen beim Systemdesign haben wir FastAsy dennoch klar von einem „ewigen Prototyp“ abgegrenzt. Eindeutig in der Rolle der Prototypen sehen wir hingegen die beiden früheren Codebases von FastAsy (siehe unten), die sich dabei aber auch in mehrfacher Hinsicht bewährt haben: Neben dem Wert als Proof-of-Concept hat uns vor allem die Zwischenversion zu einem vertieften Verständnis der Techniken geführt, mit deren Hilfe sich die gewünschten Freiheitsgrade und die angestrebte Modularität in ein zunächst doch verhältnismäßig stark verzahntes Softwaresystem wie FastAsy einführen lassen. Die hier vorgestellte jüngste Codebase von FastAsy ist im Gegensatz dazu als Produktivversion konzipiert: Das Design ist langfristig auf Wartbarkeit und Erweiterbarkeit ausgelegt, der Code ist nach einer speziellen Konvention kommentiert, welche die maschinelle Extraktion der Dokumentation in verschiedenen Formaten ermöglicht, und es wurden moderne Techniken der Qualitätssicherung wie Regressionstests und Überwachung von Zusicherungen zur Laufzeit verwendet.

Im Gegensatz zu der erwähnten Zwischenversion sehen wir nun wieder ein CLI (Command Line Interface) als wichtigste Benutzeroberfläche an. Bewogen dazu hat uns die Erfahrung, dass man oft parametrisierte Berechnungen durchführen will und es in solchen Fällen überaus praktisch ist, solche Berechnungen in einem Skript zu formulieren und dann etwa über Nacht als Batchlauf durchführen zu können. Mit *CliArg* (siehe 9.7.3) wurde die Möglichkeit geschaffen, Kommandozeilen flexibel und mit wenig Aufwand zu verarbeiten. Sollte jedoch später eine GUI wünschenswert erscheinen, so kann auch eine solche dem System ohne großen Aufwand und mit nur lokalen Änderungen vorgeschaltet werden.

# Kapitel 5 Historie

Wie bereits angedeutet, liegt der im Rahmen dieser Arbeit vorgestellten Version von FastAsy die dritte, komplett neu entwickelte, Codebase zu Grunde. Die Entwicklung des ersten Prototypen von FastAsy begann jedoch schon 1996. Diese Urversion war in der behandelten Semantik strikt festgelegt auf Netze mit Lesekanten und impliziten [0,1]-Beschriftungen der Transitionen, wie sie in [Vog96], [Vog97] vorgestellt und etwa in [BV98] zusammen mit FastAsy verwendet wurden.

Die Repräsentation der mathematischen Objekte im Code wurde dabei im wesentlichen von Implementierungsgesichtspunkten geleitet; es wurde nicht versucht, den Code „sprechend“ zu machen in dem Sinne, dass Objekte der Problemdomäne (d.h. der mathematischen Formulierungen) zu Klassen abstrahiert wurden. Dafür gelang die Entwicklung dieses Prototyps aber in äußerst kurzer Zeit, und es konnten die Effizienzresultate von [Vog96] nun auch maschinell nachvollzogen werden. Bis zum Frühjahr 2000 wurden weitere Optimierungen integriert, außerdem erhielt diese FastAsy-Version die Fähigkeit, parametrische Modifikationen an Netzgraphen vorzunehmen, da dies mit dem als Frontend zum Bearbeiten von Netzen verwendeten PEP (siehe [PEP]) nur mühselig möglich war. Die Ergebnisse von [BV98] wurden allesamt noch mit solchen Versionen auf Basis der ersten Implementierung erzielt. Mit der in [Bih98] vorgestellten Klasse der PTT-Netze (*ST-Netze* in der damaligen Nomenklatur) wurde jedoch klar, dass wir daran interessiert waren, Varianten der Netzklassen und Effizienzsemantiken ebenfalls computergestützt zu untersuchen, und das Design der alten Codebase geriet damit an seine Grenzen.

Deswegen wurde ab 1998 die Entwicklung einer neuen Codebase von FastAsy vorangetrieben, die einerseits das Abstraktionsniveau der Problemlösung solcherart anheben sollte, dass die Behandlung verschiedener Netzklassen möglich wäre und die andererseits statt implementierungsgerichteter interner Repräsentationen zunächst Schlüsselabstraktionen einführen und deren Implementierung sauber kapseln sollte. Zudem war die Benutzeroberfläche im Gegensatz zum Vorgänger grafisch konzipiert. In [Bih00] haben wir detailliert über Anforderungen und Design dieser Zwischenversion berichtet. Bereits in der Endphase der Entwicklung wurden jedoch Probleme mit dem neuen Design offenbar: Zum ersten bezogen sich die verschiedenen Programmteile (eine Schichtenteilung in Bezug auf verschiedene Verarbeitungsschritte war eben gerade noch nicht vorhanden) zwar nun über Schlüsselabstraktionen aufeinander, deren Kopplungsgrad aber immer noch sehr hoch war, vor allem über die Grenzen dessen hinweg, was wir beim Design der jüngsten Version dann als Schichtenstruktur ausgemacht haben. Es seien an dieser Stelle kurz [FB00] erwähnt, dessen Ausführungen über Architekturpatterns entscheidend zur Förderung des Problemverständnisses in dieser Richtung beigetragen haben, und [JL96], wo auf einer technischeren Ebene zahlreiche Methoden zur Kontrolle des Kopplungsgrads in komplexen Softwaresystemen vermittelt werden. Die zweite Problematik des Designs wurde durch den Wunsch nach der Verwendung impliziter Verweigerungsmengen (siehe Teil I, Kapitel 2) offenbar: Es schien auf einmal nicht mehr ausreichend, Verarbeitungsschritte auf verschiedenen Ausprägungen von Schlüsselabstraktionen arbeiten zu lassen, wobei sich letztere dann in ihren Dateninhalten und ihrem Verhalten unterscheiden durften. Stattdessen wurde es zunehmend nötig, von den konkreten Algorithmen selbst zu abstrahieren und so zu

durch Elemente ihres Verhaltens parametrisierten Algorithmen zu kommen, zu *generischen* Algorithmen. Als richtungsweisende Literatur, wenn auch mit stark exemplarischem Charakter, sei hier [SLL02] erwähnt.

Möchte man nun den Wert jener Zwischenversion von FastAsy, deren Entwicklung schließlich doch ein gehöriges Maß an Zeit erfordert hat, kritisch würdigen, so kommt einem unweigerlich das legendäre Zitat von Frederick P. Brooks (z.B. in der Sammlung seiner Essays über Software-Engineering [FB03]) in den Sinn: „Plan to throw one away. You will anyway.“. In der Tat haben die Erfahrungen aus der Zwischenversion wesentlich mehr zum Design des jüngsten FastAsy beigetragen als die aus der Urversion. Neben dem, was bereits gesagt wurde, wurde nicht zuletzt auch die zentrale Idee, die induktive Definition eines Transitionssystems als Schlüsselabstraktion einzuführen, in ihrer Rohform in dieser Zwischenversion geboren.

Seit 2001 wurde dann die vorliegende Version von FastAsy konzipiert und entwickelt.

# Kapitel 6 Rahmen / Plattform / Tools

## *6.1 Sprache und Paradigma*

Wie schon seine Vorgängerversionen wurde auch diese Version von FastAsy in C++ implementiert, spezifischer in der ANSI/ISO-Standardisierung von 1998 (ISO/IEC 14882:1998). Bedauerlicherweise leistet sich die überwiegende Mehrheit der auf dem Markt befindlichen C++-Compiler selbst zum heutigen Zeitpunkt noch einzelne Schwächen in der Umsetzung des Standards. Vor allen Dingen betroffen ist davon die Umsetzung bestimmter Feinheiten der Template-Programmierung, beispielsweise partielle Template-Spezialisierung. [VJ03] gibt neben einer umfassenden Einführung in die Template-Programmierung auch einen Einblick in verbreitete Schwächen vorhandener Compiler. Dem Autor ist leider kein Survey bekannt, in dem mehrere Compiler systematisch gegen den ANSI/ISO-Standard getestet werden. Eine Art Benchmark für Kompatibilität können allerdings die Regressionstests der Boost-Bibliothek auf [BCPL] sein.

Entwickelt und getestet wurde FastAsy unter Borland C++ Builder 5, wobei keine compilerspezifischen Spracherweiterungen verwendet wurden. (Die Entwicklung einer dann natürlich plattformspezifischen GUI wäre jedoch mit Hilfe der in die IDE integrierten RAD-Tools schnell möglich, und die Zuhilfenahme der Spracherweiterungen geschähe rein lokal im GUI-Teil.)

Wie jedes durchdachte Softwaresystem ist auch FastAsy bemüht, große Teile des Codes allgemein zu halten und so mehrfach verwenden zu können. Dabei bedienen wir uns natürlich an vielen Stellen der in der OO üblichen Techniken, nämlich Wiederverwendung durch Vererbung und durch Aggregation. Vererbung sollte möglichst nur dann zum Einsatz kommen, wenn es zumindest konzeptionell sinnvoll erscheint, die verschiedenen Nachfahren polymorph zu behandeln. Reine Implementierungsvererbung hingegen, die sich nur einen Teil der technischen Vorgänge bei der Vererbung zunutze macht und die Substitutionsbeziehungen zwischen Basisklasse und Nachfolgern ignoriert, wollen wir vermeiden. Sie lässt sich immer durch Aggregation ersetzen, also ein Konstrukt, bei dem die vormals erbende Klasse die andere Klasse direkt oder über eine Referenz als Attribut enthält und deren Dienste über Delegation in Anspruch nimmt.

In C++ steht jedoch mit den Templates noch ein weiterer, von Vererbung völlig unabhängiger Mechanismus zur Verfügung, um Code wieder verwendbar zu machen. C++-Templates sind zunächst eine Umsetzung des Prinzips der *generischen Programmierung* (nach [BS98] die „Programmierung mit Typen als Parametern“). Das historische Verständnis dieses Paradigmas ist derart, dass man Verfahren ohne Angabe konkreter Typinformation für einige oder alle Operanden beschreibt. Dabei bedient man sich jedoch natürlich trotzdem elementarer Operationen auf den unbekannten Typen, und dies legt wiederum implizit fest, für welche konkreten Typen das generische Verfahren instanziert werden kann, nämlich für genau solche, die über die verwendeten Operationen verfügen. Im Gegensatz zur *Polymorphie durch Untertypenbildung* müssen also die später verwendeten Typen in keiner Beziehung zueinander stehen. Tatsächlich ist die historische und aus moderner Sicht reduzierte Auffassung von generischer Programmierung immer noch verbreitet und wurde zum Beispiel in Java 1.5. mit den *Generics* tatsächlich auch nur in dieser Form eingeführt.

In C++ hingegen trat der etwas paradoxe Effekt auf, dass das volle Potential dieses Sprachmittels erst nach und nach offenbar wurde. Man hätte wohl mit einigem Recht zunächst auch von einem kreativen Missbrauch des Sprachmittels sprechen können. Ausgangspunkt dieses modernen Verständnisses generischer Programmierung ist es, die generischen Typen nicht mehr nur als Stellvertreter elementarer Werte zu sehen, welche Operationen wie etwa Zuweisung, Vergleich oder Addition unterstützen, sondern auch und gerade komplexe Typen zu verwenden, welche auch oft selbst wieder Freiheitsgrade in Form von Typparametern besitzen. Templates machen es nach diesem Verständnis also nicht mehr bloß möglich, Platzhalter für Typenspezifikationen zu verwenden, sondern stellen neben Vererbung und Aggregation einen weiteren Mechanismus dar, um komplexe Typen aus anderen zusammenzusetzen.

Zur Nomenklatur sei angemerkt, dass in der auf praktische Aspekte der Programmierung ausgerichteten Literatur der Begriff *Polymorphie* manchmal etwas problematisch ist, da er nur im eingeschränkten Sinne der *Polymorphie durch Untertypenbildung* verwendet wird. Schon innerhalb der objektorientierten Programmierung gehört aber zumindest noch die *Polymorhpie durch Überladen* dazu, und aus der generischen Programmierung erhalten wir nun noch die *Polymorphie durch parametrische Typen*. Um aber nicht jedes Mal diese doch etwas wortreichen Bezeichnungen verwenden zu müssen, verwenden wir die etwas weniger exakten Begriffe *Objektpolymorphie* oder, wo dies nicht missverständlich ist, tatsächlich auch nur *Polymorphie* für *Polymorphie durch Untertypenbildung* und *Typpolymorphie* für *Polymorphie durch parametrische Typen*. *Polymorhpie durch Überladen* spielt indes in unseren Betrachtungen keine Rolle.

Vor- und Nachteile der Template-Programmierung wurden schon an vielen Stellen erschöpfend erörtert ([AA01], [SLL02], [VJ03]), so dass hier nur kurz in kompakter Form darauf eingegangen werden soll, was die für FastAsy spezifischen Gründe waren, an vielen Stellen Templates einzusetzen:

- Da Templates lediglich die tatsächliche Signatur eines Typs betrachten, fordern sie keine gemeinsame Vererbungshierarchie; insbesondere lassen sich Klassen und primitive Typen vereinheitlichen. Möchte man dies durch Objektpolymorphie nachbilden, müssten primitive Typen in Klassen verpackt werden, die sämtliche Operationen nachbilden und an den als Attribut enthaltenen primitiven Typ delegieren. Dieses sogenannte *boxing* bedingt natürlich gerade bei elementaren Operationen einen hohen *Overhead* zur Laufzeit.
  In FastAsy ist es aber wichtig, beispielsweise mit demselben Code verschiedene Automaten zu behandeln, deren Zustände durch komplexe (etwa RTState) bzw. primitive (etwa TIntState, ein *typedef* für *unsigned int*) Typen repräsentiert werden könne.
- Generell wird Typpolymorphie zur Übersetzungszeit aufgelöst, Objektpolymorphie hingegen zur Laufzeit. Daraus resultiert erstens ein gewisser Geschwindigkeitsvorteil für Lösungen, die Typpolymorhpie verwenden. (Objektpolymorphie wird von C++-Compilern durch Zuhilfenahme von Indirektionstabellen trotzdem effizient realisiert. Ob die Performance merklich schlechter ist, hängt deshalb natürlich in erster Linie davon ab, ob aufgerufene Methoden komplex genug sind, dass der erhöhte Overhead des Aufrufs größenordnungsmäßig keine Rolle mehr spielt.) Zum zweiten hat

Typpolymorphie aber auch die Tendenz, den robusteren Code zu produzieren, da Typfehler noch zur Übersetzungszeit aufgedeckt werden. Bei der Objektpolymorphie hingegen steht man bei fehlerhaftem Code womöglich zur Laufzeit Situationen gegenüber, in denen ein Objekt nicht als ein erwarteter Typ angesprochen werden kann (Fehler beim *downcast*).

- Mit der BGL (Boost Graph Library, siehe [BCPL]) steht eine leistungsfähige Bibliothek zur Modellierung von Graphen zur Verfügung, die sich ausgiebig generischer Programmierung bedient. Auch um bei der Verwendung der BGL kein Impedanzproblem durch den Übergang zwischen zwei Paradigmen zu schaffen, ist es wünschenswert, die Vorteile der Generizität der BGL in umgegebendem Code „durchzureichen", d.h. bestimmte Typen parametrisch zu belassen.

Einige praktische Aspekte der generischen Programmierung in C++ sind jedoch durchaus auch mit Problemen behaftet. Wir sind weit davon entfernt, in Templates ein Allheilmittel zu sehen. Nachteile, welche es zu erwägen gilt, sind:

- Die Fehlerbehebung bei Übersetzungsfehlern gestaltet sich oft schwieriger, da erst nach dem Präprozessorlauf wirklich derjenige Code entsteht, der den Fehler verursacht. Viele Compiler melden aber lediglich die Stelle der Instanzierung eines Templates, innerhalb deren der Fehler auftritt. Bei kaskadierten Templates kann mitunter eine langwierige Suche durch die Implementierungen aller an der Instanzierung beteiligten Klassen die einzige Möglichkeit sein, dem Fehler auf die Spur zu kommen.
- Aus demselben Grund ist auch das interaktive Debuggen zur Laufzweit problematisch, denn die zum jeweiligen Stand des *program counter* korrespondierenden Stellen sind im Quellcode eben einfach nicht explizit vorhanden.
- Da C++-Code, welcher Templates verwendet, sehr schwierig (und im Prinzip nicht lokal) zu parsen ist, stehen viele Möglichkeiten, die fortschrittliche Entwicklungsumgebungen bieten könnten, nicht zur Verfügung. Dazu gehören *Projekt-Outlining*, *Code-Browsing* und *Refactoring-Unterstützung*. (Zu letzterem ist freilich anzumerken, dass sich für *C++* auch ohne Templates der Tool-Support für Refactoring schon sehr in Grenzen hält, wenn man es etwa mit den Möglichkeiten vergleicht, die *Eclipse* in diesem Zusammenhang für Java bietet.)

All dies resultiert in einem doch stellenweise merklich erhöhten Aufwand bei der Entwicklung von Code mit Hilfe von Templates. Wir haben uns aber wegen oben erwähnter Vorteile trotzdem entschlossen, große Teile von FastAsy mit dieser Technik umzusetzen.

## *6.2 Bibliotheken*

### 6.2.1 STL

Die Standard Template Library ist, obwohl ursprünglich unabhängig davon entwickelt, mittlerweile Bestandteil der C++-Standardbibliotheken und durch ISO/IEC 14882 standardisiert. Die STL befasst sich in erster Linie mit generischen Containern nebst zugehöriger Iteratoren und elementarer Algorithmen auf den Containern. Außerdem sind die Konzepte der *iostreams*, welche die aus C bekannten Ein-/Ausgabemechanismen ablösten, mittlerweile eng mit der STL verwoben. Auch die *string*-Klasse wird in der STL als Container

aufgefasst, ihre Generizität kommt dann ins Spiel, wenn statt herkömmlichen *chars* sogenannte *wide chars* verwendet werden sollen.

Es existieren verschiedene Implementierungen der STL, die teilweise frei und teilweise kommerziell sind. Viele davon sind als reine Header-Dateien implementiert, so dass ein Austausch problemlos möglich ist. Durch den ISO-Standard sind C++-Compiler im Prinzip verpflichtet, zumindest eine Umsetzung mitzuliefern. Im Falle des verwendeten Borland-Compilers war dies die kommerzielle Roguewave-STL. Stellt sich eine in einem Compiler enthaltene STL als mangelhaft heraus, kann die quellcodeoffene und auf Portabilität ausgelegte STLPort (siehe [STLP]) meist als Ersatz dienen.

FastAsy verwendet an vielen Stellen direkt die Container der STL, weiter stützen sich aber auch die im folgenden erwähnten Bibliotheken von Boost stark auf die STL ab.

### 6.2.2 Boost

Boost ist eine äußerst umfangreiche Sammlung von freien, plattformunabhängigen Bibliotheken für C++. Viele der Teilbibliotheken sind dabei wie die STL in Form reiner Headerdateien umgesetzt, es existieren aber auch solche, für die ein Build-Lauf notwendig ist. Neue Bibliotheken gelangen nur über einen formalen Review-Prozess zur Aufnahme, der hohe Anforderungen an Codequalität, Relevanz und auch an die Dokumentation stellt. In der Tat sind viele der Teilbibliotheken von Boost dem Standardisierungskomitee zur Aufnahme in zukünftige C++-Standards vorgeschlagen worden. Bestimmte Teile von Boost waren wegweisend für das in 6.1 umrissene erweiterte Verständnis der Template-Programmierung in C++. Eine gute Einführung in die Verwendung der verschiedenen Teile von Boost gibt [BK06], allerdings in die Version 1.32.0. (zu den Versionen siehe auch weiter unten). Die Unterschiede der relevanten Teile zur verwendeten Version halten sich jedoch sehr in Grenzen.

Wir verwenden zur Zeit in FastAsy folgende Teile von Boost:

- **BGL**:
  Die Boost Graph Library ging aus der an der University of Notre Dame entwickelten Generic Graph Component Library (siehe [SLL99]) hervor und stellt einen der komplexesten Teile innerhalb von Boost dar. Sie stellt Templates zur Verfügung, die es in Anlehnung an die STL-Container erlauben, beliebige Daten typsicher in Graphen abzuspeichern und enthält weiter eine Sammlung generischer Algorithmen auf diesen Graphen. Bezüglich der Eigenschaften und der internen Repräsentation hat der Benutzer dabei weitestgehende Freiheiten.
  In FastAsy benutzen wir momentan *adjacency_list*, um die Graphen in den Automaten und Simulationen zu verwalten. Weitergehende Funktionen benutzt FastAsy momentan nicht, da sie mit dem Architekturkonzept der Bausteine (siehe 9.2.1.5) nicht sonderlich verträglich sind. Wir werden jedoch in 11.7 kurz darauf eingehen, wie man die Benutzung von BGL-Algorithmen trotzdem in die Architektur einbetten könnte.
- **dynamic_bitset**:
  FastAsy benötigt an mehreren Stellen die Möglichkeit, effizient auf Teilmengen von zur Laufzeit bekannten Mengen zu operieren. Mit *bitset<N>* enthält die STL zwar einen Container, welcher dafür im Prinzip in Frage kommt, jedoch muss die Größe des Bitvektors dazu zur Übersetzungszeit bekannt sein, was *bitset* in diesem

Zusammenhang zur Nutzlosigkeit verdammt. Weiter existiert in der STL eine Spezialisierung des *vector*-Templates für den Typ *bool*, welche allerdings im Gegensatz zu *bitset* keine speziellen Mengenoperationen zur Verfügung stellt und somit in unserem Kontext nur umständlich und ineffizient zu handhaben wäre.
Das Boost-Template *dynamic_bitset* schließt genau diese Lücke: Es verhält sich wie *bitset*, die Zuweisung der Größe muss jedoch erst zur Laufzeit erfolgen.

- **tuple**:
  Die gleichnamigen Templates sind eine Verallgemeinerung des aus der STL bekannten *pair*. Es gestattet uns, anonyme n-Tupel zu benutzen, anstatt mit Hilfe des Schlüsselworts *struct* Tupel mit benannten Elementen erstellen zu müssen. Weiter sorgen die Templates dafür, dass bestimmte auf den Elementen vorhandene Operationen wie Zuweisung, Vergleich oder Stream-I/O auf die Ebene des Tupels gehievt werden. Schließlich steht mit dem Template *tie* noch eine elegante Möglichkeit zur Verfügung, Werte der Elemente eines Tupels wieder an einzelne Variablen zuzuweisen, in dem nicht-konstante Referenzen zu einem lvalue-Tupel gebunden werden.
  Beispiel:
  ```
  int a;
  double b;
  tie(a,b) = tuple<int,double>(2,3.14);
  ```
- **scoped_ptr**:
  *scoped_ptr* implementiert eine spezielle Variante eines sogenannten *Smart Pointers*, d.h. eines Konstrukts, das weitgehend wie ein nativer C++-Zeiger verwendet werden kann, aber implizit weitere Aufgaben erledigt. (Um einen solchen handelt es sich z.B. auch bei dem in 9.7.5 beschriebenen *Reference_ptr*, wobei dessen Aufgaben völlig verschieden ausfallen.) Im Falle von *scoped_ptr* bestehen diese Aufgaben darin, mit dem Verlassen des Gültigkeitsbereichs der Zeigervariable auch den Speicherbereich, auf den gezeigt wird, freizugeben. Neben der offensichtlichen Vereinfachung der Handhabung dynamisch allozierten Speichers und der damit verbundenen Eliminierung potentieller Fehlerquellen im Code zieht man aus der Verwendung von *scoped*_ptr noch einen tiefer liegenden Vorteil: Im Falle einer zwischen Allozierung und Freigeben auftretenden *Exception* wird ohne weitere Sonderbehandlung der Speicherbereich korrekt freigegeben und eventuelle Destruktorenaufrufe finden statt.
  Es sei kurz auf den zentralen Unterschied zu dem in der STL enthaltenen *auto_ptr* eingegangen: Während eine Zuweisung eines *auto_ptr* an einen anderen den Besitz (d.h. die Verantwortung für die Freigabe und Zerstörung) an einem referenzierten Objekt auf das Ziel der Zuweisung überträgt, verbietet *scoped_ptr* solche Zuweisungen gänzlich. Es gibt also sehr spezielle Situationen (insbesondere die Rückgabe dynamisch allozierter Objekte aus Methoden heraus), in denen *auto_ptr* genau die intendierte Semantik trifft, für die meisten Kontexte ist aber *scoped_ptr* zu bevorzugen, weil es stärkere Zusicherungen macht.
- **optional**:
  Eine in vielen Programmiersprachen gängige, aber höchst unschöne Technik, um für den Wert einer Variable anzuzeigen, dass er ungültig bzw. undefiniert ist, besteht

darin, für diesen Fall einen speziellen Wert vorzusehen, im Falle von Zahlen etwa -1 oder bei Zeichenketten die leere Zeichenkette. Die Probleme, die entstehen, wenn andere Programmierer später eine solche Konvention übersehen, sind offensichtlich. Etwas weniger dramatisch ist die Situation, wenn stattdessen ein Zeiger verwendet wird, denn immerhin garantiert C++, dass ein gültiger Zeiger nie den speziellen Wert NULL haben kann. Trotzdem muss einem Programmierer explizit bekannt sein, an welchen Stellen vorzusehen ist, dass ein Zeiger NULL sein kann, denn die Verwendung von Zeigern kann natürlich auch anders motiviert sein.
Die Verwendung von *optional*<*T*> ermöglicht es nun, im Quelltext ausdrücklich festzulegen, dass ein Wert vom Typ T vorhanden sein oder fehlen kann. Der Versuch, auf einen fehlenden Wert zuzugreifen, führt durch die Verletzung einer (mit BOOST_ASSERT realisierten) Zusicherung automatisch zu einem Laufzeitfehler.

- **lexical_cast**:
  In C++ existieren (oft noch aus den von C übernommenen Bibliotheken) diverse Möglichkeiten, verschiedene Datentypen ineinander zu konvertieren, die jedoch unübersichtlich und umständlich in der Handhabung sind. Oftmals trifft man in Applikationscode deshalb auch auf ad-hoc-Code zur Konvertierung. Der *lexical_cast*<*T*>-Operator bringt all jene Konvertierungen nach T nun unter einen Hut, die stattfinden können, indem der ursprüngliche Wert unter Zuhilfenahme des Ausgabeoperators << in einen Puffer geschrieben und durch die Implementierung von >> für T wieder eingelesen wird. Dies ist insbesondere für die eingebauten Datentypen und die *string*-Klasse der STL zutreffend.
- **Unit Test Framework**:
  Das Unit Test Framework stellt einen einheitlichen Rahmen zur Verfügung, um eine Menge von Testfällen für das *Unit Testing* zu definieren und kümmert sich im Anschluss an die kontrollierte Ausführung der Testfälle auch um die Protokollierung der Ergebnisse. Dabei werden die Testfälle nicht in die Applikation kompiliert, sondern die Quelltextdateien von FastAsy werden zusammen mit den Testfällen zu einer eigenen Anwendung übersetzt, welche einzig dem Zweck dient, diese Regressionstests durchzuführen. In 9.1.2 werden wir genauer auf das *Unit Testing* in FastAsy eingehen.
- **Program Execution Monitor**:
  Der Program Execution Monitor stellt einen Wrapper um den eigentlichen Einsprungpunkt (d.h. die freie Funktion *main*) dar, mit dem Ziel, Fehlerzustände der Applikation abzufangen und in von Compiler und Plattform unabhängiger Weise in einem einheitlichen Format dem Benutzer zu melden. Solche Fehlerzustände sind das Auftreten unbehandelter *Exceptions*, ein Rückgaberwert ungleich 0 aus der Hauptfunktion oder die Verletzung einer Zusicherung.
  Ein entscheidender Vorteil einer vereinheitlichten Rückmeldung ist es, innerhalb von Skripten einfach und plattformunabhängig auf auftretende Fehler reagieren zu können.

Zum gegenwärtigen Zeitpunkt verwendet FastAsy die Version 1.31.0 von Boost, obwohl bereits jetzt eine Nachfolgeversion existiert. Wie wir nämlich bei der Umstellung von 1.30 auf 1.31 erfahren haben, kann eine Umstellung der Version durchaus einen gewissen Aufwand

bedeuten. Dieser ist allerdings nur zu einem verschwindend geringen Teil darin begründet, dass sich einzelne Schnittstellen ändern. Vielmehr sind es meist die Wechselwirkungen mit der jeweils verwendeten STL und den Besonderheiten des jeweiligen Compilers, die in der Praxis einige Änderungen nötig machen können. Aus diesem Grund haben wir die Verwendung von Boost zunächst bei Versionsstand 1.31. eingefroren.

Da FastAsy sich prinzipiell nur der als Header-Dateien implementierten Bibliotheken von Boost bedient, genügt es, die in Boost enthaltenen Dateien zu entpacken und in den Include-Pfad des Compilers das Hauptverzeichnis von Boost aufzunehmen. Man beachte, dass die Header-Dateien im Unterverzeichnis *boost* residieren und die Angabe dieses Unterverzeichnisses aber in allen *include*-Anweisungen mitgeführt wird.

### 6.2.3 BE++

Bei der Entwicklung von FastAsy waren einige Probleme in der Regel technischer Natur zu lösen, bei denen sich die Abstraktion der Lösungen zu einer allgemein verwendbaren Bibliothek anbot. Auch hier handelt es sich um Implementierungen, die als reine Header-Dateien vorliegen und so keinen statischen Linkvorgang nötig machen.

Die entstandene Utility-Bibliothek ist unter dem Namen BE++ frei verfügbar (siehe [BEPP]). Sie wurde in der Zwischenzeit auch schon in kommerziellen Softwareprojekten mit Erfolg eingesetzt, und Einflüsse aus der Verwendung in solchen Projekten wurden wiederum genutzt, um BE++ zu verbessern.

Da die Motivation für BE++ dennoch dem FastAsy-Projekt entspringt, werden die entsprechenden Teile in 9.7 als Bestandteile von FastAsy dokumentiert. Wir legen im Rahmen dieser Arbeit Wert auf die Feststellung, dass der Autor alleiniger Entwickler der BE++ ist.

## *6.3 Tools*

### 6.3.1 Doxygen

Doxygen ist ein frei verfügbares (siehe [DOX]), C++-spezifisches System, mit dessen Hilfe sich aus Quelltext maschinell Dokumentation extrahieren und in verschiedener Form (HTML, RTF, LaTeX, CHM) darstellen lässt. Die Extraktion stützt sich dabei einerseits auf den C++-Code selbst, wird aber andererseits ergänzt durch Angaben in Kommentarzeilen, die einer speziellen Syntax folgen. In dieser Funktionsweise ähnelt es dem aus der Java-Welt bekannten JavaDoc-Standard und den assoziierten Tools. Obwohl Doxygen auch eine Ausgabe in ein generisches XML-Format vorsieht, kann es mit der Flexibilität, die angepasste Java-Doclets zur Verfügung stellen, natürlich nicht konkurrieren. Gegenüber der Dokumentation, die Suns Standard-Doclet erzeugt, besitzt Doxygen hingegen einige zusätzliche Leistungsmerkmale. So besitzt Doxygen z.B. die Möglichkeit, das Graphviz/Dot-Paket (siehe 6.3.3) zu verwenden, um einerseits einfache Klassen- und Kollaborations-diagramme zu erzeugen und andererseits aber auch explizit in den Kommentaren definierte Graphen in die Dokumentation einzufügen. Letzteres hat sich insbesondere bei der Dokumentation von Testfällen in FastAsy bewährt. Doxygen ist auch in der Lage, in die HTML-Dokumentation den Quelltext mit farblichen Hervorhebungen und Hyperlinks zwischen den benannten Entitäten versehen aufzunehmen. Dies vereinfacht den Zugang zum Quelltext wesentlich und ist im Gegensatz zu IDEs, die ebenfalls Browsing-Unterstützung für

den Quelltext bieten, erstens hochgradig portabel und zweitens schneller, da die Quelltextanalyse ja nur einmal stattfindet.

### 6.3.2 Subversion

Subversion (siehe [SVN], [PCF04]) ist ein frei verfügbares Versionskontrollsystem, d.h. es speichert, genau wie das verbreitete CVS, für jede Quelltextdatei nicht nur den aktuellen Stand, sondern auch die gesamte Historie. Dabei können Snapshots des gesamten Projekts (sogenannte *Tags*) markiert und Seitenäste der Entwicklung (sogenannte *Branches*) verwaltet werden. Seitenäste kommen oft dann zum Einsatz, wenn während einer größeren Entwicklungsphase der aktuelle Quelltext schon Merkmale einer zukünftigen Version enthält aber zeitgleich ausgehend von einer älteren Version ein gepatchtes Release erstellt werden muss.

Die wichtigsten Gründe, die uns dazu bewogen haben, Subversion einzusetzen, sind:

- Subversion unterstützt symbolische Kopien und Umbenennungen statt wie CVS Dateien einzig durch ihren Pfad zu identifizieren.
- Subversion hält bei jedem Checkout eine lokale Kopie der aktuellen Revision vor und kann so die meisten Aktionen (natürlich bis auf das Zurückschreiben in das Repository) lokal durchführen.
- Das Repository kann lokal oder entfernt gehalten werden, in ersterem Fall ist nicht einmal ein Serverprozess nötig, da der Client die Repository-Datenbank *embedded* ansprechen kann.
- Zurückschreiben in das Repository (das sog. *Commit*) ist atomar, d.h. schlägt das Schreiben für einzelne Dateien fehl (etwa weil bereits eine konkurrierende Änderung geschrieben wurde), wird das gesamte *Commit* verworfen.
- Mit TortoiseSVN steht eine schlanke, aber vollständige GUI zur Verfügung, die sich über Kontextmenüs nahtlos in den Windows Fileexplorer integriert. Wird mit Eclipse/CDT gearbeitet, so steht dort mit Subclipse ebenso eine leistungsfähige Integration zur Verfügung.
- Subversion stellt serverseitig sogenannte *Hooks* zur Verfügung. Dies sind Einsprungpunkte, die bei Schlüsselaktionen aufgerufen werden. So lässt sich im Zusammenspiel mit Doxygen (siehe 6.3.1) etwa eine Funktionalität realisieren, bei der nach jedem *Commit* die HTML-Dokumentation des Projekts invalidiert und ggf. neu erstellt wird.

### 6.3.3 Graphviz/Dot

Graphviz/Dot (siehe [GVZ]) wurde ursprünglich von AT&T entwickelt und ist ein frei verfügbares System, um verschiedene Arten von Graphen automatisch zu layouten. Wir verwenden Graphviz/Dot wie bereits in 6.3.1 erwähnt, um Graphen innerhalb der Dokumentation darstellen zu lassen. Weiter haben wir FastAsy aber mit der Möglichkeit versehen, Zwischenergebnisse in der Form endlicher Automaten in einem von Graphviz/Dot lesbaren Format auszugeben, so dass wir diese Ergebnisse anschließend visualisieren können.

# Kapitel 7 Funktionsweise der Algorithmen

In diesem Abschnitt wollen wir darauf eingehen, wie die zuletzt etwa in [BV04] geschilderte schneller-als-Relation für zwei gegebene Quellsysteme, d.h. in unserem Fall Petrinetze, entschieden werden kann. Wir schildern das Vorgehen ausführlich genug, um die Grundlagen zu haben, uns bei der Beschreibung der Umsetzung in Kapitel 9 darauf beziehen zu können. Im Hinblick darauf werden wir die verschiedenen Bearbeitungsschritte bereits an dieser Stelle geeignet (im Sinne von 7.1) formulieren.

Die geschilderte Vorgehensweise entspricht in vielen Grundprinzipien derjenigen, die in den früheren FastAsy-Versionen „*hardwired*“ war, und die in [BV04] oder in höherem Detailgrad in [Bih98] beschrieben ist, besitzt aber an vielen Stellen einen höheren Abstraktionsgrad. In der vorliegenden FastAsy-Version bewirken die einzelnen Komponenten in einer bestimmten Verschaltung genau das Beschriebene, es wird jedoch später offenbar werden, dass diese Komponenten auch durchaus anders und zu anderen Zwecken kombiniert werden könnten.

## *7.1 Vorbemerkung: Induktive und lineare Systeme*

Wie wir noch sehen werden, bringt es entscheidende Vorteile für das Design mit sich, wenn wir, im Gegensatz zu früheren FastAsy-Versionen, Transformationen auf endlichen Automaten nicht als Vorschrift auf den Ecken und Kanten der Transitionsgraphen auffassen, sondern gewissermaßen als Transformation der Zustandsübergangsfunktion. Diese geänderte Kollaborationsmechanik stellt den entscheidendsten einzelnen Fortschritt gegenüber alten FastAsy-Versionen und wahrscheinlich den interessantesten Beitrag dieses zweiten Teils der vorliegenden Arbeit dar. In den Mittelpunkt des Interesses rücken werden wir diesen (und einen dazu eng verwandten Mechanismus) in Kapitel 9, wo wir in 9.2.2 die generelle Kollaborationsstrategie detailliert besprechen und dann in 9.4 die Umsetzungen dieser Strategie kennen lernen werden. Für das Verständnis des folgenden müssen wir jedoch einiges davon bereits hier vorweg nehmen; es mag für den Leser interessant sein, im Anschluss daran bereits einen Blick auf 9.2.2 zu werfen.

Wir abstrahieren also zunächst von der Repräsentation eines Automaten als Transitionsgraph. Stattdessen führen wir sogenannte *induktive Systeme* ein, welche zunächst nur zwei Operationen unterstützen:

- Ermittlung des Anfangszustands
- Ermittlung aller Übergänge samt der Nachfolgezustände für einen gegebenen Zustand

Abgesehen von den Vorzügen für das Design des Softwaresystems ist eine solche induktive Formulierung des Systemverhaltens vom mathematischen Standpunkt aus wohl auch die natürlichste. Wir verlangen von den induktiven Systemen außerdem die Zusicherung, dass die erreichbare Zustandsmenge endlich ist. Es sei auch darauf hingewiesen, dass die Sprachen, die wir untersuchen, grundsätzlich präfix-abgeschlossen sind und wir somit immer davon ausgehend dürfen, dass jeder Zustand Endzustand ist. Die Erweiterung unserer induktiven Systeme um die Abfrage, ob ein Zustand Endzustand ist, wäre indes ohne weiteres möglich.

Die meisten dieser induktiven Systeme stellen nun in FastAsy eben Transformationen anderer induktiver Systeme dar. Zu diesem Zweck werden sie mit dem Ausgangssystem

initialisiert, d.h. sie erhalten eine Referenz auf dieses. Wird dem neuen System nun eine der beiden obigen Fragen gestellt, beantwortet es sie sozusagen durch eine *many-one-Reduktion* der Anfrage auf entsprechende Fragen an das Ausgangssystem: Die Antwort wird aus den Antworten des Ausgangssystems zusammengesetzt. In Wirklichkeit erlauben wir den induktiven Systemen freilich auch noch, sich dabei bestimmte Informationen über verschiedene Anfragen hinweg zu merken, um z.B. Caching-Strategien zu ermöglichen, so dass es sich nicht immer um Reduktionen im strengen Sinne handelt.

Beim *Retracing* in 7.4 wird uns dann derselbe Gedanke in Form von *linearen Systemen* nochmals begegnen. Dort behandeln wir statt Automaten aber Kantenzüge in Automaten. Diese sind natürlich aufgrund ihrer linearen Struktur eigentlich einfacher zu handhaben, lassen sich aber trotzdem nicht ohne weiteres als induktive Systeme betrachten, denn im Gegensatz zu letzteren können bei Kantenzügen Knoten (d.h. Zustände) ja mehrfach vorkommen, so dass wir nicht damit rechnen können, auf die Frage nach dem Nachfolger eines Zustands eine eindeutige Antwort zu bekommen. Als Alternative dazu, die Zustände nun z.B. durch einen zusätzlichen Index eindeutig zu machen, versehen wir lineare Systeme mit einem internen Zustand, und statten damit das Interface mit folgenden Operationen aus:

- Rücksetzung zum Anfangszustand mit Rückgabe desselben
- Ermittlung des nächsten Schrittes und des Nachfolgezustands

Man beachte, dass im Gegensatz zu den induktiven Systemen bei der zweiten Operation kein Zustand als Parameter angegeben wird, da ja Bezug auf den internen Zustand genommen wird. Nach dem Aufruf der Operation wird dieser implizit auf den Nachfolgezustand gesetzt. (Die Semantik ähnelt damit stark der eines *forward iterator* aus der STL.)

Bei den induktiven System wäre ein solcher Ansatz, d.h. die Verwaltung der Zustandsinformation als Teil des Systems selbst, natürlich gar nicht möglich, da dort Traversierungen der Zustände ja nicht linear erfolgen, sondern in irgendeiner Weise immer *Mengen* von zu besuchenden Zuständen verwalten müssen.

Man beachte auch, dass weder lineare noch induktive Systeme ohne weiteres in der Lage sind, ihre Rückwärtsabläufe zu beschreiben. Dazu bedarf es in beiden Fällen zunächst der Materialisierung, d.h. durch eine explorative Suche muss die gesamte Struktur aufgebaut und mit Rückwärtsverweisen ausgestattet werden. FastAsy enthält Klassen, die dies erlauben, und diese stellen die Ergebnisse des Prozesses wieder in Form induktiver bzw. linearer Systeme bereit, so dass solche Materialisierungen an beliebiger Stelle vollständig transparent eingeschoben werden können, d.h. ohne dass die vorausgehende oder nachfolgende Bearbeitungsstufe verändert werden müsste. Relevant ist das Thema der Rückwärtsabläufe vor allem für die erste Stufe des Retracing (siehe 7.4.1), wo, wie wir sehen werden, sowohl die Eingabe in natürlicher Form rückwärts vorliegt als auch eine effiziente Verarbeitung nur auf genau dieser umgedrehten Eingabe erfolgen kann. (Einige Ideen zur Realisierung umkehrbarer induktiver Systeme finden sich in 11.4.)

## *7.2 Berechnung eines (deterministischen) endlichen Automaten*

In diesem ersten Schritt wollen wir aus der Beschreibung eines Systems einen deterministischen endlichen Automaten konstruieren, der sprachgleich zum r-Erreichbarkeitsgraphen des Systems ist. Man beachte, dass „sprachgleich“ hier auch

automatisch das zeitliche Verhalten des Systems an sich mit einbezieht, da die in Rede stehende Sprache ja in Form von Verweigerungsmengen bereits alle Zeitinformation enthält.

Zur späteren Berechnung einer Vorwärtssimulation zwischen zwei zu vergleichenden Systemen wie in 7.3 ist es eigentlich nicht notwendig, die Automaten *beider* Systeme in deterministischer Form vorliegen zu haben. Diese Forderung gilt nur für denjenigen Automaten, der das Verhalten des anderen simulieren soll. Wir sind aber erstens sowieso in der Regel daran interessiert, die Simulation in beiden Richtungen durchzuführen. Zweitens scheint es interessanterweise so, als wären die deterministischen Automaten, obwohl im *worst case* exponentiell größer als ihre nicht-deterministischen Pendants, in der Regel tatsächlich kleiner. So werden wir im Normalfall immer für beide Automaten die deterministische Variante berechnen.

### 7.2.1 Vom Quellsystem zum Erreichbarkeitsgraphen

Obwohl wir uns in der vorliegenden Arbeit weitestgehend auf Petrinetze als untersuchte Systeme beschränken, spricht aus der Sicht von FastAsy nichts dagegen, auch Systeme, die in anderer Form spezifiziert sind zu untersuchen, beispielsweise Prozessalgebren. Die höheren Schichten von FastAsy abstrahieren nämlich vollständig von der Beschreibung des Systems selbst und fordern von einem solchen Quellsystem wie bereits erwähnt lediglich die Fähigkeit, einerseits seinen Anfangszustand zu benennen und andererseits zu einem gegebenen erreichbaren Zustand alle Übergänge samt der zugehörigen Zielzuständen aufzulisten (siehe 7.1).

Die Art und Weise, in der FastAsy die Übergänge modelliert, gestattet es dem Quellsystem weiter, die Übergänge zu annotieren und weiter sogar gleiche Übergänge mehrfach, aber mit verschiedenen Annotationen, anzugeben. Im Falle von Petrinetzen verwenden wir dies, um zu Aktionen $l(t)$ auch $t$ selbst zu speichern. Dies wird uns die Angabe des Pfads, welcher die Simulation zum Scheitern bringt, auf Transitionsebene erleichtern. Die Simulation hingegen bezieht natürlich nur die Beschriftung der Kanten ein, nicht die Annotation.

Um tatsächlich einen endlichen Automaten zu erhalten, sind unter Umständen noch einige Dinge zu beachten: Zum einen liegt das Alphabet der Aktionen $\Sigma$ ja nicht explizit vor; nach allem, was wir wissen, kann es beliebig groß sein. Wir vergewissern uns aber wie in [BV04] schnell, dass es ausreicht, $\Sigma$ als die Menge derjenigen Aktionen anzunehmen, die die untersuchten Systeme auch zumindest potentiell ausgeben können, also $l(T_1) \cup l(T_2)$. (In Wirklichkeit werden wir sogar $l(T_1) = l(T_2)$ unterstellen, da wir andernfalls gar nicht erwarten könnten, eine sinnvolle Aussage über die relative Effizienz zu gewinnen.) Zum anderen liegt zum Beispiel bei den PTT-Netzen ein Fall vor, in dem unendlich viele Markierungen auftreten können, da ein Wert auf einer Uhr immer weiter voranschreitet, wenn die zugehörige Stelle nicht geleert wird – man betrachte etwa den pathologischen Fall einer isolierten und markierten Stelle. Man behilft sich in diesem Falle leicht damit, Uhren nur so weit voranschreiten zu lassen, wie ein erhöhter Wert noch einen Unterschied im Systemverhalten bedingen kann, konkret im Falle der PTT-Netz also nur so lange, bis keine ausgehendee Kante mehr existiert, dessen obere Intervallgrenze höher als der Uhrenwert ist. Ausführlicher wurde dies für PTT-Netze ja im ersten Teil der Arbeit im Abschnitt 1.4.6 behandelt.

Es sei noch erwähnt, dass wir an dieser Stelle in FastAsy auch die Möglichkeit haben, nicht den kompletten r-Erreichbarkeitsgraphen zu generieren, sondern nur jeweils Übergänge mit maximalen Verweigerungsmengen aufzunehmen und den gesamten Teilmengenabschluss implizit zu lassen. Die theoretische Rechtfertigung und die Implikationen für weitere Verarbeitungsschritte werden im Teil I in Kapitel 2 angegeben.

### 7.2.2 Lambda-Übergänge eliminieren

Der Erreichbarkeitsgraph, den wir im vorigen Schritt generiert haben, ist natürlich im Allgemeinen nicht-deterministisch und enthält Kanten, die mit λ beschriftet, also nicht beobachtbar, sind. Die Entfernung der λ-Kanten kann nun bekanntlich wie in Teil I, 1.3.1.9 beschrieben erfolgen, indem man einen neuen Automaten konstruiert, dessen direkte Zustandsübergangsfunktion gegeben ist durch die Einschränkung der erweiterten Zustandsübergangsfunktion auf Wörter mit Länge eins (d.h. solche, die genau einen beobachtbaren Übergang enthalten):

$$\delta'( q, a ) := \delta\hat{}( q, a )$$

Eine Umformung ergibt:

$$\delta'( q, a ) := \lambda\text{-Hülle}( \bigcup_{p \in \lambda\text{-Hülle}(q)} \delta( p, a ) )$$

Außerdem muss eventuell die Menge erlaubter Endzustände korrigiert werden, so dass sie den neuen Startzustand enthält, wenn im ursprünglichen Automaten sich ein Endzustand in der λ-Hülle des Startzustands befindet. Der uns vorliegende Fall ist ja aber insofern speziell, dass Endzustände keine Rolle spielen bzw. präziser, dass jeder Zustand ein Endzustand ist. (Wir erinnern uns, dass FastAsy ganz auf die Modellierung von Endzuständen verzichtet.) Man vergewissert sich nun einfach, dass in einem solchen Fall auch die folgenden beiden alternativen Konstruktionen zu äquivalenten Automaten führen:

Entweder beschränkt man sich darauf, nur solche Übergänge zu ersetzen, bei denen die sichtbare Aktion ganz am Anfang kommt:

$$\forall q \neq q_0\text{: } \delta'( q, a ) := \lambda\text{-Hülle}( \delta( q, a ) ),$$
$$\delta'( q_0, a ) := \lambda\text{-Hülle}( \bigcup_{p \in \lambda\text{-Hülle}(q0)} \delta( p, a )$$

oder aber auf solche, bei denen sie am Ende steht:

$$\delta'( q, a ) := \bigcup_{p \in \lambda\text{-Hülle}(q)} \delta( p, a )$$

Wie man durch Vergleich mit der Umformung weiter oben sofort sieht, produzieren beide Konstruktionen zumindest nicht mehr Kanten als die ursprüngliche; in praktischen Fällen ist sogar damit zu rechnen, dass weniger Kanten entstehen. Zusätzlich besteht dadurch auch die Möglichkeit, dass Zustände nicht mehr erreichbar sind und so entfallen können. Man beachte zu letzterem Punkt, dass durch die induktive Definition von endlichen Automaten in FastAsy solche Zustände nicht explizit gesucht und eliminiert werden müssen, sondern von selbst wegfallen.

Die unschöne Sonderbehandlung des Anfangszustands in der erstgenannten Alternative scheint störend (insbesondere auch beim späteren *Retracing*), deswegen haben wir uns entschieden, die letztgenannte Konstruktion umzusetzen.

Allen drei Konstruktionen gemeinsam ist hingegen die Eigenschaft, dass sie Determinismus nicht erhalten. Insofern muss dieser Schritt in jedem Fall vor 7.2.4 erfolgen. Man vergewissert sich weiter auch schnell, dass Satz 2.2.3.1 aus Teil I, welcher uns garantiert, dass die Elimination der Lambda-Übergänge sowohl mit vollständigen als auch mit nur maximalen Verweigerungsmengen korrekt arbeitet, für alle drei Varianten gilt.

### 7.2.3 Zustände indizieren

Nach allem, was wir auf dieser Abstraktionsebene wissen, kann die Zustandsinformation, mit denen die Knoten des Erreichbarkeitsgraphen beschriftet sind, relativ komplex sein. Spätestens wenn wir in 7.2.4 mit Mengen solcher Zustände arbeiten müssen, scheint es nicht mehr opportun, sich diese Zustandsinformation vollständig zu merken. Stattdessen indizieren wir die Zustände mit fortlaufenden natürlichen Zahlen, so dass wir Mengen davon effizient als Bitvektoren speichern können. Man beachte, dass es für alle späteren Berechnungsschritte ja nur mehr wichtig ist, die Identität von Zuständen unterscheiden zu können, die Details ihrer Beschriftung werden nicht mehr benötigt. Aus diesem Grund lässt sich nun auch die Indizierung, wie man sich leicht vorstellen kann, durch ein induktives Interface kapseln, ohne dass man später unter Umgehung dieses Konzepts Rückgriffe auf die Indexstruktur bräuchte. Lediglich zur Ausgabe des Endergebnisses wird auf die ursprüngliche Zustandsinformation dann ggf. noch einmal zugegriffen, um dem menschlichen Benutzer einen zusätzlichen Orientierungspunkt zu bieten.

Es sei bemerkt, dass die Indizierung der Zustände keine komplette Materialisierung des Automaten voraussetzt. Eine Liste der verschiedenen Zustandsbeschreibungen muss indes natürlich im Speicher gehalten werden. Die entstehenden Indizes sollen nämlich zusammenhängend sein, um die erwähnten Bitvektoren zur Mengendarstellung klein zu halten. Außerdem muss die Zuordnung zur eigentlichen Zustandsbeschreibung ja zumindest für die endgültige Ausgabe umkehrbar sein. Beides verbietet Versuche, ohne Speicherung der Zustände etwa durch Hashfunktionen an Indizes zu kommen.

Der ganze Schritt der Indizierung ist natürlich rein konzeptuell gesehen nicht Bestandteil des eigentlichen Algorithmus zur Berechnung des deterministischen Automaten, es handelt sich ja eher um ein technisches Artefakt. Wir führen ihn hier aber aus mehreren Gründen trotzdem auf: Erstens ist der darauf folgende Schritt der Potenzautomatenkonstruktion ohne die Indizierung wohl kaum effizient zu implementieren, und zweitens ist es für die Interpretation der Zwischenergebnisse in FastAsy unter Umständen wichtig, diesen Schritt zu kennen.

Die Indizierung der Zustände lässt sich offensichtlich mit der Elimination der λ-Kanten vertauschen und führt zu sprachäquivalenten Resultaten. Wir führen momentan die beiden Schritte jedoch in der genannten Reihenfolge durch, weil wie schon ausgeführt die Möglichkeit besteht, dass der Zustandsraum und somit die benötigte Menge von Indizes nach der Durchführung von 7.2.2 kleiner ist.

### 7.2.4 Potenzautomatenkonstruktion

Da wir in 7.3 mit Hilfe einer Vorwärtssimulation entscheiden wollen, ob eine Inklusion zweier durch endliche Automaten gegebenen Sprachen vorliegt, muss zumindest der Automat mit der mutmaßlich größeren Sprache (präziser: derjenige, der die Simulation zum Scheitern

bringen kann, wenn er ein Wort nicht enthält) wie bereits erwähnt deterministisch sein. Wir haben ja bereits argumentiert, dass man üblicherweise sogar beide Automaten in deterministische Form bringen möchte.

Wir bedienen uns hierzu nun der bekannten und z.B. in [HU94] beschriebenen Konstruktion des Potenzautomaten, der sprachäquivalent zum Ausgangsautomaten, aber zusätzlich per Konstruktion deterministisch ist. Auch in diesem Fall ist klar, dass die Berechnungsvorschrift induktiv formuliert werden kann: Der Anfangszustand überträgt sich einfach, und die Nachfolger eines (zusammengesetzten) Zustands im Potenzautomaten berechnen sich einfach unter Rückgriff auf die Nachfolger der einzelnen Teilzustände im ursprünglichen Automaten.

Gegenüber früheren FastAsy-Versionen müssen wir diesen Schritt deutlich verallgemeinern, denn wie in 7.2.1 erwähnt, arbeiten wir auch mit r-Erreichbarkeitsgraphen, in denen nur maximale Verweigerungsmengen explizit aufgeführt werden. In einem solchen Fall leistet aber die normale Potenzautomatenkonstruktion nicht mehr das gewünschte, da das Ergebnis im Allgemeinen immer noch nicht-deterministisch ist, und wir benötigen eine angepasste Vorschrift, um von den Übergängen der Teilzustände auf diejenigen des Potenzautomaten schließen zu können; s. Teil I , 2.2.4.

Zur Potenzautomatenkonstruktion muss noch erwähnt werden, dass es sich dabei um einen der wirklich aufwändigen und daher zeitkritischen Schritte handelt. Theoretisch kann diese Tatsache natürlich daran festgemacht werden, dass die Ausgabe mit einem *worst case* von $O(2^n)$ Knoten und $O(m*2^n)$ Kanten extrem teuer sein kann (wobei n die Anzahl der Zustände und m die Anzahl der Übergänge des ursprünglichen Systems bezeichnet). Wie schon erwähnt, stellt man bei praktischen Beispielen jedoch dankenswerter Weise oft fest, dass der deterministische Automat sogar kleiner als der ursprüngliche ist, so dass die Größe der Ausgabe nicht das wirkliche Problem darstellt. Schwerer wiegt daher die Tatsache, dass die Konstruktion mit dem nicht-deterministischen Automaten eine „große“ Eingabe bekommt, diese auf jeden Fall komplett berücksichtigen muss und in der Regel sogar jeweils mehrfach auf dieselben Teile der Eingabe zugreift.

Die Elimination der λ-Kanten lässt sich in $O(n^3)$ durchführen, und man kann pathologische Beispiele konstruieren, in denen diese mehr Schritte benötigt als die Potenzautomatenkonstruktion. Für die Praxis bleibt jedoch die Erkenntnis relevant, dass die Konstruktion des Potenzautomaten einen *bottleneck* in der Performance darstellt.

Als mögliche Erklärung, warum die Größe des Potenzautomaten bei den untersuchten Beispielen so auffällig weit unter dem *worst case* bleibt, scheint es so, als könne man einen Teil der Komplexität des nicht-deterministischen Automaten als Artefakt der Umsetzung des Petrinetzes in einen endlichen Automaten deuten. Bekanntlich führt dies ja zum Interleaving an sich nebenläufiger Vorgänge, und zumindest für solche Vorgänge, die intern und daher gar nicht beobachtbar sind, scheinen diese rein modellbedingten Auswahlmöglichkeiten beim Übergang zum deterministischen Automaten wieder zu verschwinden. Weiter halten wir es für eine interessante Beobachtung, dass Fehler in der Spezifikation eines Systems (etwa Tippfehler in der Eingabedatei) dessen Verhalten oft „komplexer“ machen in dem speziellen Sinne, dass die Größe des Potenzautomaten explodiert. Eine genauere Untersuchung des Phänomens steht indes noch aus.

## 7.3 Erstellen einer (partiellen) Simulation

In diesem Schritt versuchen wir nun, für zwei Automaten $A_1$ und $A_2$ eine Simulation im dem Sinne zu finden, wie sie in Teil I in Abschnitt 1.3.1.4 definiert wird.

Da zumindest $A_2$ deterministisch ist, ist die Simulation, wenn sie existiert, wie man sieht eindeutig bestimmt. Die Aussage, die sich in diesem Fall durch FastAsy gewinnen lässt, ist die, dass die Sprache von $A_1$ in derjenigen von $A_2$ enthalten ist. Freilich ist in der Simulation noch die Information enthalten, *wie* genau $A_2$ die Schritte von $A_1$ nachvollzieht, jedoch ist die Komplexität dieser Information allein durch die Mächtigkeit der Menge in der Regel einfach zu hoch, als dass der menschliche Benutzer noch Nutzen daraus ziehen könnte.

Schlägt der Aufbau der Simulation jedoch an einem Punkt fehl, d.h. existieren korrespondierende Zustände, in denen $A_1$ eine bestimmte Aktion ausführen könnte und $A_2$ nicht, so lässt sich eine sehr aussagekräftige Information gewinnen, wenn es uns gelingt, festzustellen, welche Schritte in $A_1$ es waren, die das Scheitern verursacht haben; sie bilden den sogenannten *Fehlerpfad*. Dieser Fehlerpfad ist im Wesentlichen ein Wort, das in der Sprache von $A_1$, aber nicht in der von $A_2$ vorkommt und somit ein Beleg für ein *mögliches langsames* Verhalten von $A_1$ ist. Er ist ein konkreter Hinweis darauf, in welchen Umgebungen es aus Sicht der Effizienz ungünstig wäre, $A_2$ als Komponente durch $A_1$ zu ersetzen.

Da wir später daran interessiert sein werden, nachzuvollziehen, welche Abläufe im ursprünglichen nicht-deterministischen Automaten den Fehlerpfad produzieren, erweitern wir die Informationen, die wir mit dem Fehlerpfad abspeichern, dahingehend, dass wir uns nicht nur die Abfolge der Kantenbeschriftungen in $A_1$ merken, sondern auch die Knoten, die dabei durchlaufen werden. Aus dem Wort, das $A_1$ produzieren kann, wird dann ein Kantenzug in $A_1$. In diesem Sinne sprechen wir dann auch von einem *Fehlerpfad in $A_1$*.

Um nun aber zunächst einen solchen Fehlerpfad gewinnen zu können, fügen wir in der beschriebenen Situation noch ein spezielles *Fehlerelement* $(q,\perp)$ in die Simulation ein, wobei q hier derjenige Zustand ist, den $A_1$ erreicht, nachdem er die letzte Aktion ausgeführt hat. (Für $A_2$ kann es natürlich genau keinen solchen Zustand geben.) Weiter erweitern wir die mit der Simulation abgespeicherte Information dahingehend, dass wir uns für jedes neu eingefügte Paar das Vorgängerpaar merken und die Aktion, die $A_1$ auf dem Weg von dort ausgeführt hat. So können wir uns anschließend vom Fehlerelement ausgehend entlang der Vorgänger bis zum Paar der Anfangszustände zurückhangeln. Projiziert man die durchlaufenen Simulationselemente auf die erste Komponente (die Zustände von $A_1$), so erhält man genau den Fehlerpfad in $A_1$, allerdings natürlich zunächst in umgedrehter Reihenfolge.

Man beachte, dass per Konstruktion der Weg zurück für eine einmal aufgebaute Simulation eindeutig ist. Beim Aufbau der Simulation selbst hingegen existieren natürlich Wahlmöglichkeiten, da weder die Reihenfolge, in der die Kanten abgearbeitet werden, spezifiziert ist noch die Reihenfolge, nach der neu entdeckte Zustände untersucht werden. Letzteres entspricht z.B. der Wahlmöglichkeit zwischen *breadth first search* und *depth first search*. FastAsy erlaubt beide Ansätze, und es sind uns bis dato noch keine Argumente bekannt, die dafür sprechen würden, klar das eine oder andere Vorgehen vorzuziehen. Für BFS spräche die Tatsache, dass man zunächst einen möglichst kurzen Fehlerpfad in $A_1$ erhält. Dies ist jedoch nur von relativem Wert, denn man beachte, dass dieser aus einem Pfad im nichtdeterministischen Vorgänger von $A_1$ mit sehr langen $\lambda$-Pfaden hervorgegangen sein kann, so dass man bestenfalls in einem heuristischen Sinne die Hoffnung hat, ein kurzes Wort

aus RFS gefunden zu haben. (In 11.5 werden wir noch kurz eine Idee ausführen, wie man tatsächlich an die kürzesten Worte in RFS kommen kann, wenn man etwas zusätzlichen Aufwand in Kauf nimmt und $A_1$ *nicht* als Potenzautomat vorliegt.) Noch „philosophischer" ist die Argumentation für DFS: Dort könnte man allenfalls anführen, bei Systemen, die einander aus Effizienzsicht ähnlich sind – und das werden die untersuchten Systeme in der Regel sein – wäre es vorteilhaft, früh zu den langen RT-Sequenzen vorzudringen, da erst diese das simulierende System genügend in Zugzwang versetzen, damit es eine Aktion nicht mehr nachbilden kann. Für den Fall, dass tatsächlich eine Inklusion der Sprachen vorliegt, wird man freilich so oder so die Simulation komplett aufbauen müssen.

Als letztes sei zur Simulation noch erwähnt, dass man im Hinblick auf eventuell nur implizit vorhandene Verweigerungsmengen auch an dieser Stelle wieder eine Generalisierung vornehmen muss (siehe dazu auch Abschnitt 2.2.5 aus Teil I): Die Vorschrift zur Auswahl der Kante, mit der $A_2$ eine Aktion von $A_1$ simuliert, wird austauschbar gemacht und kann im allgemeinen Fall in der Tat sogar von den übrigen Übergängen, die $A_2$ vom Ausgangszustand aus durchführen kann, abhängen. (Für explizit vorhandene Untermengen besteht diese Vorschrift einfach aus der Auswahl einer identisch beschrifteten Kante.)

Während diese Generalisierung den *simulierenden* Automaten betrifft, erlaubt uns übrigens nach Teil I, 2.2.5.3 der Einsatz impliziter Mengen beim *simulierten* System ohne zusätzlichen Aufwand eine beträchtliche Effizienzsteigerung: Es reicht nämlich aus, alle Zeitschritte zu prüfen, die tatsächlich explizit im Automaten stehen, statt all derer, die dadurch impliziert werden.

