Petrinetze zum Entwurf selbststabilisierender Algorithmen

D I S S E R T A T I O N

zur Erlangung des akademischen Grades
doctor rerum naturalium
(dr. rer. nat.)
im Fach Informatik

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

von
Dipl.-Inf. Tobias Vesper
geboren am 2.3.1972 in Hennigsdorf

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Jürgen Mlynek

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. Bodo Krause

Gutachter:

1. Prof. Dr. Wolfgang Reisig
2. Prof. Dr. Hans Jürgen Prömel
3. Prof. Dr. Jörg Desel

eingereicht am:	6. Oktober 2000
Tag der mündlichen Prüfung:	8. Dezember 2000

Abstract

In 1974, *Edsger W. Dijkstra* suggested the notion of *self-stabilization*. A system is self-stabilizing if regardless of the initial state it eventually reaches a stable behaviour. This thesis focuses on the design of self-stabilizing algorithms. We introduce a new Petri net based method for the design of self-stabilizing algorithms. We validate our method on several case studies. In each of the case studies, our stepwise design starts from an algorithmic idea and leads to a new self-stabilizing algorithm.

One of these algorithms is a new randomized self-stabilizing algorithm for *leader election* in a ring of processors. This algorithm is derived from a published algorithm which we show to be incorrect. We prove that our algorithm is space-minimal. A further result is the first algorithm for token-passing in a asynchronous environment which works without *time-out* actions. Petri nets form a unique framework for modelling and verification of these algorithms.

Keywords:
self-stabiilization, distributed system, distributed algorithm, Petri net

Zusammenfassung

Edsger W. Dijkstra prägte im Jahr 1974 den Begriff *Selbststabilisierung (self-stabilization)* in der Informatik. Ein System ist selbststabilisierend, wenn es von jedem denkbaren Zustand aus nach einer endlichen Anzahl von Aktionen ein stabiles Verhalten erreicht. Im Mittelpunkt dieser Arbeit steht der Entwurf selbststabilisierender Algorithmen. Wir stellen eine Petrinetz-basierte Methode zum Entwurf selbststabilisierender Algorithmen vor.

Wir validieren unsere Methode an mehreren Fallstudien: Ausgehend von algorithmischen Ideen existierender Algorithmen beschreiben wir jeweils die die schrittweise Entwicklung eines neuen Algorithmus. Dazu gehört ein neuer randomisierter selbststabilisierender Algorithmus zur *Leader Election* in einem Ring von Prozessoren. Dieser Algorithmus ist abgeleitet aus einem publizierten Algorithmus, von dem wir hier erstmals zeigen, daß er fehlerhaft arbeitet. Wir weisen die Speicherminimalität unseres Algorithmus nach. Ein weiteres Ergebnis ist der erste Algorithmus, der ohne *Time-Out*-Aktionen selbststabilisierenden *Tokenaustausch* in asynchronen Systemen realisiert. Petrinetze bilden einen einheitlichen formalen Rahmen für die Modellierung und Verifikation dieser Algorithmen.

Schlagwörter:
Selbststabilisierung, verteiltes System, verteilter Algorithmus, Petrinetz

Danksagung

An erster Stelle gebührt mein Dank meinem Doktorvater Wolfgang Reisig. Von den Anfängen der Themenfindung bis hin zum letzten Tag vor der Abgabe hat er mich in inhaltlichen und Stilfragen lehrend und wohlwollend begleitet. Herzlich bedanken möchte ich mich auch bei den weiteren Gutachtern meiner Dissertation, Hans Jürgen Prömel und Jörg Desel, für die Erweiterung meines oftmals eingeschränkten Horizonts und wertvolle Kommentare zu Vorversionen dieser Arbeit.

Viele Kollegen haben einen großen Anteil an dieser Arbeit. Während zweier Dagstuhl-Seminare zum Thema „Selbststabilisierung“ konnte ich viele Ergebnisse diskutieren. Die Teilnehmer der Berliner „Kaffeerunde“ trugen mit notwendiger Kritik bei. Ohne die Studenten der Seminare über verteilte Algorithmen und Selbststabilisierung wären mehrere hier veröffentlichte Algorithmen nie entstanden. Allen gilt mein wärmster Dank. Herr Dr. Grubel lehrte mich den Gebrauch der Präpositionen, Wolfgang den Gebrauch des Genitivs, Birgit den Gebrauch eines Zeigestabs.

Für einen starken Rückhalt bedanke ich mich bei meiner Familie und allen Freunden. (Nebenbei vielen Dank an Birgit und Axel für den Hut, an Christian für die Tasse, an Jan für die Krawatte und an Andrea für die Musik)

Finanziert wurde ich seit 1996 von der Deutschen Forschungsgemeinschaft im Rahmen der Forschergruppe „Petrinetztechnologie“ an der Humboldt-Universität und der Technischen Universität Berlin. Vielen Dank.

Berlin, im Februar 2001

Inhaltsverzeichnis

1 Einleitung

> *Algorithms are still published with inadequate correctness proofs, and synchronization errors are still a major cause of "crashes" in computer systems.*
> Leslie Lamport, Nancy Lynch, [LL90]

Edsger W. Dijkstra prägte im Jahr 1974 in [Dij74] den Begriff *Selbststabilisierung (self-stabilization)* in der Informatik. Ein System ist selbststabilisierend, wenn es von jedem Zustand aus nach einer endlichen Anzahl von Aktionen einen stabilen Zustand erreicht. Wir erläutern dies in Abbildung 1.1. $\mathsf{Z}_{\mathsf{all}}$ be-

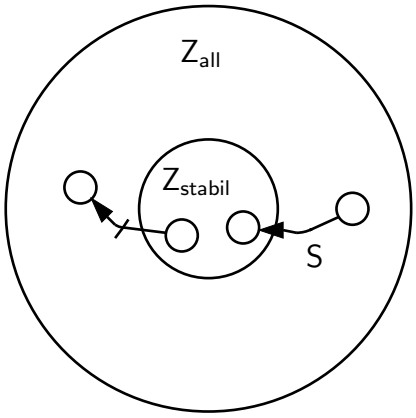


Abbildung 1.1: Selbststabilisierung

schreibt die Menge aller möglichen Zustände eines Systems. Unter diesen gibt es eine Menge $\mathsf{Z}_{\mathsf{stabil}}$ aller *stabilen* Zustände. Ein selbststabilisierendes System hat zwei wesentliche Eigenschaften:

1. Jeder Zustand aus $\mathsf{Z}_{\mathsf{all}} \setminus \mathsf{Z}_{\mathsf{stabil}}$ wird durch das System in einen Zustand aus $\mathsf{Z}_{\mathsf{stabil}}$ überführt. Dies ist in der Abbildung durch S dargestellt. S steht für eine endliche Folge von Zustandsübergängen im System.

2. Die Menge $\mathsf{Z}_{\mathsf{stabil}}$ ist abgeschlossen gegenüber den im System möglichen Zustandsübergängen: Kein Zustandsübergang führt aus $\mathsf{Z}_{\mathsf{stabil}}$ heraus.

Motivation von Selbststabilisierung:

Dijkstra traf in [Dij74] nur wenige Aussagen darüber, welche Zustände für ein bestimmtes System zur Menge $\mathsf{Z}_{\mathsf{stabil}}$ gehören. (So ist offensichtlich jedes System selbststabilisierend, wenn $\mathsf{Z}_{\mathsf{stabil}} = \mathsf{Z}_{\mathsf{all}}$ gilt.) Dijkstra definierte Selbststabilisierung für verteilte Systeme und beschrieb die Menge $\mathsf{Z}_{\mathsf{stabil}}$ intuitiv als die Menge der Systemzustände, in denen einzelne Agenten eines verteilten Systems *zulässig synchronisiert* sind. Er schrieb über solche Systeme:

> *Such systems (with what is quite aptly called "distributed control") have been designed, but all such designs I was familiar with were not "self-stabilizing" in the sense that, when once (erroneously) in an illegitimate state, they could—and usually did!—remain so forever.*
> Edsger W. Dijkstra, [Dij74]

Lamport [Lam83] motivierte die Menge $\mathsf{Z}_{\mathsf{stabil}}$, indem er als erster den Zusammenhang von Selbststabilisierung und *Fehlertoleranz* darstellte. Wir beziehen uns wieder auf Abbildung 1.1: Wir betrachten ein System in einem Anfangszustand. Wir definieren $\mathsf{Z}_{\mathsf{stabil}}$ als die Menge aller vom Anfangszustand aus erreichbaren Zustände dieses Systems. Nun betrachten wir einen Ablauf, in dem ein *transienter Fehler* eintritt. Ein transienter Fehler ist kein normaler Zustandsübergang des Systems: Er weist einer Menge von Variablen Werte zu, so daß der resultierende Zustand nicht in $\mathsf{Z}_{\mathsf{stabil}}$ ist. Ein selbststabilisierendes System garantiert in diesem Fall, daß ein stabiler Zustand von allein erreicht wird, d. h. ohne daß in das System von außen eingegriffen werden muß. Ein selbststabilisierendes System kann demnach eine bestimmte Art von Fehlern, transiente Fehler, tolerieren.

Eine andere Motivation der Beschäftigung mit selbststabilisierenden Algorithmen liegt in den Algorithmen selbst. Dijkstra suchte länger als ein Jahr lang nach einem selbststabilisierenden Algorithmus [Dij74]. Eine wesentliche Schwierigkeit beim Entwurf selbststabilisierender Algorithmen sind Fehler, die *nicht lokal erkennbar* sind. Ein Beispiel ist zyklisches Warten: Durch einen Fehler ist eine Information in einem verteilten System verloren gegangen. Jeder Agent wartet auf diese Information und geht davon aus, daß ein anderer Agent ihm diese Information irgendwann zur Verfügung stellen wird. Anhand seines lokalen Zustands kann kein Agent feststellen, daß ein Fehler aufgetreten ist. Nicht lokal erkennbare Fehler machen den Entwurf selbststabilisierender

verteilter Algorithmen zu einem schweren, aber interessanten Problem: Viele existierende selbststabilisierende Algorithmen basieren auf überraschenden und einfachen algorithmischen Ideen.

Hauptresultate dieser Arbeit:

Im Mittelpunkt dieser Arbeit steht der Entwurf selbststabilisierender Algorithmen. Wir stellen eine Petrinetz-basierte Methode zum Entwurf selbststabilisierender Algorithmen vor. Wir validieren unsere Methode an mehreren Fallstudien, die jeweils die Entwicklung eines Algorithmus in mehreren Schritten beschreiben. Im Ergebnis entstehen neue selbststabilisierende Algorithmen. Dazu gehört ein neuer randomisierter selbststabilisierender Algorithmus zur *Leader Election* in einem Ring von Agenten (siehe Abschnitt 3.1). Wir weisen die Speicherminimalität unseres Algorithmus nach. Unser Algorithmus ist abgeleitet aus einem publizierten Algorithmus, von dem wir hier erstmals zeigen, daß er fehlerhaft arbeitet. Ein weiteres Ergebnis ist der erste Algorithmus, der ohne *Time-Out*-Aktionen selbststabilisierenden *Tokenaustausch* in asynchronen Systemen realisiert. Alle bisher publizierten Algorithmen benutzen globale Time-Out-Aktionen, die in einem verteilten System nicht ohne weiteres lokal implementierbar sind (siehe Abschnitt 3.2). Petrinetze bilden einen einheitlichen formalen Rahmen für die Modellierung und Verifikation dieser Algorithmen.

Als einführendes Beispiel beschreiben wir in Abschnitt 1.3 Dijkstras ersten selbststabilisierenden Algorithmus. Zuvor stellen wir in Abschnitt 1.1 Thesen auf, die unserer Entwurfsmethode zugrunde liegen. Diese Thesen begründen den Aufbau der Arbeit. In Abschnitt 1.2 überblicken wir den Stand der Forschung.

1.1 Thesen und Aufbau der Arbeit

Diese Arbeit zeigt, daß *Petrinetze für den Entwurf selbststabilisierender Algorithmen adäquat sind.* „Entwurf“ umfaßt hierbei die algorithmische Idee, die Modellierung und die Verifikation eines Algorithmus. Diese Vorgehensweise wird durch folgende Thesen gestützt:

1. *Formale Methoden sind beim Entwurf selbststabilisierender Algorithmen unerläßlich.* In der Literatur existieren viele Veröffentlichungen über selbststabilisierende Algorithmen, die keine explizite formale Methode bei der Verifikation einsetzen. Konkret gibt es zwar fast immer ein formales operationelles Modell dieser Algorithmen (üblicherweise in einer

Programmiersprachen-ähnlichen Notation). Der Beweis der Korrektheit des Algorithmus setzt jedoch oft nicht auf dem operationellen Modell auf, sondern auf einer höheren Abstraktionsebene. Diese Vorgehensweise birgt die Gefahr, daß die gewählte Abstraktionsebene nicht zum Modell paßt. In der Tat werden wir zeigen, daß ein häufig zitierter selbststabilisierender Algorithmus inkorrekt ist. Die Ursache dafür, daß die Inkorrektheit bisher nicht erkannt wurde, vermuten wir darin, daß keine formale Methode für den Beweis verwendet wurde. Der Einsatz formaler Methoden und die Verwendung einer Abstraktion widersprechen sich jedoch nicht – sie ergänzen sich. Ausgehend vom operationalen Modell muß die Adäquatheit einer Abstraktion mit formalen Methoden gezeigt werden. Dies ist zwar noch keine Garantie für fehlerfreie Beweise, eröffnet jedoch dem Entwickler eines Algorithmus eine zusätzliche Kontrollmöglichkeit (und -pflicht) und ermöglicht im Prinzip die maschinelle Überprüfung eines Beweises (da sich formale Methoden immer auf mathematische und logische Schlußfolgerungen zurückführen lassen).

2. *Grundsätzliche Resultate über selbststabilisierende Algorithmen bedürfen einer allgemeinen formalen Systembeschreibungssprache.* Das erste grundlegende Resultat über selbststabilisierende Algorithmen war ihre Existenz (Dijkstra in [Dij74]). Dieses erste Resultat warf jedoch schon ein weiteres Problem auf: Dijkstra definierte nicht formal, was Selbststabilisierung bedeutet. Zum einen war die informelle Beschreibung von Dijkstra redundant (siehe [Sch93]). Zum anderen wurde über einzelne Wörter von Dijkstra diskutiert, um z. B. zu entscheiden, ob Dijkstra meinte, daß ein selbststabilisierendes System von jedem Zustand aus innerhalb einer beschränkten oder endlichen Anzahl von Schritten einen zulässigen Zustand erreicht [BGM93]. Eine allgemein anerkannte Definition lieferten Arora und Gouda in [AG93, Aro92], indem sie Selbststabilisierung in einen allgemeinen Kontext von Fehlertoleranz einordneten. Ein weiteres Problem sind unterschiedliche, teilweise implizite Systemannahmen. Innerhalb des Gebietes selbststabilisierender Algorithmen gibt es selten zwei Algorithmen, die dasselbe Problem unter denselben Systemannahmen lösen. Dies behindert den Vergleich der Qualität dieser Algorithmen. Einen Einblick in bisher verwendete Systemannahmen geben wir in Abschnitt 1.2. Petrinetze und ihre Abläufe bilden eine allgemeine formale Systembeschreibungssprache [Rei85]. Alle üblichen Systemannahmen lassen sich entweder durch syntaktische Restriktionen oder Einschränkungen der Menge der Abläufe (durch *Fairneßannahmen*) darstellen. Daher eignen sich Petrinetze als formale Grundlage von Möglichkeits- und Unmöglichkeitsresultaten.

3. *Es besteht Bedarf für methodische Unterstützung beim Entwurf selbststabilisierender Systeme.* Der Entwurf selbststabilisierender Systeme ist schwer, da bei selbststabilisierenden Systemen der „Ausnahmefall“ zum „Regelfall“ wird. So gibt es für die meisten konventionellen verteilten Algorithmen keine (oder zumindest keine veröffentlichte) selbststabilisierende Version. Dies liegt daran, daß ein Vorgehen, bei dem man den konventionellen Algorithmus als „Regelfall“ betrachtet und diesen Algorithmus dann um Aktionen für den Fehlerfall ergänzt, selten erfolgreich ist. (Eine Ausnahme bildet der selbststabilisierende Algorithmus zum wechselseitigen Ausschluß von Lamport [Lam86].) Vielmehr sind beim Entwurf selbststabilisierender Algorithmen oft sehr einfache *algorithmische Ideen* erfolgreich, die in einer fehlerfreien Umgebung zu einem korrekten, aber wenig effizienten Algorithmus führen. Je einfacher die Idee, desto größer ist die Chance, einen selbststabilisierenden Algorithmus erfolgreich zu entwerfen. Die von uns in dieser Arbeit zum Entwurf neuer Algorithmen benutzten algorithmischen Ideen sind sehr einfache abstrakte mathematische Ideen bzw. entstammen bereits existierenden selbststabilisierenden Algorithmen. Wir werden zeigen, daß eine Petrinetz-basierte Modellierung den Übergang von einer algorithmischen Idee zu einem formalen und nachweisbar korrekten Modell erleichtert.

   Dies läßt sich auf die Verifikation übertragen: Die Verifikation selbststabilisierender Algorithmen ist schwer und bedarf methodischer Unterstützung. Wir belegen dies mit zwei Beispielen. Dijkstra veröffentlichte erst mehr als zehn Jahre nach seinem ersten selbststabilisierenden Algorithmus dessen Beweis ([Dij86]). Ein Beweis dieses Algorithmus galt auch später als ein Prüfstein für Verifikationsmethoden (z. B. [Kes88]). Herman veröffentlichte in [Her90] und [Her92] zwei probabilistische selbststabilisierende Algorithmen. Beide Algorithmen sind strukturell sehr einfach. Jeder Agent hat nur eine Variable, die nur zwei bzw. drei verschiedene Werte annehmen kann. Alle Agenten haben das gleiche Programm. Das Programm besteht nur aus einer „if ... then A else B“-Anweisung, wobei A und B einfache Zuweisungen sind. Die Beweise dieser Algorithmen sind jedoch 5 bzw. 7 Seiten lang (und unbestritten elegant). Dies deutet auf eine probleminhärente Beweiskomplexität hin. Die Gründe dafür, daß das Verhältnis des Beweisaufwands zur Länge des Algorithmus bei selbststabilisierenden Algorithmen so groß ist, liegt in der Einzigartigkeit der zu beweisenden Eigenschaften. Es ist oft leichter (lokal für jede Aktion) zu zeigen, daß eine Eigenschaft *nie verletzt wird*, als daß sie *nur endlich oft verletzt werden kann*. Letztere Eigenschaft ist jedoch

typisch für selbststabilisierende Algorithmen. Die einzigen bislang etablierten Beweistechniken für solche Eigenschaften beruhen auf fundierten Mengen (Stabilisierungsfunktionen [Kes88] oder Induktion [LS97]). Jedoch gibt es viele Möglichkeiten, eine solche Technik einzusetzen – als globale Technik im Beweis (z. B. eine Stabilisierungsfunktion, aus der alle relevanten Aussagen folgen) oder als lokale Technik im Beweis (eine Stabilisierungsfunktion, die die algorithmische Idee widerspiegelt). Für den Einsatz als lokale Technik sind Petrinetze prädestiniert, da sich die notwendigen Abstraktionen direkt am Modell definieren lassen.

Der Aufbau dieser Arbeit ordnet sich diesen Teilthesen unter. Nachdem wir in den folgenden Abschnitten den Stand der Forschung zusammengefaßt und die Modellierungsmethode anhand eines Beispiels erläutert haben, führen wir die formalen Grundlagen unserer Modellierungs- und Spezifikationsmethode in Kapitel 2 ein. Wir modellieren einen Algorithmus mit einem *algebraischen Petrinetz* [Rei98] und benutzen zur Spezifikation der dynamischen Eigenschaften des Algorithmus eine *linear-time temporale Logik* [MP92]. Neu ist die Integration von *Randomisierungsannahmen* in den gewählten Formalismus. Der Formalismus bildet eine Grundlage, um selbststabilisierende Algorithmen, gleich welcher Systemannahmen, darzustellen und zu verifizieren. Wir werden hierbei exemplarisch Bezug auf unser Beispiel aus Abschnitt 1.3 nehmen.

In Kapitel 3 stellen wir Petrinetzmodelle selbststabilisierender Algorithmen vor. Bei neuen Algorithmen stellen wir die Methodik ihres Entwurfs in den Vordergrund. Wir beschreiben das zu lösende Problem und existierende Lösungsansätze. Wir entwickeln eine algorithmische Idee und auf dieser aufbauend einen vollständigen Algorithmus. Wir diskutieren die Schwierigkeiten beim Übergang von einer algorithmischen Idee zu einem Algorithmus. Petrinetze bringen die Möglichkeit einer graphischen Darstellung ihrer Abläufe mit sich. Wir zeigen, wie sich eine algorithmische Idee graphisch in typischen Abläufen widerspiegelt. Wir werden auch Petrinetzmodelle existierender Algorithmen angeben; entweder um aus ihnen neue, effizientere Algorithmen abzuleiten, oder um anhand des Petrinetzmodells einen intuitiveren Korrektheitsbeweis zu führen.

Auf Petrinetzen basierende Verifikationstechniken stellen wir in Kapitel 4 vor. Wir übernehmen bekannte Verifikationstechniken und entwickeln neue. Die speziellen Anforderungen, die selbststabilisierende Algorithmen stellen, ergeben einerseits die Möglichkeit, diese Techniken einzuschränken (im Vergleich zu Techniken, die sich bei konventionellen verteilten Algorithmen als vorteilhaft herausgestellt haben), andererseits jedoch auch die Notwendigkeit, die herkömmlichen Techniken zu erweitern. Wir stellen ein *Verifikationsschema* vor,

nach dem wir unsere Beweise strukturieren werden, um deren Komplexität besser zu beherrschen.

Im Kapitel 5 wenden wir die Beweistechniken an, um die Korrektheit der Algorithmen in Kapitel 3 nachzuweisen. Die Beweise folgen dem in Kapitel 4 eingeführten Verifikationsschema. Die algorithmischen Ideen, die zum Entwurf des Algorithmus beitragen, finden wir als Abstraktionsstufe im Beweis wieder.

Im Kapitel 6 resümieren wir und geben einen Ausblick.

1.2 Stand der Forschung

In diesem Abschnitt stellen wir wichtige Resultate und Meilensteine vor, an denen wir in dieser Arbeit anknüpfen. Im ersten Teil betrachten wir paradigmatische Erkenntnisse zu selbststabilisierenden Algorithmen. Wir konzentrieren uns hierbei auf Arbeiten, die grundlegende theoretische Resultate zur Möglichkeit bzw. Unmöglichkeit selbststabilisierender Algorithmen unter bestimmten Systemannahmen darstellen. Wir stützen uns hierbei auf Zusammenfassungen in [Sch93, FDG94, Her96]. Für unsere neuen Algorithmen führen wir die jeweils relevante Literatur separat in Kapitel 3 auf. Im zweiten Teil gehen wir auf grundlegende Arbeiten ein, die zur von uns gewählten Modellierungs- und Verifikationsmethode beitragen.

1.2.1 Selbststabilisierung

Nach der initialen Veröffentlichung Dijkstras [Dij74] wurden über mehrere Jahre hinweg keine weiteren Arbeiten über selbststabilisierende Algorithmen veröffentlicht. Lamport lobte die Ideen Dijkstras 1983 in [Lam83] und löste damit eine Welle von Arbeiten zu Selbststabilisierung aus. Diese führte schließlich dazu, daß selbststabilisierende Algorithmen heute ein anerkanntes Teilgebiet innerhalb der Welt der verteilten Algorithmen bilden – bei allen wesentlichen Konferenzreihen zu verteilten Algorithmen (z. B. WDAG/DISC, OPODIS, PODC, ICDCS) wurden in den letzten Jahren selbststabilisierende Algorithmen vorgestellt. Lamport wies auch als erster in [Lam83] auf den engen Zusammenhang von Selbststabilisierung und Fehlertoleranz hin.

Dijkstra hatte in [Dij74] sehr starke Systemannahmen: Er setzte die Existenz eines *zentralen Dämons (central demon)* voraus. Ein zentraler Dämon ist ein theoretisches Konzept, mit dessen Hilfe Dijkstra Abläufe eines verteilen Systems definiert. Ein zentraler Dämon hat eine globale Sicht auf das verteilte System. Jede Aktion eines Agenten hat eine Aktivierungsbedingung, die vom lokalen Zustand des Agenten und den lokalen Zuständen seiner Nachbarn ab-

hängt. In jedem Zustand überprüft der Dämon die Aktiviertheit aller Aktionen aller Agenten und wählt eine der aktivierten Aktionen aus. Diese Aktion tritt ein, und es entsteht ein Nachfolgezustand. Nun beginnt eine neue Runde. Der Dämon überprüft wiederum die Aktiviertheit aller Aktionen, wählt eine aus, usw. Die Annahme eines zentralen Dämons ist jedoch sehr unrealistisch: So ist es möglich, daß eine Aktion eines Agenten, die in einer Runde aktiviert ist, in der darauffolgenden Runde nicht mehr aktiviert ist, obwohl dieser Agent gar keine lokale Aktion ausgeführt und auch niemand seinen Eingabepuffer geleert hat. Somit setzt ein zentraler Dämon auch eine reale zentrale Kontrollinstanz voraus, die es in verteilten Systemen nicht gibt.

Brown, Gouda und Wu schlugen daraufhin in [BGW89] vor, die Annahme des zentralen Dämons durch die Forderung nach *Interferenzfreiheit (non-interfering state transitions)* zu ersetzen. Interferenzfreiheit bedeutet, daß eine aktivierte Aktion solange aktiviert bleibt, bis sie ausgeführt wird. Wenn also eine Aktion eines Agenten aktiviert ist, dann bleibt sie solange aktiviert, bis sie ausgeführt wird, unabhängig davon, welche weiteren Aktionen, inklusive der Aktionen dieses Agenten selbst, ausgeführt werden. Später wurde diese Forderung auch *verteilter Dämon (distributed demon)* genannt: Der verteilte Dämon kann in einer Runde eine beliebige nichtleere Teilmenge aller aktivierten Aktionen eintreten lassen. Interferenzfreie Systeme sind ohne eine zentrale Kontrollinstanz implementierbar. Interferenzfreiheit bedeutet intuitiv, daß ein Agent, der Werte seines Nachbarn liest und diese in seinen lokalen Variablen speichert, sich über die Aktualität dieser Werte keine Gedanken machen muß. Wenn eine seiner Aktionen aufgrund dieser Werte aktiviert ist, dann bleibt sie auch aktiviert, unabhängig von anderen Aktionen im System.

Zurück zu Dijkstra: Seine ersten selbststabilisierenden Algorithmen für den Tokenaustausch in Ringen (vgl. Abschnitt 1.3) waren *asymmetrisch*; es gab Agenten, die nach verschiedenen Protokollen arbeiteten. Dijkstra bemerkte in [Dij74], daß diese Eigenschaft wesentlich für die Existenz eines selbststabilisierenden Algorithmus in einem Ring von Prozessoren ist. Diese Eigenschaft wurde später genauer untersucht, da sie einen Schlüssel zum Verständnis selbststabilisierender Algorithmen bildete. (Das allgemeine Symmetrieproblem in verteilten Algorithmen wurde von Angluin in [Ang80] unabhängig von Selbststabilisierung umfassend charakterisiert.) Burns und Pachl gaben in [BP89] einen symmetrischen selbststabilisierenden Algorithmus für Tokenaustausch in Ringen an, in denen die Anzahl der Agenten jedoch prim sein muß. Hier ist das zugrundeliegende Agentennetzwerk asymmetrisch. Seit der Veröffentlichung von Gouda [Gou87] legen einige Algorithmen die Id-Basiertheit des Netzwerks zugrunde. Somit kann ein Agent seinen individuellen Identifikator bei einer Berechnung benutzen. Eine weitere Möglichkeit, mit Symmetriepro-

blemen umzugehen, ist Symmetriebruch durch *randomisierte Algorithmen*. Intuitiv erlaubt ein randomisierter Algorithmus den Agenten, als spezielle Aktion eine Münze zu werfen. Selbststabilisierung bedeutet dann, daß sich das System mit Wahrscheinlichkeit 1 stabilisiert. Wir werden auf randomisierte Algorithmen spezieller in Abschnitt 2.2.2 eingehen. Der erste randomisierte selbststabilisierende Algorithmus wurde von Israeli und Jalfon in [IJ90] vorgestellt. Zusammengefaßt gibt es mehrere Ansätze, Symmetrieprobleme zu vermeiden: verschiedene Protokolle, asymmetrische Netzwerke, eindeutige Identifikatoren und Randomisierung.

Bis 1989 wurden nur selbststabilisierende Algorithmen für verteilte Systeme entworfen, in denen die Agenten über gemeinsamen Speicher kommunizieren. Gouda und Multari veröffentlichten in [GM91, Mul89] die ersten selbststabilisierenden Algorithmen für *asynchrone* Systeme (in denen die Agenten über Nachrichtenaustausch kommunizieren). Wir werden einen neuen selbststabilisierenden Algorithmus für ein asynchrones System in Abschnitt 3.2 entwickeln. Deswegen verschieben wir die Diskussion der existierenden Algorithmen bis dahin und gehen in diesem Abschnitt nicht weiter auf diese Systeme ein.

Als letzten wichtigen Meilenstein werden wir im folgenden die *faire Komposition* selbststabilisierender Algorithmen betrachten. Faire Komposition wurde von Dolev, Israeli und Moran in [DIM93] vorgestellt. Wir beschreiben die Idee informell an einem einfachen Spezialfall. Angenommen, wir haben zwei selbststabilisierende Algorithmen A und B, so daß (grob gesprochen) das stabilisierte Verhalten von A als Eingabe für B dient. A könnte beispielsweise ein Algorithmus sein, der sich dahingehend stabilisiert, daß irgendwann ein spannender Baum in einem beliebigen zusammenhängenden Agentennetzwerk erzeugt wird. Nun könnte B ein Algorithmus sein, der irgendein Problem selbststabilisierend auf einem Baum von Agenten löst. Algorithmus B greift nur lesend auf die Variablen zu, die Algorithmus A benutzt. Faire Komposition von A und B bedeutet, daß beide Algorithmen nebeneinander ausgeführt werden, wobei gesichert werden muß, daß jeder Agent immer wieder Aktionen aus A und immer wieder Aktionen aus B ausführt. Das bedeutet, daß Algorithmus B ausgeführt wird, als hätte A schon einen stabilen Zustand erreicht. Der komponierte Algorithmus ist selbststabilisierend für ein beliebiges zusammenhängendes Agentennetzwerk. In jedem Ablauf erreicht A irgendwann dauerhaft einen stabilen Zustand, der als Anfangszustand für B genommen wird. Faire Komposition selbststabilisierender Algorithmen wurde umfassend untersucht, z. B. in [GH91, Sto93, Sie97]. Eine einfache Einführung gibt [Dol00].

1.2.2 Petrinetze

Petrinetze sind eine formale graphische Notation, die die *Lokalität von Ereignissen* in den Vordergrund stellt. Jedes Ereignis benötigt lokale Ressourcen und hat lokale Auswirkungen. Die Betonung der Lokalität von Ereignissen machen Petrinetze zu einem geeigneten Kandidaten für eine formale Beschreibung verteilter Algorithmen. Alle Operationen eines verteilten Algorithmus haben eine einfache korrespondierende Darstellung im Petrinetz: Eine lokale Ressource (z. B. eine lokale Variable eines Agenten) wird durch eine *Stelle* dargestellt; eine atomare Aktion wird durch eine *Transition* dargestellt, die durch Pfeile (*Kanten*) mit den jeweils benötigten Ressourcen verbunden ist. Ein *Zustand* eines Petrinetzes ist eine Verteilung von Werten (*Marken*) auf den Stellen.

Die *Distributed Algorithms' Working Notation (DAWN)* [Rei98, WWV+97, RKV+96] ermöglicht einen methodischen Zugang zur Modellierung und Verifikation verteilter Algorithmen. Die wichtigsten Bestandteile sind:

1. *algebraische Spezifikationen* [EM85] zur Beschreibung der statischen Grundlagen eines verteilten Algorithmus als abstrakten Datentyp, also die vorausgesetzten Eigenschaften des Agentennetzwerks und die Datenstruktur;

2. *algebraische Petrinetze* [Rei91] zur Beschreibung der Protokolle der Agenten als Systemnetz einerseits und des dynamischen Verhaltens in Form von Abläufen andererseits;

3. *linear-time temporale Logik* [MP92] zur Spezifikation der wesentlichen dynamischen Eigenschaften eines Algorithmus;

4. eine Sammlung von *Verifikationstechniken* zum Beweis der Korrektheit des Algorithmus: Jeder Ablauf des Systemnetzes erfüllt die spezifizierten Eigenschaften.

Wir werden auf diese Bestandteile in Kapitel 2 im einzelnen eingehen.

DAWN wurde vielfach zur Modellierung und Verifikation verteilter Algorithmen eingesetzt. Wir beschreiben im folgenden die wesentlichen Resultate. In [Rei98] verwendet Reisig DAWN, um viele aus der Literatur bekannte Algorithmen zu modellieren und zu verifizieren. Reisig zeigt, daß alle typischen Phänomene verteilter Algorithmen lokal beschrieben werden können. Wesentlich ist die Verwendung von *Netzschemata*, mit denen symbolisch eine ganze Klasse von Systemen beschrieben wird. Dies ist bei Netzwerkalgorithmen wichtig, in denen kein konkretes verteiltes System festgelegt

wird, sondern eine Klasse verteilter Systeme durch ihre Eigenschaften beschrieben wird. Weber et al benutzen in [WWV+97] mit dem gleichen Ziel einen technisch geringfügig abweichenden Formalismus. Unser hier gewählter Formalismus basiert auf [Rei98] und [WWV+97]. Exemplarisch wurden in [KRVW97, Des97, Wal95, RK97, DK98, KR96] einzelne verteilte Algorithmen modelliert und verifiziert. In [KV96, BKV99] wird gezeigt, wie DAWN-Beweise prinzipiell automatisch mit Theorembeweisern überprüft werden können.

DAWN wurde bereits in [Ves96, WWV+97, Rei98] zur Verifikation zweier selbststabilisierender Algorithmen verwendet. Diese Ansätze zeigen, daß es prinzipiell möglich ist, selbststabilisierende Algorithmen mit DAWN zu beschreiben. In dieser Arbeit suchen wir nach einem systematischen Ansatz. Ausgehend von DAWN untersuchen wir, welche DAWN-Techniken für selbststabilisierende Algorithmen relevant sind. Darüber hinaus erweitern wir DAWN um neue Konzepte für selbststabilisierende Algorithmen.

Als Abschluß dieses Abschnitts betrachten wir noch drei Arbeiten, die neben den genannten Arbeiten Petrinetze und selbststabilisierende Algorithmen verbinden. Gouda, Howell und Rosier untersuchen in [GHR90] ein Phänomen, welches sie *Look-Alike Configurations* nennen. Eine Look-Alike-Konfiguration tritt auf, wenn in zwei verschiedenen Systemzuständen die gleiche Sequenz von Aktionen möglich ist, wobei der erste Systemzustand zulässig, der zweite Systemzustand jedoch unzulässig ist. Die Autoren zeigen, daß ein verteiltes System mit Look-Alike-Konfigurationen keine Selbststabilisierung garantieren kann. Dieses Phänomen ist typisch für allgemeine Petrinetzsysteme. Der Hintergrund ist, daß Petrinetze als Beschreibungssprache sehr mächtig sind. Prinzipiell könnte man beispielsweise in einem Petrinetz modellieren, daß eine Variable in einem Zustand beliebig viele Werte hat. Ähnliche Resultate (mit teilweise anderen Phänomenen) finden die Autoren auch für andere formale Beschreibungssprachen. Wichtig ist daher ein Begriff von *Zustandsäquivalenz*: Für welche Zustände im formalen Modell gibt es einen korrespondierenden Zustand im verteilten System? Wie wird ein Zustand des verteilten Systems im formalen Modell repräsentiert? Typischerweise geht bei einem liberalen Zustandsäquivalenzbegriff wie in [GHR90] die Eigenschaft Selbststabilisierung verloren. Wir werden einen restriktiveren Äquivalenzbegriff betrachten:

1. Jeder *Anfangszustand* eines Petrinetzes entspricht einem *beliebigen Zustand* des verteilten Systems und umgekehrt.

2. Jede Aktion in einem Ablauf eines Petrinetzes entspricht einer Aktion in einem Ablauf des verteilten Systems und umgekehrt.

Cherkasova, Howell und Rosier zeigen in [CHR93] darüber hinaus, daß auch

für eingeschränkte Petrinetzklassen eine liberale Definition von Selbststabilisierung nicht sinnvoll ist, da alle Petrinetze, die eine derart definierte Eigenschaft erfüllen, eine pathologische Struktur haben (insbesondere keine Entsprechung als verteiltes System).

Peng und Makki verfolgen in [PM96] einen Ansatz, der verschiedene Erweiterungen der herkömmlichen Definition von Petrinetzen enthält. Mit diesen Erweiterungen modellieren Peng und Makki einen selbststabilisierenden Algorithmus. Diese Erweiterungen haben jedoch zwei Nachteile: Erstens sind sie im allgemeinen nicht in einem verteilten System implementierbar, und zweitens gehen durch diese Erweiterungen viele Verifikationstechniken verloren. Jedoch nutzen Peng und Makki wie wir die graphische Darstellbarkeit von Petrinetzen bei der Modellierung eines selbststabilisierenden Algorithmus aus.

1.3 Ein Beispiel

Wir werden im folgenden einen einfachen Algorithmus entwickeln, der *selbststabilisierenden Tokenaustausch* zwischen zwei Agenten realisiert. Aufbauend auf einem selbststabilisierenden Tokenaustausch können viele Probleme (Leader Election, Mutex, etc.) selbststabilisierend gelöst werden. Dolev präsentiert in [Dol00] mehrere Algorithmen, die auf selbststabilisierendem Tokenaustausch basieren.