## *7.4 Retracing*

Wir haben in 7.1 gesehen, dass wir in mehreren Schritten von Worten aus RFS zu Worten in einem deterministischen Automaten gelangen, der für die Simulation verwendet wird. In 7.3 haben wir angesprochen, wie es zu verstehen ist, wenn wir vom Fehlerpfad *in einem bestimmten* Automaten reden. Unter dem Begriff *Retracing* verstehen wir nun generell das Auffinden eines Fehlerpfads *im* jeweils vorausgehenden Zwischenergebnis. Für jede Transformation, die wir beim Aufbau des DEA durchführen, benötigen wir nun eine passende Vorschrift für das Retracing. Freilich existiert dabei immer der letzte Rückzugspunkt, einen solchen Fehlerpfad (derer es ja durchaus mehrere geben kann) durch explorative Suche zu finden, doch dies wollen wir der hohen Komplexität wegen möglichst vermeiden, und in der Tat konnten wir für jeden der Schritte auch eine angepasste und effizientere Vorschrift für das Retracing finden. Einen Überblick über die Datenflüsse beim Retracing gibt die Abbildung am Anfang von 9.6.

Man beachte, dass die Vorschrift 7.4.1 die Betrachtung von Rückwärtsabläufen erfordert; d.h. die Vorschrift arbeitet in ihrer natürlichen Form auf den Umkehrungen von Fehlerpfad und Automat. Wie erwähnt kann FastAsy Umkehrungen sowohl linearer als auch induktiver Systeme zur Verfügung stellen (um den Preis einer Materialisierung der ursprünglichen Objekte). Wir werden in 9.4.2.7 noch genauer besprechen, wie sich dies in die Architektur von FastAsy einfügt. Einstweilen soll es genügen, den Leser darauf hinzuweisen, dass eine Formulierung der Vorschrift auf Rückwärtsabläufen trotzdem ohne weiteres umgesetzt werden kann. Grundsätzlich formulieren wir aber die Retracing-Vorschriften auf Vorwärtsabläufen – dies wird hier deshalb explizit erwähnt, weil das Verständnis des

Retracing in älteren FastAsy-Versionen sich immer komplett auf Rückwärtsabläufe bezogen hat.

Weiter sei noch erwähnt, dass der Fehlerpfad, interpretiert man die Simulation als Ausschnitt des Produktautomaten, ein Weg in der Simulation ist, d.h. es kommt kein Element der Simulation zweimal vor. Leider stellt die Projektion des Fehlerpfades auf den Automaten $A_1$ (um in den obigen Bezeichnungen zu bleiben) nur mehr einen allgemeinen Kantenzug dar. Intuitiv ist dies auch einsichtig: Für $A_1$ kann es nötig sein, mehrmals dasselbe Verhalten an den Tag zu legen, was $A_2$ dann jedes Mal auf eine andere Art und Weise zu simulieren versucht. $A_2$ wird sozusagen durch die Wiederholung „in die Ecke gedrängt".

Wie wir nun also in 7.2 induktive Systeme aneinandergereiht haben, so werden wir es beim Retracing in umgekehrter Reihenfolge mit linearen Systemen tun. Dabei ist offensichtlich, dass wir als sekundäre Eingabe auch den Rückgriff auf das jeweilige induktive System brauchen. Präziser lässt sich ein einzelner Schritt beim Retracing folgendermaßen formulieren:

Gegeben sind induktive Systeme I, J und eine beschriftete Kante (q,a,q') in J. Weiterhin ist die Vorschrift bekannt, durch die J aus I hervorgeht. Gesucht ist nun ein zu diesem Übergang *korrespondierender* Kantenzug $(p_0,b_1,p_1,b_2,\ldots,p_n)$ in I. Dabei ist $p_0$ als das „alte" $p_n$ aus dem letzten Retracing-Schritt bekannt. (Diese letzte Bedingung bedeutet nichts anderes, als dass sich nacheinander gefundene Kantenzüge aneinander reihen lassen.) Man beachte, dass sich der Begriff *korrespondierend* in dieser Allgemeinheit nicht schärfer fassen lässt. Wir werden uns in der Regel im folgenden für jeden einzelnen Bearbeitungsschritt Klarheit darüber verschaffen müssen, welche Kriterien für den gesuchten Kantenzug existieren. (Formal so nicht haltbar, aber nichtsdestotrotz der Intuition dienlich wäre es vielleicht zu sagen, man sucht einen Kantenzug, der durch „Rückwärtsanwendung" der gegebenen Vorschrift aus dem Übergang entsteht.)

Vergleicht man die neu geschaffene modulare Struktur des Retracing mit dem monolithischen Algorithmus älterer FastAsy-Versionen, so stellt man einen deutlichen Zuwachs an Aufwand bei der Programmierung fest, der natürlich gerechtfertigt sein will. Zu diesem Zweck halte man sich vor Augen, dass die modulare Struktur der Algorithmen in 7.2 ja nur dann voll zur Entfaltung kommen kann, wenn auch gewährleistet ist, dass sich auch für Neukombinationen der Komponenten aus der Simulation ein verwertbares Ergebnis gewinnen lässt, ohne dass dafür jedes Mal das komplette Retracing über alle Stufen hinweg ausgetauscht werden muss.

### 7.4.1 Retracing der Einzelzustände

In diesem wohl substantiellsten Schritt beim Retracing haben wir einen (vorwärts-gerichteten!) Übergang (Q,a,Q') im Potenzautomaten (DEA) gegeben und suchen einen Kantenzug im (buchstabierenden, zustands-indizierten) NDEA mit folgenden Eigenschaften:

- Der Kantenzug besteht nur aus einem einzigen Übergang (p,b,p')
- Es gilt b R a
  (Die Relation R ist abhängig davon, ob mit impliziten Teilmengen gearbeitet wird. Falls nicht, gilt einfach R = id.)
- $p \in Q$, $p' \in Q'$
- p entspricht dem p' des vorherigen Retracing-Schritts

Man bemerkt nun aber, dass eine beliebige Auswahl von p' noch nicht ausreicht, um sicherzustellen, dass sich die zusammengesetzte Sequenz später bis zum Ende fortsetzen lässt; d.h. jede Sequenz, die nach obigen Kriterien schrittweise aufgebaut wird, entspricht zwar dem Anfang *irgendeines* Ablaufs des nicht-deterministischen Automaten, nur eben im Allgemeinen nicht gerade eines Ablaufs, der den gesamten Fehlerpfad produzieren kann. Wir wollen dies an einem kurzen Beispiel verdeutlichen:

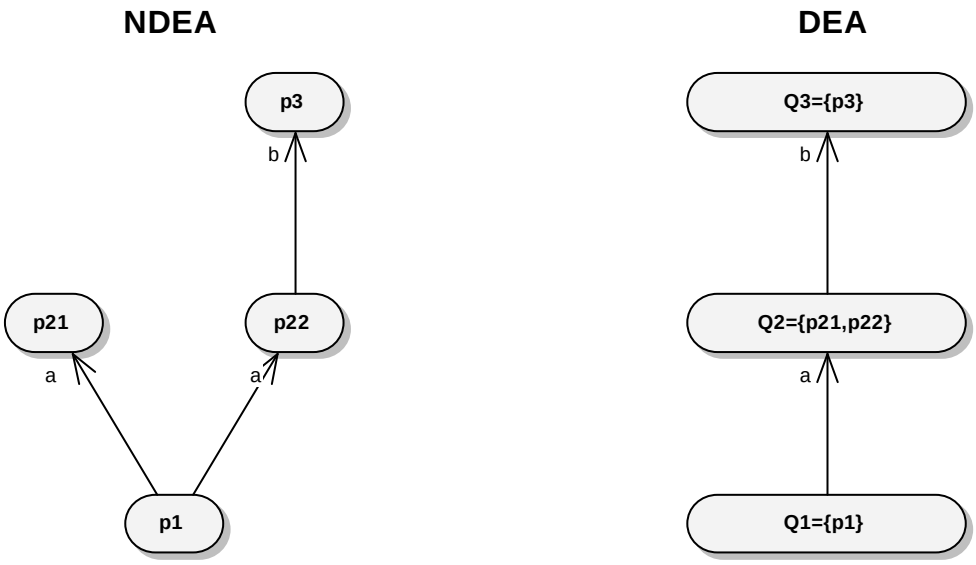



**Abbildung 27: Beispiel Retracing**

Im gegebenen Beispiel entsprechen für den Retracing-Schritt mit p=p1, Q=Q1 und Q'=Q2 sowohl (p1, a, p21) als auch (p1, a, p22) den obigen Kriterien. Trotzdem lässt sich das Retracing nur für die Wahl von p'=p22 fortsetzen.

Die Forderung, einen Retracing-Schritt so zu wählen, dass sich das Retracing konsistent fortsetzen lässt, haben wir allgemein *clairvoyance* genannt. Es scheint nicht möglich, diese Eigenschaft für Vorwärtsabläufe ohne *brute force*, d.h. Ausprobieren aller Möglichkeiten, hier zu erreichen.

Betrachtet man jedoch die Rückwärtsabläufe von Fehlerpfad und nicht-deterministischem Automaten, so stellt man fest, dass man aus dem Q des allerersten Retracing-Schritts (im Beispiel also Q3), welches ja aus dem Fehlerelement (Q3,⊥) der Simulation stammt, ein beliebiges p wählen darf und damit bei Einhaltung obiger Regeln immer bis zum Anfangszustand gelangen kann. Im Beispiel ergibt sich von p3 aus automatisch der Rückwärtsschritt nach p22 und von dort aus nach p1. (Für den komplexeren Fall, dass implizite Teilmengen verwendet werden, überlegt man sich schnell, dass man für den Rückwärtsschritt im NDEA jeden Zeitschritt verwenden darf, der den im Fehlerpfad angegebenen enthält.)

An dieser Stelle des Retracing steht man also vor der unangenehmen Wahl, entweder mit dem NDEA ein sehr großes Zwischenergebnis zu materialisieren, nur um Rückwärtsabläufe behandeln zu können, oder sich für die wenig elegante und recht aufwändige Methode des *Backtracking* zu entscheiden. In diesem speziellen Fall existiert jedoch noch eine weitere Möglichkeit: Es sei (Q,a,Q') wie oben die Vorwärtskante im Potenzautomaten. Da wir rückwärts vorgehen, haben wir ein $p' \in Q'$ (statt eines $p \in Q$) gegeben. Hätten wir nun die

Rückwärtsabläufe des NDEA zur Verfügung, könnten wir unter den Rückwärtsübergängen von p' einen mit passender Kante (d.h. mit a beschriftet und mit Ziel in Q) suchen. Sollen wir die Frage nur unter Zuhilfenahme des normalen NDEA beantworten, können wir aber immer noch die Vorwärtsübergänge aller Teilzustände aus Q auf eine passend beschriftete Kante mit Ziel p' hin untersuchen. Im Beispiel: Wenn wir im NDEA alle passenden b-Kanten, die in p3 *eingehen* suchen, so müssten wir eigentlich alle Knoten darauf prüfen, ob von ihnen entsprechende b-Kanten ausgehen. Da aber Q2={p21, p22} bekannt ist, können wir diese Prüfung auf p21 und p22 beschränken. Diese Möglichkeit ist momentan allerdings noch nicht implementiert.

### 7.4.2 Umkehrung des Fehlerpfads

Obwohl wir ja im Rahmen dieses Kapitels von der genauen Umsetzung der Rückwärtsabläufe noch abstrahieren wollen, erwähnen wir diesen Verarbeitungsschritt hier trotzdem, um dem Benutzer die korrekte Interpretation etwaiger Zwischenergebnisse von FastAsy zu ermöglichen:

Da alle weiteren Stufen des Retracing vorwärts ablaufen, wird unmittelbar nach dem Retracing der Einzelschritte der entstandene (umgekehrte) Fehlerpfad nochmals umgekehrt. Die dazu notwendige Materialisierung verursacht nur unerheblichen Aufwand, da es sich ja um ein Objekt mit vergleichsweise sehr beschränkter Größe handelt.

### 7.4.3 Retracing der Zustandsbeschreibung

In 7.2.3 haben wir gesehen, dass es nötig ist, vorkommende Zustandsbeschreibungen zu indizieren und im Weiteren mit den Indizes zu arbeiten. Auch diesen Schritt wollen wir für den Fehlerpfad rückwärts abwickeln, denn obwohl die Zustandsbeschreibung ja für die meisten Stellen in FastAsy transparent ist, enthält sie im Allgemeinen Information, die es dem menschlichen Benutzer erleichtern, das Systemverhalten nachzuvollziehen. Bei Petrinetzen beispielsweise erhält der Benutzer nicht nur die geschalteten Transitionen, sondern auch die einzelnen (zeitbehafteten) Markierungen ausgegeben.

Die entsprechende Retracing-Komponente muss natürlich mit der Indextabelle, die in 7.2.3 als Seitenergebnis entstanden ist, initialisiert werden. Darüber hinaus erfolgt ja einfach nur ein Zugriff auf genau diese Tabelle für jeden Zustand, so dass die Implementierung dieses Schritts trivial ist. Offenbar arbeitet dieser Schritt auf Vorwärtsabläufen ebenso gut wie auf Rückwärtsabläufen, wir verwenden jedoch momentan Vorwärtsabläufe. Ebenso offenbar ist die *clairvoyance*-Forderung erfüllt, da alle Auswahlen ja deterministisch sind.

### 7.4.4 Wiederherstellen der Lambda-Übergänge

In 7.2.2 wurden aus dem Erreichbarkeitsgraphen die λ-Übergänge entfernt. Um diesen Schritt auf dem Fehlerpfad rückgängig zu machen, gehen wir folgendermaßen vor: Wir haben einen Übergang (q,a,q') im buchstabierenden induktiven System J gegeben und suchen einen Kantenzug im ursprünglichen System I mit λ-Kanten, der folgende Eigenschaften hat:

- Der Kantenzug enthält genau am Ende den einzigen beschrifteten Übergang ($p_{n-1}$,a,$p_n$), alle anderen Übergänge sind intern
- $p_1$=q, $p_n$=q'
- $p_1$ entspricht dem $p_n$ des vorherigen Retracing-Schrittes

Nach Konstruktion wissen wir, dass es mindestens einen solchen Kantenzug $(p_1,\lambda,p_2,\lambda,\ldots,p_{n-1},a,p_n)$ geben muss, und wir können ihn durch eine angepasste Breitensuche finden. Da die Breitensuche bis auf den letzten Schritt innerhalb der $\lambda$-Hülle von $p_1$ verläuft, ist die Komplexität noch gering. Man beachte, dass die Komplexität sich bereits deutlich erhöhen würde, wenn man sich in 7.2.2 für die ursprüngliche Konstruktion entschieden hätte, denn in einem solchen Fall hätte man die zusätzliche Auswahl, an welcher Stelle man den beschrifteten Übergang unterbringt, was im Prinzip schon auf ein *backtracking* führt. So aber muss beim Besuchen eines Knotens lediglich zusätzlich geprüft werden, ob von dort aus ein a-Übergang nach $p_n$ möglich ist.

Die Auswahl eines passenden Kantenzugs ist im Allgemeinen nicht deterministisch, durch die Breitensuche legen wir aber fest, dass wir an möglichst kurzen Kantenzügen interessiert sind. Weiter beeinflussen sich die Auswahlen bei den Kantenzügen untereinander nicht, da der Zustand an der Verbindungsstelle zweier Schritte ja sowieso feststeht. Damit ist auch hier *clairvoyance* trivialerweise erfüllt.

Mit diesem Retracing-Schritt haben wir den Fehlerpfad zurückverfolgt bis in den ursprünglichen Erreichbarkeitsgraphen. Unter Umständen kann es sinnvoll sein, den Fehlerpfad nun für den menschlichen Benutzer noch mit weiteren Informationen aus dem Ursprungssystem anzureichern. Diese Informationen (z.B. bei PTT-Netzen Beschriftungen von Transitionen oder Stellen) sind dann aber spezifisch für die Modellierung und gehören deswegen nicht mehr in diejenige Applikations-Schicht, in der das Retracing stattfindet. Obwohl technisch der Retracing-Mechanismus ebenso dafür geeignet wäre, schien es stimmiger, eine solche Anreicherung stattdessen einfach als spezielles Ausgabeformat zu interpretieren und auch dergestalt zu implementieren. (Ein Blick auf die Schichtenaufteilung in 9.2.1 bestätigt uns darin, dass das Retracing möglichst nicht auf Details der Quellsystem-Schicht zugreifen sollte. Alle anderen Zugriffe auf diese Schicht laufen ja über Adapter ab.)

# Kapitel 8 Anwenderhandbuch

In diesem Kapitel soll für den Anwender praxisnah erläutert werden, wie er die derzeit „vorverdrahteten“ Möglichkeiten von FastAsy nutzen kann. Dabei sind wir uns bewusst, dass diese momentan lediglich dem entsprechen, was wir für unsere eigenen Untersuchungen benötigen und somit dem Potential der Architektur von FastAsy nicht gerecht werden. Fokus dieser Arbeit war die Schaffung der Infrastruktur sowie die Einbindung einiger neuer Konzepte aus unseren theoretischen Arbeiten, nicht die Implementierung eines „Zoos“ von vorgefertigten Komponenten. Der Anwender erhält in Kapitel 10 jedoch ausreichend Hilfestellung, um eigene Komponenten umsetzen und einbinden zu können.

## *8.1 Bedienung des CLI*

Das CLI ist ein einfaches (und wie wir später sehen werden leicht erweiterbares) Kommandozeilen-Interface. Ein Aufruf von FastAsy über das CLI hat immer folgende Gestalt:

```
> FastAsy <Befehl> [<Option>*] [<positionaler Parameter>*]
```

Der Befehl wählt dabei zunächst eine bestimmte vorgefertigte Kette von Verarbeitungsschritten aus. Durch bestimmte Optionen lässt sich diese Verarbeitungskette modifizieren oder bestimmte Glieder der Kette lassen sich in ihrem Verhalten beeinflussen. Es existieren aber auch einige globale Optionen, die sich für jeden Befehl angeben lassen. Die positionalen Parameter schließlich hängen vom Befehl ab, es handelt sich in der Regel einfach um die Angabe der Eingabedateien. Als Eingabesystem werden momentan erweiterte PTT-Netze (d.h. solche mit Lesekanten) unterstützt, welche unter anderem die Netzklassen aus [Bih98] und [Vog96] subsumieren.

### 8.1.1 Verzeichnis derzeit implementierter Befehle

#### *8.1.1.1 Erstellung des r-Erreichbarkeitsgraphen*

**Kommando**: NDEA

**Parameter**:

- [-o|--outfile <ausgabedatei>]
- [-s|--subsets]
- [-f|--outfmt <ausgabeformat>]
- <eingabedatei>

**Synopsis**: Nichtdeterministischen endlichen Automaten aus dem Eingabesystem erstellen

Das Eingabesystem wird aus der Eingabedatei eingelesen und es wird daraus der r-Erreichbarkeitsgraph berechnet.

Mit der Option –o kann eine Datei angegeben werden, in die der NDEA geschrieben wird, ansonsten erfolgt die Ausgabe auf die Konsole. Diagnoseinformationen (z.B. Debugging-Informationen oder Fortschrittsanzeigen) werden nie in die Ausgabedatei geschrieben, sondern immer auf die Konsole.

Wird die Option –s angegeben, so werden dabei alle Teilmengen von Verweigerungsmengen angegeben, auch wenn aus der maximalen Verweigerungsmenge auf

ihre Existenz geschlossen werden kann. Ohne diese Option werden solche Mengen nicht angegeben.

Nach der Option –f kann das Format der Ausgabe bestimmt werden. Momentan sind als Parameter zu dieser Option *fastasy* (Ausgabe in einer kompakten, FastAsy-eigenen Schreibweise) und *dot* (Ausgabe in einem Format, welches an Graphviz/Dot weitergereicht werden kann, um eine graphische Ausgabe zu generieren) möglich. Der Vorgabewert dabei ist *fastasy*.

#### *8.1.1.2 Berechnung eines zum r-Erreichbarkeitsgraphen äquivalenten buchstabierenden Automaten*

**Kommando**: SNDEA

**Parameter**:

- [-o|--outfile <ausgabedatei>]
- [-s|--subsets]
- [-f|--outfmt <ausgabeformat>]
- [-lc|--lambdacache]
- [-slc|--smartlambdacache]
- <eingabedatei>

**Synopsis**: Buchstabierenden nichtdeterministischen endlichen Automaten aus dem Eingabesystem erstellen

Das Eingabesystem wird aus der Eingabedatei eingelesen und es wird daraus zunächst der r-Erreichbarkeitsgraph berechnet. Im nächsten Schritt werden wie in 7.2.2 beschrieben alle nicht sichtbaren Übergänge eliminiert und dieses Ergebnis ausgegeben.

Die Optionen –lc und –slc schalten dabei Optimierungen bei der Elimination der λ-Übergänge zu (siehe 9.4.2.4). Es kann nur jeweils eine dieser Optionen gewählt werden.

Die weiteren möglichen Optionen sind dieselben wie in 8.1.1.1.

#### *8.1.1.3 Berechnung eines zum r-Erreichbarkeitsgraphen äquivalenten deterministischen Automaten*

**Kommando**: DEA

**Parameter**:

- [-o|--outfile <ausgabedatei>]
- [-s|--subsets]
- [-ms|--maxstates <maximum_number_of_states>]
- [-f|--outfmt <ausgabeformat>]
- [-lc|--lambdacache]
- [-slc|--smartlambdacache]
- <eingabedatei>

**Synopsis**: Deterministischen endlichen Automaten aus dem Eingabesystem erstellen

Das Eingabesystem wird aus der Eingabedatei eingelesen und es wird daraus zunächst der r-Erreichbarkeitsgraph berechnet. Im nächsten Schritt werden wie in 7.2.2 beschrieben alle nicht sichtbaren Übergänge eliminiert, nach einer Indizierung der Zustände erfolgt die Potenzautomatenkonstruktion und der so entstandene deterministische Automat wird

ausgegeben. Die Semantik der Potenzautomatenkonstruktion wird dabei in Abhängigkeit vom Vorhandensein der Option –s passend gewählt.

Die möglichen Optionen sind dieselben wie diejenigen in 8.1.1.1. Zusätzlich kann mit der Option –ms als Parameter eine Obergrenze für die Anzahl der Zustände im buchstabierenden Automaten, wie er in 8.1.1.2 berechnet wird, angegeben werden. Wie in 9.4.2.5 dargestellt wird, muss eine solche Grenze aus implementierungstechnischen Gründen nämlich bekannt sein. Im Normalfall wird der Benutzer aber keine solche Grenze selber angeben, und FastAsy wird in diesem Fall den fraglichen Automaten materialisieren, um die genaue Zustandszahl zu bestimmen. (So lange der Arbeitsspeicher dafür ausreichend ist, hat dies davon unabhängig auch einen positiven Einfluss auf die Performance.) Stellt sich ein mit –ms angegebener Wert als zu klein heraus, wird die Berechnung unterbrochen und ein Fehler ausgegeben. In 11.8 wird eine Möglichkeit angegeben, wie man die etwas unnatürliche Notwendigkeit, die Anzahl dieser Zustände zu kennen, um den Preis eines etwas höheren Aufwands zur Laufzeit gänzlich umgehen könnte.

#### *8.1.1.4 Ausgabe des Eingabesystems*

**Kommando**: ECHO

**Parameter**:

- [-o|--outfile <ausgabedatei>]
- <eingabedatei>

**Synopsis**: Eingabesystem in normalisiertem Format wieder ausgeben.

Dieses Kommando dient in erster Linie zu Diagnosezwecken. Ein System wird eingelesen und in die interne Repräsentation umgewandelt. Gleich darauf wird diese Repräsentation wieder ausgegeben. Die Option –o legt dabei wie gewohnt eine Datei als Ausgabeziel fest.

Wenn spätere weitere Dateiformate unterstützt werden sollen, kann mit diesem Befehl eine Konvertierung zwischen verschiedenen Dateiformaten stattfinden.

#### *8.1.1.5 Durchführung einer (partiellen) Simulation*

**Kommando**: SIM

**Parameter**:

- [-o|--outfile <ausgabedatei>]
- [-s|--subsets]
- [-ms|--maxstates <obergrenze zustandszahl>]
- [-e|--errpath]
- [-b|--backdir]
- [-u|--summary]
- [-lc|--lambdacache]
- [-slc|--smartlambdacache]
- <eingabedatei1>
- <eingabedatei2>

**Synopsis**: Simulation(en) zwischen zwei Eingabesystemen berechnen, ggf. Fehlerpfad im DEA des langsameren Systems ausgeben.

Für beide Eingabedateien wird wie in 8.1.1.3 ein zum r-Erreichbarkeitsgraphen äquivalenter DEA aufgebaut. Die entsprechenden Optionen haben dieselbe Bedeutung wie dort und gelten jeweils für beide Systeme. Anschließend wird versucht, das erste System mit Hilfe des zweiten zu simulieren.

Mit der Option –e wird als Ausgabe im Fall einer fehlgeschlagenen Simulation der Fehlerpfad angefordert. Es findet jedoch noch kein Retracing statt, d.h. der Fehlerpfad ist als Kantenzug im DEA des simulierten Systems zu interpretieren.

Die Option –b bewirkt, dass unter Zuhilfenahme derselben Zwischenergebnisse auch eine Simulation in die Gegenrichtung erfolgt. Bei Angabe der Option –b kann durch zusätzliche Angabe von –u noch eine zusammenfassende Interpretation der beiden Einzelergebnisse in den Begriffen der *schneller-als*-Relation angefordert werden.

Die bei der Simulation verwendete Vorschrift zum Auffinden einer simulierenden Kante wird automatisch durch die Option –s gesteuert.

### *8.1.1.6 Durchführung einer (partiellen) Simulation mit Retracing*

**Kommando**: SIMR

**Parameter**:

- [-o|--outfile <ausgabedatei>]
- [-s|--subsets]
- [-b|--backdir]
- [-u|--summary]
- [-lc|--lambdacache]
- [-slc|--smartlambdacache]
- <eingabedatei1>
- <eingabedatei2>

**Synopsis**: Simulation(en) zwischen zwei Eingabesystemen berechnen, ggf. Fehlerpfad im DEA des langsameren Systems ausgeben.

Der Befehl SIMR entspricht in weiten Teilen dem oben beschriebenen SIM, nur dass sich in diesem Fall wie in 7.4 beschrieben noch das Retracing des Fehlerpfads anschließt. Der Fehlerpfad wird jeweils für die verschiedenen Stufen des Retracing ausgegeben.

Die Option –e ist dabei offensichtlich überflüssig. Auch die Option –ms ist nicht zulässig, da wegen der Verwendung von Rückwärtsabläufen beim Retracing der indizierte NDEA sowieso materialisiert werden muss und die Anzahl der Zustände so automatisch bekannt ist.

Alle anderen Optionen funktionieren wie oben beschrieben.

SIMR stellt für das momentan implementierte Setting wohl die mit Abstand häufigste Anwendung dar.

### *8.1.1.7 Durchführung einer (partiellen) Simulation mit Retracing aus den Erreichbarkeitsgraphen*

**Kommando**: SIMR-NDEA

**Parameter**:

- [-o|--outfile <ausgabedatei>]
- [-s|--subsets]

- [-b|--backdir]
- [-u|--summary]
- <eingabedatei1>
- <eingabedatei2>

**Synopsis**: Simulation(en) zwischen zwei gegebenen Erreichbarkeitsgraphen berechnen, ggf. Fehlerpfad im DEA des langsameren Systems ausgeben.

Der Befehl SIMR-NDEA leistet genau dasselbe wie SIMR, nur liest er statt zweier Petrinetze zwei Erreichbarkeitsgraphen aus den Eingabedateien ein und überspringt so die erste Verarbeitungsstufe.

**Bemerkung**: Man beachte, dass die Verweigerungsmengen in den Dateien ja nur als Bitvektoren gespeichert sind und insofern zunächst nicht gewährleistet ist, dass die Interpretation der Vektoren in verschiedenen Dateien übereinstimmt. (Konkreter könnte sich einfach die Stelle eines Bits, das für eine bestimmte Aktion anzeigt, dass sie verweigert werden kann, unterscheiden.) Dies wird jedoch dadurch verhindert, dass die Zuordnung der Stellen zu Aktionsnamen festgelegt wird, indem die Aktionsnamen lexikographisch sortiert werden. Für zwei Dateien mit übereinstimmenden Aktionsnamen werden auf diese Weise auch immer die Interpretationen der Verweigerungsmengen übereinstimmen, ohne dass diese explizit in den Dateien definiert werden müssten.

#### *8.1.1.8 Ausgabe der im CLI verfügbaren Befehle*

**Kommando**: HELP
**Parameter**: keine
**Synopsis**: Hilfeseite anzeigen.

Der Befehl HELP gibt eine Hilfeseite aus, in der die jeweils im CLI verfügbaren Befehle samt ihrer möglichen Parameter aufgelistet werden. Da die Ausgabe von HELP von FastAsy automatisch aus den jeweils implementierten Funktionen generiert wird, ist sichergestellt, dass auch in aktualisierten Versionen immer genau die jeweiligen Möglichkeiten ausgegeben werden.

#### *8.1.1.9 Zusätzliche globale Optionen*

Zusätzlich zu obigen Optionen existieren noch die folgenden globalen Optionen. Ihre Angabe ist in Kombination mit jedem der Befehle erlaubt:

**Option**: -v | --verbose
**Synopsis**: Zusätzlich Meldungen über interne Aktionen ausgeben

Die Angabe von –v bewirkt, dass ggf. zusätzliche Diagnosemeldungen mit ausgegeben werden. Der Inhalt dieser Diagnosemeldungen ist Entwicklern von FastAsy im Prinzip freigestellt und kann sich jederzeit ändern.

In der momentanen Version von FastAsy betreffen die Diagnosemeldungen in erster Linie das Retracing. Wer sich anhand eines Beispiels genauer in die Funktionsweise des Retracing einarbeiten möchte, kann dies z.B. tun, indem er bei einem SIMR zusätzlich diese Option angibt.

Die Einführung eigener Diagnoseinformation ist dank der BE++ sehr leicht zu bewerkstelligen.

**Option**: -w | --stopwatch
**Synopsis**: Ausführungszeit ausgeben

Die Option –w ist ein stark vereinfachter Ersatz für den *time*-Befehl, der auf UNIX-basierten Systemen meist vorhanden ist und für den Fall gedacht, dass auf der jeweiligen Plattform kein derartiger Befehl zur Verfügung steht. Durch die Angabe von –w wird die Ausführungszeit für einen Befehl gemessen und anschließend ausgegeben.

## 8.1.2 Format der Eingabedatei

Das von FastAsy für PTT-Netze verwendete Eingabeformat ist eine Erweiterung des von PEP verwendeten Formats für Low-Level-Netze (siehe [PEP]). Ein Minimalbeispiel sähe z.B. wie folgt aus:

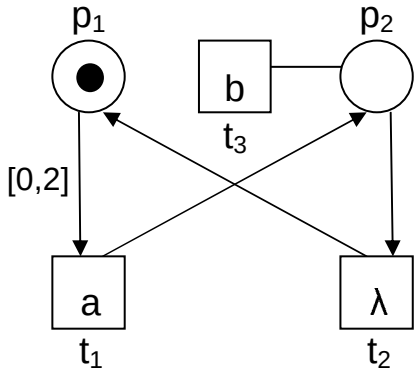



**Abbildung 28: PEP-Eingabe**

Die zugehörige Eingabedatei dazu hat folgende Form:

```
PEP
PetriBox
FORMAT_N
% Die obigen drei Zeilen sind fix.
% Kommentarzeilen beginnen mit Prozentzeichen.
% Leerzeilen sind überall erlaubt.

% „PL“ leitet den Block mit den Stellen ein
PL
1"p1"0@0M1
2"p2"0@0

% „TR“ leitet den Block mit den Transitionen ein
TR
1"t1"0@0b"a"
2"t2"0@0
3"t3"0@0b"b"
```

```
% „TP“ leitet den Block mit T→P-Kanten ein
TP
1<2
2<1

% „PT“ leitet den Block mit P→T-Kanten ein
PT
1>1I[0,2]
2>3w0
2>2

% ENDE
```

Die Zeilen für Stellen und Transitionen haben jeweils folgende Struktur:

*<id>“<name>“<x>@<y>[<tag-symbol><tag-wert>]**

Dabei ist

- *<id>* ein ganzzahliger eindeutiger Identifikator, der nur für das Eingabefile selbst Gültigkeit hat und dazu dient, in Kanten Bezug auf die Stelle oder Transition zu nehmen.
- *<name>* ein beliebiger symbolischer Name. FastAsy verlangt technisch gesehen keine Eindeutigkeit, im Interesse einer sinnvollen Ausgabe sollte diese aber offensichtlich dennoch gegeben sein.
- *<x>* und *<y>* sind grafische Koordinaten und für FastAsy irrelevant. Ihr Vorhandensein ist jedoch im PEP-Format zwingend. Sie können einfach immer durch *0@0* angegeben werden.

FastAsy kann alle *tags* lesen, die PEP schreibt, die allermeisten sind jedoch wie die Koordinaten irrelevant und werden einfach ignoriert.

Relevante *tags* sind:

- *b“<aktion>“*, wodurch das Aktionssymbol für eine Transition definiert wird. Interne Transitionen können durch Weglassen des *b-tags* oder durch Angabe von *b““*, *b“@“* oder *b“lambda“* angegeben werden.
- *M<markierung>*, wodurch die Anfangsmarkierung einer Stelle angegeben wird. Da PTT-Netze *sicher* sind, wird hier nur zwischen 0 (unmarkiert) und >0 (markiert) als Wert unterschieden. Das *M-tag* darf nicht mit dem *m-tag* verwechselt werden, letzteres ist nur für PEP wichtig und enthält die *aktuelle* Markierung der Stelle. Ist *M* nicht angegeben, so wird die Stelle als unmarkiert angenommen.

Die Zeilen für TP-Kanten haben folgende Struktur:

*<T-id>**<**<P-id>*

Für TP-Kanten existieren keine *tags*, *T-id* und *P-id* sind die Identifikatoren der jeweiligen Transition bzw. Stelle. Beide müssen weiter oben in der Datei bereits definiert sein.

PT-Kanten hingegen können *tags* besitzen, sie werden wie folgt angegeben:

*<P-id>**>**<T-id>[<tag-symbol><tag-wert>]**

Erlaubte *tags* sind dabei:

- *w<Gewicht>* gibt eigentlich in PEP ein Kantengewicht für die Kante an. In FastAsy hingegen ist lediglich die Angabe *w0* von Bedeutung, und sie klassifiziert die Kante als Lesekante. Man beachte, dass in diesem Fall nicht auch noch eine TP-Kante von *T-id* nach *P-id* vorhanden sein darf.
  Lesekanten werden in erster Linie deswegen mit *w0* notiert, weil auf diese Art das interaktive Token-Game in PEP weiterhin benutzt werden kann, die in PEP aktivierten Transitionen sind so lediglich eine Obermenge tatsächlich aktivierter, die Effekte beim Schalten einer Transitionen entsprechen sich.
- *I[<lb>,<ub>]* gibt das Zeitintervall für die Kante an, *lb* und *ub* müssen dabei ganzzahlige Werte mit $ub \geq lb \geq 0$ sein. Der *I-tag* ist eine FastAsy-eigene Erweiterung und kann von PEP nicht verarbeitet werden. Fehlt der *tag*, wird *[0,1]* angenommen, so dass die Semantik von reinen PEP-Netzen der Intuition entspricht. Man bemerke, dass sich *I-* und *w-tag* kombinieren lassen, um auch Lesekanten mit Zeitintervallen zu versehen.

## *8.2 Erste Anwendungsfälle*

### 8.2.1 Erstellung eines Erreichbarkeitsgraphen

Wir möchten für das Netz aus 8.1.2 den r-Erreichbarkeitsgraphen generieren und graphisch darstellen. Dabei sollen nur maximale Verweigerungsmengen berücksichtigt werden. Die obige Datei habe den Namen *Bsp_612.ll_net*.

Mit der Eingabe

```
> FastAsy ndea -f dot –o Bsp_612.dt Bsp_612.ll_net
```

erstellt FastAsy den r-Erreichbarkeitsgraphen und speichert ihn in einem für Graphviz bestimmten Format in der Datei *Bsp_612.dt*. Das grafische Layout erfolgt nun mit folgendem Befehl:

```
> dot –Tjpg –oBsp_612.jpg Bsp_612.dt*
```

Als Ausgabe erhalten wir:

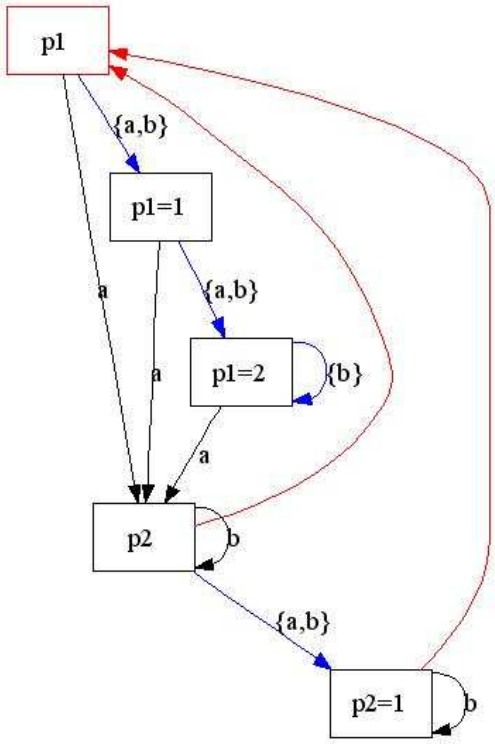



**Abbildung 29: Visualisierung mit dot**

Startzustand ist der rot gekennzeichnete Knoten p1. Die beiden unbeschrifteten roten Kanten stellen dabei λ-Übergänge dar, Kanten mit Verweigerungsmengen werden der Übersichtlichkeit halber blau dargestellt. In den Zuständen werden jeweils die markierten Stellen angegeben. Ist dabei der Uhrenwert größer als 0, d.h. ist bereits Zeit vergangen, seit die Marke auf der Stelle liegt, so wird der Uhrenwert mit aufgeführt.

* Im Gegensatz zu FastAsy verlangt dot, dass kein Leerraum zwischen –o und Dateiname steht.

### 8.2.2 Vergleich zweier Netze

Um ein zweites Netz zu Vergleichszwecken zu haben, führen wir auf dem Netz aus Abbildung 28 eine *Elongation* der Kante ($p_2$,$t_2$) durch und speichern die Datei als *Bsp_622.ll_net* ab:

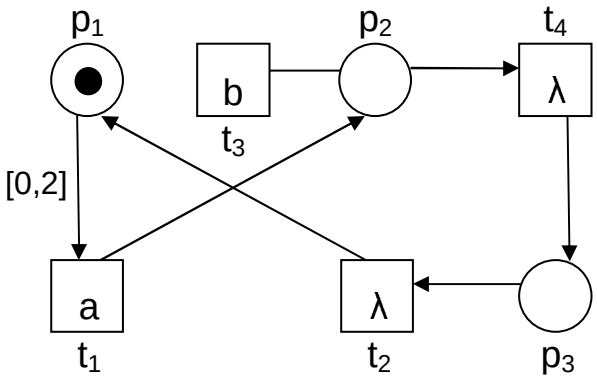



**Abbildung 30: Elongation einer Kante**

Einen Vergleich in beide Richtungen einschließlich *Retracing* führen wir nun durch mit:

```
> FastAsy simr –b –u Bsp_612.ll_net Bsp_622.ll_net
```

Wir erhalten folgende Ausgabe:

```
Simulating Bsp_612.ll_net using Bsp_622.ll_net:
Simulation successful.
```

Wie zu erwarten war, kann *Bsp_622* also sämtliches Verhalten von *Bsp_612* nachvollziehen. Wir sehen sofort, dass dies beispielsweise dadurch geschehen kann, dass man immer sofort nach dem Schalten von $t_4$ auch $t_2$ schaltet.

```
Simulating Bsp_622.ll_net using Bsp_612.ll_net:
Simulation failed.
```

Im Gegensatz dazu hat *Bsp_622* also zusätzliches (langsames) Verhalten, welches nicht Teil des Verhaltens von *Bsp_612* ist. FastAsy baut nun den Fehlerpfad auf und führt das *Retracing* durch:

```
Now building error path.
Picking state 5 as start for retracing.
0
-- (ACT:"a","t1") --> 1
-- (REF(2):[11],*) --> 4
-- (REF(2):[11],*) --> 3
-- (REF(2):[11],*) --> 2
-- (REF(2):[11],*) --> 5
```

Das Ausgabeformat der Übergänge entspricht an dieser Stelle demjenigen, das auch zur Serialisierung verwendet wird und bedarf deswegen vielleicht einer kurzen Erklärung: Vor dem Doppelpunkt steht zunächst die Typisierung, dabei bedeutet ACT eine Aktion und REF(n) eine Verweigerungsmenge aus einer n Aktionen großen Grundmenge. Im Falle von

ACT folgt nach dem Doppelpunkt zunächst der Name der Aktion und dann der Name der Transition. Im Falle von REF folgt die Verweigerungsmenge. (In dieser Stufe sind die Verweigerungsmengen noch als Bitvektoren notiert, *[11]* ist als *{a,b}* zu interpretieren.)

Der folgende * zeigt lediglich an, dass dem Übergang keine Annotation mitgegeben wurde und ist indes nur für die Entwickler von FastAsy interessant. (Technisch gesehen ist z.B. genau die Angabe der Transition in ACT eine solche Annotation. Näheres dazu wird in 9.4.2.3 und 9.4.3.1.6 besprochen.)

Nach dieser ersten Stufe haben wir den Fehlerpfad im buchstabierenden und zustandsindizierten NDEA vorliegen. FastAsy informiert uns auch, dass es aus dem letzten Multizustand des Potenzautomaten den Zustand 5 ausgewählt hat, um von dort aus rückwärts den Fehlerpfad aufzubauen. (Die Ausgabe des Zwischenergebnisses erfolgt indes trotzdem in Vorwärtsrichtung, um die Lesbarkeit zu erhöhen.)

Trotzdem sehen wir bis hierher nur, dass das langsame Verhalten darin besteht, mit $t_1$ ein *a* zu produzieren und dann viermal zu warten.

```
Reconstructing state information:
(0,-1,-1)
-- (ACT:"a","t1") --> (-1,0,-1)
-- (REF(2):[11],*) --> (-1,1,-1)
-- (REF(2):[11],*) --> (-1,-1,1)
-- (REF(2):[11],*) --> (1,-1,-1)
-- (REF(2):[11],*) --> (2,-1,-1)
```

In obigem Zwischenergebnis wurden die Zustandsnummern bereits wieder gegen Beschreibungen des Systemzustands ausgetauscht. Die Tupel sind als Vektor der Uhrenwerte über den Stellen zu lesen. Dabei bedeutet der Wert *-1* das Fehlen einer Marke.

```
Reconstructing internal transitions:
{p1}
-- "a"["t1"] -->{p2}
-- {a,b} -->{p2=1}
-- LAMBDA["t4"] -->{p3}
-- {a,b} -->{p3=1}
-- LAMBDA["t2"] -->{p1}
-- {a,b} -->{p1=1}
-- {a,b} -->{p1=2}
```

Nach dem letzten *Retracing*-Schritt wurden nun auch die internen Übergänge wieder eingefügt und um die Information ergänzt, welche Transition für einen Übergang verantwortlich war. Wir sehen, dass wir in *Bsp_622* zwischen dem Schalten von $t_4$ und $t_2$ Zeit vergehen lassen können, um die Simulation durch *Bsp_612* zum Scheitern zu bringen. Man beachte, dass der Fehler aber erst verschleppt auftritt. *Bsp_612* kann nämlich, da $t_2$ ja nicht beobachtbar ist, noch versuchen, mit $t_1$ ausreichend lange zu warten. Da die erlaubte Zeitspanne vom Markieren von $p_1$ bis zum Schalten von $t_1$ jedoch in beiden Netzen gleich ist, kann *Bsp_612* nach zwei Zeiteinheiten das Scheitern der Simulation nicht mehr verhindern.

```
*** Summary of results:
```

```
Bsp_612.ll_net is strictly faster than Bsp_622.ll_net.
```

Da wir die Option –u angegeben haben, interpretiert FastAsy das Ergebnis noch für uns: Weil in die eine Richtung eine Inklusion vorliegt und in die andere nicht, ist die Inklusion also echt.

```
no errors detected
```

Diese letzte Meldung bezieht sich nun nicht mehr auf die untersuchten Systeme, sondern stellt lediglich eine Bestätigung des in 6.2.2 erwähnten *Program Execution Monitor* dar, dass der Programmablauf von FastAsy damit beendet ist und keine unerwarteten Laufzeit-Zustände aufgetreten sind.

# Kapitel 9 Aspekte der Umsetzung

## *9.1 Einflüsse moderner SWT*

Bevor wir nun in den folgenden Kapiteln auf die Umsetzung der in Kapitel 7 beschriebenen Verfahren an sich eingehen, möchten wir zunächst noch einige Konzepte aus der Softwaretechnik erwähnen, die maßgeblichen Einfluss auf die Vorgehensweise bei der Entwicklung von FastAsy hatten. Dabei wurden in 6.3 bereits die Werkzeuge für Konfigurationsmanagement bzw. Quelltextversionierung sowie für maschinenunterstützte Dokumentation vorgestellt, so dass wir diese nicht noch einmal aufführen werden.

### 9.1.1 Entwurfsmuster

Seit Entwurfsmuster in der Softwaretechnik mit dem Erscheinen von [GoF95] im Jahr 1995 erstmals einem breiten Publikum vorgestellt wurden, haben sie rasend schnell Verbreitung gefunden, allerdings nicht ohne dass sich dabei innerhalb mancher Kreise die Auffassung des Begriffs auf subtile Art gewandelt hätte.

Entwurfsmuster, wie wir sie in unserer Arbeit verstehen und wie sie auch in oben genanntem Werk eingeführt werden, haben nicht den Charakter von Musterimplementierungen oder gar allgemein verwendbarer Utility-Klassen. Der ursprüngliche Gedanke hinter den Entwurfsmustern zielt auf eine abstraktere Ebene: Seit den Anfangszeiten der Softwaretechnik wurde immer wieder offenbar, dass eine Ausbildung von Programmierern, die sich auf bloße Fakten über Systeme, Sprachen und Frameworks beschränkt, nicht ausreichend ist. Es schien, als würde erst eine nicht näher greifbare Form von langjähriger Erfahrung gute Programmierer ausmachen, und diese wiederum waren nicht in der Lage, diese Erfahrungen zu konkretisieren und so deren Weitergabe möglich zu machen. Mit den Entwurfsmustern wird nun genau eine Form vorgeschlagen, in der archetypische Probleme und deren Lösungen beschrieben werden können. Beschrieben werden in der Regel nicht nur Einsatzzweck, Motivation und Lösung, sondern auch Konsequenzen im Vergleich zu benachbarten Entwurfsmustern, Einflüsse der Lösung auf den Kontext, auf die Wartbarkeit und auf den Aufwand. Außerdem sollen Entwurfsmusterkataloge durch einheitliche Benennungen ein Vokabular schaffen, das es Entwicklern ermöglicht, in abstrakterer und dennoch präziser Form über Entwürfe zu sprechen.

Seit dem Auftauchen von Entwurfsmustern in der Softwaretechnik wurde deren Prinzip auch mit den Architekturmustern auf höhere und mit den Idiomen (einer Art „Implementierungsmuster“) auf niedrigere Abstraktionsstufen ausgeweitet. Das Verdienst der oben erwähnten ursprünglichen Arbeit über Entwurfsmuster war neben der Einführung des Prinzips an sich auch schon ein erster Katalog an Mustern, der sich im Nachhinein als erstaunlich weitblickend erwiesen hat.

Wir werden später sowohl bei der Diskussion der Architektur wie auch der Umsetzung der einzelnen Komponenten immer wieder dort auf einzelne Muster zurückkommen, wo sie Anwendung gefunden oder Anregungen gegeben haben.

### 9.1.2 Unit Testing

Will man die Funktionstüchtigkeit einer Applikation sicherstellen, so besteht die naive Herangehensweise meist darin, beispielhafte Testfälle per Hand durchzuspielen und auf das gewünschte Ergebnis hin zu prüfen. Stößt man dabei auf einen Fehler, so wird man, um die Ursache des Fehlers zu finden, zusätzlichen Code für das Mitschreiben von Zwischenergebnissen einfügen oder aber den Programmablauf mit dem Debugger verfolgen müssen. Sollen Teile einer Bibliothek oder einer unfertigen Applikation getestet werden, so muss eventuell noch geeigneter Code zum Aufruf des zu testenden Teils samt Bereitstellung des Kontexts erstellt werden. Nachdem alle Fehler beseitigt scheinen, wird solcher Code meist wieder entfernt, und mit der nächsten durch Änderung oder Erweiterung ausgelösten Release-Iteration beginnt man mit dem Testen wieder von vorne. Zusätzlich entsteht die Komplikation, dass sogenannte Regressionsfehler (d.h. solche, in denen Änderungen die korrekte Funktion bereits vorhandener Programmteile zerstören) oft nur schwer gefunden werden, weil die Ursachen in Abschnitten liegen, die im Bewusstsein des Fehlersuchenden bereits als „verifiziert“ abgehakt sind.

Unit Testing beschreibt nun eine systematischere Herangehensweise:

- Tests werden vollständig automatisiert, d.h. es erfolgt keine Eingabe durch den Benutzer und keine Ausgabe, die zu überprüfen wäre. Ein Test ist ein eigenständiges Programmstück, welches zu testenden Code unter kontrollierten Bedingungen aufruft und seine Rückgabe auswertet. Die einzige Ausgabe eines Testfalls ist die eines booleschen Wertes, der angibt, ob der Test bestanden wurde.
- Testfälle bleiben, wenn sie einmal geschrieben sind, so lange erhalten, wie sich die Spezifikation der zu testenden Unit nicht ändert. Beim Durchlauf der sogenannten *test suite* werden jedes Mal *alle* Testfälle durchlaufen. Dieses deswegen auch *Regressionstesten* genannte Verfahren stellt sicher, dass Änderungen nicht unvorhergesehene Auswirkungen auf entfernte Stellen im Code haben.
- Kleine Teile mit weitgehend abgeschlossener Funktionalität („Units“) werden einzeln getestet. Im Falle der objektorientierten Programmierung wird es sich bei den Units meist um Klassen handeln. Der Test einer Unit teilt sich im Normalfall weiter in einzelne Testfälle auf.
- Diese Testfälle sind klein genug, dass im Falle eines Fehlschlags die Fehlerursache ausreichend genau lokalisiert werden kann. Dazu sollte auch gewährleistet sein, dass ein Testfall nur durch einen Fehler in der jeweils getesteten Unit fehlschlägt.
  Um aber trotzdem Units testen zu können, die zum Arbeiten Kontext aus anderen Units benötigen, existieren mehrere Möglichkeiten: Zum einen kann man die Tests hierarchisch anlegen, so dass sich Tests nur auf weiter unten liegende, bereits getestete Units stützen. Bei Fehlschlägen werden dann immer zuerst die Module korrigiert, die sich ausschließlich auf korrekte Units abstützen. Eine zweite Möglichkeit stellt es dar, zu Testzwecken bestimmte „Seiteneinstiege“ in den ursprünglichen Units einzubauen, die auch mit weniger Kontext den Zustand der Unit beeinflussen oder beobachten können. Eine stark verfeinerte Variante dieser Technik stellt es schließlich dar, mit sogenannten *mock objects* zu arbeiten, d.h. Objekte des Kontexts gegen solche (typischerweise von einer gemeinsamen Wurzelklasse abgeleiteten) auszutauschen, die komplexen Kontext mit einfachem Verhalten vortäuschen. Ein Beispiel wäre hier

eine *mock*-Variante eines Datenbank-Interface, unter der gar keine Datenbank liegt (für welche es nämlich sehr aufwändig wäre, sie jedes Mal für den Test in einen kontrollierten Zustand zu versetzen), sondern die auf Anfragen aus der zu testenden Unit immer nur eine statische, für den Test benötigte Antwort gibt oder die ihrerseits beispielsweise den Text der Anfrage protokolliert und dem Testfall zur Verfügung stellt. In FastAsy wären solche *mock objects* beispielsweise Instanzen der Klassen *MutableAutomaton* (siehe9.4.2.2) oder *TestRetracer* (siehe 9.6.2.7).

- Unit Testing an sich stellt auf das Gesamtsystem bezogen natürlich ein White-Box-Testing dar, da es einzelne Teile des Codes testet. Bezogen auf die einzelnen Units wird hingegen nicht spezifiziert, wie intrusiv die Tests sein dürfen (siehe auch die Diskussion zum vorigen Punkt). Es erscheint aber sinnvoll, Units so weit als möglich als geschlossene Systeme zu betrachten, da auf diese Art die Implementierung ausgetauscht werden kann, ohne dass die Testfälle verändert werden müssten. Gerade im Zusammenspiel mit *Refactoring* (siehe unten) ist man darauf angewiesen, nicht durch die Testfälle bereits Interna der Units unnötig zu fixieren.
- Unit Testing ist deterministisches Testen, d.h. ohne Änderung des Codes liefern aufeinander folgende Durchläufe der *test suite* immer wieder dasselbe Ergebnis. Der Grund ist der, dass man nach einer Änderung des Codes sofort sehen möchte, ob alte Fehler beseitigt oder neue aufgetreten sind (z.B. wenn die *policy* innerhalb eines Entwicklerteams festlegt, dass nur Code, der die *test suite* erfolgreich durchlaufen hat, in die Versionskontrolle eingechecked werden darf).

Neben der offensichtlichen Funktion, die Korrektheit des Codes sicherzustellen, ergeben sich noch andere Vorteile aus dem konsequenten Einsatz von Unit Testing: Klar formulierte Testfälle können zunächst als Teil der Dokumentation einer Unit angesehen werden, da sie musterhaft das Verhalten der Unit spezifizieren. Es ist oft sehr intuitiv, sich an Hand der Testfälle in die Benutzung einer Unit einarbeiten, da an der gleichen Stelle sowohl Beispiele für die Verwendung als auch das zu erwartende Ergebnis dargestellt sind. Ein nächster Schritt kann sogar so weit führen, dass man noch vor der Entwicklung der Unit selbst die Testfälle erstellt und diese als halbformale Spezifikation betrachtet. (Halbformal deswegen, weil Testfälle in der Regel nicht erschöpfend sein werden. Ein Klasse, die ganze Zahlen modelliert, wird man nicht so genau testen wollen, dass man beispielsweise die Addition für *alle ganzen Zahlen* im Wertebereich testet.)

Als Faustregel für den Aufwand kann dienen, dass die Implementierung elementarer Testfälle für eine Unit in etwa so viel Zeit in Anspruch nimmt wie die Implementierung der Unit selbst. Gerade in komplexeren Projekten wird sich diese Zeit jedoch bei der Fehlersuche rasch amortisieren.

Nach dem vorher gesagten dürfte jetzt auch bereits offensichtlich sein, warum auch Unit Testing kein Allheilmittel ist. Das zentrale Problem lautet natürlich „Qui custodiet ipsos custodes?“, d.h. auch den Testfällen kann man nur bedingt vertrauen. Die Wahrscheinlichkeit ist zwar hoch, dass ein fehlerhafter Testfall unabhängig von der Korrektheit der Unit tatsächlich auch einen Fehler erzeugt, aber dies ist bei weitem nicht immer der Fall. Außerdem sind Testfälle, da sie ja zu einem ausführbaren Programm übersetzt werden, von ihrer Natur her nicht parametrisch und somit nicht erschöpfend. (Es existieren Ansätze, durch geschickte Anlage der Testfälle wenigstens eine möglichst hohe *code coverage* herzustellen,

d.h. möglichst alle Teile des Quellcodes der zu testenden Unit anzuspringen, aber dies führt uns sofort zu einem rigorosen White-Box-Testing. Eine solche *code coverage* könnte natürlich auch durch Tools geprüft werden, etwa einen Präcompiler, der entsprechenden Protokollcode in bedingte Strukturen einfügt. In FastAsy haben wir jedoch keine solchen Ansätze verwendet.)

Zuletzt sei noch erwähnt, dass in Zusammenhang mit der Verwendung von C++-Templates dem Unit Testing beiläufig noch eine weitere Rolle zukommt: Durch den Code der Testfälle wird schon bei der Entwicklung von Template-Klassen sichergestellt, dass umfassende Instanzierungen der Templates stattfinden. Auf diese Art hat man allein durch die Übersetzbarkeit der *test suite* schon eine wichtige Prüfung der statischen Korrektheit von Template-basiertem Code. Findet nämlich keine Instanzierung statt, so ist es C++-Compilern weitgehend freigestellt, inwieweit sie die Template-Definitionen selbst überhaupt auf Korrektheit überprüfen.

### 9.1.3 Refactoring

Martin Fowler definiert in seinem gleichnamigen Buch [MF00] den Begriff *Refactoring* als den „Prozess, ein Softwaresystem so zu verändern, dass das externe Verhalten nicht geändert wird, der Code aber eine bessere interne Struktur erhält“. Charakteristisch für Refactoring sind weiterhin folgende Merkmale:

- Idealerweise findet die Umwandlung in kleinen (meist atomaren), festgelegten und formal definierten Schritten dar, für die bekannt ist, dass sie die operative Semantik eines Programms nicht ändern.
- Werden diese Refaktorisierungsschritte von einem Menschen durchgeführt (leider unterstützen selbst fortschrittliche Werkzeuge immer noch nicht annähernd den Umfang an Refaktorisierungen, die man bei der Arbeit benötigt), so ist natürlich nicht auszuschließen, dass Fehler bei der Anwendung eines an sich korrekten Schritts vorkommen.

  Eine Forderung des Refactoring ist deshalb eigentlich, nach jeder einzelnen Umwandlung das Programm zu übersetzen und die *test suite* (s. oben) zu durchlaufen. In der Praxis wird man schon auf Grund zu hoher Compilerzeiten erst nach mehreren Umwandlungen übersetzen und testen, typischerweise über Nacht (*nightly build*).
- Neben der beschriebenen Verflechtung zum Unit Testing existiert auch eine Verbindung zu den Entwurfsmustern, denn es wird oft das Ziel einer Reihe von Refaktorisierungen sein, eine unstrukturierte Lösung derart umzuwandeln, dass sie sich durch einzelne Entwurfsmuster oder Kollaborationen von solchen beschreiben lässt.

Enthusiasten der Refactoring-Community, darunter auch Martin Fowler selbst, deuten immer wieder an, Refactoring könne die Notwendigkeit, vorab das Design eines geplanten Softwaresystems festzulegen, vermindern. Inwieweit dies den Tatsachen entspricht, ist Kern lang anhaltender Diskussionen und hängt sicher auch maßgeblich von der Komplexität des geplanten Systems ab. Der Autor ist eher geneigt, Refactoring als eine Art „doppelten Boden“ anzusehen: Sollte sich während späterer Entwicklungsphasen das ursprüngliche Design aus welchen Gründen auch immer (geänderte Anforderungen dürften die weitaus häufigste

Ursache darstellen) als unzureichend erweisen, so stellt Refactoring eine geeignete Herangehensweise dar, die Struktur des Systems anzupassen.

### 9.1.4 Code Robustness durch Laufzeit-Zusicherungen

Unter *code robustness* versteht man allgemein die Eigenschaft von Programmcode, sich nicht blind darauf zu verlassen, dass der eigene oder kolaborierender Code einwandfrei funktioniert. Insbesondere sollten Situationen vermieden werden, in denen durch falsches Verhalten anderer Programmteile das eigene Verhalten undefiniert wird. In den Code eingebettete Zusicherungen stellen eine Möglichkeit dar, zur Laufzeit solche Situationen zu erkennen. Auch Nachbedingungen von komplexeren Verarbeitungsschritten können so geprüft werden. Wird eine Zusicherung verletzt, so muss man davon ausgehen, dass ein Fehler im Programmcode vorliegt, so dass es meist keinen sinnvollen Weg gibt, die Programmausführung im Sinne des Benutzers zu Ende zu bringen. Ausnahmen hiervon stellen freilich menügesteuerte oder stark modularisierte Softwaresysteme dar, in denen die Ausführung eines einzelnen Befehls bzw. eine Aktion eines einzelnen Moduls isoliert betrachtet und ggf. verworfen werden kann, so dass man einen Punkt für ein kontrolliertes „Wiederaufsetzen“ hat.

Die gängige Technik, Zusicherungen im Code unterzubringen, besteht darin, zentral einen Makrobefehl (in BE++ *assertion*, in der STL *assert*) zu definieren, der in Abhängigkeit von einer Makrokonstante (in BE++ wie in der STL *NDEBUG*) verschieden expandiert:

- Ist *NDEBUG* gesetzt (d.h. handelt es sich beim zu übersetzenden Code um Produktivcode, der bereits als vollständig stabil gilt), so wird zu *NOP* expandiert. Auf diese Art fällt keine Rechenzeit für Prüfungen an, die nicht mehr als notwendig gelten.
- Ist *NDEBUG* hingegen nicht gesetzt (bei einem ständig in der Weiterentwicklung befindlichen Programm ohne definierte Release-Zyklen wie FastAsy der Normalfall), so wertet *assertion* seinen einzigen Parameter, einen booleschen Wert, aus. Ergibt dieser *falsch*, so wird eine Ausnahme (in BE++ *assertion_failed*) geworfen.

In FastAsy wird die BE++-Variante verwendet, da diese ggf. automatisch in der Lage ist, bei Verwendung der Borland-VCL statt C++-Ausnahmen die davon etwas verschiedenen VCL-Ausnahmen zu benutzen. In Standard-C++ sind *assertion* und *assert* gleichwertig.

Die BE++ erweitert obiges Konzept noch etwas und stellt mit *DEBUGONLY* ein weiteres Makro zu Verfügung, das längere Codeblöcke (die als Parameter angegeben werden) nur bei nicht gesetztem *NDEBUG* übersetzt und ansonsten ebenfalls zu *NOP* expandiert. So können auch aufwändigere Prüfungen bedingt durchgeführt werden.

## *9.2 Architektur*

Wir wollen uns zunächst einen Überblick über die Architektur von FastAsy verschaffen. Dabei beschreiben wir zuerst die vertikale Aufteilung in Schichten, bevor wir genauer auf die Kollaborationsmechanismen innerhalb der mittleren Schichten eingehen.

### 9.2.1 Schichten

Wir haben bereits angedeutet, dass FastAsy in verschiedene Schichten aufgeteilt ist (siehe das Muster *layers* in [FB00]). Die Aufteilung von Softwaresystemen (und allgemeiner sogar von eingebetteten Systemen) in Schichten ist natürlich schon lange vor der Verbreitung

von Architekturmuster-Katalogen vorgeschlagen worden (als kanonisches Beispiel ist hier das OSI-Schichtenmodell anzuführen) und stellt für viele Systeme wohl eine sehr natürliche Entwurfsentscheidung dar: Eine Schicht stellt jeweils weiter oben stehenden Schichten bestimmte Dienste zur Verfügung und bedient sich bei der Abarbeitung angeforderter Dienste der unter ihr stehenden Schichten. Unterschieden wird weiterhin in strikte Schichteneinteilung und sogenanntes *relaxed layering*. Bei letzterem ist der Durchgriff durch Zwischenschichten erlaubt, bei ersterem nicht.

Im Allgemeinen bestehen die verschiedenen Schichten selbst wiederum aus einzelnen Bestandteilen, die sich innerhalb der Schichten gegenseitig benutzen dürfen. Wir verwenden im folgenden auch den allgemeinen Begriff *Komponenten*, möchten ihn jedoch nicht in dem sehr speziellen Sinne komponentenbasierter Systeme verstanden haben. Insbesondere zielen wir mit dem Begriff nicht auf ein eigenständiges *Deployment* ab, was sich jedoch in diesem Kontext von selbst verstehen sollte. Konzeptionelle Zusammenfassungen solcher Komponenten bezeichnen wir auch als *Package*, ohne jedoch bereits eine Strukturierung des Quelltextes zu implizieren.

Das nachstehende Diagramm gibt zunächst einen Überblick über Schichten und Komponenten in FastAsy:

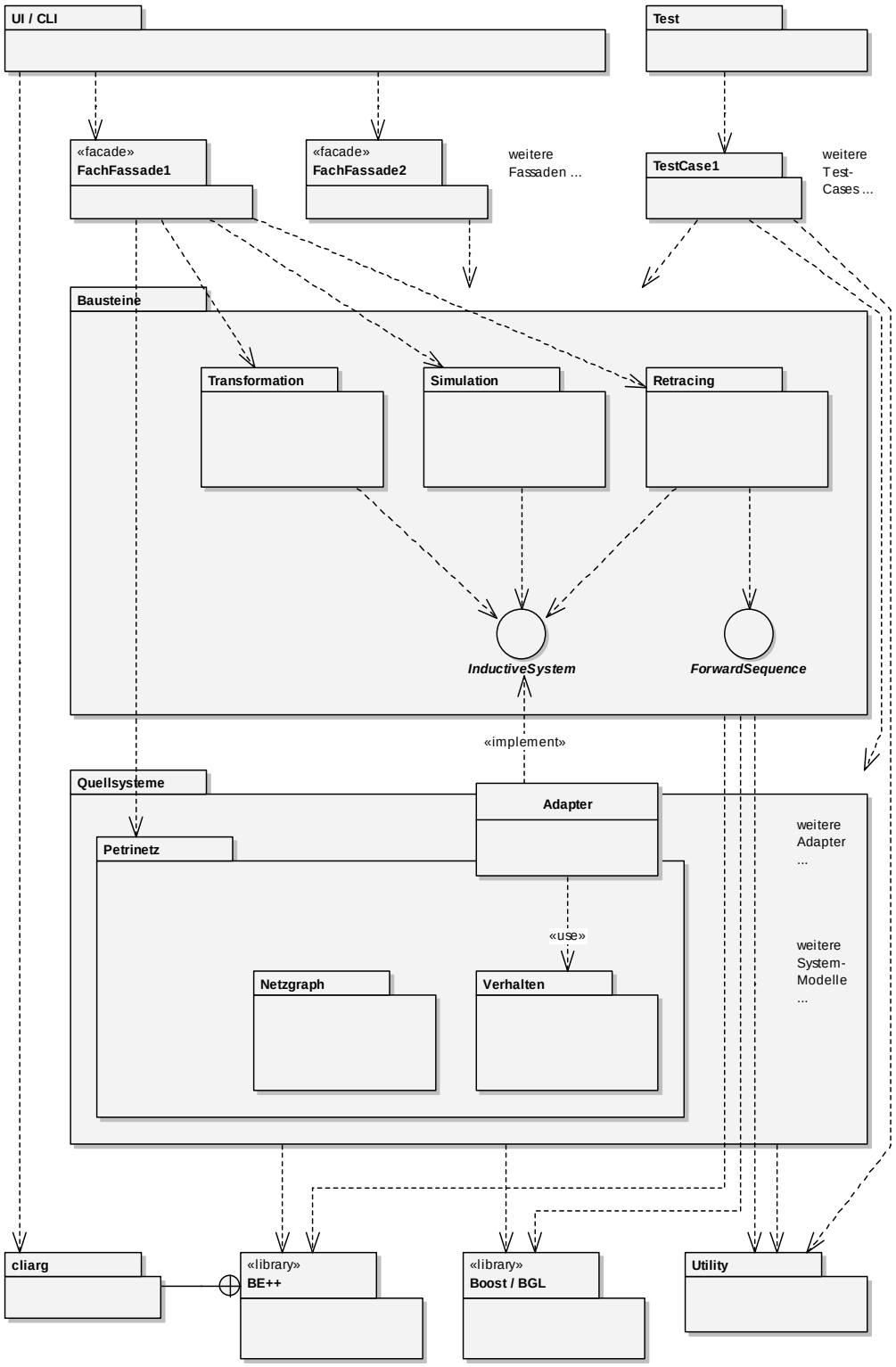



**Abbildung 31: Schichten und Komponenten in FastAsy**

Die Aufteilung ist dabei wie folgt:

### *9.2.1.1 UI / CLI*

An oberster Stelle regelt ein Benutzerinterface die Kommunikation mit dem Benutzer. In der momentanen Version ist dies wie bereits erwähnt ein Kommandozeileninterface. Im Durchgriff benutzt dieses noch die Dienste der *cliarg*-Komponente (siehe 9.7.3)der BE++-Bibliothek, um die Kommandozeile zu evaluieren. Genausogut könnte statt des CLI auch eine

GUI stehen, welche dann typischerweise Bibliotheken wie VCL, QT oder wxWidgets benutzen würde, um Befehle des Benutzers entgegenzunehmen und Ergebnisse darzustellen. Entscheidend dabei ist, dass die UI-Schicht solche Befehle lediglich entgegennimmt, aber nicht dafür verantwortlich ist, das Zusammenspiel der entsprechenden Komponenten aus der Schicht „Bausteine“ zu kontrollieren. Stattdessen delegiert die UI-Schicht die Ausführung des entsprechenden Befehls an die geeignete Fachfassade. (Selbst wenn dies teilweise redundanten Code bedeuten kann, schien uns eine 1:1-Zuordnung von Befehl und Fachfassade bisher am stimmigsten. Die Architektur von FastAsy erzwingt dies jedoch nicht, eine Fachfassade könnte genauso gut für die Ausführung mehrerer Befehle verantwortlich sein.)

#### *9.2.1.2 Fachfassaden*

Den Begriff der *Fassade* verdanken wir dem Entwurfsmusterkatalog aus [GoF96], wo er folgendermaßen umrissen wird: „[Eine Fassade] bietet eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems […], welche die Benutzung eines Subsystems vereinfacht.“ Unsere Benennung „Fachfassaden“ deutet darüber hinausgehend an, dass das angebotene Interface nicht technischer Natur sein wird, sondern sich mit dem Vokabular der Problemdomäne, nämlich der Untersuchung asynchroner Systeme, befasst.

In unserem Fall ist es konkret die Aufgabe der einzelnen (untereinander völlig entkoppelten) Fachfassaden, für einen Befehl des Benutzerinterface zu wissen, welche Komponenten aus der Baustein-Schicht (und teilweise aus der Quellsystem-Schicht, siehe weiter unten) nötig sind, um die angeforderte Verarbeitung durchzuführen, und wie diese verschaltet werden müssen. Der Befehl wird im Allgemeinen natürlich noch parametrisiert sein, so dass die Fachfassade die Parameter entweder auswerten und in die Verschaltung der Komponenten einfließen lassen oder aber sie an die Komponenten selbst weiterreichen muss.

Der Wunsch, die einzelnen Fachfassaden untereinander völlig entkoppelt zu halten, bedarf womöglich noch der weiteren Erklärung. Schließlich ist es ja so, dass die meisten der Befehle in 8.1.1 keineswegs völlig unabhängige Funktionen ausführen, sondern im Gegenteil große Überschneidungen vorhanden sind. Halten wir uns aber einmal die übergreifende Entwurfsmetapher für FastAsy vor Augen: Wir haben einen Experimentierkasten vor uns, in dem wir die verschiedenen Bausteine untereinander verdrahten können. Im Gegensatz zu einem Experimentierkasten aus der wirklichen Welt hat unser virtuelles Exemplar aber noch einen Vorteil, nämlich den, dass wir uns erfolgreiche Aufbauten ohne weiteres zur späteren Verwendung merken können. Eine Fachfassade ist nun einfach eine solche eingefrorene Konfiguration des Experimentierkastens in Gestalt eines einzelnen, unabhängigen Quelltext-Files. Bezögen sich die Versuchsaufbauten nun aufeinander, verließen wir nicht nur endgültig unsere Metapher, sondern als direkte Folge davon könnten wir einzelne Aufbauten nicht mehr ändern, ohne unter Umständen abhängige Fachfassaden in ihrer Funktion zu verändern.

Indem wir nun solche Fachfassaden, die sich als sinnvoll erwiesen haben, in das CLI einbinden, stellen wir auch Benutzern, die sich über die Interna der Versuchsaufbauten keine Gedanken machen wollen, eine Frontplatte mit den wichtigsten Bedienelementen zur Verfügung, von der aus sie Abläufe nicht nur starten, sondern auch parametrisieren können. Die Frontplatte wird jedoch keine Bedienelemente umfassen können, die die Verdrahtung der Bausteine ändern – allenfalls werden sich, wo es sich anbietet, Schalter zum Einsatz

alternativer Baugruppen finden. (Wir erinnern uns, dass die Verdrahtung der Baugruppen eben durch die Typisierung der generischen Interfaces einen deutlichen statischen Aspekt besitzt. Die erwähnten Schalter können sich aber natürlich trotzdem die Polymorphie gleich typisierter Interfaces zunutze machen.)

Außerdem besitzt die konsequente Trennung der Fachfassaden noch einen technischen Hintergrund: Ändern wir an der Quelldatei einer Fachfassade etwas, so wird nur dieses eine Modul neu übersetzt werden müssen. Dies mag Lesern, welche mit der Problematik großer in C++ realisierter Softwaresysteme nicht vertraut sind, nicht bedeutsam erscheinen. Man halte sich jedoch vor Augen, dass selbst mit dem Einsatz der Technik der *include-guards*, welche das mehrfache Auswerten einer Include-Datei von einem Quelltext-Modul aus unterbindet, die Übersetzungszeiten eines C++-Programms schon ohne Templates quadratisch mit der Größe des Systems wachsen können. Eine ausführliche Behandlung dieser Thematik findet sich etwa in [JL96].

Zu Durchgriffen der Fachfassaden auf die Schicht der Quellsysteme sei noch folgendes gesagt: Da die Fachfassaden unter anderem auch dafür verantwortlich sind, dass persistent gespeicherte Spezifikationen der Quellsysteme überhaupt erst einmal in die entsprechenden Objekte geladen werden, ist es ihnen zum Zweck von E/A-Operationen auch erlaubt, sich der Quellsystem-Schicht zu bedienen. Ansonsten aber sollten sie aus dieser Schicht lediglich die zur Verfügung gestellten Adapter nutzen. Dies entkoppelt sehr effektiv die für spätere Berechnungen genutzten Baustein-Komponenten von den Quellsystemen. Dadurch wiederum wird sowohl die Entwicklung neuer Bausteine wie auch neuer Quellsystem-Modelle vereinfacht, da die Spezifika der jeweils anderen Schicht nicht zu interessieren brauchen.

#### *9.2.1.3 Test*

Diese Schicht wird (zusammen mit den Testfällen) zu einem von UI/CLI und den Fachfassaden getrenntem ausführbaren Programm übersetzt. „Test“ enthält lediglich ein Hauptprogramm, welches die verschiedenen Testfälle der Schicht darunter nacheinander aufruft, voneinander isoliert (in Bezug auf die Behandlung von e*xceptions*), und deren Ergebnisse protokolliert. Dazu bedient es sich des Unit Testing Framework der Boost-Bibliothek. In der Tat wertet auch „Test“ Kommandozeilenparameter aus, jedoch genau jene, die das Unit Testing Framework versteht (siehe [BCPL]). Wir bedienen uns dazu nicht des *cliarg*-Package der BE++, sondern überlassen diese Auswertung dem Unit Testing Framework selbst.

#### *9.2.1.4 Testfälle*

Die Packages „$\text{TestCase}_n$“ enthalten gruppiert mehrere Einzeltests, die jeweils verschiedene Bereiche der darunter liegenden Schichten auf ihre korrekte Funktion hin prüfen. Den Testfällen ist natürlich jeglicher Durchgriff auf niedere Schichten gestattet, anders wäre ein wirkliches Unit Testing nicht möglich.

#### *9.2.1.5 Bausteine*

In dieser Schicht sind Komponenten versammelt, die jeweils einzelne Schritte der in Kapitel 7 beschriebenen Verarbeitungskette realisieren. Bemerkenswert dabei ist die Tatsache, dass diese Bausteine durch einen äußerst flexiblen Mechanismus aneinandergereiht werden. Von diesem soll in 9.2.2 noch ausführlich die Rede sein.

Die Baustein-Schicht ist in sich noch horizontal unterteilt in die Aufgabenbereiche Transformation, Simulation und Retracing. Ersterer umfasst dabei alle Transformationen von induktiven Systemen, wie wir sie in 9.4 näher beschreiben. Im zweiten Aufgabenbereich finden wir die zur Durchführung von Simulationen benötigten Komponenten (siehe 9.5), und im letzten Transformationen linearer Systeme, die für die Rückgewinnung des Fehlerpfads notwendig sind (siehe9.6).

Weiter definiert die Baustein-Schicht noch die beiden wichtigsten Interfaces, nämlich *InductiveSystem* und *ForwardSequence* (ein Interface für lineare Systeme). Offenbar implementieren die Klassen im Bereich Transformation *InductiveSystem*, die im Bereich Retracing dagegen *ForwardSequence*. Der Hauptbestandteil des Bereichs Simulation, nämlich *GenericSimulation*, implementiert keines der beiden Interfaces direkt, jedoch stehen mit *SimRevErrorSystem* und *SimRevErrorSequence* zwei Adapter bereit, mit denen sich Varianten des Fehlerpfads (oder genauer dessen Umkehrung) mit den Interfaces *InductiveSystem* bzw. *ForwardSequence* ansprechen lassen.

In 9.2.2 werden wir sehen, welche Kollaborationsmechanismen FastAsy nutzt, um die verschiedenen Bausteine möglichst flexibel zu verketten.

#### *9.2.1.6 Quellsysteme*

In dieser Schicht befinden sich alle Komponenten, die dafür notwendig sind, aus der Beschreibung von Systemen ihre zeitliche Semantik zu gewinnen. Dabei ist wie bereits erwähnt diese Schicht stark gekapselt, die Ergebnisse werden den höheren Schichten einzig dadurch zur Verfügung gestellt, dass Adapter auf das *InductiveSystem*-Interface implementiert werden.

#### *9.2.1.7 Bibliotheken und Utility-Schicht*

In dieser untersten Schicht sind als Bibliotheken BE++ und Teile von Boost vertreten, außerdem existiert noch ein Utility-Package, in dem grundlegende Strukturen wie etwa *InfinityInteger* oder *IntegerIntervall* definiert sind. In dieses Package gehören auch die diversen I/O-Adapter.

### 9.2.2 Verschaltung der Bausteine

Es wurde bereits mehrfach hervorgehoben, dass die Kernbestandteile von FastAsy als Bausteine zu sehen sind, deren Anzahl jederzeit erweitert werden kann und die untereinander neu kombiniert werden können. An dieser Stelle wollen wir einen Blick darauf werfen, wie die Verschaltung dieser Bausteine erfolgt.

Machen wir uns zunächst die Ausgangssituation klar:

- Der Gesamtalgorithmus läuft in klar trennbaren Phasen ab.
- Diese Phasen sollen unabhängig sein und somit ausgetauscht, in ihrer Reihenfolge verändert und durch weitere ergänzt werden können.
- Die Zwischenergebnisse haben unterschiedliche Typisierung. Trotzdem lassen sie sich einteilen in endliche Transitionssysteme und Kantenzüge von Transitionsgraphen. (Die Simulationsrelation spielt allerdings eine Sonderrolle.)
- Wir wollen nicht jedes Zwischenergebnis materialisieren.

- Beim Retracing werden alte Zwischenergebnisse in irgendeiner Form benötigt. Dabei finden jedoch typischerweise nur wenige Zugriffe statt.

Ausgehend von diesen Anforderungen sehen wir schnell, dass traditionelle Realisierungen nicht das gewünschte leisten:

- Die Realisierung der einzelnen Phasen als eigenständige Prozeduren, die das Ergebnis ihres Vorgängers als komplexes, konstantes Objekt entgegennehmen und im Zuge ihrer eigenen Berechnungen wieder ein solches Objekt erstellen, hat genau zur Folge, dass wir eine hohe Anzahl von großen Zwischenobjekten erstellen.
- Indem mehrere Phasen in einer Prozedur kombiniert werden, lässt sich diese Anzahl verringern. Eine solche Lösung verletzt jedoch direkt die Forderung nach der Austauschbarkeit. (Wird diese Forderung fallengelassen, ist diese Lösung aber, wie alte FastAsy-Versionen gezeigt haben, praktikabel.)
- Eigenständige Prozeduren, die aber keine Zwischenergebnisse produzieren, sondern wenige globale Objekte *in situ* ändern, sind aus mehreren Gründen ebenfalls keine Alternative: Zum einen ist ein solcher Ansatz durch die unterschiedliche Typisierung der Zwischenergebnisse nur schwer umzusetzen, zum anderen ist etwa die Konstruktion des Potenzautomaten gar nicht auf eine solche Art und Weise zu bewerkstelligen. Außerdem verlieren wir die Möglichkeit, beim Retracing auf Zwischenergebnisse zuzugreifen.

Für ähnliche Situationen findet man in der Literatur nun das Architekturmuster *Pipes & Filters* (etwa in [FB00]). Als archetypisches Beispiel dafür kann tatsächlich der gleichnamige Mechanismus in den verbreiteten UNIX-Shells samt den zugehörigen Filter-Befehlen gelten, der zu Veränderungen textbasierter Datenströme herangezogen wird. Diese Lösung sieht vor, einen Datenstrom nacheinander durch die benötigten Komponenten fließen zu lassen, welche ihre Aufgabe jeweils lokal auf dem ihnen vorliegenden Ausschnitt aus dem Datenstrom durchführen. Dabei kann die Ausgabe an eine *Datensenke* bereits beginnen, wenn nur ein geringer Teil der Eingabe aus der *Datenquelle* gelesen wurde.

Leider existieren einige Punkte, die uns daran hindern, das Pipes&Filters-Muster einzusetzen:

- Wir haben es (zumindest bei den Transitionssystemen) nicht mit einem Datenstrom zu tun, sondern mit einem komplexen Objekt.
- Eine Serialisierung solcher Objekte wäre zwar ein Datenstrom, jedoch keiner, für den die lokale Bearbeitung kurzer Ausschnitte sinnvoll möglich ist.
- Das Problem unterschiedlicher Typisierung (in diesem Kontext als unterschiedlich formatierte Datenströme) würde auch hier die Austauschbarkeit der einzelnen *Filter* zunichte machen.

Indem wir das letztgenannte Problem noch für einen Moment zurückstellen, erinnern wir uns noch eines weiteren Musters, nämlich des Entwurfsmusters *Dekorierer* (etwa [Gof96]). Ein Dekoriererobjekt „erweitert ein Objekt dynamisch um Zuständigkeiten“, indem es selbst das Interface des ursprünglichen Objekts implementiert, eine Referenz auf jenes Objekt hält und Anfragen auf Delegationen an das ursprüngliche Objekt umsetzt. (Die Verwendung des Begriffs der *Reduktion* aus der Komplexitätstheorie scheint hier durchaus angebracht.)

Während dieses Muster eigentlich auf gänzlich andere Verwendungen und wesentlich leichtgewichtigere Objekte abzielt, enthält es doch eine wichtige Idee, nämlich die Delegation an ein einzelnes weiteres Objekt *mit gleicher Schnittstelle* - wodurch nämlich eine Kaskadierung solcher Dekorationen möglich wird.

Diese Idee stellt nun genau das fehlende Steinchen im Mosaik dar, das wir benötigen, um in unserem Kontext eine Pipes&Filters-ähnliche Architektur einsetzen zu können: Wir ersetzen die Datenströme durch Ketten von Anfragen. Dabei tritt an die Stelle des einen festgelegten Formats des Datenstroms ein ganzes Interface mit verschiedenen Anfragen, und so können die einzelnen Komponenten bestimmen, welchen Ausschnitt ihrer Eingabe sie zu Gesicht bekommen wollen. Es steht ihnen im Unterschied zu Pipes&Filters z.B. auch frei, mehrfach die gleichen Teile ihrer Eingabe zu betrachten.

Bevor wir aber nun im Detail die Kollaborationen induktiver und linearer Systeme betrachten, wollen wir wie versprochen noch das Problem der unterschiedlichen Typisierungen lösen. Dazu machen wir uns lediglich die Tatsache bewusst, dass das Dekorierer-Muster immer noch funktioniert, wenn das Interface durch Typen parametrisiert wird. Die verschiedenen Dekorierer werden dabei je nach Funktion gewisse Anforderungen an ihre aktuellen Typparameter stellen und damit Einschränkung in der Vertauschbarkeit verschiedener Phasen bewirken. Dies ist jedoch nicht nur natürlich, sondern darüber hinausgehend eine sehr starke Realisierung von Typsicherheit, denn der Compiler ist somit in der Lage, zur Übersetzungszeit festzustellen, ob eine bestimmte Verschaltung von Bausteinen zulässig ist. Die Typsicherheit der Zwischenergebnisse wird so auf eine Typsicherheit ganzer Bausteine hochgezogen. Über die Beziehungen der Typparameter im Interface der Dekorierer untereinander wird im Allgemeinen nichts ausgesagt. Sie werden oft nicht unabhängig voneinander sein (siehe weiter unten), insbesondere werden die Typen des vom Dekorierer bei seinem Vorgänger angeforderten Interface meist diejenigen des vom Dekorierer angebotenen Interface bedingen. Im Falle der in 7.2.3 beschriebenen Indizierung der Zustände ist jedoch beispielsweise der Knoten-Typ im angebotenen Interface fix als Ganzzahlwert gegeben. Die Potenzautomatenkonstruktion aus 7.2.4 wiederum fordert genau einen solchen Knoten-Typ im Interface ihres Vorgängers.

Im folgenden sehen wir nun zunächst exemplarisch einen solchen Systementwurf mit einer Quelle, zwei Verarbeitungsphasen und einer Senke. Das zentrale Interface *S*, welches die Verarbeitungsstufen sowohl implementieren als auch benutzen soll, ist hier in zwei Typen *Arg1* und *Arg2* parametrisch und besitzt zwei Operationen. (Die Signaturen der Operationen werden typischerweise von den Typparametern des Interface abhängen, aber wir unterdrücken diese Tatsache hier, um das Diagramm nicht unnötig zu belasten.) Wie wir sehen, hält jede Komponente genau eine Referenz auf ihren Vorgänger. Man beachte, dass durch den Typ dieser Referenz schon die Typparameter für den Vorgänger spezifiziert sind. Die Komponenten selbst sind möglicherweise wie hier in eigenen Parametern typparametrisch, und sowohl die Typparameter der Referenz wie die der zur Verfügung gestellten Template-Instanz von *S* können von ihnen abhängen. Die einzelnen Phasen werden in der Regel noch eigene, private Berechnungen durchführen müssen, um Anfragen delegieren zu können. Man beachte, dass die Operationen des Interface notwendigerweise mit dem Zugriffsspezifizierer *public* angegeben sind, denn verschieden parametrisierte (Template-)Instanzen von *S* sind natürlich unterschiedliche Typen auch in dem Sinne, dass sie nicht auf ihre *private*- oder *protected*-Operationen zugreifen dürfen.

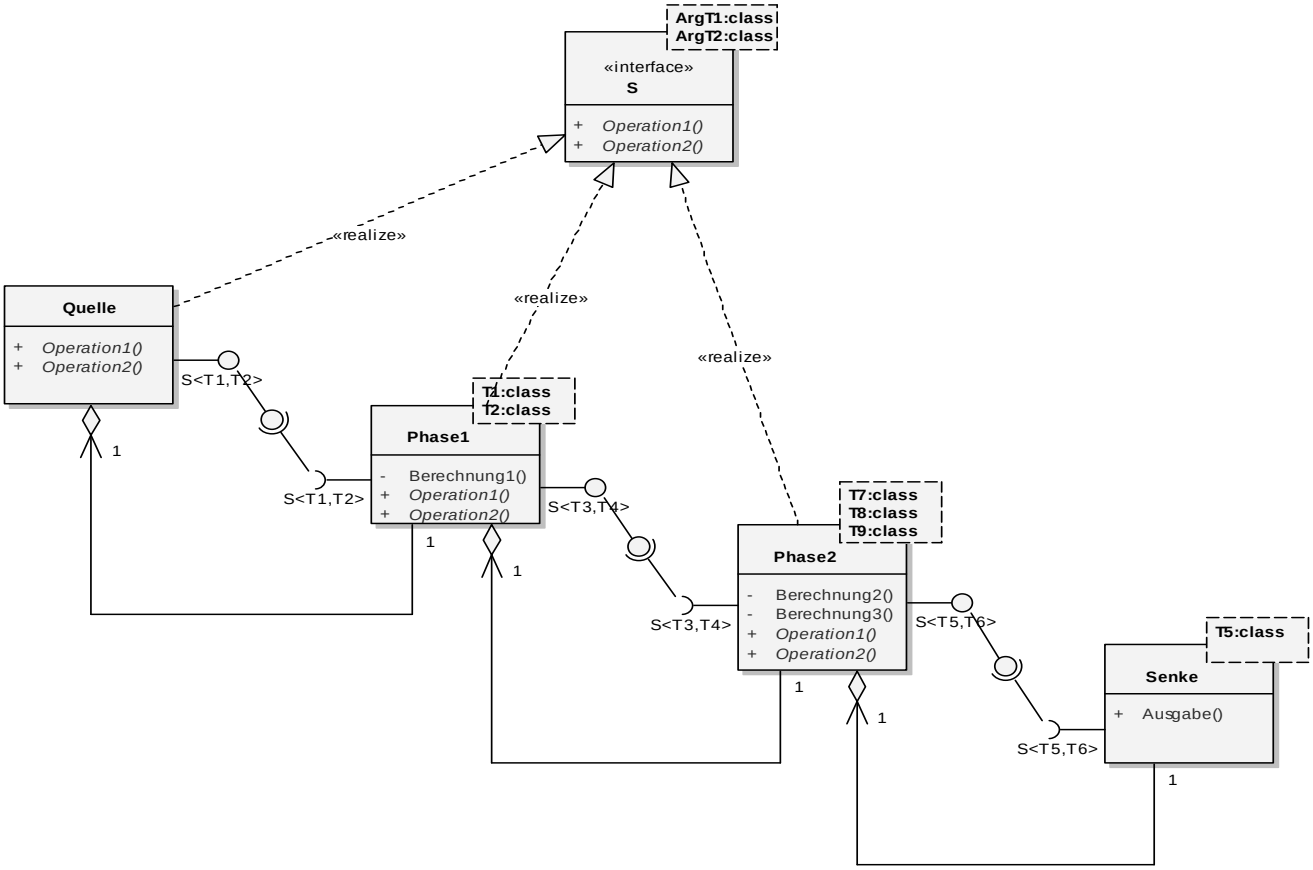



**Abbildung 32: Verschaltung von Bausteinen**

Werfen wir nun weiter noch einen Blick auf ein Sequenzdiagramm, in dem ein mögliches Zusammenspiel der obigen Komponenten angedeutet wird. Selbstverständlich hängt die genaue Struktur der Abrufe von den konkreten Berechnungen ab, die zu den Delegationsaufrufen führen, so dass wir hier nur eine willkürliche Möglichkeit herausgreifen können.

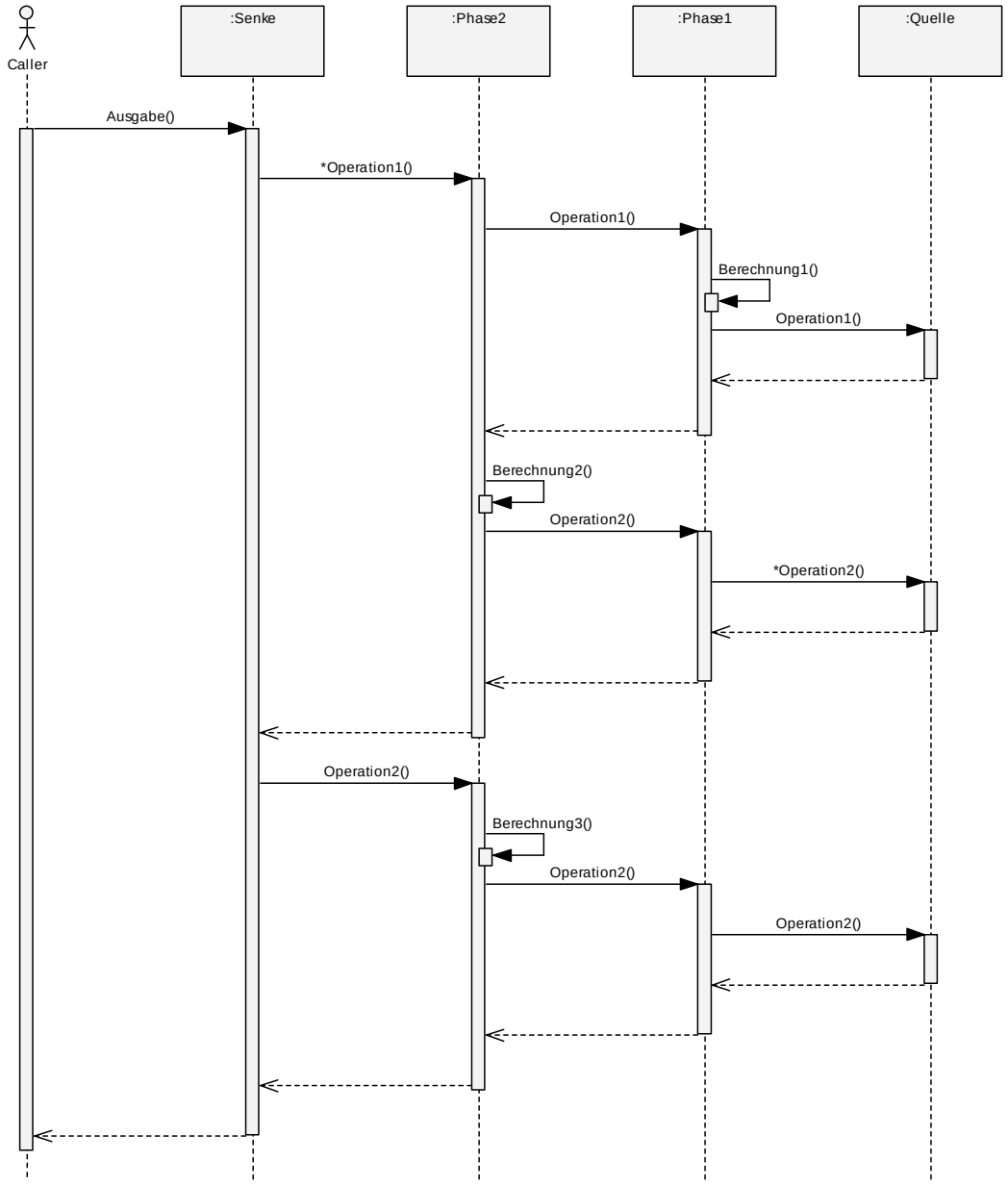



**Abbildung 33: Aufrufstruktur der Bausteine**

Der aufrufende Code stößt dabei die Verarbeitung an, indem er der Senke einen Startbefehl erteilt, hier den Befehl zu einer Ausgabe. Die Senke iteriert nun Aufrufe der ersten Operation von Phase2. Man beachte an dieser Stelle, dass *spätere* Berechnungsschritte *früher* das erste Mal die Kontrolle erhalten. Die einzelnen Phasen setzen nun Aufrufe der beiden Operationen aus *S* jeweils in weitere Aufrufe der Operationen um. Diese Aufrufe erfolgen immer direkt an den Vorgänger, aber ihre Reihenfolge, Multiplizität und Parameter dürfen beliebig sein. Insbesondere dürfen Aufrufe einer festen Operation $Operation_i$ trotzdem *beide* Operationen des Vorgängers benutzen.

Da die einzelnen Phasen ja Objekte darstellen, die selbst einen Zustand haben dürfen, ist es ihnen auch freigestellt, sich Informationen, die sie bereits von ihrem Vorgänger erhalten haben, zu merken. In einem solchen Fall kann es vorkommen, dass ein Berechnungsschritt eine Anfrage selbständig beantwortet, ohne seinen Vorgänger nochmals nach derselben Information zu fragen. Den Extremfall einer solchen Komponente, nennen wir eine Materialisierung: Diese stellt bei (oder in FastAsy, wie wir später sehen werden: vor) der ersten Anfrage sofort *alle* Anfragen, die überhaupt relevant sein können, d.h. sie traversiert systematisch alle Daten, die der Vorgänger überhaupt liefern kann, und merkt sich dieselben. Alle Anfragen werden dann direkt ohne weitere Rückfrage beantwortet. Wie man sich vorstellen kann, eignen sich solche Materialisierungen auch gut, um Zwischenobjekte persistent zu machen. Man wird aber natürlich gerade nicht jedes Zwischenergebnis materialisieren, sonst verlöre das ganze Design ja seinen Sinn. Das Reizvolle an den Materialisierungen ist aber, dass sie für Vorgänger und Nachfolger völlig transparent sind. Sie ändern zwar massiv die Aufrufstruktur, jedoch in keiner Form etwas am Ergebnis.

Benutzer haben durch Auswahl verschiedener Materialisierungsstrategien ein ausgezeichnetes Mittel, den *tradeoff* zwischen Laufzeit und Speicherbedarf zu bestimmen.

Ergänzend sei die vorgestellte Architektur noch kurz mit dem Schichten-Muster (bzw. in diesem Falle einer zweiten Instanz desselben innerhalb der Bausteine-Schicht) verglichen: Verschiedene Schichten realisieren in der Regel nicht dasselbe Interface und sind somit untereinander nicht vertauschbar. Eine solche Vertauschbarkeit widerspräche auch der Unterteilung eines Systems in Dienste von hohem Abstraktionsgrad, die in den oberen Schichten zu finden sind, und solche von niedrigem Abstraktionsgrad in den unteren Schichten. Eine eindeutige Parallelität zum Schichten-Muster stellt jedoch natürlich die kaskadierte Delegation von Anfragen dar.

### 9.2.3 Seitenergebnisse

In Abweichung von dem ganz strengen Konzept, das wir gerade vorgestellt haben, kann es natürlich trotzdem sein, dass zusätzliche Ergebnisse produziert werden, die evtl. erst spätere Phasen benötigen. Dies stellt an sich kein Problem dar, denn die einzelnen Phasen können solche *Seitenergebnisse* ja in ihrem Zustand speichern und ihr Interface derart erweitern, dass diese öffentlich ausgelesen werden können. Man mache sich nur bewusst, dass jede Benutzung eines Seitenergebnisses sowohl Aus- als auch Vertauschbarkeit der Komponenten weiter einschränkt.

In FastAsy haben wir es momentan nur mit zwei solchen Seitenergebnissen zu tun, beide sind aus demselben Grund nötig und werden auf dieselbe Art gespeichert: Es handelt sich zum einen um die Indizierung der Aktionsnamen des Petrinetzes und zum anderen um die Indizierung der Zustände wie in 7.2.3 beschrieben. Beide werden mit Hilfe des *regref*-Template aus der BE++ gespeichert (siehe 9.7.2).

Die Aktionsnamen werden bereits in dem Adapter, der das Quellsystem darstellt, indiziert, denn ohne diese Indizierung ließen sich Verweigerungsmengen, die ja eben Mengen aus Aktionen sind, nicht effizient (als Bitvektor) darstellen. Diese sehr frühe Indizierung ist (im Gegensatz zu den Zuständen) daher so einfach möglich, weil wir ausschließen können, dass die Namen der Aktionen für spätere Phasen eine Rolle spielen.

Wie wir bereits gesehen haben, erfolgt die Indizierung der Zustände explizit als eigene Phase. Dies hat eben den Grund, dass wir somit den Zeitpunkt der Indizierung verschieben können und so für den Fall vorbereitet sind, dass Komponenten noch Details der Zustandsinformation sehen möchten.

Zwingend notwendig ist die Indizierung der Zustände sowieso erst bei der Konstruktion des Potenzautomaten. Da wir gesehen haben, dass die Anzahl erreichbarer Zustände bis dahin abnehmen kann (Elimination der λ-Kanten), stellt einen weiteren Grund dar, warum wir Zustände erst spät indizieren wollen. Alternativ könnte man natürlich auch mehrfach indizieren, wobei in dem Fall, dass man Ganzzahlen mit überschaubarem Wertebereich indiziert, natürlich eine spezialisierte Indexer-Klasse zum Einsatz kommen sollte.

Als kleiner Exkurs sei noch kurz erwähnt, wie sich die Bedingung an das Vorgängersystem, indizierte Zustände zu haben, im Rahmen der Typsicherheit aus 9.2.2 ausdrückt: Die Phase der Potenzautomatenkonstruktion fordert nämlich von derjenigen Template-Instanz des *InductiveSystem*-Interface, von der der Vorgänger erbt, dass sie im Typparameter *TState* den Wert *TIntState* hat. (*TIntState* ist dabei nichts anderes als ein über **typedef** definierter Alias für einen Ganzzahltyp.) In den Begriffen der generischen Programmierung ausgedrückt entsteht also ein als Vorgänger des Potenzautomaten taugliches *InductiveSystem* aus dem allgemeinen, indem der erste Parameter an den Wert *TintState* gebunden wird, wodurch das verbleibende Template immer noch im zweiten Argument, nämlich dem Typ *TStep* der Schrittbeschreibung, parametrisch ist.

## *9.3 Entwurf des Package „Petrinetz“*

Innerhalb der Gesamtarchitektur von FastAsy fasst das Petrinetz-Package innerhalb der Quellsystem-Schicht jegliche Funktionalität zusammen, die spezifisch für Petrinetze ist. In Abgrenzung dazu zählt etwa der zeitbehaftete Erreichbarkeitsgraph schon nicht mehr dazu, da auch für andere Arten von Transitionssystemen, die nicht als Petrinetz notiert sind, eventuell ein solcher angegeben werden kann, und sich solche Systeme sehr wohl auch im Fokus von FastAsy befinden.

Im folgenden wollen wir zunächst dem üblichen Ansatz folgen und die Anforderungen an die Schicht aus dem Blickwinkel eines Clients, welcher in unserem Fall durch die höher liegenden Schichten gegeben sein wird, sammeln und konkretisieren. Auf eine formale Ausarbeitung der Anforderungen wie etwa in [BS02] dargestellt, wollen wir hier verzichten, da zum einen der Platzaufwand unverhältnismäßig hoch wäre und zum anderen in einem Szenario wie unserem, in dem die rein funktionalen Anforderungen ja größtenteils bereits mathematisch formalisiert vorliegen, der Nutzen einer echten Anforderungsanalyse gar nicht gegeben wäre. Es sei noch bemerkt, dass viele der Anforderungen natürlich in der Tat bereits den Blick auf eine spätere Implementierung gerichtet haben. Dies scheint natürlich, wenn man bedenkt, dass die Clients eben durch andere Programmteile gegeben sein werden.

In einem zweiten Schritt folgt eine Verdichtung zu Verantwortlichkeiten und deren Abgrenzung und Aufteilung. Nicht ganz zu Unrecht erscheint der Schritt von den Anforderungen hin zu einem Entwurf des Klassenverbunds Neulingen im objektorientierten Vorgehen als derjenige, bei dem man von den formalen Methoden noch am meisten im Stich gelassen wird. Ein Großteil erfahrener Designer wird auf die Frage nach ihrem Vorgehen an dieser Stelle zugeben müssen, „aus dem Bauch“ heraus und meist evolutionär vorzugehen, bis

eine ausreichend hohe Kohärenz der Klassen erreicht ist und nur noch wenige vermeidbare Kopplungen vorliegen. Der Autor möchte sich davon nicht ausnehmen. Letzten Endes fällt dieser Schritt einfach mit dem Übergang von natürlicher Sprache in formalisierte oder zumindest formal darstellbare Begrifflichkeiten zusammen, und dieser Tatsache liegt es wohl inne, dass es keine rein mechanische Umsetzung geben kann. Ein guter Überblick über zumindest halbformale Vorgehensweisen wird aber beispielsweise in Grady Boochs immer noch gültigem Standardwerk [GB94] gegeben. Agile Softwaremethoden, insbesondere XP, verfolgen an dieser Stelle sowieso andere Ansätze, da sie meist die Aufteilung der Verantwortlichkeiten über den Zeitraum der Entwicklung eines Programms hinweg nicht als stabil ansehen. Wir werden uns bemühen, auch der Dynamik dieser Entwurfsphase in unserer Darstellung Rechnung zu tragen, konzentrieren uns ansonsten aber in erster Linie auf die Präsentation des Ergebnisses.

Abschließend werden wir an Hand ausgewählter UML-Diagramme einen Überblick über die zentralen Klassen und deren Kollaborationsmechanismen geben, auch mit dem Hintergedanken, Interessierten den Zugang zum Quelltext zu erleichtern.

Die letzte Detailebene, nämlich die Dokumentation des Quelltexts, kann an dieser Stelle natürlich nicht mehr ausgebreitet werden, es sei dazu auf die online verfügbare Quelltextdokumentation [FSYO] verwiesen.

Zur Nomenklatur innerhalb dieses Kapitels sei bemerkt, dass mit dem Begriff Netzklasse keine Klasse im Sinne objektorientierter Programmierung gemeint ist, sondern eine konkrete Netzvariante innerhalb der Theorie der Petrinetze. Wir verwenden in Abgrenzung dazu den Begriff Petrinetz-Klasse, wenn wir eine OO-Klasse bezeichnen.

## 9.3.1 Anforderungen

### *9.3.1.1 Verwaltung von Netzstrukturen*

Unabhängig von deren späterer Interpretation benötigen wir zunächst natürlich eine Möglichkeit, die reinen Petrinetze im Arbeitsspeicher zu verwalten. Wir bemerken dazu folgende Punkte:

- Es ist sehr wahrscheinlich, dass wir in Zukunft Netzklassen verarbeiten wollen, die wir momentan noch nicht kennen.
- Wir würden uns auf der einen Seite diesbezüglich gerne viele Möglichkeiten offen halten, auf der anderen Seite wünschen wir uns aber, dass „kleine“ Erweiterungen an einer Netzklasse auch nur „kleine“ Erweiterungen des Codes nötig machen. Es wird also ein mehrstufiges Konzept für Erweiterungen geben müssen.
- Insbesondere sollen Erweiterungen an der Beschriftung von Transitionen, Kanten oder Stellen möglichst einfach durchzuführen sein.
- Wir benötigen Persistenzmechanismen, die einerseits darauf ausgelegt sind, unterschiedliche externe Formate zu bedienen und die sich andererseits aber auch leicht auf neue Netzklassen anpassen lassen.
- Wir werden in höher liegenden Schichten von den wirklichen Transitionen, Stellen und Kanten abstrahieren und höchstens noch Teile ihrer Beschriftung verwenden wollen. Deswegen spricht nichts dagegen, die Bestandteile des Netzgraphen tatsächlich als *heavyweight*-Klassen, d.h. mit nichttrivialen Dateninhalten wie z.B. Adjazenzinformationen, anzulegen.

- An dieser Stelle fällt bereits auf, dass die Netzmarkierung möglicherweise ein Grenzgänger sein wird: Einerseits soll die Anfangsmarkierung definitiv von den erwähnten Persistenzmechanismen abgedeckt sein, andererseits wollen wir den *Laufzeitkontext* unserer Netze, d.h. die Zustandsinformation im r-Erreichbarkeitsgraphen, gerne getrennt verwalten wie im folgenden beschrieben.

#### *9.3.1.2 Verwalten des Laufzeitkontexts*

Obwohl die Anfangsmarkierung ja Teil des Netzes ist, wollen wir die während des Token-Game entstehenden „Markierungen“ nicht im Netz verwalten. Dies hat mehrere Gründe:

- Die Inhalte, die der Laufzeitkontext für eine Netzklasse beschreibt, sind i.A. nicht identisch mit denen der Anfangsmarkierung. Beispielsweise braucht die Anfangsmarkierung einer Netzklasse, deren Schaltregel mit zeitbehafteten Markierungen operiert, selbst keine Zeitinformation zu tragen.
  (Wir benötigen natürlich eine Vorschrift, um den initialen Laufzeitkontext aus der Anfangsmarkierung zu berechnen.)
- Unterschiedliche Schaltregeln für ein und denselben Netzgraphen werden i.A. unterschiedliche Arten von Laufzeitkontexten benötigen.
- Bei Berechnungen können unterschiedliche Instanzen des Laufzeitkontexts zur selben Zeit (es ist von der Zeit im Anwendungsprogramm die Rede, nicht von der Zeit im Petrinetz) benötigt werden.

#### *9.3.1.3 Berechnungen auf dem Laufzeitkontext*

Wir wollen für ein gegebenes Petrinetz und einen gegebenen Laufzeitkontext folgende Fragen beantworten können:

- Welche Transitionen sind aktiviert?
- Welche Transitionen sind dringend?
- Wie sieht der nachfolgende Laufzeitkontext nach dem Schalten einer gegebenen Transition aus?
- Wie sieht der nachfolgende Laufzeitkontext aus, nachdem eine Zeiteinheit verstreicht?

Indem wir diese Fragen so stellen, treffen wir für die Petrinetz-Schicht einige Grundannahmen, denn für Netzklassen, die etwa einen anderen Begriff von Zeit haben, ergibt ein Teil dieser Fragen womöglich keinen Sinn. In unseren theoretischen Arbeiten ist jedoch genau diese Auffassung von Zeit ebenfalls grundlegend, so dass wir der Meinung sind, hiermit die richtige Abstraktionsebene getroffen zu haben. Darüber hinaus ist es ja sogar so, dass selbst auf dem benachbarten Gebiet der Prozessalgebra immer noch ein derartiger Zeitbegriff Verwendung finden kann (siehe z.B. [CVJ02]).

Zu unserer Entscheidung sei auch noch bemerkt, dass die höher liegenden Schichten natürlich trotzdem auch mit Netzen gespeist werden könnten, deren zeitliche Semantik ganz anders aufgebaut ist, nur müsste dann eben alles unterhalb der Adapterklasse speziell implementiert werden. Letzten Endes stellen aber Aktivierung und Dringlichkeit einfach die Fundamente der Bestimmung von Verweigerungsmengen dar.

Wir bemerken weiterhin an dieser Stelle bereits, dass sich die oben gestellten Fragen zumindest in den untersuchten Netzklassen auch aus lokaleren Fragen zusammensetzen lassen, wenn man Kenntnis von der Netzstruktur hat:

- Ist eine gegebene Transition aktiviert?
- Ist eine gegebene Transition dringend?

Das Schalten einer Transition scheint sowieso inhärent lokal zu sein, das Vergehen von Zeit jedoch inhärent global. Weiterhin scheint es, als könne man gegebenenfalls manche Fragen noch weiter lokalisieren (etwa die Aktivierung auf die Kanten, s.u.), wenn man erst einmal eine konkrete Schaltregel vor sich hat. Wir merken uns diese Beobachtungen bereits vor, um dem bei der Aufteilung der Verantwortlichkeiten Rechnung tragen zu können. Je lokaler Fragen sein werden, die Implementierungen vollständig abstrakter Klassen (=Interfaces in der Java-Diktion) zu beantworten haben, desto einfacher werden neue Implementierungen zu erstellen sein. In diesem Versuch der Lokalisierung schlägt sich letzten Endes nur das allgemeine Bestreben in der OO nieder, Sachverhalte möglichst früh, d.h. nah bei den Basisklassen, zu formulieren, um Duplikation von Funktionalität zu vermeiden. In noch klarerer Form werden wir diesem Grundsatz bei den *InductiveSystem*-Klassen begegnen, wo wir dann auch genauer darauf eingehen wollen.

Die ursprüngliche Idee, die Aktivierung einer Transition durch einen streng lokalen Blick zu entscheiden, stammt aus dem Petri Net Cube, einer Erweiterung des Petri Net Kernel (siehe [PNK]). Dort werden dazu alleine die zu einer Transition inzidenten Kanten und die jeweils aktuelle Markierung betrachtet. In FastAsy haben wir diesen Ansatz konsequent erweitert, um neben der Aktivierung auch entscheiden zu können, ob eine Transition dringend ist. Außerdem wurde in FastAsy die Struktur der Delegierung mehrstufig und damit flexibler gestaltet, so dass die Entscheidung nicht in den Kantenobjekten fallen *muss* (siehe 9.3.2.1).

#### *9.3.1.4 Trennung von Struktur und Verhalten*

Wir stellen noch einmal explizit fest, dass wir Aspekte unserer Petrinetzklassen, die sich unabhängig voneinander ändern können, auch in der Modellierung gerne unabhängig halten möchten:

- Die Inhalte der Laufzeitkontexte sollen sich auswechseln lassen.
- Die Schaltregel soll sich austauschen lassen, sie wird sich ihrerseits jedoch natürlich auf den Laufzeitkontext und die Beschriftungen von Stellen, Transitionen und Kanten abstützen. (Wo selbst einige dieser Spezifika noch eliminiert werden können, böte sich generische Programmierung an, um zu generalisierten Schaltregeln zu gelangen. Dies ist momentan jedoch nicht der Fall.)
- Die Berechnung des initialen Laufzeitkontexts aus der Anfangsmarkierung soll variabel sein.
- Zur Verwaltung der Netzstruktur ist keine Kenntnis von Laufzeitkontext oder Schaltregel notwendig.

#### *9.3.1.5 Verwendbarkeit als InductiveSystem*

Als Grenze der Petrinetz-Schicht haben wir bereits eingangs den Erreichbarkeitsgraphen ausgemacht. Während die Schaltregel noch eindeutig in unsere Schicht gehört, steht er außerhalb. Wir benötigen einen Mechanismus auf der Schichtgrenze, der folgendes bietet:

- Jedes Petrinetz soll sich als Implementierung der *InductiveSystem*-Abstraktion ansprechen lassen.
- Die Sicht auf ein Petrinetz als *InductiveSystem* ist aber ein sekundärer Aspekt des Netzes, so dass wir Klassen dieser Schicht nicht mit entsprechender Funktionalität verunreinigen möchten. Es bietet sich also an, die Systemgrenze als *Adapter* (nach [GoF96] die Umsetzung einer Schnittstelle auf eine andere) zu konzipieren. Damit fügen sich Petrinetze lückenlos in unser Bauskastensystem ein.
- An die Speicherung von Knoten eines Erreichbarkeitsgraphen werden andere Anforderungen gestellt als an die Modelle der Laufzeitkontexte. Während die Information des Laufzeitkontexts semantisch reichhaltig ist und im Interesse einer problemnahen Formulierung der Schaltregel eine Schnittstelle zur Verfügung stehen sollte, die diese Semantik in ihrer ganzen Breite respektiert, soll die Information in den Knoten möglichst effizient gespeichert sein und muss lediglich elementare Operationen wie Zuweisung oder Vergleich zur Verfügung stellen. Übergänge zwischen verschiedenen Repräsentationen müssen also möglich sein.
- In noch stärkerer Form finden wir diese Forderung nach einer Umsetzung der Repräsentation bei den Kanten wieder: Wo die Schaltregel von Aktiviertheit und Dringlichkeit von Transitionen spricht, sollen im Erreichbarkeitsgraphen ja Aktionen und Verweigerungsmengen stehen.
- Idealerweise möchten wir erreichen, dass höhere Schichten nur über diesen einzelnen Punkt, den geplanten *InductiveSystem*-Adapter, mit der Petrinetz-Schicht sprechen, alle darunter liegenden Details sollen als Interna angesehen werden. Nach dieser strengen Auffassung würde der Adapter dann in der Diktion von [GoF96] ebenfalls wieder eine *Fassade* darstellen, jedoch nicht zu verwechseln mit den Fachfassaden. Wir werden später jedoch in Abweichung vom Fassadenmuster Aufgaben der Persistenz direkt nach außen führen, um aufrufendem Code an dieser Stelle Möglichkeiten zu weitergehenden Manipulationen einzuräumen, ohne die Fassadenschnittstelle unnötig aufzublähen. Dies wird aber, wie in 9.3.2.5 ausgeführt, in sehr kontrollierter Art und Weise geschehen.
- Die Entkopplung durch einen Adapter wird insbesondere dann wichtig werden, wenn wir nicht nur Petrinetze, sondern auch andere Quellsysteme behandeln möchten, denn in einem solchen Fall sehen diese Details eben tatsächlich ganz anders aus.

## 9.3.2 Evolution des Klassenverbunds

### *9.3.2.1 Netzstruktur und Delegation von Berechnungen*

Ausgehend von dem in 9.3.1.1 gesagten scheint es selbstverständlich, die Verwaltung von Netzstrukturen an einer Stelle, der Klasse *PetriNet*, zusammenzufassen. Zunächst obliegen dieser Klasse die folgenden Aufgaben:

- Verwaltung und Veränderung des Netzgraphen
- Automatische Indizierung von Transitionen, Stellen und Kanten mit ganzen Zahlen
- Beantworten von Abfragen über die mit Transitionen, Stellen und Kanten assoziierten Beschriftungen
- Beantwortung von Abfragen über Adjazenzen

(Die Indizierung ist sowieso ein Nebenprodukt der Speicherung in *vector*-Containern und wird nach außen geführt, um eine einfache Möglichkeit zu haben, etwa bei der Realisierung von Persistenzmechanismen Objekte referenzieren können, ohne dies über ihren Wert zu tun und somit Objektidentität mit Wertgleichheit gleichzusetzen.)

Wir bemerken, dass zumindest die beiden letzten Fragen sich ja direkt an Transitionen, Stellen und Kanten richten und beschließen deshalb wie schon angedeutet, diese Entitäten auch explizit als Klassen zu modellieren. Sie sollen jedoch nur Fragen beantworten, die sie nicht zurück an das Netz selbst delegieren müssen, denn wir möchten nicht jedes einzelne Objekt mit einer rückwärtsgerichteten Assoziation belasten. Daraus folgt aber insbesondere, dass die Strukturinformationen verteilt in diesen Klassen vorliegen und diese über Zeiger realisiert sein müssen, nicht über die vom Netz verwalteten Indizes. Wir treffen die weiteren Entscheidungen:

- Transitionen, Stellen und Kanten bilden mit dem Netz eine sog. *Komposition* (deren Charakteristikum nach [BO01] die Existanzabhängigkeit der Teile vom Ganzen, hier die Abhängigkeit von Transitionen, Stellen und Kanten vom Netz, ist).
- Obwohl also das Netz für den Lebenszyklus solcher Objekte verantwortlich ist, wollen wir die Möglichkeit behalten, Unterklassen für Transitionen, Stellen und Kanten zu erstellen. Zu diesem Zweck führen wir [GoF96] folgend eine *Factory*-Klasse ein, welche die Erstellung konkreter Objekte kapselt und so austauschbar macht.
- Wir entscheiden uns dagegen, die erwähnten Klassen im Sinne des *Whole-Part*-Konzepts aus [FB00] hinter der Petrinetz-Klasse zu verbergen, da aufrufender Code, der sich direkt an die untergeordneten Objekte wenden kann, wesentlich weniger *semantische Entfernung* aufweist. (Der Begriff semantische Entfernung dürfte aus [MF00] stammen und wird dort als das Phänomen aufgefasst, dass Code, der sich mit vielen technischen Details beschäftigen muss, als „Prosatext“ gelesen nur mehr wenig über seine eigene Funktion aussagt. In diesem Sinne ist der Codetext dann „entfernt“ von der Semantik. Die Vermeidung semantischer Entfernung führt idealerweise zu Code, der ohne explizite Kommentare trotzdem auch dem oberflächlichen Leser preisgibt, was er tut.)
- Wir wollen verschiedene Arten von Kanten (Stelle→Transition, Transition→Stelle, Schleife, ReadArc) nicht mit Hilfe von Polymorphie unterscheiden, da dies den Aufwand für eventuelle Nachfolgerklassen erhöhen würde. Stattdessen interpretieren wir den *ArcType* als Teil der Beschriftung (s.u.). Um trotzdem bequem Anfragen in gewohntem Sinne nach Vor-, Nach- und Lesebereich stellen zu können, parametrisieren wir die *GetAdjacentPlaces()*-Methode in *Transition* und *GetAdjacentTransitions()* in *Place* mit einem *Prädikatsobjekt* (siehe [BS98], S. 550), in welchem die zu beachtenden Werte von *ArcType* festgelegt sind. So würden wir beispielsweise {*P→T, Loop*} in einem Aufruf von *GetAdjacentPlaces()* als Parameter übergeben, um den (konventionellen) Nachbereich zu erhalten. Um die Lesbarkeit des Codes weiter zu erhöhen, führen wir symbolische Konstanten für gängige Kombinationen ein, im genannten Beispiel etwa *atsTransitionReducedPostset*. (Das genaue Aufrufschema wird in 9.3.3.2.2 dargestellt.)

Die mit Abstand häufigsten Erweiterungen von Transitionen, Stellen und Kanten werden sich wie bemerkt auf die Beschriftungen beziehen. Weiter scheint es zumindest denkbar, auch

inhomogene Beschriftungen zulassen zu wollen, was ja das Konzept der *Factory*-Klasse sprengen würde. Aus diesen Gründen

- lagern wir jede Art von Beschriftung in einer weiteren Stufe in eine Familie polymorpher Klassen mit Basisklasse *Inscription* aus,
- definieren wir drei Klassen *BasicTransitionInscription*, *BasicPlaceInscription* und *BasicArcInscription*, welche Anfragen unterstützen, die unabhängig von der Netzklasse Sinn ergeben (beispielsweise der symbolische Name oder die Richtung einer Kante),
- definieren wir mit *GenericPetriNetInscription* sogenannten *boilerplate*-Code (ein Begriff aus [AA01]), d.h. einen Mechanismus, der uns bei Erweiterungen der drei obigen Klassen alle Routineaufgaben soweit wie möglich abnimmt.

Somit ist das Ableiten von *Transition*, *Place* oder *Arc* nur mehr nötig, wenn tiefer in die Funktionalität des Netzes eingegriffen werden soll, beispielsweise wenn die Menge unterstützter Abfragen nicht ausreichen sollte.

In 9.3.1.3 hatten wir festgestellt, dass sich einige Berechnungen lokal anstellen lassen, und dass es aber von der konkreten Ausprägung der Netzklasse abhängig sein kann, wie weit diese Lokalisierung gehen darf. Es bietet sich an, hier eine flexible Lösung in der Art des *Zuständigkeitsketten*-Musters aus [GoF96] zu verwenden, allerdings mit einem Unterschied: Während entlang einer Zuständigkeitskette lediglich der erste Empfänger gesucht wird, der sich für ein Ereignis verantwortlich fühlt, haben wir in unserem Fall die Möglichkeit vorgesehen, Ereignisse bzw. Anfragen auf verschiedenen Ebenen partiell abzuarbeiten. (Wir werden in 9.3.2.3 aber sehen, dass wir die Arbeitsteilung zwischen den Ebenen statisch festgelegen können.)

Konkret wird z.B. das Feuern einer Transition t wie folgt abgearbeitet:

- globale Aktionen werden auf dem Laufzeitkontext ausgeführt
- für t spezifische Aktionen werden auf dem Laufzeitkontext ausgeführt
- für alle an t anliegenden Kanten a spezifische Aktionen werden auf dem Laufzeitkontext ausgeführt

(Man beachte, dass eine weitere Delegation an die über a verbundene Stelle p trotz des zugegebenermaßen vorhandenen ästhetischen Reizes keinen Sinn mehr ergeben würde, da p ja bereits eindeutig bestimmt ist und so alle für p spezifischen Aktionen genauso gut als Teil der für a spezifischen durchgeführt werden können.)

Demselben Delegationsschema folgen auch das Schalten eines Zeitschritts sowie die Ermittlung aktivierter oder dringender Transitionen. Wollen wir nun z.B. eine uns vorliegende Schaltregel (es sei vorweggenommen, dass wir solche in 9.3.2.3 als Klasse modellieren werden) implementieren, können wir uns für jede Aufgabe aussuchen, auf welcher Ebene wir sie am elegantesten formulieren können.

Da wir ja die Transitionen und Kanten als Klassen vorliegen haben und diese auch alle zur Delegation benötigten Daten lokal vorhalten, verwenden wir natürlich Methoden dieser Klassen zur Delegation. Dies stellt einen Unterschied zum Petri Net Cube dar, in welchem das Netz „von oben herab“ direkt über die Liste der Kanten iteriert.

Letzten Endes sehen wir an dieser Stelle, dass die Behandlung der Ein-/Ausgabe von Netzen nicht trivial sein wird, da sowohl unterschiedliche Formate möglich sein sollen als auch unterschiedliche Objektklassen (insbesondere Nachfolger von *Inscription*) ins Spiel

kommen. Wir beschließen deshalb, diese Funktionalität mit Hilfe von IO-Adaptern zu realisieren (s. 9.3.2.5), statt die Petrinetz-Klasse damit zu belasten.

#### *9.3.2.2 Abstrakte Markierungen*

Sehen wir uns die Anforderungen aus 9.3.1.2 an, so bemerken wir:

- Über die Operationen, die ein Laufzeitkontext zur Verfügung stellen muss, lässt sich auf dieser Abstraktionsebene keinerlei Aussage treffen.
- Trotzdem wird es sich anbieten, die verschiedenen Arten von Laufzeitkontexten durch zueinander polymorphe Klassen zu modellieren, da sie an vielen Stellen transparent behandelt werden.

Das Interface der Basisklasse *AbstractMarking* wird also ausschließlich technische Operationen (Zuweisung, Vergleich, E/A) spezifizieren. Konkrete Funktionalität werden erst die abgeleiteten Klassen bieten können. Daraus folgt aber sofort, dass es im aufrufenden Code eine korrespondierende Ableitungshierarchie geben muss, die diese individuell erweiterte Funktionalität dann kennt und nutzen kann. Wir machen uns kurz klar, an welchen Stellen solche Aufrufe erfolgen:

- bei der Formulierung der Schaltregel
- bei der Umsetzung der Information auf Knotenbeschriftungen im Erreichbarkeitsgraphen (siehe 9.3.1.5)

Wir werden in 9.3.2.3 respektive 9.3.2.6 sehen, dass an diesen Stellen sowieso in natürlicher Weise der aufrufende Code spezifisch für die jeweilige Ausprägung des Laufzeitkontexts ist.

#### *9.3.2.3 Abstrakte Schaltregeln*

Offenbar scheint es sinnvoll, die vier anfangs in 9.3.1.3 angegebenen Operationen in einer Klasse zusammenzufassen. Weiter haben wir jedoch gesehen, dass wir die Implementierung dieser Operationen auf verschiedenen Ebenen zulassen möchten. Aus diesem Grund führen wir zunächst eine Klasse *FiringRule* ein, welche jedoch im Wesentlichen nur eine *Komposition* aus weiteren Klassen darstellt:

- Ein Nachfolger von *GlobalFiringRule*, der ggf. die Anfragen auf globaler Ebene beantwortet
- Ein Nachfolger von *TransitionFiringRule,* der ggf. die Anfragen auf Transitionsebene beantwortet
- Ein Nachfolger von *ArcFiringRule*, der ggf. die Anfragen auf Kantenebene beantwortet

Wir stellen fest, dass es im Sinne der Kohärenz ist, in diese Komposition auch noch einen Nachfolger von *InitialMarkingRule* aufzunehmen, welcher verantwortlich sein wird für das Zustandekommen des initialen Laufzeitkontexts. Wir werden dazu in 9.3.2.4 noch gesondert einige Dinge bemerken.