Das Ziel von Tokenaustausch ist es, daß zwei Agenten eine spezielle Information (das „Token“) immer wieder austauschen. Der Begriff Token ist hierbei nicht als eine konkrete eindeutige Ressource zu verstehen. Vielmehr ist der Begriff Token eine Abstraktion. Wichtig ist nur, daß jeder Agent aufgrund seines lokalen Zustands entscheiden kann, ob er über diese Information verfügt oder nicht. Der folgende Algorithmus realisiert selbststabilisierenden Tokenaustausch in einem System mit zwei Agenten. Die zugrundeliegende algorithmische Idee ist so alt wie selbststabilisierende Algorithmen insgesamt; im ersten Artikel von Dijkstra [Dij74] stellt sich dieser Algorithmus als ein Spezialfall des ersten selbststabilisierenden Algorithmus für einen Ring von Agenten dar. Später wurde diese algorithmische Idee noch in vielfältiger Form benutzt, wie z. B. von Herman in einem randomisierten synchronen Algorithmus [Her90].

In Abbildung 1.2 ist das unserem Algorithmus zugrundeliegende Agentennetzwerk dargestellt. Zwei Agenten (l und r) kommunizieren über gemeinsamen Speicher: Es gibt Variablen, die l lesen und schreiben kann, und r nur liest, und es gibt Variablen, die r lesen und schreiben kann, und l nur liest.

Wir konkretisieren nun das Ziel von Tokenaustausch: Jeder Agent hat einen lokalen Zustand `token`. Er kann also anhand seines lokalen Zustands entschei-

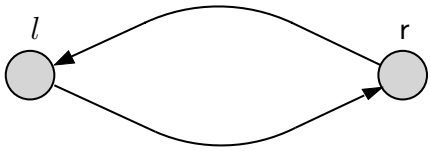


Abbildung 1.2: Ein einfaches Netzwerk

den, ob er ein Token hat. Wir fordern zwei Eigenschaften:

1. Höchstens ein Agent besitzt ein Token.

2. Jeder Agent besitzt das Token immer wieder.

Wir fordern nicht, daß immer ein Agent das Token hat, da es Zwischenzustände geben kann, in dem ein Agent seinen lokalen Zustand token bereits verlassen hat, um dem anderen Agenten zu signalisieren, daß dieser in seinen lokalen token-Zustand übergehen kann.

Wir betrachten nun Auswirkungen transienter Fehler. Transiente Fehler führen in einen beliebigen globalen Zustand. Typische unzulässige globale Zustände sind:

1. Beide Agenten besitzen ein Token.

2. Beide Agenten erwarten das Token.

Im zweiten Zustand geht jeder Agent aufgrund seines lokalen Zustandes davon aus, daß entweder der andere Agent das Token hat, oder daß der andere Agent ihm irgendwann das Token geben wird. Es ist anhand des unzulässigen globalen Zustands nicht entscheidbar, bei welchem Agenten der Fehler tatsächlich aufgetreten ist. Das Problem besteht darin, das Auftreten eines Fehlers zu erkennen.

Wir beschreiben im folgenden eine algorithmische Idee für unser System mit zwei Agenten: Jeder Agent hat einen Zustand, den wir *kritischen Bereich* nennen. Wenn ein Agent in seinem kritischen Bereich ist, dann hat er ein Token. Jeder Agent hat ein lokales Datenbit. Der eine Agent darf seinen kritischen Bereich betreten, wenn der Wert seines Bits dem Wert des Bits des anderen Agenten *gleicht*; der andere Agent darf den kritischen Bereich betreten, wenn die beiden Bits *verschiedene* Werte haben. Jeder Agent, der seinen kritischen Bereich verläßt, ändert den Wert seines Bits.

Die folgende Beschreibung des Algorithmus entspricht im wesentlichen Dijkstras Algorithmus in [Dij74]. Wir haben hier die Anzahl der Agenten festgelegt und den token-Zustand explizit modelliert. Bei Dijkstra hat ein Agent

ein Token, falls eine seiner Aktionen aktiviert ist. Dies ist jedoch bei Dijkstra nicht lokal entscheidbar.

Wir beschreiben zuerst die Datenstruktur. Jeder Agent hat zwei Bits z und s. Wir benutzen zur Unterscheidung Indizes; z_l ist beispielsweise das z-Bit des linken Agenten. z kann zwei Werte annehmen – pending oder token. s ist das Datenbit zum Vergleichen und kann die Werte 0 oder 1 annehmen.

In Abbildung 1.3 ist das Protokoll für den linken Agenten angegeben. Es

if $(z_l = \mathit{pending} \wedge z_r = \mathit{pending} \wedge s_l = s_r)$
    $z_l := \mathit{token};$
if $(z_l = \mathit{token})$
    $z_l := \mathit{pending};\ s_l := \neg s_l;$

Abbildung 1.3: Protokoll für den Agenten l

besteht aus zwei *guarded commands*. Ein guarded command besteht aus einer Aktivierungsbedingung und einem Aktionsteil. Wenn die Aktivierungsbedingung eines guarded commands erfüllt ist, dann kann der Aktionsteil als eine atomare Aktion ausgeführt werden.

In Abbildung 1.4 ist das Protokoll für den rechten Agenten angegeben.

if $(z_l = \mathit{pending} \wedge z_r = \mathit{pending} \wedge s_l \neq s_r)$
    $z_r := \mathit{token};$
if $(z_r = \mathit{token})$
    $z_r := \mathit{pending};\ s_r := \neg s_r;$

Abbildung 1.4: Protokoll für den Agenten r

Wir werden nun in zwei Schritten ein Petrinetzmodell dieses Algorithmus angeben. Zuerst stellen wir das System in einem konkreten Anfangszustand dar. Danach stellen wir die Menge aller möglichen globalen Zustände des verteilten Systems in einem Netzschema parametrisiert dar. In Abbildung 1.5 ist das Petrinetzmodell Σ_1 für den Algorithmus von Dijkstra angegeben. Als Anfangszustand haben wir folgende Variablenwerte gewählt: $z_l = z_r =$ token und $s_l = s_r = 0$. Dies entspricht einem der oben beschriebenen unzulässigen Zustände: Beide Agenten haben ein Token. Wir beschreiben die einzelnen Netzelemente im folgenden für den linken Agenten, für den rechten Agenten gilt Analoges. Der Agent l ist im linken Teil der Abbildung modelliert. Die runden Stellen $\mathsf{pending}_l$ und token_l modellieren die beiden möglichen Werte der Variable z_l. Die viereckigen Transitionen modellieren die möglichen Zustandsübergänge. Jede Transition steht für ein guarded command des Protokolls in Abbildung 1.3. Den Wert 0 der Variable s_l haben wir explizit dargestellt.

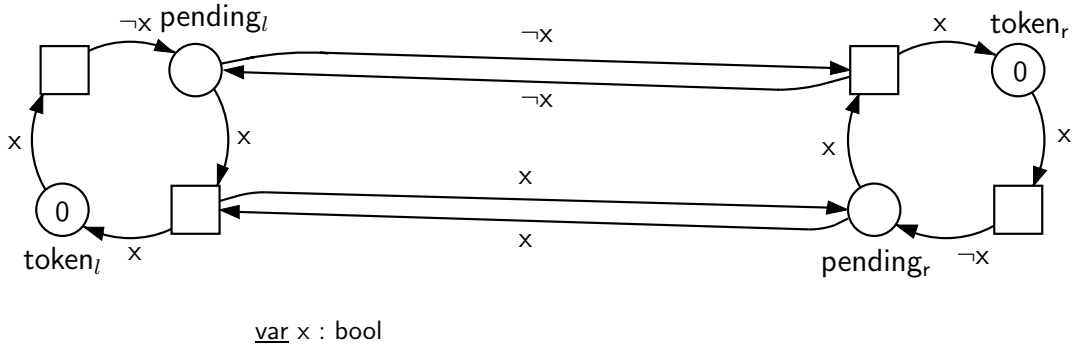


Abbildung 1.5: Σ_1 – Tokenaustausch

Ein Zustand des verteilten Systems entspricht einem Zustand des Netzes. Ein Zustand eines Netzes ist eine Verteilung von *Marken* auf den Stellen des Netzes. In unserem Beispiel liegt auf token_l eine Marke mit dem Wert 0, auf $\mathsf{pending}_l$ liegt keine Marke.

Im Protokoll in Abbildung 1.3 sind z_l und s_l zwei gleichberechtigte Variablen. In unserem Modell in Abbildung 1.5 sind sie nicht mehr gleichberechtigt. Wir haben die Werte von z_l explizit durch zwei Stellen modelliert; die Werte von s_l sind als mögliche Werte von Marken gegeben. Diese Unterscheidung ist durch den Algorithmus nicht vorgegeben. Wir haben diese Darstellung aufgrund ihrer Kompaktheit gewählt.

Eine Transition eines Petrinetzes kann in verschiedenen *Modi* schalten. Dies korrespondiert mit einem guarded command in einer Pseudocode-Beschreibung: Ein guarded command kann in verschiedenen Zuständen aktiviert sein. Beispielsweise kann das erste guarded command in Abbildung 1.3 sowohl in einem Zustand aktiviert sein, in dem $s_l = s_r = 0$ gilt, als auch in einem Zustand, in dem $s_l = s_r = 1$ gilt. Wie im Pseudocode stellen wir auch im Petrinetz diese beiden Möglichkeiten jeweils nicht explizit dar, sondern wir fassen sie zusammen. Hierzu benutzen wir in unserem Beispiel die Variable x, die den Wertebereich Boolesche Zahlen hat (siehe Abbildung 1.5). Jede Belegung der Variablen x ergibt einen Schaltmodus einer Transition. In Abbildung 1.6 ist das Schalten einer Transition von Σ_1 in einem Modus dargestellt. Im linken Teil der Abbildung ist eine *aktivierte Transition* dargestellt. Eine Transition ist aktiviert, wenn sie in einem Modus aktiviert ist, hier im Modus $x = 0$. Die Anschrift $\neg x$ an der Kante nach $\mathsf{pending}_l$ beschreibt, welche Marke auf $\mathsf{pending}_l$ produziert werden soll, wenn die Transition schaltet. $\neg$ ist die Negation in den Booleschen Zahlen. Im rechten Teil der Abbildung ist der Zustand nach dem Schalten der Transition dargestellt. Der entstandene Zustand des Petrinetzes

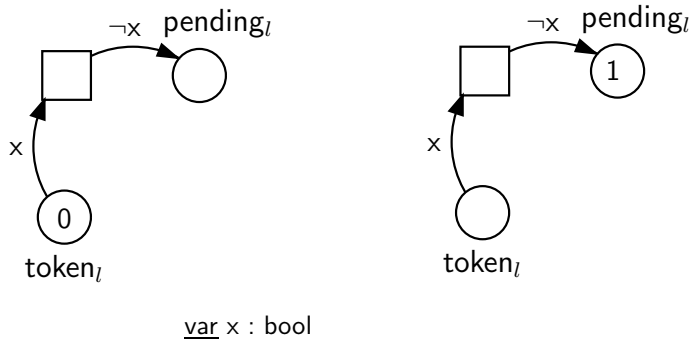


Abbildung 1.6: Schalten in einem Modus

entspricht wiederum einem Zustand des verteilten Systems.

Zusammengefaßt haben wir in Abbildung 1.5 ein Modell eines verteilten Systems in einem bestimmten Zustand dargestellt. Ein Zustand entspricht einer Belegung mit Marken. Das Eintreten einer Aktion entspricht dem Schalten einer Transition in einem Modus.

Im folgenden beschreiben wir das *dynamische Verhalten* des modellierten Systems. Das dynamische Verhalten eines verteilten Systems ist als die Menge der *Abläufe* des Systems definiert. Informell ist ein Ablauf eine maximale Folge von Aktionen, die ausgehend vom Anfangszustand im System eintreten können. Im Petrinetzmodell entspricht ein Ablauf einer maximalen Schaltfolge von Transitionen (in bestimmten Modi). Wir stellen einen Ablauf ebenfalls graphisch dar. In Abbildung 1.7 ist ein Ablaufstück des Netzes Σ_1 angegeben. Tatsächlich sind sogar zwei Schaltfolgen dargestellt, da wir durch die Graphik

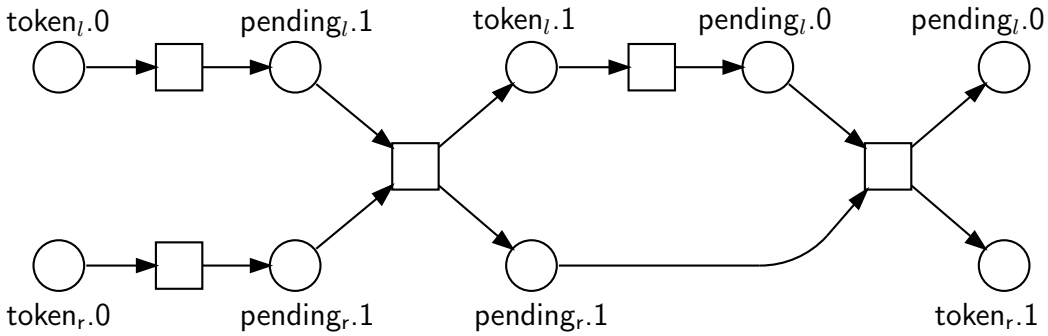
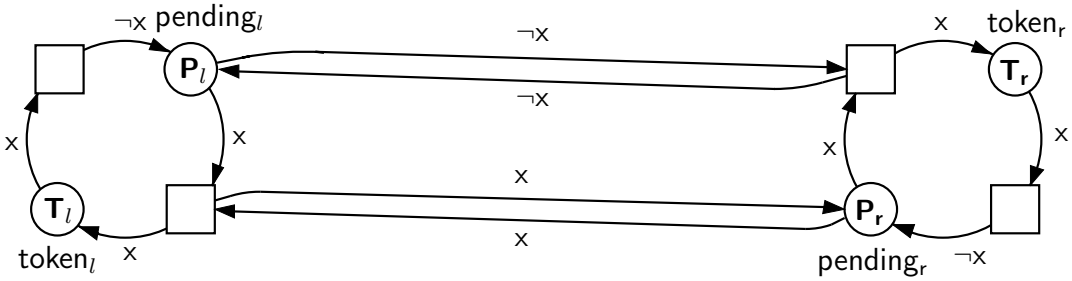


Abbildung 1.7: Ein Ablaufstück von Σ_1

nicht festgelegt haben, ob zuerst der linke Agent oder der rechte Agent eine Aktion ausführt. Wir haben also wiederum eine kompakte Darstellung für eine

Menge von Abläufen gewählt. Diese Darstellung betont nur die kausale Abhängigkeit der Aktionen voneinander und wird als *Halbordnungsablauf* bezeichnet. Sie entspricht der Intuition eines verteilten Systems: In einem realen verteilten System werden die beiden Aktionen *unabhängig* voneinander ausgeführt, da sie keinerlei gemeinsame Ressourcen benötigen.

Bisher haben wir im Petrinetz nur einen Anfangszustand des verteilten Systems dargestellt. Jeden anderen Zustand des verteilten Systems (das heißt, jede andere Belegung der Variablen z_l, z_r, s_l und s_r) können wir auch als ein Systemnetz mit einem anderen Anfangszustand darstellen. In Abbildung 1.8 fassen wir die verschiedenen möglichen Anfangszustände wiederum durch Parameter zusammen. Wir erhalten das *Netzschema* Σ_2 mit den Parametern P_l,



const T_l, P_l, T_r, P_r : multiset of bool
var x : bool

$\exists\, x \in \mathsf{bool} : \mathsf{T}_l + \mathsf{P}_l = [x]$
$\exists\, x \in \mathsf{bool} : \mathsf{T}_r + \mathsf{P}_r = [x]$

Abbildung 1.8: Σ_2 – Tokenaustausch

T_l, P_r und T_r. Wir erklären das Schema wiederum nur für den linken Agenten. In der Spezifikation unter dem Netz legen wir den Typ der Parameter P_l und T_l fest. Beide stehen für eine beliebige Ansammlung (sogenannte *Multimengen*) Boolescher Werte. Durch das Axiom $\exists\, x \in \mathsf{bool} : \mathsf{T}_l + \mathsf{P}_l = [x]$ schränken wir die möglichen Werte jedoch wieder ein. Intuitiv sagt dieses Axiom, daß eine Boolesche Zahl x existiert, so daß auf den Stellen $\mathsf{pending}_l$ und token_l zusammen genau eine Marke mit dem Wert x liegt und darüber hinaus keine Marke mit einem anderen Wert. Das Axiom ermöglicht beispielsweise eine Belegung des Parameters T_l mit einer 0-Marke und des Parameters P_l mit gar keiner Marke. Der resultierende Anfangszustand ist genau der Anfangszustand des linken Agenten in Abbildung 1.5. Insgesamt haben wir in Abbildung 1.8 ein Modell des Algorithmus von Dijkstra mit einem beliebigen globalen Zustand angegeben. Jeder mögliche Anfangszustand des Netzschemas entspricht einem globalen Zustand des verteilten Systems. Jeder globale Zustand des verteilten Systems entspricht einem Anfangszustand des Netzschemas. Basierend auf

diesem Modell können wir die Korrektheit des Algorithmus spezifizieren und verifizieren.

Wir spezifizieren dynamische Eigenschaften eines Algorithmus mit *linear-time temporaler Logik*. Eine Eigenschaft gilt in einem Netz, falls alle Abläufe dieses Netzes die Eigenschaft erfüllen. Analog gilt eine Eigenschaft in einem Netzschema, wenn alle Abläufe aller Instanzen des Netzschemas diese Eigenschaft erfüllen.

Die drei wesentlichen Eigenschaften sind:

1. Der linke Agent wird immer wieder das Token erhalten.
2. Der rechte Agent wird immer wieder das Token erhalten.
3. Schließlich gibt es höchstens ein Token.

Eine Formulierung dieser drei Eigenschaften in unserer temporalen Logik sieht wie folgt aus:

$$\Sigma_2 \models \mathit{true} \mapsto (|\mathsf{token}_l| > 0) \tag{1.1}$$

$$\Sigma_2 \models \mathit{true} \mapsto (|\mathsf{token}_\mathsf{r}| > 0) \tag{1.2}$$

$$\Sigma_2 \models \Diamond\,\Box(|\mathsf{token}_l + \mathsf{token}_\mathsf{r}| \leqslant 1) \tag{1.3}$$

Hier kommen in den Formeln Namen von Stellen des Netzschemas vor. Weiterhin kommen die *temporalen Operatoren* $\mapsto$ (*leads-to*), $\Diamond$ (*irgendwann* oder *schließlich*) und $\Box$ (*immer*) vor. Intuitiv kann man die Formeln wie folgt lesen: (1.1) sagt, daß in jedem Ablauf (einer Instanz) von Σ_2 von jedem Zustand aus (*true*) ein Zustand erreicht wird, in dem auf der Stelle token_l eine Marke liegt. Eigenschaft (1.2) sagt das gleiche über den rechten Agenten aus. (1.3) sagt, daß in jedem Ablauf schließlich immer auf den Stellen token_l und $\mathsf{token}_\mathsf{r}$ zusammen höchstens eine Marke liegen wird.

Die ersten beiden Eigenschaften sind (wenn wir die Struktur der vorkommenden temporalen Operatoren betrachten) typisch für verteilte Algorithmen. DAWN stellt für ihren Beweis mächtige Techniken zur Verfügung: *Stelleninvarianten, Beweisgraphen* und *Ableseregeln für Leads-to-Eigenschaften*. Die Formelstruktur $\Diamond\,\Box\ldots$ ist typisch für selbststabilisierende Algorithmen. Wir werden in Kapitel 4 Techniken für ihren Beweis zur Verfügung stellen. Im Kapitel 5 werden wir diese Techniken an mehreren Fallstudien erproben. Wir werden hierbei insbesondere die Methodik ihrer Anwendung diskutieren. Prinzipiell

könnten wir bei unserem Beispiel zwei Ansätze zum Beweis von (1.3) verfolgen. Beide führen aufgrund der Einfachheit des Beispiels problemlos zum Ziel. Beim ersten Ansatz unterscheiden wir viele Einzelfälle einer möglichen Markenverteilung im Netz. Für jeden Einzelfall zeigen wir die Eigenschaft. Der zweite Ansatz greift die algorithmische Idee wieder auf. Ein Token wird abwechselnd umhergereicht. Dazu führen wir eine Abstraktion ein: Wir zählen die Token im System. Wenn auf token_l eine Marke liegt, addieren wir eins. Wenn auf $\mathsf{token}_\mathsf{r}$ eine Marke liegt, addieren wir ebenfalls eins. Wenn auf $\mathsf{pending}_l$ und $\mathsf{pending}_\mathsf{r}$ jeweils eine Marke liegt, addieren wir auch eins. Diese Summe definiert eine *Anzahl der Token in einem Zustand*. Die verbleibende Argumentation ist dann wie folgt. Keine Transition kann die Anzahl der Token erhöhen. Also bleibt schließlich die Anzahl der Token konstant. Wir wissen durch Stelleninvarianten, daß die Anzahl der Token immer eins oder zwei ist. Für den Fall „zwei Token“ ergibt sich schnell ein Widerspruch. Also ist schließlich die Anzahl der Token immer eins, was die Eigenschaft impliziert. Wir werden argumentieren, daß beim Beweis selbststabilisierender Algorithmen die zweite Vorgehensweise zu wesentlich intuitiveren und strukturierteren Beweisen führt. Hierzu werden wir in einer Fallstudie auch mehrere existierende Beweise vergleichen (einen neuen von uns mit dem Originalbeweis und einem anderen DAWN-basierten Beweis). Auf geht's zum nächsten Kapitel mit der notwendigen Formalisierung der Konzepte für eine Petrinetz-basierte Modellierung selbststabilisierender Algorithmen.

2 Netze und verteilte Algorithmen

In diesem Kapitel definieren wir die wesentlichen Grundlagen unserer Petrinetzmodelle. Wir beschreiben zunächst semantisch *Systemnetze* und ihre *Abläufe*. Insbesondere formalisieren wir, welche Abläufe *fair* sind und einer *Randomisierungsannahme* genügen. Danach führen wir die DAWN-typische syntaktische Repräsentation eines Systemnetzes ein. Wir folgen hierbei im wesentlichen den Definitionen von Reisig [Rei98]. Spezielle Topologien von Agentennetzwerken, wie Ringe oder Ketten, werden wir als abstrakten Datentyp beschreiben. Nach der Formalisierung dynamischer Eigenschaften werden wir abschließend die Darstellung eines Algorithmus als ein *Netzschema* ausnutzen, um das zugrundeliegende Agentennetzwerk und einen beliebigen initialen Zustand eines verteilten Systems zu modellieren. Wir werden die Definitionen anhand des bereits aus der Einleitung bekannten Algorithmus von Dijkstra erläutern. Wir empfehlen dem Leser, der sich in erster Linie für die Algorithmen interessiert, direkt in Kapitel 3 weiterzulesen und gegebenenfalls auf dieses Kapitel zurückzugreifen.

2.1 Systemnetze

Definition 2.1 (Netz) Für zwei disjunkte Mengen P und T sowie eine Relation $F \subseteq (P \times T) \cup (T \times P)$ definiert $N = (P, T, F)$ ein *Netz*.

Ein Element von P, T und F heißt *Stelle*, *Transition* bzw. *Kante*. (Der Buchstabe P entstammt der alternativen Bezeichnung „Platz“; die Menge der Kanten (F) wird in der Petrinetzliteratur auch als „Flußrelation“ bezeichnet.) In Abbildungen stellen wir eine Stelle durch einen Kreis oder eine Ellipse, eine Transition durch ein Quadrat und eine Kante durch einen Pfeil dar.

Notation 2.2 Für ein gegebenes Netz N bezeichnen wir die Mengen der Stellen, Transitionen und Kanten von N mit P_N, T_N bzw. F_N.

Beispiel 2.3 In Abbildung 2.1 ist ein Netz N_1 graphisch dargestellt.

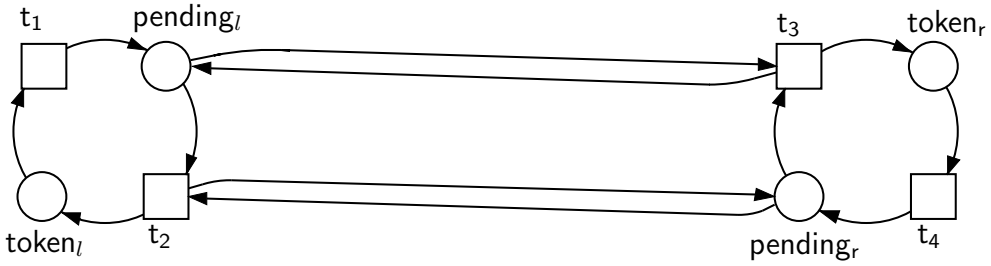


Abbildung 2.1: N_1 – ein Netz

Definition 2.4 (Vor- und Nachbereich) Für ein Element $x \in P_N \cup T_N$ eines Netzes N sind der *Vorbereich* ${}^\bullet x$ und der *Nachbereich* $x^\bullet$ definiert durch ${}^\bullet x = \{y \in P_N \cup T_N : (y, x) \in F_N\}$ und $x^\bullet = \{y \in P_N \cup T_N : (x, y) \in F_N\}$.

Notation 2.5 Wir werden im weiteren nur noch Stellen-berandete Netze betrachten. Für ein Stellen-berandetes Netz gilt, daß der Vor- und der Nachbereich jeder Transition mindestens eine Stelle enthält.

In einer Multimenge kann (im Gegensatz zu einer Menge) ein Element mehrfach auftreten. Wir definieren eine Multimenge formal als eine Abbildung, die jedem Element einer Grundmenge eine Häufigkeit zuordnet. Bei der Modellierung eines verteilten Algorithmus fassen wir gleichartige Ressourcen durch eine Multimenge zusammen.

Notation 2.6 $\mathbb{N}$ bezeichnet die Menge der natürlichen Zahlen, inklusive 0, $\mathbb{B}$ bezeichnet die Menge der Booleschen Werte. Wir benutzen alternativ die Schreibweisen $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$ oder $\mathbb{B} = \{1, 0\}$.

Definition 2.7 (Multimenge, endliche $\sim$) Für eine Menge A heißt eine Funktion $M : A \longrightarrow \mathbb{N}$ *Multimenge über A*. M ist *endlich*, gdw. die Menge $\{a \in A : M(a) \neq 0\}$ endlich ist.

Notation 2.8 Sei A eine Menge.

i. $\mathcal{M}(A)$ bezeichnet die Menge aller endlichen Multimengen über A;

ii. $[\,]$ bezeichnet die leere Multimenge, d. h. $\forall a \in A : [\,](a) = 0$;

iii. Für $a_0, \ldots, a_n \in A$ ist $[a_0, \ldots, a_n] \in \mathcal{M}(A)$, wobei für $a \in A$ gilt $[a_0, \ldots, a_n](a) = |\{i : 0 \leqslant i \leqslant n \text{ und } a_i = a\}|$;

iv. Für eine Multimenge M schreiben wir $M[a]$ anstelle von $M(a)$, um die *Vielfachheit eines Elements* a in M auszudrücken;

v. Wenn durch den Kontext Eindeutigkeit garantiert ist, schreiben wir nur A für die kanonische Multimenge $M_A \in \mathcal{M}(A)$, wobei $\forall a \in A : M_A[a] = 1$;

vi. Wenn durch den Kontext Eindeutigkeit garantiert ist, schreiben wir nur a für die einelementige Multimenge $[a]$.

Definition 2.9 (Operationen auf Multimengen) Für eine Menge A seien $M, M' \in \mathcal{M}(A)$. Dann gilt:

i. Für $n \in \mathbb{N}$ ist $(n \cdot M) \in \mathcal{M}(A)$, wobei $(n \cdot M)[a] = n \cdot M[a]$,

ii. $M' \leqslant M$, gdw. $\forall a \in A : M'[a] \leqslant M[a]$,

iii. $M + M' \in \mathcal{M}(A)$, wobei $(M + M')[a] = M[a] + M'[a]$,

iv. für $M' \leqslant M$ ist $M - M' \in \mathcal{M}(A)$, wobei $(M - M')[a] = M[a] - M'[a]$,

v. $|M| \in \mathbb{N}$, mit $|M| = \sum_{a \in A} M[a]$.

Notation 2.10 Bei der Addition von Multimengen benutzen wir analog zu den natürlichen Zahlen die abkürzende Summenschreibweise. Ein Beispiel gibt die nachfolgende Definition.

Definition 2.11 (Kanonische Erweiterung einer Funktion) Für eine Funktion $f : A \longrightarrow B$ heißt die Funktion $f_{\mathcal{M}} : \mathcal{M}(A) \longrightarrow \mathcal{M}(B)$ *kanonische Erweiterung von* f *auf Multimengen*. Sie ist wie folgt definiert:

$$f_{\mathcal{M}}(m) = \sum_{a \in A} \big(m[a] \cdot [f(a)]\big).$$

Beispiel 2.12 Wir definieren *Projektionen* für Multimengen von Tupeln als kanonische Erweiterung der entsprechenden Definition für Mengen. Da wir Projektionen in den folgenden Kapiteln sehr häufig verwenden werden, vereinbaren wir eine Kurzschreibweise: Wir notieren die Projektion einer (Multi-) Menge auf die i-te Komponente durch den nachgestellten Index i, z. B.:

$$[(0,0),(0,0),(1,0)]_2 = 2 \cdot [0] + 1 \cdot [0] = [0,0,0].$$

In einem Petrinetz ist ein *Zustand* durch eine Verteilung von Marken auf den Stellen des Netzes definiert: Ein Zustand weist jeder Stelle eine Multimenge von Marken eines zugehörigen Domains zu. In der Literatur wird ein Zustand eines Petrinetzes auch Markierung genannt.

Definition 2.13 (Universum, Domain, Zustand, Marke, Aktion) Ein *Universum* $\mathcal{U}$ für ein Netz N weist jeder Stelle $p \in P_N$ eine Menge A_p, den *Domain* von p, zu. Ein *Zustand* C von N weist jeder Stelle $p \in P_N$ eine Multimenge $C(p) \in \mathcal{M}(A_p)$, die *Marken auf* p, zu. Für eine Transition $t \in T_N$ weist eine *Aktion* (t, m) jeder Kante $f \in F_N$, mit $f = (p, t)$ oder $f = (t, p)$ eine nichtleere Multimenge $m(f) \in \mathcal{M}(A_p)$ zu. Für jedes andere Paar $f \in (P \times T) \cup (T \times P)$ gilt $m(f) = [\,]$.

Beispiel 2.14 In Abbildung 2.2 ist ein Zustand C_0 eines Netzes N_2 graphisch dargestellt. Jeder Stelle weisen wir den Domain $\mathbb{B}$ zu. Der dargestellte Zustand

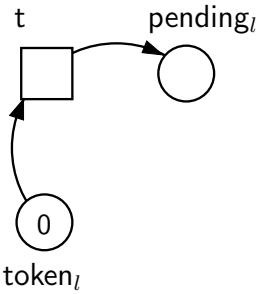


Abbildung 2.2: Ein Zustand C_0 eines Netzes N_2

ordnet der Stelle token_l die Multimenge [0] und der Stelle $\mathsf{pending}_l$ die leere Multimenge zu. Für die Transition t definieren wir die Aktion (t, m_0), die der Kante (token_l, t) die Multimenge [0] zuweist und der Kante $(t, \mathsf{pending}_l)$ die Multimenge [1]. Außerdem definieren wir die Aktion (t, m_1), die der Kante (token_l, t) die Multimenge [1] zuweist und der Kante $(t, \mathsf{pending}_l)$ die Multimenge [0].

Marken werden in der Petrinetzliteratur oft auch Token genannt. Da hier eine Kollision mit dem Begriff Token aus der Literatur zu verteilten Algorithmen auftritt (z. B. „Tokenaustausch“), benutzen wir den Begriff Marke. Ein Zustand kann durch das Eintreten einer Aktion modifiziert werden. Das Eintreten einer Aktion ändert lokal die Verteilung der Marken.

Definition 2.15 (Aktiviertheit, Eintreten, Schritt) Sei N ein Netz mit einem Universum $\mathcal{U}$. Eine Aktion (t, m) ist *aktiviert* in einem Zustand C, gdw. $\forall p \in {}^\bullet t : m(p,t) \leqslant C(p)$ gilt. Das *Eintreten* einer aktivierten Aktion (t, m) in einem Zustand C führt zu einem Zustand C' mit $\forall p \in P_N : C'(p) = C(p) - m(p,t) + m(t,p)$. In diesem Fall nennen wir $(C, (t, m), C')$ einen *Schritt*.

Beispiel 2.16 Im Beispiel 2.14 ist im Zustand C_0 die Aktion (t, m_0) aktiviert, die Aktion (t, m_1) ist nicht aktiviert. Das Eintreten der Aktion (t, m_0) führt zu einem Zustand C_1, der der Stelle token_l die leere Multimenge und der Stelle $\mathsf{pending}_l$ die Multimenge [1] zuweist.

Intuitiv weist jede Aktion (t, m) jeder Stelle im Vorbereich von t die Multimenge von Marken zu, welche die Transition zu ihrem Eintreten benötigt. Wenn in einem Zustand C alle diese Marken vorhanden sind, dann ist (t, m) aktiviert, und diese Aktion kann eintreten. Hierbei kann ein *Konflikt* auftreten, falls mehrere Transitionen durch dieselbe Marke aktiviert sind. Wir formalisieren den Begriff des Konflikts zweier Aktionen:

Definition 2.17 (Konflikt) Sei wiederum N ein Netz mit einem Universum $\mathcal{U}$. Zwei Aktionen (t, m) und (t', m') stehen *in Konflikt* in einem Zustand C, gdw. (t, m) und (t', m') beide in C aktiviert sind und es eine Stelle $p \in P_N$ und eine Marke $a \in A_p$ gibt, mit $C(p)[a] < m(p,t)[a] + m'(p,t')[a]$.

Beispiel 2.18 Im Beispiel 2.14 stehen im Zustand C_0 die Aktionen (t, m_0) und (t, m_1) nicht in Konflikt, da die Aktion (t, m_1) nicht aktiviert ist. Die Aktion (t, m_0) steht jedoch in Konflikt zu sich selbst, da auf der Stelle token_l die Marke 0 nur einmal vorhanden ist.

Die konkrete Definition eines Konflikts ist (für Netze mit Multimengen von Marken) in der Literatur nicht einheitlich. Je nach Ursprung werden durch die Definition verschiedene Konsequenzen eines Konflikts betont: In [WWV+97] gilt für zwei in Konflikt stehende Aktionen, daß es eine Marke gibt, die von beiden Aktionen *genutzt werden könnte*. Für uns ist insbesondere bei der Formalisierung von Randomisierung (Abschnitt 2.2.2) wichtig, daß zwei Aktionen, die in Konflikt zueinander stehen, nicht *nebenläufig* schalten können, d. h. daß es eine Marke gibt, die von beiden Aktionen *gebraucht wird* und nicht in ausreichender Anzahl zur Verfügung steht. Wir werden auf die Nebenläufigkeit bei der Definition eines Kausalnetzes (siehe Def. 2.23) zurückkommen.

Ein Systemnetz ist ein Netz mit einem Universum, einer definierten Menge von Aktionen und einem ausgezeichneten Anfangszustand:

Definition 2.19 (Systemnetz) $\Sigma = (N, \mathcal{U}, \mathcal{A}, C_I)$ ist ein Systemnetz, gdw:

i. N ist ein Netz;

ii. $\mathcal{U}$ ist ein Universum für N;

iii. $\mathcal{A}$ weist jeder Transition $t \in T_N$ eine Menge von Aktionen $\mathcal{A}(t)$ zu; sowie

iv. C_I ist ein Zustand von N, genannt *Initial-* oder *Anfangszustand*.

2.2 Abläufe eines Systemnetzes

Das dynamische Verhalten eines Systemnetzes Σ ist durch die Menge der *Abläufe von* Σ gegeben. Wir unterscheiden *Interleavings* und *Halbordnungsabläufe*. Die Menge der Interleavings eines Systemnetzes korrespondiert mit der klassischen operationellen Semantik paralleler Programme, die in der Literatur über selbststabilisierende Algorithmen ausnahmslos zugrunde gelegt wird.

Ein Interleaving beschreibt eine vom Anfangszustand aus mögliche Sequenz des Eintretens von Aktionen.

Definition 2.20 (Interleaving) Für ein Systemnetz $\Sigma = (N, \mathcal{U}, \mathcal{A}, C_I)$ nennen wir eine endliche Sequenz von Schritten $\big((C_0, (t_0, m_0), C_1), (C_1, (t_1, m_1), C_2), \ldots, (C_{n-1}, (t_{n-1}, m_{n-1}), C_n)\big)$ ein *(endliches) Interleaving*, gdw. $C_0 = C_I$. Eine unendliche Sequenz von Schritten $\big((C_0, (t_0, m_0), C_1), \ldots\big)$ nennen wir ein *(unendliches) Interleaving*, gdw. $C_0 = C_I$.

Definition 2.21 (Erreichbarer Zustand) Ein Zustand C eines Systemnetzes Σ heißt *erreichbar*, gdw. es ein endliches Interleaving $\big((C_0, (t_0, m_0), C_1), (C_1, (t_1, m_1), C_2), \ldots, (C_{n-1}, (t_{n-1}, m_{n-1}), C_n)\big)$ von Σ gibt, mit $C = C_n$.

In der Literatur hat sich bisher keine einheitliche Semantik für verteilte Systeme durchgesetzt. Wir verwenden neben der Interleavingsemantik auch eine Halbordnungssemantik. In einer Halbordnung werden die *kausalen* Abhängigkeiten von Aktionen dargestellt. Wir fassen die Vorteile der Halbordnungssemantik (die sich im Rahmen dieser Arbeit ergeben) zusammen:

1. Halbordnungen drücken die Nebenläufigkeit von Aktionen adäquat aus. D. h. genau die Aktionen, die voneinander abhängig sind, werden auch als voneinander abhängig dargestellt.