Man beachte nun zwei Details:

- Während die …*FiringRule*-Objekte polymorph sind, wurde *FiringRule* selbst als nicht-polymorphe Klasse entworfen, da sie nur eine Sammlung von Objekten darstellt, die ihrerseits polymorph sind.

- Die Klasse *FiringRule* soll nicht dafür verantwortlich sein, ankommende Anfragen auf die verschiedenen Ebenen weiterzuleiten, da sie ja gar keine Kenntnis der Netzstruktur hat. ([MF00] spricht in diesem Zusammenhang vom „Neid“ einer Klasse auf die internen Daten einer anderen, welchen es zu vermeiden gilt. Dies wäre hier gegeben, wenn FiringRule sich ständig der Daten von PetriNet bedienen müsste, um Anfragen zu bearbeiten.)

Die in unseren Augen eleganteste Aufgabenteilung sieht nun vor, die Beantwortung der eingangs erwähnten vier Anfragen in das Interface von *PetriNet* zu verschieben. Dies scheint einerseits intuitiv und zweitens kann *PetriNet* aufgrund seiner Kenntnis der Netzstruktur die Anfragen sofort an die Teile von *FiringRule* delegieren. Freilich müssen wir dazu die Anfrage um die Angabe der Schaltregel ergänzen. Wir parametrisieren also Anfragen an das Petrinetz mit einer *Strategie* (nach [GoF96] die Kapselung einer zusammengehörigen Familie von Algorithmen, in unserem Fall der Teile der Schaltregel), und übergeben der Petrinetz-Klasse die Verantwortung, die Bestandteile der Strategie mit dem jeweils passenden Kontext aufzurufen.

Zur Aufgabenteilung zwischen *FiringRule* und *AbstractMarking* sei allgemein noch folgendes bemerkt: Prinzipiell wäre es natürlich auch denkbar, die beiden Klassenhierarchien zusammenzufassen, so dass entsprechende Abfragen an *PetriNet* ein *AbstractMarking*-Objekt als Parameter erhalten, welches dann sowohl die für Berechnungen nötigen Daten wie auch die zugehörige Strategie kennt. Dies scheint sowohl den Forderungen nach hoher Kohärenz wie auch dem OO-Grundsatz nach Zusammenfassung von Daten und zugehörigen Operationen gerecht zu werden. Dazu ist zu sagen, dass die übersteigerte Kohärenz an dieser Stelle dadurch erkauft würde, dass wir zwei Aspekte, die orthogonal sein sollten, nun nicht mehr unabhängig voneinander austauschen könnten. Ergänzend sei deutlich darauf hingewiesen, dass sich die Zusammenfassung von Daten und Operationen in der OO auf *elementare* und den Daten *inhärente* Operationen bezieht - eine Trennung in Klassen, bei denen entweder der operative oder der Datencharakter überwiegt, ist aus guten Gründen eher die Regel als die Ausnahme.

#### *9.3.2.4 Zustandekommen der Anfangsmarkierung*

Wir haben in 9.3.2.3 angedeutet, dass wir die Bestimmung des initialen Laufzeitkontexts in eine Reihe stellen möchten mit den erwähnten Berechnungen auf dem Laufzeitkontext. Dabei werden die Ebenen der jeweiligen Implementierung von *InitialMarkingRule* jeweils für alle Transitionen, Stellen und Kanten sowie einmal mit dem globalen Kontext des Netzes aufgerufen und erhalten so die Gelegenheit, an der jeweils passenden Stelle mit minimalem Aufwand den initialen Laufzeitkontext zu berechnen. Beispielsweise würde bei PTT-Netzen (siehe Teil I ,1.4) lediglich der Stellen-Kontext verwendet und die entsprechende Komponente von $\rho$ auf 0 bzw. $-\infty$ gesetzt, wohingegen etwa Netzklassen, die noch die *Instantaneous Description* verwenden (siehe etwa [WV96]), auch den Transitionskontext benötigen, um die *Urgent*-Menge zu initialisieren. Wir sehen also, dass es auch von technischer Seite sinnvoll erscheint, *InitialMarkingRule* als Teil der *FiringRule* zu sehen.

#### *9.3.2.5 Adapter für Persistenz*

Es wurde bereits argumentiert, warum wir die Aufgaben für Ein- und Ausgabe in eine Familie von Adapterklassen auslagern möchten. (Der Begriff *Adapter* trifft dabei zwar in seiner technischen Bedeutung auch zu, denn es werden die von *PetriNet* zur Verfügung gestellten Operationen zur Manipulation der Netzstruktur umgesetzt auf die von den C++-IOStreams erwartete Schnittstelle, wir verwenden ihn aber hier bei der Klassenbezeichnung in erster Linie der Anschaulichkeit halber.)

Ein solcher Adapter soll dafür zuständig sein

- ein bestimmtes Datenformat zu lesen und dabei unter Verwendung von *PetriNet* das entsprechende Netz im Speicher zu erstellen,
- von *PetriNet* eine vorhandene Netzstruktur abzufragen und diese in einem bestimmten Format zu schreiben.

Dabei obliegt es dem Adapter, konkrete Instanzen von *Inscription* zu erstellen, welche ja für *PetriNet* transparent sind. Wir haben also damit die gewünschte Möglichkeit, Netze inhomogen zu beschriften. (Dahingegen sind die genauen Klassen für Transitionen, Stellen und Kanten durch die Factory-Klasse, mit der das *PetriNet*-Objekt initialisiert wurde, festgelegt.)

Obwohl das äußere Format einer Eingabedatei natürlich geprüft wird, ist es jedoch nicht die Aufgabe der Persistenzadapter, die Plausibilität oder Zulässigkeit von Beschriftungen zu prüfen. Stattdessen steht es *FiringRule* frei, beim Auftreten einer unerwarteten Inschrift eine entsprechende Ausnahme zu werfen. Im Gegensatz zum Persistenzadapter hat nämlich die Schaltregel auch die Möglichkeit, Fehler zu finden, die dynamischer Natur sind, wie beispielsweise Verletzung der *Sicherheit* bei einer Netzklasse, die diese zusichert.

Man beachte, dass es nicht nötig ist, eine abstrakte Basisklasse für die Adapterfamilie zu definieren, da an keiner Stelle im aufrufenden Code Transparenz bezüglich des konkreten Typs benötigt wird.

Weiter könnte man noch auf den Gedanken kommen, die Erstellung des Netzes (welche ja von der Netzklasse abhängt) vom Einlesen der Datei (welche vom Format abhängig ist) zu trennen. In einem solchen Fall würde man zu wiederum durch Strategien parametrisierten Adaptern gelangen, was bisher nicht notwendig ist, sich andererseits aber problemlos in die Architektur einfügen ließe.

#### *9.3.2.6 Interface nach außen*

Als Systemgrenze implementieren wir das Interface von *InductiveSystem*, welches (wie auch die Knoten- und Kantenbeschriftungen *RTState* und *RTStep*) später im Detail beschrieben wird, womit wir ein weiteres Mal auf das Entwurfsmuster des *Adapters* zurückgreifen. Kurz gesagt ist der entsprechende Nachfolger von *InductiveSystem* (in unserem Fall der PTT-Netze *PTTInductiveSystem*) dadurch dafür verantwortlich, zu einem System (hier dem bei der Erstellung festgelegten Petrinetz) eine induktive Definition zu geben, und zwar durch Beantwortung zweier Fragen:

- Anfangszustand des Systems
- Übergänge und Zielzustände für einen gegebenen Zustand

Wie in 9.3.1.5 gefordert, wird der Adapter dabei auch die Umsetzung der Inschriften für Knoten und Kanten übernehmen. Im konkreten Falle von *PTTInductiveSystem* sieht dies wie folgt aus:

- Objekte vom Typ *PTTMarking*, welche durch Verwendung von *InfinityInteger* (eines FastAsy-eigenen Typs, der ein Modell von $\mathbb{Z} \cup \{\infty, -\infty\}$ darstellt) zur Speicherung der Uhrenwerte ρ eleganten Code zur Formulierung der Schaltregel zulassen, werden umgesetzt in *RTState*-Objekte, welche eine maschinennähere Modellierung durch den Typ *int* unter Benutzung des Werts -1 als *magic number* verwenden.
  Man beachte, dass eine Bijektion zwischen den verwendeten Wertebereichen gegeben ist.

Aufwändiger ist die Umsetzung der Kantenbeschriftung:

- Besteht ein Übergang im Schalten einer Transition, so wird deren Beschriftung in ein Objekt vom Typ *RTStep* eingetragen und dieses als Kantenbeschriftung verwendet.
- Da wir die spätere Einbettung der Petrinetz-Schicht in die Applikation ja nicht kennen (bzw. uns als Designer an dieser Stelle gegenüber höheren Schichten bewusst „blind stellen“ müssen), rechnen wir damit, dass wir in der Aktionstabelle auch Aktionsnamen anderer Netze indiziert haben, welche in diesem Netz gar nicht verwendet werden. Um nun bei der Repräsentation der Verweigerungsmenge, welche in *RTStep* wiederum maschinennah als *dynamic_bitset* dargestellt wird, den Speicherbedarf nicht unnötig zu erhöhen, werden die Aktionen *gesliced*, d.h. es wird ein zweiter, privater Index mit dem kleinstmöglichen Wertebereich erstellt.
  Es wird die Menge der dringenden Transitionen daraufhin untersucht, ob ein Verweigerungsschritt möglich ist. Falls ja, wird dieser eben unter Benutzung des privaten Index als *RTStep* dargestellt.
- Es obliegt *PTTInductiveSystem* ebenfalls, falls die aufrufende Schicht dies bei der Objekterstellung angegeben hat, nicht nur die maximale Verweigerungsmenge, welche im vorigen Schritt in natürlicher Weise entsteht, zu liefern, sondern auch alle Schritte, die mit Teilmengen davon beschriftet sind, denn wie wir aus Teil I, 2.2.1.5 wissen, stellen diese ja ebenfalls gültige Übergänge zum selben Zielzustand dar.

Abschließend versehen wir *PTTInductiveSystem* noch mit einer Möglichkeit, den erwähnten privaten Index öffentlich auszulesen, um dem Benutzer bei der Ausgabe des Ergebnisses wieder die ursprünglichen Aktionsnamen anzeigen zu können.

## 9.3.3 Ausgewählte UML-Dokumentation

### *9.3.3.1 Klassendiagramme*

#### 9.3.3.1.1 PetriNet

Im folgenden Diagramm sehen wir zunächst die Assoziationsbeziehungen zwischen den für die Verwaltung der Netzstruktur wichtigen Klassen.

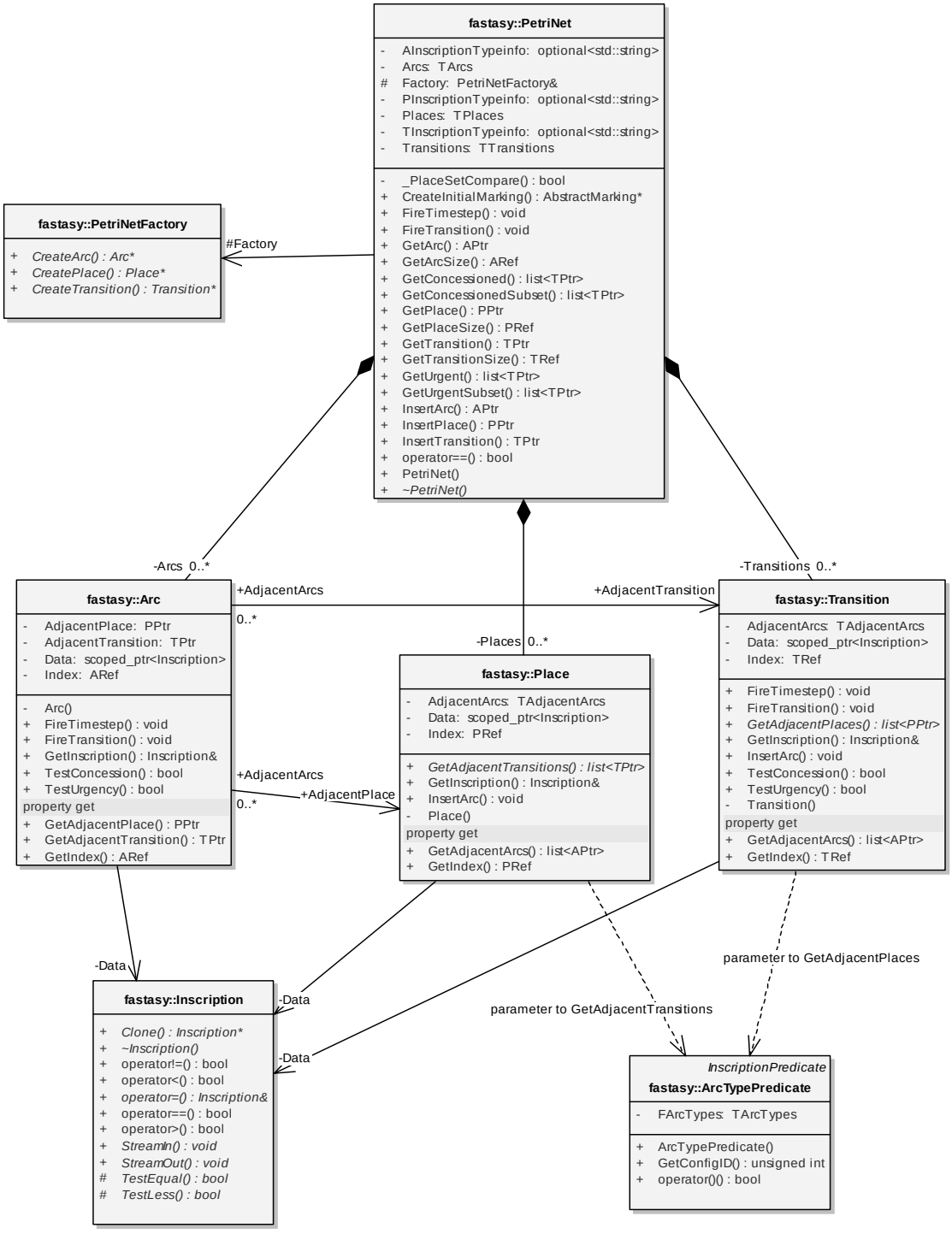



**Abbildung 34: Klassendiagramm PetriNet**

#### 9.3.3.1.2 Inscription

Im folgenden Diagramm wird die Vererbungsstruktur der *Inscription*-Klassen darsgestellt. Man beachte zwei Dinge:

- Obwohl *GenericPetriNetInscription* nicht explizit von Inscription erbt, erbt es von seinem *InscriptionBase*-Parameter und implementiert damit auch *Inscription*.
- Die Klassen *SimpleInscription* und *IntervalInscription* dienen lediglich Testzwecken.

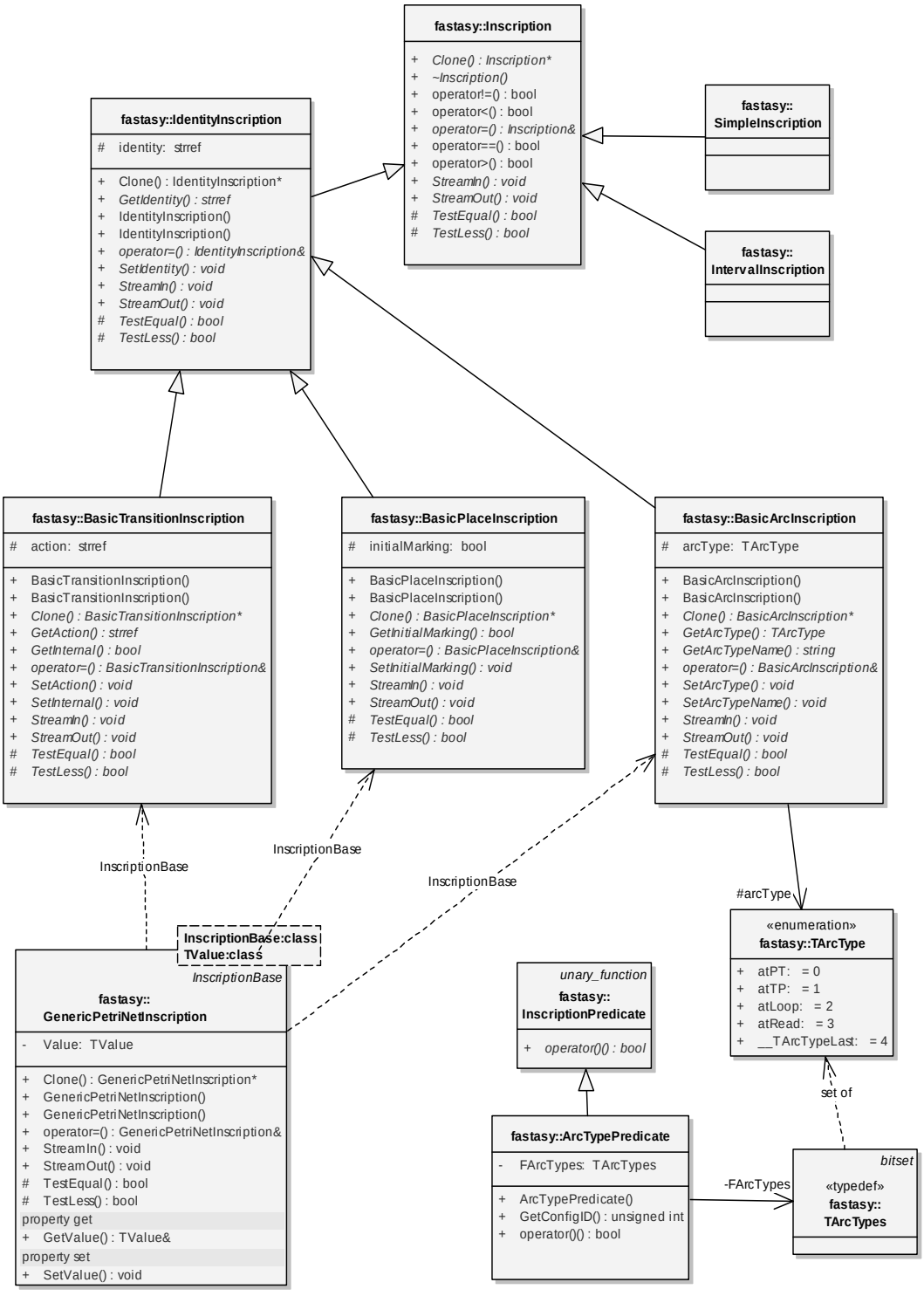



**Abbildung 35: Klassendiagramm Inscription**

#### 9.3.3.1.3 FiringRule

Dargestellt ist das Aggregat *FiringRule* und die konkreten Ausprägungen der einzelnen Teile im Falle der PTT-Netze. Wie man an diesem Beispiel sieht, ist es nicht nötig, alle Kontexte zu überladen. Für die PTT-Netze kann daher der Transitions-Kontext unbehandelt bleiben, da die natürliche Formulierung der Schaltregel im globalen und Kanten-Kontext stattfindet.

Man sieht ebenfalls, wie die konkrete Ausprägung der Markierung die im Rahmen der Schaltregel relevanten Teile des Interfaces erst an dieser Stelle definiert (siehe 9.3.2.2.).

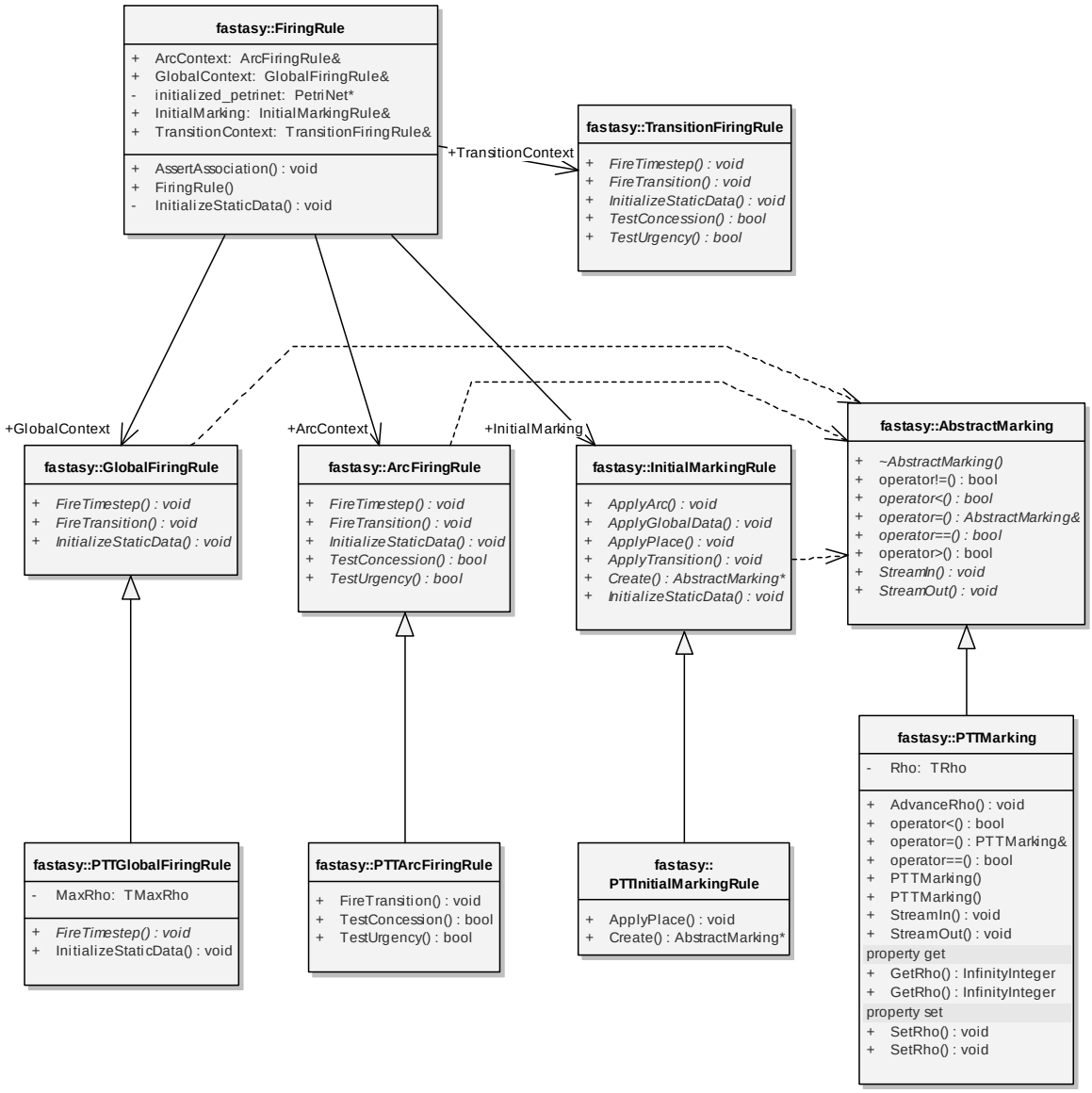



**Abbildung 36: Klassendiagramm FiringRule**

### *9.3.3.2 Kollaborationsmechanismen*

#### 9.3.3.2.1 FireTransition()

Als Beispiel für das in 9.3.2.1 beschriebene Delegationsverfahren sei hier ein Aufruf von *FireTransition()* beschrieben. Dabei ist **m** eine *AbstractMarking*, **t** eine *Transition* und **fr** die *FiringRule*, welche zur Anwendung kommen soll. (**m** wird durch *FireTransition()* geändert.)

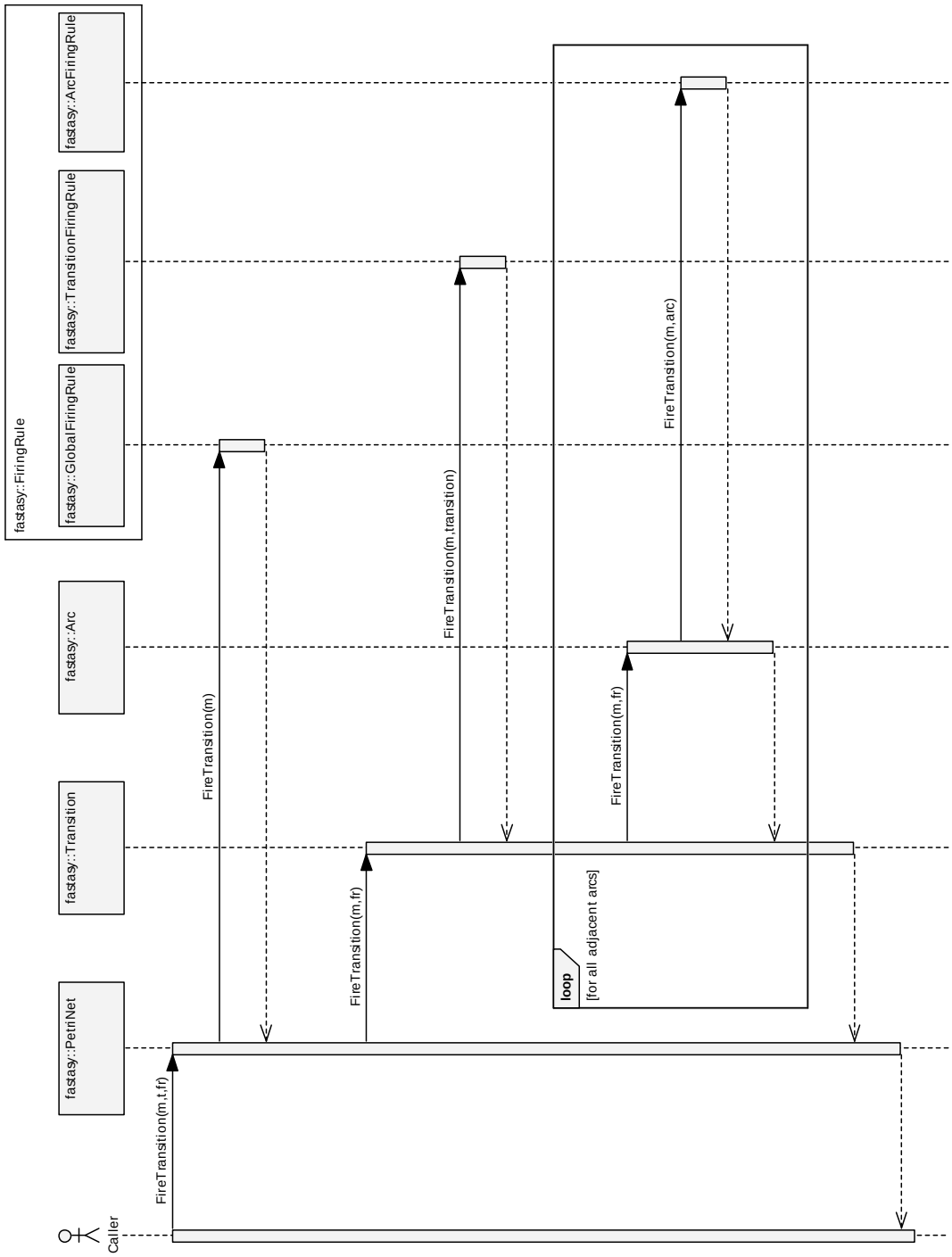



**Abbildung 37: Sequenzdiagramm FireTransition()**

#### 9.3.3.2.2 GetAdjacentPlaces()

Wie in 9.3.2.1 beschrieben, werden Abfragen nach Vor-, Nach-, Lesebereichen usw. mit Hilfe eines Prädikatsobjekts verallgemeinert. Hier wird das Aufrufschema von *GetAdjacentPlaces()* dargestellt, dasjenige von *GetAdjacentTransitions()* ist bis auf die Vertauschung der Rollen von Stellen und Transitionen identisch. Man beachte für die Lifeline von *ArctTypePredicate*, dass die Instanz **pred** ein als Parameter übergebenes, typischerweise konstantes, Objekt ist.

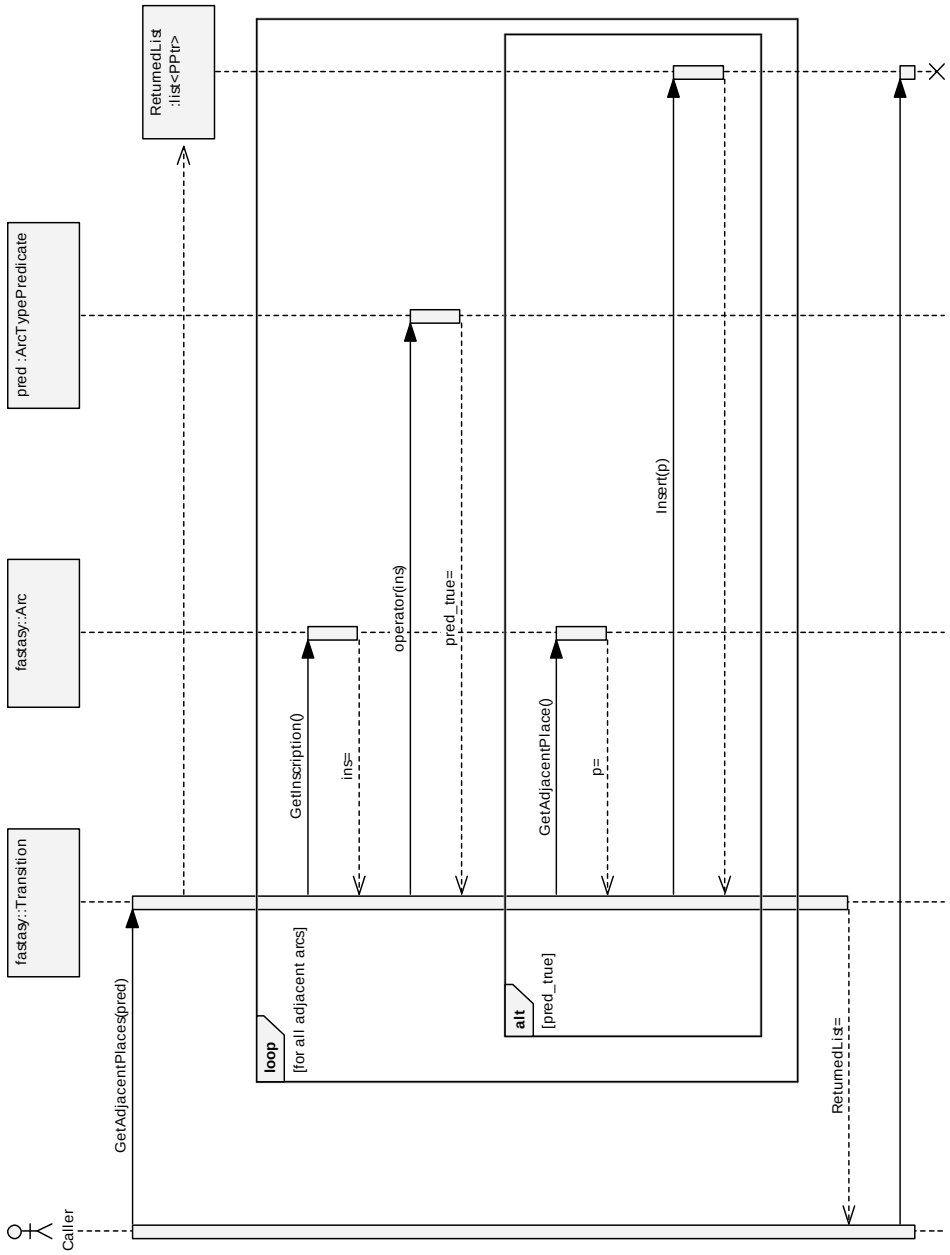



**Abbildung 38: Sequenzdiagramm GetAdjacentPlaces()**

#### 9.3.3.2.3 CreateInitialMarking()

Im letzten Diagramm für diese Schicht sei nun noch der in 9.3.2.4 motivierte Mechanismus zur Erstellung einer Anfangsmarkierung beschrieben, und zwar exemplarisch für die Ausprägung *PTTInitialMarkingRule*. Die Abläufe in der linken Swimlane sind dabei aber noch allgemein für jede Ausprägung gültig. Man sieht schön, wie konkrete Ausprägungen sich nur mit den für sie wichtigen Kontexten beschäftigen müssen, für *ApplyTransition()*, *ApplyArc()*und *ApplyGlobalData()* greifen hier nämlich einfach leere Default-Implementierungen.

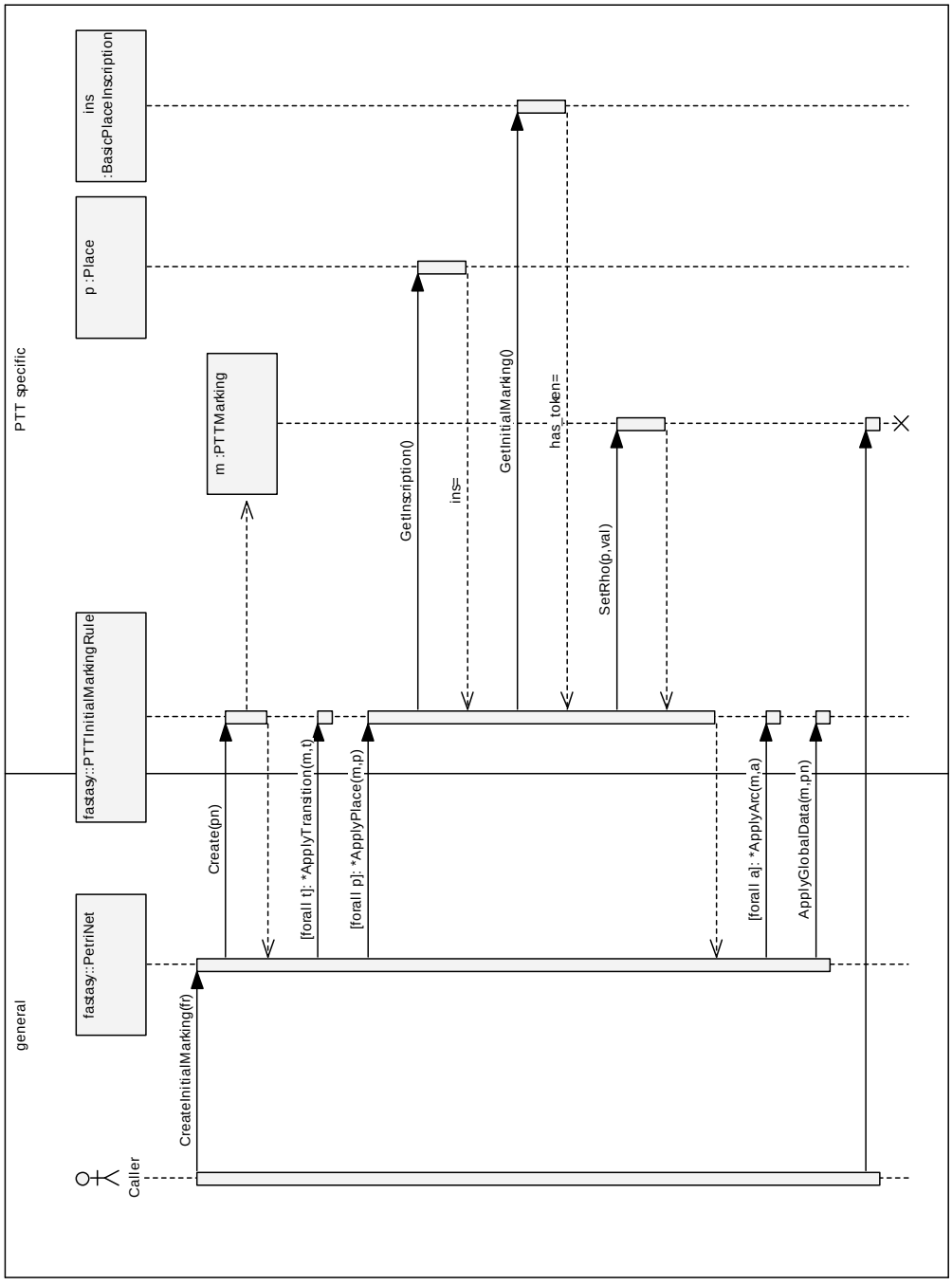



**Abbildung 39: Sequenzdiagramm CreateInitialMarking()**

## *9.4 Entwurf des Package „Transformation“*

Das Package „Transformation“ stellt die erste Gruppe von Komponenten dar, die sich innerhalb der „Bausteine“-Schicht befinden. In dessen Verantwortlichkeit gehören alle Berechnungsschritte, die den Erreichbarkeitsgraphen eines Quellsystems schrittweise in die Eingabe der Simulation transformieren. Obwohl das Interface *InductiveSystem* auch an anderer Stelle implementiert wird (etwa als Adapter zum Verhalten eines Quellsystems oder von *SimRevErrorSystem* als alternativer Zugang zu einer Variante des Fehlerpfads), ist *InductiveSystem* in diesem Package sozusagen zu Hause. Konkret implementieren alle Komponenten im Package „Transformation“ diese Schnittstelle und werden durch sie verbunden. Auch die Verbindung zum nächsten Package in der Verarbeitungskette, nämlich „Simulation“, erfolgt über dieses Interface.

### 9.4.1 Anforderungen

Wir haben ja an verschiedenen Stellen durch die Diskussion um *InductiveSystem* bereits Teile der Problemlösung in diesem Paket vorweggenommen. Insbesondere wurde mit den in 9.2.2 getroffenen Architekturentscheidungen bereits die Aufteilung der zentralen Komponenten dieses Package vorgegeben. Trotzdem wollen wir noch einmal zusammenfassen, welche Anforderungen uns zu der gewählten Lösung geführt haben.

#### *9.4.1.1 Durchführung der Berechnungen*

Mit den in 7.2 aufgeführten Vorgehensweisen sind wir in der dankbaren Situation, dass die rein funktionalen Anforderungen sich nicht nur automatisch aus der Theorie ergeben haben, sondern dort bereits in formalisierter Form vorliegen. Deswegen können wir diesen an sich zentralen Punkt unter Verweis auf 0 sehr knapp halten.

Die Berechnungsschritte im Verantwortungsbereich von „Transformation“ sind momentan:

- Elimination der λ-Kanten
- Indizierung von Zustandsmengen
- Konstruktion des Potenzautomaten eines zustandsindizierten Systems

Wir halten an dieser Stelle fest, dass es eines der obersten Ziele sein soll, die Implementierung neuer Berechnungsschritte, sowie deren Ver- und Austauschbarkeit möglichst einfach zu machen. Wir wünschen uns eine flexible Verschaltung von Bausteinen ganz im Sinne unserer Systemmetapher eines Experimentierkastens.

#### *9.4.1.2 Abstraktion von Schritten und Zuständen*

Wir werden es mit verschiedenen Arten von Zustands- und Schrittinformationen zu tun haben, und es ist damit zu rechnen, dass deren Anzahl in späteren Erweiterungen noch ansteigt.

Die einzelnen Berechnungsschritte sollen aber auf Details von Schritten und Zuständen nur soweit zurückgreifen, wie es unbedingt notwendig ist. Beispielsweise setzt die Komponente zur Elimination der λ-Kanten eine Operation *IsInternal()* auf den Schritten voraus, um feststellen zu können, ob es sich um einen internen Übergang handelt. Die

weiteren Details der Beschriftung sind dann aber irrelevant, wir setzen beispielsweise keine Operation zum Bestimmen von Aktionsnamen voraus.

Wir möchten weiter ausdrücklich die Möglichkeit haben, native Datentypen (insbesondere z.B. *int* als Ergebnis einer Indizierung) als Schritt- oder Zustandstypen zu verwenden, ohne diese in Objekte verpacken zu müssen (sog. *boxing*).

#### *9.4.1.3 Beförderung von Zwischen- und Seitenergebnissen*

Das Package soll selbst dafür verantwortlich sein, Zwischenergebnisse von einer Komponente zur nächsten zu befördern. Wir erwarten von dem entsprechenden Mechanismus, dass Ergebnisse der jeweiligen Berechnungsphasen auch zur nächsten Stufe gelangen können, ohne dass das komplette Ergebnis im Speicher aufgebaut wird. Eine explizite (zunächst transiente) Speicherung von Zwischenergebnissen soll jedoch trotzdem an jeder Stelle möglich sein. Weiter sollen solche Zwischenergebnisse dann aber auch persistent gemacht werden können.

Anders als Zwischenergebnisse sollen Seitenergebnisse (s. 9.2.3), wenn nötig, immer explizit gespeichert werden (in der Regel nur transient). Da die Benutzung von Seitenergebnissen immer ein Ansteigen des Kopplungsgrads zwischen Produzent und Konsument(en) bedingt, sind sie natürlich sowieso zu meiden.

#### *9.4.1.4 Extraktion von Traversierungs-Strategien*

Die genannten Berechnungsschritte haben alle eine auffallende Gemeinsamkeit: Obwohl sie in ihrer naiven Formulierung beschriftete Graphen als Ein- und Ausgabe haben, kann man die eigentlichen Berechnungsvorschriften jeweils wesentlich lokaler formulieren. Alle Schritte gehen nämlich derart vor, dass sie jeweils ausgehend von einem eigenen Zustand Nachfolger berechnen, diese als noch unvollständig kennzeichnen, und anschließend mit der Bearbeitung des nächsten unvollständigen Zustands fortfahren.

Mit anderen Worten stellt der Aufbau der Ausgabe in allen Fällen nichts anderes dar als die Traversierung eines bis dahin nur implizit (nämlich durch die Eingabe und eine Berechnungsvorschrift) gegebenen Transitionssystems dar. Wir wollen den Implementierungsaufwand für neue Berechnungsschritte niedrig halten, daher soll die Traversierungs-Strategie nicht mehr Teil derjenigen Komponenten sein, die die Berechnungsschritte durchführen.

Als zusätzliche, optionale Anforderung wünschen wir uns noch, dass notfalls auch Berechnungsschritte umgesetzt werden können, die einem anderen Schema folgen, d.h. keine Traversierung enthalten. Wie wir sehen werden, ist sogar dies möglich.

### 9.4.2 Evolution des Klassenverbunds

#### *9.4.2.1 Basisklasse InductiveSystem*

Wir haben ja bereits bei der Besprechung der Systemarchitektur ausführlich argumentiert, wie die Anforderungen 9.4.1.1, 9.4.1.3 und 9.4.1.4 durch ein gemeinsames Interface reflektiert werden können. Wir wollen nun zunächst festlegen, welche Operationen das Interface unterstützen soll. Auf einer abstrakten Ebene wissen wir dies natürlich, es handelt sich um:

- die Bestimmung des Anfangszustands

- die Bestimmung aller Übergänge und Folgezustände eines gegebenen Zustands.

Auf technischer Ebene müssen wir uns jedoch noch Gedanken machen, in welcher Form die zweite Operation ihr (mengenwertiges) Ergebnis zurückliefern soll. Gängige Alternativen hierzu sind:

- Bereitstellen einer Iterator-Schnittstelle
- Rückgabe eines Containers
- Füllen einer als Referenzparameter übergebenen Struktur

Bevor wir die Vor- und Nachteile der Varianten diskutieren, machen wir uns an zwei Beispielen kurz klar, dass die Reihenfolge der übergebenen Folgezustände bzw. der Übergänge dorthin aus Sicht der Effizienz durchaus eine Rolle spielen kann: Bei der Elimination von λ-Kanten ist es günstig, sortiert nach Kantenbeschriftungen vorgehen zu können, denn in diesem Fall finden wir, vorausgesetzt wir wählen ein geeignetes Sortierkriterium, alle λ-Übergänge in einem zusammenhängenden Bereich. Beim Aufbau einer Materialisierung hingegen kann man, wenn man die Folgezustände nach der Zustandsbeschriftung sortiert, den Effekt ausnutzen, dass mehrere Kanten zum selben Zustand direkt aufeinander folgen und man so nur einmal nach dem Zustand suchen bzw. ihn einfügen muss. Wir sehen also, dass wir gerne beide Möglichkeiten unterstützen würden.

Widmen wir uns aber nun den angesprochenen Varianten für einen Rückgabemechanismus:

- Eine Iterator-Schnittstelle müsste in jedem Fall dafür sorgen, dass ein angefordertes Ergebnis bis zum Ende der einzelnen Abrufe seine Gültigkeit behält. Insbesondere wäre es aufrufendem Code nicht möglich, einen Aufruf für einen weiteren Zustand zu starten, ohne das Ergebnis des vorausgehenden Aufrufs bereits vollständig bearbeitet zu haben. Da wir es in unserem „Bausteine“-Setting mit recht komplexen und in Bezug auf Erweiterungen unvorhersehbaren Kollaborationsmustern zu tun haben können, scheidet diese Möglichkeit aus.
  Auch verunreinigt eine Iterator-Schnittstelle natürlich das eigentliche Interface. Diesem Effekt könnte jedoch entgegengetreten werden, indem man das Iterator-Interface extrahiert und in den Stand einer eigenen Klasse erhebt, welche zurückgegeben wird. (Dies stellt eine gängige Technik dar, da das angesprochene Problem der Verunreinigung in Zusammenhang mit Iteratorschnittstellen ein allgemeines ist.) Die oben angesprochenen Probleme würden dadurch jedoch nicht gelöst.
  Letzten Endes würde die Komplikation der variablen Sortierung auch noch dazu führen, dass zwei verschiedene Iterator-Schnittstellen vorhanden wären.
- Bei der Rückgabe eines Containers haben wir diese Probleme nicht. Die Daten gehen mit dem Rücksprung aus der Operation in den Gültigkeitsbereich des aufrufenden Codes über. Die Schnittstelle von *InductiveSystem* wird nur mit einer einzigen Operation belastet, bzw. mit zweien, wenn wir verschiedene Sortierungen in verschiedenen Operationen zurückliefern.
  Ein deutlicher Nachteil gegenüber der obigen Lösung ist es, dass bei der Rückgabe ein Kopieren des Containers stattfindet. Die Rückgabe einer Referenz verbietet sich

natürlich, da ansonsten ja das referenzierte Objekt über die Methode hinaus gültig bleiben müsste.
- Das Umkopieren kann aber trotzdem vermieden werden, wenn der Container vom aufrufenden Code erzeugt und als Referenzparameter übergeben wird. Die Methode kann somit direkt diesen Container befüllen.

Aus dem gesagten wird klar, dass wir der Übergabe als Referenzparameter in diesem Fall klar den Vorzug geben. Allerdings haben wir immer noch das Problem, dass alle Klassen, die *InductiveSystem* implementieren, zwei sehr ähnliche Operationen implementieren müssten, nämlich eine für jede Sortierung der Rückgabe. Dies widerspräche jedoch der Forderung aus 9.4.1.1, die Erstellung neuer Bausteine so einfach wie möglich zu gestalten.

Ein erster Ansatz, beide Methoden auf eine einzige zu reduzieren wäre, schon in *InductiveSystem* zwei konkrete Methoden *GetFollowersByState* und *GetFollowersByStep* einzuführen, die sich auf eine einzige abstrakte Methode, etwa *GetUnsortedFollowers* abstützen. Letztere könnte dann etwa zunächst eine einfache Liste füllen, die ersten beiden Methoden könnten den Inhalt dann in entsprechend sortierte Container (in diesem Fall wohl *multimap* aus der STL, ein assoziativer Container, der auch mehrere Einträge mit identischem Schlüssel enthalten darf) umkopieren. Diese wohlbekannte Technik, konkrete Methoden auf abstrakte abzustützen und so allgemeine Funktionalität möglichst nahe in Richtung der Basisklasse zu schieben, wird von [GoF96] etwa als *Schablonenmethode* bezeichnet (eine gerade im englischen Sprachraum unglückliche Bezeichnung, da diese Technik rein gar nichts mit C++-Templates zu tun hat).

Wir wollen diesen Ansatz jedoch noch etwas verfeinern, da uns das unnötige Umkopieren aus einer Liste stört. Zu diesem Zweck ersetzen wir die Funktion *GetUnsortedFollowers* durch eine Prozedur *ComputeFollowers*, welche zusätzlich ein Funktionalobjekt *Inserter* übergeben bekommt. Dieses Funktionalobjekt wird nun von *ComputeFollowers* für jeden Übergang einmal aufgerufen, jeweils mit dem Übergang und dem Folgezustand als Parameter. Der *Inserter* ist dann dafür zuständig, die Struktur, welche *GetFollowersByState* bzw. *GetFollowersByStep* als Referenzparameter erhalten haben, zu befüllen, dabei übergeben die beiden Methoden natürlich gerade eben unterschiedliche *Inserter*, welche für die jeweilige Sortierung und damit für den Typ des Containers spezialisiert sind. (Gehen wir von einer *multimap* aus, haben wir es konkret entweder mit einer *multimap<Zustandstyp, Schritttyp>* oder aber *multimap<Schritttyp, Zustandstyp>* zu tun, d.h. Schlüssel und Wert tauschen jeweils die Rolle.)

In 9.4.1.2 haben wir ja auch noch weitestgehende Unabhängigkeit der Bausteine von den Ausprägungen der Schritte und Zustände gefordert. Nach dem in 9.2.2 gesagten sollte auch bereits klar sein, dass wir *InductiveSystem* parametrisch in den Typen von Schritten und Zuständen machen. (Man beachte, dass ein typischer, nicht spezialisierter Nachfolger von *InductiveSystem* damit schon in vier Typen parametrisch ist, nämlich in den Typen der eigenen Schritte und Zustände und denen des vom Vorgängersystem verwendeten.) Die Hauptvorzüge von Typpolymorphie gegenüber der Objektpolymorphie an dieser speziellen Stelle sind:

- Speziell für die Behandlung von nativen Datentypen ist es unter Effizienzgesichtspunkten vorteilhaft, nicht zu einer *single rooted hierarchy* und damit

zu *boxing* gezwungen zu sein. Objektpolymorphie würde dies jedoch bedingen (s. auch 6.1).

- Beim Einsatz von Objektpolymorphie wären wir gezwungen, für Operationen auf Schritten und Zuständen, die nur von einzelnen *InductiveSystem*-Nachfolgern benötigt werden, jedes Mal Untertypen der Schritte bzw. Zustände zu bilden. Insbesondere führt die Kombination von *n* solcher Operationen im schlimmsten Fall zu $2^n$ benötigten Unterklassen (und typischerweise zu Klassennamen der Kategorie *AAbleBAbleCAbleSomething*).

Es sei noch darauf hingewiesen, dass innerhalb des „Transformation"-Package eigentlich gar keine Senke vorgesehen ist, also eine Klasse, die letzten Endes die ganze Verarbeitungskette in Bewegung setzt. Wir werden noch sehen, dass *Automaton* diese Rolle zwar aus technischen Gründen übernehmen kann, jedoch meist trotzdem nicht am Ende der Kette steht. Im Normalfall besteht die eigentliche Senke tatsächlich aus einer Klasse, welche außerhalb des Packages beheimatet ist, nämlich der Klasse *GenericSimulation*.

Sehen wir uns nun abschließend kurz an, welche Operationen wir in *InductiveSystem* ohne Materialisierung *nicht* unterstützen können, und zwar einfach auf Grund der Tatsache, dass die Zustandsmenge nicht explizit vorliegt:

- *Keine Referenzen*. Da kein langlebiges Objektgeflecht für Knoten und Kanten des Graphen vorliegt, können wir nicht mit Referenzen arbeiten. Wir werden somit in jedem Fall Zustände und Schritte über ihre Werte identifizieren müssen, und mit aufrufendem Code muss immer das komplette Objekt ausgetauscht werden. (Dies bedingt, dass Zustands- und Schritttypen *leichtgewichtig* sein müssen; d.h. Objekte dieser Typen benötigen wenig Speicher und elementare Operationen darauf wie Vergleich oder Kopieren sind schnell durchführbar.)
- *Keine Prüfung auf Erreichbarkeit*. Ein *InductiveSystem* kann nicht entscheiden, ob ein Zustand erreichbar ist. Wird es nach den Nachfolgern eines nicht erreichbaren Zustands gefragt, ist das Verhalten im Prinzip undefiniert.
  (Bemerkung: Im Fall des Petrinetz-Adapters *PTTInductiveSystem* freilich wird die Antwort trotzdem der erwarteten entsprechen, da die Nachfolger einer nicht erreichbaren Markierung ja trotzdem durch die Schaltregel definiert sind. Dies gilt allerdings nicht, wenn eine Nachfolgemarkierung nicht mehr das Merkmal der Sicherheit erfüllt. In diesem Fall wäre eine *exception* die Folge, und zwar vom Typ *firing_rule_error*.)
- *Unbekannte Größe*. Schließlich ist auch die Größe der Zustandsmenge nicht bekannt.

#### *9.4.2.2 Materialisierungen*

Um Zwischenergebnisse wie angekündigt explizit materialisieren zu können, konstruieren wir zwei Klassen, nämlich *Automaton* und *LazyEvaluation*, die etwas unterschiedlich verwendet werden.

Bleiben wir zunächst bei *LazyEvaluation*. Dieser Nachfolger von *InductiveSystem* stellt in Bezug auf Typparameter genau dasjenige Interface wieder zur Verfügung, welches er von seinem Vorgänger anfordert. Außerdem liefern Anfragen an *LazyEvaluation* genau dieselben Ergebnisse, als wären sie seinem Vorgänger direkt gestellt worden. Allerdings werden

wiederholt gestellte Fragen ab dem zweiten Mal nicht mehr delegiert, sondern aus dem eigenen Speicher beantwortet. In Situationen, in denen mit wiederholten Anfragen zu rechnen ist, jedoch nicht aus anderen Gründen a priori eine komplette Materialisierung stattfinden muss, bietet *LazyEvaluation* gegenüber *Automaton* den Vorteil, dass nicht benötigte Teile des Ergebnisses auch nicht berechnet werden. (Arbeitet man ohne jegliche Zwischenstufe, ist dies natürlich ebenfalls der Fall.) Man beachte, dass es nicht sinnvoll erscheint, in *LazyEvaluation* einen echten Cache zu implementieren, d.h. einen begrenzten Speicher zu verwalten, aus dem Ergebnisse mit Hilfe einer Verdrängungsstrategie auch wieder entfernt werden. Es gibt nämlich gerade bei der Potenzautomatenkonstruktion, also bei der Stufe, die signifikant wiederholte Anfragen stellt, keinen Grund anzunehmen, dass diese in irgendeiner weise lokal gehäuft auftreten und damit eine sinnvolle Caching-Strategie ermöglichen.

Im Gegensatz zu *LazyEvaluation* traversiert nun *Automaton* sofort ausgehend vom Anfangszustand systematisch alle von dort erreichbaren Zustände und beantwortet anschließend jede Anfrage aus dem eigenen Speicher. Anders als die anderen Nachfolger von *InductiveSystem* enthält *Automaton* eine Methode *Build*, die mit dem Vorgänger als Parameter aufgerufen werden muss und diese Traversierung bewirkt. *Automaton* kann damit für den Fall, dass man gar nicht am Endergebnis der Verarbeitung (d.h. der Simulation) interessiert ist, als Senke (s.o.) verwendet werden. Anschließend kann der Zustand des Automaten ausgegeben oder abgespeichert werden.

Eine denkbare Alternative wäre es ja, die Traversierung direkt aus dem Konstruktor anzustoßen – schließlich nehmen andere *InductiveSystem*-Nachfahren ebenso ihre Vorgängersysteme im Konstruktor entgegen. Dies wollten wir jedoch aus mehreren Gründen vermeiden:

- Es ist wichtig, Instanzen von *Automaton* erstellen zu können, ohne einen Vorgänger zu benennen, denn einerseits gibt es mit *MutableAutomaton* einen Nachfolger, der über eigene Operationen explizit ohne Vorlage aufgebaut werden kann, und andererseits verfügt *Automaton* über die Möglichkeit, persistente Daten aus einem *stream* wiederherzustellen. (Notwendigkeit für letzteres ist lediglich die Anforderung an die Schritt- und Zustandstypen, sich selbst über *stream*-Operatoren korrekt zu serialisieren und wieder zu deserialisieren.)
- Durch Überladen des Konstruktors, so dass bei optionaler Angabe eines Vorgängers automatisch *Build* aufgerufen wird, könnte natürlich trotzdem ganz einfach das gewohnte Verhalten der anderen von *InductiveSystem* abgeleiteten Klassen ermöglicht werden. Der Autor teilt jedoch die verbreitete Auffassung, dass langwierige Operationen besser explizit angestoßen werden sollten und nicht in einen Konstruktor gehören. Andernfalls besteht die Gefahr, dass ein oberflächlicher Leser des Programmcodes nicht wahrnimmt, dass an einer bestimmten Stelle eine komplexe Verarbeitung stattfindet, was wiederum zu *semantischer Entfernung* (siehe 9.3.2.1) führt.
- In unserem Kontext entstünde außerdem die zusätzliche Komplikation, dass im Konstruktor *exceptions* auftreten können - eine Situation, die zwar technisch beherrschbar, aber unschön ist.

In 9.4.1.4 haben wir am Rande noch gefordert, dass auch Transformationen abbildbar sein sollen, welche nicht dem Muster der Traversierung eines implizit gegebenen Graphen entsprechen. Obwohl wir derartige Transformationen momentan nicht benötigen, können sie *notfalls* immer wie folgt implementiert werden: Von *Automaton* wird etwa eine Klasse *UglyTransformation* abgeleitet. Diese kann dann den internen Graphen von *Automaton* zunächst mit dem Vorgängersystem initialisieren, anschließend darauf beliebige Berechnungen ausführen (entweder *in situ* oder sie erzeugt einen zweiten Graphen), und zuletzt *ComputeFollowers* überschreiben, um das Berechnungsergebnis zu reflektieren. Natürlich verschenkt man so an dieser Stelle die Wahlmöglichkeit, Ergebnisse explizit darzustellen oder implizit zu lassen, aber immerhin würde sich selbst eine solche Komponente noch problemlos in das „Bausteine“-Muster einfügen.

#### *9.4.2.3 Typen von Schritten und Zuständen*

In 9.4.3.1 werden wir genau aufzählen, welche Typen von Schritten und Zuständen momentan in FastAsy existieren. Zuvor wollen wir uns hier aber kurz Gedanken machen, welche Anforderungen an die Typen zu stellen sind.

Zunächst werden wir diese Typen mit Sicherheit in Containern der STL verwenden wollen. Viele dieser Container (z.B. *set*, *map*, *multimap*) benötigen eine Ordnung auf ihren Elementen, die im einfachsten Fall durch den „<“-Operator gegeben ist. Diese Ordnung muss allerdings nicht total sein, es reicht, eine *strict weak order* zu haben, also eine Einteilung der Werte in Äquivalenzklassen, wobei die Klassen untereinander total geordnet, je zwei Elemente derselben Klasse jedoch immer unvergleichbar sind. Außerdem verlangen wir, dass durch den „=“-operator genau diese Äquivalenzrelation getestet wird.

Man wird sich nun fragen, warum wir nicht in jedem Fall einfach eine totale Ordnung angeben, zudem diese ja beliebig sein darf. Dazu ist zu sagen, dass wir uns an einer speziellen Stelle genau obige Tatsachen zu Nutze machen, nämlich bei *TaggedRTStep*. Dies ist ein Template, welches es erlaubt, zusätzlich zu der Schrittinformation Werte von parametrisierbarem Typ als Annotation mitzuführen. Annotation heißt in diesem Fall, dass diese Zusatzinformation bei Vergleichen eben nicht berücksichtigt wird. Schritte, die sich nur in der Annotation unterscheiden, sind also nicht identisch, aber äquivalent. (Eine Alternative zu dieser Form der Annotationen wird kurz in Kapitel 11 besprochen.)

Es ist *keine* generelle Voraussetzung, dass sich Schritt- und Zustandstypen in *streams* serialisieren und deserialisieren können. Lediglich wenn die I/O-Operationen von *Automaton* verwendet werden sollen, ist dies nötig.

Bei der Konstruktion des Potenzautomaten werden wir gleich feststellen, dass wir, je nachdem, ob wir Teilmengen von Verweigerungsmengen implizit lassen oder nicht (siehe 7.2.4), verschieden „tief“ in die Schrittinformation hineinblicken müssen. Wie wir sehen werden, reflektieren die Typanforderungen, die der Compiler prüft, genau den Grad dieses „Hineinblickens“.

In 9.5 und 9.6 werden wir noch sehen, dass natürlich spezielle Typen von Simulation und Retracing jeweils auch noch erweiterte Anforderungen an ihre Eingabetypen stellen werden. (Insbesondere bei der Simulation lohnt sich wiederum ein Blick darauf, wie ein Austausch der Simulationssemantik auch die Anforderungen an die Typen ändert.)

#### *9.4.2.4 Elimination von Lambda-Kanten*

Zur Elimination von nicht sichtbaren Übergängen konstruieren wir eine von *InductiveSystem* abgeleitete Komponente *ExternalInductiveSystem*. Diese ist generisch in den Typen von Zuständen und Schritten, verlangt aber von letzterem Typ das Vorhandensein einer Operation *IsInternal*. Eine denkbare Verallgemeinerung wäre an dieser Stelle, *ExternalInductiveSystem* noch mit einer Funktionalklasse zu parametrisieren, welche als Prädikat agiert und dabei bestimmt, ob ein Übergang als sichtbar gilt. Wir sehen dafür jedoch momentan keine Notwendigkeit, da die Eigenschaft eines Schritts, intern oder extern beobachtbar zu sein, nicht variabel ist. (Reizvoll wäre eine solcherart verallgemeinerte Lösung etwa dann, wenn wir elegant ein *hiding* durchführen wollten, indem wir lediglich das Prädikat austauschen.)

Während der Elimination der λ-Kanten müssen wir naturgemäß die λ-Hülle von Zuständen in Erfahrung bringen. Die Berechung der λ-Hülle haben wir in eine (polymorphe) Klasse mit Interface *AbstractLambdaHull* ausgelagert. Damit können wir erstens zwischen einer Variante mit eigenem Speicher (*CachingLambdaHull*) und ohne (*LambdaHull*) wählen. Zweitens könnte man an dieser Stelle auch mit verschiedenen Strategien zur Bestimmung der λ-Hülle experimentieren, so z.B. Implementierung als Matritzen-Multiplikation oder Berechnungen nach vorheriger Bestimmung des Strukturgraphen (siehe z.B. [VT96]). Es ist damit zu rechnen, dass sich die optimale Strategie stark unterscheidet, je nachdem, wie viel internes Verhalten ein System enthält. Unsere momentane Implementierung verwendet eine einfache Tiefensuche, welche vor allem für solche Erreichbarkeitsgraphen geeignet erscheint, bei denen die Projektion auf λ-Kanten in einem dünnen Graphen resultiert.

Wir haben weiter von der Version mit eigenem Speicher eine Variante implementiert, welche bei der Berechnung auch bereits vorhandene Ergebnisse *anderer* Knoten mit einzubezieht, wenn diese bei der Suche erreicht werden (*SmartCachingLambdaHull*). Diese letzte Variante erwies sich bei allen bisherigen Beispielen als den anderen beiden Implementierungen zumindest leicht überlegen (siehe Anhang C).