2. Die Halbordnungssemantik stellt unmittelbar eine graphische Beschreibung eines Ablaufs zur Verfügung, anhand derer Aussagen über die Lokalität von Zuständen getroffen werden können. So stellen sich beispielsweise lokale Zustandsübergänge eines Agenten oder der Weg eines Tokens durch ein Netzwerk unmittelbar als Linien in einem Halbordnungsablauf dar.

3. Halbordnungen helfen bei der Analyse von Algorithmen. Das von Walter in [Wal95] formalisierte Konzept *asynchroner Runden* ermöglicht Korrektheitsbeweise, die im Interleavingmodell in dieser Klarheit und Einfachheit oft nicht möglich sind.

Weitere Vorteile ergeben sich auch bei der Komplexitätsanalyse: Oft wird die Zeitkomplexität eines Algorithmus in Abhängigkeit von der Anzahl von Runden angegeben [PSL97, Dol00]. Eine Runde bedeutet hierbei informell, daß jeder Agent eine definierte Anzahl atomarer Aktionen ausgeführt und sich genau einmal mit seinen Nachbarn synchronisiert hat. In einem Interleaving eines verteilten Systems sind diese Runden jedoch im allgemeinen ineinander verschränkt. Die Komplexitätsanalyse betrachtet daher repräsentative Interleavings, in denen die Runden nicht ineinander verschränkt sind. Dies ergibt den zusätzlichen Aufwand nachzuweisen, daß ein bestimmtes Interleaving tatsächlich repräsentativ ist. Implizit wird hierzu die kausale Halbordnung der Aktionen eines Interleavings betrachtet. Halbordnungsabläufe benötigen diesen Umweg nicht, da die einzelnen Runden explizit dargestellt werden.

Definition 2.22 (Minimale und maximale Stellen)

Die Menge der *minimalen Stellen* $^\circ N \subseteq P_N$ sowie die Menge der *maximalen Stellen* $N^\circ \subseteq P_N$ eines Netzes N sind definiert durch $^\circ N = \{p \in P_N : {}^\bullet p = \emptyset\}$ sowie $N^\circ = \{p \in P_N : p^\bullet = \emptyset\}$.

Kausalnetze sind spezielle azyklische Netze, in denen alle Stellen unverzweigt sind. Somit können keine Konflikte zwischen zwei verschiedenen Transitionen auftreten, vgl. Def. 2.17.

Definition 2.23 (Kausalnetz) Ein Netz $K = (B, E, \lessdot)$ ist ein *Kausalnetz*, gdw.:

i. $^\circ K$ ist endlich;

ii. $\forall b \in B : {}^\bullet b \leqslant 1$ und $b^\bullet \leqslant 1$;

iii. Die transitive Hülle von $\lessdot$, die wir im folgenden mit $<$ bezeichnen, ist azyklisch, d. h. $\forall x_0, \ldots, x_n \in B \cup E : (x_0 \lessdot x_1, \ldots, x_{n-1} \lessdot x_n) \Rightarrow x_0 \neq x_n$;

iv. $\forall x \in B \cup E : \{y \in B \cup E : y < x\}$ ist endlich.

Wir nennen die Stellen eines Kausalnetzes *Bedingungen* und die Transitionen eines Kausalnetzes *Ereignisse*.

Beispiel 2.24 In Abbildung 2.3 ist ein Kausalnetz dargestellt.

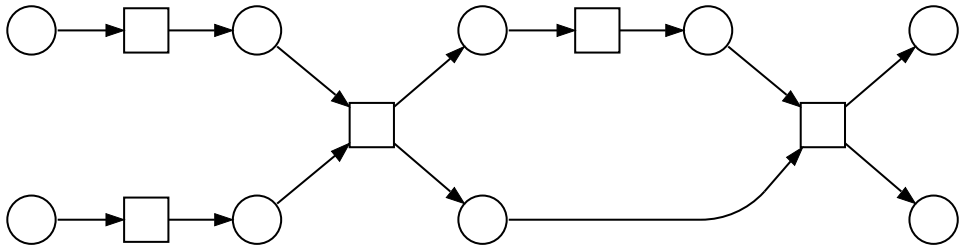


Abbildung 2.3: Ein Kausalnetz

Das Eintreten einer Aktion eines Systemnetzes Σ wird als ein beschriftetes Kausalnetz [SR87] (mit genau einem Ereignis) repräsentiert:

Definition 2.25 (Aktionsnetz) Seien $\Sigma = (N, \mathcal{U}, \mathcal{A}, C_I)$ ein Systemnetz mit einem Universum $\mathcal{U}$ und $(t, m) \in \mathcal{A}$ eine Aktion von Σ.

Ein Kausalnetz $K = (B, E, \lessdot)$ zusammen mit einer Beschriftungsfunktion $l : B \times E \longrightarrow \mathcal{A} \cup (P_N \times \mathcal{U})$ ist ein *Aktionsnetz* für (t, m), gdw.:

i. $E = \{e\}$;

ii. $B = {}^\bullet e \cup e^\bullet$;

iii. $l(e) = (t, m)$;

iv. $\forall b \in B : l(b) \in P_N \times \mathcal{U}$ sowie

v. $\forall (p, a) \in P_N \times \mathcal{U} : l_{\mathcal{M}}({}^\bullet e)[(p, a)] = m(p, t)[a]$ und $l_{\mathcal{M}}(e^\bullet)[(p, a)] = m(t, p)[a]$.

Notation 2.26 Wir schreiben eine Beschriftung $(p, a) \in P_N \times \mathcal{U}$ einer Bedingung im folgenden (insbesondere in Abbildungen) als $p.a$.

Beispiel 2.27 In Abbildung 2.4 ist ein Aktionsnetz für die Aktion (t, m_0) aus Beispiel 2.14 dargestellt.

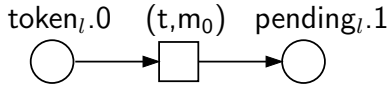


Abbildung 2.4: Ein Aktionsnetz

Ein Halbordnungsablauf ist ein beschriftetes Kausalnetz, in dem die einzelnen Ereignisse zusammen mit ihrer jeweiligen Umgebung Aktionsnetze bilden.

Definition 2.28 (Halbordnungsablauf) Sei $\Sigma = (N, \mathcal{U}, \mathcal{A}, C_I)$ ein Systemnetz. Ein Kausalnetz $K = (B, E, \lessdot)$ zusammen mit einer Beschriftungsfunktion l heißt *Halbordnungsablauf von* Σ, gdw:

i. $\forall (p, a) \in P_N \times \mathcal{U} : l_{\mathcal{M}}(^\circ K)[p.a] = C_I(p)[a]$ sowie

ii. $\forall e \in E \, \exists (t, m) \in \mathcal{A} : \big({}^\bullet e \cup e^\bullet, \{e\}, ({}^\bullet e \times \{e\}) \cup (\{e\} \times e^\bullet)\big)$ ist ein Aktionsnetz für (t, m).

Beispiel 2.29 In graphischen Darstellungen stellen wir die Beschriftung eines Ereignisses nicht explizit dar. In Abbildung 2.5 ist ein Halbordnungsablauf für unser einführendes Beispiel aus Abschnitt 1.3 angegeben.

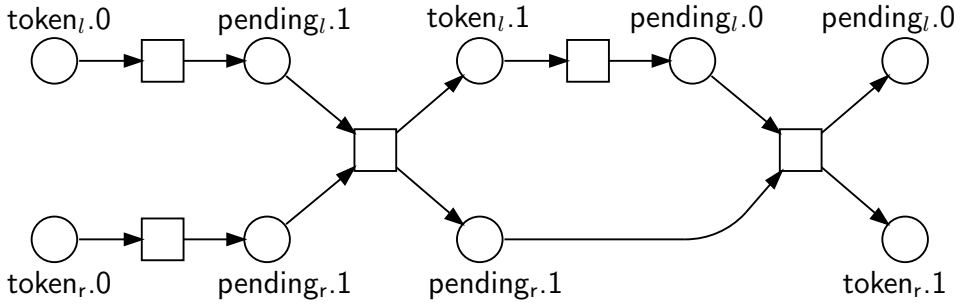


Abbildung 2.5: Ein Halbordnungsablauf

Interleavings und Halbordnungsabläufe hängen eng zusammen: Jeder Halbordnungsablauf charakterisiert eine Menge von Interleavings; jedes Interleaving wird durch mindestens eine Halbordnung charakterisiert. In vielen Fällen gilt sogar, daß für jedes Interleaving *genau* eine Halbordnung existiert (beispielsweise bei Systemen, in denen in keinem erreichbaren Zustand zwei gleiche Marken auf derselben Stelle liegen [Rei98]). Wir werden dies in Satz 2.35 präzisieren: Jede Totalordnung der Ereignisse eines Halbordnungsablaufs repräsentiert ein Interleaving.

Definition 2.30 (Nebenläufigkeit, Schnitt) Wir definieren für ein Kausalnetz $K = (B, E, \lessdot)$:

i. Zwei Elemente $x, y \in B \cup E$ sind *nebenläufig*, gdw. weder $x < y$ noch $y < x$ gilt.

ii. $S \subseteq B$ heißt *Schnitt*, gdw. S eine Menge paarweise nebenläufiger Bedingungen ist und S maximal ist, d. h. es keine Bedingung $b' \in B \setminus S$ gibt, so daß b' nebenläufig zu allen $b \in S$ ist.

Lemma 2.31 Für ein Kausalnetz K ist ${}^\circ K$ ein Schnitt.

Beweis: folgt direkt aus Def. 2.30 (Schnitt), Def. 2.22 (minimale Elemente) und Notation 2.5 (Stellen-berandete Netze). □

Definition 2.32 (Schritt eines Kausalnetzes) Seien S ein Schnitt eines Kausalnetzes $K = (B, E, \lessdot)$ und $e \in E$ ein Ereignis, mit ${}^\bullet e \subseteq S$. Dann ist $(S,\ e,\ (S \setminus {}^\bullet e) \cup e^\bullet)$ ein *Schritt in* K.

Lemma 2.33 Sei (S, e, S') ein Schritt eines Kausalnetzes K. Dann ist S' ein Schnitt von K.

Beweis: Der Beweis folgt aus Def. 2.30 (Schnitt) und Def. 2.23 (Kausalnetz). □

Definition 2.34 (Erreichbarer Schnitt) Wir definieren induktiv die Menge der *erreichbaren* Schnitte für ein Kausalnetz $K = (B, E, \lessdot)$:

i. ${}^\circ K$ ist ein erreichbarer Schnitt.

ii. Seien S ein erreichbarer Schnitt und (S, e, S') ein Schritt in K. Dann ist S' ein erreichbarer Schnitt.

Wir geben nun die beiden wichtigen Korrespondenzsätze an, die Schrittsequenzen eines Halbordnungsablaufs und erreichbare Zustände korrelieren. Jede Totalordnung der Ereignisse eines Halbordnungsablaufs ergibt ein Interleaving, und jedes Interleaving wird durch einen Halbordnungsablauf repräsentiert.

Satz 2.35 Sei $K = (B, E, \lessdot)$ zusammen mit der Beschriftungsfunktion l ein Halbordnungsablauf eines Systemnetzes Σ. Sei $(e_0, e_1, \dots)$ eine (möglicherweise endliche) Totalordnung aller Ereignisse von E, d. h. für jedes Ereignis $e \in E$ existiert genau ein e_i, so daß $e = e_i$. Weiterhin respektiere die Totalordnung die Halbordnung von K, d. h.: $\forall e_i, e_j : (e_i < e_j) \Rightarrow (i \leqslant j)$.

Dann existieren Zustände $C_0, C_1, \dots$, so daß die Sequenz $\big((C_0,\ l(e_0),\ C_1),\ (C_1,\ l(e_1),\ C_2),\ \dots\big)$ ein Interleaving von Σ ist.

Satz 2.36 Für ein Interleaving $\big((C_0,(t_0,m_0),C_1),\dots\big)$ eines Systemnetzes $\Sigma=(N,\mathcal{U},\mathcal{A},C_I)$ existieren ein Halbordnungsablauf (K,l) und eine Sequenz von Schritten $\big((S_0,e_0,S_1),\dots\big)$ von K, so daß für alle i, mit $0 \leqslant i$, gilt: $\forall(p,a)\in P_N\times\mathcal{U}: C_i(p)[a]=l_{\mathcal{M}}(S_i)[p.a]$ und $l(e_i)=(t_i,m_i)$.

Die Beweise der beiden Sätze geben Best und Fernández in [BF88] an. Dort sind insbesondere Beispiele angegeben, die die Notwendigkeit aller Voraussetzungen bei unserer Definition von Kausalnetzen und Halbordnungsabläufen begründen.

2.2.1 Fortschritt und Fairneß

Fortschritts- und Fairneßannahmen schränken die Menge der Abläufe eines Systemnetzes ein [Bes95]. Wir werden im folgenden die für uns relevanten Fortschritts- und Fairneßannahmen definieren. Die Begriffe Fortschritt und Fairneß werden in der Literatur stark vermischt. Fortschritt bezieht sich bei uns im wesentlichen darauf, daß wir das Verhalten lokaler Teile eines Systems nicht durch die Addition *unabhängiger* Komponenten beeinflussen können.

Jeder Algorithmus hat implizit eine *Fortschrittsannahme*. Die Fortschrittsannahme besagt intuitiv, daß ein lokaler Agent so lange weiterarbeitet, bis er keine Aktion mehr ausführen kann. Ausnahmen hiervon bilden beispielsweise fehlertolerante Systeme, in denen *Crash-Fehler* auftreten können, nach denen Agenten keine Aktion mehr ausführen.[1] Wir nehmen Fortschritt für alle unsere Algorithmen an. Wir werden im folgenden die Fortschrittsannahme für Interleavings formalisieren: Ein Interleaving verletzt die Fortschrittsannahme, falls es ein Suffix gibt, in dem eine Aktion immer aktiviert ist und keine in Konflikt stehende Transition eintritt.

Definition 2.37 (Fortschritt) Sei $\Sigma=(N,\mathcal{U},\mathcal{A},C_I)$ ein Systemnetz.

i. Ein endliches Interleaving $\big((C_0,\,(t_0,m_0),\,C_1),\,\dots,\,(C_{n-1},\,(t_{n-1},m_{n-1}),\,C_n)\big)$ von Σ *respektiert Fortschritt*, gdw. für jede Aktion $(t,m)\in\mathcal{A}$ gilt: (t,m) ist im Zustand C_n nicht aktiviert.

ii. Ein unendliches Interleaving $R=\big((C_0,\,(t_0,m_0),\,C_1),\,\dots\big)$ von Σ *respektiert Fortschritt*, gdw. für jede Aktion $(t,m)\in\mathcal{A}$ und jedes Suffix $\big((C_i,\,(t_i,m_i),\,C_{i+1}),\,\dots\big)$ von R, in dem (t,m) in jedem Zustand C_j mit $i\leqslant j$ aktiviert ist, ein $k\geqslant i$ existiert, so daß (t_k,m_k) in Konflikt zu (t,m) steht.

[1] Ausführlich wird diese Art von Fehlern im Zusammenhang mit Petrinetzen in [Völ00] untersucht.

Beispiel 2.38 Wir erläutern diese Definition an einem technischen Beispiel in Abbildung 2.6. Der Domain aller Stellen besteht aus der Menge $\{a\}$ und jede

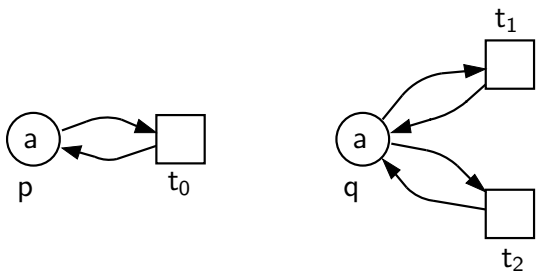


Abbildung 2.6: Ein Systemnetz

Transition t hat einen Modus, der allen Umgebungskanten die Multimenge $[a]$ zuweist.

In diesem Systemnetz gilt:

1. Ein endliches Interleaving verletzt die Fortschrittsannahme.

2. Ein unendliches Interleaving, in dem $\mathtt{t_0}$ nur endlich oft auftritt, verletzt die Fortschrittsannahme.

3. Ein unendliches Interleaving, in dem $\mathtt{t_1}$ und $\mathtt{t_2}$ nur endlich oft auftreten, verletzt die Fortschrittsannahme.

Wir betrachten im folgenden nur noch Abläufe, die die Fortschrittsannahme respektieren. Fortschrittsannahmen allein genügen manchmal nicht, um das Verhalten eines verteilten Systems adäquat zu beschreiben. Viele Systeme nehmen deshalb verschiedene Arten von *Fairneß* an. Es gibt grundsätzliche Probleme, deren Lösung sogar eine Fairneßannahme erfordert: Kindler und Walter zeigen in [KW97] (siehe auch [Wal95]), daß *wechselseitiger Ausschluß (mutual exclusion)*, nur unter Fairneßannahmen realisierbar ist. Wie bereits in Abschnitt 1.2.1 beschrieben, bedarf die faire Komposition selbststabilisierender Algorithmen ebenfalls einer Fairneßannahme. Wir werden in Abschnitt 3.2 ein anderes Problem vorstellen, den selbststabilisierenden asynchronen Tokenaustausch, das ebenfalls Fairneß benötigt.

In der Literatur üblich ist die Annahme *schwacher Fairneß (weak fairness)* [LPS81]. Intuitiv verletzt ein Interleaving die (schwache) Fairneßannahme, falls es eine Aktion gibt, die in einem Suffix andauernd aktiviert ist, jedoch ignoriert wird.

Definition 2.39 (Fairneß eines Interleavings) Sei $\Sigma = (N, \mathcal{U}, \mathcal{A}, C_I)$ ein Systemnetz.

i. Ein endliches Interleaving von Σ *respektiert Fairneß*.

ii. Ein unendliches Interleaving $\big((C_0,\ (t_0, m_0),\ C_1),\ \ldots\big)$ von Σ *respektiert Fairneß*, gdw. für jede Aktion $(t, m) \in \mathcal{A}$ und jedes Suffix $\big((C_i,\ (t_i, m_i),\ C_{i+1}),\ \ldots\big)$ von R, in dem (t, m) in jedem Zustand C_j mit $i \leqslant j$ aktiviert ist, ein $k \geqslant i$ mit $(t_k, m_k) = (t, m)$ existiert.

Beispiel 2.40 Wir beziehen uns wiederum auf das technische Beispiel in Abbildung 2.6.

1. Ein Interleaving, in dem unendlich oft $\mathtt{t_1}$ aber nur endlich oft $\mathtt{t_2}$ auftritt, verletzt die Fairneßannahme.

2. Ein Interleaving, in dem unendlich oft $\mathtt{t_2}$ aber nur endlich oft $\mathtt{t_1}$ auftritt, verletzt die Fairneßannahme.

In Definition 2.39(ii) haben wir Fairneß pauschal für alle Aktionen gefordert. Dies ist sinnvoll, weil ein selbststabilisierenden Algorithmus im allgemeinen nie in Isolation abläuft, sondern als eine fehlertolerante Schicht in einem System, konkret will man Algorithmen also komponieren. Dafür muß man jedoch Fairneß für jede Aktion fordern (vergleiche Abschnitt 1.2.1).

Wenn wir jedoch einen selbststabilisierenden Algorithmus in Isolation betrachten, dann gibt es aber Transitionen, für deren Aktionen wir explizit Fairneß fordern müssen, um die Korrektheit des Algorithmus zu sichern. Wir werden aber bei der syntaktischen Darstellung in (siehe Abschnitt 2.3) die betreffenden Transitionen besonders kennzeichnen.

2.2.2 Randomisierung

Mit DAWN können bisher noch keine randomisierten verteilten Algorithmen modelliert und verifiziert werden. Wir integrieren daher in diesem Abschnitt das Konzept von Randomisierung formal in den DAWN-Rahmen. Randomisierung an sich ist ein bekanntes Konzept: Randomisierung kann als *eine spezielle Ausprägung von Fairneß* aufgefaßt werden [Rao90]. Unsere Definition ist ein Spezialfall von *extremer Fairneß* [Pnu83] und *α-Fairneß* [Zuc86]. Pnueli zeigte in [Pnu83], daß eine Eigenschaft, die in allen extrem fairen Abläufen eines Systems gilt, auch mit Wahrscheinlichkeit 1 im System gilt. Wir werden in diesem Abschnitt formulieren, was eine *Randomisierungsannahme* für ein System ist und wann ein Interleaving *randomisiert* ist.

Eine *Randomisierungsannahme* legt zwei disjunkte Mengen von Aktionen fest. Diese Mengen korrespondieren mit Münzwürfen mit den Resultaten *Kopf* oder *Zahl* (bzw. 0 oder 1). Wir fordern, daß immer, wenn es möglich ist, eine 0 zu werfen, dann ist es auch möglich, eine 1 zu werfen. Wir weisen den einzelnen Aktionen und Aktionsmengen keine festen Wahrscheinlichkeiten ihres Eintretens zu, aus zwei Gründen:

1. Die *Korrektheit* eines Algorithmus sollte nicht von konkreten Wahrscheinlichkeiten (z. B. $0,5$ zu $0,5$) abhängen, da die Implementierung einer „perfekten Münze“ praktisch unmöglich ist (siehe hierzu auch die Diskussion in [Rao90]).

2. Die *Komplexität* eines Algorithmus (also die erwartete Anzahl der Aktionen bis zur Stabilisierung) hängt oft nicht von konkreten Wahrscheinlichkeitsverteilungen ab. Dies ist insbesondere bei unseren Algorithmen der Fall.

Definition 2.41 (Randomisierungsannahme) Für ein Systemnetz $\Sigma = (N, \mathcal{U}, \mathcal{A}, C_I)$ seien $A, A' \subseteq \mathcal{A}$, so daß $\mathcal{R} = (A, A')$ ein Paar disjunkter Mengen von Aktionen von Σ ist. $\mathcal{R}$ ist eine Randomisierungsannahme für Σ, gdw. für jeden erreichbaren Zustand C von Σ gilt:

i. Für jede in C aktivierte Aktion $(t, m) \in A$ gibt es eine in C aktivierte Aktion $(t', m') \in A'$.

ii. Für jede in C aktivierte Aktion $(t', m') \in A'$ gibt es eine in C aktivierte Aktion $(t, m) \in A$.

iii. Wenn zwei Aktionen $(t, m), (t', m') \in A \cup A'$ in C aktiviert sind, dann stehen (t, m) und (t', m') im Konflikt.

Die ersten beiden Forderungen obiger Definition bedeuten informell, daß immer wenn es möglich ist, die eine Seite der Münze zu werfen, dann ist es auch möglich, die andere Seite der Münze zu werfen. Die dritte Forderung schließt aus, daß (innerhalb einer Randomisierungsannahme) zwei oder mehr Münzen unabhängig von einander geworfen werden: Es gibt immer eine Ressource, die zwei Münzwurfaktionen gemeinsam benötigen (vergleiche Definition 2.17). Der Sinn dieser Forderung spiegelt sich in einem Beispiel aus [Rao90] wider:

Beispiel 2.42 Zwei Münzen werden unabhängig voneinander von zwei Agenten immer wieder geworfen. In diesem verteilten System gilt nicht die Eigenschaft, daß mit Wahrscheinlichkeit 1 einen Zustand erreicht wird, in dem beide

Münzen 1 zeigen: Es gibt einen Ablauf, in dem beide Agenten mit der Münzseite 0 starten. Nun wirft der eine Agent solange seine Münze, bis die Münze eine 0 zeigt. Danach wirft der andere Agent solange seine Münze, bis auch er eine 0 geworfen hat. Danach beginnt der erste Agent wieder, usw. In diesem Ablauf gilt die beschriebene Eigenschaft nicht, da immer mindestens eine Münze 0 zeigt. Beide Agenten haben sich an ihr Protokoll gehalten, und keiner von ihnen hat beim Werfen „geschummelt". Daher betrachten wir in einer Randomisierungsannahme immer genau eine Münze.

In der nächsten Definition beschreiben wir, wann auf einen Zustand eines Ablaufs eine Sequenz von Münzwürfen folgt.

Definition 2.43 (Sequenz von Münzwürfen) Sei Σ ein Systemnetz mit einer Randomisierungsannahme $\mathcal{R} = (A, A')$. Sei C ein Zustand eines unendlichen Interleavings $R = \big((C_0,\ (t_0, m_0),\ C_1),\ \dots\big)$, in dem unendlich viele Münzwurfaktionen vorkommen. Auf C *folgt die Sequenz von Münzwürfen* $S = (S_0, S_1, \dots, S_{n-1})$, $S_i \in \{A, A'\}$, gdw. für die nächsten n in R auf C folgenden Münzwurfaktionen $(t'_i, m'_i) \in A \cup A'$, $0 \leqslant i < n$, gilt $(t'_i, m'_i) \in S_i$.

Die Sequenz S ist eine endliche Sequenz von Nullen und Einsen und enthält die Ergebnisse von Münzwürfen.

Definition 2.44 (Randomisierter Ablauf) Ein Interleaving R eines Systemnetzes Σ mit einer Randomisierungsannahme $\mathcal{R}$ ist ein *randomisierter Ablauf*, gdw. eine der folgenden Bedingungen zutrifft:

i. In R treten nur endlich oft Aktionen aus $\mathcal{R}$ auf.

ii. Für jeden Zustand C, der in R unendlich oft vorkommt, und jede endliche Sequenz von Münzwürfen S gilt, daß C unendlich oft von S gefolgt wird.

Wir erläutern und begründen im folgenden die einzelnen Bestandteile dieser Definition. In einem randomisierten Ablauf sind die Ergebnisse eines Münzwurfs unabhängig vom Algorithmus. So ließe eine Definition, in der wir lediglich fordern, daß jede endliche Sequenz von Münzwürfen immer wieder auftritt, einen Ablauf zu, in dem ein Agent in einem bestimmten, immer wiederkehrenden Zustand immer wieder die Münze mit demselben Ergebnis wirft. In diesem Fall ist also das Ergebnis des Münzwurfes in einem bestimmten Zustand nicht unabhängig vom Algorithmus.

Beispiel 2.45 Wir modifizieren in Abbildung 2.7 das System aus Beispiel 2.6.

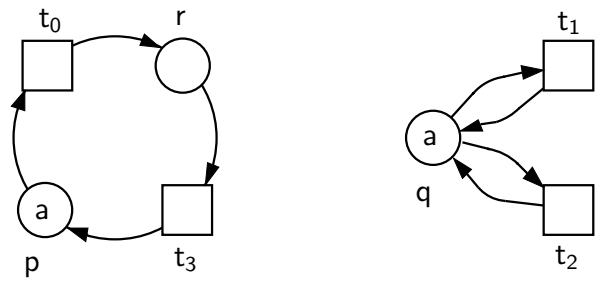


Abbildung 2.7: Ein Systemnetz

Wir fixieren eine Randomisierungsannahme, bestehend aus der (einzigen) Aktion von t_1 als 0-Münzwurf und der Aktion von t_2 als 1-Münzwurf. Die eindeutige Marke a auf der Stelle q interpretieren wir als Münze und das Eintreten von t_1 oder t_2 als einen Münzwurf.

In diesem Systemnetz gilt:

1. Ein Interleaving, in dem t_1 nur dann eintritt, falls die Stelle p mit a markiert ist, ist kein randomisierter Ablauf (da der Zustand, in dem p und q markiert sind, unendlich oft vorkommt, aber nie von der Sequenz (0) gefolgt wird).

2. Ein Interleaving, in dem t_1 und t_2 abwechselnd eintreten, ist kein randomisierter Ablauf (da die Sequenz $(0, 0)$ nie eintritt).

Unsere Definition ist nicht vollständig, d. h. es gibt Eigenschaften, die mit Wahrscheinlichkeit 1 im System gelten, die jedoch nicht in allen randomisierten Abläufen gelten. Dies trifft jedoch nur auf Systeme zu, deren Zustandsraum unendlich ist (siehe [Zuc86]). Wir haben hier eine Definition gewählt, die die Verifikation unserer randomisierten selbststabilisierenden Algorithmen erleichtert.

2.3 Termdarstellung eines Systemnetzes

In diesem Abschnitt legen wir die Syntax der Systemnetze fest. Mit unserer Syntax beschreiben wir mögliche Aktionen einer Transition durch *Terme* an den Umgebungskanten. Wir benutzen im wesentlichen Konzepte algebraischer Spezifikationen [EM85], erweitert um Multimengen-Strukturen. Wir orientieren die Definitionen an [Rei99, WWV+97]. Zuerst definieren wir *Strukturen*.

Definition 2.46 (Sorte) Seien $A_0, \ldots, A_n$ Mengen.

i. Für ein $0 \leqslant i \leqslant n$ sei $a \in A_i$. Dann heißt a eine *Konstante in den Mengen* $A_0, \ldots, A_n$ *der Sorte* A_i.

ii. Für $i = 0, \ldots, k$ sei $B_i \in \{A_0, \ldots, A_n\}$ und $f : B_0 \times \cdots \times B_{k-1} \longrightarrow B_k$ eine Funktion.

Dann ist f eine *Funktion über* $A_0, \ldots, A_n$. $B_0, \ldots, B_{k-1}$ sind die *Argumentsorten*, B_k die *Zielsorte*, $(B_0, \ldots, B_k)$ die *Stelligkeit* (geschrieben $B_0 \times \cdots \times B_{k-1} \longrightarrow B_k$) *von* f.

Definition 2.47 (Struktur) Seien $A_0, \ldots, A_n$ Mengen, $a_0, \ldots, a_m$ Konstanten in $A_0, \ldots, A_n$ und $f_0, \ldots, f_l$ Funktionen über $A_0, \ldots, A_n$.

Dann ist $\mathcal{S} = (A_0, \ldots, A_n; a_0, \ldots, a_m; f_0, \ldots, f_l)$ eine *Struktur*. $A_0, \ldots, A_n$ sind die *Träger*, $a_0, \ldots, a_m$ die *Konstanten* und $f_0, \ldots, f_l$ die *Funktionen* von $\mathcal{S}$.

Wir nehmen im folgenden an, daß jede Struktur den ausgezeichneten Träger $\mathbb{B} = \{0, 1\}$ hat.

Definition 2.48 (Multimengen-Struktur) Eine Struktur $\mathcal{S} = (A_0, \ldots, A_{2n-1}; a_0, \ldots, a_m; f_0, \ldots, f_l)$ ist eine *Multimengen-Struktur*, falls für $0 \leqslant i < n$ gilt $A_{n+i} = \mathcal{M}(A_i)$. Dann heißen $A_0, \ldots, A_{n-1}$ die *Basis-Träger* und $A_n, \ldots, A_{2n-1}$ die *Multimengen-Träger von* $\mathcal{S}$.

Beispiel 2.49 Wir betrachten die Multimengen-Struktur $\mathcal{S}_{\mathbb{B}} = (\mathbb{B}, \mathcal{M}(\mathbb{B}); 0, 1, [\,]; \neg, \vee, [\cdot])$. Hierbei ist:

1. $\mathbb{B}$ der Basis-Träger,
2. $\mathcal{M}(\mathbb{B})$ der zugehörige Multimengen-Träger,
3. $0, 1$ Konstanten der Sorte $\mathbb{B}$,
4. $[\,]$ eine Konstante der Sorte $\mathcal{M}(\mathbb{B})$,
5. $\neg$ die Negation mit der Stelligkeit $\mathbb{B} \longrightarrow \mathbb{B}$,
6. $\vee$ die Disjunktion mit der Stelligkeit $\mathbb{B} \times \mathbb{B} \longrightarrow \mathbb{B}$,
7. $[\cdot]$ die (implizite) Funktion, die aus einem Booleschen Wert seine kanonische Multimenge erzeugt.

Aus einer Struktur und Variablen können wir *Terme* aufbauen. Durch Einsetzen konkreter Werte für die Variablen können wir Terme *auswerten*.

Definition 2.50 (Term) Sei $\mathcal{S} = (A_0, \ldots, A_n; a_0, \ldots, a_m; f_0, \ldots, f_l)$ eine Struktur.

i. Seien $X_0, \ldots, X_n$ paarweise disjunkte Mengen von Symbolen. A_i ist die Sorte von $x \in X_i$. $X = (X_0, \ldots, X_n)$ ist eine *Familie $\mathcal{S}$-sortierter Variablen*.

ii. Sei X eine Familie $\mathcal{S}$-sortierter Variablen. Wir definieren die Menge $\mathcal{T}_B(X)$ der *Terme der Sorte B über X* für alle $B \in \{A_0, \ldots, A_n\}$ induktiv:

   a) $X_i \subseteq \mathcal{T}_{A_i}(X)$;

   b) Für alle $a \in \{a_0, \ldots, a_m\}$ mit der Sorte B ist $a \in \mathcal{T}_B(X)$;

   c) Für eine Funktion $f \in \{f_0, \ldots, f_l\}$ mit der Stelligkeit $B_0 \times \cdots \times B_k \longrightarrow B$ und $u_j \in \mathcal{T}_{B_j}(X)$ für $0 \leqslant j \leqslant k$, ist $(f, u_0, \ldots, u_k) \in \mathcal{T}_B(X)$.

iii. $\mathcal{T}_{\mathcal{S}}(X) = \mathcal{T}_{A_0}(X) \cup \cdots \cup \mathcal{T}_{A_n}(X)$ ist die Menge der *$\mathcal{S}$-Terme über X*.

In obiger Definition benutzen wir die Tupelschreibweise für Terme, um Mißverständnisse auszuschließen. So soll beispielsweise für die Konstante 0 und die Negations-Funktion $\neg$ in den Booleschen Werten „$\neg 0$“ einen zusammengesetzten Term und nicht die Konstante 1 bezeichnen. In der Schreibweise $(\neg, 0)$ ist der Unterschied zu 1 deutlich. Dieser Unterschied ist jedoch nur bei der Definition der Auswertung von Termen erheblich; wir werden ansonsten die übliche Schreibweise $f(u_0, \ldots, u_k)$ für einen Term $(f, u_0, \ldots, u_k)$ benutzen.

Definition 2.51 (Variablenbelegung, Auswertung) Sei $\mathcal{S} = (A_0, \ldots, A_n; a_0, \ldots, a_m; f_0, \ldots, f_l)$ eine Struktur und $X = (X_0, \ldots, X_n)$ eine Familie $\mathcal{S}$-sortierter Variablen.

i. Eine Abbildung $\beta : \bigcup_{0 \leqslant i \leqslant n} X_i \longrightarrow \bigcup_{0 \leqslant i \leqslant n} A_i$ heißt *Variablenbelegung*, gdw. $\forall i \in \{0, \ldots, n\} \, \forall x_i \in X_i : \beta(x_i) \in A_i$.

ii. Sei β eine Variablenbelegung. Die Funktion $\beta_{\mathcal{T}} : \mathcal{T}_{\mathcal{S}}(X) \longrightarrow \bigcup_{0 \leqslant i \leqslant n} A_i$ heißt *Auswertung eines Terms* und ist induktiv über den Aufbau der Terme definiert:

a) $\forall x \in \bigcup_{0 \leqslant i \leqslant n} X_i : \beta_{\mathcal{T}}(x) = \beta(x)$,

b) $\forall a \in \{a_0, \ldots, a_m\} : \beta_{\mathcal{T}}(a) = a$,

c) $\forall f \in \{f_0, \ldots, f_l\}\ \forall 0 \leqslant i \leqslant k\ \forall u_i \in \mathcal{T}_{A_i}(X) : \beta_{\mathcal{T}}(f, u_0, \ldots, u_k) = f(\beta_{\mathcal{T}}(u_0), \ldots, \beta_{\mathcal{T}}(u_k))$.

Beispiel 2.52 Die Menge $\{x, 0, \neg 1, [1, 1], [\neg x]\}$ ist eine Menge von $\mathcal{S}_{\mathbb{B}}$-Termen über der Familie von Variablen $X = (\{x\})$, wobei die Sorte der Variable x der Träger $\mathbb{B}$ ist. Durch eine Variablenbelegung β, die der Variable x den Wert 1 zuweist, entstehen nach der Auswertung der Terme folgende Elemente von $\mathbb{B}$ bzw. $\mathcal{M}(\mathbb{B})$: $1, 0, 0, [1, 1], [0]$.

Definition 2.53 (Termdarstellung eines Netzes) $\Sigma_{\mathcal{T}}$ ist eine *Termdarstellung eines Netzes N über der Multimengen-Struktur* $\mathcal{S} = (A_0, \ldots, A_n; a_0, \ldots, a_m; f_0, \ldots, f_l)$, gdw.:

i. Jeder Stelle $p \in P_N$ ist ein Basis-Träger A_p von $\mathcal{S}$ zugeordnet.

ii. Jeder Stelle $p \in P_N$ ist ein Multimengen-Term $u_I(p) \in \mathcal{T}_{\mathcal{M}(A_p)}(\emptyset)$ zugeordnet.

iii. Jeder Transition $t \in T_N$ ist eine Familie X_t $\mathcal{S}$-sortierter Variablen, die *Variablen von t* zugeordnet.

iv. Jeder Transition $t \in T_N$ ist ein Term $g_t \in \mathcal{T}_{\mathbb{B}}(X_t)$, die *Schaltbedingung von t*, zugeordnet.

v. Jedem $f = (p, t) \in P_N \times T_N$ oder $f = (t, p) \in T_N \times P_N$ ist ein Multimengen-Term m_f zugeordnet, mit $m_f \in \mathcal{T}_{\mathcal{M}(A_p)}(X_t)$ und $m_f = [\,]$ gdw. $f \notin F_N$.

Wir lassen nur Multimengen an Kanten $f \in F_N$ zu, die in keiner Variablenbelegung zur leeren Multimenge ausgewertet werden.

Wir definieren exemplarisch die Termdarstellung des Systemnetzes aus Abbildung 1.5:

Beispiel 2.54 Wir definieren eine Termdarstellung $\Sigma_{\mathcal{T}}$ des Netzes $N_{\Sigma_1} = (P_{\Sigma_1}, T_{\Sigma_1}, F_{\Sigma_1})$ über der Multimengenstruktur $\mathcal{S}_{\mathbb{B}}$ (vergleiche Beispiel 2.49):

1. Wir ordnen jeder Stelle $p \in P_{\Sigma_1}$ den Basis-Träger $\mathbb{B}$ zu.

2. Den Stellen token_l und token_r wird jeweils der Multimengen-Term [0] zugeordnet; den Stellen $\mathsf{pending}_l$ und $\mathsf{pending}_r$ wird jeweils der Multimengen-Term [ ] zugeordnet.

3. Wir ordnen jeder Transition $t \in T_{\Sigma_1}$ die einelementige Menge von Variablen $\{x\}$ zu.

4. Die Schaltbedingung jeder Transition ist durch den Term *true* gegeben.

5. Wir weisen jeder Kante $f \in F_{\Sigma_1}$ einen der Multimengen-Terme $m_f = [x]$ oder $m_f = [\neg x]$ zu, siehe Abbildung 1.5.

Notation 2.55 In Abbildungen werden wir nicht die vollständige Termdarstellung angeben. Wir führen daher die folgenden Notationen ein:

i. Wir geben den Basis-Träger einer Stelle nicht explizit an. Er ergibt sich immer implizit aus der Beschriftung der die Stelle umgebenden Kanten. Beispielsweise ist die Sorte der Multimengen-Terme m_f der die Stelle $\mathsf{pending}_l$ umgebenden Kanten jeweils $\mathcal{M}(\mathbb{B})$, daher ist $\mathbb{B}$ der Basis-Träger der Stelle $\mathsf{pending}_l$.

ii. Falls nicht explizit anders angegeben, nehmen wir für jede Transition $t \in T$ die Schaltbedingung $g_t = \mathit{true}$ an. In diesem Fall geben wir die Schaltbedingung nicht explizit an. Anderenfalls stellen wir die Schaltbedingung in Abbildungen in der Transition dar.

iii. Die Menge der Variablen X_t einer Transition $t \in T$ ergibt sich durch die Variablen, die in den Termen an den Umgebungskanten von t benutzt werden.

iv. Wir geben die Multimengen-Struktur ebenfalls nicht explizit an. Wir definieren lediglich bei Bedarf neue Sorten (außer den Standardsorten $\mathbb{B}$ und $\mathbb{N}$), Konstanten und Funktionen.

v. Wir geben die Sorten aller benutzten Variablen explizit an.

Definition 2.56 (Aktion einer Termdarstellung) Sei $\Sigma_{\mathcal{T}}$ eine Termdarstellung eines Netzes N über der Multimengen-Struktur $\mathcal{S}$. Seien $t \in T_N$ und β eine Variablenbelegung.

(t, m) ist eine *Aktion einer Transition t in der Termdarstellung* $\Sigma_{\mathcal{T}}$, gdw.:

i. für jede Kante $f \in F_N$ gilt: $m(f) = \beta_{\mathcal{T}}(m_f)$ und

ii. $\beta_{\mathcal{T}}(g_t) = \mathit{true}$.

Definition 2.57 (Systemnetz einer Termdarstellung) Sei $\Sigma_{\mathcal{T}}$ eine Termdarstellung eines Netzes N über der Multimengen-Struktur $\mathcal{S}$. Das *Systemnetz* $\Sigma = (N, \mathcal{U}, \mathcal{A}, C_I)$ von $\Sigma_{\mathcal{T}}$ ist definiert durch:

i. das Universum $\mathcal{U}$, das jeder Stelle $p \in P_N$ den durch $\Sigma_{\mathcal{T}}$ gegebenen Basis-Träger von $\mathcal{S}$ zuordnet,

ii. die Menge $\mathcal{A}$ der Aktionen aller Transitionen $t \in T_N$ in $\Sigma_{\mathcal{T}}$ und

iii. den Anfangszustand C_I, der durch $C_I(p) = \beta_{\mathcal{T}}(u_I(p))$, für eine beliebige Variablenbelegung β, gegeben ist.

Zum Abschluß dieses Abschnitts legen wir noch die schon beschriebene graphische Konvention fest, um Fairneß für einzelne Transitionen zu fordern: Wir stellen eine eingehende Kante mit weißer Pfeilspitze dar. Diese Konvention ist aus [WWV+97] übernommen. In [Rei98] werden faire Transitionen durch eine spezielle Anschrift in der Transition dargestellt; dies kollidiert jedoch mit der Darstellung von Schaltbedingungen.

2.4 Eigenschaften eines Systemnetzes

In diesem Abschnitt führen wir die temporale Logik ein, mit der wir Eigenschaften eines verteilten Systems formal spezifizieren. In unserem einführenden Beispiel in Abschnitt 1.3 erwähnten wir bereits, daß wir Namen von Stellen in Formeln verwenden. Sie dienen als Variablen, die in einem Zustand für die Multimenge von Marken auf dieser Stelle stehen. Wir definieren zunächst die Syntax.

Definition 2.58 (Systemeigenschaften) Sei N ein Netz und $X = (X_0, \ldots, X_n)$ eine Familie $\mathcal{S}$-sortierter Variablen, mit $X_i \cap P_N = \emptyset$, für $0 \leqslant i \leqslant n$. Wir interpretieren jedes $p \in P_N$ mit dem Basis-Träger A_p als eine neue Variable der Sorte $\mathcal{M}(A_p)$ und definieren die Menge $\mathcal{SF}(X)$ der *Systemeigenschaften über* X induktiv:

i. $\mathcal{T}_{\mathbb{B}}(X_0, \ldots, X_n, P_N) \subseteq \mathcal{SF}(X)$,

ii. für $\varphi \in \mathcal{SF}(X)$ ist $\neg\varphi \in \mathcal{SF}(X)$,

iii. für $\varphi, \psi \in \mathcal{SF}(X)$ ist $(\varphi \vee \psi) \in \mathcal{SF}(X)$,

iv. für $\varphi \in \mathcal{SF}(X)$ und $x \in X_i$ mit der Sorte A ist $(\exists\, x \in A : \varphi) \in \mathcal{SF}(X)$,

v. für $\varphi \in \mathcal{SF}(X)$ ist $\Diamond\, \varphi \in \mathcal{SF}(X)$ und

vi. *lockout* $\in \mathcal{SF}(X)$.

Notation 2.59

i. Wir verwenden die üblichen Schreibabkürzungen $\wedge, \Rightarrow, \Leftrightarrow$ und $\forall x \in A$.

ii. Eine Systemeigenschaft, in der kein $\Diamond$ (gesprochen „irgendwann“) vorkommt, nennen wir auch *Zustandseigenschaft*.

iii. $\Box\, \varphi$ („immer φ“) steht für $\neg \Diamond \neg \varphi$.

iv. $\varphi \mapsto \psi$ („φ leads-to ψ“) steht für $\Box(\varphi \Rightarrow \Diamond\, \psi)$.

v. Für eine Stelle $\mathsf{p} \in P_N$ und einen Multimengen-Term u steht $\mathsf{p}(u)$ für $u \leqslant \mathsf{p}$. Also ist $\mathsf{p}(u)$ ein Term der Sorte $\mathcal{T}_{\mathbb{B}}(X_0, \ldots, X_n, P_N)$.

vi. Für eine endliche Multimenge A und jede natürliche Zahl n sind die Ausdrücke $|A| \leqslant n, |A| > n$ und $|A| = n$ ebenfalls Boolesche Terme.

Beispiel 2.60 Die folgenden Formeln sind Systemeigenschaften des Systemnetzes Σ_1 aus Abschnitt 1.3:

$$\mathit{true} \mapsto (|\mathsf{token}_l| > 0) \tag{2.1}$$

$$\Diamond\, \Box(|\mathsf{token}_l + \mathsf{token}_\mathsf{r}| \leqslant 1) \tag{2.2}$$

Wir erweitern nun die Definition der Termauswertung (siehe Def. 2.51), um Stellen-Variablen auswerten zu können.

Definition 2.61 (Auswertung von Stellen-Variablen) Für eine Variablenbelegung $\beta : \bigcup_{0 \leqslant i \leqslant n} X_i \longrightarrow A$ und einen Zustand $C : P \longrightarrow A'$ definieren wir eine erweiterte Variablenbelegung $\beta_C : X \cup P \longrightarrow A \cup A'$ wie folgt:

i. Für $x \in X$ gilt $\beta_C(x) = \beta(x)$, sowie

ii. für $p \in P$ gilt $\beta_C(p) = C(p)$.

Wie in Definition 2.51 erweitern wir kanonisch über die Struktur der Terme β_C zu einer Auswertung $\beta_{\mathcal{T},C} : \mathcal{T}_{\mathcal{S}}(X_0, \ldots, X_n, P) \longrightarrow A \cup A'$.

Beispiel 2.62 In einem Zustand, der der Stelle token_l die Multimenge $[0]$ zuordnet, wird der Term $|\mathsf{token}_l| > 0$ für jede Variablenbelegung zu *true* ausgewertet (da $|[0]| = 1$).

Definition 2.63 (Semantik von Systemeigenschaften) Für ein (endliches oder unendliches) Interleaving $R = \big((C_0, (t_0, m_0), C_1), \dots\big)$ eines Systemnetzes $\Sigma = (N, \mathcal{U}, \mathcal{A}, C_I)$ und einen Zustand C_i von R definieren wir die *Gültigkeit einer Systemeigenschaft* $\varphi \in \mathcal{SF}(X)$ unter einer Belegung β induktiv:

i. Für jedes $u \in \mathcal{T}_{\mathbb{B}}(X_0, \dots, X_n, P_N)$ gilt $(C_i, \beta) \models u$, gdw. $\beta_{\mathcal{T}, C_i}(u) = \mathit{true}$,

ii. $(C_i, \beta) \models \neg\varphi$, gdw. $(C_i, \beta) \models \varphi$ nicht gilt,

iii. $(C_i, \beta) \models (\varphi \vee \psi)$, gdw. $(C_i, \beta) \models \varphi$ oder $(C_i, \beta) \models \psi$ gilt,

iv. $(C_i, \beta) \models (\exists\, x \in A : \varphi)$, gdw. eine Belegung β' existiert, so daß $(C_i, \beta') \models \varphi$ gilt und für alle $y \in \bigcup\limits_{0 \leqslant i \leqslant n} X_i \setminus \{x\}$ gilt $\beta'(y) = \beta(y)$,

v. $(C_i, \beta) \models \Diamond\, \varphi$, gdw. es einen Zustand C_j in R gibt, mit $i \leqslant j$ und $(C_j, \beta) \models \varphi$ und

vi. $(C_i, \beta) \models \mathit{lockout}$, gdw. in C_i keine Aktion $(t, m) \in \mathcal{A}$ aktiviert ist.

Eine Systemeigenschaft $\varphi \in \mathcal{SF}(X)$ gilt im Zustand C, geschrieben $C \models \varphi$, gdw. für jede Variablenbelegung β gilt: $(C, \beta) \models \varphi$.

φ gilt in R, geschrieben $R \models \varphi$, gdw. $C_0 \models \varphi$ gilt.

φ gilt in Σ, geschrieben $\Sigma \models \varphi$ gdw. sie in jedem Fortschritt und Fairneß respektierenden Interleaving von Σ gilt.

φ gilt unter einer Randomisierungsannahme $\mathcal{R}$, geschrieben $\Sigma \models_{\mathcal{R}} \varphi$, gdw. sie in jedem Fortschritt und Fairneß respektierenden randomisierten Interleaving gilt.

Informell können wir die Gültigkeit unserer temporalen Operatoren wie folgt beschreiben: Eine Eigenschaft $\Diamond\, \varphi$ gilt in einem System Σ, wenn in jedem Ablauf von Σ irgendwann ein Zustand erreicht wird, in dem φ gilt. Eine Eigenschaft $\Box\, \varphi$ gilt in Σ, wenn in jedem erreichbaren Zustand von Σ die Eigenschaft φ gilt. Eine Eigenschaft $\varphi \mapsto \psi$ gilt in Σ, wenn in jedem Ablauf von Σ auf jeden Zustand, in dem φ gilt, irgendwann ein Zustand folgt, in dem ψ gilt. Eine Eigenschaft $\Diamond\, \Box\, \varphi$ gilt in Σ, wenn jeder Ablauf von Σ ein Suffix hat, in dem in jedem Zustand φ gilt.

Wir haben in Definition 2.58 das Auftreten freier Variablen in Systemeigenschaften zugelassen. In Definition 2.63 interpretieren wir diese Variablen als implizit allquantifiziert.

Die von uns gewählte Semantik $\Sigma \models_{\mathcal{R}} \varphi$ unterscheidet sich von anderen Ansätzen aus der Literatur. Rao führt beispielsweise in [Rao90, Rao95] für eine randomisierte Variante des Operators leads-to ein. Wir führen keinen neuen Operator ein, schränken aber die Menge der betrachteten Interleavings ein. In Beweisen randomisierter Algorithmen werden zuerst Eigenschaften gezeigt, die von einer Randomisierungsannahme unabhängig sind. Durch die beiden Formen der Gültigkeit ($\Sigma \models \varphi$ und $\Sigma \models_{\mathcal{R}} \varphi$) können wir in Beweisen einfach betonen, welche Eigenschaften nur unter einer Randomisierungsannahme gelten.

2.5 Netzschemata

Mit einem Systemnetz können wir *ein* verteiltes System mit *einem* Anfangszustand modellieren. Wir werden in diesem Abschnitt beschreiben, wie wir *selbststabilisierende Netzwerkalgorithmen* modellieren. Konkret werden wir mit Hilfe von *Netzschemata* zwei Probleme lösen:

1. Ein selbststabilisierender Algorithmus ist korrekt, wenn er sein algorithmisches Ziel von *jedem* Anfangszustand aus erreicht. In einem Modell eines selbststabilisierenden Algorithmus möchten wir daher auch *alle* Anfangszustände repräsentieren.

2. Viele verteilte Algorithmen sind *Netzwerkalgorithmen*. Ein Netzwerkalgorithmus funktioniert für eine *Klasse* von Systemen, d. h. er legt kein konkretes verteiltes System fest. Beispielsweise gibt es Algorithmen für Ringe von Agenten. Diese funktionieren für jedes verteilte System, in dem das zugrundeliegende Agentennetzwerk eine Ringstruktur hat. Wir möchten daher Klassen verteilter Systeme in unserem Modell repräsentieren.

Ein *Netzschema* ist ein unvollständig spezifiziertes Systemnetz. Zur Modellierung eines Netzwerkalgorithmus spezifizieren wir das zugrundeliegende Agentennetzwerk unvollständig (beispielsweise legen wir die Anzahl der Agenten nicht fest). Zur Modellierung selbststabilisierender Algorithmen spezifizieren wir den Anfangszustand unvollständig, vergleiche das folgende Beispiel aus Abschnitt 1.3:

Beispiel 2.64 In Abbildung 2.8 ist ein Netzschema dargestellt. In Abweichung von der bisherigen Definition eines Systemnetzes haben wir hier den Anfangszustand nicht vollständig spezifiziert: Wir beschreiben den Anfangszustand durch die Parameter P_l, T_l, P_r, T_r sowie zwei Axiome, die die Wahl der Werte

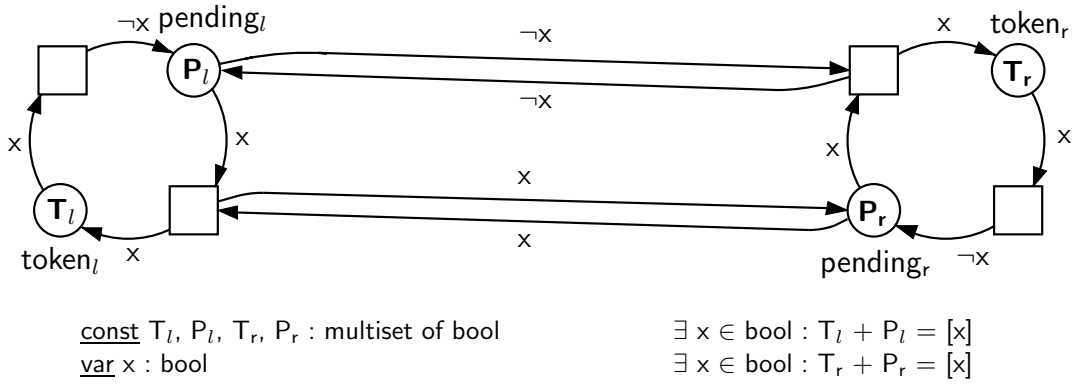


Abbildung 2.8: Σ_2 – Tokenaustausch

für diese Parameter einschränken. Im Algorithmus von Dijkstra (vergleiche Abschnitt 1.3 hat jeder Agent zwei Bits. Also gibt es 16 mögliche Anfangszustände. Jede durch die Axiome zugelassene Belegung der Parameter repräsentiert einen dieser Anfangszustände.

In Korrektheitsbeweisen werden wir formal mathematisch ein verteiltes System und einen Anfangszustand betrachten. Wir werden jedoch nur die explizit spezifizierten Eigenschaften der dem Systemnetz zugrundeliegenden Struktur verwenden. Die Korrektheit des Algorithmus für alle verteilten Systeme mit diesen Eigenschaften und für jeden Anfangszustand folgt dann aus der Beliebigkeit des gewählten konkreten Systems. Netzschemata wurden bisher nur durch die Modellierung von Netzwerkalgorithmen motiviert (siehe [KR96]). Nicht vollständig spezifizierte Anfangszustände wurden bereits in [Ves96, Rei98] zur Modellierung selbststabilisierender Algorithmen verwendet.

Wir spezifizieren zum Abschluß dieses Kapitels die wesentlichen Eigenschaften der unseren selbststabilisierenden Netzwerkalgorithmen zugrundeliegenden Agentennetzwerke. Unsere Notation lehnt sich hierbei an (parametrisierte) algebraische Spezifikationen ([EM85, LEW96]) mit der Deklaration von *Sorten*, *Konstanten*, *Funktionen*[2] und *Axiomen* an. Zur Formulierung der Axiome benutzen wir die Logik-orientierte Schreibweise mit Quantoren ([EGL89]). Wir setzen Standardspezifikationen mit den üblichen Funktionen voraus (bool, nat, finite set, siehe [EM85]). Wir formalisieren explizit die Konstanten und Funktionen, die wir in unseren Petrinetzmodellen benutzen.

Zuerst definieren wir einen *unidirektionalen Ring*. Ein einfaches Beispiel eines unidirektionalen Rings ist in Abbildung 2.9 angegeben. Die Knoten be-

[2]Funktionen werden in der Literatur häufig auch *Operatoren* genannt.

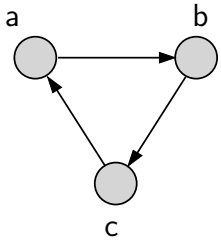


Abbildung 2.9: Ein unidirektionaler Ring

zeichnen die Elemente einer Menge U von Agenten, die Kanten bezeichnen die mögliche Kommunikation. Die Art der Kommunikation (Nachrichtenaustausch oder gemeinsamen Speicher) haben wir nicht festgelegt. So bedeutet unidirektionale Kommunikation bei Nachrichtenaustausch, daß beispielsweise Agent a an Agent b Nachrichten senden kann, aber nicht umgekehrt. Bei gemeinsamem Speicher bedeutet unidirektional, daß beispielsweise Agent b lokale Werte des Agenten a lesen kann, aber nicht umgekehrt.

Für eine Funktion l und eine natürliche Zahl n bezeichnet l^n die n-malige Hintereinanderausführung von l, d. h. beispielsweise ist $l^3(x) = l(l(l(x))$.

Definition 2.65 (Unidirektionaler Ring) Ein *unidirektionaler Ring* (U, l) genügt folgender Spezifikation:

spec unidirectional ring =
  sort site
  fct l : site $\longrightarrow$ site
  const U : finite set of sites
  $\forall$ x $\in$ U $\exists$ y $\in$ U : l(x) = y
  $\forall$ x, y $\in$ U $\exists$ n $\in$ nat : l^{n}(x) = y

Nun definieren wir eine *unidirektionale Kette*. Ein einfaches Beispiel einer unidirektionalen Kette ist in Abbildung 2.10 angegeben.

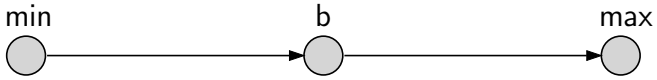


Abbildung 2.10: Eine unidirektionale Kette

Definition 2.66 (Unidirektionale Kette) (U, r, min, max) ist eine *unidirektionale Kette*, gdw. (U, r, min, max) folgender Spezifikation genügt:

spec unidirectional chain =
sort site
fct r : site $\longrightarrow$ site
const U : finite set of sites
const min, max : site
$\exists$ x $\in$ U : x = min
$\exists$ x $\in$ U : x = max
$\forall$ x $\in$ U $\exists$ y $\in$ U : r(x) = y
r(max) = max
$\forall$ x $\in$ U $\exists$ n $\in$ nat : r^n(min) = x

Zum Abschluß definieren wir eine *bidirektionale Kette*. Ein einfaches Beispiel einer bidirektionalen Kette ist in Abbildung 2.11 angegeben.

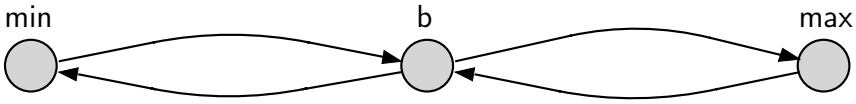


Abbildung 2.11: Eine bidirektionale Kette

Definition 2.67 (Bidirektionale Kette) (U, r, l, min, max) ist eine *bidirektionale Kette*, gdw. (U, r, l, min, max) folgender Spezifikation genügt:

spec bidirectional chain =
sort site
fct r : site $\longrightarrow$ site
fct l : site $\longrightarrow$ site
const U : finite set of sites
const min, max : site
$\exists$ x $\in$ U : x = min
$\exists$ x $\in$ U : x = max
$\forall$ x $\in$ U $\exists$ y $\in$ U : r(x) = y
r(max) = max
$\forall$ x $\in$ U $\exists$ n $\in$ nat : r^n(min) = x
l(min) = min
$\forall$ x $\in$ U $\setminus$ {min} : x = r(l(x))

3 Fallstudien

Wir werden in diesem Kapitel Petrinetzmodelle verteilter selbststabilisierender Algorithmen entwickeln. Wir gliedern dieses Kapitel in drei Abschnitte, die man verschiedenen Richtungen innerhalb der aktuellen Forschung über selbststabilisierende Algorithmen zuordnen kann. In Abschnitt 3.1 stellen wir randomisierte selbststabilisierende Algorithmen vor. In Abschnitt 3.2 betrachten wir asynchrone Systeme. Beide Systemklassen wurden erst in den letzten zehn Jahren im Zusammenhang mit Selbststabilisierung untersucht. In Abschnitt 3.3 stellen wir Beispiele eher klassischer selbststabilisierender Algorithmen für Systeme mit gemeinsamem Speicher vor.

Die Darstellung innerhalb dieser Abschnitte folgt einem Schema. Wir stellen zunächst das Umfeld der betrachteten Algorithmen und den Stand der Forschung dar. Danach legen wir uns jeweils auf ein konkretes Systemmodell und ein algorithmisches Ziel fest. Bei den Algorithmen selbst betonen wir die ihnen zugrunde liegenden algorithmischen Ideen. Wir werden zur Illustration die Pseudocode-Beschreibungen der Quellen angeben. Im Mittelpunkt stehen Petrinetzmodelle. Wir beschreiben die Datenstrukturen und Operationen auf den Datenstrukturen. Wir geben typische Halbordnungsabläufe an, wobei wir das Verhalten eines konkreten Systems im *stabilen* Fall darstellen. Aus diesem Verhalten lassen sich manchmal Verbesserungen eines Algorithmus ableiten (Abschnitt 3.1 und Abschnitt 3.3.2). Wir schließen jede Beschreibung eines Algorithmus mit der Spezifikation seiner entscheidenden Eigenschaften ab. Hierbei unterscheiden wir *Sicherheits-* und *Lebendigkeitseigenschaften* (wie von Alpern und Schneider in [AS85] definiert). Intuitiv wird diese Trennung oft so beschrieben: Eine Sicherheitseigenschaft garantiert, daß nie etwas Schlechtes passiert; eine Lebendigkeitseigenschaft garantiert, daß irgendwann etwas Gutes passiert. Diese Unterscheidung hat sich bei der Beschreibung verteilter Systeme als sehr geeignet herausgestellt; einen umfassenden Überblick gibt Kindler in [Kin94].

3.1 Randomisierte Algorithmen

Dijkstra bemerkte in [Dij74], daß für die Entwicklung seines ersten selbststabilisierenden Algorithmus die Erkenntnis entscheidend war, daß nicht alle Agenten *dasselbe Protokoll* haben können. Dieses Phänomen hat ursächlich nichts mit Selbststabilisierung zu tun; vielmehr bewies Angluin in [Ang80], daß in symmetrischen Netzwerken (z. B. einem Ring) für eine bestimmte Problemklasse kein deterministischer uniformer Algorithmus existiert. Ein Algorithmus ist *uniform*, falls alle Agenten nach demselben Protokoll arbeiten. Ein Algorithmus ist *deterministisch*, falls es für jeden erreichbaren Zustand eines Agenten höchstens einen durch eine lokale Aktion dieses Agenten erreichbaren Zustand gibt.

Ein Beispiel eines Problems, zu dessen Lösung kein uniformer deterministischer Algorithmus existiert, ist *Tokenaustausch* (vergleiche Abschnitt 1.3). Ein Token ist eine ausgezeichnete (d. h. von jeder anderen unterscheidbare) Nachricht, die zwischen den Agenten herum gereicht wird. Wir demonstrieren die Unmöglichkeit intuitiv an einem kleinen Beispiel:

> Zwei Agenten sollen deterministischen uniformen Tokenaustausch realisieren. Beide Agenten starten in demselben Zustand (also insbesondere auch mit der gleichen Anzahl an Token). Angenommen, beide Agenten führen zur gleichen Zeit die gleichen Aktionen aus. Dann wird es immer eine gerade Anzahl von Token im System geben, weil die Agenten entweder beide Nachrichten versenden oder beide Nachrichten empfangen. Also können sie keinen Tokenaustausch realisieren.

Bei der Lösung solcher Probleme hilft *Randomisierung*. Sie basiert auf der Annahme, daß mindestens ein Agent eine Münze werfen kann. Der erste randomisierte sequentielle Algorithmus wurde 1976 von Michael Rabin vorgestellt ([Rab76]). Verteilte randomisierte Algorithmen wurden ab dem Beginn der 80er Jahre vorgestellt (z. B. [FR80, LR81]). In [GSB94] findet sich ein umfangreicher Überblick. Es gab hierbei im Vergleich zu sequentiellen randomisierten Algorithmen einen bedeutenden Paradigmenwechsel: Sequentielle randomisierte Algorithmen lösen Probleme schneller oder mit weniger Speicher; verteilte randomisierte Algorithmen lösen Probleme, für die es gar keine deterministische Lösung gibt, wie oben beschrieben.

Israeli und Jalfon [IJ90] veröffentlichten den ersten randomisierten selbststabilisierenden Algorithmus. Sie entwickelten ein Schema, das Tokenaustausch in einem beliebigen zusammenhängenden Agentennetzwerk realisiert. Sie waren außerdem die ersten, die die algorithmische Idee von *Random Walk* ausnutzten, um Selbststabilisierung zu erreichen. Die Idee von Random Walk ist abstrakt die folgende: Viele Token laufen in zufällige Richtungen; immer wenn sich zwei Token treffen, wird eines von ihnen vernichtet. In endlichen zusammenhängenden Netzwerken, wie z. B. einem Ring, ist gesichert, daß mit Wahrscheinlichkeit 1 irgendwann nur noch ein Token zirkuliert ([IJ90, AKL$^+$79, Fel68]).

Weitere randomisierte Algorithmen folgten schnell: Israeli und Jalfon präsentierten in [IJ93] einen Algorithmus zur *Orientierung im Ring*: Jeder Agent im Ring hat zwei Nachbarn. Jeder Agent soll einen dieser Nachbarn als seinen „linken“ Nachbarn auszeichnen, derart daß global durch die Funktionen „linker Nachbar von x“ für jeden Agenten x ein unidirektionaler Ring (vergleiche Abschnitt 2.5) definiert wird.

Herman [Her92] sowie Flatebo und Datta [FD94] griffen einen Vorteil randomisierter sequentieller Algorithmen auf: lokalen Speicherplatz einzusparen. Sie gaben randomisierte nicht-uniforme selbststabilisierende Algorithmen zum Tokenaustausch in unidirektionalen Ringen an. Der Algorithmus von Flatebo und Datta benötigt die Information, ob der Ring aus einer geraden oder ungeraden Anzahl von Agenten besteht. Beide Algorithmen benötigen nur konstanten Speicherplatz (zwei bzw. ein Bit) pro Agent, was zu damaliger Zeit einen entscheidenden Durchbruch darstellte, da alle damals bekannten Algorithmen (z. B. der Algorithmus von Dijkstra [Dij74]) pro Agent wenigstens Speicher benötigten, der linear von der Anzahl der Agenten im Netzwerk abhing. (Später wurde dieser Durchbruch allerdings relativiert, da in [GH96] ein deterministischer nicht-uniformer Algorithmus angegeben wurde, der dasselbe Problem ebenfalls mit konstantem Speicher, drei Bits pro Agent, löst.)

Herman [Her90] betrachtete randomisierte Selbststabilisierung als erster in einer *synchronen* Umgebung. In einer synchronen Umgebung führen alle Agenten parallel eine atomare Aktion aus und synchronisieren sich vor der nächsten Ausführung einer Aktion. Diese Einschränkung mag sehr restriktiv erscheinen; es gibt aber mindestens zwei gute Gründe hierfür: Erstens wurde in [BGJ99] gezeigt, daß kein randomisierter selbststabilisierender *Leader Election-Algorithmus* (siehe Abschnitt 3.1.1) für uniforme unidirektionale *asynchrone* Ringe existiert. Es läßt sich einfach zeigen, daß daher auch kein randomisierter selbststabilisierender Tokenaustausch-Algorithmus existiert (da Leader Election auf Tokenaustausch zurückgeführt werden kann; wir stellen mehrere solcher Algorithmen in diesem Abschnitt vor). Zweitens gibt es Anwendungen synchro-

ner unidirektionaler Ringe in Hochgeschwindigkeits-Netzwerken: Ein FDDI-Ring *(Fiber Distributed Data Interconnect)* ist ein auf Glasfaserkabeln basierendes Hochgeschwindigkeitsnetz, das Daten mit vielfacher Geschwindigkeit eines Ethernet-Netzes überträgt (100 Millionen Bits pro Sekunde) [Com97]. Durch die hohe Geschwindigkeit der Datenübertragung arbeitet es praktisch (die lokalen Uhren weichen höchstens zu $0,005$ Prozent voneinander ab) synchron [MOY96]. Ein FDDI-Ring besteht aus zwei unidirektionalen Ringen, die gegenläufig geschaltet sind. Im Normalzustand wird nur einer dieser Ringe benutzt, fällt ein Agent aus, so wird der andere Ring auch benutzt, um den Ring zwischen den verbliebenen Agenten wieder zu schließen.

Weitere randomisierte Algorithmen [MOOY92, MOY96] arbeiten ebenfalls auf synchronen Ringen. In [MOOY92] wird ein Algorithmus zum Tokenaustausch in bidirektionalen Ringen vorgestellt. Auf diesen Algorithmus wird ein Algorithmus für Leader Election aufgebaut. Mayer, Ostrovsky und Yung schlagen in [MOY96] einen „Protokoll-Compiler“ vor, der jeden beliebigen Algorithmus für bidirektionale Ringe selbststabilisierend auf unidirektionalen Ringen ausführt.

Wir werden im folgenden Abschnitt zeigen, daß der von Mayer, Ostrovsky und Yung vorgeschlagene Algorithmus in seinem wesentlichen Teil, einem selbststabilisierenden Leader Election-Algorithmus, inkorrekt ist. Diese Erkenntnis ist neu. Konkret werden wir im folgenden Abschnitt 3.1.1 drei randomisierte Leader Election-Algorithmen für unidirektionale Ringe vorstellen: Der erste Algorithmus ist der von Mayer, Ostrovsky und Yung als ein wesentlicher Teil des Protokoll-Compilers in [MOY96] publizierte. Der zweite Algorithmus stellt eine veränderte Version des ersten Algorithmus dar, die die ursprüngliche algorithmische Idee von Mayer, Ostrovsky und Yung repräsentiert [Yun99]. Der dritte Algorithmus ist ein neuer Algorithmus, der so effizient wie möglich mit dem Speicher der Agenten arbeitet. Im Abschnitt 5.1 werden wir die Korrektheit des dritten Algorithmus und dessen Speicherminimalität beweisen.

3.1.1 Leader Election

Leader Election ist eine der fundamentalen Aufgaben verteilten Rechnens. Viele verteilte Algorithmen setzen die Existenz eines ausgezeichneten Agenten (eines „Leaders“) voraus, der ein Protokoll abarbeitet, das sich von den Protokollen der anderen Agenten („Follower“) unterscheidet. Wenn in einem verteilten System kein solcher Agent von vornherein existiert, dann ermöglicht die Wahl eines Leaders die nachfolgende Ausführung eines derartigen Algorithmus.

Der Algorithmus von Mayer, Ostrovsky und Yung

Der folgende Algorithmus wurde von Mayer, Ostrovsky und Yung in [MOY96] veröffentlicht. Er arbeitet auf einem synchronen unidirektionalen Ring und baut auf einem selbststabilisierenden Tokenaustausch auf. Er setzt also voraus, daß genau ein Token existiert, das im Ring zirkuliert. Wir nennen dieses Token im folgenden *Münze*. Nach der Beschreibung des Algorithmus von Mayer, Ostrovsky und Yung werden wir zeigen, daß dieser Algorithmus nicht selbststabilisierend ist. Hierzu geben wir ein Netzwerk und einen Anfangszustand an, für den der Algorithmus keinen eindeutigen Leader wählt.

Sei (U, l) ein unidirektionaler Ring. Jeder Agent $x \in U$ hat drei Bits: ein Zustandsbit ($x.state$), welches angibt, ob ein Agent ein *Leader* oder ein *Follower* ist, sowie zwei weitere Bits ($x.f$ und $x.l$), die wir *Follower-Bit* bzw. *Leader-Bit* nennen. Es gibt genau eine Münze *coin* im System, die initial bei einem Agenten $x \in U$ ist. Die Münze kann zwischen den Agenten versendet (send coin) und empfangen (receive coin) werden. Sie hat zwei Bits, die wir mit *coin.f* und *coin.l* bezeichnen. Wir nennen sie *Follower-Bit* bzw. *Leader-Bit der Münze*. Zusätzlich zu den üblichen Aktionen kann ein Agent eine Münze werfen, d. h. einer Variable wird zufällig einer der Werte 0 oder 1 zugewiesen. Hierfür führen wir die folgende Anweisung ein:

$$a := \textit{coin-toss;}$$

Hierbei steht a für eine beliebige Variable.

Jeder Agent $x \in U$ arbeitet nach dem in Abbildung 3.1 angegebenen Protokoll [MOY96]. Die einmalige Abarbeitung des Protokolls ist eine atomare Aktion. Die Semantik des Algorithmus ist durch die Menge aller aus atomaren Aktionen aufgebauten Interleavings gegeben. Dementsprechend übernehmen wir die Semantik von Systemeigenschaften direkt von Abschnitt 2.4.

Wir zeigen im folgenden, daß dieser Algorithmus keine selbststabilisierende Leader Election garantiert. Konkret gibt es einen unidirektionalen Ring, in dem dieser Algorithmus keinen Leader wählt. Hierzu fixieren wir zunächst ein System Σ_F, d. h. ein konkretes Netzwerk und einen Anfangszustand:

Wir betrachten den unidirektionalen Ring (U, l), mit $U = \{u_0, u_1, u_2\}$ und $l(u_i) = u_{(i+1) \bmod 3}, 0 \leqslant i < 3$. Wir betrachten einen Anfangszustand, in dem die Follower-Bits der Agenten und der Münze wie folgt verteilt sind: $u_0.f = u_1.f = 1$, $u_2.f = 0$ sowie $coin.f = 1$. Die Münze sei initial bei u_1. Die anderen Werte sind beliebig verteilt.

Definition 3.1 (Rechter Nachbar) Für einen Agenten $u \in U$ eines Ringes (U, l) nennen wir einen Agenten $v \in U$ den *rechten Nachbarn* von u, genau dann wenn $l(v) = u$ gilt.

```
receive coin;
if (x.state = follower)
      if (coin.f = x.f)
            coin.f := ¬coin.f; x.state := leader; x.l := coin.l;
else
      if (coin.f = x.f)
            coin.f := ¬coin.f;
x.f := coin.f;
if (x.state = leader)
      if (coin.l ≠ x.l)
            x.state := follower;
      else
            coin.l := coin-toss; x.l := coin.l;
send coin to l(x);
```

Abbildung 3.1: Protokoll für einen Agenten $x \in U$

Da l eine umkehrbar eindeutige Funktion ist (folgt aus Definition 2.65), ist der rechte Nachbar eines Agenten eindeutig. Wir bezeichnen den rechten Nachbarn eines Agenten $x \in U$ mit $r(x)$. Weiterhin drücken wir mit $coin(x)$ aus, daß Agent x die Münze hat. Beispielsweise gilt im Anfangszustand $coin(u_1)$.

Wir werden zwei wichtige, im System Σ_F invariante Eigenschaften zeigen:

1. Wenn ein Agent x die Münze hat, dann gleicht das Follower-Bit der Münze dem Follower-Bit des rechten Nachbarn von x;

2. Wenn ein Agent x die Münze hat, dann gleicht das Follower-Bit der Münze dem Follower-Bit von x selbst.

In Abbildung 3.2 ist eine typische Situation für Σ_F veranschaulicht. Hier sind die Werte der Follower-Bits symbolisch mit Hilfe einer booleschen Variable i dargestellt.

Jeder Agent, der die Münze hat, kopiert das Follower-Bit der Münze, bevor er die Münze weitersendet:

Lemma 3.2 $\Sigma_F \models \forall x \in U : \Box(coin(x) \Rightarrow coin.f = r(x).f)$.