#### *9.4.2.5 Abstrakte Potenzautomatenkonstruktion*

Wie bereits erwähnt, unterscheidet sich die Potenzautomatenkonstruktion wesentlich, je nachdem, ob wir mit impliziten Teilmengen (siehe Teil I, Kapitel 2) in den Schritten arbeiten oder nicht:

- Stellen wir alle Verweigerungsmengen explizit dar, so müssen wir lediglich Kanten mit gleicher Beschriftung zusammenfassen. Die einzige Operation, die wir dazu auf Kantenbeschriftungen ausführen müssen, ist die Prüfung auf Gleichheit.
- Belassen wir aber Teilmengen, auf deren Existenz wir durch maximale Verweigerungsmengen schließen können, implizit, so müssen wir an dieser Stelle auch alle Schnittmengen von Verweigerungsmengen auf den Kanten berücksichtigen. Das genaue Verfahren wird in 2.2.5 geschildert, wichtig ist an dieser Stelle nur, dass wirklich auf die Verweigerungsmengen zugegriffen werden muss und wir Mengenoperationen auf diesen ausführen müssen. Wir verlieren also durch eine stärkere Zusicherung an den Typ der Schritte gegenüber der Version für explizite Verweigerungsmengen an Generizität.

Um nun die für die Konstruktion des Potenzautomaten zuständigen Klasse *DeterministicInductiveSystem* entsprechend zu generalisieren, versehen wir diese mit einem zusätzlichen Typparameter *TDetCombiner*, in welchem eine weitere Klasse angegeben werden muss, die als Funktional fungiert und (wohlgemerkt nur für einen festen Anfangszustand und ohne Kenntnis desselben) aus den Übergängen und Nachfolgezuständen des Vorgängersystems jeweils diejenigen des deterministischen Systems berechnet. In diesem Sinne ist die Konstruktion des Potenzautomaten damit abstrakt.

Wiederum wird der Compiler für uns prüfen, ob bei einer Instanzierung der Schritttyp des Vorgängersystems den Anforderungen der als *TDetCombiner* angegebenen Klasse gerecht wird. Konkret benötigt *SimpleStepCombiner* eben nur eine Prüfung auf Gleichheit, während *ImplicitStepCombiner* noch die Operationen *GetType* (Unterscheidung von Aktionen und Verweigerungsmengen) und *GetRefusalset* (Ermittlung der Verweigerungsmenge als *dynamic_bitset*) fordert.

Wir wollen an dieser Stelle noch eine rein technische Komplikation erwähnen: Der Zustandstyp im von *DeterministicInductiveSystem* implementierten Interface muss ja Mengen der Zustände des Vorgängersystems darstellen. Um dies noch effizient darstellen zu können, beschränken wir die Zustände des Vorgängers auf *TIntState*, also auf ganze Zahlen. Damit können wir mit *dynamic_bitset* einfach Bitvektoren für die Beschreibung der Zustände verwenden. Leider ist *dynamic_bitset* so ausgelegt, dass zwei Bitvektoren mit unterschiedlicher Länge nicht miteinander verglichen werden können. Dies mag darauf zurückzuführen sein, dass für das Design dieser Klasse ja das Template *bitset*<*N*> aus der STL Pate gestanden hat, und bei diesem sind *bitset*<$N_1$> und *bitset*<$N_2$> für $N_1 \neq N_2$ eben einfach unterschiedliche Klassen und somit unvergleichbar. Wäre die Semantik von *dynamic_bitset* so, dass bei einer Operation auf zwei unterschiedlich langen Bitvektoren die fehlenden Bits des kürzeren einfach implizit als nicht gesetzt angenommen würden, könnte *DeterministicInductiveSystem* die Bitvektoren einfach immer in der gerade nötigen Länge erstellen. So wie es ist, müssen wir aber die Länge des längsten Bitvektors, sprich: den höchsten vorkommenden Index und damit die Anzahl der Zustände des Vorgängersystems kennen.

Nun sollte man auf den ersten Blick denken, bei der Indizierung fiele doch genau diese Information an. Wir rufen uns jedoch in Erinnerung, dass die Indizierung als Instanz von *InductiveSystem* ja die Zustände erst *on demand* liefert und somit erst am Schluss eine definitive Angabe über den höchsten verwendeten Index liefern könnte. In der Tat wird es aber so sein, dass wir im Normalfall sinnvollerweise vor der Potenzautomatenkonstruktion eine Materialisierung durchführen – schließlich führt gerade dieser Verarbeitungsschritt zu vielen Wiederholungen von Anfragen an das Vorgängersystem. Somit erhalten wir ganz automatisch an der richtigen Stelle die Anzahl der Zustände. (Als zweite Möglichkeit ist im CLI aber wie erwähnt auch die Möglichkeit vorgesehen, explizit eine feste Länge der Vektoren vorzugeben. Diese muss natürlich mindestens der Anzahl der Zustände entsprechen.)

#### *9.4.2.6 Lokale Transformationen*

Es existiert ein spezieller Typ von *InductiveSystem*-Baustein, der bereits innerhalb unserer vorgefertigten Klassen zweimal auftritt (nämlich beim Indizieren von Zuständen und

dem Löschen von *Tags* aus Schritten), und für den es sehr wahrscheinlich ist, dass weitere Bausteine dieser Art hinzukommen werden. Die Rede ist von Transformationen, welche die Struktur des Transitionsgraphen komplett unangetastet lassen und nur Funktionen $f_{State}$ und $f_{Step}$ auf Zustände und Schritte anwenden. (Dabei können sich im Allgemeinen natürlich wieder die Typen ändern.) Essentiell ist dabei natürlich, dass $f_{State}$ injektiv ist, ansonsten wäre es unmöglich, die Frage nach den Nachfolgern eines gegebenen Zustands (der ja damit aus dem Bild von $f_{State}$ stammt) so umzusetzen, dass sie an das Vorgängersystem (welches nur Werte aus dem Urbild von $f_{State}$ akzeptiert) delegiert werden kann.

Zu diesem Zweck konstruieren wir ein Template namens *SystemTransformator*, welches über die Typen von Schritten und Zuständen hinaus noch mit zwei Typparametern *TStateTransform* und *TStepTransform* initialisiert wird. Bei der Instanzierung von *SystemTransformator* enthalten diese beiden Parameter Funktional-Klassen, deren „()“-Operator genau die gewünschte Transformation ausführt.

Denken wir nun an die Indizierung der Zustände (siehe 7.2.3), so bemerken wir, dass wir noch einen Mechanismus benötigen, um auf Seitenergebnisse, in diesem Fall die Tabelle der Indizes, später wieder zugreifen zu können. Dazu erweitern wir *SystemTransformator* noch um zwei Operationen *GetStateTransform* und *GetStepTransform*, welche die Instanzen der jeweiligen Funktionale zurückgeben. Da diese Instanzen für die Lebensdauer des *SystemTransformator*-Objekts dieselben bleiben, können sie die Seitenergebnisse in ihrem eigenen Zustand speichern und beliebige Methoden zur Verfügung stellen, um diese abzurufen.

#### *9.4.2.7 Umkehrung der Ablaufrichtung*

Eine weitere nahe liegende Umformung eines Transitionsgraphen ist es, die Richtung der Kanten umzudrehen. Bei einem endlichen Automaten müsste man sich freilich noch der Komplikationen um Anfangs- und Endzustand annehmen, um einen Automaten zu erhalten, der die Umkehrung der Sprache akzeptiert. Endzustände fließen ja aber in unsere Betrachtung gar nicht ein. Weiter sehen wir bei der Betrachtung der Anwendung (nämlich beim ersten Retracing-Schritt), dass wir den Startzustand eines Rückwärtslaufs sowieso sozusagen „per Hand“ festlegen müssen und gar nicht erwarten können, ihn aus dem Vorgängersystem zu erhalten.

Bei der Frage der Umsetzung einer solchen Umkehrung wäre die offensichtlichste Lösung, eine durch *Automaton* gegebene vollständige Materialisierung in irgendeiner Form als Eingabe einer Operation zu verwenden, die daraus einen weiteren *Automaton* erstellt und dabei die Kantenrichtungen umkehrt. (Offensichtlich muss natürlich irgendeine Form der Materialisierung vorliegen, eine reine Abstützung auf ein allgemeines *InductiveSystem* würde diese dann eben auf die eine oder andere Weise intern vornehmen.)

Als Alternative dazu speichern wir in einem Graphen, von dem wir wissen, dass wir ihn umkehren wollen, die Kanten schon in einer Form, mit der wir auch eingehende Kanten zu einem Zustand linear in der Anzahl dieser Kanten finden, nämlich mit einer zweiten Adjazenzliste für die Rückrichtung. Wir erstellen mit *ReversibleAutomaton* einen Nachfolger von *Automaton*, der so eine Operation *ComputeReverseFollowers* implementiert, welche in der gleichen Laufzeit wie *ComputeFollowers* arbeitet. Ein leichtgewichtiger Adapter *RevAutomatonSystem* schließlich implementiert nun das Interface von *InductiveSystem*, indem

er Aufrufe von *ComputeFollowers* ohne Änderung an *ComputeReverseFollowers* weiterreicht.

### 9.4.3 Ausgewählte UML-Dokumente

#### *9.4.3.1 Klassendiagramme*

#### 9.4.3.1.1 Induktive Systeme in „Transformation“

Das folgende Diagramm zeigt eine Übersicht über alle Nachfolger von *InductiveSystem*, die im Package „Transformation“ implementiert sind. (Auf die explizite graphische Darstellung der Aggregation zum Vorgängersystem wurde verzichtet.)

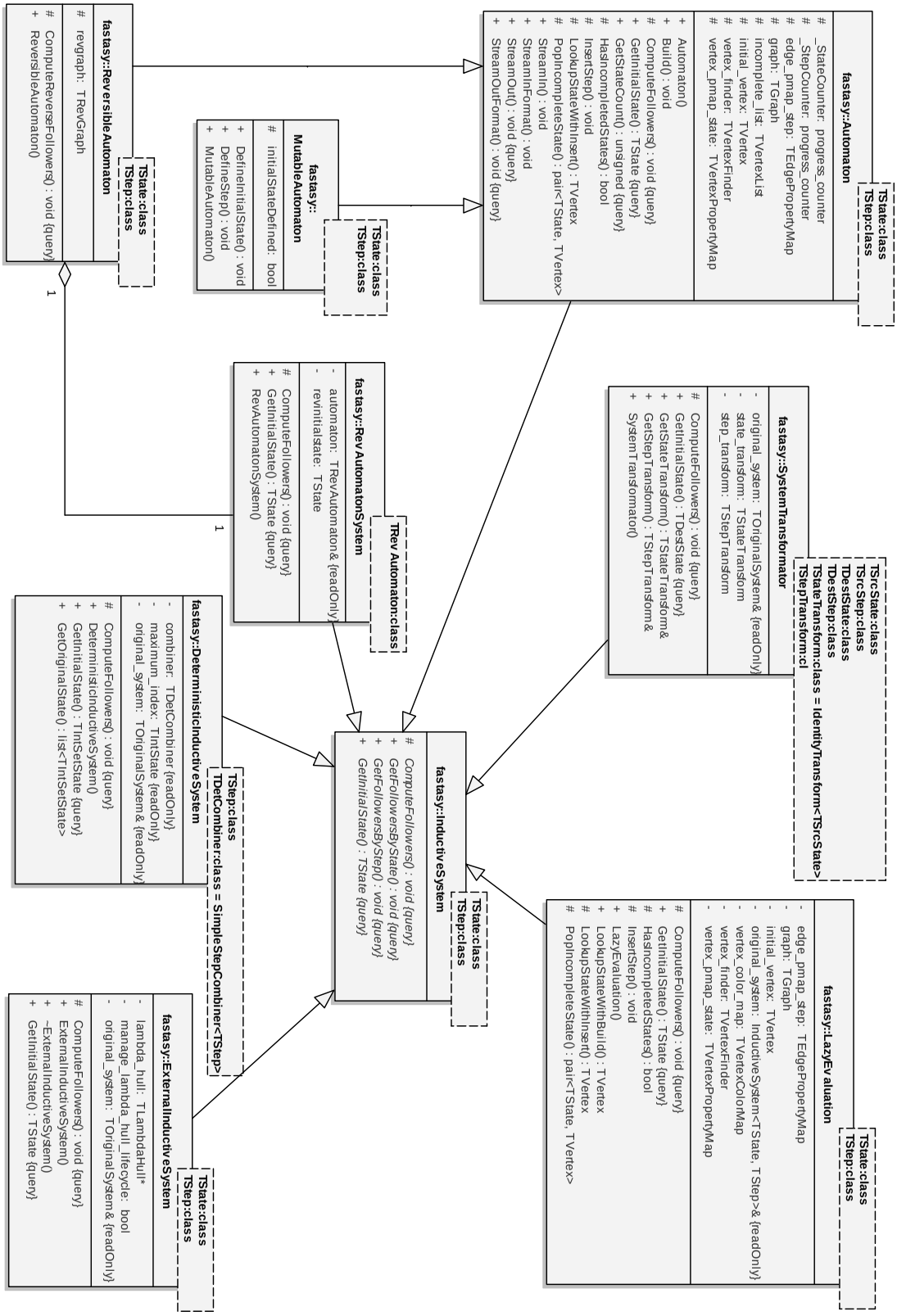



**Abbildung 40: Klassendiagramm InductiveSystems**

#### 9.4.3.1.2 Induktive Systeme und Inserter

Im folgenden ist *InductiveSystem* noch einmal isoliert dargestellt, um die Beziehung zu den *Inserter*-Klassen zu verdeutlichen.

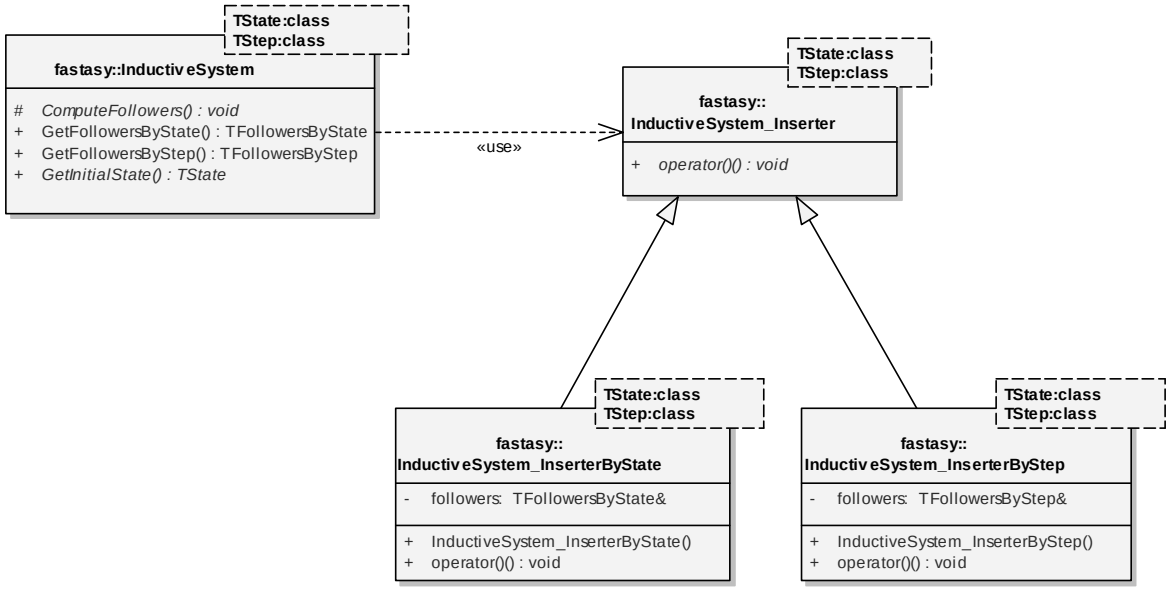



**Abbildung 41: Klassendiagramm InductiveSystem-Base**

#### 9.4.3.1.3 Ein-/Ausgabe von Automaten

Wir sehen im folgenden die Klassen, welche bei der Ein-/Ausgabe von Automaten zusammenarbeiten. Die einfachste, aber unflexibelste Stufe stellen die Operationen *StreamIn* und *StreamOut* dar, die noch rein innerhalb von *Automaton* abgearbeitet werden.

Als zweite Stufe kann der Operation *StreamOutFormat* eine Implementierung des Interface *AutomatonIOFormat* übergeben werden, welche genauer spezifiziert, wie die Ausgabe auszusehen hat. Eine noch feinere Möglichkeit der Kontrolle erhält man in *AutomatonDotFormat*, wenn man einen der Nichtstandard-Konstruktoren verwendet, um Spezialisierungen von *OutputDotState* und/oder *OutputDotStep* zu übergeben, welche dann noch die genaue Form der Ausgabe von Schritten und Zuständen festlegen (und dabei, wie etwa bei den Spezialisierungen für *RTState* und *RTStep*, weitere Daten aus dem spezifischen Kontext verwenden können).

Analog steht für das Einlesen benutzerdefinierter Format die Methode *StreamInFormat* zur Verfügung. Auf diese werden wir in Kapitel 10 noch sehr detailliert eingehen, wenn wir zu Demonstrationszwecken eine Klasse zum Einlesen von Ausgaben des PAFAS-Tools (s. [PFS]) entwerfen.

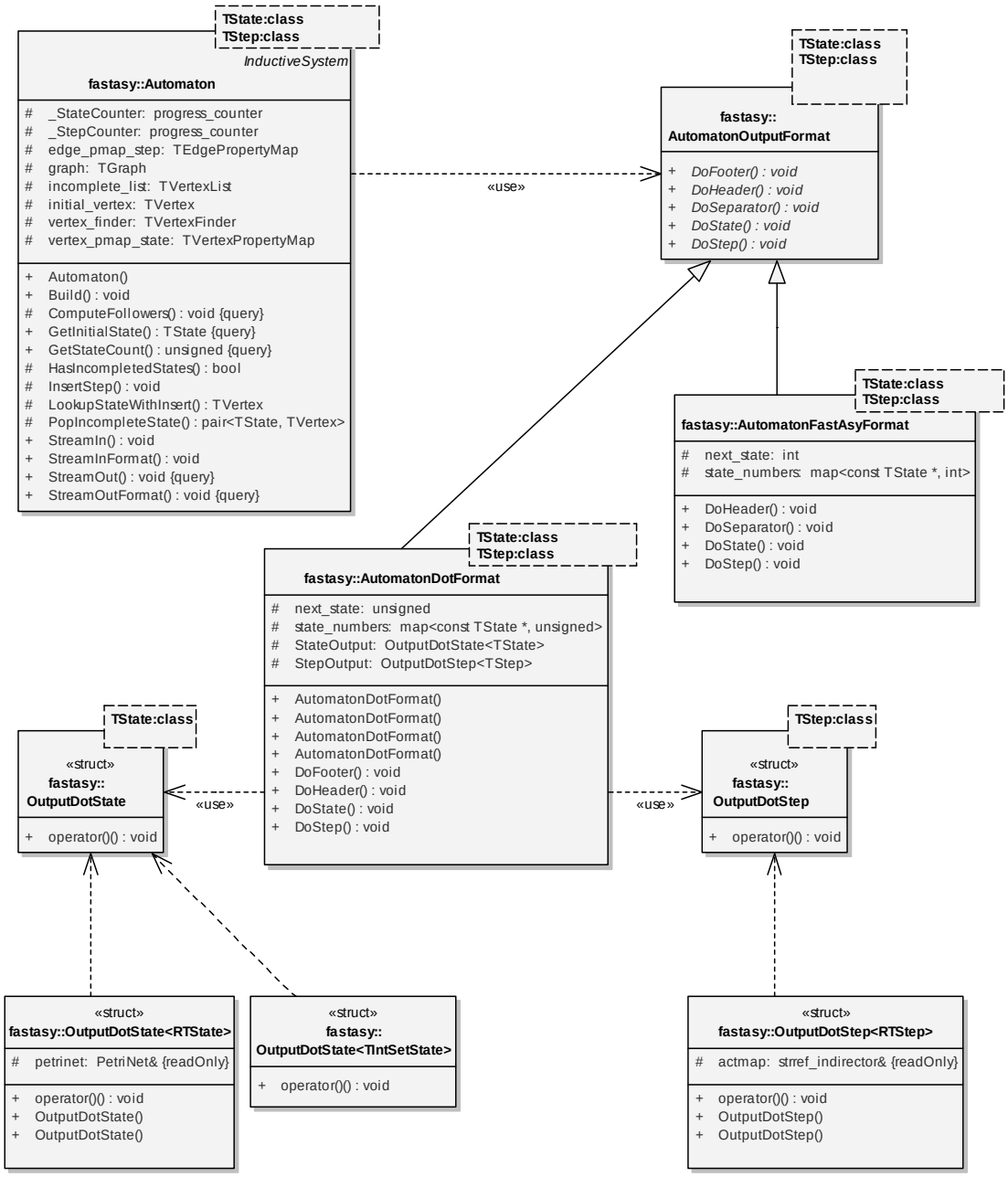



**Abbildung 42: Klassendiagramm Automaton-IO**

#### 9.4.3.1.4 ExternalInductiveSystem und Lambda-Hüllen

In diesem Diagramm finden wir die für die Elimination der λ-Übergänge verantwortliche Klasse *ExternalInductiveSystem* und die Beziehungen zum Interface *AbstractLambdaHull* und dessen Implementierungen, an welche die Berechnung der λ-Hüllen der Zustände delegiert wird.

Wie man weiter sieht, hält *AbstractLambdaHull* eine eigenständige Verbindung zu der Instanz von *InductiveSystem*, deren λ-Hülle berechnet wird. Dies macht Instanzen von *AbstractLambdaHull* universell einsetzbar, d.h. sie können auch ohne den Kontext von *ExternalInductiveSystem* operieren.

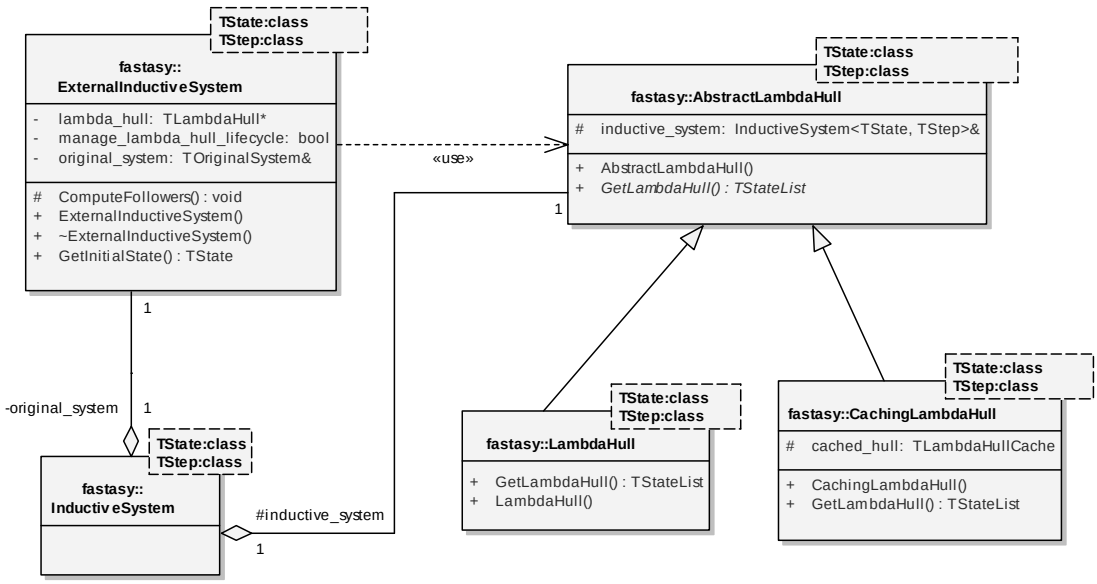



**Abbildung 43: Klassendiagramm ExternalInductiveSystem**

#### 9.4.3.1.5 DeterministicInductiveSystem

Hier sehen wir, wie *DeterministicInductiveSystem* mit *InductiveSystem* auf zwei verschiedene Arten verbunden ist und dabei den Typparameter *TState* beide Male verschieden bindet: Einerseits ist es von *InductiveSystem* abgeleitet, wobei *TState* den Wert *TIntSetState* hat (wir erinnern uns, dass Zustände im Potenzautomaten immer als Bitvektor codiert sind), andererseits verwendet es *InductiveSystem* als Vorgänger und bindet dabei *TState* an den Wert *TIntState* (was genau eine Notwendigkeit dafür ist, Multizustände als Bitvektoren ausdrücken zu können).

Wir sehen mit *SimpleStepCombiner* und *ImplicitStepCombiner* auch noch die zwei Typen, die momentan als Bindungen des Typparameters *TDetCombiner* in Frage kommen und dabei kontrollieren, in welcher Form die Übergänge des Vorgängersystems verknüpft werden, um die Übergänge aus einem Multizustand im Potenzautomaten zu erhalten.

Zur Notation sei kurz folgendes Bemerkt: Das Stereotyp „possible_bind“ ist eine private, regelkonforme UML-Erweiterung und stellt eine abkürzende Schreibweise dafür dar, eine (partielle) Spezialisierung einer parametrischen Klasse anzugeben und über die „bind“-Beziehung unter Angabe des Wertes, an den der Typparameter gebunden wird, mit der parametrischen Klasse zu verbinden. Wird die Spezialisierung selbst nämlich wie in diesem Fall nirgendwo im Diagramm verwendet, stellt ihre Angabe nur überflüssigen Ballast ohne zusätzlichen Informationsgehalt dar.

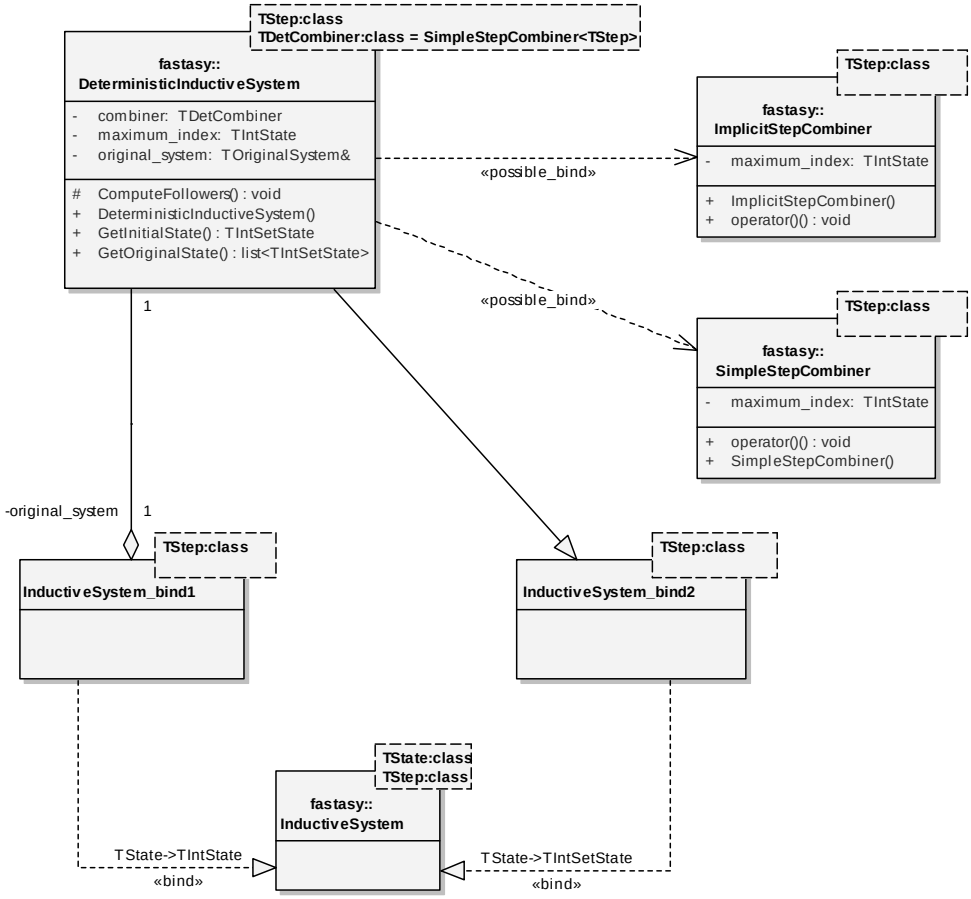



**Abbildung 44: Klassendiagramm DeterministicInductiveSystem**

#### 9.4.3.1.6 Zustands- und Schrittypen

Im folgenden sind diejenigen Typen, die momentan in FastAsy verwendet werden, dargestellt. Man beachte, dass *RTState* und *TaggedRTStep* beide aus technischen Gründen von *boost::totally_ordered* erben, nämlich weil diese Basisklasse die Semantik der Operatoren „==“ und „<“ korrekt auf alle weiteren Vergleichsoperatoren hochzieht. Während Objekte vom Typ *RTStep* nun in der Tat auch total geordnet sind, trifft dies jedoch, je nach Bindung des Typparameters *TStepTag*, auf *TaggedRTStep* nicht unbedingt zu. *boost::totally_ordered* arbeitet aber genauso korrekt, wenn „==“ und „<“ nur eine *strict weak order* induzieren.

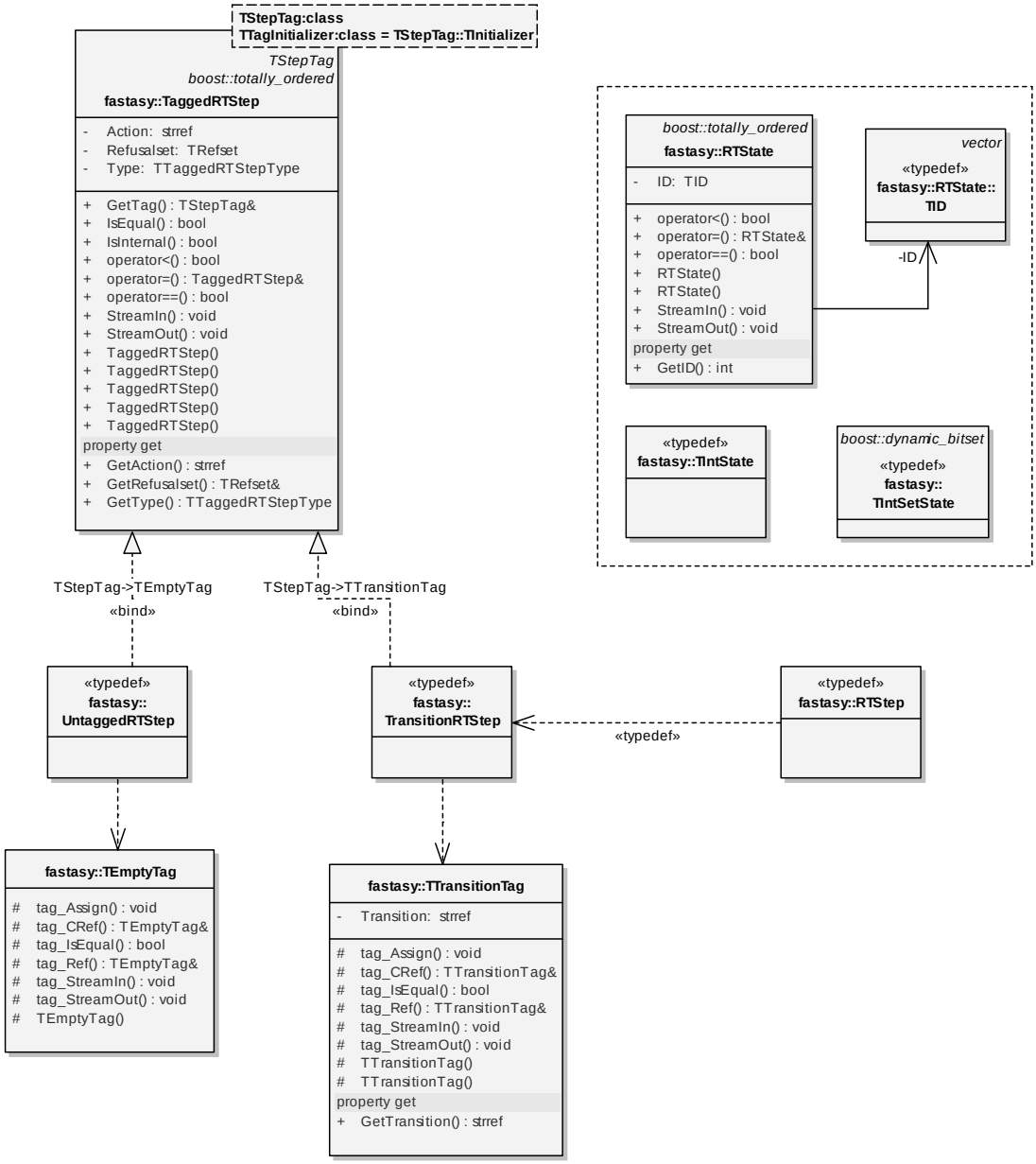



**Abbildung 45: Klassendiagramm Steps+States**

### *9.4.3.2 Kollaborationsmechanismen*

#### 9.4.3.2.1 Materialisierung durch Automaton-Objekt

Hier sehen wir das Aufrufschema, welches zum Aufbau einer Materialisierung führt. Sehr schön erkennt man an dieser Stelle, wie die Operation *Automaton::Build* genau jene extrahierte Traversierung enthält, von der in 9.4.1.4 die Rede war.

Im allgemeinen Fall können wir uns dabei vorstellen, dass *aSystem* die Aufrufe nicht sofort beantwortet, sondern wie in 9.2.2 dargestellt an ein weiteres Objekt mit einer Schnittstelle *InductiveSystem* weiterreicht.

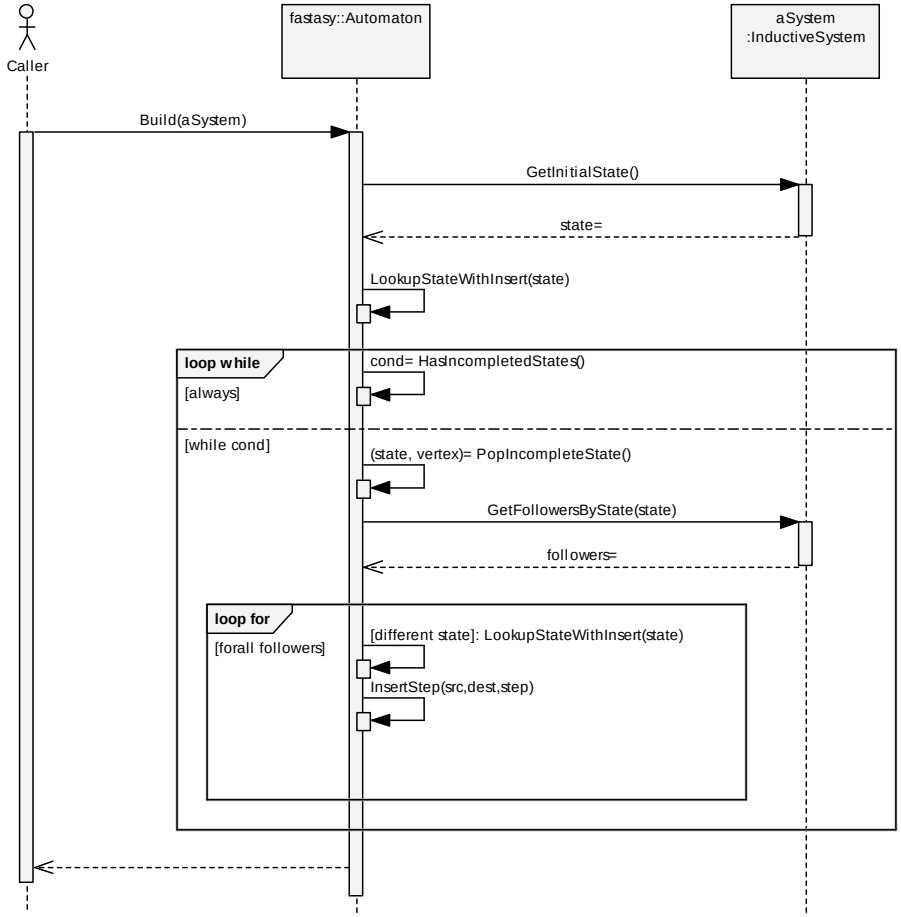



**Abbildung 46: Sequenzdiagramm Build Automaton**

#### 9.4.3.2.2 Ausgabe eines Automaten

Am Beispiel eines Automaten für Zustände des Typs *RTState* und Schritte des Typs *RTStep* sehen wir hier die Aufrufsequenz bei der Ausgabe als Datei für dot/Graphviz. Wie man sieht, ist hier der aufrufende Code für die Erstellung der atomaren Funktionale und der Initialisierung des komplexen *Strategieobjekts* (siehe das gleichnamige Entwurfsmuster in [GoF96]) verantwortlich. Dies ist notwendig, weil der Aufrufer u.U. zuerst die atomaren Funktionale mit Seitenergebnissen initialisieren muss.

In diesem konkreten Beispiel benötigt *OutputDotState<RTState>* eine Referenz auf das zu Grunde liegende Petrinetz, um Markierungen unter Angabe der Stellennamen auszugeben,

und *OutputDotStep<RTStep>* muss die (aus dem *PTTInductiveSystem*-Objekt bekannte) Codierung der Aktionsnamen kennen, um die Aktionen symbolisch darzustellen.

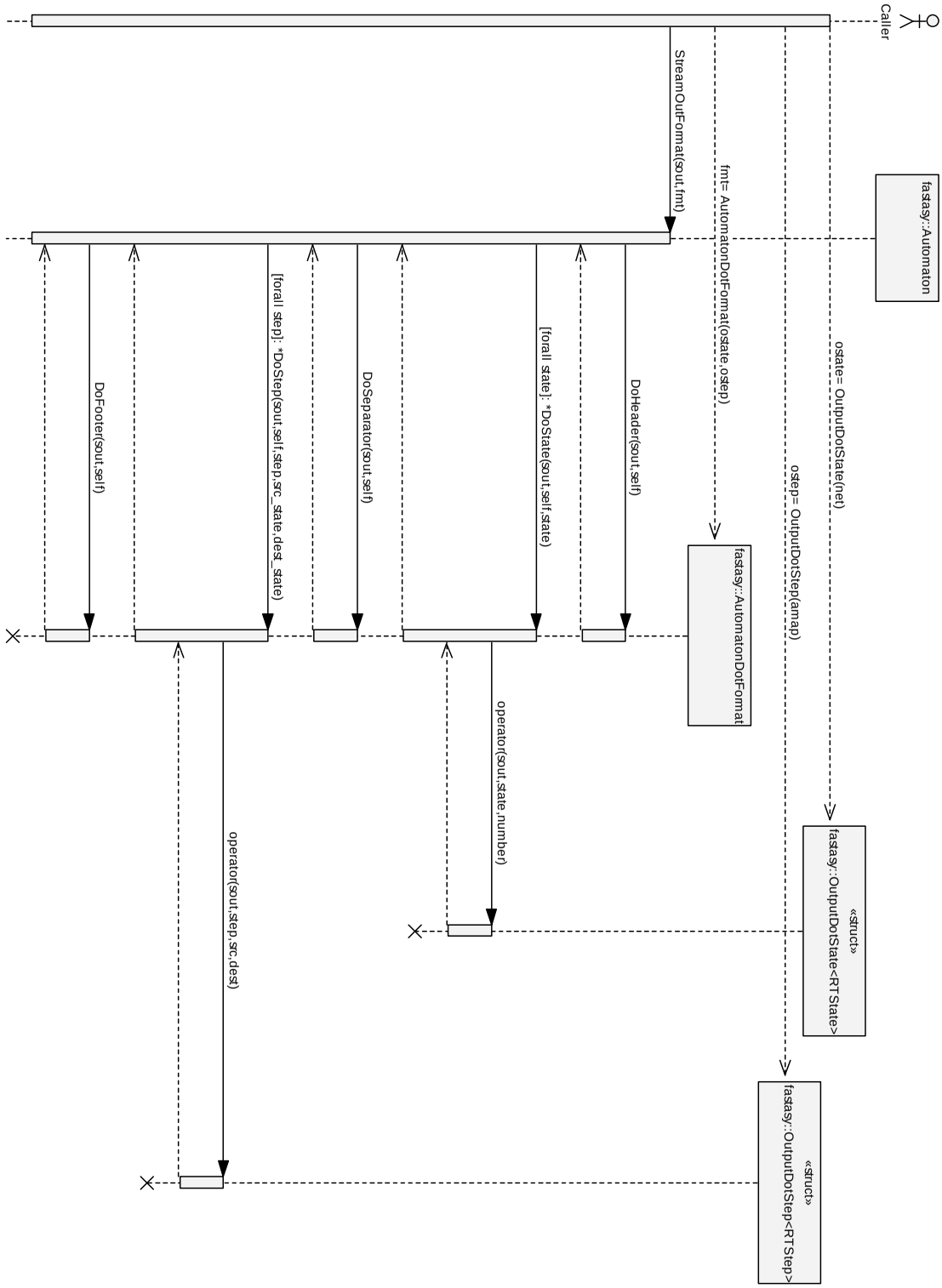



**Abbildung 47: Sequenzdiagramm Automaton-Output**

#### 9.4.3.2.3 Berechnung der Lambda-Hüllen in ExternalInductiveSystem

Im folgenden Sequenzdiagramm finden wir die Szenarien für die Berechnung des Startzustands eines *ExternalInductiveSystem*-Objekts (wobei die Lambda-Hülle keine Rolle spielt, siehe 7.2.2) und der Nachfolger eines gegebenen Zustands.

Wir sehen dabei, wie der Lebenszyklus des *LambdaHull*-Objekts wiederholte Aufrufe von *GetFollowerByStep* überdauert und so die Möglichkeit hat, ggf. seinen internen Zwischenspeicher aufzubauen. In diesem konkreten Beispiel hat sich der aufrufende Code für diejenige Konstruktorvariante von *ExternalInductiveSystem* entschieden, welche ihm die Verantwortung für den Lebenszyklus des *LambdaHull*-Objekts überlässt, so dass weitere Komponenten (hier dargestellt als sogenannte *Found-Message* aus *SomeCode*) dessen Zustand als Seitenergebnis nutzen könnten.

Man beachte, dass hier *GetFollowersByStep* verwendet wird und nicht *GetFollowersByState*. Diese Sortierung erlaubt uns, schnell die jeweils zusammenhängenden Blöcke mit internen bzw. sichtbaren Übergängen zu finden. Während wir mit dem ersten Aufruf die internen suchen, sind wir beim zweiten Aufruf nur an den sichtbaren interessiert. (Die Aufrufe an der ersten Stelle werden nämlich verwendet, um iterativ die λ-Hülle aufzubauen, mit dem zweiten Aufruf wird ein abschließender, sichtbarer Übergang angefügt; s. 7.4.4)

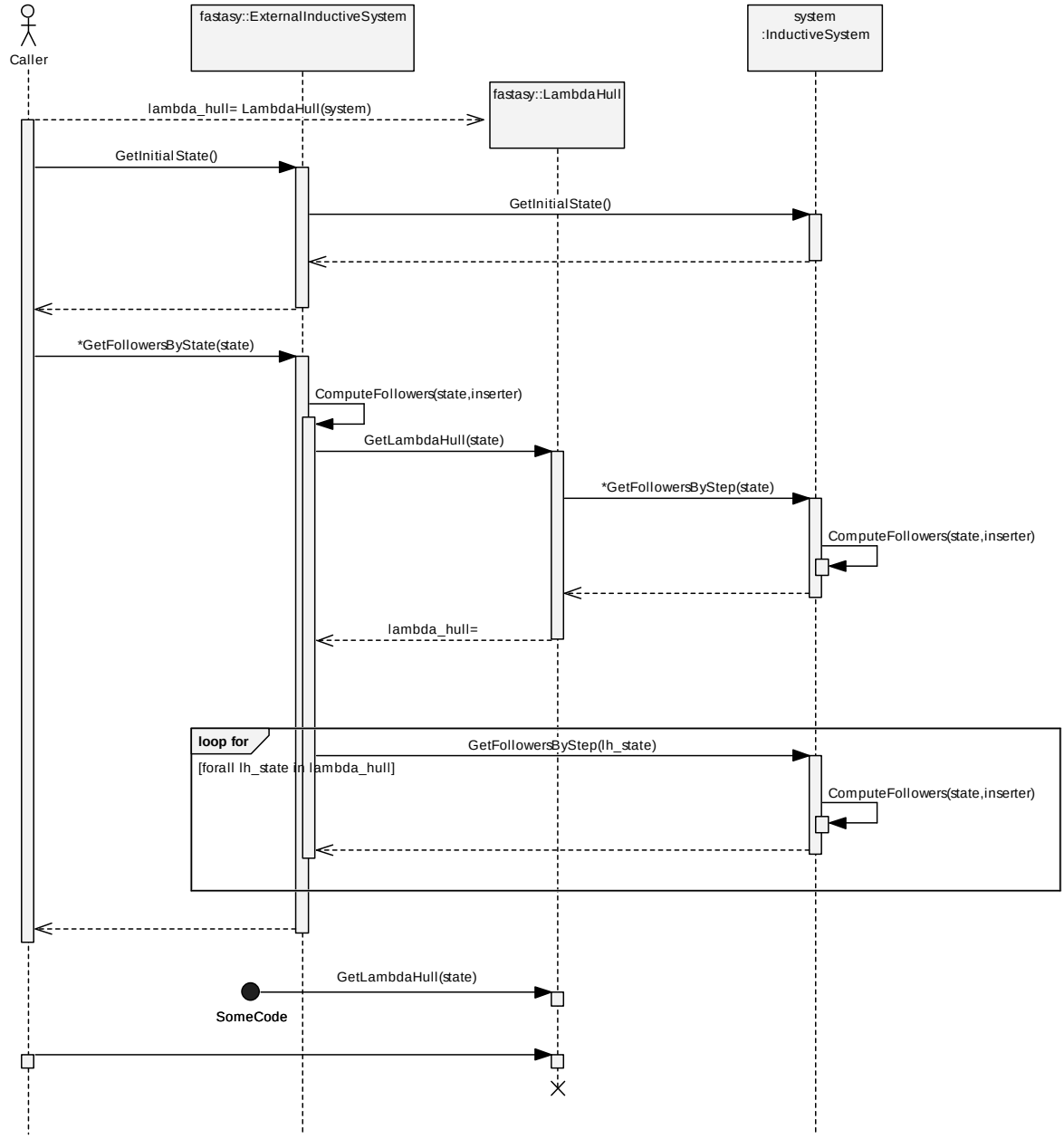



**Abbildung 48: Sequenzdiagramm LambdaHull**

#### 9.4.3.2.4 Parametrisierte Berechnung bei der Konstruktion des Potenzautomaten

Nachfolgend dargestellt ist die Aufrufkette für Anfragen an ein Objekt vom „Typ“ *DeterministicInductiveSystem* mit *SimpleStepCombiner* als Typparameter. Wir sehen, dass der Konstruktor des *DeterministicInductiveSystem* sich für seine gesamte Lebenszeit ein privates Funktional vom Typ *SimpleStepCombiner* erschafft und erst am Ende der eigenen Lebenszeit wieder zerstört. Man vergleiche den vorhergehenden Fall des *ExternalInductiveSystem*, in dem das dortige Funktional die Lebenszeit des Systems selbst überdauern durfte; der vorliegende Fall wird deshalb abweichend (und damit einfacher) behandelt, weil die *Combiner*-Objekte erstens typischerweise gar keinen Zustand haben und zweitens dieser für externen Code als Seitenergebnis uninteressant wäre.

Man beachte auch, dass dem *Combiner*-Funktional einfach der *Inserter* (siehe 9.4.2.1) des *DeterministicInductiveSystem*-Objekts durchgereicht wird, wodurch es resultierende Übergänge direkt in die als Referenzparameter übergebene Liste von *Caller* einfügen kann.

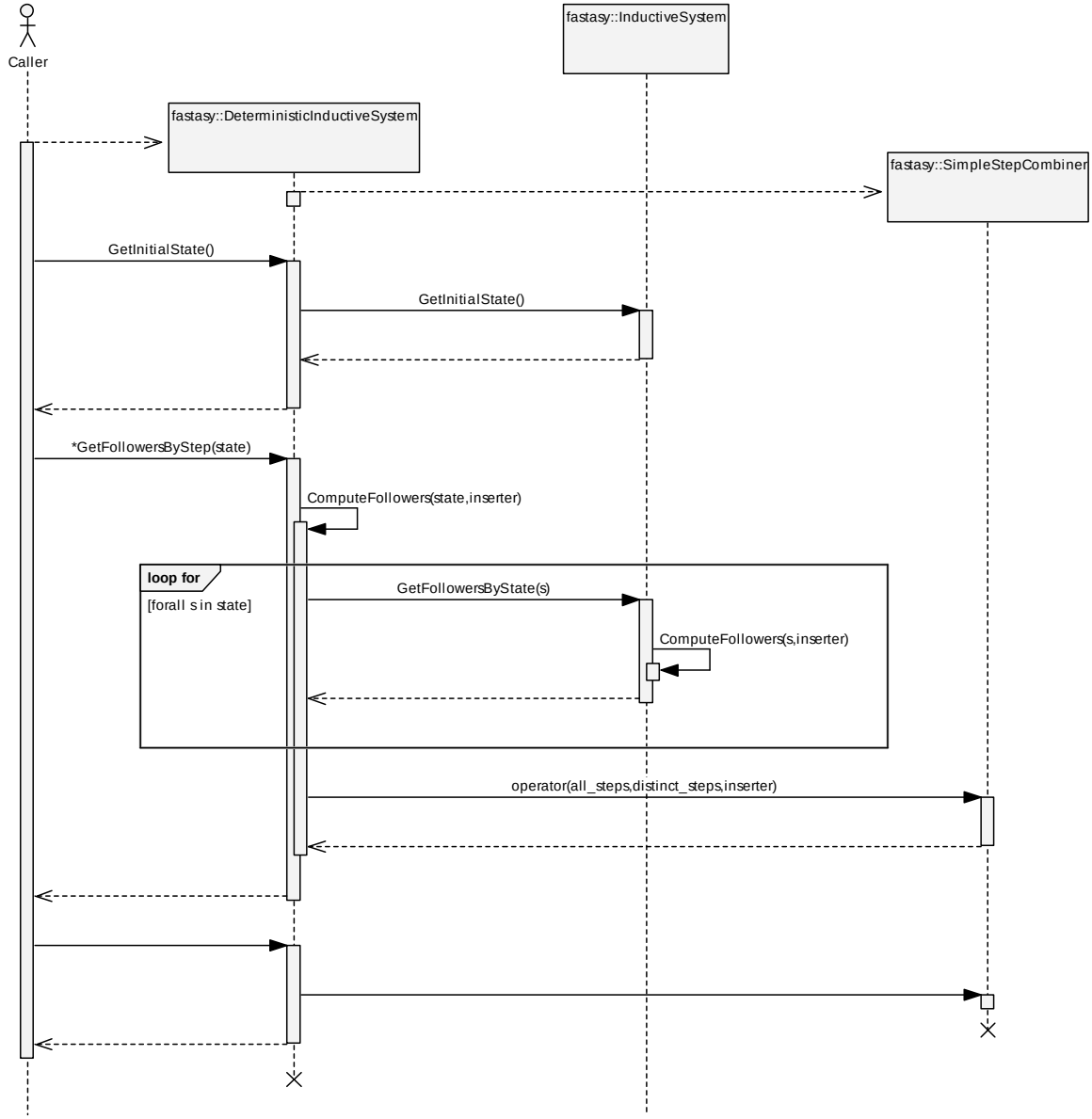



**Abbildung 49: Sequenzdiagramm PowerAutomaton**

## *9.5 Entwurf des Package „Simulation“*

Das Package „Simulation“ bildet insofern den Abschluss der Reihe von *InductiveSystem*-Komponenten, als dass die Simulationskomponente im Normalfall die Datensenke einer Kette von Bausteinen darstellt (genau genommen sogar von zwei Ketten). Zwar stellt die Simulation, wie wir sehen werden, selbst wieder ein *InductiveSystem*-Interface zur Verfügung (wenn auch extern über *SimRevErrorSystem*, s.u.), jedoch ist dessen Bedeutung sekundär und momentan eher von experimentellem Charakter.

Den Ausführungen zu diesem Package sei noch eine Begriffsklärung vorangestellt: Eine Simulation nimmt als Eingabe zwei Transitionssysteme $T_1$ und $T_2$ an und versucht, alle Schritte von $T_1$ mit Hilfe von $T_2$ zu simulieren. Wir nennen im folgenden nun $T_1$ die *Implementierung* und $T_2$ die *Spezifikation*, obwohl wir davon in vollem Sinne ja erst sprechen können, wenn tatsächlich eine Simulationsrelation existiert. Ständig von einer „potentiellen“ Implementierung bzw. Spezifikation zu sprechen erscheint uns aber schwerfällig, die Einführung der Begriffe „Simulant“ und „Simuland“ verbietet sich, obwohl zutreffend, auf Grund der phonetischen Ähnlichkeit. Wir machen uns kurz die wichtigsten Sachverhalte aus 7.3 an Hand der vorgenannten Begriffe klar: Existiert eine vollständige Simulation, so besitzt die Implementierung das kleinere, in unserem Kontext also *schnellere*, Verhalten. Schlägt die Simulation aber fehl, so existiert ein Verhalten der Implementierung, welches nicht in der Spezifikation vorgesehen ist, in unserem Fall also ein *langsames* Verhalten. Dieses langsame Verhalten findet seine Entsprechung im *Fehlerpfad*.

Im Falle von FastAsy werden wir $T_1$ und $T_2$ beide als deterministisch annehmen können.

### 9.5.1 Anforderungen

Wie schon in 9.4.1 ergeben sich die rein funktionalen Anforderungen natürlich wiederum aus der Theorie: Wir benötigen einen Mechanismus, der eine Vorwärtssimulation zwischen zwei Automaten durchführt (welche sinnvollerweise als *InductiveSystem*-Instanzen gegeben sein werden). Im Erfolgsfall werden wir kein weiteres Ergebnis benötigen, im Fehlerfall jedoch soll das Package eine Möglichkeit beinhalten, Informationen über den *Fehlerpfad* zu liefern.

Wie in 7.3 erwähnt und in 0 ausführlich dargestellt, hängt die Interpretation der Übergänge in den Automaten an dieser Stelle noch einmal entscheidend davon ab, ob man sich beim Aufbau des Automaten dafür entschieden hat, Teilmengen von Verweigerungsmengen explizit anzuführen oder nicht. Wir werden also auf jeden Fall eine Möglichkeit vorsehen müssen, das Verhalten der Simulations-Komponente dahingehend zu parametrisieren.

Hinsichtlich der Zustands- und Schritttypen gilt natürlich das im Zusammenhang mit dem Package „Transformation“ bereits gesagte. Wir möchten soweit wie möglich die Beschriftungen von Zuständen und Schritten transparent behandeln, wissen aber im speziellen Fall impliziter Verweigerungsmengen bereits, dass wir Mengenoperationen auf den Beschriftungen benötigen werden. Wir gehen so weit, auch unterschiedliche Schritt- und Zustandstypen für Implementierung und Spezifikation zuzulassen. Auf diese Weise wäre es sogar möglich, eine Simulationsrelation auf verschieden beschrifteten Automaten zu definieren.

## 9.5.2 Evolution des Klassenverbunds

### *9.5.2.1 Klasse GenericSimulation*

Wir beschließen, die Funktionalität zum Aufbau der Simulationsrelation, zu deren Speicherung und zur Gewinnung eines Rückwärtspfads von einem Fehlerelement aus innerhalb derselben Klasse zusammenzuführen. Die Kopplung zwischen diesen Verantwortlichkeiten ist ohnehin schon deshalb hoch, weil sie ja auf derselben Datenstruktur operieren.

Als Datenstruktur, welche die vollständige oder partielle Simulationsrelation aufnimmt, kommt zunächst natürlich eine Menge von Paaren in Frage. Wir sehen aber schnell, dass dies zur Gewinnung eines Rückwärtspfads nicht ausreichend ist, da zu einem Zustandspaar noch keine Information existiert, von welchem Vorgänger aus und mit welchem Übergang ein Paar von Zuständen beim Aufbau der Simulation erreicht wurde.

Wir reichern also die Struktur, in der die Simulationsrelation gespeichert wird, mit Verweisen auf das jeweilige Vorgängerpaar an. Beschriftet ist dieser Verweis mit dem entsprechenden Schritt der Implementierung. (Man beachte, dass dieser in Abhängigkeit von der beabsichtigten Verhaltensparametrisierung nicht identisch mit dem Übergang ist, mit dem die Spezifikation das Verhalten der Implementierung simuliert.) Außerdem erlauben wir für den Fall, dass keine vollständige Simulation existiert, zusätzlich zu den Paaren von Zuständen noch ein (eindeutiges) Fehlerelement, in dem lediglich der Zustand der Implementierung angegeben wird, der Zustand der Spezifikation aber undefiniert bleibt.

Durch diese Anreicherungen erhalten wir einen Graphen, der folgende Eigenschaften aufweist:

- Jeder Knoten hat höchstens eine ausgehende Kante (man beachte, dass die Kanten ja in Richtung des Vorgängers, also entgegen der Ablaufrichtung zeigen), jeder Knoten außer dem Startelement hat genau eine ausgehende Kante.
  (Genau genommen speichern wir etwas allgemeiner auch weitere ausgehende Kanten, jedoch in einer Form, dass wir uns diejenige Kante, über die ein Zustand das erste Mal entdeckt wurde, explizit merken. Dies ist jedoch lediglich ein Implementierungsdetail und zielt auf spätere Erweiterungen ab, so z.B. dem Auffinden kürzester Fehlerpfade im NDEA.)
- Von jedem Simulationszustand $q_S$ aus existiert ein eindeutiger Kantenzug zum Startelement (also dem Paar der Anfangszustände). Für das so durch die Kantenbeschriftungen erhaltene Wort w gilt immer, dass *Rev(w)* in der Sprache der Implementierung enthalten ist.
- Für den Fall der klassischen Simulation (d.h. es werden identische Kantenbeschriftungen bei der Simulation eines Schritts gefordert) gilt noch: Dreht man alle Kantenrichtungen um (und entfernt ggf. das Fehlerelement), erhält man einen Ausschnitt des Produktautomaten der beiden Eingabesysteme.
  (Für andere Fälle kann man für entsprechend verallgemeinerte Varianten der Definition des Produktautomaten zum selben Resultat kommen.)

Der zweite Punkt bedeutet nichts anderes, als dass wir im Falle einer nur partiellen Simulation den Fehlerpfad finden, indem wir vom Fehlerelement aus rückwärts den Verweisen folgen und die entstehende Sequenz umkehren. Wir werden jedoch beim *Retracing*

feststellen, dass genau die erste Stufe *NondetRetracer* (siehe 9.6.2.8) in natürlicher Weise auf der Umkehrung des Fehlerpfads operiert, nicht auf dem Fehlerpfad selbst. Aus diesem Grund stellt *GenericSimulation* als Ergebnis auch diese Umkehrung bereit und führt selbst nicht die Umsetzung in den vorwärts laufenden Fehlerpfad durch.

Für die Speicherung des Graphen verwenden wir wie in 6.2.2 vorweggenommen die *Boost Graph Library* und die dort definierte *adjacency_list<…>*. Da die mit Abstand häufigste Operation auf diesem Graphen das Einfügen neuer Zustände und Kanten ist, wird in diesem Fall a*djacency_list* so konfiguriert, dass Zustände und Adjazenzen als *std::list* gespeichert werden und nicht als *std::vector*.

Es bleibt die Frage zu klären, in welcher Form der umgekehrte Fehlerpfad aufrufendem Code mitgeteilt wird. Dazu existieren drei verschiedene Ansätze, die alle implementiert sind:

- Es existieren zwei Methoden *GetErrorState()* und *GetErrorPredecessor()*, welche direkten Zugriff auf den Fehlerpfad erlauben. Da *GetErrorPredecessor()* den Vorgänger eines übergebenen Zustands (und nicht automatisch des letzten zurückgegebenen Zustands) angibt, haben wir nicht das in 9.4.2.1 beschriebene Problem, dass eine Iteratorposition innerhalb des Objekts gespeichert wird und somit ineinander greifende Abfragen verhindern würde (oder besser deren Ergebnis falsch werden ließe).
  Trotzdem betrachten wir diese Form der Abfrage als eher unelegant und ziehen die beiden folgenden Alternativen vor. Wir haben auch erwogen, die beiden Methoden *protected* zu deklarieren, uns jedoch dagegen entschieden, weil ihre öffentliche Sichtbarkeit kein Aufbrechen der Kapselung zur Folge hat und die beiden Klassen *SimRevErrorSystem* und *SimRevErrorSequence* (s.u.) ansonsten notwendigerweise als *friend* deklariert werden müssten.
- Es existieren zwei getrennte Zugreiferklassen *SimRevErrorSystem* und *SimRevErrorSequence*, die den Fehlerpfad komfortabel nach außen zur Verfügung stellen. (Unter einer *Zugreiferklasse*, engl. *accessor class*, verstehen wir eine Klasse, die den Zugriff auf bestimmte Daten einer anderen Klasse kapselt. Prominentestes Beispiel ist diejenige Zugreiferklasse, die die STL als Ergebnis von *vector<bool>::operator[]* zurückliefert. Wir verwenden absichtlich nicht den Begriff *Zugriffsklasse*, da dieser im Kontext von C++ bereits anderweitig eine feste Bedeutung hat.) In gewissem Sinne kann man beide Klassen natürlich auch wiederum als Beispiele für das Entwurfsmuster *Adapter* verstehen, wobei eine Besonderheit darin besteht, dass hier nur ein sehr eingeschränkter Teil der Aufgaben der ursprünglichen Klasse auf eine gegebene Schnittstelle umgesetzt wird, nämlich die Sicht auf die Simulation als Fehlerpfad. Beide Klassen sind weiter unten gesondert beschrieben.

Der Aufbau der Simulationsrelation wird bei *GenericSimulation* nun durch eine Methode *Build()* angestoßen. Diese fügt nun zunächst das initiale Element, bestehend aus einem Objekt der Klasse *SimState* (s.u.), in welchem die Anfangszustände der beiden Systeme vermerkt sind, ein. Ausgehend davon traversiert *Build()* nun in der üblichen Weise den implizit gegebenen Ausschnitt des Produktautomaten. Dabei wurde das Verhalten, welches für einen gegebenen Schritt den Zielzustand der Spezifikation ermittelt, in eine eigene Klasse extrahiert (s. 9.5.2.2).

In der Tat sind die Gemeinsamkeiten zur Methode *Build()* in *Automaton* mehr als oberflächlich, so dass durchaus in Betracht gezogen werden könnte, den gesamten Simulationsprozess als *InductiveSystem* zu formulieren und tatsächlich einem *Automaton* diese Traversierung zu überlassen. Letzten Endes haben aber technische Details den Ausschlag gegeben, dies trotz der offensichtlichen Eleganz einer solchen Lösung anders zu realisieren: Das *InductiveSystem*, welches einem *Automaton* als Vorlage dient, hat zum einen keine Möglichkeit, die Traversierung vorzeitig zu beenden; im Falle einer fehlgeschlagenen Simulation ist dies jedoch notwendig (oder zumindest im Hinblick auf Effizienz sehr wünschenswert). Es schien nun aber ziemlich unnatürlich, die Schnittstelle von *InductiveSystem* dahingehend zu erweitern, etwa indem die *GetFollowers*-Methoden mit einem zusätzlichen Flag als Rückgabe ausgestattet werden. Außerdem ist *Automaton* nicht in der Lage, im Falle eines solchen Abbruchs das Fehlerelement zu speichern. Zu guter Letzt enthielte das *Automaton*-Objekt als Auschnitt des Produktautomaten ja vorwärtsgerichtete Verweise, die wir erst wieder umkehren müssten. So wurde *GenericSimulation* also als von *Automaton* völlig unabhängige Klasse realisiert, zum Verständnis des Quelltextes kann aber durchaus beitragen, sich die Gemeinsamkeiten vor Augen zu halten.