Beweis: Initial gilt die Behauptung, da die Münze bei u_1 ist und $coin.f = u_0.f = 1$. In jeder Aktion eines beliebigen Agenten $x \in U$ kopiert x das Follower-Bit der Münze beim Ausführen der Anweisung $x.f := coin.f;$. Danach werden bis zum Versenden der Münze Follower-Bits nicht mehr geändert. Also gilt die Behauptung auch nach dem Versenden der Münze. □

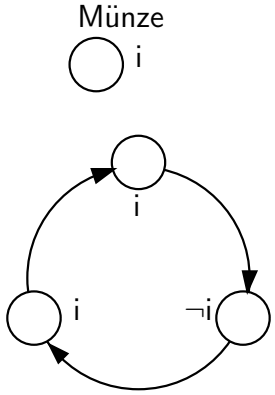


Abbildung 3.2: Eine Invariante des Systems Σ_F

Wir betrachten nun die Eigenschaft:

$$\forall x \in U : x.f = r(x).f \Leftrightarrow coin(x) \tag{3.1}$$

Sie impliziert, daß wenn ein Agent x die Münze hat, dann gleicht das Follower-Bit der Münze dem Follower-Bit von x selbst. Sie ist ebenfalls eine Invariante des Systems:

Lemma 3.3 $\Sigma_F \models \forall x \in U : \Box(x.f = r(x).f \Leftrightarrow coin(x))$.

Beweis: (3.1) gilt offensichtlich im Anfangszustand des Systems. Nun zeigen wir, daß jede Aktion eines Agenten diese Eigenschaft erhält. Wir nehmen also an, daß (3.1) für einen Agenten $u \in U$ gilt und daß u die Münze besitzt. Es gilt also:

$$coin(u) \wedge u.f = r(u).f \wedge u.f \neq l(u).f$$

Weiterhin gilt wegen Lemma 3.2:

$$coin.f = u.f$$

Wir werden zeigen, daß nach der Ausführung einer atomaren Aktion $l(u).f = u.f$ und $u.f \neq r(u).f$ gilt. Unabhängig von seinem Zustandsbit führt u die Anweisung: $coin.f := \neg coin.f;$ aus. Nun gilt:

$$coin.f \neq u.f$$

Danach führt u die Anweisung $u.f := coin.f;$ aus. Somit gilt:

$$r(u).f \neq u.f \wedge u.f = l(u).f$$

Alle folgenden Anweisungen ändern die Follower-Bits nicht. Nachdem u die Münze an $l(u)$ gesendet hat, ist die atomare Aktion beendet, und (3.1) gilt wiederum. □

Nun verbinden wir Lemma 3.2 und Lemma 3.3 zu einer Invariante:

Korollar 3.4 $\Sigma_F \models \forall x \in U : \Box(coin(x) \Rightarrow coin.f = x.f)$

Beweis: folgt direkt aus Lemma 3.2 und Lemma 3.3. □

Lemma 3.5 $\Sigma_F \models \forall x \in U : \Box \Diamond(x.state = leader)$

Beweis: Jeder Agent $x \in U$ wird die Münze immer wieder erhalten (Endlichkeit des Rings). Jeder Agent, der die Münze erhält, ist entweder schon ein Leader oder er wird wegen Korollar 3.4 die Anweisungsfolge *x.state* := *leader; x.l* := *coin.l;* ausführen. Dadurch wird er auch am Ende seines atomaren Schritts ein Leader sein. Dies impliziert die Behauptung. □

Nun beweisen wir, daß der Algorithmus von Mayer, Ostrovsky und Yung in Σ_F keine selbststabilisierende Leader Election gewährleistet. Dazu spezifizieren wir, was Leader Election bedeutet. Uns genügt eine Implikation von Leader Election (für ein System mit mehreren Agenten): In jedem Ablauf gibt es einen Agenten, der schließlich nie mehr Leader ist:

$$\exists\, x \in U : \Diamond \Box \neg(x.state = leader) \tag{3.2}$$

Theorem 3.6 Der Algorithmus von Mayer, Ostrovsky und Yung ist nicht selbststabilisierend.

Beweis: Wir gehen von Lemma 3.5 aus:

$$\Sigma_F \models \forall x \in U : \Box \Diamond(x.state = leader)$$

Dies impliziert (Definition □):

$$\Sigma_F \models \forall x \in U : \neg \Diamond \neg \Diamond(x.state = leader)$$

Dies impliziert (Definition □):

$$\Sigma_F \models \forall x \in U : \neg \Diamond \Box \neg(x.state = leader)$$

Dies steht im Widerspruch zu Leader Election (3.2). Also gibt es ein System und einen Anfangszustand, für den der Algorithmus keine Leader-Election gewährleistet. Dies impliziert die Behauptung. □

Im folgenden Abschnitt entwickeln wir aus der algorithmischen Idee des Algorithmus einen korrekten Algorithmus.

Die algorithmische Idee von Mayer, Ostrovsky und Yung

Die algorithmische Idee, die dem Algorithmus von Mayer, Ostrovsky und Yung zugrunde liegt, beschreiben die Autoren in [MOY96] wie folgt: Die Follower-Bits sorgen dafür, daß schließlich *mindestens* ein Leader existiert. Um dies zu illustrieren, gehen wir von einem Zustand aus, in dem kein Leader existiert. Falls das Follower-Bit der Münze sich vom Follower-Bit des Agenten unterscheidet, dann kopiert der Agent das Follower-Bit der Münze. Irgendwann erreicht die Münze einen Agenten, dessen Follower-Bit dem Follower-Bit der Münze gleicht. Dieser wird dann ein Leader.

Die Leader-Bits sorgen dafür, daß schließlich nur noch *höchstens* ein Leader existiert. Die zugrundeliegende Idee ist, daß jeder Leader bei der Münze einen zufälligen *Fußabdruck*[1] (rechter Fuß oder linker Fuß) im Leader-Bit hinterläßt und sich diesen Abdruck in seinem eigenen Leader-Bit merkt. Wenn die Münze auf einen Leader mit einem anderen Fußabdruck trifft, dann kann dieser Leader nicht der einzige Leader sein – er wird zum Follower.

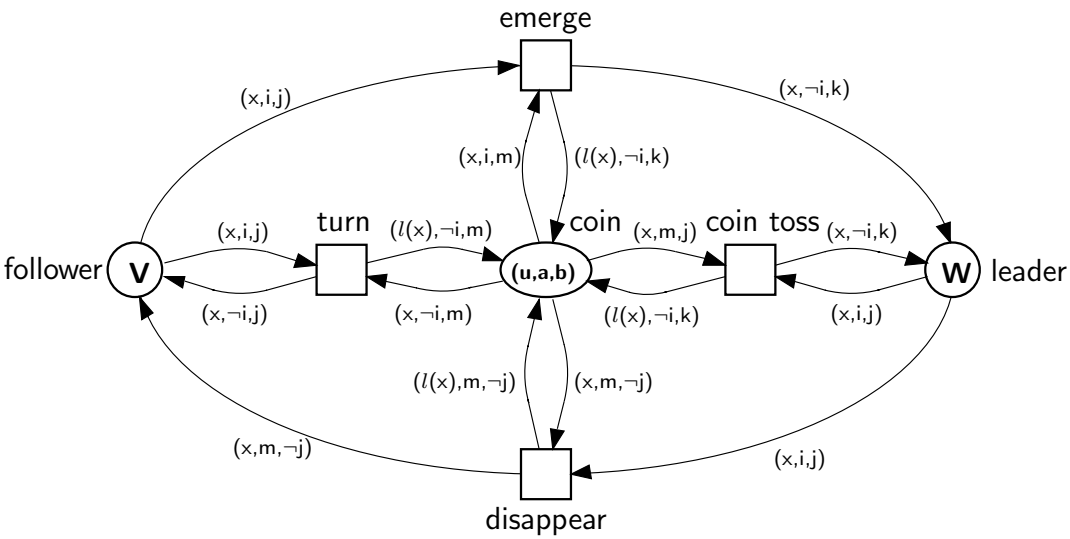
Beim originalen Algorithmus von Mayer, Ostrovsky und Yung ist die Behandlung von Follower-Bits durch Leader fehlerhaft. Wir schlagen eine Veränderung vor, deren Protokoll in Abbildung 3.3 dargestellt ist.

```
receive coin;
if (x.state = follower)
      if (coin.f = x.f)
            coin.f := ¬coin.f; x.state := leader; x.l := coin.l;
      else
            x.f := coin.f;
if (x.state = leader)
      if (coin.l ≠ x.l)
            x.state := follower; x.f := coin.f;
      else
            coin.l := coin-toss; x.l := coin.l; x.f := ¬x.f;
send coin to l(x);
```

Abbildung 3.3: Protokoll für einen Agenten $x \in U$

In Abbildung 3.4 ist ein Petrinetzmodell dieses Algorithmus angegeben. Wir gehen wiederum von einem unidirektionalen Ring (U, l) aus. Der Domain der Stellen follower und leader besteht aus Tripeln. Ein Tripel $(x, i, j) \in U \times \mathbb{B} \times \mathbb{B}$

[1] Der Begriff des Fußabdrucks wurde erstmals in [MOOY92] geprägt. Er sollte zur Intuition eines „wandernden Tokens“ passen.



spec (U,l) : unidirectional ring
sort value = site × bool × bool
const V, W : set of values
const (u,a,b) : value

var x : site
var i, j, k, m : bool
$V_1 \cup W_1 = U$
$V_1 \cap W_1 = \emptyset$
$\exists x \in U : x = u$

Abbildung 3.4: Σ_3 – Randomisierte Leader-Election im unidirektionalen Ring

auf follower (leader) beschreibt, daß Agent x das Follower-Bit i und das Leader-Bit j hat und sich im Zustand Follower (Leader) befindet. Jeder Agent befindet sich immer in einem dieser beiden Zustände. Dementsprechend nennen wir in jedem Zustand jeden Agenten entweder einen Leader oder einen Follower.

Der Anfangszustand jedes Agenten ist beliebig. Das heißt, initial gibt es eine (möglicherweise sogar leere) Menge von Followern und eine (ebenfalls möglicherweise leere) Menge von Leadern.

Es gibt vier unterschiedliche Aktionen, die ein Agent ausführen kann. Diese entsprechen den vier Transitionen im Modell. Ein Follower kann ein Follower bleiben (turn) oder ein Leader werden (emerge). Ebenso kann ein Leader ein Leader bleiben (coin toss) oder ein Follower werden (disappear).

Der Domain der Stelle coin besteht aus Tripeln. Der erste Wert eines Tripels $(x, i, j) \in U \times \mathbb{B} \times \mathbb{B}$ auf coin kennzeichnet denjenigen Agenten, der gerade die Münze besitzt. Es gibt immer genau eine Münze im System. Ein Agent muß die Münze besitzen, um eine Aktion ausführen zu können. Beim Ausführen einer

Aktion wird die Münze an den linken Nachbarn verschickt, der sie danach besitzt. Initial besitzt irgendein Agent $u \in U$ die Münze.

Der zweite Wert i (das Follower-Bit) der Münze bestimmt, welche Aktion ein Follower ausführen wird. Wenn i dem Follower-Bit eines Followers gleicht, dann wird dieser Agent zum Leader. Anderenfalls bleibt der Agent ein Follower. Dementsprechend bestimmt der dritte Wert j (das Leader-Bit) der Münze, welche Aktion ein Leader ausführt. Wenn j dem Leader-Bit eines Leaders gleicht, dann bleibt dieser Agent ein Leader. Anderenfalls wird der Leader ein Follower.

Das Follower- und das Leader-Bit der Münze werden nur von Agenten verändert, die nach der Ausführung ihrer Aktion Leader sind (emerge und coin toss). Jeder dieser Agenten negiert sein eigenes Follower-Bit und schreibt sein neues Follower-Bit auf die Münze. Ein Leader-Bit wird nichtdeterministisch gewählt, d. h. wenn emerge oder coin toss in einem Modus β aktiviert sind, mit $\beta(\mathsf{k}) = 0$, dann sind sie auch in einem Modus β' aktiviert, mit $\beta'(\mathsf{k}) = 1$. Wir nehmen Randomisierung für diese nichtdeterministische Auswahl an – dies korrespondiert mit der Intuition eines Münzwurfs.

Für den originalen Algorithmus von Mayer, Ostrovsky und Yung kann man ein ähnliches Petrinetzmodell angeben. Der einzige Unterschied zu Abbildung 3.4 besteht in der Beschriftung der Kante von disappear nach coin: Die zweite Komponente des Tripels ist $\neg\mathsf{i}$ anstelle von m. Dies unterstreicht die nahe Verwandtschaft der beiden Algorithmen.

Abbildung 3.5 zeigt ein Ablaufstück von Σ_3 in einem System mit drei Agenten u_0, u_1 und u_2. Der Anfangszustand ist zulässig: es gibt genau einen Leader u_0. Die Transitionen emerge und disappear treten im Ablauf nicht auf, also bleibt jeder Follower ein Follower, und der Leader bleibt ein Leader. u_0 wirft in diesem Ablauf zweimal die Münze: zuerst mit dem Resultat 1 und danach mit dem Resultat 0.

Wir spezifizieren nun die entscheidenden Eigenschaften von Σ_3. Die Sicherheitseigenschaft beschreibt, daß es immer genau eine Münze gibt, also die Anzahl der Marken auf coin immer gleich 1 ist:

$$\Sigma_3 \models \Box(|\mathsf{coin}| = 1) \tag{3.3}$$

Die Lebendigkeitseigenschaft beschreibt, daß (unter der Annahme von Randomisierung) in jedem Ablauf von Σ_3 schließlich genau ein Leader gewählt wird:

$$\Sigma_3 \models_{\mathcal{R}} \exists\, x \in U : \Diamond\, \Box(\mathsf{leader}_1 = [x]) \tag{3.4}$$

Im folgenden Abschnitt stellen wir einen hinsichtlich des Speicherplatzbedarfs besseren Algorithmus für dasselbe Problem vor.

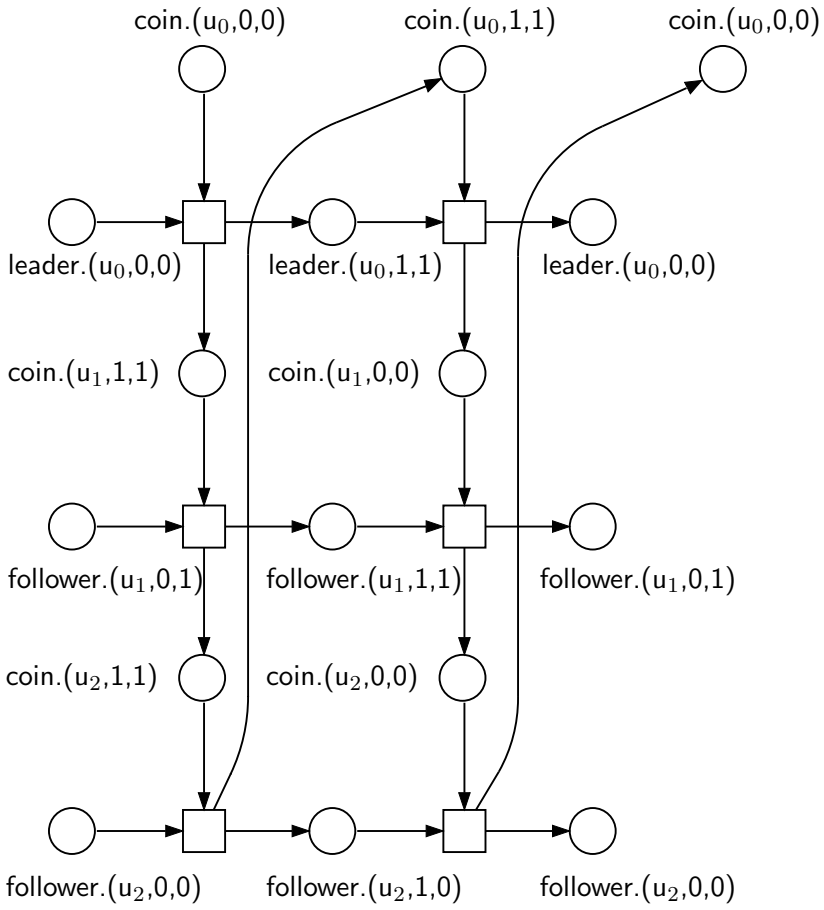


Abbildung 3.5: Ein Ablaufstück von Σ_3

Ein neuer Algorithmus

Wir betrachten noch einmal den Algorithmus Σ_3 in Abbildung 3.4. Im Netzmodell ist ersichtlich, daß ein Follower sein Leader-Bit nie benötigt. Weder bei turn noch bei emerge wird dieses Bit zur Synchronisation oder zur Veränderung von Bits der Münze genutzt. Das Follower-Bit eines Leaders wird nur bei coin toss benutzt, um den neuen Follower-Wert der Münze zu berechnen. Anhand des Ablaufs von Σ_3 in Abbildung 3.5 können wir jedoch vermuten, daß zumindest im stabilisierten Fall das Follower-Bit der Münze dem Follower-Bit eines Leaders, der die Münze besitzt, ohnehin gleicht. Wir werden in diesem Abschnitt einen Algorithmus angeben, der pro Agent ein Bit weniger verwendet.

Eine zweite Beobachtung betrifft emerge. Offensichtlich wirft ein Follower,

der zum Leader wird, sofort die Münze. Wenn ein Follower in diesem Fall jedoch nicht die Münze wirft, sondern das Leader-Bit der Münze übernimmt, dann gibt es nur noch eine Transition (coin toss) im Algorithmus, bei der die Münze geworfen wird. Eine genaue Analyse unseres neuen Algorithmus wird zeigen, daß dies die erwartete Anzahl der Münzwürfe bis zur Stabilisierung senken wird.

Wir geben wiederum zuerst in Abbildung 3.6 eine Pseudocode-Beschreibung des Protokolls eines Agenten $x \in U$ an. Anstelle der Follower- und Leader-Bits $x.f$ und $x.l$ hat x nur noch einen *lokalen Wert* $x.value$.

```
receive coin;
if (x.state = follower)
    if (x.value = coin.f)
        x.state := leader; x.value := coin.l; coin.f := ¬coin.f;
    else
        x.value := coin.f;
else
    if (x.value = coin.l)
        coin.l := coin-toss; x.value := coin.l; coin.f := ¬coin.f;
    else
        x.state := follower; x.value := coin.f;
send coin to l(x);
```

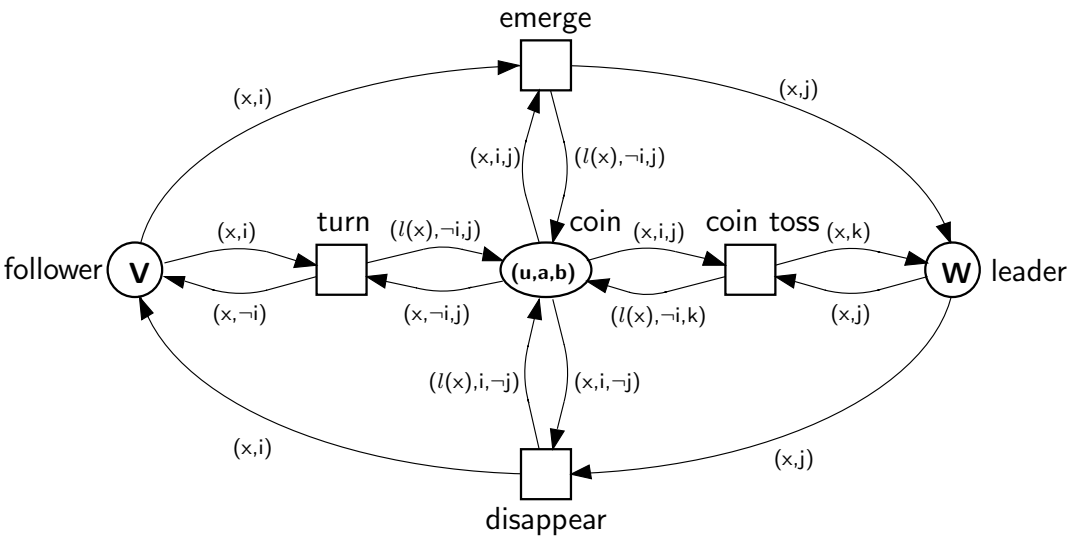
Abbildung 3.6: Protokoll für einen Agenten $x \in U$

In Abbildung 3.7 ist ein Petrinetzmodell dieses Algorithmus angegeben. Wir gehen wiederum von einem unidirektionalen Ring (U, l) aus. Der Domain der Stellen follower und leader besteht aus Paaren. Ein Paar $(x, i) \in U \times \mathbb{B}$ auf follower (leader) beschreibt, daß Agent x den lokalen Wert i hat und sich im Zustand Follower (Leader) befindet.

Der Anfangszustand jedes Agenten ist beliebig. Das heißt, initial gibt es eine (möglicherweise sogar leere) Menge von Followern und eine (ebenfalls möglicherweise leere) Menge von Leadern.

Es gibt wiederum vier unterschiedliche Aktionen, die ein Agent ausführen kann. Diese entsprechen den vier Transitionen im Modell. Ein Follower kann ein Follower bleiben (turn) oder ein Leader werden (emerge). Ebenso kann ein Leader ein Leader bleiben (coin toss) oder ein Follower werden (disappear).

Der Domain der Stelle coin besteht aus Tripeln. Der erste Wert eines Tripels $(x, i, j) \in U \times \mathbb{B} \times \mathbb{B}$ auf coin kennzeichnet denjenigen Agenten, der gerade die Münze besitzt. Es gibt immer genau eine Münze im System. Ein Agent muß die Münze besitzen, um eine Aktion ausführen zu können. Beim Ausführen einer



spec (U,l) : unidirectional ring	var x : site
sort value = site × bool	var i, j, k : bool
sort coin = site × bool × bool	$V_1 \cup W_1 = U$
const V, W : set of values	$V_1 \cap W_1 = \emptyset$
const (u,a,b) : coin	$\exists x \in U : x = u$

Abbildung 3.7: Σ_4 – Randomisierte Leader Election im unidirektionalen Ring

Aktion wird die Münze an den linken Nachbarn verschickt, der sie danach besitzt. Initial besitzt irgendein Agent $u \in U$ die Münze.

Der zweite Wert i (das Follower-Bit) der Münze bestimmt, welche Aktion ein Follower ausführen wird. Wenn i dem lokalen Wert eines Followers gleicht, dann wird dieser Agent zum Leader. Anderenfalls bleibt der Agent ein Follower. Dementsprechend bestimmt der dritte Wert j (das Leader-Bit) der Münze, welche Aktion ein Leader ausführt. Wenn j dem lokalen Wert eines Leaders gleicht, dann bleibt dieser Agent ein Leader. Anderenfalls wird der Agent zum Follower.

Das Follower- und das Leader-Bit der Münze werden nur von Agenten verändert, die nach der Ausführung ihrer Aktion Leader sind (emerge und coin toss). Jeder dieser Agenten negiert das Follower-Bit der Münze. Ein Leader-Bit wird nichtdeterministisch gewählt. Wir nehmen wiederum Randomisierung für diese nichtdeterministische Auswahl an.

Abbildung 3.8 zeigt einen Ablauf von Σ_4 in einem System mit drei Agenten

u_0, u_1 und u_2. Das System ist stabil, es gibt genau einen Leader u_0. Der Ablauf

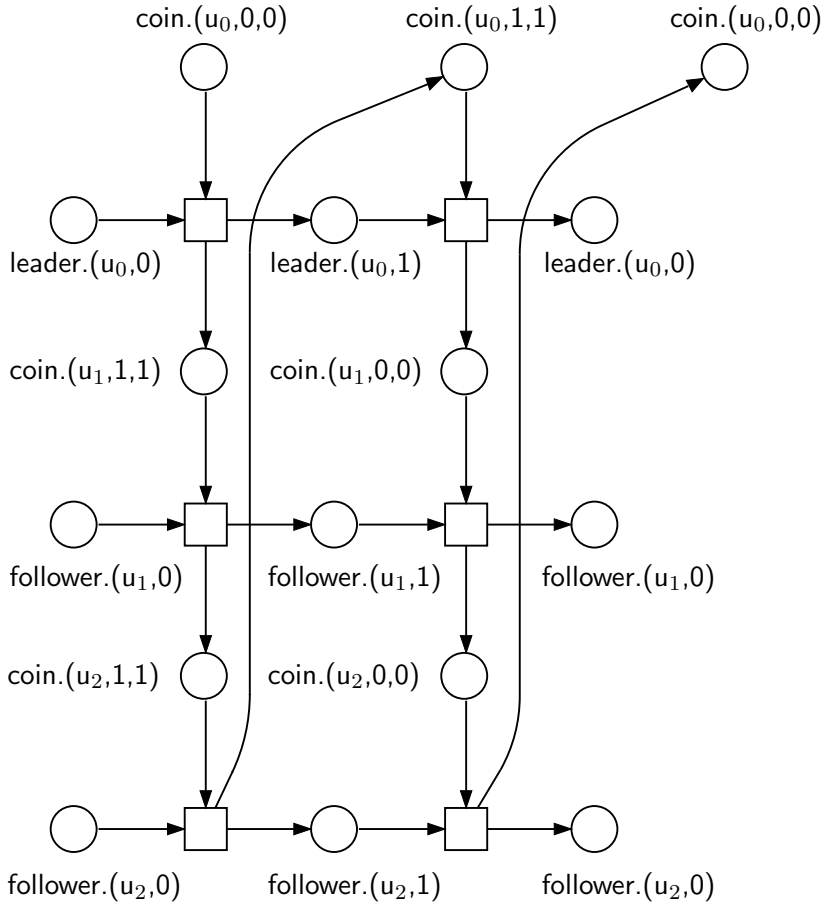


Abbildung 3.8: Ein Ablaufstück von Σ_4

gleicht dem Ablauf in Abbildung 3.5 bis auf die Anzahl der Bits der Agenten.

Wir fordern dieselben Eigenschaften wie von Σ_3, also einerseits die Sicherheitseigenschaft, daß es immer genau eine Münze gibt:

$$\Sigma_4 \models \Box(|\mathsf{coin}| = 1) \tag{3.5}$$

und andererseits die Lebendigkeitseigenschaft, daß schließlich ein eindeutiger Leader gewählt wird:

$$\Sigma_4 \models_{\mathcal{R}} \exists\, x \in U : \Diamond\, \Box(\mathsf{leader}_1 = [x]) \tag{3.6}$$

Wir werden die Gültigkeit dieser Eigenschaften in Abschnitt 5.1 nachweisen.

3.2 Asynchrone Systeme

In diesem Abschnitt entwickeln wir schrittweise einen Algorithmus, der unter einer Fairneßannahme Selbststabilisierung in einem asynchronen System gewährleistet. Es ist in der Literatur belegt ([GM91, KP93]), daß eine Fairneßannahme hierzu notwendig ist.

In einem asynchronen System ist die Kommunikation zwischen den Agenten asynchron. Jeder Agent hat lokale Variablen, auf die kein anderer Agent zugreifen kann. Agenten kommunizieren ausschließlich über Nachrichtenkanäle. Die Agenten zusammen mit den Nachrichtenkanälen bilden ein Netzwerk. Jeder Nachrichtenkanal hat eine endliche, aber unbeschränkte Kapazität. Jede Nachricht, die versendet wird, erreicht irgendwann ihren Empfänger. Wir nehmen weiterhin an, daß sich Nachrichten in den Kanälen beliebig überholen können. In der Literatur wird diese Annahme oft eingeschränkt – es werden zumeist FIFO- (first in first out) Kanäle betrachtet. Manchmal ist diese Annahme jedoch nicht wesentlich für die Korrektheit eines Algorithmus. Darüberhinaus kann sie in unserem Modell erzwungen werden, indem Nachrichten mit einer Nummer (einem Zeitstempel) versehen werden.[2]

In einem asynchronen System ergeben sich folgende speziellen Schwierigkeiten beim Entwurf selbststabilisierender Algorithmen:

1. Der Zustand eines asynchronen Systems ergibt sich aus der Menge der lokalen Zustände der Agenten sowie der Menge der Nachrichten, die in den Kanälen unterwegs sind. Wir betrachten zuerst einen unzulässigen Anfangszustand ohne Nachrichten. In solch einem Zustand wird irgendwann eine Nachricht versendet (in jedem nicht-trivialen Algorithmus, der Kommunikation voraussetzt).

   Nun betrachten wir einen beliebigen Anfangszustand. In diesem Zustand muß es einen Agenten geben, der in seinem lokalen Zustand unabhängig von zu empfangenden Nachrichten selbst eine Nachricht versenden wird, da die Nachrichtenkanäle als Folge eines Fehlers leer sein könnten. Dies bedeutet insbesondere, daß mindestens ein Agent in jedem lokalen Zustand immer wieder Nachrichten versenden kann.

2. Da es einen Agenten gibt, der unabhängig von den Inhalten der Nachrichtenkanäle immer wieder eine Nachricht versendet, ist es prinzipiell auch

[2] In unserem Algorithmus wird ein Zeitstempel auch eine Rolle spielen, jedoch nicht, um FIFO zu simulieren, sondern um Stabilisierung zu garantieren. In einem System mit FIFO-Kanälen würde dieser Zeitstempel also auch auftreten.

möglich, daß dieser Agent keine anderen Aktionen ausführt, insbesondere also nie eine Nachricht empfängt. Um diesen Agenten zu zwingen, irgendwann auch einmal auf eine Nachricht zu reagieren, brauchen wir eine *Fairneßannahme*.

3. Initial kann es Nachrichten geben, die nie versendet worden sind, also potentiell fehlerhaft sind. Ein selbststabilisierender Algorithmus muß gewährleisten, daß die Auswirkungen solcher fehlerhaften Nachrichten endlich sind. Dies birgt häufig Schwierigkeiten, da Nachrichten, die aufgrund fehlerhafter Nachrichten versendet worden sind, manchmal gar nicht mehr von fehlerfreien Nachrichten zu unterscheiden sind. Wir werden diesem Phänomen in diesem Abschnitt mehrfach begegnen.

In der Literatur wurden über einen langen Zeitraum hinweg nur selbststabilisierende Algorithmen betrachtet, bei denen die Agenten über gemeinsamen Speicher kommunizieren. Die ersten Veröffentlichungen, die das in diesem Abschnitt betrachtete asynchrone Systemmodell verwendeten, waren von Gouda und Multari ([Mul89, GM91]). Sie bewiesen die folgenden drei notwendigen Bedingungen für selbststabilisierende asynchrone Systeme:

1. Jeder Ablauf ist unendlich (Nichtterminierung).

2. Mindestens ein Agent hat unendlich viele lokale zulässige Zustände.

3. Mindestens ein Agent hat *Time-Out*-Aktionen.

Die erste Bedingung ergibt sich intuitiv aus der oben betrachteten ersten Schwierigkeit. Mindestens ein Agent kann immer eine Nachricht versenden. Die zweite Bedingung ergibt sich aus der Notwendigkeit eines Zeitstempels. Jeder Anfangszustand kann unbeschränkt viele fehlerhafte Nachrichten in den Kanälen haben. Die dritte Bedingung ergibt sich aus den Systemannahmen von Gouda und Multari: Die Existenz von *Time-Out*-Aktionen ist eine Fairneßannahme. Eine *Time-Out*-Aktion tritt ein, wenn ein Agent keine Antwort auf eine von ihm gesendete Nachricht erhält. Die *Time-Out*-Aktion garantiert, daß ein Fehler im System aufgetreten ist, also daß die Antwort nicht nur verzögert, sondern gar nicht mehr ankommt. *Time-Out*-Aktionen sind daher nur schwer, wenn überhaupt, implementierbar [GM91].

Afek und Brown geben in [AB93] eine probabilistische selbststabilisierende Version des *Alternating-Bit-Protokolls* ([BSW69]) an. Eine weitere Version in [AB93] setzt die Fähigkeit eines Agenten voraus, aperiodische Sequenzen zu erzeugen. Burns, Gouda und Miller geben in [BGM93] eine *pseudo-stabilisierende* Version des Alternating-Bit-Protokolls an. Ein System

ist pseudo-stabilisierend, wenn es die *Convergence*-Eigenschaft erfüllt, die *Closure*-Eigenschaft jedoch nicht. Technisch bedeutet dies, daß jeder Ablauf ein Suffix hat, in dem das System stabilisiert ist, die Stabilisierung kann aber im allgemeinen nicht anhand eines Präfixes entschieden werden. Wir geben in Abbildung 3.9 das Beispiel aus [BGM93] an. Es zeigt den Zustandsraum eines

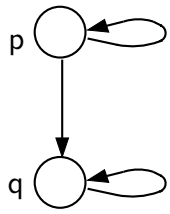


Abbildung 3.9: Ein pseudo-stabilisierendes System

pseudo-, aber nicht selbststabilisierenden Systems. Im System gilt die Eigenschaft, daß in jedem Ablauf schließlich entweder nur noch p-Zustände oder nur noch q-Zustände auftreten. Allerdings kann man in einem p-Zustand nicht entscheiden, ob das System bereits stabil ist oder nicht.

Katz und Perry geben in [KP93] eine Methode an, um *jeden* asynchronen Algorithmus zu einem selbststabilisierenden zu erweitern. Die Idee besteht darin, regelmäßig einen globalen „Schnappschuß“ (engl. snapshot) des Systems zu erstellen. Wenn ein Agent alle Informationen über einen (bereits vergangenen) globalen Zustand gesammelt hat, entscheidet er, ob dieser Zustand zulässig war[3]. Falls nicht, wird ein globaler Reset-Algorithmus gestartet, der das System in einen zulässigen Zustand setzt.

Erwähnenswert sind auch die Algorithmen von Dolev et. al. [DIM97, Dol00]. Diese Algorithmen setzen voraus, daß in jedem initialen unzulässigen Zustand *mindestens eine* Nachricht im System existiert. Dadurch kann das ansonsten typische „Fluten“ des Systems mit Nachrichten vermieden werden: Es gibt Algorithmen, die die Anzahl der initialen Nachrichten konstant lassen. Dolev nennt diese Algorithmen *message-driven*. Dies bedeutet, daß jeder Agent nur dann eine Aktion ausführt, wenn er eine Nachricht empfängt.

Wir werden uns im folgenden auf ein konkretes Problem und ein konkretes System festlegen: Wir werden einen Algorithmus entwickeln, der selbststabilisierenden Tokenaustausch zwischen zwei Agenten realisiert, vergleiche Abschnitt 1.3.

[3]Für diese Entscheidung muß schlimmstenfalls der gesamte Zustandsraum durchsucht werden. Daher führt diese Methode nur selten zu effizienten Algorithmen.

3.2.1 Tokenaustausch

Wir werden in diesem Abschnitt ausgehend von einer einfachen algorithmischen Idee in mehreren Teilschritten einen selbststabilisierenden Algorithmus für Tokenaustausch in asynchronen Systemen entwickeln. Wir beginnen mit dem Algorithmus von Dijkstra für ein *synchrones* System, den wir bereits in Abschnitt 1.3 modelliert haben. Wir geben daher hier ohne weitere Erläuterungen nur das Modell und die spezifizierten Eigenschaften an.

Das Petrinetzmodell ist in Abbildung 3.10 angegeben, und die wesentlichen

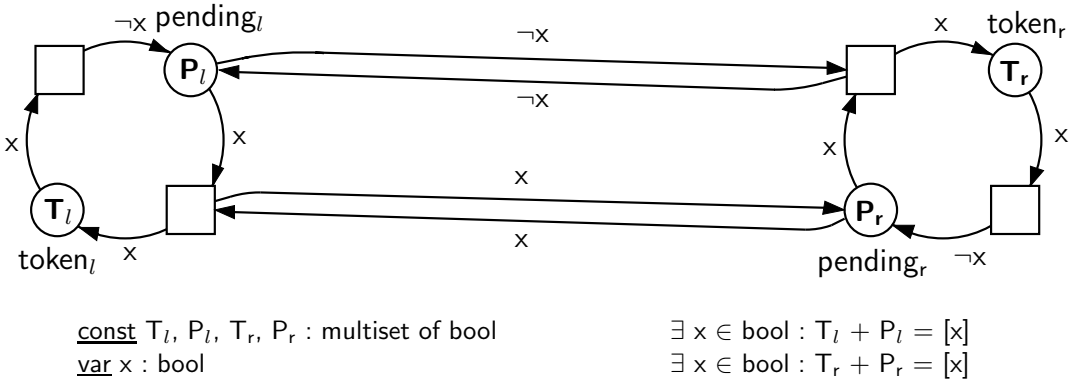


Abbildung 3.10: Σ_5 – Synchroner Tokenaustausch

Eigenschaften haben wir wie folgt spezifiziert:

$$\Sigma_5 \models \mathit{true} \mapsto (|\mathsf{token}_l| > 0) \tag{3.7}$$

$$\Sigma_5 \models \mathit{true} \mapsto (|\mathsf{token}_r| > 0) \tag{3.8}$$

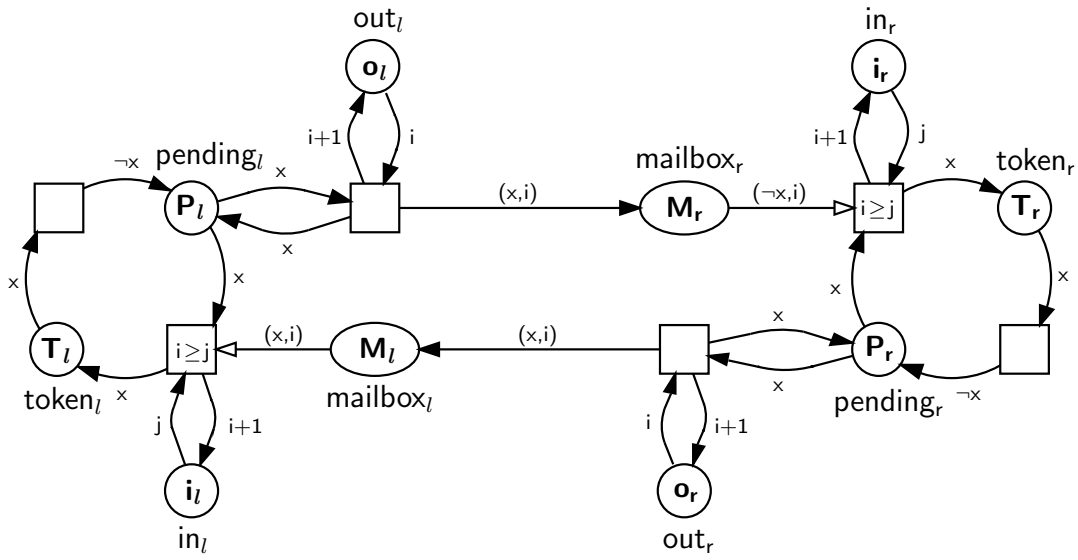
Diese beiden Eigenschaften beschreiben die Lebendigkeitseigenschaften für die beiden Agenten: Jeder Agent wird immer wieder das Token erhalten.

$$\Sigma_5 \models \Diamond\,\Box(|\mathsf{token}_l + \mathsf{token}_r| \leqslant 1) \tag{3.9}$$

Diese Eigenschaft beschreibt, daß schließlich höchstens ein Token existiert.

Eine naheliegende asynchrone Lösung

Wir greifen den Algorithmus von Dijkstra auf und verfeinern die Kommunikation zu einer asynchronen Kommunikation. Eine naheliegende Lösung, die



sort message = bool × nat	var x : bool
const T_l, P_l, T_r, P_r : multiset of bool	var i, j : nat
const o_l, i_l, o_r, i_r : nat	$\exists$ x $\in$ bool : $T_l + P_l$ = [x]
const M_l, M_r : finite multiset of messages	$\exists$ x $\in$ bool : $T_r + P_r$ = [x]

Abbildung 3.11: Σ_6 – Asynchroner Tokenaustausch mit Rundenfehler

zumindest die Eigenschaften (3.7) und (3.8) erfüllt, ist in Abbildung 3.11 angegeben. Wir gehen wiederum von zwei Agenten l und r aus. Ihre lokalen Zustände und die Zustandsübergänge sind im Vergleich zum Algorithmus von Dijkstra unverändert.

Die Agenten l und r kommunizieren über Nachrichten auf den Stellen mailbox$_l$ und mailbox$_r$. Der Domain dieser Stellen sind Paare. Der erste Wert einer Marke $(x, i) \in \mathbb{B} \times \mathbb{N}$ auf einer Stelle mailbox bezeichnet den lokalen Wert des Senders zum Zeitpunkt des Absendens dieser Nachricht. Der zweite Wert ist ein *Zeitstempel*. Er sorgt dafür, daß zwei Nachrichten, die in einer Reihenfolge abgesendet werden, nicht in einer anderen Reihenfolge empfangen werden können.

Der Anfangszustand der Nachrichtenkanäle ist beliebig. Das heißt, initial gibt es eine endliche Multimenge von Nachrichten für den linken Agent und eine endliche Multimenge von Nachrichten für den rechten Agent.

Zur Verwaltung der Zeitstempel hat jeder Agent zwei lokale Zähler in und out. Auf diesen Stellen ist immer jeweils eine natürliche Zahl. Beim Senden ei-

ner Nachricht wird der out-Zähler inkrementiert. Eine Nachricht kann nur dann empfangen werden, wenn ihr Zeitstempel größer oder gleich dem in-Zähler ist. Veraltete Nachrichten werden ignoriert. Initial sind die Zählerstände beliebig.

Wir verlangen, daß eingehende Nachrichten *fair* behandelt werden. Ein Ablauf von Σ_6 ist unfair, falls eine Nachricht existiert, die schließlich immer empfangen werden kann, aber nie empfangen wird (weil der betreffende Agent nur sendet).

Dieser Algorithmus hat einen *Rundenfehler*. Der Begriff „Rundenfehler" wurde von Walter in [Wal95] geprägt. Er beschreibt ein Phänomen, das bei verteilten asynchronen Algorithmen allgemein auftreten kann: Intuitiv arbeiten Agenten in Runden. Eine Runde beschreibt prinzipiell zyklisches Verhalten: Bestimmte lokale Zustände werden in jedem Ablauf immer wieder erreicht. Beispielsweise empfängt der linke Agent immer wieder eine Nachricht mit dem Wert 0. Jede gesendete Nachricht kann somit einer Runde zugeordnet werden. Falls es möglich ist, daß eine Nachricht, die in Runde n gesendet wurde von einem Agenten in Runde $n-1$ empfangen wird, dann liegt ein Rundenfehler vor.

Abbildung 3.12 zeigt ein Ablaufstück mit einem Rundenfehler. Hierbei sind die Stellen pending, token und mailbox jeweils durch ihre Anfangsbuchstaben abgekürzt. Der Anfangszustand und der Endzustand sind beide unzulässig, da sich beide Agenten im Zustand token befinden. Außerdem sind der Anfangszustand und der Endzustand in gewisser Weise symmetrisch: Alle Zähler und Zeitstempel der Nachrichten sind um 2 erhöht. Der Rundenfehler wird an der Nachricht $\mathsf{m}_r.(0,3)$ deutlich: Der rechte Agent r geht davon aus, daß diese Nachricht die neueste Nachricht ist, die er von l erhält. Weiterhin geht r jedoch auch davon aus, daß sie eine Reaktion auf seine gesendete Nachricht $\mathsf{m}_l.(0,4)$ ist. Hier tritt also ein Rundenfehler auf, der im nächsten Algorithmus durch eine zusätzliche Synchronisation der Zähler behoben wird. Das Einführen zusätzlicher Synchronisation wurde auch in [Wal95] genutzt, um Rundenfehler zu beheben.

Ein neuer Algorithmus

Die oben beschriebene zusätzliche Synchronisation werden wir in zwei Schritten erreichen. Zum einen synchronisieren wir die lokalen Zähler in und out nach dem Empfang einer Nachricht. Dies allein genügt jedoch noch nicht. Daher erweitern wir den Definitionsbereich unserer Nachrichten. Bisher konnte eine Nachricht zwei Werte annehmen (0 oder 1), nun kann sie drei verschiedene Werte annehmen (0, 1 oder 2). Die Notwendigkeit einer solchen Erweiterung wird bei der detaillierten Analyse des Algorithmus deutlich. Wir werden am

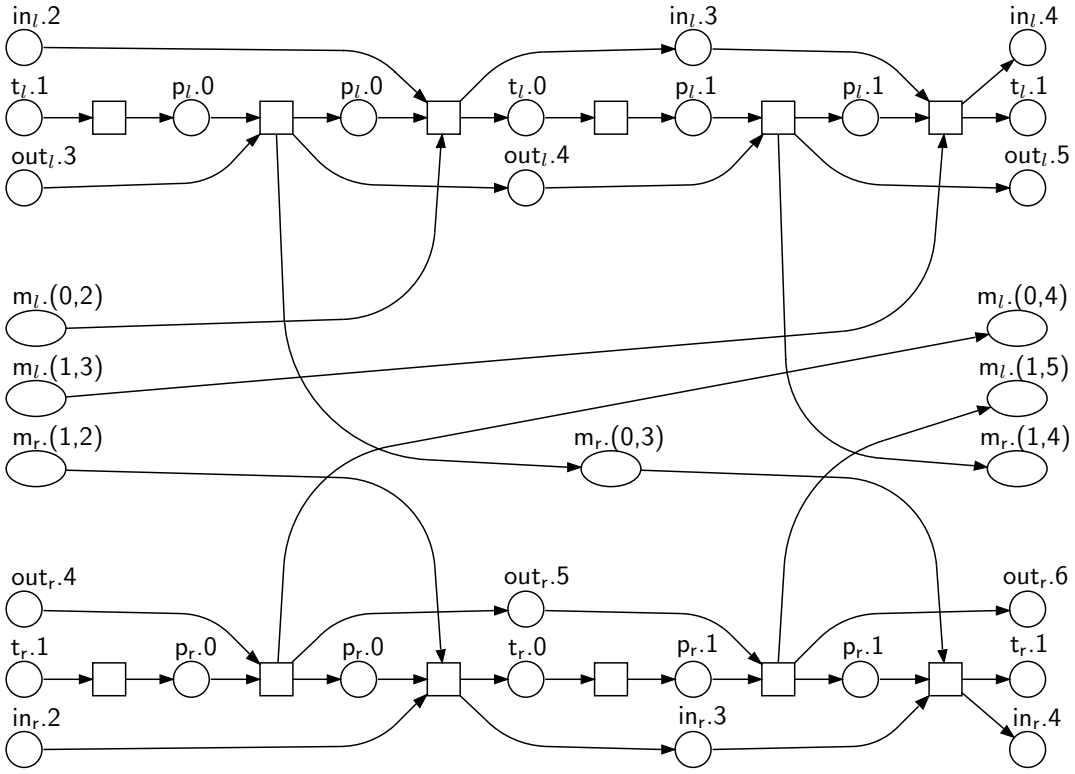
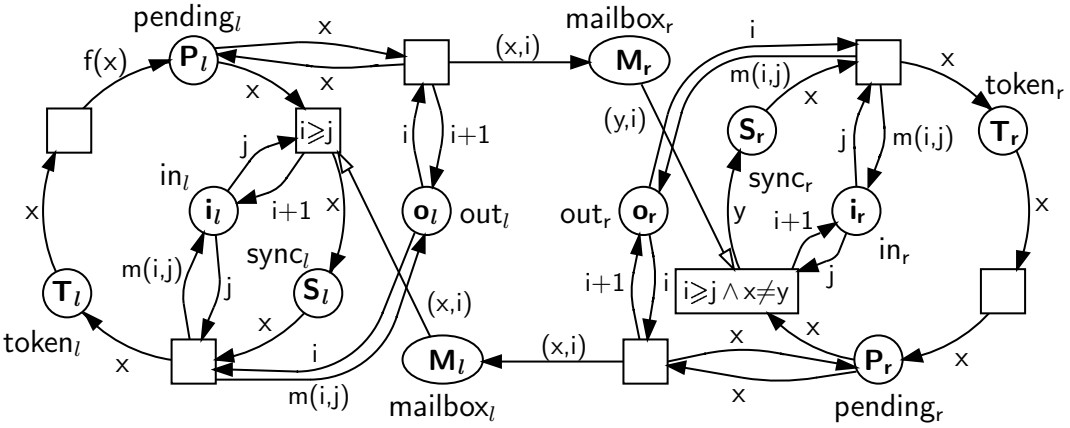


Abbildung 3.12: Ein Ablaufstück von Σ_6 mit Rundenfehler

Ende des Abschnitts 5.2 darauf zurückkommen.

In Abbildung 3.13 ist ein Petrinetzmodell eines neuen Algorithmus für den selbststabilisierenden Tokenaustausch zwischen zwei Agenten angegeben. Wir



sort 3val	var x, y : 3val
sort message = 3val × nat	var i, j : nat
fct f : 3val ⟶ 3val	f(0) = 1
const 0, 1, 2 : 3val	f(1) = 2
const P_l, S_l, T_l, P_r, S_r, T_r : multiset of 3val	f(2) = 0
const i_l, o_l, i_r, o_r : nat	$\exists$ x ∈ 3val : $P_l + S_l + T_l$ = [x]
const M_l, M_r : finite multiset of messages	$\exists$ x ∈ 3val : $P_r + S_r + T_r$ = [x]

Abbildung 3.13: Σ_7 – Asynchroner Tokenaustausch

gehen wiederum von einem asynchronen System mit zwei Agenten l und r aus. l und r verfügen jeweils über einen lokalen Wert. Dieser kann 0, 1 oder 2 betragen. Für $i \in \{l, r\}$ besteht der Domain der Stellen $\mathsf{pending}_i$, sync_i und token_i aus diesen drei Werten. Eine Marke $x \in \{0, 1, 2\}$ auf einer dieser Stellen p bedeutet, daß sich der Agent i im lokalen Zustand p befindet und den Wert x hat. Jeder Agent befindet sich immer in einem der Zustände pending, sync oder token. Initial sind der lokale Zustand und die Werte der Agenten beliebig.

Die Agenten kommunizieren asynchron über Nachrichtenaustausch. Die Nachrichtenkanäle sind durch die Stellen $\mathsf{mailbox}_l$ und $\mathsf{mailbox}_r$ modelliert. Der Domain dieser Stellen und die Anfangszustand sind wie in Algorithmus Σ_6; lediglich der Wert einer Nachricht kann nun 0, 1 oder 2 betragen. Wir verlangen wiederum Fairneß bei der Behandlung eingehender Nachrichten.

Die Zähler in und out sind ebenfalls wie in Σ_6. Der wesentliche Unterschied besteht darin, daß die Zähler hier synchronisiert werden – beim Übergang

in den Zustand token werden die Werte des lokalen in- und des lokalen out-Zählers verglichen und beide auf den maximalen Wert gesetzt. Im Netzmodell in Abbildung 3.13 steht m(i,j) abkürzend für max(i,j), d. h. das Maximum von i und j.

Beim Übergang in den Zustand $\mathsf{pending}_l$ nimmt der linke Agent periodisch aufeinanderfolgend die Werte 0, 1 und 2 an. Im Zustand $\mathsf{pending}_l$ kann der linke Agent analog zu Σ_6 nur Nachrichten empfangen, die den gleichen Wert haben, wie die aktuell von ihm versendeten Nachrichten. Analog dazu kann der rechte Agent im Zustand $\mathsf{pending}_\mathsf{r}$ nur Nachrichten empfangen, die einen anderen als seinen lokalen Wert haben. Da der empfangene Wert nicht mehr eindeutig ist, merkt er sich beim Übergang in den Zustand $\mathsf{token}_\mathsf{r}$ diesen Wert. Er kann ihn wieder an den linken Agenten zurücksenden, wenn er wieder im Zustand $\mathsf{pending}_\mathsf{r}$ ist.

Abbildung 3.14 zeigt ein Ablaufstück von Σ_7. Im oberen Teil ist das Ver-

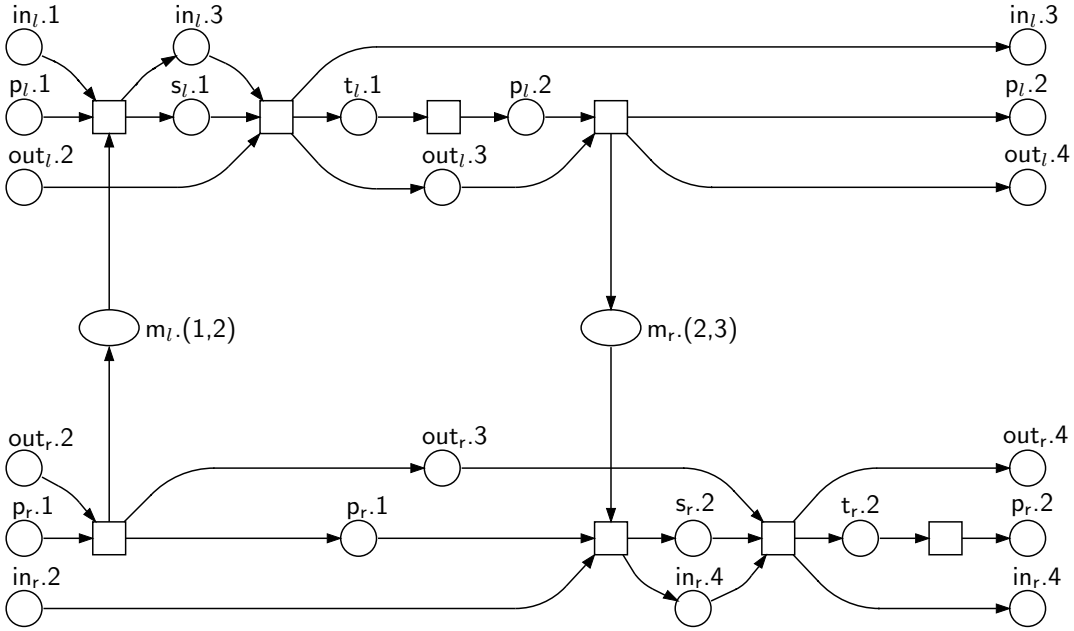


Abbildung 3.14: Ein Ablaufstück von Σ_7

halten des Agenten l dargestellt; im unteren Teil ist das Verhalten des Agenten r dargestellt. Initial befinden sich beide Agenten im Zustand pending. Außerdem haben beide den lokalen Wert 1. Analog zum Algorithmus von Dijkstra hat l in diesem Fall die Priorität – r sendet eine Nachricht, l empfängt diese, synchronisiert seine Zähler und geht in den Zustand token_l über. Nach dem

Verlassen des Zustands token_l sind die lokalen Werte von l und r ungleich. l sendet r eine Nachricht, r empfängt diese Nachricht und verhält sich analog.

Wir fordern dieselben Eigenschaften wie von Σ_5:

$$\Sigma_7 \models \mathit{true} \mapsto (|\mathsf{token}_l| > 0) \tag{3.10}$$

$$\Sigma_7 \models \mathit{true} \mapsto (|\mathsf{token}_\mathsf{r}| > 0) \tag{3.11}$$

$$\Sigma_7 \models \Diamond\,\Box(|\mathsf{token}_l + \mathsf{token}_\mathsf{r}| \leqslant 1) \tag{3.12}$$

Wir werden die Korrektheit von Σ_7 im Abschnitt 5.2 beweisen.

3.3 Systeme mit gemeinsamem Speicher

In diesem Abschnitt stellen wir weitere Algorithmen vor. Alle arbeiten auf Ketten von Agenten. Alle Algorithmen sind „typische“ selbststabilisierende Algorithmen, da sie für Systeme mit gemeinsamem Speicher entworfen wurden (vergleiche Abschnitt 1.2).

Der erste Algorithmus realisiert wiederum selbststabilisierenden Tokenaustausch (vergleiche Abschnitt 3.2). Der zweite Algorithmus realisiert einen verteilten Zähler. Er basiert auf einem selbststabilisierenden Tokenaustausch. Jedoch sind die beiden folgenden Algorithmen nicht ohne weiteres kombinierbar, da der Algorithmus zum verteilten Zählen eine zusätzliche Information über die aktuelle Richtung des Tokenflusses benötigt.

3.3.1 Tokenaustausch

In Abschnitt 3.2 haben wir einen Algorithmus vorgestellt, der in einer asynchronen Umgebung funktioniert. Hier betrachten wir eine *synchrone* Umgebung sowie eine *Kette* von Agenten (vergleiche Abschnitt 2.5). Wir stellen im folgenden den Algorithmus von Brown, Gouda und Wu aus [BGW89] vor. Ein Petrinetzmodell wurde von Reisig erstmalig in [Rei98] veröffentlicht. Unser Modell gleicht im wesentlichen dem von Reisig (bis auf kleinere Abweichungen in der algebraischen Spezifikation des zugrundeliegenden Netzwerks). Unser Beweis (in Abschnitt 5.3) hingegen ist neu und unterscheidet sich grundlegend von dem von Reisig sowie dem von Brown, Gouda und Wu.

Wir gehen von einer unidirektionalen Kette von Agenten $(U, r, \mathit{min}, \mathit{max})$ aus. Jeder Agent $x \in U \setminus \{\mathit{min}, \mathit{max}\}$ besitzt eine Zustandsvariable $(x.\mathit{state})$,

die vier Werte annehmen kann: *critical, right, waiting* und *left*. Der Wert *critical* bezeichnet den lokalen *kritischen Bereich* eines Agenten x. Die Bezeichnung „kritischer Bereich“ stammt daher, daß das Problem des Tokenaustauschs oft mit dem Problem des *wechselseitigen Ausschlusses* (engl. *mutual exclusion*, Mutex) gleichgesetzt wird. Üblicher ist heute jedoch eine genauere Spezifikation von Mutex, bei der gefordert wird, daß ein Agent nur dann seinen kritischen Zustand betritt, wenn er dies auch möchte.[4] Der Unterschied zwischen „ein Agent *muß* seinen kritischen Bereich betreten“ und „ein Agent *möchte* seinen kritischen Bereich betreten“ ist erheblich, siehe [KW97]. Beim Tokenaustausch hingegen muß jeder Agent kritisch werden. In diesem Sinne ist unser Algorithmus kein Mutex-Algorithmus. In unserem Algorithmus bedeutet der lokale kritische Bereich eines Agenten x nur, daß x ein Token hat. Jeder Agent wird diesen Zustand irgendwann verlassen, um das Token weiterzureichen.

Wir nehmen an, daß zwei benachbarte Agenten sich synchronisieren können, d. h. sie können eine *gemeinsame Aktion* ausführen, die die Zustände beider Nachbarn verändert. Diese Annahme ist sehr stark: Sie widerspricht unserer Definition eines verteilten Systems insofern, als eine Aktion nicht mehr lokal zu einem Agenten sondern lokal zu einem Paar benachbarter Agenten ist. Eine Konsequenz dieser Annahme ist, daß die von Desel in [Des97] formulierten hinreichenden syntaktischen Bedingungen für Petrinetzmodelle verteilter Algorithmen mit gemeinsamem Speicher nicht eingehalten werden können. Konkret lassen sich Desels Kriterien auf eine spezielle Klasse verteilter Algorithmen mit gemeinsamem Speicher anwenden – solche, bei denen die Lese- und Schreibzugriffe auf gemeinsame Variablen atomar sind. Die meisten selbststabilisierenden Algorithmen verlangen jedoch stärkere Atomaritätsannahmen [DIM93].

Ein Protokoll des Algorithmus besteht aus einer Menge von *guarded commands*, vergleiche Kapitel 1: Ein „guard“ ist eine Bedingung, die einer Reihe von Anweisungen (einem „command“) vorangestellt ist. Eine atomare Aktion ist die Ausführung eines (vollständigen) command-Teils, für den diese Bedingung erfüllt ist. Jeder Agent außer *min* und *max* arbeitet nach dem in Abbildung 3.15 dargestellten Protokoll [BGW89].

Der Agent *min* befindet sich immer in einem der Zustände *critical, right* oder *waiting*. Sein Protokoll ist in Abbildung 3.16 angegeben.

Der Agent *max* befindet sich immer in einem der Zustände *critical* und *left*. Er führt das Protokoll in Abbildung 3.17 aus. Zusätzlich zu dem explizit beschriebenen Zustandsübergang kann *max* auch als Synchronisationspartner seines linken Nachbarn agieren (siehe Abbildung 3.15) und somit vom Zustand

[4] In der Dissertation [Wal95] werden Petrinetzmodelle der gängigsten Mutex-Algorithmen angegeben und verifiziert.

if $(x.state = critical)$
    $x.state := right;$
if $(x.state = right \wedge r(x).state = left)$
    $x.state := waiting;\ r(x).state := critical;$
if $(x.state = waiting \wedge r(x).state = left)$
    $x.state := left;$

Abbildung 3.15: Protokoll für einen Agenten $x \in U \setminus \{min, max\}$

if $(min.state = critical)$
    $min.state := right;$
if $(min.state = right \wedge r(min).state = left)$
    $min.state := waiting;\ r(min).state := critical;$
if $(min.state = waiting \wedge r(min).state = left)$
    $min.state := critical;$

Abbildung 3.16: Protokoll für den Agenten min

left in den Zustand *critical* übergehen.

if $(max.state = critical)$
    $max.state := left;$

Abbildung 3.17: Protokoll für den Agenten max

In Abbildung 3.18 ist ein Petrinetzmodell dieses Algorithmus angegeben. Wir gehen ebenfalls von einer unidirektionalen Kette von Agenten (U, r, min, max) aus. Der Domain der Stellen ist site, d. h. auf den Stellen befinden sich Marken der Sorte site. Eine Marke $x \in U$ auf einer Stelle bedeutet, daß sich der Agent x in diesem Zustand befindet. Jeder Agent $x \in U$ befindet sich immer in einem der Zustände critical, right, waiting oder left. Aus Symmetriegründen haben wir auch für die Agenten min und max alle vier Zustände (zusammen mit den kanonischen Transitionen für die neuen Zustandsübergänge) eingeführt. Der Anfangszustand jedes Agenten ist beliebig. D. h. initial gibt es paarweise disjunkte Mengen A, B, C und D, die jeweils die Menge der Agenten im Zustand critical, right, waiting bzw. left beschreiben. Die Vereinigung dieser Mengen ergibt die Menge U.

Die wesentlichen drei Zustandsübergänge des oben beschriebenen Protokolls finden sich im Netzmodell analog: Ein Agent x kann den Zustand critical verlassen und in den Zustand right übergehen. Im Zustand right wartet er, bis sein rechter Nachbar $r(x)$ sich im Zustand left befindet. Dann kann x sich mit

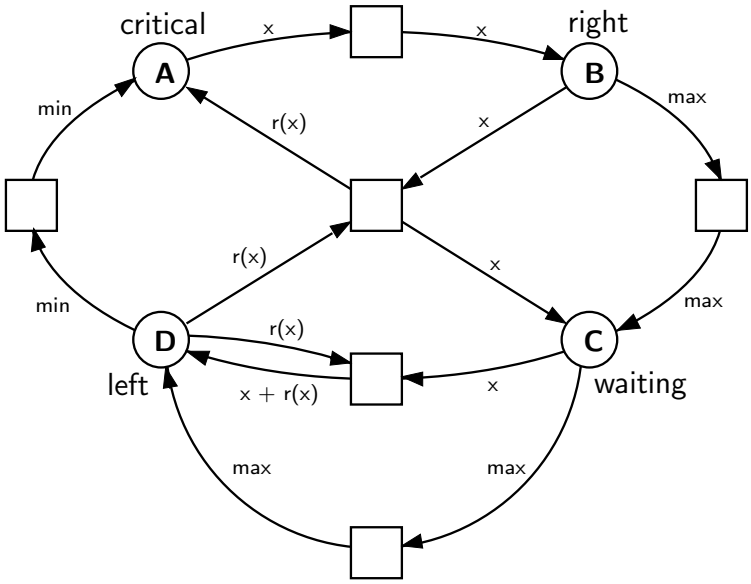


Abbildung 3.18: Σ_8 – Tokenaustausch in Ketten

seinem rechten Nachbarn synchronisieren und in den Zustand waiting übergehen. Sein rechter Nachbar geht hierbei in den Zustand critical über. Ein Agent verbleibt solange im Zustand waiting, bis sich sein rechter Nachbar im Zustand left befindet. Dann geht er ebenfalls in den Zustand left über (und wartet wiederum darauf, daß sein linker Nachbar den Zustand right erreicht).

Abbildung 3.19 zeigt ein Ablaufstück für ein System mit drei Agenten $u_0 = min, u_1$ und $u_2 = max$. Die Stellen waiting und critical sind hierbei jeweils durch wait und crit abgekürzt. Wir sehen am Ablaufstück, wie die Agenten in der Reihenfolge u_0, u_1, u_2 den Zustand critical betreten und verlassen. In umgekehrter Reihenfolge betreten sie den Zustand left. Die „Treppenlinie“ durch den Ablauf beschreibt den Weg des Tokens im System. Die Charakterisierung dieser Linie wird einen zentralen Schritt im Beweis der Korrektheit dieses Algorithmus darstellen.

Die zentralen Eigenschaften des Algorithmus spezifizieren wir wie folgt. Jeder Agent wird immer wieder ein Token besitzen:

$$\Sigma_8 \models \forall x \in U : true \mapsto \mathsf{critical}(x) \tag{3.13}$$

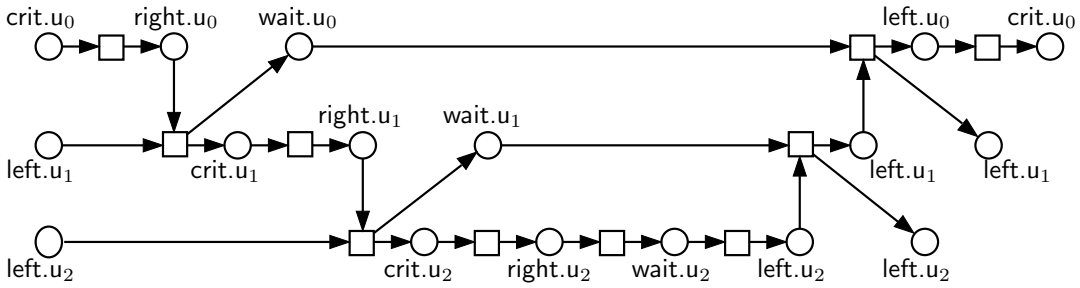


Abbildung 3.19: Ein Ablaufstück von Σ_8

Irgendwann gibt es immer höchstens einen Agenten, der ein Token besitzt:

$$\Sigma_8 \models \Diamond \Box \, |\mathsf{critical}| \leqslant 1 \tag{3.14}$$

Diese beiden Eigenschaften spiegeln die wesentlichen Theoreme aus [BGW89] wieder. Beide Eigenschaften sind Lebendigkeitseigenschaften. Ihre gemeinsame Gültigkeit garantiert Tokenaustausch. Wir beweisen die Korrektheit dieses Algorithmus in Abschnitt 5.3.

3.3.2 Verteilter Zähler

Wir präsentieren in diesem Abschnitt zwei selbststabilisierende Algorithmen zum *verteilten Zählen*. Der erste Algorithmus wurde von Abello und Dolev in [AD97] vorgestellt. Der zweite Algorithmus ist neu und vereinfacht den Algorithmus von Abello und Dolev.

Beide Algorithmen generieren einen *verteilten Binärzähler* auf einer Kette von Agenten. Jeder globale Zustand entspricht einer Zahl. Der Binärzähler zählt von 0 bis 2^n, wobei n die Anzahl der Agenten in der Kette ist. In jedem Ablauf treten alle Werte zwischen 0 und 2^n immer wieder auf.

Abello und Dolev benutzen ihren Zähler in [AD97], um periodisch ein *Reset*-Signal an alle Agenten zu verschicken. Dieses Reset-Signal teilt allen Agenten mit, daß eine bestimmte Berechnung von vorn beginnen soll. Somit wird eine auf falschen (durch transiente Fehler entstandene) Daten beruhende Berechnung garantiert irgendwann abgebrochen. Abello und Dolev zeigen, daß somit prinzipiell *jeder* Algorithmus zu einem selbststabilisierenden Algorithmus erweitert werden kann.

Wir stellen im folgenden einen Teil des Algorithmus von Abello und Dolev [AD97] vor. Konkret betrachten wir die Realisierung eines verteilten Zählers. Im Originalalgorithmus hat jeder Agent zwei Zählbits; für unsere Zwecke genügt ein Zähler mit einem Bit pro Agent.

Der Algorithmus von Abello und Dolev

Wir gehen von einer bidirektionalen Kette von Agenten (U, r, l, min, max) aus. Jeder Agent $x \in U$ besitzt eine binäre Variable $x.count$. Ein Agent benutzt diese Variable als einen lokalen binären Zähler. Hierfür führen wir die Operation $\oplus$ ein:

Definition 3.7 (Binäre Addition) Die Funktion $\oplus : \mathbb{B} \times \mathbb{B} \longrightarrow \mathbb{B}$ heißt *binäre Addition*. Sie ist für alle $x, y \in \mathbb{B}$ definiert als

$$\oplus(x, y) = (x \wedge \neg y) \vee (\neg x \wedge y)$$

Wir verwenden die Infixnotation $x \oplus y$.

Der Algorithmus basiert auf einem selbststabilisierenden Tokenaustausch, vgl. Abschnitt 3.3.1. Daher betrachten wir Anfangszustände, in denen es genau ein Token gibt. Das Token hat eine Richtung: Es ist entweder in Richtung des Agenten max oder in Richtung des Agenten min unterwegs. Die Richtung des Tokens bestimmt, welcher Agent das Token als nächster empfängt. Das Token trägt darüberhinaus zwei zusätzliche Bits – ein Übertrags-Bit $token.carry$ und ein Reset-Bit $token.reset$.

Der Agent min führt das in Abbildung 3.20 angegebene Protokoll aus [AD97]:

```
receive token from r(min);
if (token.carry = 1 ∧ min.count = 1)
      token.reset := 1;
min.count := min.count ⊕ token.carry;
send token to r(min);
```

Abbildung 3.20: Protokoll für den Agenten min

Jeder Agent $x \in U \setminus \{min, max\}$ führt zwei Teilprotokolle aus. Eines nach dem Erhalt eines Tokens in Richtung max (siehe Abbildung 3.21) und eines nach dem Erhalt eines Tokens in Richtung min. Für diesen Fall geben wir zunächst in Abbildung 3.22 das Originalprotokoll aus [AD97] an. Der Pseudocode in [AD97] hat eine ungewöhnliche Semantik. Sie weicht von der üblichen Semantik von Pseudocode ab: So ist z. B. der Wert von $token.carry$, der in der sechsten Zeile in der Zuweisung verwendet wird, nicht unbedingt der Wert, der $token.carry$ in Zeile 3 bzw. 5 zugewiesen wird. Dies liegt daran, daß einige Variablen für benachbarte Agenten *sichtbar* sind. Das Lesen von $token.carry$ bedeutet das Lesen einer Variablen des rechten Nachbarn. Schreiben von $token.carry$ bedeutet das Schreiben einer eigenen lokalen Variablen, die der linke

```
receive token from l(x);
if (token.reset = 1)
      x.count := 0;
send token to r(x);
```

Abbildung 3.21: Protokoll für einen Agenten $x \in U \setminus \{min, max\}$, Token in Richtung max

```
receive token from r(x);
if (token.carry = 1 ∧ x.count = 1)
      token.carry := 1;
else
      token.carry := 0;
x.count := x.count ⊕ token.carry;
send token to l(x);
```

Abbildung 3.22: Protokoll für einen Agenten $x \in U \setminus \{min, max\}$, Token in Richtung min ([AD97])

Nachbar lesen kann. Da die in [AD97] gewählte Pseudocode-Semantik kontraintuitiv ist, geben wir in Abbildung 3.23 eine neue Version des Protokolls an, die der üblichen Semantik folgt.

```
receive token from r(x);
if (token.carry = 1)
      if (x.count = 1)
            x.count := 0;
      else
      x.count := 1; token.carry := 0;
send token to l(x);
```

Abbildung 3.23: Protokoll für einen Agenten $x \in U \setminus \{min, max\}$, Token in Richtung min (neu)

Das Protokoll für den Agenten max ist in Abbildung 3.24 angegeben.

In Abbildung 3.25 ist ein Petrinetzmodell dieses Algorithmus angegeben. Wir gehen von einer bidirektionalen Kette von Agenten (U, r, l, min, max) aus. Die Domains der Stellen first und last sind Boolesche Werte. Sie bezeichnen jeweils den lokalen Zählerwert der Agenten min und max. Initial sind diese Werte beliebig. Der Domain der Stelle count besteht aus Paaren. Das erste Element eines Paares auf dieser Stelle bezeichnet einen Agenten $x \in U \setminus \{min, max\}$.

receive token from $l(max)$;
if $(token.reset = 1)$
    $max.count := 0$; $token.reset := 0$;
if $(max.count = 1)$
    $token.carry := 1$;
else
    $token.carry := 0$;
$max.count := 1$; $max.count := max.count \oplus 1$;
send token to $l(max)$;

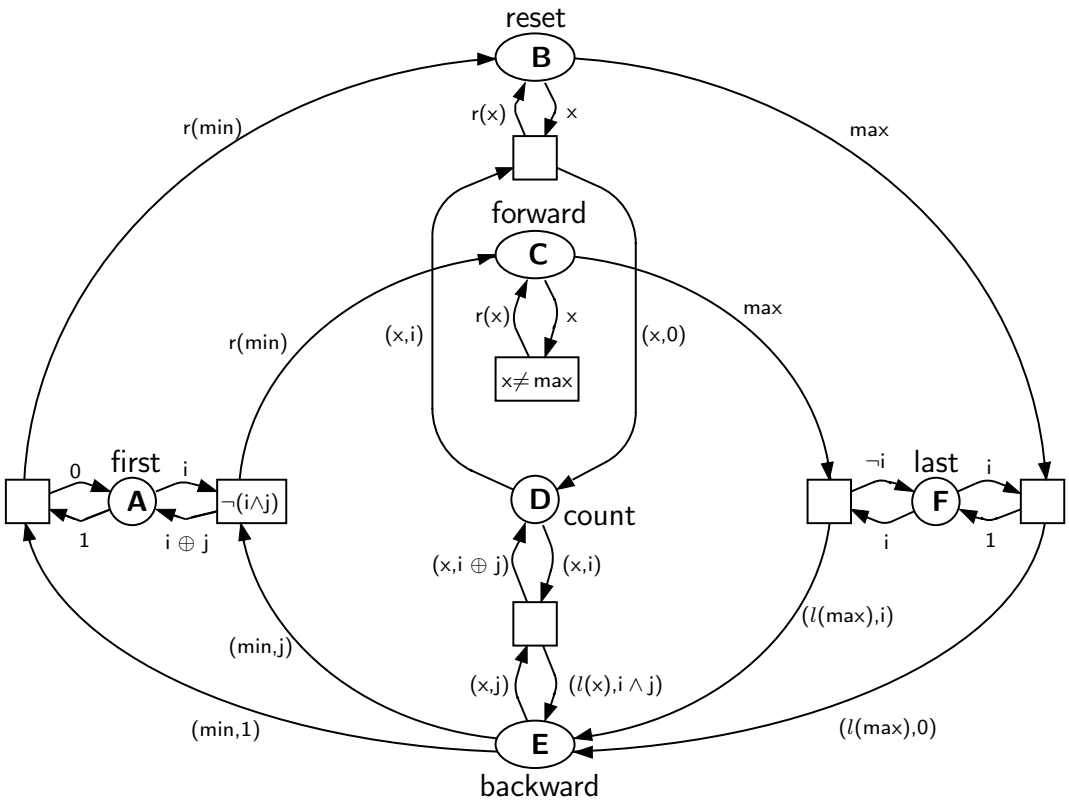
Abbildung 3.24: Protokoll für den Agenten max

Das zweite Element beschreibt den zu x gehörenden lokalen Zählerwert. Initial ist jeder Agent $x \in U \setminus \{min, max\}$ mit genau einem Paar auf dieser Stelle vertreten.