#### *9.5.2.2 Parametrisierung durch Simulatoren*

Hinsichtlich der Parametrisierung des Verhaltens bietet es sich wiederum an, wie schon angedeutet eine *Strategieklasse* als Typparameter einzuführen, so dass *GenericSimulation* die Details der Simulation eines einzelnen Schritts an die jeweilige Klasse delegiert.

Konkret wird die *Strategieklasse*, im folgenden auch *Simulator* genannt, am Beginn der Bearbeitung eines Zustandspaares der Simulation mit allen Übergängen, die in der Spezifikation in diesem Zustand möglich sind, initialisiert. Diese Übergänge stellen sozusagen die Auswahlliste dar, aus der der Simulator dann für jeden einzelnen Übergang der Implementierung den passenden Übergang auswählen und dessen Zielzustand zurückgeben muss. Dieser Rückgabewert ist jedoch optional. (Technisch ist dies durch Verwendung des *optional*<>-Templates aus *boost* gelöst.) Findet der Simulator nämlich keinen solchen Übergang, wird durch das Fehlen eines Rückgabewerts dem Aufrufer signalisiert, dass die Spezifikation keinen entsprechenden Übergang mehr durchführen kann und so der Fehlerzustand erreicht ist.

Diese Technik, den Simulator für jeden erreichten Zustand der Simulation einmal mit den Übergängen zu initialisieren, ist der offensichtlichen Alternative, nämlich dem Simulator einmal eine Referenz auf das Spezifikationssystem zu übergeben, deshalb überlegen, weil letzteres zur Folge hätte, dass der Simulator sich wiederholt die Übergänge für denselben Zustand verschaffen müsste. Wird an späterer Stelle erneut ein Zustandspaar mit demselben Zustand in der Spezifikation erreicht, muss der Simulator freilich trotzdem die Übergänge neu ermitteln, wir speichern ja nichts über erneute Aufrufe von *InitState()* hinweg.

Die Anforderungen an die Typen von Schritten und Zuständen werden im Übrigen einzig durch die jeweiligen Simulatoren festgelegt, denn nur diese sind ggf. darauf angewiesen, Schritte oder Zustände genauer zu untersuchen.

Man beachte zuletzt, dass sich die Funktion von *GenericSimulation* durch Einführung eines zweiten Simulators auch problemlos auf Bisimulation erweitern ließe.

#### *9.5.2.3 Klasse SimState*

*SimState* ist eine einfache Klasse, welche die Details der Speicherung eines Zustandspaares in der Simulation kapselt: Es bietet die üblichen Vergleichsoperatoren (bezüglich einer von den beiden einzelnen Zustandstypen abgeleiteten Ordnung), ermöglicht (im Gegensatz zu *std::pair*) den benannten Zugriff auf die einzelnen Komponenten und enthält eine Abfrage, ob es sich um das Fehlerelement handelt.

(Am Rande sei bemerkt, dass auch der Typname *SimStep* verwendet wird, jedoch nur ein *typedef* für den Schritttyp der Implementierung ist.)

#### *9.5.2.4 Zugreiferklasse SimRevErrorSystem*

Als eine von den beiden erwähnten Zugreiferklassen stellt *SimRevErrorSystem* den aus einer nur partiellen Simulation erhaltenen umgekehrten Fehlerpfad über das *InductiveSystem*-Interface zur Verfügung. Die Schrittypen entsprechen dabei denen der Implementierungsseite, die Zustände jedoch sind vom Typ *SimState*. Obwohl wir eigentlich auch nur an der Zustandsinformation der Implementierung interessiert sind, überlegt man sich schnell, dass es nicht möglich ist, ein entsprechendes *InductiveSystem* zu konstruieren: Die Nachfolger für einen gegebenen Zustand der Implementierung wären nämlich im Allgemeinen nicht eindeutig, schließlich können mehrere Simulationselemente existieren, die in der ersten Komponente identisch sind.

#### *9.5.2.5 Zugreiferklasse SimRevErrorSequence*

Das von *SimRevErrorSystem* zur Verfügung gestellte *InductiveSystem* erscheint vielleicht aus zwei Gründen ein wenig künstlich: Zum einen scheint das Interface etwas aufgebläht, da ja die Zustände im umgekehrten Fehlerpfad nur immer höchstens einen Nachfolger haben. Zum anderen aber trägt *SimState* in diesem Zusammenhang zu viel Information, indem der Zustand der Spezifikation mitgeschleppt werden muss, um den Zustand innerhalb des Fehlerpfads eindeutig zu machen. Letzteres mag zunächst nicht als gravierender Nachteil erscheinen, aber man mache sich bewusst, dass dieses Zuviel an Information sich durch sämtliche Stufen des Retracing zöge und dort die Dinge unnötig und vor allem auch unnatürlich verkomplizieren würde.

Aus diesen Gründen führen wir ja beim Retracing die linearen Systeme ein, und es bietet sich an, den umgekehrten Fehlerpfad bereits in dieser Form zur Verfügung zu stellen. *SimRevErrorSequence* implementiert also *ForwardSequence* (wie wir in 9.6.2.1 sehen werden die grundlegende Schnittstelle linearer Systeme; man beachte, dass „forward“ dabei auf die Analogie zu Vorwärtsiteratoren aus der STL verweist und nichts über die Ablaufrichtung des Fehlerpfades aussagt.). Damit ist einerseits die Zusicherung umgesetzt, dass sich das Systemverhalten nicht verzweigt, und andererseits wird die Position innerhalb des Ablaufs damit Teil des Objektzustands von *SimRevErrorSequence*. Genauer diskutieren wir diesen zentralen Unterschied zu induktiven Systemen dann in 9.6.

Diese Form des Zugriffs stellt in FastAsy die bevorzugte Methode dar.

## 9.5.3 Ausgewählte UML-Dokumente

### *9.5.3.1 Klassendiagramme*

#### 9.5.3.1.1 Simulationsklasse und Simulatoren

Im folgenden Diagramm findet sich die Klasse *GenericSimulation* und weiter die zwei momentan möglichen Werte für den Typparameter *TSimulator*, nämlich *SimpleSimulator* zur Erzeugung einer herkömmlichen Simulation und *ImplicitSimulator* zum Berechnen einer Simulation auf Systemen, bei denen Teilmengen der Verweigerungsmengen implizit gehalten werden.

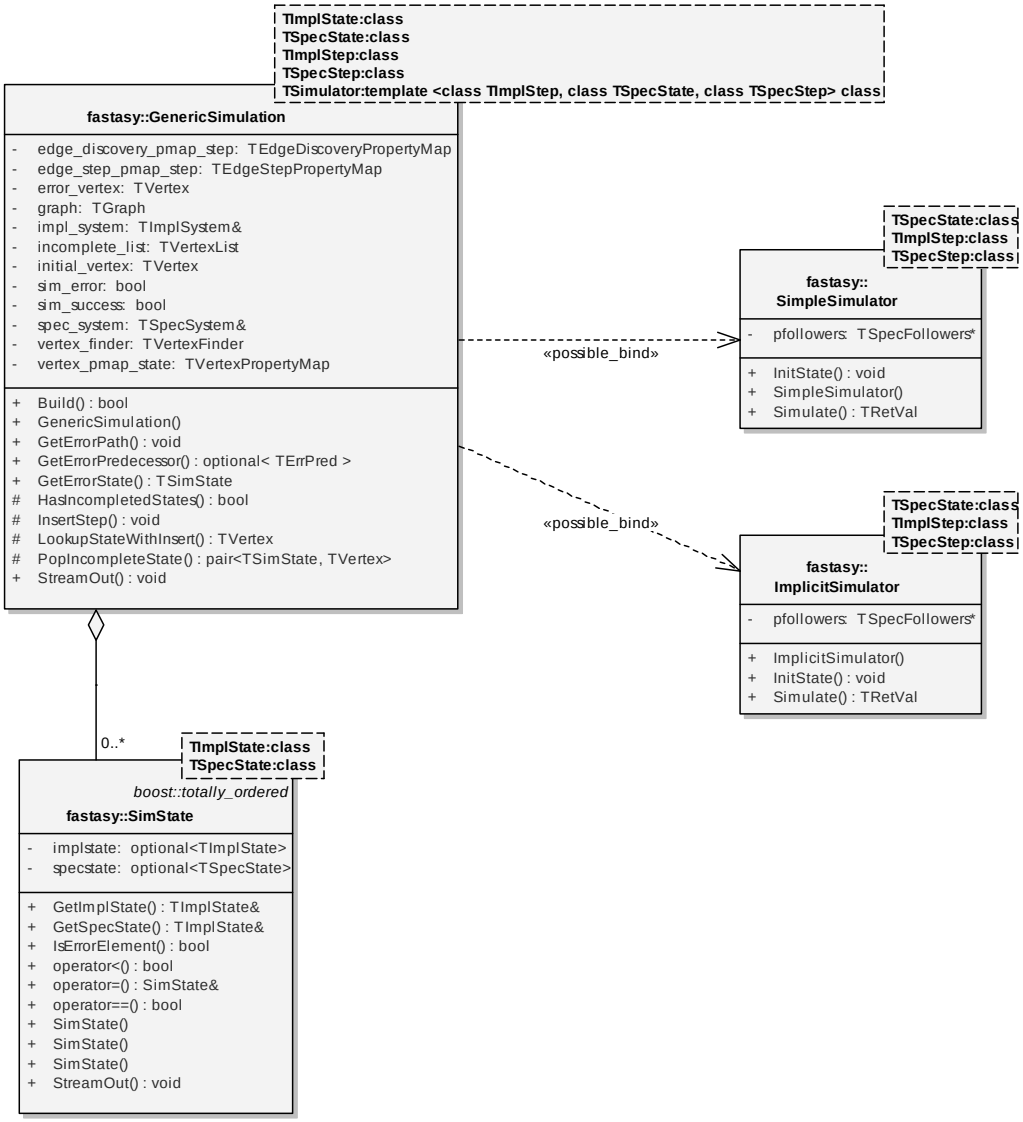



**Abbildung 50: Klassendiagramm GenericSimulation**

Wie man weiter sieht, sind die Typparameter für Schritte und Zustände getrennt aufgeführt für Implementierung (*TImplState*, *TImplStep*) und Spezifikation (*TSpecState*, *TSpecStep*). Als einziger Typparameter wird dabei der Zustandstyp der Implementierung nicht

an die Simulatoren weitergegeben, da dieser Typ für die Simulatoren belanglos ist. Für *ImplicitSimulator* hingegen ist beispielsweise der Zustandstyp der Spezifikation sehr wohl von Belang, da auf diesem Berechnungen ausgeführt werden und insofern ein Interface wie dasjenige von *TIntSetState* erwartet wird.

#### 9.5.3.1.2 Zugreiferklassen für den umgekehrten Fehlerpath

Hier sehen wir noch einmal die beiden Zugreiferklassen, die von *InductiveSystem* abgeleitete Klasse *SimRevErrorSystem* und *SimRevErrorSequence*, welches von *ForwardSequence* abgeleitet ist. (Wie wir in 9.6 sehen werden, modelliert *ForwardSequence* einfach lineare Systeme.) Beide Klassen sind dabei mit dem Typ der Simulation *TSimulation* parametrisiert, nicht direkt mit den Typen von Schritten und Zuständen, die sie verwenden. Dies ist erstens geringfügig bequemer, da nur ein einziger Parameter angeben werden muss, aus dem sich die anderen Typen ableiten lassen, zweitens aber benötigen beide Klassen ja eine Referenz auf das *TSimulation*-Objekt, denn sie materialisieren den umgekehrten Fehlerpath nicht, sondern delegieren alle Anfragen einzeln an die Methoden *GetErrorState()* und *GetErrorPredecessor()* der Simulation.

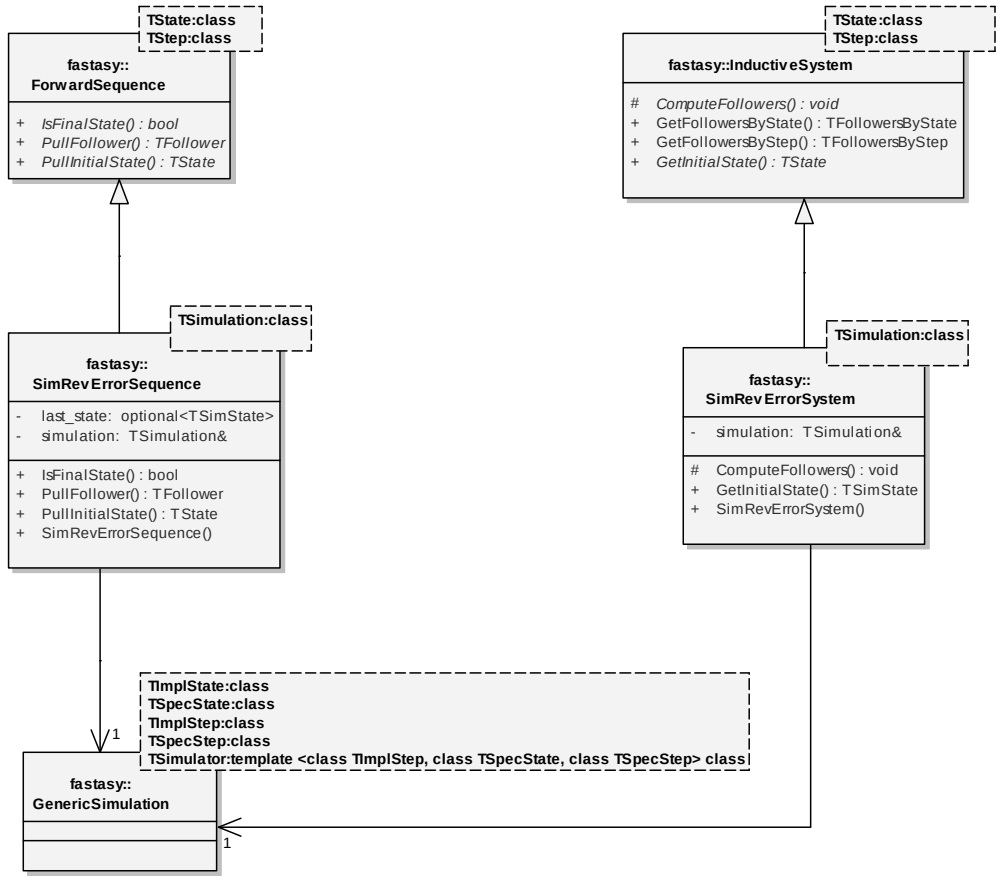



**Abbildung 51: Klassendiagramm ErrorPathAccess**

## *9.5.3.2 Kollaborationsmechanismen*

#### 9.5.3.2.1 Aufbau der Simulation

Vergleicht man das folgende Diagramm mit 9.4.3.2.1, sieht man schnell die engen Parallelen. (Der Übersichtlichkeit halber wurde der Ausführungspfad, der das Fehlerelement einfügt und den Aufbau der Simulation abbricht, nicht gesondert eingetragen.)

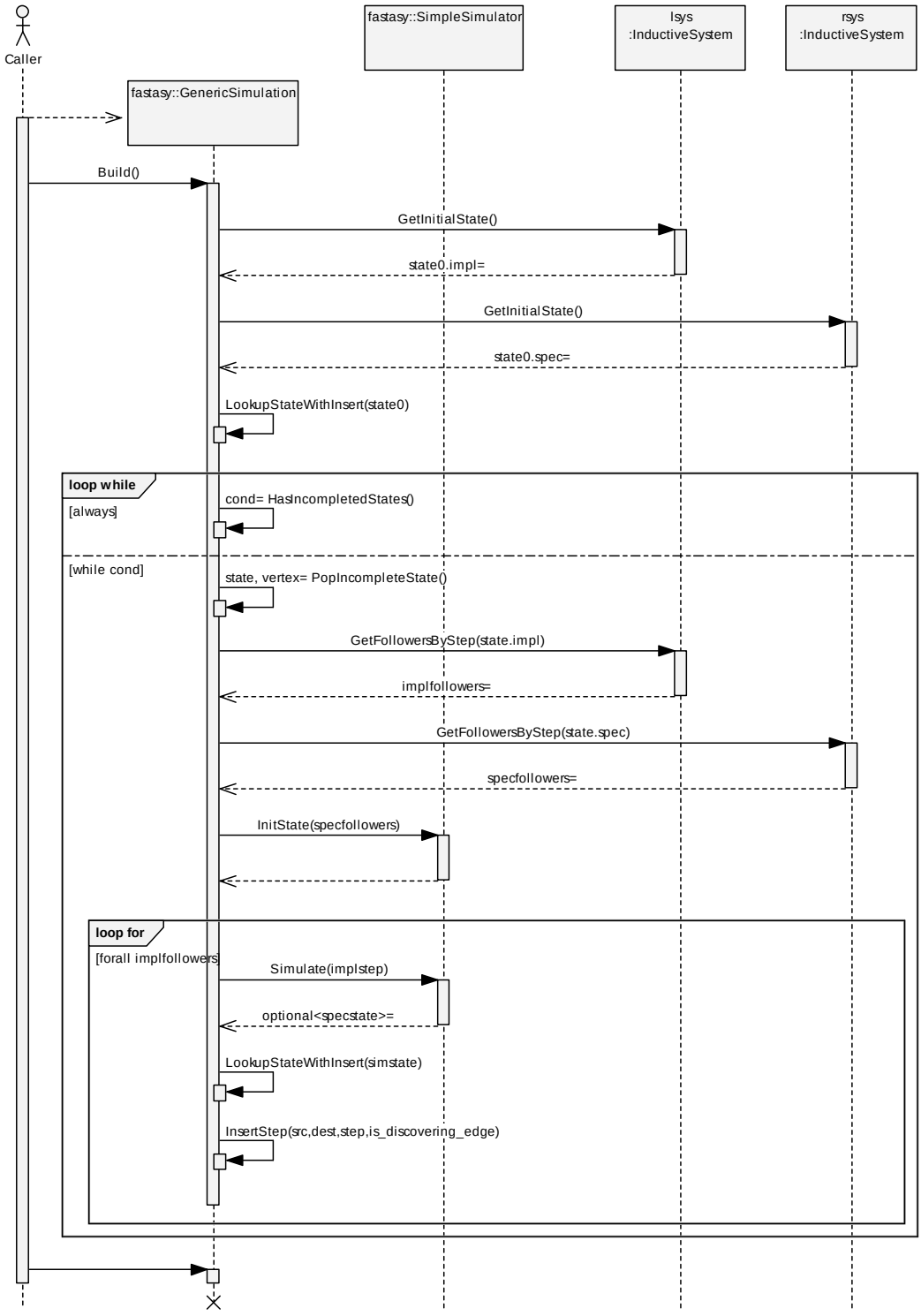



**Abbildung 52: Sequenzdiagramm Build Simulation**

Man beachte aber auch, dass das obige Diagramm wie auch das entsprechende für *Automaton* eine bestimmte Detailebenene gar nicht enthält und dass die dortigen Unterschiede gar nicht aufgezeigt werden: Aufgrund der abweichenden Speicherstruktur ist die Funktion der internen Methoden *InsertStep()* und *LookupStateWithInsert()* bei beiden Klassen natürlich verschieden.

#### 9.5.3.2.2 Abruf des umgekehrten Fehlerpfads über SimRevErrorSequence

Im folgenden sehen wir das Aufrufschema für den Fall, dass aufrufender Code den gesamten umgekehrten Fehlerpfad ausgibt. Dazu initialisiert der Aufrufer zunächst mit *PullInitialState()* das lineare System und zieht dann sukzessiv die weiteren Vorgänger (d.h. die *Nachfolger* im *umgekehrten* Fehlerpfad) mit *PullFollower()* ab. Wie wir sehen, ist *SimRevErrorSequence* dafür verantwortlich, die Rückgabewerte vom *GenericSimulation* auf die Implementierungs-Komponente zu projizieren.

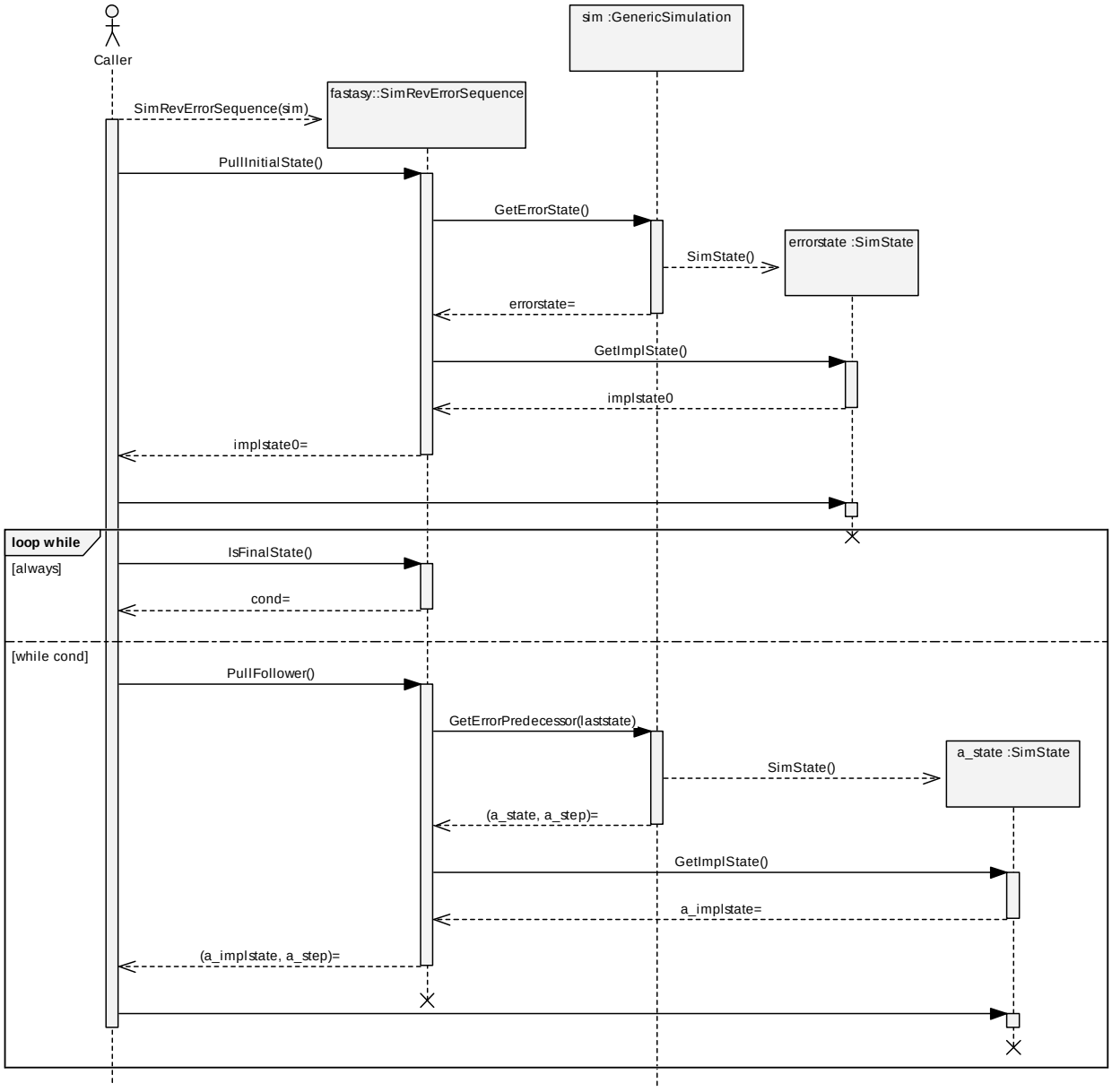



**Abbildung 53: Sequenzdiagramm ErrorPathAccess**

## 9.6 Entwurf des Package „Retracing“

Die Vorgehensweise beim *Retracing* wurde in 7.4 bereits ausführlich geschildert. Es geht dabei darum, den mit Hilfe des Package „Simulation“ erhaltenen Fehlerpfad, der ja ein Kantenzug in der direkten Eingabe der Simulation (genauer der Implementierungsseite) ist, schrittweise auf die jeweiligen Vorgängersysteme abzubilden, d.h. jeweils einen Kantenzug im Vorgängersystem zu finden, der den gegebenen Ablauf des Fehlerpfads verursacht.

Indem wir den Fehlerpfad wie eben als Kantenzug in der Implementierung beschreiben, machen wir bereits klar, dass die folgenden Beschreibungen davon ausgehen, dass der Fehlerpfad (genauer natürlich seine Umkehrung, siehe weiter unten) durch *SimRevErrorSequence* modelliert wird (siehe 9.5.2.1 und 9.5.2.5). Wir werden sehen, dass der Begriff des linearen Systems in „Retracing“ ebenso zentral sein wird wie der des induktiven Systems in „Transformation“. In der Tat herrscht zwischen beiden (bzw. deren Instanzen) eine gewisse Dualität.

Dies sehen wir auch an der untenstehenden Abbildung, in der wir uns (am Beispiel der Bausteine, die zur Abarbeitung des CLI-Befehls *SIMR* verwendet werden, siehe 8.1.1.6) die Beziehungen der einzelnen Stufen und deren (gedachter) Zwischenergebnisse verdeutlichen. Um *gedachte* Zwischenergebnisse handelt es sich deshalb, weil diese ja nicht unbedingt aufgebaut werden, sondern größtenteils nur implizit vorhanden sind. Für unsere Überlegungen und die Art der Darstellung spielt dies indes keine Rolle, ggf. stelle man sich alle Objektflüsse des Aktivitätsdiagramms mit einem entsprechenden Stereotyp gekennzeichnet vor. Weiterhin ist die Hintereinanderausführung von Aktivitäten aus dem gleichen Grund natürlich auf der Detailebene der tatsächlichen Abläufe in FastAsy als ein Ineinandergreifen zu lesen, in diesen Fällen wurden die entsprechenden Aktivitäten tatsächlich mit dem Stereotyp <<*adaptor*>> gekennzeichnet.

Derjenige Teil des Gesamtsystems, der für die Verarbeitung hin zum deterministischen induktiven System der Spezifikationsseite verantwortlich ist, wurde in der Darstellung ausgespart, da er einerseits völlig analog zur dargestellten Implementierungsseite verläuft und andererseits für das Retracing keinerlei Bedeutung hat. Weiterhin wurden im vorliegenden Schema keine Materialisierungen verwendet, weder von induktiven noch von linearen Systemen, da diese ja einfach nur Typ und Inhalt ihrer Vorgänger übernehmen.

Eine Komplikation wollten wir jedoch trotzdem in das Diagramm aufnehmen: Wie schon gesagt stellt *SimRevErrorSequence* ja die *Umkehrung* des Fehlerpfads im Potenzautomaten bereit, und *NondetRetracer* (Details siehe weiter unten) produziert daraus die *Umkehrung* des nicht-deterministischen Fehlerpfads im zustandsindizierten System. Erst der Adapter *RevSequence* stellt dann den vorwärtsgerichteten Fehlerpfad zur weiteren Verarbeitung bereit.

Zuletzt ist im Diagramm auch noch verdeutlicht, dass die jeweiligen Retracing-Prozesse als sekundäre Eingabe Kenntnis desjenigen induktiven Systems haben, innerhalb dessen sie ein Ergebnis liefern sollen; diese Notwendigkeit macht man sich indes schnell klar.

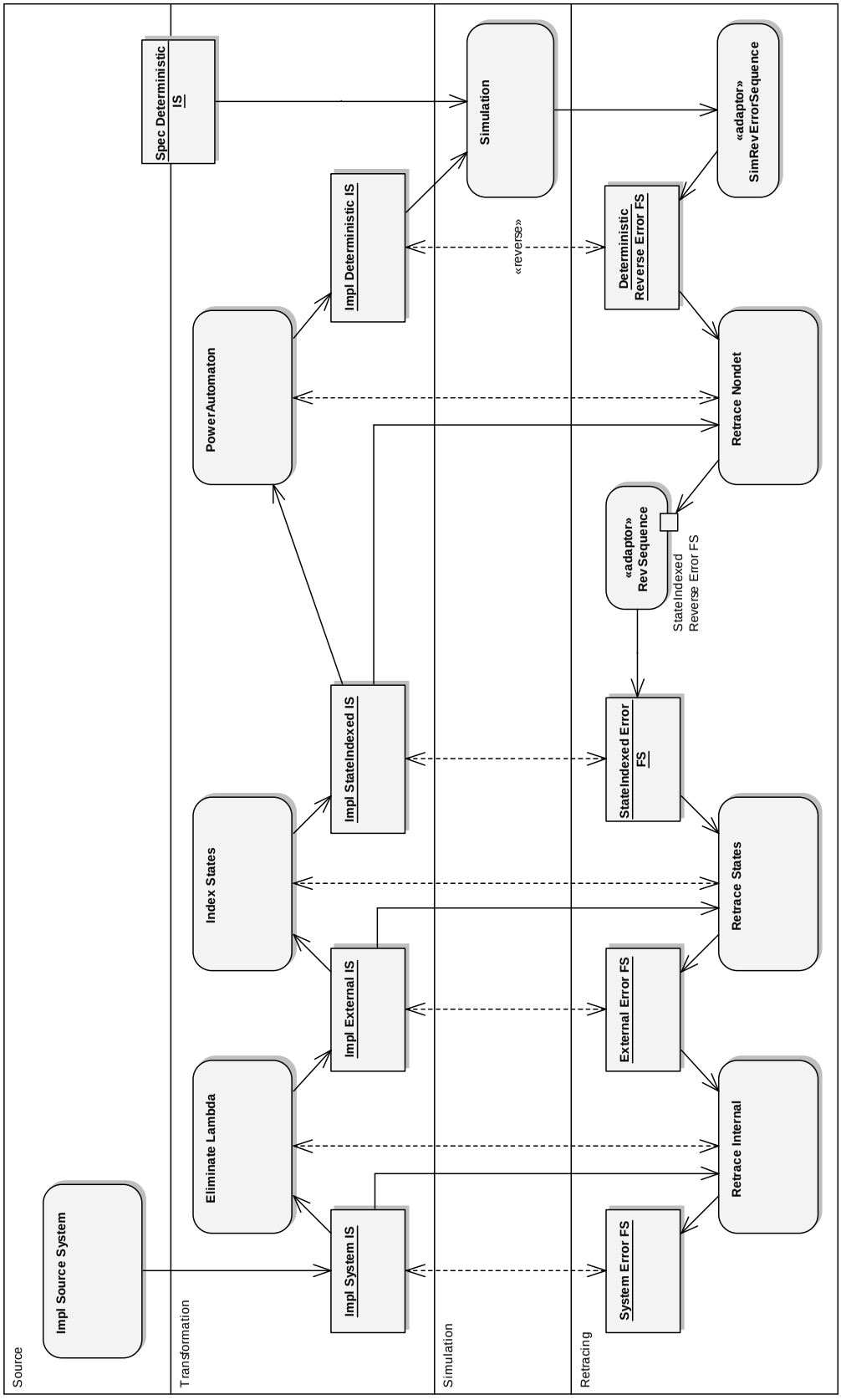



**Abbildung 54: Zusammenhang zwischen induktiven und linearen Systemen**

## 9.6.1 Anforderungen

### *9.6.1.1 Durchführung des Retracing*

Wir haben Sinn und Notwendigkeit des Retracing ja bereits dargelegt (siehe 7.4) und fassen deshalb nur nochmals kurz zusammen: Der Fehlerpfad, welchen wir aus der Simulation erhalten, ist ja zunächst ein Ablauf der direkten Eingabe der Simulation, also eines deterministischen Automaten, der mittels der Potenzautomatenkonstruktion und weiterer vorangehender Transformationen erstellt wurde. Die Zustände, welche in diesem Fehlerpfad enthalten sind, lassen im Allgemeinen keine Rückschlüsse über die Zustände des ursprünglichen Systems zu. Allein auf Grund der Übergänge im Fehlerpfad auf das Verhalten des Quellsystems zu schließen gestaltet sich im allgemeinen Fall jedoch recht schwierig. (Zudem wäre es denkbar, dass neue Bausteine im „Transformation“-Package auch die Übergänge nicht unangetastet lassen, so dass der Zusammenhang mit dem Ursprungssystem überhaupt nicht mehr direkt sichtbar ist.)

Wir benötigen also eine weitere Kette von Verarbeitungsstufen in FastAsy, die die Schritte, welche durch Komponenten in „Transformation“ vorwärts stattfinden, nun wieder rückwärts abwickelt und dabei allerdings nicht Automaten (gegeben als induktive Systeme) transformiert, sondern Abläufe in diesen. Die Anforderungen an die Mechanismen dieser Kette werden wir gleich im Anschluss darlegen. Die prinzipielle Arbeitsweise der einzelnen Stufen haben wir in 7.4 ausgeführt.

Es sei nochmals erwähnt, dass das in alten Versionen von FastAsy implementierte monolithische Retracing keine Alternative mehr darstellt, da es den ganzen Aufwand, der getrieben wurde, um die Kette der Transformationen flexibel zu halten, ad absurdum führen würde.

Eine Alternative zum stufenweisen Retracing, die hingegen erwogen werden könnte, wäre es, eine explorative Suche des Fehlerpfads im Automaten des Quellsystems vorzunehmen. Dabei wären wir sowieso nur an möglichst kurzen Expansionen internen Verhaltens interessiert, so dass uns Kreise interner Übergänge keine Probleme bereiten würden. Tatsächlich wäre dies sogar eine einfache Art, wirklich an die kürzeste Entsprechung des Fehlerpfads im Quellsystem zu gelangen. (Man beachte hier allerdings, dass ja selbst die kürzeste Entsprechung des kürzesten Fehlerpfads in der Simulation noch nicht der kürzeste Fehlerpfad ausgedrückt im Quellsystem sein muss.) Allerdings bedingt ein solches Verfahren eine ziemlich harte Einschränkung der Generizität unserer Bausteine: Der Typ der Schritte müsste im Quellsystem nämlich dann derselbe sein wie in demjenigen transformierten System, auf dem die Simulation durchgeführt wird. Zumindest müsste bei verschiedenen Typen eine surjektive Abbildung von der Menge der Übergänge im Quellsystem in die Menge der Übergänge im transformierten System existieren, die regelt, welche Übergänge einander entsprechen. In jedem Falle müsste der Schritttyp des Quellsystems aber die *IsInternal()*-Operation anbieten. Interessant für die Zukunft wäre die Implementierung eines solchen *ExplorativeRetracer* jedoch in jedem Fall, zumal eine solche Klasse ja selbst bei abweichenden Schritttypen als „Lückenfüller“ zwischen verschiedenen Retracing-Stufen gleichen Schritttyps arbeiten könnte.

#### *9.6.1.2 Lineare Systeme*

Wir wollen die Kette der erwähnten Verarbeitungsstufen ebenso flexibel und erweiterbar halten wie die aus induktiven Systemen bestehende, so dass wir einen ähnlichen Mechanismus wie dort zur Verschaltung einsetzen wollen (siehe 9.2.2). Dazu halten wir uns aber auch die zentralen Unterschiede vor Augen:

- Lineare Systeme verzweigen sich nicht (Spezialisierung)
- Lineare Systeme haben einen Endzustand (Generalisierung)
- Zustandsbeschreibungen in linearen Systemen sind nicht eindeutig (Generalisierung)
- Materialisierungen linearer Systeme in FastAsy sind Objekte von vernachlässigbarer Komplexität verglichen mit Materialisierungen induktiver Systeme

Der kritischste Punkt der obigen Auflistung ist dabei natürlich die Tatsache, dass sich Zustandsbeschreibungen innerhalb eines Ablaufs wiederholen können. Somit ist bereits klar, dass es nicht wie bei induktiven Systeme in Frage kommt, die Zustandsbeschreibung als eine Art Iteratorposition zu verwenden. Unsere Lösung wird daraus bestehen, die Iteratorposition in das Objekt, welches das lineare System repräsentiert, hineinzuziehen. Der vierte Punkt oben gibt uns dazu insofern eine Rechtfertigung, als dass wir für den Fall, dass wir verschiedene Positionen gleichzeitig verwalten müssen, einfach Kopien einer Materialisierung anfertigen können.

Hinsichtlich der Generizität der folgenden Klassen bezüglich der Typen von Schritt- und Zustandsbeschreibungen gilt natürlich wieder alles, was schon im Package „Transformation“ erörtert wurde. Da die dortige Diskussion des Themas bereits erschöpfend war und an dieser Stelle keine neuen Aspekte hinzutreten werden, wollen wir hier lediglich nochmals erwähnen, dass die linearen Systeme natürlich diese Freiheitsgrade ebenso und in derselben Form besitzen wie die induktiven Systeme.

### 9.6.2 Evolution des Klassenverbunds

Wir wollen im folgenden zunächst diejenigen Klassen vorstellen, mit deren Hilfe lineare Systeme in FastAsy modelliert werden. Dies liefert sozusagen die handfeste Grundlage, auf der wir dann die Umsetzung der einzelnen Stufen des Retracing entwickeln. Wir werden an den jeweiligen Stellen auch immer Hinweise zu eventuellen Parallelen bei den induktiven Systemen bzw. dem Package „Transformation“ geben.

Eine Bemerkung zur Nomenklatur der Klassen sei noch vorausgeschickt: Da historisch die Nachfolger von *InductiveSystem* mit dem Postfix *System* benannt wurden, schien es ungeschickt, das Modell linearer Systeme mit *LinearSystem* zu benennen. Stattdessen wurde der Name *ForwardSequence* eingeführt, womit die Implementierungen dann passend mit dem Postfix *Sequence* versehen werden können. Die Bezeichnung *LinearSequence* wurde ebenfalls erwogen, aber aufgrund der unschönen Redundanz verworfen. (*Forward* ist dem STL-Modell des *forward-iterator* entlehnt, dessen Semantik teilweise durch *ForwardSequence* gegeben ist.)

### *9.6.2.1 Repräsentation linearer Systeme durch ForwardSequence*

Die Klasse *ForwardSequence* ist im Gegensatz zu *InductiveSystem* vollständig abstrakt definiert, da aufgrund der einfacheren Signaturen der Operationen kein zu *GetFollowersBy...()* / *ComputeFollowers()* korrespondierender Mechanismus benötigt wird.

*ForwardSequence* verlangt von abgeleiteten Klassen die Implementierung der folgenden Methoden:

- *PullInitialState()* versetzt das System in die Anfangsposition und gibt die Beschreibung des dortigen Zustands zurück.
- *IsFinalState()* beantwortet die Frage, ob das System bereits im Endzustand angekommen ist.
- *PullFollower()* gibt die Beschreibung des nächsten Übergangs und dessen Zielzustands zurück, wobei die aktuelle Position des Systems entsprechend fortschreitet. Ein Aufruf von *PullFollower()* ist natürlich nur zulässig, solange *IsFinalState()* nicht *true* liefert.

Wir bemerken dazu mehrere Dinge:

- Wir unterscheiden hier notwendigerweise zwischen Zustandsbeschreibung und Position. Eine Zustandsbeschreibung kann an mehreren Positionen auftreten. Es liegt in der Verantwortung von *ForwardSequence*, sich die Position innerhalb des linearen Systems zu merken. (In 9.4.2.1 wurden mehrere Argumente angeführt, warum wir dies bei induktiven Systeme ausdrücklich *nicht* so haben wollten, so dass wir diesbezüglich einen Bruch der Parallelität akzeptieren müssen.)
- Statt Methodennamen mit dem Präfix *Get* schreiben wir hier *Pull*. Dies soll verdeutlichen, dass es sich nicht um *const* Methoden handelt. Das Objekt, auf welchem die Methoden aufgerufen werden, wird nämlich sehr wohl verändert, da ja die interne Position weitergezählt bzw. zurückgesetzt wird.
- Dadurch, dass bei einer Hintereinanderschaltung im Stile von 9.2.2 ein lineares System auch von seinem Vorgänger das *ForwardSequence*-Interface voraussetzen darf, muss es sich möglicherweise die eigene Position gar nicht explizit merken, sondern kann dies dem Vorgänger überlassen, der wiederum seinem Vorgänger, usw. Diese Kette bricht freilich in dem Moment ab, in dem keine Bijektion zwischen den Positionen mehr vorhanden ist, wie z.B. beim Wiederhinzufügen interner Übergänge. In einem solchen Fall muss sich das entsprechende Objekt tatsächlich zumindest partiell seine Position merken.

### *9.6.2.2 Klassen SequenceBookmarking und ForwardByBookmark*

Wie wir obiger Beschreibung entnehmen können, ist keine Möglichkeit vorgesehen, Objekte von *ForwardSequence*-Nachfolgern dazu zu bringen, ihre Position (auch *bookmark* genannt) herauszugeben und das Objekt später wieder in diesen Zustand zu versetzen (vgl. auch das Entwurfsmuster *Memento* bei [GoF96]). Wollten wir dies im allgemeinen Fall erreichen, müssten wir eben genau die schon öfter erwähnte Anreicherung der Zustandsinformation durchführen, um diese so eindeutig zu machen.

Wird ein lineares System aber materialisiert, so sollte es ein Leichtes sein, derartige *bookmarks* herauszugeben, im einfachsten Falle in der Form natürlicher Zahlen, wobei *n* dann

etwa auf die *n*-te Position und den Übergang zwischen den Zuständen an Position *n-1* und *n* verweist.

Neben der Tatsache, dass eine Materialisierung ein entsprechendes Interface bereitstellen kann sehen wir auch schnell, dass die Aufgabe, ein lineares System umzukehren, dieses Interface auch *benötigt*. Auf der Schnittstelle *ForwardSequence* jedenfalls wird man in natürlicher Weise keine Umkehrung durchführen können.

Wir führen also eine weitere Schnittstelle *SequenceBookmarking* ein, welche die folgenden Operationen unterstützt:

- Abfrage des *bookmark* der Anfangsposition
- Abfrage des *bookmark* der Endposition
- Prüfung eines *bookmark* auf Gültigkeit
- Abfrage der Zustandsbeschreibung zu einem *bookmark*
- Abfrage des Übergangs von der durch ein *bookmark* beschriebenen Position zur nachfolgenden
- Abfrage des Übergangs auf die durch ein *bookmark* beschriebenen Position von der vorhergehenden

Außerdem soll der Typ des *bookmark* selbst Inkrementierung und Dekrementierung unterstützen. Die Gültigkeit eines derart modifizierten *bookmark* muss dann allerdings erst wieder durch *SequenceBookmarking::IsValidBookmark()* überprüft werden. (Bleiben wir bei der Repräsentation eines *bookmark* als natürlicher Zahl, so findet durch *IsValidBookmark()* einfach eine Bereichsprüfung statt und die Modifikatoren bestehen beispielsweise aus den Operatoren ++ und --.)

Die nahe liegende Möglichkeit, die obigen Operationen einzuführen, bestünde nun natürlich darin, *SequenceBookmarking* von *ForwardSequence* erben zu lassen und so das Interface zu erweitern. Dies würde jedoch für alle Implementierungen von *SequenceBookmarking* bedeuten, dass sie *beide* Sätze von Operationen implementieren müssten. Dabei macht man sich schnell klar, dass sich die Operationen in *ForwardSequence* einfach durch Einführen einer Zustandsvariablen in allgemeiner Form auf die obigen reduzieren lassen. Deshalb gehen wir anders vor: Wir führen *SequenceBookmarking* als eigenständiges Interface neben *ForwardSequence* ein. Anschließend definieren wir eine neue Klasse *ForwardByBookmark*, welche beide Schnittstellen erbt und die Methoden von *ForwardSequence* mit Hilfe derer von *SequenceBookmarking* implementiert. Letztere bleiben dabei natürlich immer noch abstrakt. Leiten wir nun Klassen von *ForwardByBookmark* ab, so müssen diese lediglich das *SequenceBookmarking*-Interface implementieren und erhalten automatisch die Methoden aus *ForwardSequence* dazu.

Da wir dieselbe Technik im Übrigen im Kontext linearer Systeme nochmals bei *SequenceMutator* / *BuildByMutator* antreffen werden, nehmen wir uns kurz die Zeit, das Vorgehen zu einem kleinen Design Pattern zu abstrahieren, welches wir das by-Pattern getauft haben:

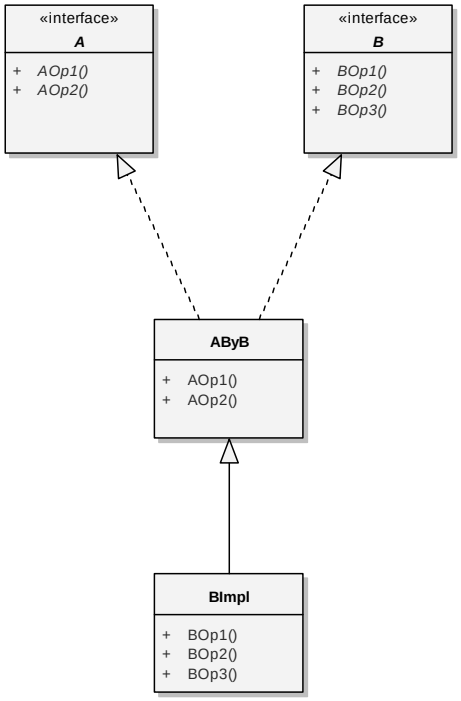



**Abbildung 55: Das by-Pattern**

Ausgangspunkt stellen zwei Interfaces (d.h. vollständig abstrakte Klassen) *A* und *B* dar, die jeweils Sätze von Operationen $AOp_i$ und $BOp_j$ enthalten. Die Operationen $AOp_i$ lassen sich nun auf die $BOp_j$ reduzieren. Genau diese Reduktion fassen wir in der (immer noch abstrakten, aber natürlich nicht mehr vollständig abstrakten) Klasse *AByB* zusammen, indem wir dort die $AOp_i$ mit Hilfe der $BOp_j$ implementieren. (Letztere werden dadurch zu *Schablonenmethoden*.) Indem wir nun in einer beliebigen *B*-Implementierung *B* als direkte Basisklasse gegen *AByB* austauschen, erhalten wir eine Klasse, die sowohl *A* als auch *B* implementiert.

Man beachte, dass wir das Interface *SequenceBookmarking* bereits breit genug angelegt haben um damit auch eine Umkehrung des Ablaufs zu ermöglichen. Dies wollen wir mit der folgenden Klasse nun ausnutzen.

### *9.6.2.3 Umkehrungen mit RevSequence*

*RevSequence* stellt den Rückwärtsablauf seines Vorgängersystems dar. Wie schon mehrfach erwähnt, arbeitet ja das Retracing der Einzelzustände (*NondetRetracer*, s. auch 9.6.2.8) auf Rückwärtsabläufen. *RevSequence* wird nun verwendet, um nachfolgenden Verarbeitungsstufen im Retracing wieder einen Vorwärtsablauf als Eingabe zur Verfügung zu stellen. (Wäre nicht die Ausgabe der Simulation in natürlicher Weise ein Rückwärtsablauf, so würden wir *RevSequence* freilich auch zum Umkehren der Eingabe von *NondetRetracer* heranziehen.)

Die Klasse *RevSequence* implementiert das Interface *SequenceBookmarking*, indem sie Aufrufe selbst wieder an ein solches Interface delegiert. Damit stellt auch *RevSequence* natürlich wieder eine Art *Dekorierer* dar, man beachte allerdings, dass es sich damit trotzdem *nicht* in die Kette der Bausteine linearer Systeme analog zu 9.2.2 einfügt. Es verlangt von seinem Vorgänger nämlich eben nicht nur die Methoden aus *ForwardSequence*, sondern

diejenigen aus *SequenceBookmarking*. Damit haben wir eine deutliche Parallele zu 9.4.2.7, wo *RevAutomatonSystem* eben auch nicht auf einem Vorgänger des Typs *InductiveSystem* operieren kann, sondern einen *ReversibleAutomaton* voraussetzt.

Dafür setzt *RevSequence* aber Aufrufe der eigenen Methoden in konstanter Zeit und mit konstantem Platzbedarf an Aufrufe des *SequenceBookmarking*-Objekts um.

### *9.6.2.4 Materialisierung durch Trace*

Die Klasse *Trace* stellt neben *RevSequence* die andere Implementierung von *SequenceBookmarking* dar. In gewissem Sinne ist *Trace* das Pendant zu *Automaton*: *Trace* enthält ebenfalls eine Methode *Build()*, welche aus einem in Form einer *ForwardSequence* gegebenem linearen System eine Materialisierung erstellt.

Außerdem enthält *Trace*, ähnlich wie *MutableAutomaton*, Methoden, um explizit aus aufrufendem Code heraus ein lineares System zu definieren. Dabei merkt man *Trace* gegenüber *Automaton* in gewisser Hinsicht die Gnade der späteren Geburt an: Während bei *Automaton* die *Build*-Operation direkt implementiert ist und erst in der Spezialisierung *MutableAutomaton* weitere Operationen hinzugefügt werden, die die direkte Manipulation ermöglichen, bedienen wir uns im folgenden wieder des *by-Patterns* (s. 9.6.2.2) als Technik, ein Interface auf ein anderes zu reduzieren, so dass *Trace* nur noch eines davon (nämlich *SequenceMutator*) implementieren muss.

(Dabei können aber die Vererbungsstrukturen von *Trace* und besonders auch von *Automaton* innerhalb des Entwicklungsprozesses von FastAsy als stabil gelten, so dass einem Refactoring von *Automaton* nach dem Muster von *Trace* ein rein kosmetischer Charakter zukäme und wir deshalb darauf verzichtet haben.)

### *9.6.2.5 Klassen SequenceMutator und BuildByMutator*

Wie gerade schon bemerkt, soll *Trace* sowohl Operationen zur Manipulation des eigenen linearen Systems beinhalten als auch natürlich die Fähigkeit, ein als *ForwardSequence* gegebenes lineares System zu materialisieren. Dazu konstruieren wir zunächst ein Interface *SequenceMutator* mit den folgenden Operationen:

- Löschen des linearen Systems
  (Man beachte, dass selbst ein leeres lineares System aus einem einzelnen Zustand besteht und insofern das System nach dem Löschen nicht als leer, sondern als undefiniert zu gelten hat.)
- Definieren des Anfangszustands
- Hinzufügen eines Schrittes (und dessen Zielzustand)

*Trace* implementiert nun dieses Interface. Um aber jetzt auch eine *Build()*-Methode analog zu *InductiveSystem* bereitzustellen, führen wir noch die Klasse *BuildByMutator* ein, welche *Build()* auf Basis der obigen Operationen implementiert.

Weiterhin implementiert *BuildByMutator* noch eine verwandte Methode, die in *InductiveSystem* kein Pendant besitzt: Mit *Append()* wird ein als *ForwardSequence* gegebenes lineares System an das durch das *Trace*-Objekt repräsentierte angehängt. Dabei kommt es noch zu einer Auffälligkeit: Wir würden eigentlich gerne voraussetzen, dass die so konkatenierten linearen Systeme insofern kompatibel sind, dass der Endzustand des vorderen Ablaufs dem Anfangszustand des angehängten entspricht. Diese Forderung kann jedoch nicht

auf der Basis von *SequenceMutator* formuliert werden. Deshalb macht die Methode *Append()* diese Zusicherung nicht (und ignoriert gegebenenfalls den Anfangszustand der angehängten Sequenz); in *Trace* jedoch existiert eine zusätzliche Methode *AppendStrict()*, welche lediglich die erwähnte Zusicherung prüft und anschließend das ererbte *Append()* aufruft. (Man könnte auch der Meinung sein, stattdessen sollte ein *Trace::Append()* mit der erwähnten zusätzlichen Zusicherung eingeführt werden, welches *BuildByMutator::Append()* überschreibt. Wir halten es jedoch für angebracht, mit dem Namen die veränderte Semantik kenntlich zu machen.)

Man beachte auch, dass natürlich die als Parameter an *Build()* bzw. *Append()* übergebenen Vorgängersysteme nicht *const* sein dürfen, da deren interne Positionen ja verändert werden.

#### *9.6.2.6 Retracer Basisklasse*

Während im Package „Transformation" jeder Baustein direkt *InductiveSystem* implementiert, existieren in „Retracing" noch einige Verwaltungsaufgaben, welche wir noch in einer allgemeinen Basisklasse erledigen können; somit lässt sich der Aufwand für die Implementierung einzelner Bausteine noch reduzieren. (Wenn man so will, hätte man *InductiveSystem* ja ebenfalls in einen reinen Interface-Teil und die Reduzierung der *GetFollowersBy…()*-Methoden auf *ComputeFollowers()* aufspalten können.)

Die wichtigste dieser Verwaltungsaufgaben ist dadurch bedingt, dass das Retracing eines einzelnen Übergangs im Allgemeinen nicht ebenfalls einen einzelnen Übergang als Ergebnis haben wird, sondern eine ganze Sequenz. Da *PullFollower()*, wie in *ForwardSequence* definiert, jedoch nur jeweils einen einzelnen Übergang abruft, muss ein Retracing-Baustein ggf. sozusagen auf Vorrat die Entsprechung eines einzelnen Übergangs berechnen und dann schrittweise über *PullFollower()* zurückliefern. Dazu reduziert die Basisklasse *Retracer* das gesamte *ForwardSequence*-Interface auf eine einzige Schablonenmethode *RetraceStep()*, welche dafür allerdings eine recht komplexe Signatur hat: Die Methode erhält als Eingabe natürlich zunächst den einzelnen Schritt, für den das Retracing durchgeführt werden soll (und der damit aus dem linearen System stammt, das als Vorgänger agiert), dessen Zielzustand, den Zustand *q* im induktiven(!) System, von dem der nachzuvollziehende Schritt *ausgeht*, und eine Referenz auf das zugehörige induktive System selbst. Zur Ausgabe des Ergebnisses erhält *RetraceStep()* noch einen Referenzparameter vom Typ *Trace*. Dieser ist bereits mit *q* als Startzustand initialisiert und es wird von Implementierungen von *RetraceStep()* erwartet, dass sie den Ausgabeparameter mit der dem übergebenen einzelnen Übergang entsprechenden Sequenz füllen.

Anschließend übernimmt *Retracer* die restliche Arbeit, d.h. speichert das Ergebnis von *RetraceStep()* und ruft diese Methode erst dann wieder auf, wenn durch sukzessive *PullFollower()*-Aufrufe die ganze *Trace* konsumiert worden ist.

Wie man sieht, arbeitet *Retracer* streng lokal, d.h. ohne Kenntnis der Fortsetzung der Vorgängersequenz. Trotzdem treffen die abgeleiteten Klassen im Falle mehrerer Entsprechungen eines gegebenen Übergangs eine Auswahl und sind durch die Angabe des letzten Zustands, den sie im Zuge eines *RetraceStep()* in die *Trace* schreiben auch im nächsten Aufruf von *RetraceStep()* auf diese Auswahl festgelegt. Eigentlich können solche Bausteine im Allgemeinen nun gar nicht vorhersehen, ob eine Auswahl sich überhaupt später korrekt fortsetzen lässt. Die Forderung, dass ein *Retracer*-Nachfolger dies aber doch erkennen

soll, nennen wir, wie bereits erwähnt, passenderweise *clairvoyance*. Zumindest im Falle von *NondetRetracer* werden wir feststellen, dass dies nicht immer ohne einen größeren Aufwand zu bewerkstelligen ist.

Im Übrigen stellen wir fest, dass der in 9.6.1.1 angerissene *ExplorativeRetracer* eben genau nicht auf *clairvoyance* angewiesen wäre, da er nicht lokal arbeitet. (Somit sollte er streng genommen auch mit *ExplorativeRetracingForwardSequence* benannt werden, da er ja das *Retracer*-Interface gar nicht implementiert.)

#### *9.6.2.7 TestRetracer*

In der Klasse *TestRetracer* ist die denkbar einfachste Version des Retracing umgesetzt, nämlich die, bei der einfach nach exakt dem durch das lineare System gegebenen Ablauf im induktiven System gesucht wird. Man wird sich nun fragen, zu welchem Zweck diese Klasse dienen soll, denn schließlich entspricht ihr Ergebnis ja genau einem Teil der Eingabe, nämlich dem linearen System. Man erinnere sich aber der in 9.1.2 erklärten Technik, Unit Testing mittels eines *mock objects* durchzuführen. Konkret möchten wir an dieser Stelle die Funktion der Klasse *Retracer* testen und verwenden dazu mit *TestRetracer* einen minimalistischen Nachfolger. Da *TestRetracer* nun eine derart einfache Struktur aufweist, können beim Unit Testing auftretende Fehler wesentlich zielsicherer aufgespürt werden, als wenn wir *Retracer* lediglich implizit im Zuge der Tests der komplexeren Nachfolgeklassen testen würden.

Man beachte, dass bei *TestRetracer* die Eigenschaft der *clairvoyance* im Allgemeinen nicht gegeben ist. Wir müssen unseren Testfall also derart konstruieren, dass die Eigenschaft für diese spezielle Eingabe trotzdem erfüllt wird, d.h. die gegebenen Abläufe müssen jeweils dadurch auffindbar sein, dass für den nächsten gegebenen Übergang des linearen Systems ein eindeutig bestimmter Übergang des induktiven Systems durchgeführt wird.

#### *9.6.2.8 NondetRetracer*

*NondetRetracer* führt wie in 7.4.1 beschrieben denjenigen Schritt des Retracing durch, bei dem ein als lineares System gegebener Ablauf in einem Potenzautomaten (dessen Zustände durch Werte vom Typ *TIntSetState* gegeben sind) in einem als induktives System gegebenen nichtdeterministischen, aber buchstabierenden Automaten (dessen Zustandsbeschreibungen vom Typ *TIntState* sind und darüber mit den Elementen der *TIntSetState*-Multizustände identifiziert werden können) nachvollzogen wird. Obwohl die Aufgabenstellung von *NondetRetracer* keinesfalls trivial ist, stellt die Klasse insofern eine Vereinfachung des allgemeinen *Retracer* dar, dass ein Schritt der Eingabesequenz immer zu genau einem Schritt in der Ausgabe wird.

Wir haben bereits früher darauf hingewiesen, dass sich *clairvoyance* hier nur erreichen lässt, wenn man den Ablauf rückwärts nachvollzieht. Dies bedeutet, dass für einen buchstabierenden Automaten **A** und einen Ablauf **w** das *NondetRetracer*-Objekt mit Rev(**A**) und Rev(**w**) gespeist werden muss und dass das Ergebnis ebenfalls revertiert werden muss, bevor es an die nächste Stufe weitergegeben wird. Auf der technischen Ebene kann dies zunächst z.B. so umgesetzt sein:

- **A** wird mit einem *ReversibleAutomaton* materialisiert.
  (Es existieren sowieso auch andere Gründe, dies zu tun.)

- Mit diesem *ReversibleAutomaton* wird ein *RevAutomatonSystem* **RA** initalisiert. (**RA** stellt also den revertierten Automaten modulo Start-/Endzustände dar.)
- **w** wird in einer *Trace* materialisiert.
- Ein Objekt vom Typ *RevSequence* **rw** wird mit dieser *Trace* initialisiert. (Damit stellt **rw** die Umkehrung von **w** dar.)
- Ein *NondetRetracer*-Objekt **NR** bekommt **RA** und **rw** als Parameter übergeben.
- Für eine weitere *Trace* **rt** wird *Build(***NR***)* aufgerufen.
- Es wird eine weitere *RevSequence* mit **rt** initialisiert.
- Diese *RevSequence* repräsentiert nun das Ergebnis als Vorwärtsablauf.

(Man beachte hierzu allerdings, dass in FastAsy **rw** bereits als Ergebnis der Simulation zur Verfügung steht und so die zu dessen Erzeugung notwendigen Schritte entfallen können.)

Obiges Vorgehen mag nun in der Tat etwas aufwändig scheinen, so dass wir uns kurz Gedanken machen wollen, ob man diese Komplexität wenn schon nicht vermeiden so doch verbergen kann: In 7.4.1 wurde bereits angedeutet, dass man mit etwas zusätzlichem Aufwand auf die Bestimmung der Umkehrung von **A** verzichten könnte. (Mit zusätzlichem Aufwand ist dabei nicht in erster Linie höhere Laufzeit gemeint, sondern vor allem auch der Umstand, dass die Signatur von *RetraceStep()* dahingehend erweitert werden muss, dass sie nicht nur den Ziel- sondern auch den Ausgangszustand des aktuellen Übergangs in der Sequenz übergibt.) Damit hat man bei obigem Vorgehen den einzigen teuren Teilprozess, nämlich die Materialisierung von **A**, eliminiert, und man könnte auf den Gedanken kommen, die Benutzung der zwei *Trace*- und *RevSequence*-Instanzen zusammen mit *NondetRetracer* in eine neu zu schaffende Fassadenklasse zu integrieren, so dass der Benutzer (unter dem Benutzer dieser Klasse wird natürlich trotzdem ein Programmierer zu verstehen sein) von der Notwendigkeit der expliziten Umkehrungen abgeschirmt wird. Dann allerdings sollte konsequenterweise auch eine weitere Fassade *SimErrorSequence* geschaffen werden, welche statt *SimRevErrorSequence* eingesetzt werden kann und den Fehlerpfad in Vorwärtsrichtung repräsentiert. Auf diese Weise käme es natürlich hinter den Kulissen zu zwei Umkehrungen, die sich aufheben; diese Umkehrungen samt der Materialisierungen wären aber vom Laufzeitaufwand vernachlässigbar, da es sich ja nur um lineare Systeme handelt, und man hätte den Benutzer der Klassen effektiv von der gesamten Thematik der Umkehrungen abgekapselt.

Ein weiterer Punkt, der bei *NondetRetracer* noch Beachtung verdient ist der, dass der Klasse im Konstruktor durch einen booleschen Parameter mitgeteilt wird, ob sie explizite Teilmengen erwartet oder nicht. Dadurch, dass dieser Parameter dynamisch ist, ist die Klasse in jedem Fall darauf angewiesen, das der Schritttyp über die Operationen *GetType()* und *GetRefusalset()* verfügt. Sollte die Klasse später einmal auf gänzlich abweichenden Schritttypen operieren müssen, so kann dies am einfachsten dadurch ermöglicht werden, dass das erwähnte Flag zu einem Templateparameter gemacht und die *RetraceStep()*-Methode jeweils für *true* und *false* spezialisiert wird. Eleganter und dem Stil von FastAsy besser entsprechend wäre natürlich die Extraktion der Prüfung, ob eine Kante als übereinstimmend gewertet wird, in eine Funktionalklasse, welche als Templateparameter übergeben wird.

#### *9.6.2.9 IndexRetracer*

Wir haben eben schon darauf hingewiesen, dass die Zustände, die im Ergebnis von *NondetRetracer* stehen, vom Typ *TIntState* sind. Diese Zustände entsprechen ja denen im buchstabierenden, zustandsindizierten induktiven System und wurden typischerweise dadurch gewonnen, dass sich in der Kette der Transformationen induktiver Systeme ein *SystemTransformator* mit einem *StateIndexer*-Funktional befunden hat. Wollen wir nun den ursprünglichen Zustandstyp vor der Indizierung wiederherstellen (etwa um im Falle von Petrinetzen dem Anwender von FastAsy tatsächlich Hinweise auf Markierungen im Petrinetz zu geben), so bietet es sich an, dazu genau diejenige Information zu benutzen, die das *StateIndexer*-Objekt während seines Einsatzes in *SystemTransformator* (in Form einer Indextabelle) gesammelt hat.

Dazu akzeptiert *IndexRetracer* in seinem Konstruktor einen weiteren Parameter, nämlich eine *const*-Referenz auf ein Objekt vom Typ *StateIndexer*. Die weitere Implementierung von *RetraceStep* für *IndexRetracer* ist damit trivial, da lediglich der ursprüngliche Zustand nachgeschlagen und in das Ergebnis eingefügt werden muss. Wie man dabei bemerkt, benötigt *IndexRetracer* also als einziger Nachfolger von *Retracer* nicht einmal wirklich das induktive System, um seine Arbeit zu erledigen. Da das *Retracing* aber in FastAsy sowieso nicht als zeitkritisch zu werten ist und wir uns ja *code robustness* auf die Fahnen geschrieben haben, enthält *RetraceStep* eine Laufzeit-Zusicherung, welche das gegebene induktive System daraufhin überprüft, ob der Übergang, den wir rekonstruiert haben, überhaupt enthalten ist. Um diesen Punkt aber klarzustellen: Zu einer Verletzung dieser Zusicherung kann es nur kommen, wenn eine andere Stelle in FastAsy fehlerhaften Code enthält (etwa wenn vom aufrufenden Code ein falsches induktives System übergeben wird).

#### *9.6.2.10 InternalRetracer*

Der letzte Nachfolger von *Retracer*, den wir hier betrachten wollen, ist *InternalRetracer*. Seine Aufgabe ist es, interne Übergänge, die während der Transformationsphase typischerweise von *ExternalInductiveSystem* eliminiert wurden, zu rekonstruieren. Damit ist *InternalRetracer* natürlich in den Typen der Zustände völlig frei von Beschränkungen, wohingegen der Schritttyp des übergebenen induktiven Systems offensichtlich die Operation *IsInternal()* besitzen muss.

Die Vorgehensweise, um die internen Übergänge wiederherzustellen, wurde ja in 7.4.4 bereits beschrieben. Es sei erwähnt, dass *InternalRetracer*, anders als man vielleicht annehmen könnte, keine Information nutzt, die während der Transformation durch *ExternalInductiveSystem* anfällt. Obwohl letztere Klasse nämlich Möglichkeiten bietet, Informationen über die λ-Hülle als Instanz von *AbstractLambdaHull* der Kontrolle des Aufrufers zu unterstellen und so als Seitenergebnis verfügbar zu machen, ist diese Information bei näherer Betrachtung für *InternalRetracer* nicht reichhaltig genug. Wir sind ja nämlich genau nicht daran interessiert, zu prüfen, *ob* ein λ-Pfad zwischen Zuständen existiert (dieses Wissen haben wir nämlich bereits), sondern daran, diesen Pfad zu erhalten.

Man kann sich aber durchaus die Frage stellen, ob es nicht möglich wäre, Implementierungen von *AbstractLambaHull* zu konstruieren, die sich auch diese zusätzliche Information merken und dann von einer spezialisierte Version von *InternalRetracer* verwendet werden. In der Tat scheint ein solches Vorgehen unproblematisch. Eine

Verbesserung der Gesamtlaufzeit wird man indes kaum feststellen können, da in typischen Systemen Komponenten mit starkem λ-Zusammenhang von überschaubarer Größe im Vergleich zum Gesamtsystem sind. Sollte man sich jedoch einmal in einer Situation befinden, in der eine bestimmte Art untersuchter Systeme eine solche Benutzung eines Seitenergebnisses wünschenswert erscheinen lässt, so bietet die Architektur von FastAsy wie skizziert ausreichend Flexibilität, um dies nachzurüsten.

## 9.6.3 Ausgewählte UML-Dokumente

### *9.6.3.1 Klassendiagramme*

#### 9.6.3.1.1 Repräsentation linearer Systeme

Das folgende Klassendiagramm stellt noch einmal in der Übersicht die an der Repräsentation linearer Systeme beteiligten Klassen dar. Wie gesagt stellt *ForwardSequence* die grundlegende Schnittstelle dar. *RevSequence* und *Trace* sind mit dieser natürlich über Vererbung verbunden. Ein *RevSequence*-Objekt initialisiert in seinem Konstruktor darüber hinaus eine Referenz auf diejenige *SequenceBookmarking*-Instanz, deren Umkehrung es repräsentiert. Eine *Trace* hingegen erbt die Methode *Build()* aus *BuildByMutator*, welche als Parameter eine Referenz auf eine *ForwardSequence* benötigt.

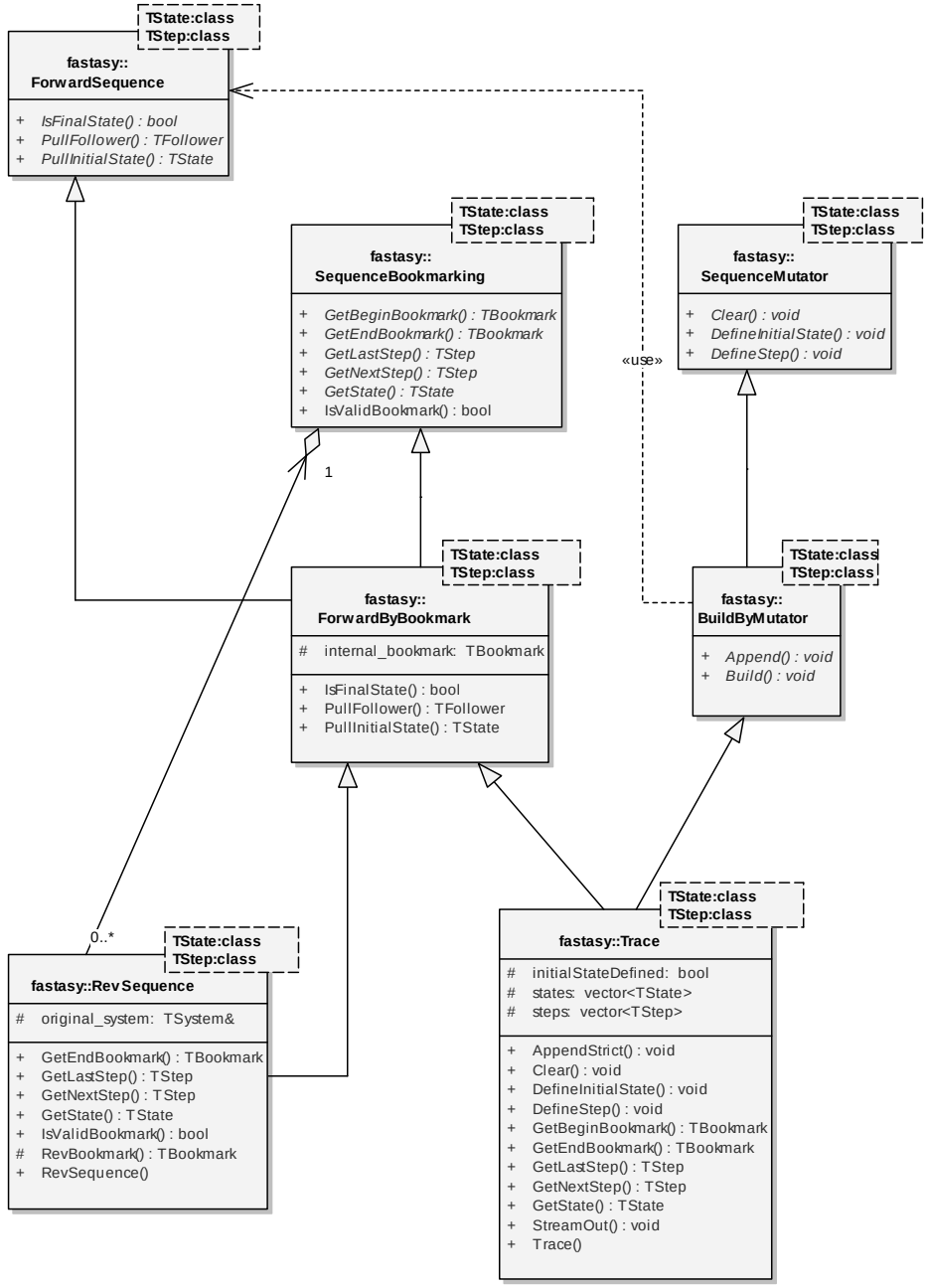



**Abbildung 56: Klassendiagramm Linear Systems**

#### 9.6.3.1.2 Umkehrung einer ForwardSequence

Aus dem Klassendiagramm in 9.6.3.1.1 geht schon hervor, dass eine *RevSequence* nicht direkt genutzt werden kann, um eine *ForwardSequence* umzukehren. Wie schon dargelegt, wäre eine Umkehrung auf Grundlage von *ForwardSequence* ja auch gar nicht direkt möglich. Deshalb wird die folgende Verwendung von *Trace* als Zwischenstufe stereotyp eingesetzt (siehe auch die Bemerkungen zu 9.6.2.8):

Eine *Trace* **T** materialisiert zunächst die umzukehrende *ForwardSequence* **S** und stellt diese so auch mit dem Interface *SequenceBookmarking* zur Verfügung. Dieses Interface nun kann die *RevSequence* **RS** nutzen, um das lineare System in umgekehrter Richtung bereitzustellen. (Insbesondere findet in **RS** keine Materialisierung statt, die Methoden in **RS** operieren alle mit konstantem Aufwand an Zeit und Platz.)

Man beachte, dass in einer Kette von Retracing-Bausteinen **RS** dann von seinem nachfolgenden Baustein typischerweise wieder über das *ForwardSequence*-Interface angesprochen wird, nicht über *SequenceBookmarking*, also genauso, wie ein Nachfolger auch **S** ansprechen würde.

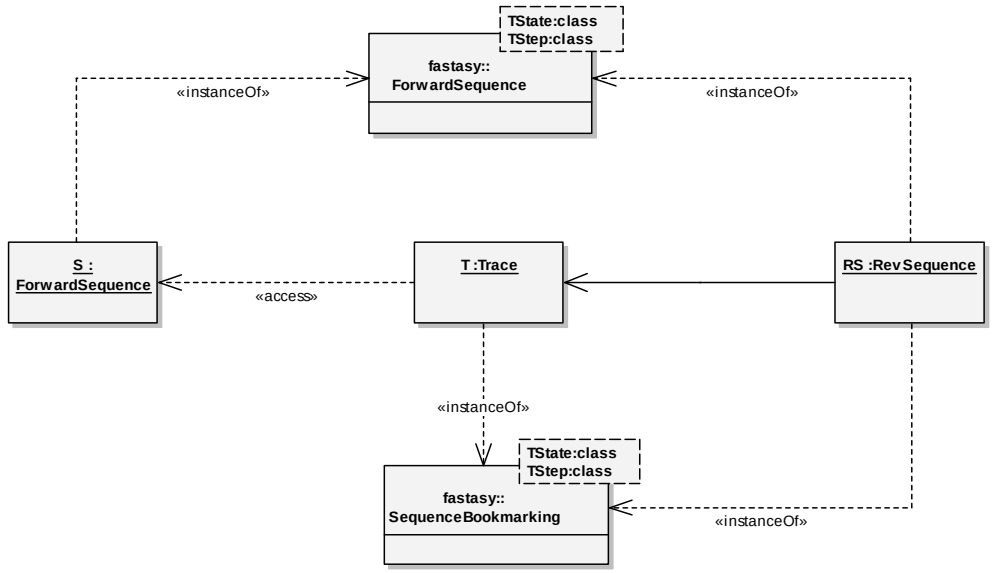



**Abbildung 57: Objektdiagramm Reverse Sequence**

#### 9.6.3.1.3 Retracer-Bausteine

Nachfolgend ist die Einordnung der Basisklasse *Retracer* samt ihrer direkten Kollaborationspartner sowie ihrer abgeleiteten Klassen zu sehen.

Wir sehen dabei, dass *Retracer* sowohl das Interface *ForwardSequence* implementiert wie auch im Attribut *input_sequence* einen Verweis auf dasjenige lineare System enthält, welches als Vorgänger (im Sinne der Baustein-Kette) agiert. Zudem enthält es auch einen Verweis *input_system* auf das induktive System, innerhalb dessen es den gegebenen Ablauf rekonstruieren soll.

Weiterhin nutzt es mit *stored_trace* intern eine private Instanz von *Trace*, um sich in *RetraceStep()* generierte Nachfolger zu merken, die vom Aufrufer dann sukzessiv über *PullFollower()* abgerufen werden können.

Ganz unten sehen wir noch, dass *IndexRetracer* im Gegensatz zu anderen *Retracer*-Implementierungen noch eine Assoziation zu einem Kollaborationspartner hat, nämlich zu demjenigen *StateIndexer*, der bei der Transformation des induktiven Systems mit *SystemTransformator* zum Indizieren der Zustände benutzt wurde.

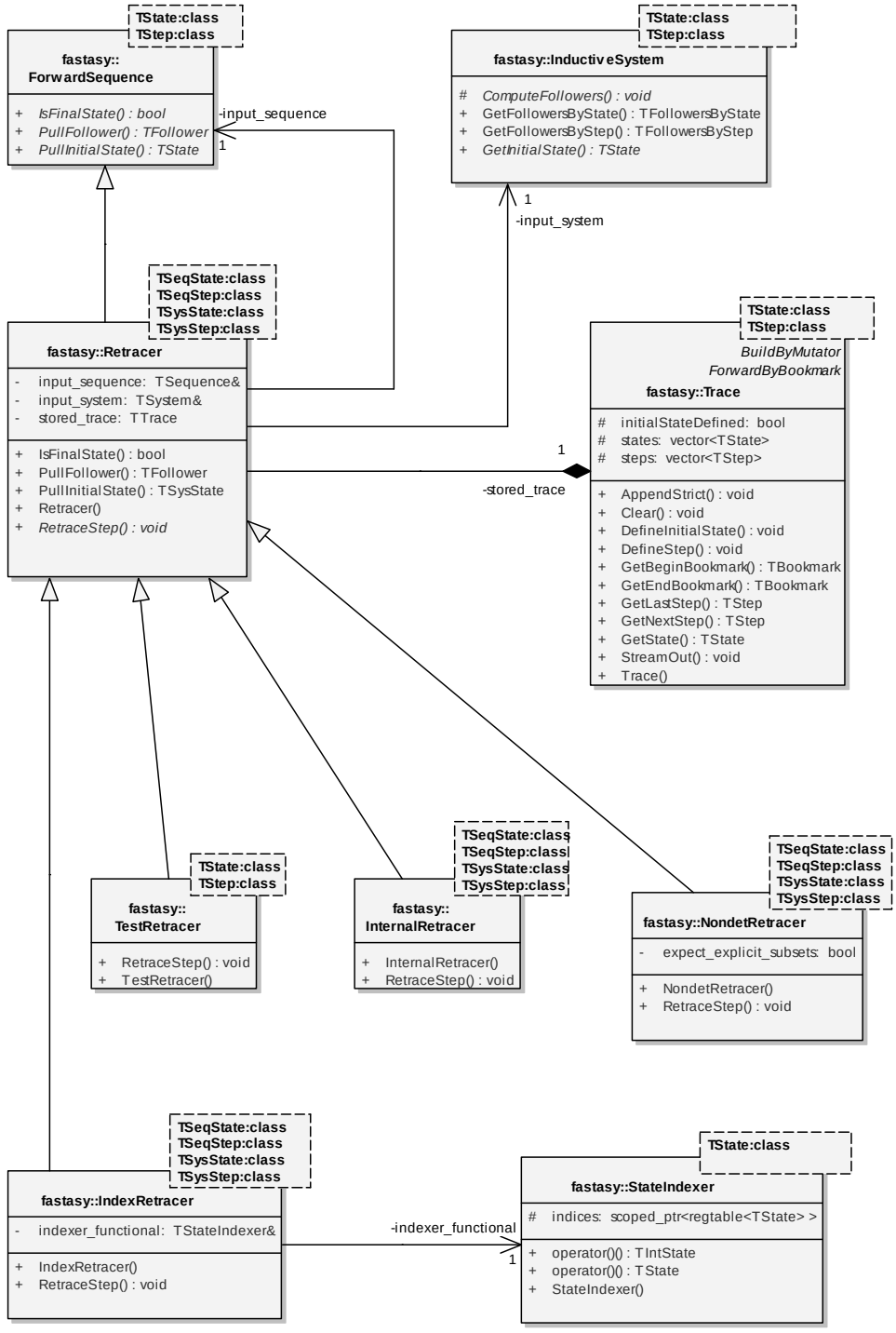



**Abbildung 58: Klassendiagramm Retracer**

### *9.6.3.2 Kollaborationsmechanismen*

#### 9.6.3.2.1 Zusammenhang zwischen PullFollowers() und RetraceStep()

Im folgenden Diagramm sehen wir am Beispiel von *InternalRetracer*, wie Aufrufe von *PullFollowers()* nur dann einen Aufruf von *RetraceStep()* nach sich ziehen, wenn die interne *Trace* abgearbeitet ist (zu sehen am *alt if*-Fragment in der UML-Notation).

Der Vollständigkeit halber ist auch ein Aufruf von *PullInitialState()* zu sehen. Dessen Rückgabewert berechnet sich zwar einfach aus einem Aufruf von *GetInitialState()* des induktiven Systems, aber es ändert sich außerdem der Zustand der *InternalRetracer*-Objekts. Die interne *Trace* wird nämlich zunächst gelöscht (da *PullInitialState()* ja eine Rücksetzung der *ForwardSequence* auf den Anfang beinhaltet) und anschließend mit dem späteren Rückgabewert als Anfangszustand initialisiert. Dies erlaubt die konsequente Verwendung von *AppendStrict()* statt *Append()* im späteren *RetraceStep()*.

Wir bemerken, dass nicht der Versuch gemacht wurde, die genaue Verarbeitungslogik von *InternalRetracer::RetraceStep()* im folgenden Diagramm darzustellen. Dazu sei nur kurz gesagt, dass die iterierten Aufrufe von *GetFollowersByStep()* dazu dienen, eine explorative Suche durch die λ-Hülle von *sysstate* durchzuführen, in deren Verlauf der gesuchte Pfad in *backtrace* gespeichert wird. Anschließend überträgt *AppendStrict()* den gefundenen λ-Pfad in das Ergebnis und *DefineStep()* hängt noch den sichtbaren Schritt an.

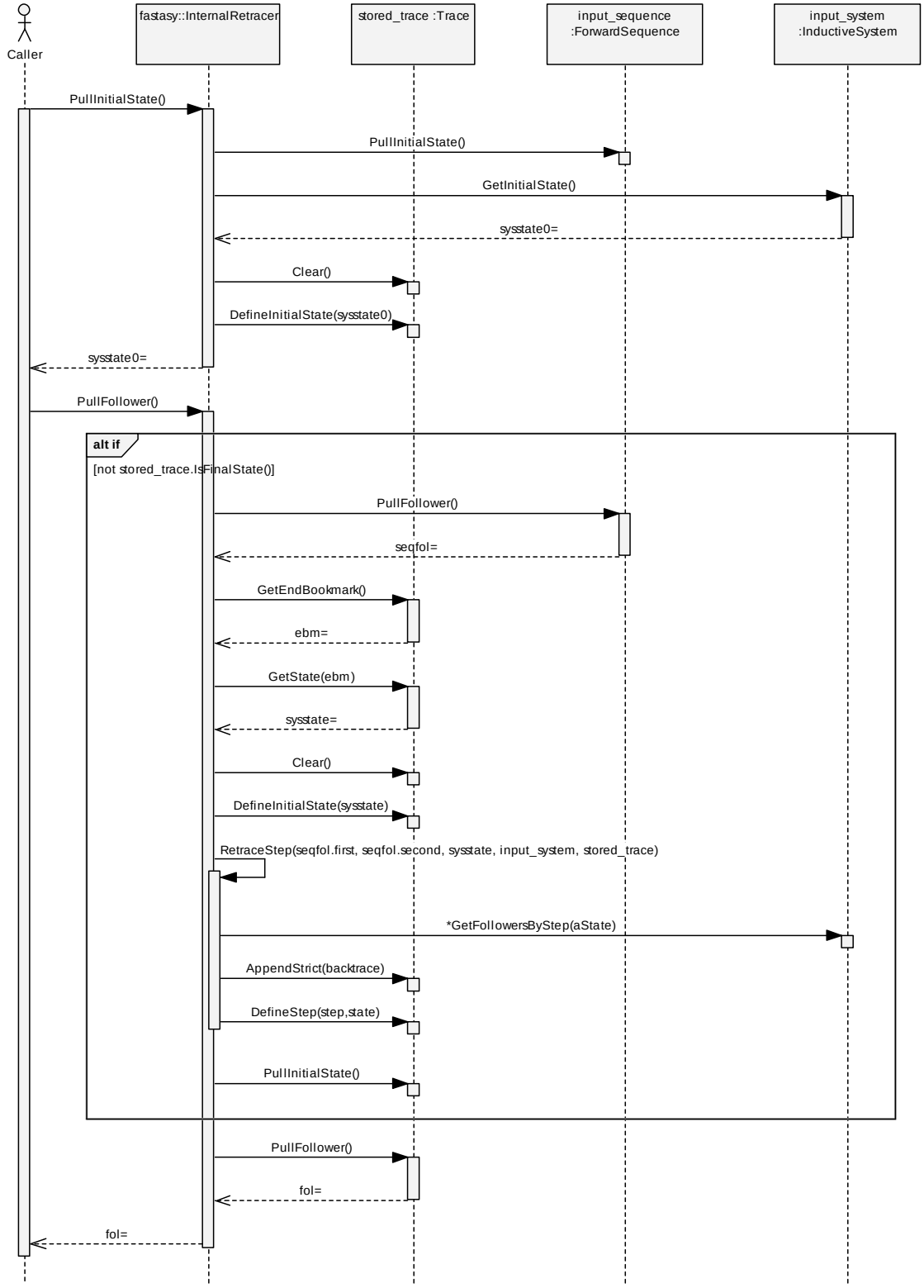



**Abbildung 59: Sequenzdiagramm RetraceStep**

## *9.7 Weitere ausgewählte Komponenten*

In diesem Kapitel sollen noch kurz einige hilfreiche Konstruktionen zur Sprache kommen, die abseits der zentralen Logik von FastAsy anzusiedeln sind und deswegen zum allergrößten Teil in die Bibliothek BE++ verschoben wurden.

Genauere Details zu den einzelnen Utilities entnimmt man der Doxygen-Dokumentation der BE++-Bibliothek und dem Quellcode des Unit-Testing der BE++.

### 9.7.1 I/O-Wrapper

Will man die Inhalte von Container-Objekten in einen Stream ausgeben, so kann beim späteren Einlesen das Problem entstehen, dass das Ende derjenigen Daten, die noch zum Container gehören, nicht klar erkennbar ist. Um zu vermeiden, jedes Mal auf eigens codierte Lösungen für dieses Problem zurückgreifen zu müssen, führen wir das Template *iowrapper* und vier zugehörige freie Funktionen *iowrap(), iolistwrap(), iosetwrap(), ioarraywrap()* ein.

*iowrapper* ist folgendermaßen deklariert:

```
template <class TContainer, char OpenChar, char CloseChar,
    bool CheckSize> class iowrapper;
```

Dabei ist *TContainer* der Typ des Container-Objekts, für den die Ein-/Ausgabe durchgeführt werden soll, *OpenChar* und *CloseChar* sind die Zeichen, die zur Klammerung verwendet werden und *CheckSize* ist ein Flag, welches angibt, ob *iowrapper* für eine korrekte Eingabeoperation verlangt, dass die Größe des Containers (d.h. das Ergebnis eines *size()*-Aufrufs) der Größe der Eingabe entspricht. Ist *CheckSize* nicht gesetzt, so wird die Größe des Containers entsprechend angepasst.

Die vier freien Funktionen übernehmen nun im Stil von STL-Funktionen wie *make_pair()* die Aufgabe, die Erzeugung von *iowrapper*-Instanzen zu vereinfachen, indem sie das *TContainer*-Typargument aus dem Typ ihres (nicht-Template-)Parameters deduzieren. Außerdem unterscheiden sich die vier Funktionen darin, dass sie sinnvolle Werte für die übrigen Templateparameter vordefinieren.

Ein Anwendungsbeispiel für *iowrapper* sähe folgendermaßen aus:

```
vector<int> v;
list<int> l;
set<int> s;
for(int i=0; i<5; ++i) {
    v.push_back( i*i );
    l.push_back( -i );
    s.insert( i );
}

stringstream sout;
sout << bepp::iowrap( v ) << " ";
sout << bepp::iolistwrap( l ) << " ";
sout << bepp::iosetwrap( s ) << " ";
sout << bepp::ioarraywrap( v ) << " ";
```

```
BOOST_CHECK_EQUAL( sout.str(),
    "(0,1,4,9,16) "
    "<0,-1,-2,-3,-4> "
    "{0,1,2,3,4} "
    "[0,1,4,9,16] "
);
```

Dieses Codebeispiel entstammt übrigens dem Unit-Testing-Teil der BE++-Bibliothek. Man beachte, wie derselbe Code, der den Test mittels *BOOST_CHECK_EQUAL* durchführt, dem menschlichen Leser die Erwartung an die Funktionsweise des Programmstücks vor Augen führt. Idealerweise erfüllen Unit-Testing-Module diese Doppelfunktion.

Analog zu *iowrapper* existiert noch ein weiteres Template *quoted_iowrapper* und zugehörige Funktionen *quoted_iowrap()*, *quoted_iolistwrap()*, *quoted_iosetwrap()*, *quoted_ioarraywrap()*. Sowohl Template als auch Funktionen sind in ihren Parametern identisch mit den vorigen Varianten. Der einzige Unterschied besteht darin, dass sie für den Fall ausgelegt sind, dass in dem verwendeten Container-Objekt etwa Zeichenketten gespeichert sind, so dass es sinnvoll ist, sie mit Anführungszeichen zu umgeben, um Ambivalenzen auszuschließen.

Mit den Templates *quotation* und *const_quotation* und der (für *string* und *const string* überladenen) freien Funktion *quoted()* stellt die BE++ diese Möglichkeit natürlich auch für Zeichenketten zur Verfügung, die nicht in einem Container gespeichert sind.

### 9.7.2 Regref

In FastAsy haben wir an vielen Stellen die Situation, dass Objekte (etwa Aktionen, Transitionen, Stellen) mit symbolischen Namen versehen sind. Die einzigen Operationen, die auf diesen Namen ausgeführt werden, sind jedoch Vergleiche und Zuweisungen. Würde man diese Namen nun naiv als Zeichenketten speichern, so stünden zwar noch viele weitere Operationen zur Verfügung, jedoch wären genau diejenigen Operationen, die wir benötigen, nicht in konstanter Zeit möglich, sondern abhängig von der Länger der Namen.

Das Template *regref* stellt nun eine leichtgewichtige Alternative zur naiven Speicherung dar: Beim Zuweisen einer als *string* gespeicherten Zeichenkette oder bei einer Eingabeoperation wird die Zeichenkette in einer Tabelle (welche in einem zentralen Objekt der Klasse *regtable* gespeichert ist) gesucht und ggf. dort eingefügt. Im *regref*-Objekt selbst wird nun nur die Position innerhalb dieser Tabelle gespeichert. Somit reduzieren sich Zuweisungen und Vergleiche zwischen *regref*-Objekten auf Zuweisungen und Vergleiche von *unsigned int*-Werten. Dabei legen die Vergleichsoperatoren natürlich eine beliebige totale Ordnung zu Grunde. Nachdem die Namen aber rein symbolisch sind, existiert keine spezielle Ordnung, die es zu modellieren gilt.

Eine kleine Ausnahme existiert diesbezüglich in FastAsy jedoch: Wir möchten aus gewissen Gründen sicherstellen, dass die leere Zeichenkette (die nämlich für die intern Aktion $\lambda$ steht), immer an Position 0 steht. Dies kann man natürlich aber leicht erreichen, indem man vor irgendwelchen anderen Zuweisungen ein Dummy-Objekt mit der leeren Zeichenkette initialisiert. Tatsächlich existiert eine Spezialisierung des Konstruktors für *string*, der dies

bereits für uns erledigt. (Eine Spezialisierung deshalb, weil *regref* ja ein Template ist und somit auch für andere schwergewichtige Klassen verwendet werden kann statt nur für *string*.)

Man erkauft sich mit *regref* die schnellen Vergleiche und Zuweisungen eben dadurch, dass Initialisierungen aus Zeichenketten oder Eingabeoperationen relative teuer sind, da sie die Tabelle durchsuchen müssen. Diese Initialisierungen finden aber sowieso nur beim Einlesen von Eingabedateien statt und sind somit nie zeitkritisch.

Es findet übrigens kein *reference counting* oder ähnliches statt, d.h. Werte bleiben für den gesamten Programmlauf in der Tabelle, auch wenn kein *regref*-Objekt sie mehr referenziert. Somit ist *regref* nur geeignet für Programme, die die referenzierten Objekte nicht ständig austauschen oder für die die Dauer eines Programmlaufs nicht beliebig wird. Andernfalls bestünde die Gefahr, dass die Tabelle irgendwann überläuft. FastAsy erfüllt offenbar diese Bedingung, da wie bereits erwähnt keine Objekte mehr dazukommen, nachdem die Eingabe eingelesen wurde.

Mit *regref_indirector* existiert noch eine weiterführende Funktionalität, die darauf abzielt, Untermengen einer festen Menge von *regref*-Objekten effizient zu speichern. Instanzen von *regref_indirector* erlauben es, mittels *incorporate()* zunächst beliebig viele *regref*-Objekte in eine Grundemenge aufzunehmen. Nachdem die Grundmenge auf diese Art festgelegt wurde, steht mit *map()* eine Funktion zur Verfügung, für die angegebenen Objekte einen Index zu erhalten. Dieser Index gibt jedoch gerade nicht die Position innerhalb der *regtable* an, sondern befindet sich für *n* verschiedene Objekte in Aufrufen zu *incorporate()* im Bereich *[0..n-1]*. Der Indexbereich wird also für eine ausgewählte Grundmenge so zusammengeschoben, dass er minimale Größe hat. Werden diese Indices nun zur Repräsentation von Untermengen der Grundmenge in einem Bitvektor verwendet, kann dieser Bitvektor ebenfalls mit minimaler Größe erstellt werden.

Natürlich steht mit *unmap()* auch die Umkehrung von *map()* zur Verfügung. Während *map()* in konstanter Zeit ein Ergebnis liefert, benötigt *unmap()* allerdings *log(n)* für *n* verschiedene *regref*-Objekte. Typischerweise wird *unmap()* jedoch erst wieder herangezogen, wenn es um die Ausgabe eines Ergebnisses in einer für den menschlichen Benutzer lesbaren Form geht, so dass wir diesen Aufwand gut akzeptieren können. (Durch Speicherung einer zweiten Tabelle könnte andernfalls aber auch *unmap()* in konstanter Zeit implementiert werden, lediglich der Aufwand für *incorporate()* würde dadurch erhöht.)

### 9.7.3 CliArg

Die zu *CliArg* gehörigen Klassen (im wesentlichen *cliargs*, *cliparam*, *clikeyword*, *cliswitch*) stellen einen Parser für Eingaben von der Kommandozeile dar. *CliArg* funktioniert plattformunabhängig. Unterstützt werden benannte Parameter, Schlüsselworte und Switches, dabei haben Parameter und Schlüsselworte feste Positionen, wohingegen Switches an beliebiger Stelle stehen können und über ihre Namen erkannt werden. Namen von Switches starten mit „-“, und im Anschluss an einen Switch kann dieser noch weitere zu ihm gehörige Parameter akzeptieren.

Der FastAsy-Programmierer instanziert für eine gegebene Kommandozeilen-Syntax, die er verarbeiten möchte, ein *cliargs*-Objekt und je nachdem weitere Objekte vom Typ *cliparam*, *clikeyword* und *cliswitch*, die er dann wie unten ausgeführt an das *cliargs*-Objekt anschließt.

Das so konfigurierte *cliargs*-Objekt ist in der Lage, eine gegebene Kommandozeile erstens auf Konformanz zu prüfen und zweitens aus dieser die Werte für Parameter und Vorhandensein von Switches auszulesen. Das *cliargs*-Objekt generiert außerdem automatisch aus seiner Konfiguration eine kurze Hilfemeldung, in der die zulässigen Kombinationen von Argumenten angegeben sind. Der Benutzer wird so der Aufgabe enthoben, immer parallel die Konfiguration des Parsers *und* den Hilfetext zu ändern.

FastAsy selbst ist über *CliArg* hinausgehend so angelegt, dass es eine ganze Liste von *cliargs*-Objekten enthält und für eine gegebene Kommandozeile das erste passende *cliargs*-Objekt auswählt. Dazu müssen die Syntaxregeln der einzelnen *cliargs*-Objekte sich natürlich gegenseitig ausschließen. Durch die obligatorische Verwendung von *clikeyword*-Objekten wird dies jedoch einfach gewährleistet. (Spätere BE++-Versionen werden solche Listen von *cliargs*-Objekten mit Hilfe einer zentralen *cliargslist*-Klasse selbst verwalten können.)

Der folgende Code ist an den tatsächlichen Quelltext von FastAsy angelehnt und soll die Handhabung von *CliArg* exemplarisch verdeutlichen. Nehmen wir an, wir wollten in FastAsy nur die folgenden beiden Aufrufe unterstützen:

```
FastAsy NDEA [–o filename1] filename2
```

und

```
FastAsy DEA [–o filename1] [-v] filename2
```

Alternativ zu den Kurznamen *–o* und *–v* der Switches sollen die ausgeschriebenen Namen *--outfile* und *--verbose* akzeptiert werden.

Dazu schreiben wir folgenden Code:

```
enum arg_version_names { av_UNDEF=-1,
    av_ndea,
    av_dea,
    av_END_LIST
};
const int arg_version_count = av_END_LIST;
cliargs arg_versions[arg_version_count];
```

Für jede gültige Version einer Kommandozeile wird hier ein *cliargs*-Objekt angelegt. Der Aufzählungstyp dient lediglich dazu, den Quelltext klarer verständlich zu machen, genauso gut könnten wir natürlich direkt mit *int*-Werten arbeiten.

```
clikeyword k_ndea("NDEA");
clikeyword k_dea("DEA");
cliparam p_netfile("netfile");
const char* s_outfile_par[] = {"filename"};
cliswitch s_outfile("outfile","o",1,s_outfile_par);
cliswitch s_verbose("verbose","v");
```

Hier werden zwei Schlüsselworte NDEA und DEA definiert, über die später die Version der Kommandozeilen-Syntax ausgewählt werden soll. Außerdem definieren wir den Namen des Eingabefiles noch mit *cliparam* als positionalen Parameter und fügen zwei positionsunabhängige Switches *s_outfile* und *s_verbose* hinzu. Dabei soll *s_outfile* noch einen

Parameter nach sich ziehen, nämlich den Namen einer Ausgabedatei. Für die Switches ist jeweils der Name der Kurz- und der Langversion definiert.

```
arg_versions[av_ndea] << k_ndea << s_outfile << p_netfile;
arg_versions[av_dea] << k_dea << s_outfile << s_verbose << p_netfile;
```

Hier wurden nun die zwei Kommandozeilen-Versionen definiert. (Zu Demonstrationszwecken haben wir beschlossen, dass nur für die DEA-Version der *verbose*-Switch angegeben werden kann.) Die Verwendung des Operators << ermöglicht eine kompakte und intuitive Notation, hat aber natürlich nichts mit Stream-Ausgabe zu tun.

```
int version_detected = av_UNDEF;
for(int i=0; i<arg_version_count; ++i) {
    if( arg_versions[i].safe_parse( argc, const_cast<const char**>(argv)
) ) {
        version_detected = i;
        break;
    }
}
```

In einer Schleife wird nun diejenige Kommandozeilen-Version gesucht, die auf die tatsächliche Eingabe auf der Kommandozeile passt. Dazu wird für jedes *cliargs*-Objekt die Methode *safe_parse()* aufgerufen, welche genau dann *wahr* zurückgibt, wenn sich alle untergeordneten Objekte auf Angaben aus der Kommandozeile abbilden lassen (ggf. bis auf optionale Switches natürlich) und umgekehrt. In diesem Falle enthalten dann die Objekte der Switches und Parameter weitere Informationen.

```
string ofname;
if( s_outfile.given() ) {
    ofname = s_outfile.argv();
} else {
    ofname = "FastAsy.txt";
}
```

Hier wird beispielsweise für den Switch *s_outfile* zunächst mit der Methode *given()* abgefragt, ob er überhaupt auf der Kommandozeile angegeben war und für diesen Fall dann mit *argv()* der Wert des nachgezogenen Parameters ermittelt. (Für den Fall, dass mehrere Parameter nachgezogen werden, wird *argv( int n )* verwendet.)

```
switch( version_detected ) {
    ...
```

Zu guter Letzt wird in Abhängigkeit von der aufgerufenen Version der Kommandozeile (wir erinnern uns, dass diese durch ein Schlüsselwort bestimmt wurde) bedingter Code aufgerufen, der in FastAsy z.B. die Ausführung an die jeweiligen Fachfassaden delegiert (siehe 9.2.1.2).

### 9.7.4 Powset_Iterator/Subset_Iterator

In FastAsy werden Mengen mehrfach durch Bitvektoren (konkret durch *boost::dynamic_bitset*) dargestellt. Weiter müssen wir uns gerade im Umgang mit der Thematik, den Teilmengenabschluss von Verweigerungsmengen aus der expliziten Darstellung zu eliminieren, oft die Potenzmenge bzw. alle Teilmengen einer derart dargestellten Menge verschaffen. Die dazu nötigen Schritte wollten wir gerne an einer zentralen Stelle kapseln, so dass fachlicher Code entlastet und dessen Komplexität vermindert wird. Unsere Lösung besteht aus zwei Templates *subset_iterator* und *powset_iterator*, welche sich größtenteils sehr ähnlich sind.

Beide besitzen eine Iterator-Semantik und stellen diese über dasselbe Interface bereit. Dieses Interface wurde absichtlich nicht exakt nach dem Muster der STL-Iteratoren gestaltet. Für STL-Iteratoren ist nämlich die Abfrage, ob ein Iterator das Ende seines Bereichs erreicht hat, notationell etwas schwerfällig:

```
container<value_type> C;
…
container<value_type>::iterator I;
for( I=C.begin(); I!=C.end(); ++I )
      do_something_with( *I );
```

Insbesondere entstünde in unserem Zusammenhang ja auch das Problem, dass der Containertyp bereits gegeben ist und dessen Methoden *begin()* und *end()* ja die herkömmlichen Iteratoren für diesen Container erzeugen. Wir müssten also, um konform mit der STL-Notation zu sein, eine Adapterklasse anlegen (etwa *subset*<*T*> und *powset*<*T*>), welche dann über *begin()* und *end()* die neuen Iteratoren zurückgibt. Dieser Aufwand schien uns nicht gerechtfertigt, und so haben wir stattdessen *begin()* und *end()* als nicht-*const* Operationen in *subset_iterator* und *powset_iterator* mit der offensichtlichen Semantik integriert, den Iterator auf die jeweilige Position zu setzen. Wie häufig in nicht-STL-Iteratoren der Fall, wird per *operator bool()* (d.h. eine Konversion) überprüft, ob ein Iterator gültig ist. Bewegen des Iterators erfolgt wie üblich durch ++ und --, die Dereferenzierung geschieht durch *. Damit sieht obiges Beispiel folgendermaßen aus:

```
bitset_type C;
…
subset_iterator<bitset_type> I( C );
for( I.begin(); I; ++I )
      do_something_with( *I );
```

Die Funktion *do_something_with()* würde in diesem Beispiel mit allen Teilmengen der durch *C* repräsentierten Menge aufgerufen. (Von *bitset_type* wird dabei erwartet, dass er sich wie *std::bitset* bzw. *boost:dynamic_bitset* verhält.)

Der erwähnte Unterschied zu *powset_iterator* besteht nun darin, dass letzterer keine Menge in seinem Konstruktor erwartet, sondern lediglich die Angabe, wie viele Elemente die Menge haben soll:

```
bitset_type::size_type n;
powset_iterator<bitset_type> I( n );
```

```
for( I.begin(); I; ++I )
      do_something_with( *I );
```

In diesem Beispiel erfolgt die Iteration dann einfach über alle $2^n$ Elemente von *Pow({0..n})*. Im Gegensatz zu *subset_iterator* erspart man sich hier zunächst natürlich die Konstruktion eines Containers *C*, der dann mit der vollen Menge initialisiert werden müsste. Weiter arbeitet *powset_iterator* intern aber auch effizienter.

### 9.7.5 Reference_ptr

Um das Template *reference_ptr* zu motivieren, müssen wir ein wenig ausholen: Bekanntlich hat man sich beim Entwurf der STL darauf geeinigt, dass Container Werte der Typen, mit denen sie instanziert werden, speichern sollen und keine Referenzen. Dies hat zur Folge, dass Objekte immer nur in einem einzigen Container gespeichert sein können, weiterhin findet sowohl beim Schreiben als auch beim Lesen von Objekten ein impliziter Kopiervorgang statt. Die Gründe für diese Entwurfsentscheidung sind unter anderem darin zu sehen, dass die umgesetzte Semantik prinzipiell das flexiblere Konzept darstellt, immerhin könnten Container auch mit Referenz- oder Zeigertypen instanziert werden, wenn dies gewünscht wird. Besieht man sich aber die Details der STL, bemerkt man schnell, dass eine Instanzierung mit einem Referenztyp aus rein praktischen Gründen zu keinem sinnvollen Ergebnis führt; man überlege sich allein, dass es ja in C++ keine uninitialisierten oder leeren Referenzen gibt.

Instanzierungen mit Zeigertypen sind hingegen sowohl möglich als auch korrekt, allein treffen sie in den meisten Fällen nicht die intuitive Semantik: Gleichheit von Objekten meint in diesem Falle nämlich tatsächlich die Objektidentität im Speicher, und noch fataler finden Vergleiche auf Basis der Speicheradresse von Objekten statt. Beides dürfte nur selten wünschenswert sein.

Für den geübten Programmierer ist es jedoch eine triviale Aufgabe, ein Template zu konstruieren, welches die gewünschte hybride Semantik aus Zeiger und Referenz in sich vereint. Aus unerfindlichen Gründen ist ein solches Template bislang nicht Bestandteil der STL. Eine beispielhafte und kommentierte Umsetzung einer solchen Klasse findet der Leser etwa in [JN96], so dass wir hier nicht weiter darauf eingehen. Das Template *reference_ptr* in der BE++ erweitert lediglich die obige Umsetzung um Details wie Konversionsoperatoren oder direkte Vergleiche mit dem ursprünglichen Typ.

### 9.7.6 ProgressCounter

Die Klasse *progress_counter* stellt eine einfache Möglichkeit für Programme dar, den Benutzer über den Fortschritt langwieriger Prozesse auf dem Laufenden zu halten, ohne den Quellcode mit übermäßig viel Code ohne Fachfunktionalität zu überfrachten.

Objekte vom Typ *progress_counter* werden bei ihrer Konstruktion mit einem symbolischen Namen und einem ganzzahligen Wert *n* initialisiert. Außerdem besteht die Möglichkeit, sie über *set_ostream* mit einem beliebigen Ausgabe-Stream zu verbinden. Wird kein solcher Ausgabe-Stream eingestellt, so findet die Ausgabe über das *BOOST_MESSAGE*-Makro statt, falls der *Programm Execution Monitor* von Boost benutzt wird, oder andernfalls über *clog*, den Log-Stream der Konsole.

Innerhalb der Schleife, über die der Benutzer informiert werden soll, wird nun auf dem Objekt der ++-Operator aufgerufen. Jedes *n*-te Mal findet dann eine kurze Ausgabe auf dem oben erwähnten Weg statt, die den symbolischen Namen und die Fortschrittszahl meldet.

FastAsy verwendet *progress_counter*-Objekte in erster Linie beim Aufbau von großen Objekten wie Materialisierungen induktiver Systeme.

### 9.7.7 DebugLog

Mit *DEBUGLOG* und *DDEBUGLOG* stehen zwei Makros zur Verfügung, um Kontrollmeldungen, die in erster Linie für Entwickler von FastAsy interessant sind, während des Programmlaufs auf die Konsole auszugeben.

In der Version *DEBUGLOG* wird dabei die Präprozessor-Konstante *NDEBUG* abgeprüft. Nur wenn diese nicht definiert ist, wird das Makro zu Code expandiert, ansonsten ist die Expansion leer. *NDEBUG* wird beispielsweise auch von *assertion* bzw. der STL-Version *assert* geprüft. Im Normalfall wird diese vom Compiler beim Übersetzen von Produktivcode gesetzt und ansonsten nicht.

Die Version *DDEBUGLOG* verwendet hingegen eine globale Variable *DLOGLEVEL*, welche andernorts definiert sein muss, um dynamisch zu entscheiden, ob eine Ausgabe erfolgt. Man beachte, dass *DDEBUGLOG* somit zwar flexibler ist, im Gegensatz zu ihrem statischen Pendant aber in jedem Falle (wenn auch geringe) Auswirkungen auf die Programmlaufzeit hat.

### 9.7.8 Assertion und DebugOnly

Das Makro *assertion* aus der BE++ ist eine Variante des STL-Makros *assert*. Ebenso wie dieses wird es nur bei nicht gesetztem *NDEBUG* (siehe 9.7.7) zu Code expandiert. Sein Argument ist ein boolescher Ausdruck. Ist dieser Ausdruck zur Laufzeit *false*, so wird als Fehlermeldung der Ausdruck selbst sowie der Ort des Auftretens im Quelltext ausgegeben. Während nun aber *assert* den Programmlauf sofort beendet, wirft *assertion* lediglich eine Ausnahme *assertion_failed*. Letzteres hat im Rahmen des Unit Testing den entscheidenden Vorteil, dass getestet werden kann, ob zu testender Code bei verletzten Vorbedingungen korrekt, d.h. mit dem Auslösen der Fehlerbedingung der Zusicherung reagiert.

Weiterhin existieren in der BE++ noch eine Variante *custom_assertion*, die statt *assertion_failed* eine benutzerdefinierte Ausnahme werfen kann, eine Variante *rassertion*, welche *NDEBUG* ignoriert, und letztlich mit *custom_rassertion* eine Kombination aus beiden.

Für den Fall, dass die Formulierung einer Zusicherung zu aufwändig ist, um sie als booleschen Ausdruck zu schreiben, existiert noch ein Makro *DEBUGONLY*, mit welchem zusätzlicher Prüfcode umschlossen werden kann, so dass auch er nur bei nicht definiertem *NDEBUG* übersetzt wird.

### 9.7.9 Exceptions

In FastAsy werden außerdem folgende Ausnahmen definiert:

- *fastasy_exception* stellt die Basisklasse aller weiteren speziell für FastAsy definierten Ausnahmen dar.

- *firing_rule_error* wird aus dem Package „Petrinetz“ geworfen, wenn die Schaltregel feststellt, dass eine dynamische Zusicherung an die erreichbaren Markierungen (in unserem Falle Sicherheit) verletzt wird.
- *semantics_error* wird von Bausteinen aus dem Package „Transformation“ geworfen, wenn eine Zusicherung der Eingabe (etwa Determinismus) verletzt wird.
- *retrace_error* wird aus dem Package „Retracing“ geworfen, wenn ein *Retracer* seine Eingabesequenz nicht im gegebenen induktiven System finden kann.
- *fastasy_cli_error* wird aus den Fachfassaden heraus geworfen, wenn eine Anforderung aus der CLI-Schicht auf Grund fehlerhafter Parameter nicht beantwortet werden konnte (etwa weil eine Eingabedatei nicht existiert).

# Kapitel 10 Developers' Tutorial

Im folgenden Kapitel wollen wir an Hand eines Beispiels eine Einführung in die Erweiterung von FastAsy geben. Wir entwickeln eine Möglichkeit, Ausgaben des PAFAS-Tools (eines Generators erreichbarer Zustände für die zeitbehaftete Prozessalgebra PAFAS, siehe [CVJ02], [PFS]), in FastAsy einzulesen und Simulationen darauf durchzuführen.

Der Natur der Sache gemäß werden Schilderungen dieses Kapitels etwas detailreicher ausfallen und teilweise bis auf Quellcodeebene herabreichen.

## *10.1 Anforderungen*

### 10.1.1 Schnittstelle

Um eine Verbindung zwischen PAFAS und FastAsy herzustellen, bedienen wir uns der Übergabe von Textdateien. PAFAS unterstützt mehrere Ausgabeformate, im folgenden sehen wir ein Beispiel für dasjenige Format, welches wir einlesen wollen:

```
Q
10"B = b.nil|{b}|b.C"
20"nil|{b}|C"
30"_b.nil|{b}|_b.C"
40"nil|{b}|(_b.nil|{a}|C)"
50"nil|{b}|(_a._b.nil|{a}|_a.C)"
60"nil|{b}|(_b.nil|{a}|(_a._b.nil|{a}|_a.C))"
E
10>20"b"
10>30"{b,a}"
20>40"a"
20>50"{b,a}"
30>20"b"
30>30"{a}"
40>60"{b,a}"
50>40"a"
50>50"{b}"
60>60"{b,a}"
```

Das PAFAS-Format orientiert sich größtenteils an demjenigen, das FastAsy auch zur Persistierung seiner eigenen Automaten benutzt, es sind jedoch einige Unterschiede zu beachten:

- Zwischen Zustandsnummer und Zustandsbeschreibung steht kein Leerzeichen
- Die Zustandsinformation besteht aus durch Anführungszeichen umschlossenem Text (dessen Inhalt FastAsy aber nicht analysieren, sondern lediglich für Vergleiche speichern muss)
- Zwischen Quell- und Zielzustand und der Übergangsbeschreibung steht ebenfalls kein Leerzeichen
- Übergänge werden ebenfalls als durch Anführungszeichen umschlossener Text angegeben.

Der Text muss allerdings von FastAsy analysiert werden, um zu einer internen Darstellung der Aktionen bzw. Verweigerungsmengen zu kommen. Dies ist notwendig, weil das PAFAS-Tool alle Teilmengen der Verweigerungsmengen implizit lässt und deshalb FastAsy diese berechnen muss.

### 10.1.2 Interne Repräsentation

Es erscheint sinnvoll, zur internen Speicherung des PAFAS-Automaten einen FastAsy-*Automaton* (siehe 9.4.2.2) heranzuziehen. Zur Repräsentation der Zustandsinformation kommt zunächst ein bloßer *string* aus der STL in Frage, wir werden jedoch sehen, dass es vorteilhaft ist, einen *Wrapper* dafür einzusetzen. Die Information auf den Kanten zwischen den Zuständen entspricht genau der eines *UntaggedRTState* (siehe 9.4.2.3 bzw. 9.4.3.1.6). Wir bemerken an dieser Stelle nur bereits, dass das Einlesen eines *UntaggedRTState* über die dort implementierte Streaming-Operation nicht ausreichend sein wird, da erstens natürlich das Format abweicht, zweitens aber bei den Verweigerungsmengen die Angaben über die Grundmenge fehlen und erst beschafft werden müssen.

### 10.1.3 Einlesen der Eingabe

Durch die Verwendung von *Automaton* bietet es sich an, einen I/O-Adapter (siehe 9.4.3.1.3) für PAFAS zu implementieren. Wie erwähnt, existiert dabei allerdings ein prinzipielles Problem: Während FastAsy bei der Ausgabe von Übergängen erstens die Größe der Grundmenge (und damit des *dynamic_bitset*) anführt und zweitens die Verweigerungsmengen als Bitvektor exportiert, müssen wir uns bei PAFAS die nötigen Informationen erst beschaffen. Dazu müssen in einem ersten Durchgang alle Übergänge gelesen und die Aktionsbeschriftungen in eine Tabelle eingefügt werden. Diese Tabelle wird später auch zur Übersetzung von Aktionsnamen in Positionen innerhalb des Bitvektors verwendet werden. Man beachte, dass die Tabelle aus einem Grund unbedingt sortiert sein muss: Beim Vergleich zweier PAFAS-Eingaben müssen gleiche Aktionsnamen in Verweigerungsmengen gleich codiert werden. Selbstverständlich müssen auch die Aktionsnamen, die für Aktionen selbst stehen, konsistent codiert werden. Dies jedoch wird von *UntaggedRTState* automatisch sichergestellt, da zur Speicherung *strref* (siehe 9.7.2) und somit eine globale *regtable* verwendet wird. Die Codierung aus dieser *regtable* kommt jedoch nicht in Frage, da sie im Allgemeinen nicht nur Aktionsnamen enthält und deshalb nicht kompakt genug ist; wir erinnern uns in diesem Zusammenhang an die Klasse *regref_indirector*, mit deren Hilfe bei den Petrinetzen aus der globalen *regtable* ein Slice berechnet wird (siehe 9.3.2.6).

### 10.1.4 Unit Tests

Um die gleiche Codequalität wie im Rest von FastAsy zu garantieren, wollen wir für die neu zu erstellenden Klassen auch Testfälle generieren und in das Test Framework einbinden.

### 10.1.5 Einbinden in das CLI

Zu guter Letzt müssen wir es dem Anwender noch ermöglichen, die neue Funktionalität auch aufzurufen, weshalb wir uns noch damit zu beschäftigen haben, wie wir FastAsy eine

neue Fachfassade hinzufügen (siehe 9.2.1.2) und diese von der UI aus aufrufen (siehe 9.2.1.1).

## *10.2 Umsetzung*

### 10.2.1 Klasse PafasInductiveSystem

Obwohl wir wie bereits erwähnt *Automaton* sowohl für eine Speicherung des PAFAS-Systems als auch für das Einlesen von Eingabedateien heranziehen wollen, leiten wir *PafasInductiveSystem* von *InductiveSystem* ab, nicht von *Automaton*. Obwohl dies vielleicht zunächst überraschend erscheinen mag, macht man sich schnell klar, dass dies das konzeptionell überzeugendere Vorgehen ist: Innerhalb von FastAsy soll *PafasInductiveSystem* eben nur als *InductiveSystem* agieren. Schon das Vorhandensein der öffentlichen(!) Methode *Build()* in *Automaton* macht klar, dass es falsch wäre, diese in der geplanten Klasse zu erben. Die Benutzung von *Automaton* ist streng genommen ja nur ein geschickter Missbrauch von vorhandenem Code, der deswegen möglich ist, weil Pafas-Systeme eben auch nur endliche Transitionssysteme sind. So gesehen wird klar, dass wir die Verwendung von *Automaton* stärker kapseln müssen, als uns dies eine (öffentliche) Vererbung erlauben würden: Wir entschließen uns deshalb, die Funktionalität von Automaton über eine Komposition zu integrieren.

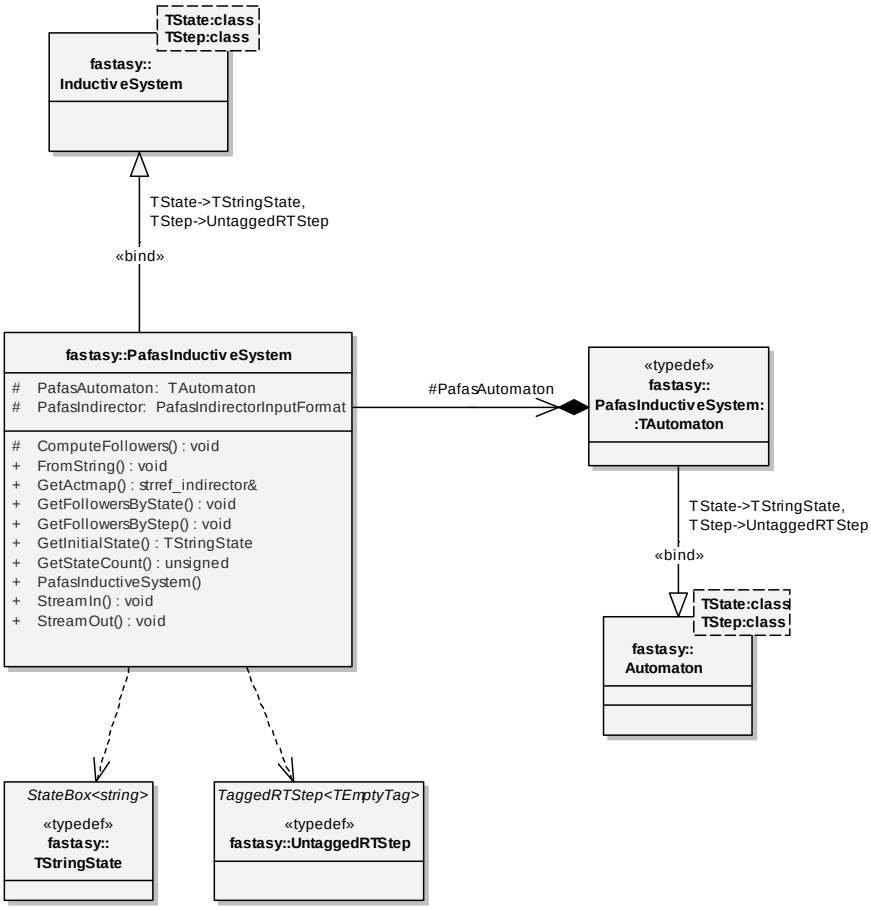



**Abbildung 60: Klassendiagramm PafasInductiveSystem**

Wie man sieht, sind dabei die Template-Parameter von Automaton schon festgelegt. Für die Schritte verwenden wir wie schon erwähnt *UntaggedRTStep*, über den Zustandstyp soll

gleich noch mehr gesagt werden. Mit denselben Parameterwerten wird auch das *InductiveSystem*-Template instanziert, dessen Interface wir implementieren.

Alle lesenden Methodenaufrufe werden einfach an das *Automaton*-Objekt weitergereicht. Die Methoden *StreamIn() und FromString()* dienen der Initialisierung des Systems (s. 10.2.3).

### 10.2.2 Zustandstyp TStringState

Es wurde bereits angemerkt, dass die Verwendung eines bloßen *string* als Typparameter zur Repräsentation von Zuständen problematisch wäre. Dies hat folgende Gründe:

- Die Deserialisierung aus einem Stream sollte erst bei einem abschließenden Anführungszeichen beendet werden. Im Falle von *string* würde das Einlesen jedoch normalerweise durch *whitespace* beendet.
- Die Anführungszeichen selbst sollten nach Möglichkeit nicht Teil der internen Zustandsbeschreibung sein, da sie völlig redundant sind.
- Die Transformationsvorschrift *StateIndexer* verwendet eine *regtable<TStateType>* zur Bildung des Index. Nun existiert aber für *regtable<std::string>* eine Template-Spezialisierung des Konstruktors, die bewirkt, dass die leere Zeichenkette immer den Index 0 erhält. Folglich wären alle vergebenen Indices um 1 höher als erwartet, was wiederum größere Bitvektoren in *TIntSetState* erfordern würde.

Zur Lösung des Problems möchten wir einen *Wrapper* um *string* einsetzen. Jedoch wollen wir keine konventionelle *Wrapperklasse* konstruieren, da wir ja sonst ein *boxing* einführen würden, welches wir wie in 9.4.1.2 beschrieben ja unbedingt vermeiden wollen.

Eine elegante Möglichkeit an dieser Stelle ist es, ein Template *StateBox<T>* einzuführen, welches alle für die Behandlung als Zustand notwendigen Operationen von T nach außen führt, durch die Implementierung als Template mit *inline*-Methoden jedoch keinen Overhead erzeugt.

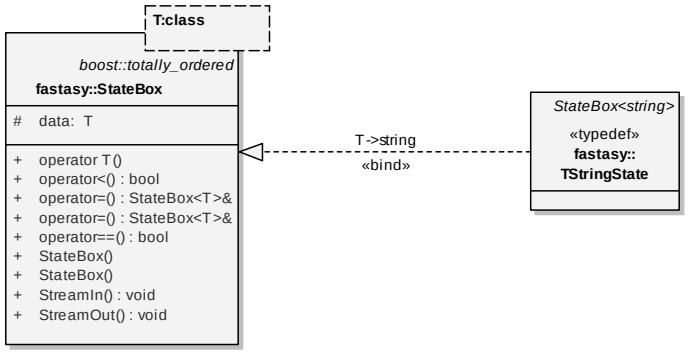



**Abbildung 61: Klassendiagramm StringState**

Die IO-Methoden *StreamIn()* und *StreamOut()* sind für den Typparameter *string* spezialisiert, so dass sie in diesem Fall das *quotation*-Template aus der BE++ (siehe 9.7.1) verwenden, um komfortabel durch Anführungszeichen begrenzte Strings einzulesen. Weiter implementieren wir die üblichen Streaming-Operatoren, welche lediglich noch an *StreamIn()* und *StreamOut()* delegieren müssen.

Es sei noch erwähnt, dass die Verwendung der *StateBox* noch einen weiteren Vorteil mit sich bringt: Da die zur Verfügung stehenden Operatoren gegenüber dem blanken Datentyp

begrenzt werden, kann *StateBox<T>* gegenüber *T* ungewollte Abhängigkeiten eines Templates von Operatoren von *T* aufdecken. In diesem Sinne kann es andernorts sinnvoll sein, in Fällen, in denen Zustände zwar durch *int* modelliert werden, diese spezielle Tatsache aber nicht für Berechnungen herangezogen wird, stattdessen *StateBox<int>* zu verwenden.

### 10.2.3 Klassen zum Einlesen der Eingabe

Wir vergegenwärtigen uns zunächst, in welcher Form das Einlesen eines *Automaton* generalisiert ist: Man könnte versucht sein, die normale *StreamIn()*-Methode von *Automaton* aufzurufen und würde aber spätestens im zweiten Abschnitt der Eingabe scheitern. Die Notation der Verweigerungsmengen in der Eingabedatei kann nämlich von *UntaggedRTState* nicht eingelesen werden. Wäre dies nur durch die äußere Form bedingt, könnte man etwa durch Ableiten von *UntaggedRTState* und Überschreiben der dortigen *StreamIn()*-Methode mit wenig Aufwand einen neuen Zustandstyp schaffen, welcher das Problem löst. Wir haben jedoch bereits ausgeführt, dass das Problem tieferliegend ist: Wir benötigen zunächst eine Übersicht über alle vorhandenen Aktionsnamen, bevor wir tatsächlich eine Speicherung der Verweigerungsmengen als Bitvektor durchführen können. Folglich werden wir zwei Durchgänge auf dem Eingabefile benötigen.

Ähnlich wie bei der in 9.4.3.2.2 beschriebenen Ausgabe eines *Automaton* durch *StreamOutFormat()* existiert auch eine konfigurierbare Eingabe. Die dortige Methode hört auf den Namen *StreamInFormat()* und wird ebenfalls durch eine als Typparameter übergebene *Strategie* in ihrem Verhalten beeinflusst. S*treamInFormat* hat folgende Signatur:

```
template<class TState, class TStep>
void Automaton<TState, TStep>::
StreamInFormat(
  istream& sin,
  AutomatonInputFormat<TState, TStep> &fmt
)
```

Das Interface der zu implementierenden Strategie *AutomatonInputFormat* ist weiter wie folgt definiert:

```
template<class TState, class TStep>
class AutomatonInputFormat
{
public:
    virtual bool CheckHeader(string line) = 0;
    virtual bool ReadState(string line, TState& state,
      int& state_num) = 0;
    virtual bool CheckSeparator(string line) = 0;
    virtual bool ReadStep(string line, TStep& step, int& src_state_num,
      int& dest_state_num) = 0;
    virtual bool CheckFooter(string line) {return false;}
};
```

Die einzelnen Methoden haben dabei folgende Bedeutung:

- *CheckHeader()* prüft für eine Zeile, ob sie zum Dateivorspann gehört. In unserem Fall müssen wir also *line* einfach auf Gleichheit mit „Q“ prüfen. Nachdem *CheckHeader()* einen wahren Wert zurückliefert, werden die folgenden Zeilen als Zustandsbeschreibungen interpretiert und in Folge dessen für jede Zeile *ReadState()* aufgerufen.
- Analog entscheidet *CheckSeparator()*, ob der Abschnitt mit den Übergangsbeschreibungen beginnen soll. Wiederum müssen wir in unserem Beispiel *line* nur auf Gleichheit mit „E“ prüfen.
- *CheckFooter()* könnte noch evtl. vorhandene Fußzeilen verarbeiten, welche aber in unserem Fall nicht vorkommen. Wir können daher die Default-Implementierung, welche immer *false* zurückgibt, unverändert lassen.
- *ReadState()* wird für jede Zeile aufgerufen, die einen Zustand beschreibt. Die Implementierung muss die beiden Referenzparameter *state* und *state_num* ausfüllen. (Dabei muss *state_num* natürlich nicht fortlaufend, sondern nur eindeutig sein.)
- *ReadStep()* funktioniert analog für Übergänge, wobei *src_state_num* und *dest_state_num* die in *ReadState()* vergebenen Nummern des Anfangs- und Zielzustands des Übergangs sind und *step* natürlich die Beschriftung des Übergangs.