Im Anfangszustand gibt es genau eine Nachricht (das Token), das durch eine Marke auf einer der Stellen reset, forward oder backward repräsentiert wird. Der Domain der Stellen reset und forward ist die Menge der Agenten. Der Domain der Stelle backward besteht aus Paaren. Das erste Element bezeichnet wiederum einen Agenten, das zweite Element ist ein Boolescher Wert. Initial kann nur ein Agent $x \in U \setminus \{max\}$ durch eine Marke auf einer der Stellen reset, forward oder backward repräsentiert sein.

Der Algorithmus Σ_9 arbeitet *deterministisch* und *deadlockfrei*. Der Agent min versendet nach dem Erhalt des Tokens entweder eine reset-Nachricht oder eine forward-Nachricht. Ein Agent $x \in U \setminus \{min, max\}$ sendet nach Erhalt einer reset-Nachricht oder forward-Nachricht diese an seinen rechten Nachbarn weiter. Im Fall einer reset-Nachricht setzt er seinen lokalen Zähler auf den Wert 0 zurück. Der Agent max generiert aus jeder erhaltenen Nachricht eine neue backward-Nachricht an seinen linken Nachbarn. Bei einer reset-Nachricht setzt max seinen lokalen Zähler auf 1. Die generierte backward-Nachricht *bekommt einen Übertrag* (ihr zweiter Wert ist 1), wenn max bisher den Zählerstand 1 hatte und eine forward-Nachricht erhält. Ein Agent $x \in U \setminus \{min, max\}$ addiert nach dem Erhalt einer backward-Nachricht den erhaltenen Wert 0 oder 1 zu seinem lokalen Zählerstand hinzu und versendet eine neue Nachricht an seinen linken Nachbarn (gegebenenfalls ebenfalls mit einem Übertrag). Wenn min einen Übertrag erhält und selbst den Wert 1 hat, dann läuft der Zähler über – min generiert eine reset-Nachricht.

Abbildung 3.26 zeigt ein Ablaufstück für ein System mit drei Agenten $u_0 = min, u_1$ und $u_2 = max$. Die Stellen forward und backward sind hierbei durch for und back abgekürzt. In diesem Ablaufstück zählt der binäre Zähler von 000



spec (U,r,l,min,max) : bidirectional chain
sort value = site × bool
const A, F : bool
const B, C : set of sites
const D, E : set of values
var x : site

var i, j : bool
$B \subseteq U \backslash \{min\}$
$C \subseteq U \backslash \{min\}$
$D_1 = U \backslash \{min,max\}$
$E_1 \subseteq U \backslash \{max\}$
$\exists x \in U : B + C + E_1 = [x]$

Abbildung 3.25: Σ_9 – Verteilter Zähler in einer Kette

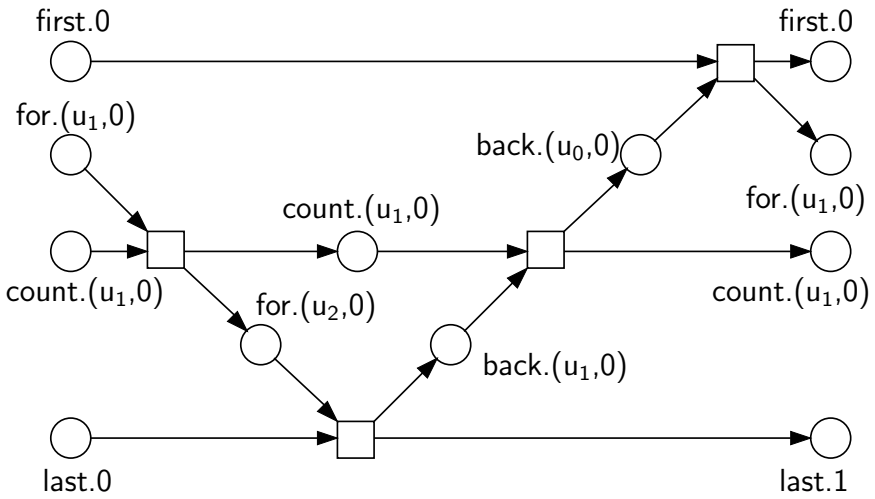


Abbildung 3.26: Ein Ablaufstück von Σ_9

bis 001. Hierzu wandert das Token einmal von *min* nach *max* und zurück.

Wir werden im folgenden die wesentlichen Eigenschaften des Algorithmus Σ_9 spezifizieren. Hierzu beschreiben wir zunächst formal, was ein *globaler Zählerstand* ist. Bei einem unverteilten binären Zähler können sich bei einem Übergang zwischen zwei Zählerständen mehrere *lokale* Zählerstände *gleichzeitig* ändern (z. B. beim Übergang vom Zählerstand 011 zum Zählerstand 100). In einem verteilten System ist dies jedoch unmöglich, da die einzelnen Agenten unabhängig voneinander arbeiten. So gibt es bei einem verteilten Zähler oft einen Zwischenzustand, in dem ein Agent seinen Zählerstand bereits verändert hat, ein anderer Agent jedoch noch nicht. Der Wert des Zählers in einem globalen Zustand hängt daher von lokalen Zählerständen *und* einem eventuellen Übertrag ab:

Definition 3.8 (Zählerstand) Sei n die Kardinalität von U. Seien $x_1 = min$ und $x_i = r(x_{i-1})$, für $1 < i \leqslant n$.

Folgende Terme $\alpha'(x)$ beschreiben den *lokalen Zählerstand eines Agenten* $x \in U$:

i. $\alpha'(x_1) = \mathsf{first}[1] + \mathsf{backward}[(x_1, 1)]$,

ii. $\alpha'(x_i) = \mathsf{count}[(x_i, 1)] + \mathsf{backward}[(x_i, 1)]$, für $1 < i < n$,

iii. $\alpha'(x_n) = \mathsf{last}[1]$.

Der *globale Zählerstand* α ist definiert durch $\alpha = \sum_{i=1}^{n} 2^{n-i}\alpha'(x_i)$.

α ist in obiger Definition ein Term, der in jedem Zustand von Σ_9 zu einer natürlichen Zahl ausgewertet werden kann. Insbesondere gibt es globale Systemzustände, deren globaler Zählerstand größer als 2^n ist (z. B. wenn *min* den lokalen Zählerstand 2 und alle anderen Agenten den lokalen Zählerstand 1 haben). Solche Systemzustände betrachten wir als unzulässig. Wir spezifizieren im folgenden die wichtigen Eigenschaften von Σ_9 .

Die Sicherheitseigenschaft beschreibt, daß sich zwei Zählerstände wechselseitig ausschließen:

$$\Sigma_9 \models \forall h, k \in \mathbb{N} : \Box((\alpha = h \wedge \alpha = k) \Rightarrow h = k) \tag{3.15}$$

Die erste Lebendigkeitseigenschaft beschreibt, daß in jedem Ablauf jeder zulässige Zählerstand unendlich oft erreicht wird:

$$\Sigma_9 \models \forall k \in \{0, \ldots, 2^n\} : \mathit{true} \mapsto \alpha = k \tag{3.16}$$

Die zweite Lebendigkeitseigenschaft beschreibt, daß in jedem Ablauf schließlich nur noch zulässige Zählerstände auftreten:

$$\Sigma_9 \models \Diamond \Box\, \alpha \leqslant 2^n \tag{3.17}$$

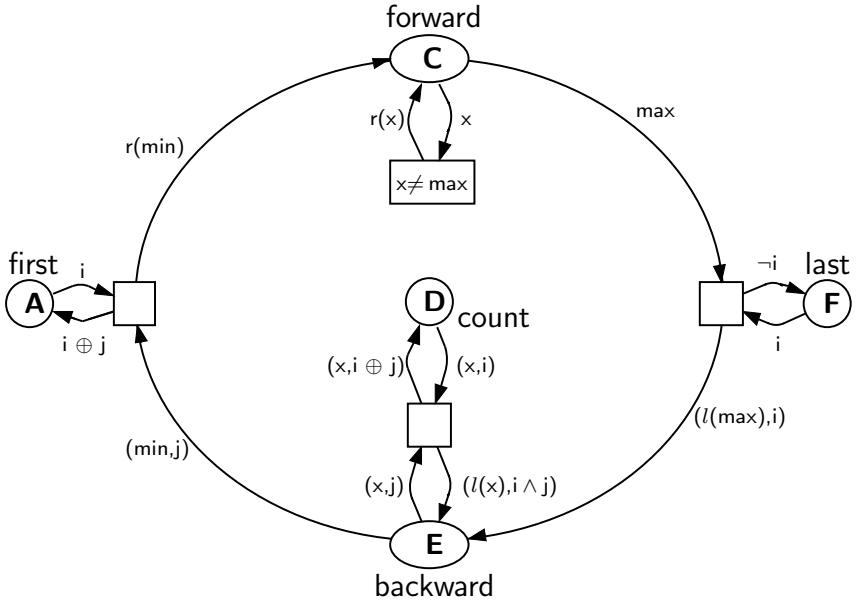
Ein neuer Algorithmus

Wir stellen im folgenden einen neuen Algorithmus zum verteilten Zählen vor. Die Idee dieses Algorithmus lehnt sich an den Algorithmus Σ_9 des vorherigen Abschnitts an. Sie basiert auf folgenden Beobachtungen:

1. In jedem Ablauf stabilisiert sich Σ_9 . Also werden schließlich auch *Reset*-Signale nur noch in zulässigen Zuständen generiert. In einem zulässigen Zustand impliziert ein *Reset*-Signal, daß der lokale Zählerstand jedes Agenten gleich 0 ist. Im Algorithmus von Abello und Dolev wird der Zähler jedoch auch in diesem Fall explizit auf 0 gesetzt.

2. In ihrem Korrektheitsbeweis gehen Abello und Dolev von der Annahme aus, daß die Korrektheit des verteilten Zählers offensichtlich ist. Der Versuch, die Korrektheit formal zu beweisen, wird jedoch dadurch erschwert, daß das Zählen und der Rücksetzmechanismus ineinander verstrickt sind. Die Korrektheit des gesamten Algorithmus in [AD97] hängt jedoch im wesentlichen davon ab, daß immer wieder *Reset*-Signale generiert werden.

Wir entwickeln nun einen selbststabilisierenden Zähler. Er kann *Reset*-Signale generieren, ist aber in seiner Funktionalität unabhängig von diesen.



spec (U,r,l,min,max) : bidirectional chain
sort value = site × bool
const A, F : bool
const C : set of sites
const D, E : set of values
var x : site
var i, j : bool
$C \subseteq U \setminus \{min\}$
$D_1 = U \setminus \{min,max\}$
$E_1 \subseteq U \setminus \{max\}$
$\exists x \in U : C + E_1 = [x]$

Abbildung 3.27: Σ_{10} – Verteilter Zähler in einer Kette

Ein solcher Algorithmus erleichtert (bzw. ermöglicht) eine kompositionale Verifikation (siehe [DIM93, GH91, Sto93, SS94]). Ein Petrinetzmodell unseres Algorithmus ist in Abbildung 3.27 angegeben. Das Petrinetzmodell ist syntaktisch ein Teilnetz des Petrinetzmodells in Abbildung 3.25[5]. Wir beschreiben daher im wesentlichen nur die Unterschiede zum Algorithmus von Abello und Dolev. Wir gehen wiederum von einer bidirektionalen Kette von Agenten (U, r, l, min, max) aus. Im Anfangszustand wird das Token auf einer der Stellen forward oder backward repräsentiert. Wie bisher sendet jeder Agent $x \in U \setminus \{min, max\}$ jede erhaltene forward-Nachricht nach rechts weiter. Jede backward-Nachricht mit Übertrag wird benutzt, um den lokalen Zähler zu inkrementieren. Der Agent min generiert ausschließlich noch forward-

[5]Wir haben bei einer Transition die Schaltbedingung weggelassen, formal also zu *true* geändert, vgl. Notation 2.55.

Nachrichten. (Er könnte natürlich auch eine spezielle forward-Nachricht als *Reset*-Nachricht deklarieren, jedoch wird diese Nachricht von jedem Agenten, soweit es den Zähler betrifft, wie eine normale forward-Nachricht behandelt.)

Wir können Definition 3.8 des globalen Zählerstandes α auch für diesen Algorithmus übernehmen. Wir fordern weiterhin dieselben Eigenschaften, wie für den Algorithmus von Abello und Dolev. Wir werden ihre Gültigkeit in Abschnitt 5.4 beweisen.

Die Sicherheitseigenschaft beschreibt, daß sich zwei Zählerstände wechselseitig ausschließen:

$$\Sigma_{10} \models \forall h, k \in \mathbb{N} : \Box((\alpha = h \wedge \alpha = k) \Rightarrow h = k) \tag{3.18}$$

Die erste Lebendigkeitseigenschaft beschreibt, daß in jedem Ablauf jeder zulässige Zählerstand unendlich oft erreicht wird:

$$\Sigma_{10} \models \forall k \in \{0, \ldots, 2^n\} : \mathit{true} \mapsto \alpha = k \tag{3.19}$$

Die zweite Lebendigkeitseigenschaft beschreibt, daß in jedem Ablauf schließlich nur noch zulässige Zählerstände auftreten:

$$\Sigma_{10} \models \Diamond \Box\, \alpha \leqslant 2^n \tag{3.20}$$

Wir werden die Gültigkeit der Eigenschaften von Σ_{10} in Abschnitt 5.4 beweisen.

4 Verifikation selbststabilisierender Algorithmen

In diesem Kapitel führen wir die wesentlichen Techniken ein, die wir in den Beweisen in Kapitel 5 für unsere Beweise verwenden werden. Wir kombinieren hierbei Petrinetz-spezifische Beweistechniken mit solchen, die allgemein in temporaler Logik gelten, oder sich in der Literatur über selbststabilisierende Algorithmen als geeignet herausgestellt haben.

In Abschnitt 4.1 betrachten wir Techniken zum Nachweis von Sicherheitseigenschaften, in unserem Fall zum Nachweis von Invarianten. In Abschnitt 4.2 betrachten wir korrespondierende Techniken für den Nachweis von Lebendigkeitseigenschaften, konkret für Leads-to- und Stabilisierungseigenschaften. Beweismuster (Abschnitt 4.3) sind eine Möglichkeit, die Gültigkeit einfacher Leads-to-Eigenschaften syntaktisch abzuleiten. Zum Abschluß stellen wir in Abschnitt 4.4 eine neue Methode für die Strukturierung von Beweisen für selbststabilisierende Algorithmen vor. Unsere Beweise in Kapitel 5 folgen dieser Struktur.

4.1 Sicherheitseigenschaften

Wir führen im folgenden zwei wesentliche Techniken für den Nachweis von Invarianten ein: *Stelleninvarianten* und *Zusicherungen*. Unser Konzept von Stelleninvarianten basiert auf einer Termdarstellung (vergleiche [Jen92]). Dies erleichtert einerseits den Übergang zu Systemeigenschaften und ist andererseits ein allgemeineres Konzept als die Darstellung durch einen Stellenvektor (z. B. in [Rei98]), da Schaltbedingungen in unserer Darstellung nicht ignoriert werden. Da wir als Sorte einer Stelleninvariante beliebige kommutative Monoide zulassen, umfaßt unser Konzept auch *Modulo-Invarianten* aus [DNR96]. In einem allgemeineren Rahmen wird dies in [KV98b] diskutiert. Wir folgen den Definitionen in [WWV+97].

Definition 4.1 (Kommutativer Monoid) Eine Sorte A einer Struktur $\mathcal{S}$ ist ein *kommutativer Monoid*, gdw. eine Konstante $0_A \in A$ sowie eine kommutative und assoziative Funktion $+_A$ mit der Stelligkeit $A \times A \longrightarrow A$ in $\mathcal{S}$ existieren, so daß für alle $a \in A$ gilt: $+_A(0_A, a) = a$.

Wir verwenden im folgenden die Infix-Notation für $+_A$.

Beispiel 4.2 Wir werden im folgenden zwei verschiedene kommutative Monoide betrachten: Die Menge der natürlichen Zahlen (mit der Konstante 0 und der Addition) sowie Multimengen von Agenten (mit der leeren Multimenge und der Multimengen-Addition).

Eine Stelleninvariante bildet jeden Zustand in einen kommutativen Monoid ab. Technisch stellen wir diese Abbildung als einen linearen Term dar.

Definition 4.3 (Gewichtung, lineare $\sim$) Sei Σ ein Systemnetz einer Termdarstellung über der Multimengen-Struktur $\mathcal{S}$. Sei $X = (X_0, \ldots, X_n)$ eine Familie $\mathcal{S}$-sortierter Variablen, so daß alle Variablen von X nicht in P_Σ sind und für keine Transition $t \in T_\Sigma$ in X_t vorkommen.

Ein Term $u \in \mathcal{T}_A(X_0, \ldots, X_n, P_\Sigma)$ heißt *Gewichtung*, gdw. A ein kommutativer Monoid ist. Eine Gewichtung $u \in \mathcal{T}_A(X_0, \ldots, X_n, P_\Sigma)$ ist *linear*, gdw. für zwei Zustände C und C' von Σ sowie jede Variablenbelegung β gilt: $\beta_{\mathcal{T},C+C'}(u) = \beta_{\mathcal{T},C}(u) +_A \beta_{\mathcal{T},C'}(u)$.

Eine *Termsubstitution* beschreibt die Auswirkung des Schaltens einer Transition auf eine Gewichtung, indem Stellenvariablen durch die entsprechenden Kantenbeschriftungen ersetzt werden.

Definition 4.4 (Termsubstitution) Für eine Transition $t \in T_\Sigma$ eines Systemnetzes Σ einer Termdarstellung definieren wir die Termsubstitution ${}^\bullet\pi_t : \mathcal{T}_A(X_0, \ldots, X_n, P_\Sigma) \longrightarrow \mathcal{T}_A(X)$, indem wir in einem Term jedes Auftreten einer Stellenvariable $p \in P_\Sigma$ durch den Multimengen-Term $m_{(p,t)}$ ersetzen. Die Substitution $\pi_t^\bullet : \mathcal{T}_A(X_0, \ldots, X_n, P_\Sigma) \longrightarrow \mathcal{T}_A(X)$ ersetzt jedes Auftreten einer Stellenvariable $p \in P_\Sigma$ in einem Term durch $m_{(t,p)}$.

Definition 4.5 (Stelleninvariante) Seien Σ ein Systemnetz einer Termdarstellung über der Multimengen-Struktur $\mathcal{S}$ und u eine lineare Gewichtung. u ist eine *Stelleninvariante* von Σ, gdw. für jede Transition t und jede Variablenbelegung β der Boolesche Term $g_t \Rightarrow \left({}^\bullet\pi_t(u) = \pi_t^\bullet(u)\right)$ zu *true* ausgewertet wird.

Beispiel 4.6 Für das Systemnetz Σ_2 aus Abschnitt 1.3 ist der Term $u = |\mathsf{token}_l + \mathsf{pending}_l|$ eine Stelleninvariante. u ist eine lineare Gewichtung, die in den kommutativen Monoid der natürlichen Zahlen abbildet. Um zu überprüfen, daß u eine Stelleninvariante ist, müssen wir die Gültigkeit der folgenden Gleichungen in der Struktur (d. h. für jede Belegung der Variablen x) beweisen:

$$\begin{aligned}
|[x] + [\,]| &= |[\,] + [\neg x]|,\\
|[\,] + [x]| &= |[x] + [\,]|,\\
|[\,] + [\neg x]| &= |[\,] + [\neg x]| \text{ und}\\
|[\,] + [\,]| &= |[\,] + [\,]|.
\end{aligned}$$

Diese Gleichungen gelten offensichtlich für beliebige Multimengen.

Intuitiv beschreiben Stelleninvarianten, daß sich beim Schalten von Transitionen der Wert einer Gewichtung nicht ändert. Dies bedeutet, daß in jedem erreichbaren Zustand dieser Wert dem Wert des Anfangszustands gleicht. Anders herum ist jeder Zustand, für den die Gewichtung einen anderen Wert erzeugt, nicht erreichbar.

Satz 4.7 Wenn u eine Stelleninvariante der Sorte A eines Systemnetzes Σ mit dem Anfangszustand C_I ist und $C_I \models u = k$ für eine Konstante $k \in A$ gilt, dann gilt auch $\Sigma \models \Box(u = k)$.

Ein Beweis eines allgemeineren Satzes ist in [KV98b] angegeben. Wir werden in unseren Beweisen eine Stelleninvariante u und ihren Wert im Anfangszustand k nicht mehr voneinander trennen. Wir sagen der Einfachheit halber $u = k$ ist eine Stelleninvariante.

Es gibt Invarianten, die nicht ausschließlich mit Stelleninvarianten beweisbar sind [Wal95]. Für diese Invarianten benötigen wir eine allgemeinere Technik – *Zusicherungen*. Eine Zusicherung ist eine Zustandseigenschaft, die im Anfangszustand eines Systemnetzes Σ gilt und invariant gegenüber dem Schalten aller Transitionen von Σ ist.

Definition 4.8 (Zusicherung) Eine Zustandseigenschaft φ eines Systemnetzes Σ ist eine *Zusicherung*, gdw. für alle Schritte $(C, (t, m), C')$ von Σ gilt: Wenn $C \models \varphi$ gilt, dann gilt auch $C' \models \varphi$.

Satz 4.9 Wenn φ eine Zusicherung eines Systemnetzes Σ mit dem Anfangszustand C_I ist, dann gilt $\Sigma \models \Box\,\varphi$ gdw. $C_I \models \varphi$ gilt.

Der Beweis ist offensichtlich mit Definition 2.63 und Definition 2.20. Unsere Definition ist semantisch; eine syntaktische Charakterisierung ist in [WWV+97] dargestellt.

4.2 Lebendigkeitseigenschaften

Im folgenden werden wir wesentliche Techniken für den Nachweis von Lebendigkeitseigenschaften einführen. *Stabilisierungsfunktionen* (*variant functions*, [Kes88]) sind als die wichtigste Verifikationstechnik für selbststabilisierende Algorithmen bekannt. Sie spielen in fast jedem Korrektheitsbeweis eine Rolle.

Die weitere Technik ist eine semantische Regel zur Verifikation einfacher *Leads-to*-Eigenschaften. Diese Regel basiert wesentlich auf [RKV$^+$96]. Wir setzen einfache Leads-to-Eigenschaften mit *Beweisgraphen* zu komplexeren Beweisen zusammen. Wir verwenden *erweiterte Beweisgraphen* [WWV$^+$97]. Eine einfache Syntax-basierte Möglichkeit zum „Ablesen“ von Leads-to-Eigenschaften stellen wir im Abschnitt 4.3 vor.

Definition 4.10 (Fundierte Menge) Eine Menge M ist *fundiert* bezüglich einer Relation $\succ \subseteq M \times M$, gdw. es keine unendliche Sequenz $(m_0 \succ m_1 \succ \dots)$ von Elementen aus M gibt.

Zum Beispiel bilden die natürlichen Zahlen zusammen mit der „größer als“-Relation eine fundierte Menge.

Definition 4.11 (Stabilisierungsfunktion) Seien Σ ein Systemnetz einer Termdarstellung über der Multimengen-Struktur $\mathcal{S}$ und u eine Gewichtung, deren Sorte eine bezüglich $\succ$ fundierte Menge ist. u ist eine *Stabilisierungsfunktion* von Σ, gdw. für alle Schritte $(C, (t, m), C')$ von Σ gilt: Wenn $C \models u = k$ und $C' \models u = k'$ gilt, dann gilt nicht $k' \succ k$.

Satz 4.12 Wenn u eine Stabilisierungsfunktion der Sorte A eines Systemnetzes Σ mit dem Anfangszustand C_I ist, dann gilt $\Sigma \models \exists k \in A : \Diamond\Box(u = k)$.

Der Beweis ist offensichtlich. Eine syntaktische, eingeschränkte Charakterisierung von Stabilisierungsfunktionen (ähnlich den Stelleninvarianten) ist in [KV98b] angegeben. Wir benötigen hier jedoch die allgemeine Form.

Beispiel 4.13 In unserem Beispiel aus Abschnitt 1.3 definieren wir eine Funktion *both*, die den Wert 1 liefert, falls beide Agenten im Zustand pending sind, und 0 sonst. Nun ist $|\mathsf{token}_l| + \mathit{both} + |\mathsf{token}_r|$ eine Stabilisierungsfunktion. Dies kann einfach mit Stelleninvarianten nachgewiesen werden. Es gilt daher, daß es in jedem Ablauf ein Suffix gibt, in dem die Stabilisierungsfunktion konstant bleibt: $\Sigma_2 \models \exists\, n \in \mathbb{N} : \Diamond\Box(|\mathsf{token}_l| + \mathit{both} + |\mathsf{token}_r| = n)$. Diese Stabilisierungsfunktion kann in jedem erreichbaren Zustand zwei Werte annehmen: 1 oder 2. Der Wert 2 wird nur angenommen, wenn beide Agenten im Zustand token sind. Die verbleibenden Beweisschritte zum Nachweis der Eigenschaft (1.3) sind offensichtlich.

Notation 4.14 Für einen Zustand C eines Systemnetzes Σ ist die Zustandseigenschaft φ_C eine abkürzende Schreibweise für $\bigwedge_{p \in P_\Sigma} p\big(C(p)\big)$. Sie beschreibt die Menge aller Zustände, in denen mindestens die durch C beschriebenen Marken vorhanden sind.

Definition 4.15 (Aktivierender Zustand, Fortschrittsmenge) Wir definieren für einen Zustand C eines Systemnetzes Σ:

i. C ist *aktivierend*, gdw. in C eine Aktion von Σ aktiviert ist.

ii. C *blockiert* eine Aktion (t, m) von Σ, gdw. es eine Stelle $p \in P_\Sigma$ gibt, mit $\Sigma \models \Box\big(\varphi_C \Rightarrow \neg p(m(p,t))\big)$.

iii. Eine Aktion (t, m) ist im *Wirkungsbereich* von C, gdw. es eine Stelle $p \in {}^\bullet t$ mit dem Domain A_p sowie ein $a \in A_p$ gibt, mit $C(p)[a] > 0$ und $m(p,t)[a] > 0$.

iv. Eine Menge M von Aktionen von Σ heißt *Fortschrittsmenge* von C, gdw. $M \neq \emptyset$ und C alle Aktionen im Wirkungsbereich von C blockiert, die nicht in M sind.

Beispiel 4.16 Wir beziehen uns wiederum auf unser einführendes Beispiel aus Abschnitt 1.3 und betrachten einen Zustand C, in dem eine Marke 0 auf der Stelle token_l liegt. Dieser Zustand ist aktivierend, da es eine aktivierte Transition t im Nachbereich gibt. Weiterhin blockiert dieser Zustand das Schalten von t im Modus $x = 1$. (Dies kann einfach aus einer Stelleninvariante abgeleitet werden.) Es gibt keine weitere Aktion im Wirkungsbereich von C. Die Transition t im Modus $x = 0$ ist daher die einzige Aktion in der Fortschrittsmenge von C.

Definition 4.17 (Effekt einer Aktion) Für ein Systemnetz Σ ist der *Effekt* einer Aktion (t, m) derjenige Zustand $\mathit{eff}(t, m)$, der für alle $p \in P_\Sigma$ durch $\mathit{eff}(t, m)(p) = m(t, p)$ definiert ist.

Satz 4.18 Für ein Systemnetz Σ, einen aktivierenden Zustand C und eine Fortschrittsmenge M von C gilt:

$$\Sigma \models \varphi_C \mapsto \bigvee_{(t,m) \in M} \varphi_{\mathit{eff}(t,m)}$$

Beweis: Sei C_i eine beliebiger Zustand eines Interleavings $R = \big(C_0, (t_0, m_0), C_1, \ldots\big)$ von Σ. Wenn $C_i \models \neg\varphi_C$, dann gilt $C_i \models \varphi_C \mapsto \bigvee_{(t,m)\in M} \varphi_{\mathit{eff}(t,m)}$ trivialerweise. Anderenfalls gilt $C_i \models \varphi_C$. Es genügt zu zeigen, daß irgendwann im weiteren Verlauf von R eine Aktion aus M eintritt. Da C nach Voraussetzung aktivierend ist, ist auch C_i aktivierend. Also gibt es eine Aktion (t, m), die in C und in C_i aktiviert ist.

Angenommen, im weiteren Verlauf von R tritt keine Aktion der Fortschrittsmenge M ein. Dann ist (t, m) in allen C_j mit $i \leqslant j$ aktiviert. Dann muß aber wegen der Forschrittsannahme für R (Definition 2.37) irgendwann eine in Konflikt zu (t, m) stehende Aktion (t', m') eintreten. Dann ist (t', m') im Wirkungsbereich von C und nicht durch C blockiert. Somit ist (t', m') aber auch in M. □

Beispiel 4.19 Wir führen Beispiel 4.16 fort und können mit Satz 4.18 ableiten:

$$\Sigma_2 \models \mathsf{token}_l(0) \mapsto \mathsf{pending}_l(1).$$

Im folgenden Lemma halten wir wesentliche Eigenschaften fest, die allgemein in linear-time temporaler Logik [MP92] gelten:

Lemma 4.20 Seien φ, φ_1 und ψ Systemeigenschaften eines Systemnetzes Σ und $\mathcal{R}$ eine Randomisierungsannahme für Σ.

1. Wenn $\Sigma \models \varphi \Rightarrow \psi$, dann $\Sigma \models \varphi \mapsto \psi$;
2. $\Sigma \models \varphi \mapsto \varphi$;
3. Wenn $\Sigma \models \varphi \mapsto \varphi_1$ und $\Sigma \models \varphi_1 \mapsto \psi$, dann $\Sigma \models \varphi \mapsto \psi$;
4. Wenn $\Sigma \models \varphi \mapsto \psi$ und $\Sigma \models \varphi_1 \mapsto \psi$, dann $\Sigma \models (\varphi \vee \varphi_1) \mapsto \psi$;
5. Wenn $\Sigma \models \Diamond\Box\,\varphi$ und $\Sigma \models \Diamond\Box\,\psi$, dann $\Sigma \models \Diamond\Box(\varphi \wedge \psi)$;
6. Wenn $\Sigma \models \Box(\varphi \Rightarrow \Box\,\varphi)$ und $\Sigma \models \Diamond\,\varphi$, dann $\Sigma \models \Diamond\Box\,\varphi$;
7. Wenn $\Sigma \models \varphi$, dann $\Sigma \models_{\mathcal{R}} \varphi$.

Beweis: Folgt aus Definition 2.63. □

Mit Satz 4.18 können wir Leads-to-Eigenschaften beweisen, die mit dem Eintreten einer Aktion einer Fortschrittsmenge zusammenhängen. Wir benötigen in Beweisen komplexere Leads-to-Eigenschaften, denen Schaltsequenzen zugrundeliegen. Hierbei können wir die Transitivität von Leads-to ausnutzen.

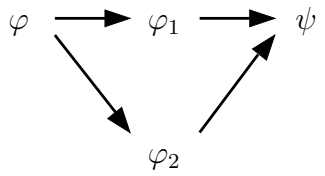


Abbildung 4.1: Ein Beweisgraph für $\varphi \mapsto \psi$

Wir werden häufig einen allgemeineren Fall betrachten: Aus $\Sigma \models \varphi \mapsto \varphi_1 \vee \varphi_2$, $\Sigma \models \varphi_1 \mapsto \psi$ und $\Sigma \models \varphi_2 \mapsto \psi$ folgt $\Sigma \models \varphi \mapsto \psi$. Wir veranschaulichen dies in einem *Beweisgraph*, siehe Abbildung 4.1.

Definition 4.21 (Graph) Für eine Menge N von *Knoten* und $E \subseteq N \times N$ von *Kanten* ist (N, E) ein *Graph*.

i. Ein *Pfad* in (N, E) ist eine Folge $(n_0, \ldots, n_k)$ von Elementen aus N, mit $(n_i, n_{i+1}) \in E$, für $0 \leqslant i < k$.

ii. Ein *Kreis* ist ein Pfad $(n_0, \ldots, n_k)$, mit $n_0 = n_k$ und $k > 0$.

iii. (N, E) ist *azyklisch*, gdw. (N, E) keinen Kreis enthält.

iv. Ein Knoten $n \in N$ ist ein *Anfangsknoten*, gdw. $\{m \in N : (m, n) \in E\} = \emptyset$.

v. Ein Knoten $n \in N$ ist ein *Endknoten*, gdw. $\{m \in N : (n, m) \in E\} = \emptyset$.

Definition 4.22 (Beweisgraph) Für ein Systemnetz Σ ist ein *Beweisgraph* ein azyklischer Graph mit einer Menge von Systemeigenschaften als Knoten. Ein Beweisgraph hat genau einen Anfangsknoten φ und genau einen Endknoten ψ.

Unsere Definition unterscheidet sich von den Definitionen in [Rei98] und [WWV+97]. In [Rei98] werden nur Zustandseigenschaften an Knoten zugelassen. In [WWV+97] werden bei erweiterten Beweisgraphen auch spezielle weitere Systemeigenschaften zugelassen. Unsere Definition hat zwei Vorteile:

1. In [Rei98] werden (insbesondere bei allen selbststabilisierenden Algorithmen) *Klassen* von Beweisgraphen betrachtet, z. B. wenn Induktion eine Rolle spielt. Wir kommen mit einem Beweisgraphen für eine Leads-to-Eigenschaft aus.

2. Für den Beweis einer Leads-to-Eigenschaft, die nur unter einer Randomisierungsannahme gilt, ist die einfache Form der Beweisgraphen unzureichend. Typisch für den Nachweis einer Eigenschaft $\Sigma \models_{\mathcal{R}} \varphi \mapsto \psi$ ist folgende Argumentationskette: Auf jeden φ-Zustand folgend gibt es entweder unendlich viele φ_1-Zustände oder einen ψ-Zustand. Die randomisierungsannahme sichert, daß unendlich viele φ_1-Zustände in einen ψ-Zustand führen, formal: $\Sigma \models \varphi \mapsto \big((\Box \Diamond \varphi_1) \vee \psi\big)$ sowie $\Sigma \models_{\mathcal{R}} (\Box \Diamond \varphi_1) \mapsto \psi$ implizieren $\Sigma \models_{\mathcal{R}} \varphi \mapsto \psi$. Wir werden in Abschnitt 5.1 solche Beweisgraphen zum Nachweis von Systemeigenschaften eines randomisierten Algorithmus verwenden.

Definition 4.23 (Gültigkeit eines Beweisgraphen) In einem Systemnetz Σ *gilt ein Beweisgraph* (N, E), gdw. für alle Knoten $\varphi \in N$, die keine Endknoten sind, gilt:

$$\Sigma \models \varphi \mapsto \bigvee_{(\varphi,\psi)\in E} \psi.$$

Ein Beweisgraph (N, E) *gilt in einem Systemnetz* Σ *unter der Randomisierungsannahme* $\mathcal{R}$, gdw. für alle Knoten $\varphi \in N$, die keine Endknoten sind, gilt:

$$\Sigma \models_{\mathcal{R}} \varphi \mapsto \bigvee_{(\varphi,\psi)\in E} \psi.$$

Satz 4.24 Sei (N, E) ein im Systemnetz Σ gültiger Beweisgraph. Seien φ der Anfangs- und ψ der Endknoten von (N, E). Dann gilt:

$$\Sigma \models \varphi \mapsto \psi.$$

Dieser Satz wurde für Zustandseigenschaften N von Reisig in [Rei98] bewiesen. Der Beweis in [Rei98] gilt genauso auch für Systemeigenschaften N, da dieser Unterschied im Beweis keine Rolle spielt.

Satz 4.25 Sei (N, E) ein im Systemnetz Σ unter einer Randomisierungsannahme $\mathcal{R}$ gültiger Beweisgraph. Seien φ der Anfangs- und ψ der Endknoten von (N, E). Dann gilt:

$$\Sigma \models_{\mathcal{R}} \varphi \mapsto \psi.$$

Der Beweis ist analog. Wir setzen Beweisgraphen auch zur Verifikation anderer Lebendigkeitseigenschaften ein. So ist eine Stabilisierungseigenschaft $\Diamond \Box \varphi$ äquivalent als $\mathit{true} \mapsto \Diamond \Box \varphi$ darstellbar (siehe [MP92]). Wir beginnen den Beweisgraph mit einer bereits bewiesenen Eigenschaft und enden mit dem Knoten $\Diamond \Box \varphi$.

4.3 Beweismuster

In Abschnitt 4.2 gaben wir eine allgemeine semantische Regel für den Beweis von *Leads-to*-Eigenschaften an. Eine vollständig syntaktische Charakterisierung dieser Regel ist in [WWV+97] dargestellt. In konkreten Beweisen erscheint die Anwendung dieser Regel jedoch als zu aufwendig, da in der allgemeinen Regel sehr viele Spezialfälle ausgeschlossen werden müssen. Reisig schlug daher in [Rei98] die Verwendung von *Beweismustern* (pick-up patterns) vor. Er gab zwei Beweismuster (Alternative und Synchronisation) für Leads-to-Eigenschaften an. Wir werden die Idee von Reisig aufgreifen und in diesem Abschnitt für selbststabilisierende Algorithmen spezialisieren und erweitern.

Das generelle Ziel von Beweismustern besteht darin, die Verifikation von Leads-to-Eigenschaften syntaktisch zu charakterisieren. Die syntaktische Charakterisierung liefert ein Muster, d. h. eine Beweisregel, die in einem anderen Systemnetz genau so wieder verwendet werden kann. Die syntaktische Charakterisierung lohnt sich in jedem Fall, da gerade der Übergang von einem Systemnetz zu einer Lebendigkeitseigenschaft erfahrungsgemäß fehleranfällig ist. Die syntaktische Charakterisierung liefert Informationen, welche Invarianten nötig sind, um eine Leads-to-Eigenschaft zu zeigen. Die von unseren Systemnetzen generierten Muster sind komplex; sie werden nur in Ausnahmefällen mehrfach verwendet. Um das Wort „Muster“ zu rechtfertigen, werden wir Gemeinsamkeiten extrahieren, die in mehreren komplexen Mustern vorkommen. Konkret gehen wir in zwei Schritten vor:

1. Wir wandeln unser Systemnetz in ein semantisch äquivalentes Systemnetz um, das syntakisch für Muster handhabbarer ist. Wir erhalten ein *komplexes Muster*.