Es fällt auf, dass *ReadState()* und *ReadStep()* noch jeweils einen Wahrheitswert zurückliefern, dessen Bedeutung nicht sofort ersichtlich ist. Tatsächlich wird dieser aber sehr wichtig für uns sein, da wir ja eine zweistufige Verarbeitung des Eingabefiles vorhaben. Indem nämlich aus den genannten Methoden *false* zurückgeliefert wird, wird kein Zustand bzw. kein Übergang eingefügt und die Information in *state* und *state_num* bzw. *step*, src_*state_num* und *dest_state_num* wird verworfen.

Aufgrund der besonderen Anforderung, das Eingabefile in zwei getrennten Durchgängen zu verarbeiten, werden wir keinen Nachfolger von *AutomatonInputFormat* bauen können, der die Aufgabe in einem einzelnen Aufruf von *StreamInFormat()* erledigt. Das durch die Rückgabe nur optionale Einfügen von Zuständen bzw. Übergängen erlaubt uns jedoch folgendes Vorgehen: Wir leiten zwei Klassen von *AutomatonInputFormat* ab, nämlich *PafasIndirectorInputFormat* und *AutomatonPafasInputFormat*. Wie im folgenden beschrieben, wird erstere dafür verantwortlich sein, die Informationen über Aktionsbeschriftungen zu sammeln (und dabei weder Zustände noch Schritte einfügen) und letztere wird in einem zweiten Durchgang dann mit Hilfe dieser Informationen den Automaten aufbauen. Dies bedingt natürlich zwei aufeinander folgende Aufrufe von *StreamInFormat()*, welche wir in der *StreamIn()*-Methode von *PafasInductiveSystem* zusammenfassen.

Das folgende Klassendiagramm zeigt die am Einlesen beteiligten Klassen in der Übersicht.

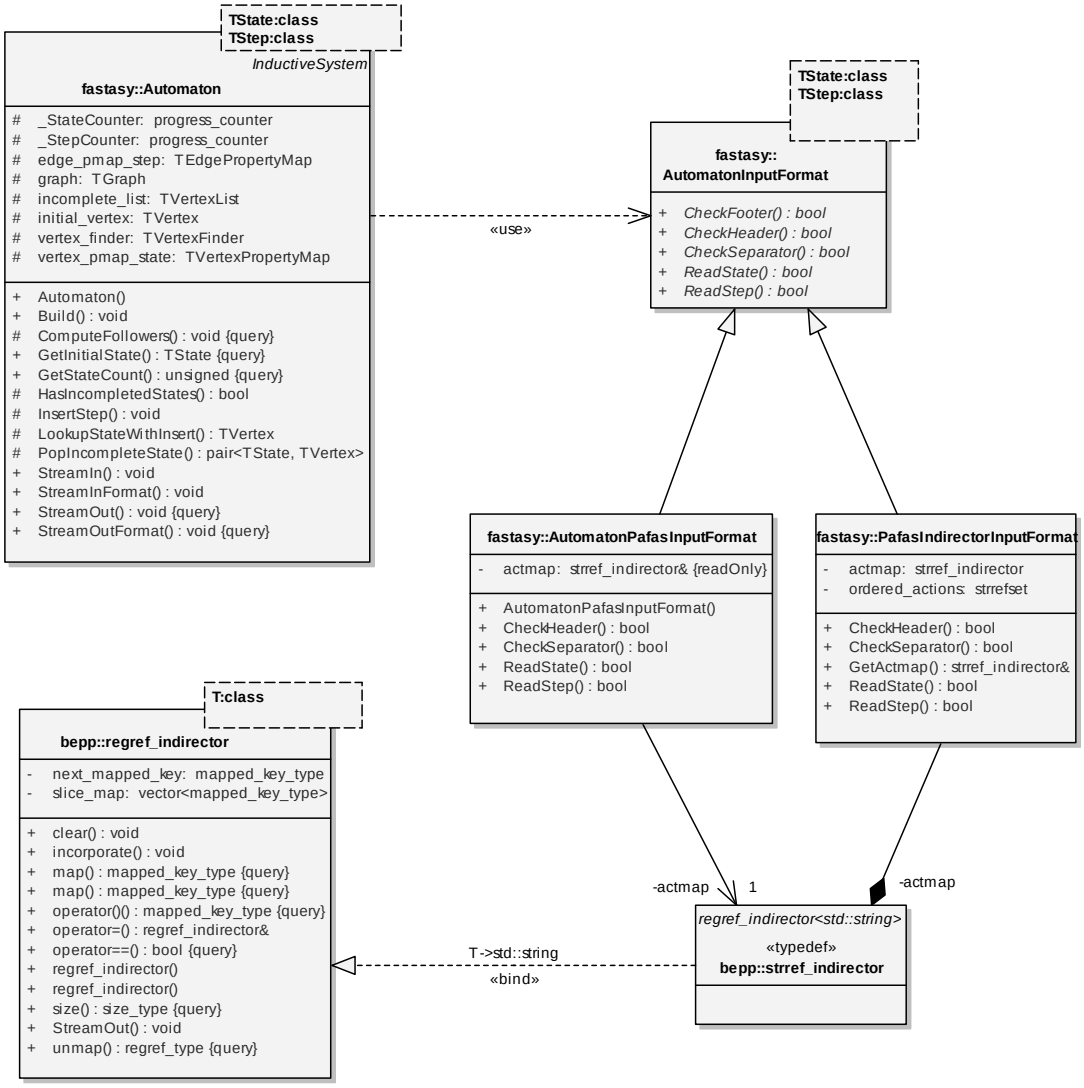



**Abbildung 62: Klassendiagramm Pafas-Input**

Wie wir sehen, sammelt *PafasIndirectorInputFormat* alle Aktionsbeschriftungen in einer Tabelle, als deren Implementierung wir sinnvollerweise die Klasse *strref_indirector* (s. 9.7.2) verwenden. Essentiell dabei ist aber, dass beim Einlesen eines anderen PAFAS-Systems mit denselben Aktionsbeschriftungen auch dieselbe Tabelle herauskommt, d.h. mit denselben Indizes. Da sich eine Aktion, die in s*trref_indirector* steht, prinzipbedingt nicht mehr mit einem anderen Index versehen lässt (dies würde ja alle *strref*-Objekte potentiell ungültig machen), müssen wir mit *ordered_actions* zunächst eine sortierte Menge von Aktionsbeschriftungen erzeugen und aus dieser dann nach Abschluss des Einlesevorgangs einen *strref_indirector*.

Wie wir sehen, bietet das Interface von *AutomatonInputFormat* dazu aber keinen geeigneten Einsprungpunkt. Eine einfache Lösung wäre es, eine zusätzliche Methode einzuführen, die der Benutzer der Klasse eben nach dem Einlesen und vor dem ersten Aufruf von *GetActmap()* aufzurufen hat. Derartige Methoden führt man aber nur sehr ungern ein, da sie die *code robustness* merklich beeinträchtigen; jeder Programmierer muss nämlich diese Konvention ständig im Kopf haben, und aufrufender Code, in dem die fragliche Methode versehentlich fehlt, sieht auf den ersten Blick trotzdem korrekt aus. (Es handelt sich hier

wiederum um einen Fall erhöhter *semantischer Entfernung*, s. auch 9.3.2.1, und nicht zuletzt auch um ungenügende Kapselung.)

Stattdessen verwenden wir einen Trick: Wir nutzen aus, dass Aufrufe von *GetActmap()* selten sein werden und schreiben deshalb vor die Rückgabe von *actmap* den Code, der *actmap* aus *ordered_actions* aufbaut (und ggf. den alten Inhalt von *actmap* vorher löscht). Müsste man damit rechnen, dass *GetActmap()* sehr häufig aufgerufen wird, ließe sich das Prinzip natürlich trotzdem retten. Man würde lediglich zusätzlich ein Flag verwalten, das anzeigt, ob sich *ordered_actions* seit dem letzten Aufruf von *GetActmap()* geändert hat.

Jedenfalls kann das Ergebnis von *GetActmap()* direkt im Anschluss von *AutomatonPafasInputFormat* verwendet werden, um die durch Bitvektoren dargestellten Verweigerungsmengen in den Schritten vom Typ *UntaggedRTStep* korrekt zu initialisieren.

Die Methode *FromString()* dient lediglich dem Einlesen aus einem String statt aus einer Datei. Sie wird aus Kompatibilitätsgründen benötig, da nicht alle Versionen der C++-Standardbibliothek das Rücksetzen der Position in einem Eingabe-Stream beherrschen. (Und in der Tat ist dies für bestimmte Streams wie z.B. Konsoleneingabe auch überhaupt nicht möglich.) Um es nun *PafasInductiveSystem::StreamIn()* trotzdem zu erlauben, auf dem als Parameter angegebenen Stream zwei Durchläufe auszuführen, wird dieser zunächst in einem String gepuffert und dann an *FromString()* übergeben – welches daraus wiederum zwei Streams zur Bearbeitung durch *PafasIndirectorInputFormat* bzw. *AutomatonPafasInputFormat* erstellt und das Einlesen durchführt. So gesehen könnte *FromString()* auch *protected* bleiben, gerade in Zusammenhang mit Unit Testing erweist es sich aber als vorteilhaft, auch direkt darauf zugreifen zu können.

In folgendem Diagramm wollen wir das Aufrufschema noch einmal in der Übersicht darstellen. Die Lebenszyklen der Objekte werden dabei allerdings ignoriert, um die Darstellung übersichtlich zu halten. Insbesondere werden in Wirklichkeit natürlich die *xxxInputFormat*-Objekte und damit auch das existenzabhängige *actmap* dynamisch erstellt.

Man sieht aber schön, wie in *AutomatonPafasInputFormat* die Aufrufe von *ReadState()* zum Einfügen von Zuständen über *LookupStateWithInsert()* führen bzw. die von *ReadStep()* zu *InsertStep()*, während dies in *PafasIndirectorInputFormat* nicht der Fall ist. Dort wird stattdessen über *ordered_actions->insert()* die Tabelle der Aktionsbeschriftungen aktualisiert.

Dieses *insert()* und das spätere *actmap->map()* sind deshalb als iteriert dargestellt, weil im Gegensatz zum Einlesen eines regulären Schrittes das Einlesen einer Verweigerungsmenge im Allgemeinen mit mehreren Aktionsbeschriftungen umgehen muss.

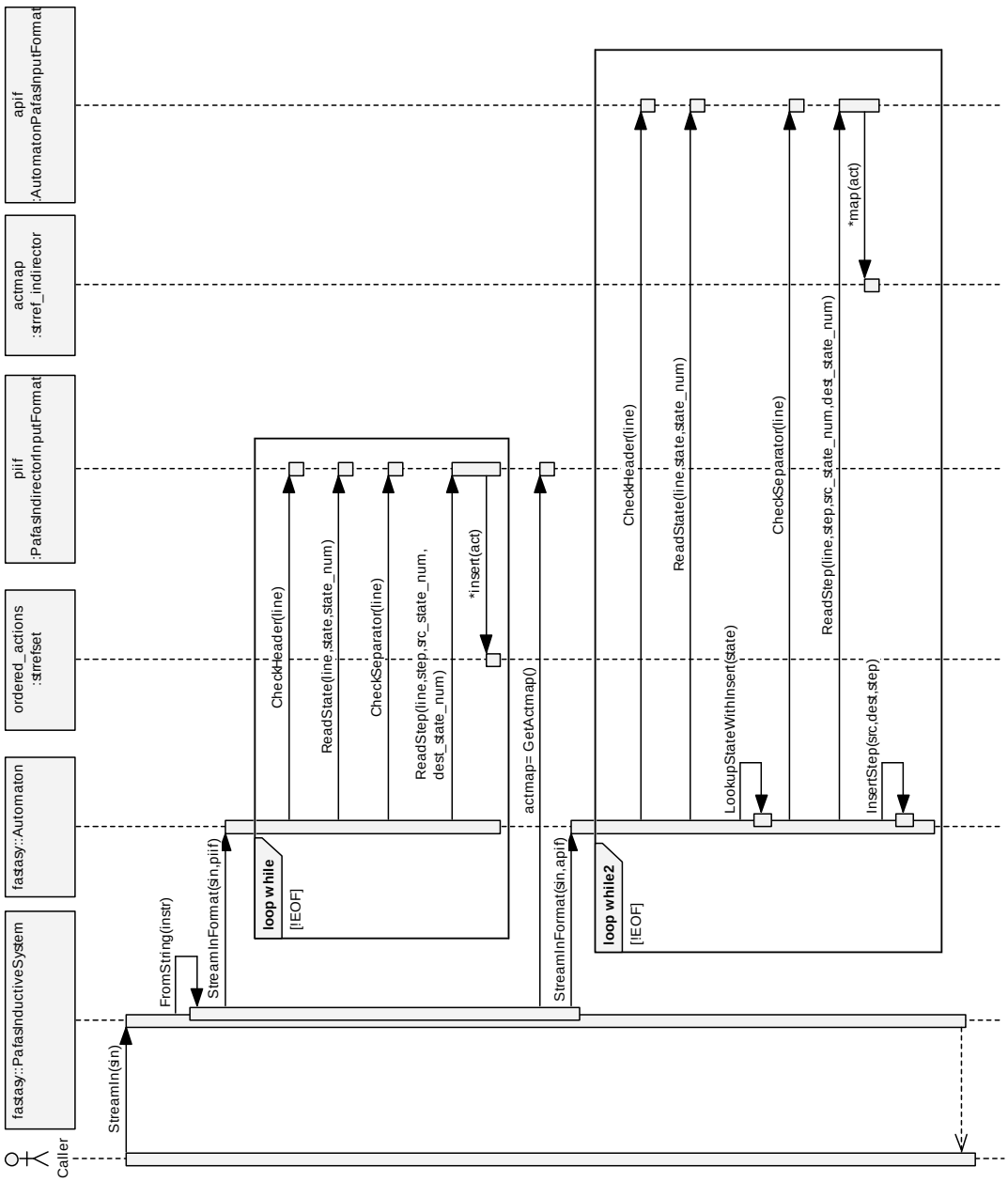



**Abbildung 63: Sequenzdiagramm Pafas-StreamIn**

## 10.2.4 Das CLI-Modul

Um nun unser *PafasInductiveSystem* in FastAsy einzubinden und als Quelle für eine Verarbeitungskette wie in 9.2.2 zu nutzen, erstellen wir eine neue Datei *create_retrace_pafas.cpp* im Verzeichnis *source/trunk/cli* von FastAsy. Dabei stellt *source* die Wurzel der C++-Quelltexte dar, *trunk* bedeutet nach der Namenskonvention von Subversion (s. [SVN]), dass es sich um den Hauptpfad der Versionsverwaltung handelt (im Gegensatz zu den *branches* und den *tags*).

Das File enthält im Wesentlichen eine freie Funktion mit folgender Signatur:

```
void create_retrace_pafas(string pafasname1, string pafasname2, ostream
&sout, bool backdir, bool summary  );
```

Dies entspricht den Parametern, die wir vom CLI erwarten. Dabei stellen $pafasname_i$ die Dateinamen der zu untersuchenden PAFAS-Systeme dar, *sout* ist der Stream, in den die Ausgabe zu schreiben ist, *backdir* gibt an, ob in beiden Richtungen simuliert werden soll und wenn das Flag *summary* gesetzt ist, erwartet der Anwender im Anschluß an die Simulation eine kurze Interpretation des Ergebnisses in den Begriffen der schneller-als-Relation.

Wir öffnen nun mit dem gegebenen Filenamen einen Stream und lesen die Eingaben ein. Dabei verwenden wir die globale Variable *DLOGLEVEL*, um festzustellen, ob Ausgaben zu Debugging-Zwecken gewünscht werden. Falls dem so ist, geben wir das eingelesene System sofort wieder aus.

```
// --- I/O ---
    ifstream sin1( pafasname1.c_str() );
    if( !sin1 ) throw fastasy_cli_error(
        string("Could not open file ") + pafasname1 + string(".")
    );
    PafasInductiveSystem IS1;
    IS1.StreamIn( sin1 );
    if( DLOGLEVEL ) IS1.StreamOut( sout );
```

Selbstverständlich führen wir dieselbe Prozedur für das andere System ebenfalls durch. Anschließend verschalten wir die Bausteine und führen die Materialisierung des nicht-deterministischen, zustandsindizierten und buchstabierenden Automaten durch:

```
    typedef ExternalInductiveSystem<TStringState, UntaggedRTStep> PTTEIS;
    typedef SystemTransformator<TStringState, UntaggedRTStep, TIntState,
UntaggedRTStep, StateIndexer<TStringState> > PTTSIS;
    typedef ReversibleAutomaton<TIntState, UntaggedRTStep> SISAutomaton;

    // eliminate lambda
    PTTEIS EIS1( IS1 );

    // index states
    PTTSIS SIS1( EIS1 );

    // Always materialize automaton, since we have to reverse it
    SISAutomaton SISA1;
    SISA1.Build( SIS1 );
    int used_maxstates1 = SISA1.GetStateCount();
    if( DLOGLEVEL ) SISA1.StreamOut( sout );
```

Man beachte, dass erst durch das *Build()* die echte Verarbeitung angestoßen wird. Wie in 9.4.2.5 beschrieben, werden wir die Anzahl der Zustände aus *SISA1* bei der Potenzautomatenkonstruktion als Länge für unsere Zustandsvektoren benötigen, so dass wir sie uns an dieser Stelle gleich verschaffen. Bei gesetztem *DLOGLEVEL* geben wir auch hier wieder ein Zwischenergebnis aus. Aus dem *typedef* für *SISAutomaton* erkennen wir noch,

dass *SISA1* ein *ReversibleAutomaton* ist. Beim Retracing wird dies wichtig werden. Nun schalten wir noch eine weitere Transformation, nämlich die Erstellung des Potenzautomaten nach und führen auch dafür ein *Build()* durch:

```
    typedef DeterministicInductiveSystem<UntaggedRTStep,
ImplicitStepCombiner<UntaggedRTStep> > PTTDIS_implicit;
    typedef Automaton<TIntSetState, UntaggedRTStep> DISAutomaton;
    // make deterministic
    DISAutomaton A1;
    PTTDIS_implicit DIS_im1( SISA1, used_maxstates1 );

    // and build
    A1.Build( DIS_im1 );
```

Am Einsatz von *ImplicitStepCombiner* sehen wir, dass wir hier mit impliziten Abschlüssen der Verweigerungsmengen arbeiten. Dies verwundert auch nicht weiter, da das PAFAS-Tool gar keine Möglichkeit enthält, die Abschlüsse explizit zu machen.

Nachdem wir eine analoge Verarbeitung für das zweite Eingabefile durchgeführt und mit *A2* den entsprechenden deterministischen Automaten haben, führen wir eine Simulation zunächst in der einen Richtung durch:

```
    typedef GenericSimulation<TIntSetState, TIntSetState, UntaggedRTStep,
UntaggedRTStep, SimpleSimulator> SimpleSimulation;
    typedef GenericSimulation<TIntSetState, TIntSetState, UntaggedRTStep,
UntaggedRTStep, ImplicitSimulator> ImplicitSimulation;

    bool result12;
    ImplicitSimulation Sim12( A1, A2 );
    if( result12 = Sim12.Build() ) {
        clog << "Simulation successful." << endl;
        // ggf.: clog << Sim12 << endl;
    } else {
        clog << "Simulation failed." << endl;
        clog << "Now building error path." << endl;
        …
```

Enthält *result12* nun den Wert *true*, so war die Simulation erfolgreich und wir wissen, dass alle Abläufe von *A1* auch in *A2* vorkommen. Wäre man an der Simulationsrelation interessiert (was nur selten der Fall sein dürfte), so könnte man diese hier ausgeben lassen. Schlägt die Simulation aber fehl, so startet man das Retracing.

Dabei bemerke man, dass man als Startzustand des revertierten Fehlerpfads *Err12* einen Multizustand *starts* vom Typ *TIntSetState* erhält, zum Start des Retracing ja aber einen einzelnen durch *TIntState* gegebenen Zustand benötigt. Da wir diesen frei aus *starts* wählen können, nehmen wir den erstbesten.

```
    typedef SimRevErrorSequence<ImplicitSimulation>
ImplicitErrorSequence;

        ImplicitErrorSequence Err12( Sim12 );
        TIntSetState starts = Err12.PullInitialState();
        TIntState start = -1;
        for(int i=0; i<starts.size(); ++i) {
            if( starts[i] ) start = i; // must have one
        }
        clog << "Picking state " << start << " as start for retracing."
<< endl;
```

Mit diesem einzelnen Startzustand wird nun die Umkehrung von *SISA1* initialisiert. Wie in 9.4.2.7 beschrieben, handelt es sich dabei nicht um eine zweite Materialisierung, sondern lediglich um einen Adapter, der uns eine weitere Sicht (nämlich die Umkehrung) auf das ursprüngliche Objekt erlaubt. Das entstandene Objekt *revSISA1* wird nun wiederum zur Initialisierung eines *NondetRetracer* verwendet, welcher den umgekehrten Fehlerpfad in den nicht-deterministischen, buchstabierenden, zustandsindizierten Automaten hievt. Der Wert *false* für den dritten Parameter zeigt dabei an, dass mit impliziten Abschlüssen von Verweigerungsmengen gearbeitet wird.

```
    typedef RevAutomatonSystem<SISAutomaton> SISRevSystem;
    typedef NondetRetracer<TIntSetState, UntaggedRTStep, TIntState,
UntaggedRTStep> NDRetracer;

        SISRevSystem revSISA1( SISA1, start );
        NDRetracer retro( Err12, revSISA1, false /* expect implicit
subsets */ );
```

Wie weiter oben bei den induktiven Systemen ist bis zu dieser Stelle noch keine Verarbeitung erfolgt, es wurden lediglich die Bausteine verschaltet. Nun werden wir allerdings mit Hilfe zweier *Trace*-Instanzen der Ergebnis materialisieren und mit Hilfe des Adapters *RevSequence* umkehren, bevor wir mit der Verarbeitung fortfahren. Man beachte wie gesagt, dass Materialisierungen von linearen Systemen weder aus der Sicht des Speicherbedarfs noch der Rechenzeit ins Gewicht fallen.

```
    typedef IndexRetracer<TIntState, UntaggedRTStep, TStringState,
UntaggedRTStep> StateRetracer;
    typedef Trace<TIntState, UntaggedRTStep> RTIntTrace;
    typedef RevSequence<TIntState, UntaggedRTStep> RTRevSequence;

        RTIntTrace trace_back;
        trace_back.Build( retro );
        RTIntTrace trace_fwd;
        trace_fwd.Build( RTRevSequence( trace_back ) );
```

```
        StreamOut( trace_fwd, clog );
```

Als nächstes verschaffen wir uns mit einem *StateRetracer* die ursprünglichen Zustandsinformationen. Man bemerke, wie durch Übergabe des Funktionals, welches mit *SIS1.GetStateTransform()* ermittelt wird, die bei der Indizierung verwendete Tabelle hier in der Gegenrichtung verwendet wird (als *Seitenergebnis*, s. 9.4.1.3). Wiederum materialisieren wir das Ergebnis, hier mit dem einzigen Zweck, eine Zwischenausgabe zu ermöglichen.

```
    typedef Trace<TStringState, UntaggedRTStep> RTTrace;
        StateRetracer internal( trace_fwd, EIS1, SIS1.GetStateTransform()
);
        RTTrace trace_rts;
        trace_rts.Build( internal );
```

Ebenso gehen wir vor, um die internen Übergänge wiederherzustellen:

```
    typedef InternalRetracer<TStringState, UntaggedRTStep, TStringState,
UntaggedRTStep> IntRetracer;
        IntRetracer external( trace_rts, IS1 );
        RTTrace trace_pafas;
        trace_pafas.Build( external );
```

Schließlich geben wir das Endergebnis in den Ausgabestream *sout*, welchen wir als Parameter erhalten haben, aus.

```
        StreamOut( trace_pafas, sout );
```

Entsprechend wird, falls *backdir* gesetzt ist, der Vorgang noch einmal in der anderen Richtung durchgeführt, und bei gesetztem *summary* wie gesagt das Ergebnis noch interpretiert.

Wir sehen, wie wir mit einfachsten Mitteln unsere neue Klasse *PafasInductiveSystem* als neues Quellsystem integriert haben. Was zu tun bleibt ist, wie in 9.7.3 beschrieben in die Datei *cli_main.cpp* die geeigneten Zeilen einzufügen, um dem Kommandozeileninterpreter bekannt zu machen:

```
    …
    clikeyword k_retrace_pafas("PAFAS");
    cliparam p_pafasfile("pafasfile");
    cliparam p_pafasfile2("pafasfile2");
    …
    arg_versions[av_retrace_pafas] << k_retrace_pafas << s_outfile
        << s_backdir << s_summary << s_verbose << s_stopwatch
        << p_pafasfile << p_pafasfile2;
    help_versions[av_retrace_pafas] = "Attempt simulation between two
PAFAS systems from files, then retrace witness.";
    …
```

Weiter unten fügen wir dann einen entsprechenden *case*-Zweig mit dem eigentlichen Aufruf ein:

```
    case( av_retrace_pafas ): {
        string pafasfile1 = p_pafasfile.argv().c_str();
        string pafasfile2 = p_pafasfile2.argv().c_str();
        int maxstates = s_maxstates.given()==false ? -1 :
            lexical_cast<int>(s_maxstates.argv());
        if( s_outfile.given() ) {
            ofstream sout( s_outfile.argv().c_str() );
            if( !sout ) {
                cerr << "Could not open " << s_outfile.argv() << "." <<
endl;
                return -1;
            }
            create_retrace_pafas( pafasfile1, pafasfile2, sout,
                s_backdir.given(), s_summary.given() );
        } else {
            create_retrace_pafas( pafasfile1, pafasfile2, cout,
                s_backdir.given(), s_summary.given() );
        }
    } break;
```

Nach der Neuübersetzung des Quelltextes ist die Funktionalität nun Bestandteil von FastAsy.

# Kapitel 11 Nachbetrachtungen

Naturgemäß bleiben bei einem offen ausgerichteten Werkzeug immer Ideen, von denen eine Umsetzung wünschenswert erscheint, die aber in die aktuelle Version nicht mehr aufgenommen werden konnten. Immerhin wurde die Codebase von FastAsy vom Autor ohne Einsatz von Hilfskräften geschaffen. Trotzdem sollen einige dieser Ideen hier skizziert werden, um später vielleicht im Rahmen einer Vergabe als Diplomarbeit o.ä. verwirklicht zu werden.

## *11.1 Minimierung des Erreichbarkeitsgraphen*

Man wird feststellen, dass FastAsy auf den Einsatz bekannter Verfahren zur Minimierung von Automaten wie etwa das in [Hop71] beschriebene momentan verzichtet. Dies liegt unter anderem in der Beobachtung begründet, dass der direkt aus dem Quellsystem enstehende nicht-deterministische endliche Automat bereits das größte auftretende Zwischenergebnis darstellt. Wie aus [JR93] bekannt, ist die Minimierung nicht-deterministischer Automaten NP-vollständig. In Frage kommt also nur eine Minimierung *nach* dem in 7.2.4 beschriebenen Übergang zum deterministischen Potenzautomaten. Wie dort beschrieben, ist der Potenzautomat empirisch beobachtbar aber bereits von sich aus wesentlich kleiner, so dass eine Minimierung desselben an der falschen Stelle anzusetzen scheint und wir uns stattdessen auf die in Kapitel 2 beschriebene problemspezifische Kompression konzentriert haben.

Durch die Architektur der Bausteine-Schicht von FastAsy wäre es aber relativ einfach, auch einen Minimierungs-Baustein zu schreiben. (Interessenten sei geraten, die strukturiertere Beschreibung des Algorithmus in [Gri72] statt der ursprünglichen Beschreibung von Hopcroft zu verwenden.) Ein solcher Baustein könnte als Grundlage für Performance-Experimente verschiedener Art dienen, wenn er jeweils an unterschiedlichen Stellen verschaltet und die Größen der Zwischenergebnisse bzw. die Laufzeiten miteinander verglichen würden.

Ob ein solcher Baustein eventuell sogar ohne lokale Materialisierung des Zwischenergebnisses (d.h. innerhalb der Baustein-Instanz) auskommen könnte und wie sich ein solcher Minimierungsschritt in das Retracing (siehe 7.4) einpasst, ist zu untersuchen.

Möglicherweise sehr anspruchsvoll, aber überaus wünschenswert, wäre es, die Vorschrift zur Minimierung der Zustandsmenge so zu modifizieren, dass sie auch auf Erreichbarkeitsgraphen mit den in Kapitel 2 beschriebenen implizit bleibenden Verweigerungsmengen operiert.

## *11.2 Bestimmung starker λ-Zusammenhangskomponenten*

Für Systeme, die viel internes Verhalten besitzen, kommt noch ein anderer Minimierungsansatz in Frage. (Solche Systeme sind durchaus relevant, weil man oft nur das Timing von Ein- und Ausgaben vergleichen will. In [Vog96], [BV98] etwa werden sogar alle Aktionen bis auf die eines einzigen Benutzers für unsichtbar erklärt.)

In solchen Fällen kann man nun die starken λ-Zusammenhangskomponenten des Erreichbarkeitsgraphen bestimmen, also solche, deren Zustände sich untereinander alle über interne Aktionen erreichen lassen. Diese Komponenten kann man nun zu einem einzigen

Zustand zusammenziehen; das entstehende Transitionssystem ist sprachgleich zum ursprünglichen.

Die Bestimmung dieser Komponenten entspricht der Berechnung des Strukturgraphen (siehe etwa [VT96]) nach vorheriger Projektion des Graphen auf seine λ-Kanten. Die Komplexität des Verfahrens beträgt O(n+m), wobei wie üblich n die Zahl der Zustände und m die Zahl der Kanten ist. Selbstverständlich kann die Projektion auf die λ-Kanten bei entsprechender Formulierung des Verfahrens implizit bleiben. Dadurch, dass FastAsy interne Übergänge zusammenhängend verwaltet, bleibt m dadurch immer noch auf die Anzahl der λ-Kanten beschränkt.

Auch hier wäre zu untersuchen, wie eine effiziente Retracing-Strategie aussieht (d.h. eine solche, die einen möglichst kurzen Pfad im ursprünglichen Transitionssystem liefert, aber nicht im Aufwand einer erschöpfenden Suche resultiert). Dass zumindest ein „lokal effizientes“ Retracing möglich ist, ist aber offensichtlich: Man sucht die Zustände innerhalb der jeweiligen aktuellen Zusammenhangskomponente, die eine Kante mit der gewünschten Beschriftung zu einem Zustand in der nächsten Komponente enthalten. Dann läuft man über interne Übergänge denjenigen dieser Zustände an, der vom aktuellen Zustand aus gesehen am nächsten liegt. Anschließend läuft man in die Nachfolgerkomponente, die zur neuen aktuellen Komponente wird, und der erreichte Zustand wird zum aktuellen Zustand. Man beachte, dass diese Strategie gleichermaßen für Vorwärts- und Rückwärtsabläufe geeignet ist.

## *11.3 Verzicht auf Transitionsannotationen*

In 9.4.2.3 wird als Schritttyp *TaggedRTStep* vorgestellt, dessen Besonderheit es ist, dass er neben der Aktionsbeschriftung noch als Annotation (die aber zu Vergleichen nicht herangezogen wird) die Identität der zugrunde liegenden Transition enthält. Dies ist eine effiziente Möglichkeit, diese für den Benutzer wichtige Information zu speichern, allerdings um den Preis einer zusätzlichen Komplexität, die man in das System einführt.

Eine Alternative dazu wäre es, *PTTInductiveSystem* (siehe 9.3.2.6) einen mit Transitionen beschrifteten Erreichbarkeitsgraphen generieren zu lassen und anschließend innerhalb des *Transformation*-Package einen neuen Baustein *LabelledInductiveSystem* einzuführen, der mittels der Funktion $l:T\rightarrow\Sigma\cup\{\lambda\}$ des Netzes die Transitionsnamen durch Aktionen austauscht.

Das Retracing wäre in diesem speziellen Fall ebenfalls kein Problem, da wir mit *RTState* als Zustandstyp ja ausreichend Information besitzen, um die jeweils schaltende Transition festzustellen.

So reizvoll diese Variante auf den ersten Blick sein mag, führt sie doch an einer aus Performancesicht recht kritischen Stelle, der Erstellung des Erreichbarkeitsgraphen, zu einer Erhöhung des Aufwands gegenüber der momentan implementierten Lösung.

Besteht aber einmal Interesse an einem systematischen Ausbau des Baustein-Zoos in FastAsy, so sollten diese Überlegungen Beachtung finden (evtl. parallel zu der bestehenden Lösung, indem man das jetzige *PTTInductiveSystem* zu *TaggedPTTInductiveSystem* umbenennt.).

## *11.4 Rückwärtsabläufe*

Prinzipiell stellt die Notwendigkeit, ein Zwischenergebnis materialisieren zu müssen, um an die Rückwärtsabläufe eines Transitionssystems zu gelangen, einen unschönen Bruch des Baustein-Konzepts dar, bei dem wir ja eigentlich frei entscheiden wollen, wann Ergebnisse explizit im Speicher gehalten werden.

Für die momentane Anwendung von Rückwärtsabläufen in FastAsy, nämlich das Retracing der Schritte des nicht-deterministischen Automaten aus dem Potenzautomat, ist ja in 7.4.1 ein Trick beschrieben, mit dem man auch ohne die Rückwärtsabläufe auskommen kann.

Allgemeiner kann man sich aber durchaus die Frage stellen, inwiefern man die induktiven Systeme mit der Fähigkeit versehen könnte, auch ohne Materialisierung Rückwärtsabläufe zu liefern.

Dabei stellt man zunächst fest, dass bestimmte Typen von Systemen dies nicht oder nicht effizient leisten können. Man betrachte dazu z.B. nur Petri-Netze: Zwar können wir den umgekehrten Effekt einer gegebenen Transition auf die Markierung leicht feststellen, aber schon die möglichen Alterswerte aller Marken im Vorbereich vor dem Schalten lässt die Anzahl der möglichen Vorgänger drastisch in die Höhe schnellen. Was noch viel schwerer wiegt ist aber, dass wir unter diesen möglichen Vorgängern nur diejenigen auswählen dürfen, von denen aus sich rückwärtslaufend die Anfangsmarkierung erreichen lässt.

Umkehrbarkeit ohne Materialisierung wäre also in jedem Fall ein optionales Merkmal von Klassen, die induktive Systeme modellieren. Weiter hängt ja die Fähigkeit, die eigenen Rückwärtsabläufe zu ermitteln, auch von der entsprechenden Fähigkeit des jeweils vorgeschalteten Bausteins ab.

Bei einer Umsetzung mittels Objektpolymorphie müsste die optionale Methode für die Rückwärtsabläufe in einem abgeleiteten Interface untergebracht werden. Sehr unkomfortabel wäre hier die Propagierung der Fähigkeit zur Umkehrung: Über RTTI (z.B. geprüften *downcast*) müsste ein Objekt prüfen, ob sein Vorgänger die Fähigkeit hat. Dies funktioniert jedoch nur zur Laufzeit. Eine statische Prüfung hingegen würde umkehrbare und nicht umkehrbare Versionen der Klassen jeweils mit oder ohne umkehrbarem Vorgänger bedingen, was zu paralellen Auschnitten in der Klassenhierarchie führt – ein deutlicher Hinweis auf einen Entwurfsfehler.

Ein Behelf an dieser Stelle, der streng genommen in einer Compilersprache indiskutabel ist, aber trotzdem eine gewisse Verbreitung erfahren hat, ist der, die optionale Methode im ursprünglichen Interface zu verankern und in Klassen, die sie nicht implementieren können, mit einem Rumpf auszustatten, der lediglich eine Ausnahme wirft. Auch hier finden wir den Fehler also erst zur Laufzeit, allerdings erfolgt wenigstens die Propagierung des Fehlers transparent.

Es existiert aber in C++ auch ein Königsweg: Durch die Art, wie der Compiler die Abhängigkeiten von Code in Rümpfen von Methodentemplates prüft (nämlich erst bei Instanzierung der *Methode*), kann die optionale Methode einfach so geschrieben werden, dass sie sich ihrerseits auf das Vorhandensein der optionalen Methode im Vorgänger verlässt.

Um dies kurz an einem abstrakten Beispiel zu illustrieren:

```
struct X
{
    void f() {}
};

struct Y
{
    void g() {}
};

struct A
{
    template<typename T>
    void use_f(T b) {
         b.f();
    }
    template<typename T>
    void dont_use_f(T b) {
    }
};

A a;
X x;
Y y;

a.use_f(x);       // okay
a.dont_use_f(x);  // okay
a.use_f(y);       // Compilerfehler: "f ist kein Element von Y"
a.dont_use_f(y);  // okay
```

So wie in diesem Beispiel die Methode *f* nur bei Instanzierung der Methode *use_f* vorausgesetzt wird, so könnte sich eine Methode *GetReverseFollowers* auf die entsprechende Methode seines Vorgängers abstützen. Die Korrektheit der Verschaltung würde dann auch in dieser Hinsicht statisch vom Compiler geprüft.

## *11.5 Kürzester Fehlerpfad*

In 7.3 haben wir kurz bemerkt, dass *FastAsy* zumindest bei Verwendung von BFS in der Simulation kurze Fehlerpfade im Potenzautomaten findet. Wir haben jedoch keinerlei Garantie, dass dahinter auch tatsächlich ein kurzer Fehlerpfad im ursprünglichen System steht, denn ein kurzer sichtbarer Pfad kann zunächst beliebig viele interne Transitionen enthalten. Selbst wenn wir die Lücken zwischen den sichtbaren Aktionen mit möglichst

kurzen (insbesondere natürlich kreisfreien) internen Pfaden füllen, haben wir damit insgesamt bestenfalls eine Heuristik für kurze Pfade.

Das grundsätzliche Problem besteht einfach darin, schon bei der Simulation einen Pfad zu finden, der *nach Ergänzung der internen Übergänge* möglichst kurz ist.

Zu diesem Zweck könnte man sich bei der Elimination der λ-Kanten in jedem entstehenden sichtbaren Schritt merken, wie viele λ-Kanten er enthält. Bei der Simulation könnte man dann BFS durch Einsatz einer Vorrangwarteschlange derart modifizieren, dass immer zunächst diejenigen Pfade mit den billigsten Kosten (also den wenigsten Kanten inklusive der früheren λ-Kanten) zuerst untersucht werden.

Allerdings ist schwer zu sehen, wie man diese Information über die Potenzautomaten-konstruktion hinweg retten könnte: Dort würden teure und billige Kanten ja unkontrollierbar zusammengefasst.

Glücklicherweise ist das aber gar nicht unbedingt nötig, denn wir wissen ja aus Teil I, 1.3.1.6, dass derjenige Automat, in dem der Fehlerpfad liegt, zur Durchführung der Simulation nicht deterministisch sein muss.

Bedenkt man allerdings, dass wie schön häufig bemerkt der deterministische Automat in unserem Szenario wesentlich kleiner ist, so ist fraglich, ob man wirklich einen drastisch angestiegenen Aufwand zu Gunsten eines global kürzesten Fehlerpfads investieren möchte. In Zusammenhang mit den Überlegungen aus 11.1 könnte dies dennoch interessant sein.

## *11.6 GUI*

Ein rein kosmetisches Projekt zur Erweiterung von *FastAsy* wäre es, das CLI durch eine GUI zu ergänzen. Durch die Verwendung der Fachfassaden wäre ein derartiger Eingriff auch möglich, ohne weit in die Klassenhierarchie vorzudringen – genau diesem Zweck der Abschirmung dienen die Fachfassaden ja.

Abhängig vom verwendeten GUI-Framework (VCL, QT, wxWidgets,…) wäre lediglich das Problem zu lösen, die Ein-/Ausgabe von Strings über Streams an das Framework anzuschließen, da nicht alle mit dem standardmäßigen *std::string* der STL zurechtkommen. Einige einfache Wrapper-Klassen könnten dies übernehmen, ohne Anpassungen innerhalb der Fassaden nötig zu machen.

Eine aufwändigere Lösung könnte natürlich auch selbst neue Fachfassaden gestalten, die dann etwa spezielle grafische Kontrollelemente mit der jeweiligen Ausgabe füllen etc.

Prinzipiell hält der Autor allerdings das (schlanke und einfach skriptbare) CLI gegenüber einer GUI für überlegen.

Alternativ zu einer Visualisierung von Ergebnissen in einer GUI ließe sich übrigens Graphviz (siehe [GVZ]) hervorragend nutzen, um Ausgaben zu veranschaulichen. Denkbar sind etwa die Darstellung des Fehlerpfads als Graph, wobei Zustände geclustert werden, die nur im Alter der Marken abweichen oder die Generierung eines klickbaren Fehlerpfads, von dem ein Klick auf einen Zustand jeweils den markierten Netzgraph anzeigt.

## *11.7 BGL-Algorithmen in FastAsy*

Während die BGL (siehe 6.2.2) in FastAsy bei der Speicherung von Graphen hervorragende Dienste leistet, enthält sie ja darüber hinaus noch eine ständig wachsende Zahl

äußerst effizient implementierter Algorithmen, die sich Erweiterungen von *FastAsy* vielleicht zu Nutze machen möchten. Dies kann prinzipiell über zwei Ansätze erfolgen:

Zum einen können natürlich innerhalb von Klassen wie *Automaton*, die wirklich unter Benutzung von *adjacency_list* oder anderer Graphenklassen der BGL den Erreichbarkeitsgraphen speichern, alle Algorithmen der BGL benutzt werden. („FastAsy benutzt BGL.“) Preis dafür ist eben eine vollständige und sofortige Materialisierung an dieser Stelle.

Zum anderen kann die BGL aber mit sogenannten *ImplicitGraphs* umgehen, die lediglich über ihre Adjazenzfunktion definiert, also wie viele Zwischnergebnisse in *FastAsy* nicht explizit gespeichert sind. Ein Wrapper um *InductiveSystem* müsste dazu nach außen das Interface eines *AdjacencyGraph* zur Verfügung stellen, so dass induktive Systeme in vorhandene BGL-Konstrukte eingebettet werden könnten. („BGL benutzt FastAsy.“)

Wer sich mit letzterer Möglichkeit beschäftigen möchte, dem sei dringend ein Blick in das entsprechende Kapitel von [SLL02] ans Herz gelegt.

## *11.8 Vergleichssemantik von dynamic_bitset<>*

Wie in 8.1.1.3 bzw. 9.4.2.5 geschildert, bedingt die Verwendung des Templates *dynamic_bitset*<> aus der *Boost*-Bibliothek zur Speicherung von Multizuständen des Potenzautomaten eine etwas unnatürliche Randbedingung: Die Anzahl der Zustände des buchstabierenden nicht-deterministischen Automaten, der etwa als Produkt von *ExternalInductiveSystem* entsteht, muss vorab bekannt sein.

*dynamic_bitset*<> nämlich ahmt in seinem Verhalten größtenteils das Template *bitset*<> aus der STL nach, und für dieses sind Bitvektoren unterschiedlicher Länge verschiedene Typen und somit unvergleichbar. Bei *dynamic_bitset*<> entfällt lediglich die Notwendigkeit, die Länge *statisch* festzulegen.

Eine nahe liegende Lösung wäre es, mit einer Klasse *very_dynamic_bitset*<> wiederum das Verhalten von *dynamic_bitset*<> nachzuahmen und dabei aber die Semantik von Vergleichen dergestalt zu ändern, dass der kürzere Bitvektor grundsätzlich als mit Nullen aufgefüllt betrachtet wird.

So könnten bei der Erstellung des Potenzautomaten zur Speicherung von Multizuständen verschieden lange, insbesondere auch kürzestmögliche, Bitvektoren verwendet werden. Dies hätte überdies den Vorteil, dass der benötigte Speicherplatz etwas verringert würde.

# Schlusswort

Im Rahmen dieser Arbeit haben wir ein theoretisches Modell für Petri-Netze mit Zeit geschaffen, welches mit den Lesekanten und Intervalbeschriftungen der Kanten die Merkmale früherer Modelle subsumiert. Mit den *lazy-arcs* wurde das Modell darüber hinaus noch mit einer nützlichen Erweiterung versehen.

Frühere Resultate wurden erfolgreich auf unser neues Modell übertragen: Die Korrektheit der Diskretisierung, die Herleitung einer endlichen Charakterisierung der schneller-als-Relation. Darüberhinaus wurde der Fragenkomplex rund um zeitliche Anomalien (*~timestops*) hier noch einmal von Grund auf neu angegangen, wobei sich herausgestellt hat, dass gerade die Einführung von Lesekanten die Behandlung dieser Fragen gegenüber dem Modell in [Bih98] maßgeblich erschwert.

Mit den Betrachtungen zur Mächtigkeit von Lesekanten und Kantenintervallen haben wir unser neues Netzmodell genutzt, um eine seit längerem offene Fragestellung umfassend zu beantworten. Das Ergebnis entsprach dabei weitgehend den Erwartungen, zeigte dabei aber im Detail durch die Berücksichtigung verschiedener Auffassungen von Verhaltensäquivalenz große Vielschichtigkeit.

Durch die technisch anspruchsvollen Untersuchungen zur Ausnutzung impliziter Verweigerungsmengen haben wir nicht nur die Grundlagen für eine konzeptionelle Optimierung der automatisierten Behandlung von auf Verweigerungsmengen basierender Semantiken gelegt, wir haben diese Optimierungen dann auch im zweiten Teil der Arbeit praxistauglich umgesetzt. Wie in den Benchmarks in Anhang C zu sehen ist, konnte die Verarbeitungsgeschwindigkeit von FastAsy bei Eingaben mit vielen sichtbaren Aktionen um Größenordnungen gesteigert werden.

Mit der neuen Codebase für *FastAsy*, die als Teil der vorliegenden Arbeit entstanden ist, hat dieses Tool einen großen Schritt in Richtung Reife getan. Mit der Umorientierung weg vom konkreten einzelnen Problem hin zu der Experimentierkasten-Metapher, die uns durchgehend begleitet hat, haben wir die Grundlagen für vielfältige Weiterentwicklungen gelegt. Mit der Entwicklung der Anbindung an das externe PAFAS-Tool haben wir dieses Potential auch bereits illustriert.

Zu guter Letzt bleibt die Hoffnung, dass die „*FastAsy*-Story“ hier nicht endet. Mit den ausführlichen Schilderungen rund um Design und Umsetzung von FastAsy glauben wir nachfolgenden Entwicklern den Weg geebnet zu haben für viele weitere Bausteine des Experimentierkastens, die dann letzten Endes in ihrer Summe erst das Potential der Architektur richtig ausschöpfen können werden.

# Teil III Anhang

# A Index

# B Abbildungen

# C Benchmarks

Die folgenden Daten zeigen den Einfluss verschiedener Optimierungen auf die Rechenzeit. Gemessen wurde reale Zeit auf einem leer laufenden System (Pentium 4, 2.54Ghz, 1GB Hauptspeicher), jedoch nicht reine CPU-Zeit.

Die als Beispiele verwendeten Systeme entstammen [Vog96] und [BV98], die Verallgemeinerung von **mutex1** und **mutex2** auf mehr als zwei Benutzer wurde analog zu der in [BV98] angegebenen Verallgemeinerung von **mutex3** und **mutex4** durchgeführt.

### C.1. Berechnung von Erreichbarkeitsgraphen *)

| *Aufgabe/Opt.* *Eingabefile* | **NDEA -** | **NDEA i** | ***Speed-up*** |
|---|---|---|---|
| **mutex1** | 0,891 | 0,063 | *14,1* |
| **mutex1_3users** | 328,875 | 1,500 | *219,3* |
| **mutex2** | 0,312 | 0,016 | *19,5* |
| **mutex2_3users** | 30,563 | 0,156 | *195,9* |
| **mutex3** | 0,172 | 0,078 | *2,2* |
| **mutex3_3users** | 3,110 | 2,266 | *1,4* |
| **mutex3_4users** | 47,782 | 39,578 | *1,2* |
| **mutex4** | 0,063 | 0,031 | *2,0* |
| **mutex4_3users** | 0,438 | 0,297 | *1,5* |
| **mutex4_4users** | 2,828 | 2,203 | *1,3* |
| **mutex4_5users** | 17,578 | 14,813 | *1,2* |

| *Aufgabe/Opt.* *Eingabefile* | **SNDEA -** | **SNDEA lc** | **SNDEA slc** | **SNDEA i** | **SNDEA i lc** | **SNDEA i slc** | ***Speed-up*** |
|---|---|---|---|---|---|---|---|
| **mutex1** | 2,157 | 2,532 | 2,172 | 0,125 | 0,125 | 0,125 | *17,3* |
| **mutex1_3users** | 1296,230 | 1296,140 | 1286,440 | 3,485 | 3,516 | 3,390 | *382,4* |
| **mutex2** | 0,390 | 0,391 | 0,391 | 0,046 | 0,032 | 0,047 | *8,3* |
| **mutex2_3users** | 40,157 | 40,172 | 39,938 | 0,235 | 0,234 | 0,250 | *160,6* |
| **mutex3** | 0,781 | 0,782 | 0,734 | 0,282 | 0,265 | 0,234 | *3,3* |
| **mutex3_3users** | 56,688 | 56,484 | 48,594 | 24,203 | 24,078 | 17,547 | *3,2* |
| **mutex4** | 0,156 | 0,156 | 0,157 | 0,063 | 0,063 | 0,063 | *2,5* |
| **mutex4_3users** | 2,859 | 2,672 | 2,454 | 1,281 | 1,266 | 1,109 | *2,6* |
| **mutex4_4users** | 42,328 | 42,062 | 35,641 | 23,281 | 23,859 | 18,125 | *2,3* |
| **mutex4_5users** | 650,672 | 651,156 | 497,500 | 409,422 | 408,203 | 277,844 | *2,3* |

| *Aufgabe/Opt.* *Eingabefile* | **DEA -** | **DEA lc** | **DEA slc** | **DEA i** | **DEA i lc** | **DEA i slc** | ***Speed-up*** |
|---|---|---|---|---|---|---|---|
| **mutex1** | 3,078 | 3,000 | 3,000 | 0,125 | 0,125 | 0,140 | *22,0* |
| **mutex1_3users** | 2008,220 | 2005,970 | 2004,610 | 4,125 | 4,172 | 4,093 | *490,6* |
| **mutex2** | 0,500 | 0,500 | 0,516 | 0,032 | 0,032 | 0,031 | *16,1* |
| **mutex2_3users** | 68,281 | 68,329 | 68,313 | 0,234 | 0,234 | 0,266 | *256,7* |
| **mutex3** | 0,672 | 0,656 | 0,625 | 0,234 | 0,219 | 0,188 | *3,6* |
| **mutex3_3users** | 90,469 | 91,828 | 82,406 | 28,062 | 28,218 | 21,703 | *4,2* |
| **mutex4** | 0,125 | 0,109 | 0,109 | 0,047 | 0,047 | 0,062 | *2,0* |
| **mutex4_3users** | 2,094 | 2,109 | 1,875 | 1,078 | 1,156 | 0,922 | *2,3* |
| **mutex4_4users** | 41,438 | 41,500 | 34,594 | 23,328 | 23,360 | 17,985 | *2,3* |
| **mutex4_5users** | 848,828 | 849,485 | 701,938 | 514,313 | 514,859 | 383,843 | *2,2* |

## C.2. Berechnung von Simulationen *)

| *Eingabefile 1* | *Aufgabe/Opt. Eingabefile 2* | **SIM -** | **SIM slc** | **SIM slc i** | ***Speed-up*** |
|---|---|---|---|---|---|
| **mutex1** | **mutex2** | 2,657 | 2,610 | 0,110 | *24,15* |
| **mutex1_3users** | **mutex2_3users** | >1h | | 3,390 | *>1000* |
| **mutex3** | **mutex4** | 0,734 | 0,687 | 0,234 | *3,14* |
| **mutex3_3users** | **mutex4_3users** | 93,375 | 84,359 | 24,437 | *3,82* |

| *Eingabefile 1* | *Aufgabe/Opt. Eingabefile 2* | **SIM -** | **SIMR slc** | **SIMR slc i** | ***Speed-up*** |
|---|---|---|---|---|---|
| **mutex1** | **mutex2** | 2,703 | 2,656 | 0,140 | *19,31* |
| **mutex1_3users** | **mutex2_3users** | >1h | | 3,453 | *>1000* |
| **mutex3** | **mutex4** | 0,781 | 0,719 | 0,250 | *3,12* |
| **mutex3_3users** | **mutex4_3users** | 93,157 | 84,125 | 23,062 | *4,04* |

***)** Alle Zeiten sind, falls nicht anders angegeben, in Sekunden zu verstehen. Das Kürzel **lc** bezeichnet die *CachingLambdaHull*-Optimierung, das Kürzel **slc** die *SmartCachingLambdaHull*-Optimierung (beide siehe 9.4.2.4). Das Kürzel **i** bezeichnet die Verwendung impliziter Teilmengen (siehe Kapitel 2).

# D Literaturverzeichnis


**Bücher**

[AA01] Andrei Alexandrescu. Modern C++ Design, C++ in depth series. Addison-Wesley. 2001

[BK06] Björn Karlsson. Beyond the C++ Standard Library. Addison-Wesley. 2006

[BK99] Kent Beck. Extreme Programming Explained. Addison-Wesley. 1999

[BO01] Bernd Oesterreich. Objektorientierte Softwareentwicklung, 5. überarbeitete Auflage. Addison-Wesley. 2001

[BS02] Bruno Schienmann. Kontinuierliches Anforderungsmanagement. Addison-Wesley. 2002

[BS98] Bjarne Stroustrup. Die C++ Programmiersprache, 3. aktualisierte und erweiterte Auflage. Addison-Wesley. 1998

[FB00] Frank Buschmann et al. Patterorientierte Softwarearchitektur. Addison-Wesley. 2000

[FB03] Frederick P. Brooks jun. Vom Mythos des Mann-Monats. mitp. 2003

[GB94] Grady Booch. Objektorientierte Analyse und Design. Addison-Wesley. 1994

[GoF95] Gamma, Helm, Johnson, Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995

[GoF96] Gamma, Helm, Johnson, Vlissides. Entwurfsmuster. Addison-Wesley. 1996

[HU94] J. E. Hopcroft, J. D. Ullman. Einführung in die Automatentheorie - formale Sprachen und Komplexitätstheorie, 3. Auflage. Addison-Wesley. 1994

[JL96] John Lakos. Large scale C++ software design. Addison-Wesley. 1996

[JN96] Nicolai M. Josuttis. Die C++ Standardbibliothek. Addison-Wesley. 1996

[MF00] Martin Fowler. Refactoring. Addison-Wesley. 2000

[PCF04] C. Michael Pilato, Ben Collins-Sussman, Brian W. Fitzpatrick. Version Control with Subversion. O'Reilly. 2004

[SLL02] Jeremy G. Siek et al. The Boost Graph Library. Addison-Wesley. 2002

[Sta90] Peter Starke. Analyse von Petri-Netz-Modellen. Teubner. 1990

[VJ03] David Vandevoorde, Nicolai M. Josuttis. C++ Templates: The complete guide. Addison-Wesley. 2003

[VT96] Volker Turau. Algorithmische Graphentheorie. Addison-Wesley. 1996

**Reports und Artikel**

[AFGI03] L. Aceto, W. Fokkink, R. van Glabbeek, and A. Ingolfsdottir. Nested semantics over finite trees are equationally hard. Technical Report RS--03--27. Aalborg University. 2003

[Bih00] E. Bihler. Objektorientiertes Design eines Werkzeugs zum Effizienzvergleich asynchroner Systeme. Technischer Report. Universität Augsburg. 2000

[Bih98] E. Bihler. Effizienzvergleich bei verteilten Systemen - eine Theorie und ein werkzeug. Diplomarbeit. Universität Augsburg. 1998

[Bow00] F. D. J. Bowden. A Brief Survey and Synthesis of the Roles of Time in Petri Nets. MCM 31 55-68. Elsevier. 2000

[BV04] E. Bihler, W. Vogler. Timed Petri Nets: Efficiency of Asynchronous Systems. Lect. Notes Comp. Sci. 3185:25-58. Springer. 2004
[BV98] E. Bihler, W. Vogler. Efficiency of token-passing MUTEX-solutions - some experiments. Lect. Notes Comp. Sci. 1420:185-204. Springer. 1998
[CVJ02] F. Corradini, W. Vogler, L. Jenner. Comparing the Worst-Case Efficiency of Asynchronous Systems with PAFAS. Acta Informatica 38:735-792. 2002
[DNH84] R. De Nicola, M.C.B Hennessy. Testing equivalence for processes. Theoret. Comput. Science, 34:83-133. 1984
[Dur85] R. Durchholz. Another note on side-conditions. PetriNet Newsletters 22:13-16. 1985
[Gri73] D. Gries. Describing an algorithm by Hopcroft. Acta Informatica 2:97-109. Springer. 1973
[Hop71] J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Z. Kohavi (Ed.), Theory of Machines and Computations, pp. 189-196. Academic Press. 1971
[JR93] Tao Jiang, Bala Ravikumar. Minimal NFA Problems are Hard. SIAM J. Comput. 22(6):1117-1141, 1993
[JV95] L. Jenner, W. Vogler. Fast Asynchronous Systems in Dense Time. Report Nr. 344. Universität Augsburg. 1995
[KW97] E. Kindler, R. Walter. Mutex needs fairness. Inform. Process. Lett. 92:31-39. 1997
[LF81] N. Lynch and M. Fischer. On describing the behaviour and implementation of distributed systems. Theoret. Computer Science, 13:17-43. 1981
[MR95] U. Montanari, F. Rossi. Contextual nets. Acta Informatica 32:545-596. 1995
[Pop91] L. Popova. On time Petri nets. J. Inform. Process. Cybern. EIK, 27:227-244. 1991
[Pop98] L. Popova-Zeugmann. Essential States in Time Petri Nets. Informatik-Bericht Nr. 96. Humboldt-Universität Berlin. 1998
[Ric85] G. Richter. A note on side-conditions and inhibitor arcs. PetriNet Newsletters 21:29-37. 1985
[SDS94] P. Sénac, M. Diaz, P. de Saqui-Sannes. Toward a formal specification of multimedia scenarios. Annals of telecommunications, 297-314. 1994
[SLL99] Lie-Quan Lee, Jeremy G. Siek, Andrew Lumsdaine. The generic graph component library. In OOPSLA'99. 1999
[Vog95a] W. Vogler. Timed testing of concurrent systems. Information and Computation, 121:149-171. 1995
[Vog95b] W. Vogler. Faster Asynchronous Systems. Lect. Notes Comp. Sci. 962:299-312. Springer. 1995
[Vog95c] W. Vogler. Faster Asynchronous Systems. Report Nr. 317. Universität Augsburg. 1995
[Vog96] W. Vogler. Efficiency of Asynchronous Systems and Read Arcs in Petri Nets. Report Nr. 352. Universität Augsburg. 1996
[Vog97] W. Vogler. Efficiency of Asynchronous Systems and Read Arcs in Petri Nets. Lect. Notes Comp. Sci. 1256:538-548. Springer. 1997

[VSY98] W. Vogler, A. Semenov, A. Yakovlev. Unfolding and Finite Prefix for Nets with Read Arcs. Lect. Notes Comp. Sci. 1466:501-516. Springer. 1998
[Ram74] C. Ramchandi. Analysis of asynchronous concurrent systems by timed petri nets. Technical Report TR 120, Project MAC. MIT. 1974
[MF76] P. Merlin, D. J. Faber. Recoverability of communication protocols. IEEE Trans. Communications COM-24:1036-1043. 1976

**Webseiten**

[BCPL] Boost Homepage. http://www.boost.org/
[BEPP] BE++ Homepage. http://bepp.elmar-bihler.de/
[DOX] Doxygen Website. http://www.stack.nl/~dimitri/doxygen/
[FSYO] FastAsy-Onlinedoku. http://fastasy.elmar-bihler.de/
[GVZ] Graphviz Homepage. http://www.graphviz.org/
[PEP] Pep-Homepage. http://theoretica.informatik.uni-oldenburg.de/~pep/
[PNK] Petri Net Kernel Homepage. http://www2.informatik.hu-berlin.de/top/pnk/
[STLP] STLPort Homepage. http://www.stlport.org/
[SVN] Subversion Homepage. http://subversion.tigris.org/

**Sonstiges**

[PFS] PAFAS Tool. z.Z. nicht öffentlich verfügbar, Private Kommunikation mit Flavio Corradini, Dipartimento Matematica ed Informatica, Camerino

# E Lebenslauf

Name: Elmar Bihler
Geboren: 26. Februar 1972 in Augsburg
Staatsangehörigkeit: deutsch
Familienstand: ledig

| | |
|---|---|
| 09/1978: | Besuch der König-Otto-Grundschule in Königsbrunn |
| 09/1982: | Übertritt in das Gymnasium Königsbrunn<br>Fremdsprachenreihenfolge: Englisch, Latein<br>Ausbildungszweig: mathematisch-naturwissenschaftlich |
| 07/1991: | Allgemeine Hochschulreife<br>Prüfungsfächer:<br>Mathematik, Physik, Englisch, Religion<br>Gesamtnote: 1,7 |
| 01-10/1993: | Freiberufliche Tätigkeiten im Bereich<br>Programmierung und Systemadministration |
| 10/1993: | Immatrikulation an der Universität Augsburg<br>im Studiengang Diplom-Mathematik mit Nebenfach Informatik |
| 1994 - 1997: | Tätigkeiten als studentische Hilfskraft an der Universität Augsburg<br>(Übungsgruppenleiter) |
| 1994 - 1998: | Nebentätigkeiten im Bereich<br>Programmierung und Systemadministration |
| 12/1995: | Vordiplom<br>Prüfungsnote: sehr gut (1,42) |
| 7/1998: | Diplom<br>Prüfungsnote: sehr gut (1,31) |
| ab 1998: | Freiberuflicher IT-Berater<br>Schwerpunkt im Bereich Software-Erstellung |
| 1998 - 2003 | Nebenbei wissenschaftliche Hilfskraft an der Uni Augsburg |
| 2003 - 2008: | Nebenbei Forschungstätigkeit<br>im Rahmen einer Promotion bei Prof. Dr. Walter Vogler |
| Dez. 2008 | Rigorosum<br>Prüfungsnote: magna cum laude (1) |