2. Wir extrahieren Gemeinsamkeiten mehrerer komplexer Muster. Wir nennen die entstehenden Muster *Basismuster*. Diese entsprechen den „pick-up patterns“ von Reisig.

Die Begriffswahl „komplexes Muster“ bzw. „Basismuster“ ist informell und durch intuitive Argumente gerechtfertigt. Formal ist jedes Muster eine Beweisregel. Wir werden feststellen, daß bestimmte Basismuster oft vorkommen, andere, von Reisig vorgeschlagene, jedoch gar nicht. Im Gegensatz zu [Rei98] werden wir beispielsweise kein Muster für eine Alternative angeben, da Nichtdeterminismus *bei der Auswahl* lokaler Aktionen bei selbststabilisiernden Algorithmen typischerweise keine Rolle spielt. (Nichtdeterminismus äußert sich vielmehr in der konkreten Auswirkung einer lokalen Aktion, beispielsweise als

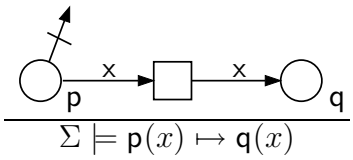
Ergebnis eines Münzwurfs bei randomisierten Algorithmen.) Wir streben also keine formale Vollständigkeit an, sondern extrahieren die Aspekte, die bei Petrinetzmodellen selbststabilisierender Algorithmen eine Rolle spielen.

Wir gehen in diesem Abschnitt wie folgt vor. Wir stellen in Abschnitt 4.3.1 zuerst 10 Basismuster vor, die intuitiv jeweils einen Aspekt der Syntax von Systemnetzen beleuchten. In Abschnitt 4.3.2 stellen wir unsere komplexen Muster vor und wenden sie auf einen unserer Algorithmen an.

4.3.1 Basismuster

Die folgende Beweisregel zeigt ein Beweismuster:

Muster 1 (Grundmuster)



$$\Sigma \models \mathsf{p}(x) \mapsto \mathsf{q}(x)$$

Wir vereinbaren für alle Muster folgende Konventionen:

- Ein aus einer Stelle herausführender durchgestrichener Pfeil bedeutet, daß im Nachbereich dieser Stelle nur die explizit aufgeführten Transitionen liegen.
- Jede Transition hat nur die explizit aufgeführten Eingangskanten. (Sie darf allerdings weitere Ausgangskanten haben.)
- Wir geben Muster in Form einer Beweisregel an, bei der eine Prämisse syntaktische Einschränkungen beschreibt.
- Wir geben für kein Muster explizit einen Korrektheitsbeweis an, da jeder Beweis jeweils einen Spezialfall des Beweises von Satz 4.18 darstellt.

Konkret ist das Grundmuster also auf alle Systemnetze Σ anwendbar, für die gilt, daß $p, q \in P_\Sigma$ und $t \in T_\Sigma$ existieren, mit:

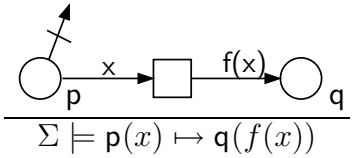
1. $p^\bullet = \{t\}$,
2. $q \in t^\bullet$ und
3. $m_{(p,t)} = m_{(t,q)} = [x]$.

Notation 4.26 Wir legen für diesen Abschnitt eine einheitliche Bedeutung aller in Beweismustern verwendeten Symbole fest:

x, y	sind Variablen der Sorte *site*.
i	ist eine Konstante der Sorte *site*.
f	ist eine Funktion mit der Stelligkeit $\mathsf{f} : \mathit{site} \longrightarrow \mathit{site}$.
n, m	sind Variablen beliebiger Sorte.
$\mathsf{g_t}$, $\mathsf{g_1}$, $\mathsf{g_2}$	sind Variablen für Schaltbedingungen.
u, v, ...	sind Variablen für Terme beliebiger Sorte.

Die folgende Beweisregel zeigt ein Beweismuster für den Nachrichtenaustausch. Diese Situation tritt häufig ein, wenn eine Aktion modelliert wird, in der ein Agent einem benachbarten Agenten eine Nachricht sendet:

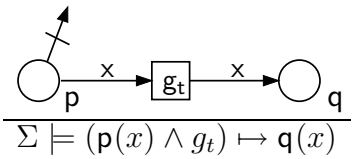
Muster 2 (Nachrichtenaustausch)



$$\Sigma \models \mathsf{p}(x) \mapsto \mathsf{q}(f(x))$$

Die folgende Beweisregel zeigt ein Beweismuster für Schaltbedingungen.

Muster 3 (Schaltbedingungen)



$$\Sigma \models (\mathsf{p}(x) \wedge g_t) \mapsto \mathsf{q}(x)$$

Für das nächste Beweismuster führen wir eine hilfreiche Definition ein. Ein *surjektiver Term* als Kantenbeschriftung einer aus einer Stelle herausführenden Kante bedeutet intuitiv, daß prinzipiell jede Marke auf dieser Stelle über diese Kante konsumiert werden kann:

Definition 4.27 (Surjektiver Term) Sei $\mathcal{S}$ eine Struktur und X eine Familie $\mathcal{S}$-sortierter Variablen.

Ein Term $u \in \mathcal{T}_B(X)$ der Sorte B über X ist *surjektiv*, gdw. zu jedem $b \in B$ eine Variablenbelegung β existiert, mit $\beta_{\mathcal{T}}(u) = b$.

Wir schreiben *surjektiv*(u), falls u surjektiv ist.

Beispiel 4.28 Im Bereich der natürlichen Zahlen gilt *surjektiv*$(x + y)$. Der Term $x + x$ ist nicht surjektiv, da für keine Belegung der Variablen x dieser Term zu ungeraden Werten ausgewertet wird.

Wir geben im folgenden eine hinreichende syntaktische Charakterisierung surjektiver Terme an, die von einer konkreten Struktur unabhängig ist und für unsere Beispiele genügt.

Satz 4.29 Sei $\mathcal{S}$ eine Struktur und X eine Familie $\mathcal{S}$-sortierter Variablen.
Für jeden Term $u \in \mathcal{T}_B(X)$ gilt *surjektiv*(u), falls gilt:

1. in u kommen keine Konstantensymbole vor,

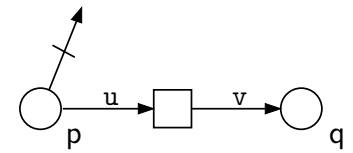
2. jede Variable $x \in X$ kommt höchstens einmal in u vor, und

3. jede in u vorkommende Funktion ist surjektiv.

Beweis: Induktiv über den Aufbau aller Terme der Sorte $\mathcal{T}_B(X)$ können wir für jedes $b \in B$ eine passende Variablenbelegung finden. □

Die folgende Beweisregel zeigt ein Beweismuster für Projektionen. Wir benutzen Projektionen zur einfachen Beschreibung komplexer lokaler Zustände. u und v sind hierbei als variable Kanteninschriften zu lesen (u und v sind also Terme, deren Sorte der Domain der Stellen p bzw. q ist.

Muster 4 (Projektionen)

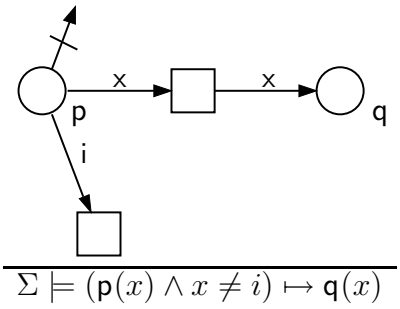


surjektiv(u),
$\mathtt{u}_1 = \mathtt{v}_1 = \mathtt{x}$
$\Sigma \models \mathtt{p}_1(x) \mapsto \mathtt{q}_1(x)$

Wir können diese Regel also z. B. mit $\mathtt{u} = (x, y)$ und $\mathtt{v} = (x, f(y))$ anwenden, vorausgesetzt der Domain der Stellen p und q ist jeweils *site* × *site*.

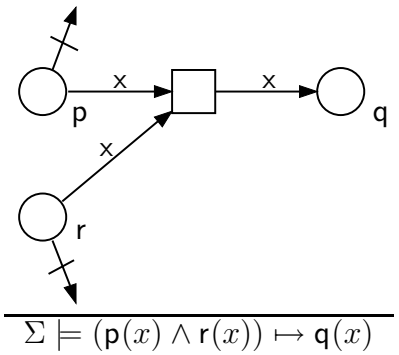
Die folgende Beweisregel zeigt ein Beweismuster für Regelverhalten. Wir sprechen von Regelverhalten, falls wir Aussagen über *alle Agenten bis auf einen,* in unserem Fall i treffen wollen.

Muster 5 (Regelverhalten)



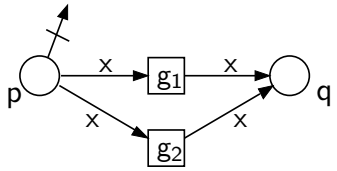
Das folgende Basismuster für Synchronisation stellt wiederum einen einfachen Spezialfall dar; eine Marke auf r kann nur gemeinsam mit einer Marke auf p verbraucht werden. Dieser Spezialfall tritt oft auf; die allgemeine Beschreibung *aller möglichen* Synchronisationsphänomene verursacht jedoch die Komplexität der syntaktischen Beweisregel in [WWV+97].

Muster 6 (Synchronisation)



Die folgende Beweisregel zeigt ein Beweismuster für einen wechselseitigen Ausschluß. Mit diesem Muster beschreiben wir den „Durchschnitt“ zweier Aktionen. Die beiden Aktionen können verschiedene Nebeneffekte haben (z. B. durch beliebige zusätzliche Ausgangskanten); der lokale Zustand des Agenten x wechselt in jedem Fall von p nach q.

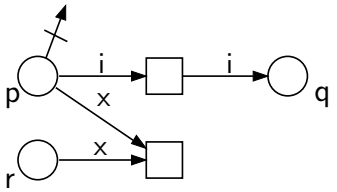
Muster 7 (Wechselseitiger Ausschluß)



$$\frac{\Sigma \models \Box\,(\mathsf{p}(x) \Rightarrow (g_1 \vee g_2))}{\Sigma \models \mathsf{p}(x) \mapsto \mathsf{q}(x)}$$

Die nächste Beweisregel zeigt ein Beweismuster für ein Ausnahmeverhalten. Es stellt das Pendant zu Muster 5 dar.

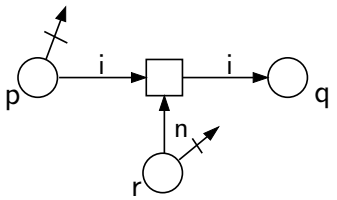
Muster 8 (Ausnahmeverhalten)



$$\frac{\Sigma \models \Box\,\neg\mathsf{r}(i)}{\Sigma \models \mathsf{p}(i) \mapsto \mathsf{q}(i)}$$

Die folgende Beweisregel zeigt ein Beweismuster für erweitertes Ausnahmeverhalten. Wir benutzen dieses Muster, wenn wir eine lokale Variable (r) eines ausgezeichneten Agenten (i) explizit modellieren, wie z. B. im Algorithmus zum verteilten Zählen, siehe Abschnitt 3.3.2.

Muster 9 (Erweitertes Ausnahmeverhalten)



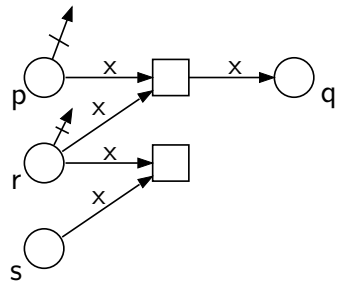
$$\frac{\Sigma \models \Box\,|\mathsf{r}| = 1}{\Sigma \models \mathsf{p}(i) \mapsto \mathsf{q}(i)}$$

In obiger Beweisregel wäre die Prämisse $\Sigma \models \Box\,|\mathsf{r}| \geqslant 1$ hinreichend, der Fall

$|\mathsf{r}| > 1$ tritt üblicherweise jedoch nicht auf (da eine Variable r immer genau einen Wert hat). Die geforderte Invariante impliziert auch die Existenz einer Kante von der Transition zur Stelle r, die wir jedoch gemäß unserer Konvention nicht anzugeben brauchen.

Die folgende Beweisregel zeigt ein Beweismuster für erweiterte Synchronisation. Ein Agent im lokalen Zustand r synchronisiert sich entweder mit der Ressource p oder der Ressource s:

Muster 10 (Erweiterte Synchronisation)



$$\frac{\Sigma \models \Box(\mathsf{p}(x) \Rightarrow \neg \mathsf{s}(x))}{\Sigma \models (\mathsf{p}(x) \wedge \mathsf{r}(x)) \mapsto \mathsf{q}(x)}$$

4.3.2 Komplexe Muster

In diesem Abschnitt stellen wir die komplexen Muster vor, die durch unsere Systemnetze generiert werden. Wir haben bereits erwähnt, daß wir unsere Systemnetze zuvor in semantisch äquivalente Systemnetze umwandeln. Das Basismuster für Projektionen verlangt surjektive Kantenbeschriftungen. Das Basismuster für Ausnahmeverhalten verlangt explizit nicht-surjektive Kantenbeschriftungen. Intuitiv kommen in einigen komplexen Mustern trotzdem beide Basismuster vor. Dies ist jedoch nur scheinbar ein Problem: Wir benutzen einen „Trick“, um nicht-surjektive Kantenbeschriftungen in surjektive Kantenbeschriftungen umzuwandeln. Es gibt zu jedem Systemnetz ein semantisch äquivalentes Systemnetz, in dem nur surjektive Kantenbeschriftungen auftreten.

Wir stellen die komplexen Muster einzeln vor und wenden sie in ihrem Kontext an. Wir erläutern zuvor unseren „Trick“ zur Umwandlung eines nichtsurjektiven Terms in einen surjektiven. Die Konstruktion ist trivial: Wir zeigen sie zuerst an zwei Beispielen. In Abbildung 4.2 ist die Übersetzung einer Konstante angegeben; in Abbildung 4.3 ist die Übersetzung eines komplexeren Terms angegeben.

Abbildung 4.2: Übersetzung eines nicht-surjektiven Terms

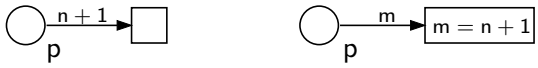


Abbildung 4.3: Übersetzung eines nicht-surjektiven Terms

Im allgemeinen ersetzen wir also jeden Term durch eine neue Variable, die als Sorte den Domain der zugehörigen Stelle hat. Die durch den ursprünglichen Term gegebenen Einschränkungen stellen wir als Schaltbedingung dar.

Ein weiteres Problem ergibt sich, wenn wir das Beweismuster für Projektionen und das Beweismuster für Synchronisation betrachten. In Abbildung 4.4 ist jede einzelne Kantenbeschriftung syntaktisch surjektiv (Satz 4.29). Wie auch

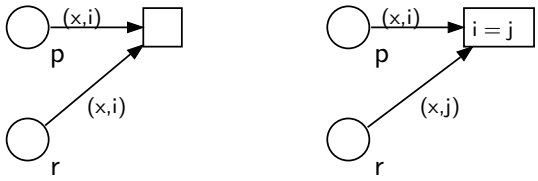


Abbildung 4.4: Synchronisation surjektiver Terme

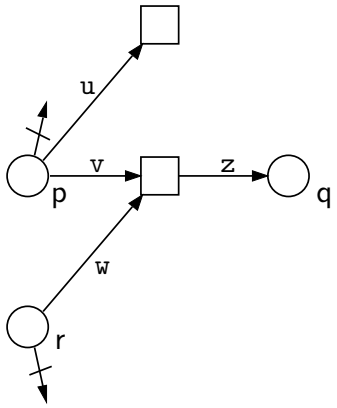
im Beweismuster für Synchronisation werden die Ressourcen auf p und r über die Variable x synchronisiert. Zusätzliche Synchronisation kann jedoch auch über andere Variablen (hier i) entstehen. Wir verwenden hierbei einen ähnlichen Trick: Für jede doppelt auftretende Variable konstruieren wir eine neue Variable (hier j), der wir durch die Schaltbedingung den Wert der ursprünglichen Variablen zuweisen.

Um die Situation im rechten Netz in Abbildung 4.4 von der Situation im linken Netz zu unterscheiden, definieren wir für einen Term $u \in \mathcal{T}_B(X)$ die Menge $Var(u)$ *in* u *vorkommender Variablen* aus X. Im rechten Netz kommt nur die Variable x in beiden Kantenbeschriftungen vor; im linken Netz kommen die Variablen x und i in beiden Kantenbeschriftungen vor.

Definition 4.30 (Vorkommende Variablen) Für einen Term $u \in \mathcal{T}_B(X)$ der Sorte B über X ist $Var(u)$ die kleinste Teilmenge von X mit $u \in \mathcal{T}_B(Var(u))$.

Im folgenden Muster kommen Muster 2 (Nachrichtenaustausch), Muster 4 (Projektionen), Muster 5 (Regelverhalten) sowie Muster 6 (Synchronisation) vor: Alle Agenten außer i benutzen die Ressourcen[1] p und r, um eine Nachricht q zu senden:

Muster 11



$surjektiv(\mathtt{v})$, $surjektiv(\mathtt{w})$,
$\mathtt{v}_1 = \mathtt{w}_1 = \mathsf{x}$,
$Var(\mathtt{v}) \cap Var(\mathtt{w}) = \{\mathsf{x}\}$,
$\mathtt{u}_1 = \mathsf{i}$,
$\mathtt{z}_1 = \mathsf{f(x)}$

$$\Sigma \models (\mathsf{p}_1(x) \wedge \mathsf{r}_1(x) \wedge x \neq i) \mapsto \mathsf{q}_1(f(x))$$

Da Projektionen vorkommen, fordern wir, daß die Terme v und w surjektiv sind und die Projektion auf ihre erste Komponente jeweils x ist. In v und w darf gemeinsam nur die Variable x vorkommen. u beschreibt die Ausnahme vom Regelverhalten; daher muß die Surjektivität von u nicht gefordert werden. Die Projektion auf die erste Komponente von z beschreibt, daß eine Nachricht gesendet wird.

Wir wenden im folgenden dieses Muster an. In Abbildung 4.5 ist ein Ausschnitt unseres Algorithmus zum verteilten Zählen dargestellt (vergleiche Abschnitt 3.3.2). Im Beweis des Algorithmus (in Kapitel 5) wird folgende Eigenschaft eine Rolle spielen:

$$\Sigma_{10} \models (\mathsf{backward}_1(x) \wedge x \neq min) \mapsto \mathsf{backward}_1(l(x)) \tag{4.1}$$

Wir können hierzu Muster 11 einsetzen, mit $\mathsf{p} = \mathsf{q} = \mathsf{backward}, \mathsf{r} = \mathsf{count}$ sowie

[1] Wir benutzen den Begriff Ressource als einen Oberbegriff für einen lokalen Zustand eines Agenten x und Nachrichten von und an x.

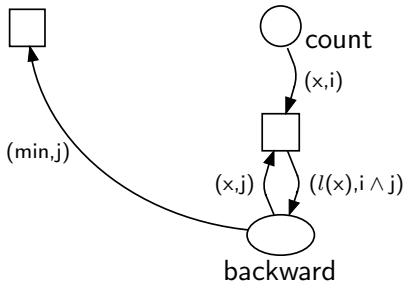


Abbildung 4.5: Ein Ausschnitt von Σ_{10}

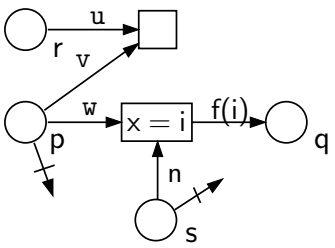
den entsprechenden Belegungen für u, v, w und z. (In diesem Fall sind alle Voraussetzungen bereits erfüllt, wir müssen also kein semantisch äquivalentes Systemnetz erzeugen.) Dies liefert uns:

$$\Sigma_{10} \models (\mathsf{backward}_1(x) \wedge \mathsf{count}_1(x) \wedge x \neq min) \mapsto \mathsf{backward}_1(l(x)) \qquad (4.2)$$

Die gewünschte Eigenschaft (4.1) werden wir mit einer Stelleninvariante ableiten.

Im folgenden Muster kommen Muster 2 (Nachrichtenaustausch), Muster 4 (Projektionen), Muster 8 (Ausnahmeverhalten) und Muster 9 (Erweitertes Ausnahmeverhalten) vor. Das Verhalten des Agenten i stellt eine Ausnahme dar. Er kann auf eine Ressource s zugreifen und versendet eine Nachricht q:

Muster 12



$$\frac{\begin{array}{l} \mathit{surjektiv}(\mathtt{w}), \\ \mathsf{n} \notin \mathit{Var}(\mathtt{w}), \\ \mathtt{w}_1 = \mathtt{v}_1 = \mathtt{u}_1 = \mathsf{x}, \\ \Sigma \models \Box\, |\mathsf{s}| = 1, \\ \Sigma \models \Box\, \neg \mathsf{r}_1(i) \end{array}}{\Sigma \models \mathsf{p}_1(i) \mapsto \mathsf{q}_1(f(i))}$$

Analog zu Muster 11 fordern wir, daß der Term w surjektiv ist und daß die

Variable n in w nicht vorkommt.

Wir beschreiben wiederum die Anwendung des Musters im Beweis des Algorithmus zum verteilten Zählen. In Abbildung 4.6(a) ist ein anderer Ausschnitt von Σ_{10} dargestellt. Wir werden das Muster für den Beweis folgender Eigen-

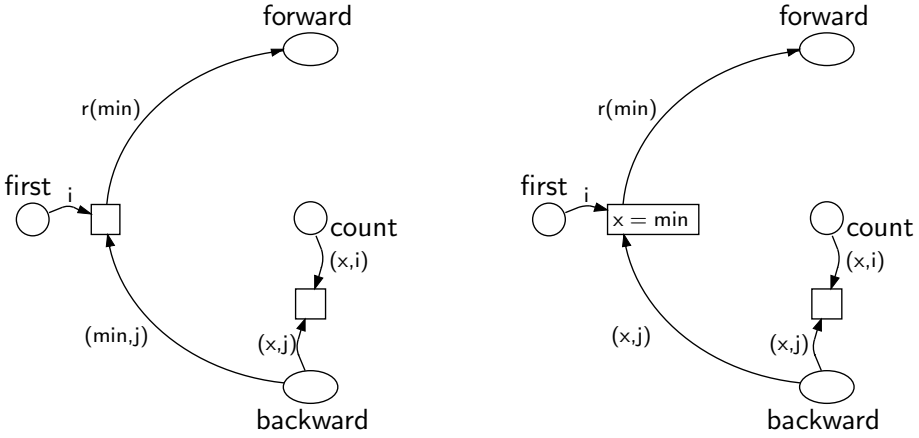


Abbildung 4.6: Ein Ausschnitt von Σ_{10}

schaft verwenden:

$$\Sigma_{10} \models \mathsf{backward_1}(min) \mapsto \exists x \in U : \mathsf{forward}(x) \tag{4.3}$$

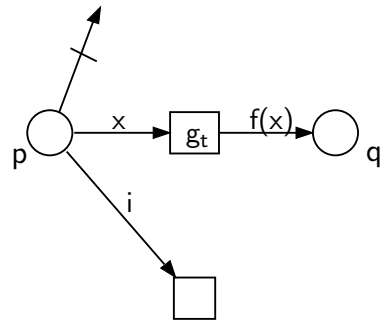
Um unser Beweismuster anzuwenden, konstruieren wir aus dem Ausschnitt in Abbildung 4.6(a) unter Verwendung des oben beschriebenen „Tricks“ einen äquivalenten Ausschnitt (Abbildung 4.6(b)). Nun können wir das Beweismuster anwenden mit $\mathsf{p} = \mathsf{backward}, \mathsf{q} = \mathsf{forward}, \mathsf{r} = \mathsf{count}, \mathsf{s} = \mathsf{first}$ sowie den entsprechenden Belegungen für u, v, w, n, f und i. Dies liefert uns (vorausgesetzt, die im Muster zusätzlich geforderten Invarianten sind bewiesen):

$$\Sigma_{10} \models \mathsf{backward_1}(min) \mapsto \mathsf{forward_1}(r(min)) \tag{4.4}$$

Dies impliziert die gewünschte Eigenschaft (4.3).

Im folgenden Muster kommen Muster 2 (Nachrichtenaustausch), Muster 3 (Schaltbedingungen) und Muster 5 (Regelverhalten) vor. Jeder Agent außer i kann, falls die Schaltbedingung erfüllt ist, eine Nachricht q versenden:

Muster 13



$$\Sigma \models (\mathsf{p}(x) \wedge g_t \wedge x \neq i) \mapsto \mathsf{q}(f(x))$$

In Abbildung 4.7 zeigen wir wiederum den Ausschnitt von Σ_{10}, auf den wir dieses Muster anwenden: Wir werden folgende Eigenschaft beweisen:

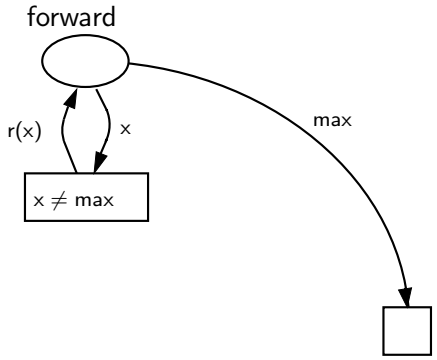


Abbildung 4.7: Ein Ausschnitt von Σ_{10}

$$\Sigma_{10} \models (\mathsf{forward}(x) \wedge x \neq max) \mapsto \mathsf{forward}(r(x)) \tag{4.5}$$

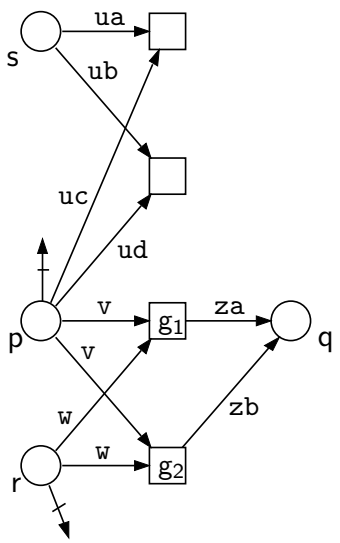
Die Anwendung des Musters mit $\mathsf{p} = \mathsf{q} = \mathsf{forward}$ liefert:

$$\Sigma_{10} \models (\mathsf{forward}(x) \wedge x \neq max \wedge x \neq max) \mapsto \mathsf{forward}(r(x)) \tag{4.6}$$

Dies impliziert die gewünschte Eigenschaft (4.5).

Im folgenden Muster kommen die Basismuster 2 (Nachrichtenaustausch), 4 (Projektionen), 6 (Synchronisation), 7 (Wechselseitiger Ausschluß) und 10 (Erweiterte Synchronisation) vor. Jeder Agent kann eine Nachricht q versenden, falls er die Ressourcen p und r hat:

Muster 14



$$\frac{\begin{array}{l} \mathit{surjektiv}(\mathtt{v}),\, \mathit{surjektiv}(\mathtt{w}), \\ \mathit{Var}(\mathtt{v}) \cap \mathit{Var}(\mathtt{w}) = \{\mathtt{x}\}, \\ \mathtt{ua}_1 = \mathtt{ub}_1 = \mathtt{uc}_1 = \mathtt{ud}_1 = \mathtt{v}_1 = \mathtt{w}_1 = \mathtt{x}, \\ \mathtt{za}_1 = \mathtt{zb}_1 = \mathsf{f}(\mathtt{x}), \\ \Sigma \models \Box\big((\mathsf{p}(\mathtt{v}) \wedge \mathsf{r}(\mathtt{w})) \Rightarrow (g_1 \vee g_2)\big), \\ \Sigma \models \Box\big(\mathsf{r}_1(x) \Rightarrow \neg \mathsf{s}_1(x)\big) \end{array}}{\Sigma \models (\mathsf{p}_1(x) \wedge \mathsf{r}_1(x)) \mapsto \mathsf{q}_1(f(x))}$$

Da wir Projektionen verwenden, fordern wir, daß die Terme v und w surjektiv sind. Da wir die Ressourcen synchronisieren, darf in ihnen nur die Variable x gemeinsam vorkommen. Die zusätzlich geforderten Invarianten spiegeln sich in den Basismustern für den wechselseitigen Ausschluß bzw. die erweiterte Synchronisation wider.

Wir werden dieses Muster im Beweis unseres Algorithmus zur randomisierten Leader Election im unidirektionalen Ring anwenden (vergleiche Abschnitt 5.1). In Abbildung 4.8 ist ein Ausschnitt dieses Algorithmus dargestellt.

Wir möchten folgende Eigenschaft beweisen:

$$\Sigma_4 \models (\mathsf{coin}_1(x) \wedge \mathsf{leader}_1(x)) \mapsto \mathsf{coin}_1(l(x)) \tag{4.7}$$

Wir wenden wieder unsere Konstruktionsvorschrift an, um unser Beweismuster einsetzen zu können. Wir erhalten den semantisch äquivalenten Ausschnitt in Abbildung 4.9. Wir wenden unser Beweismuster an mit $\mathsf{p} = \mathsf{q} = \mathsf{coin}$, $\mathsf{r} = \mathsf{leader}$,

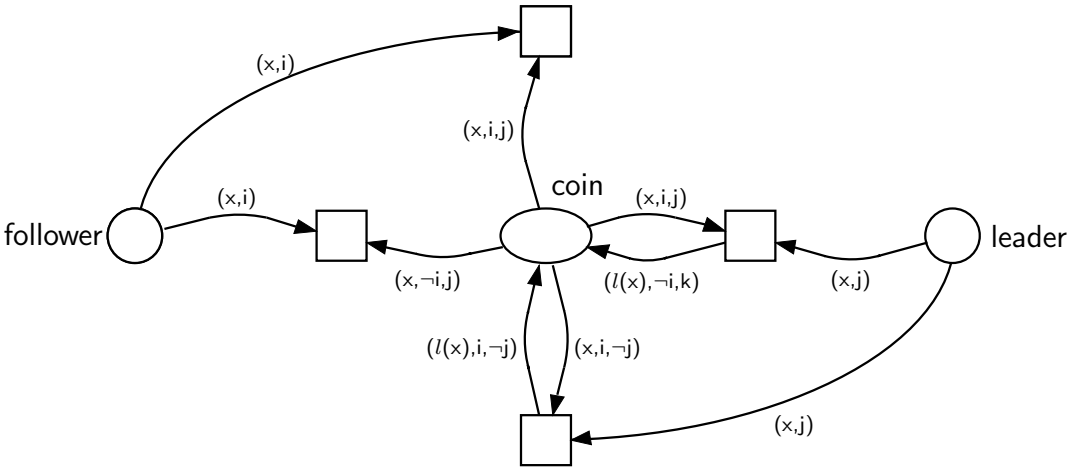


Abbildung 4.8: Ein Ausschnitt von Σ_4

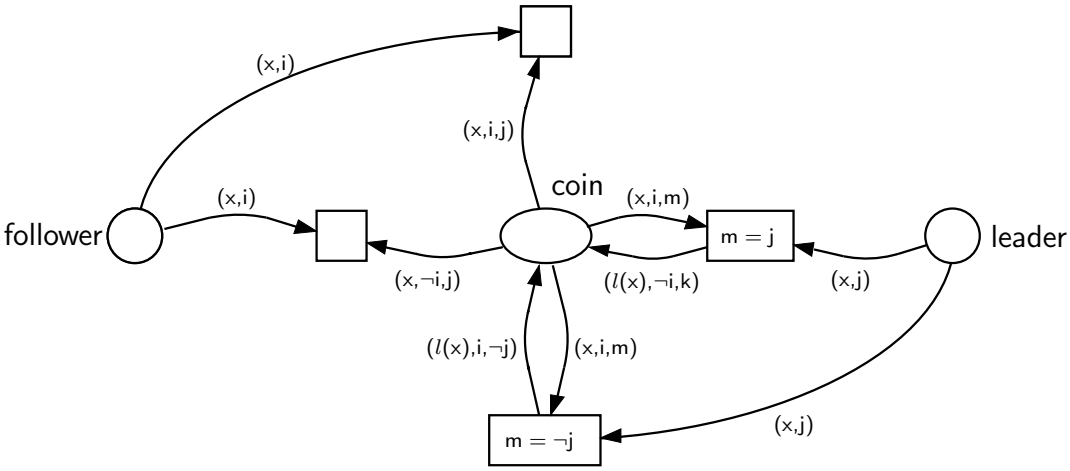


Abbildung 4.9: Ein äquivalenter Ausschnitt von Σ_4

$\mathsf{s} = \mathsf{follower}$ sowie den entsprechenden Belegungen für die anderen Variablen, die sich hieraus ergeben. Wir müssen also zuerst die folgenden Invarianten zeigen:

$$\Sigma_4 \models \Box\left((\mathsf{coin}(x,i,m) \wedge \mathsf{leader}(x,j)) \Rightarrow (m = j \vee m = \neg j)\right) \tag{4.8}$$

sowie

$$\Sigma_4 \models \Box\left(\mathsf{leader}_1(x) \Rightarrow \neg\mathsf{follower}_1(x)\right) \tag{4.9}$$

Die erste Eigenschaft gilt unmittelbar durch die algebraische Spezifikation von Σ_4, die zweite Eigenschaft ist wiederum sehr einfach aus einer Stelleninvariante ableitbar. Die Anwendung unseres Beweismusters liefert:

$$\Sigma_4 \models (\mathsf{coin}_1(x) \wedge \mathsf{leader}_1(x)) \mapsto \mathsf{coin}_1(l(x)) \tag{4.10}$$

Dies ist genau die gewünschte Eigenschaft (4.7).

4.4 Beweisstruktur

In diesem Abschnitt stellen wir ein neues Schema für die Verifikation selbststabilisierender Algorithmen vor. Das Schema orientiert sich an gemeinsamen Merkmalen selbststabilisierender Algorithmen. Wir benutzen das Schema in mehreren Fallstudien. Dazu gehören die randomisierte Leader Election im unidirektionalen Ring (Abschnitt 3.1.1), der asynchrone Tokenaustausch (Abschnitt 3.2.1) sowie der Algorithmus von Brown, Gouda und Wu zum Tokenaustausch in Ketten (Abschnitt 3.3.1). Grundlage des Schemas sind eine einheitliche Struktur von Beweisen, das Ausnutzen verschiedener Abstraktionsebenen im Beweis sowie das Zurückführen *algorithmenspezifischer* Beweisziele auf algorithmenunabhängige *netzwerkspezifische* Beweisziele. Unser Verifikationsschema verbindet bereits bekannte Techniken aus der Welt der verteilten Algorithmen (wie z. B. Stabilisierungsfunktionen) und der Welt der Verifikation von Petrinetzmodellen (wie z. B. Stelleninvarianten) zu einer Strukturierungsmethode für Beweise. Verschiedene Abstraktionsebenen im Beweis eines selbststabilisierenden Algorithmus sind ein Schlüssel für die Verständlichkeit eines Beweises (siehe z. B. [DKVW95]).

Zum Abschluß dieses Abschnitts wenden wir unser Verifikationsschema auf mehrere Algorithmen an und legen somit einen „Fahrplan“ für die Korrektheitsbeweise in Kapitel 5 fest.

4.4.1 Ein Verifikationsschema

Wir geben im folgenden unser Verifikationsschema an. Es besteht aus vier einzelnen Schritten:

1. *Basiseigenschaften:* Zuerst verifizieren wir Basiseigenschaften eines Algorithmus. Einfache strukturelle Eigenschaften lassen sich mit Hilfe von Stelleninvarianten ablesen. Wir zeigen auch einfache Lebendigkeitseigenschaften, wie beispielsweise die Unendlichkeit von Abläufen.

2. *Wechsel der Abstraktionsstufe:* Während im ersten Schritt direkt über die Zustandswechsel der einzelnen Agenten argumentiert wird, führen wir hier eine geeignete Abstraktion ein. Wir argumentieren beispielsweise über eine Nachricht, die im Netzwerk kreist und verschiedenen anderen Nachrichten begegnet. In der Wahl einer geeigneten Abstraktion liegt der Schlüssel zur Verständlichkeit eines Beweises.

3. *Konvergenzstufen:* Für jeden Zustand eines selbststabilisierenden Systems kann man ein Maß definieren, welches den Grad der Stabilisiertheit des Systems charakterisiert. Wir geben in dieser Phase Stabilisierungsfunktionen an.

4. *Beweis der Selbststabilisierung:* Wir betrachten Suffixe von Abläufen, die so gewählt sind, daß ein bestimmter Zustand immer wieder auftritt. Hiervon ausgehend beweisen wir die Selbststabilisierung. Hierbei benutzen wir allgemeine Beweisregeln temporaler Logik der vorherigen Abschnitte.

4.4.2 Anwendungen des Verifikationsschemas

Im folgenden beschreiben wir, wie wir das oben beschriebene Verifikationsschema in Kapitel 5 auf einzelne Algorithmen anwenden werden. Hierdurch ergibt sich der bereits beschriebene Fahrplan für einen Beweis.

Randomisierte Leader Election im unidirektionalen Ring

- *Schritt 1 (Basiseigenschaften):* Hier beweisen wir wesentliche strukturelle Eigenschaften des Algorithmus, z. B. daß jeder Agent immer entweder ein Leader oder ein Follower ist und daß es genau eine Münze gibt, die im Ring kreist.

- *Schritt 2 (Wechsel der Abstraktionsstufe):* Wir betrachten die *Spur* der Münze – lokale Werte der Agenten, die mit dem Zustand der Münze und der Anzahl der Leader insgesamt zusammenhängen. Dieser Abstraktionsschritt erlaubt uns zu beweisen, daß schließlich der Wert der Spur bei einem Agenten dem seines Nachbarn gleicht gdw. dieser Agent ein Follower ist.

- *Schritt 3 (Konvergenzstufen):* Jeweils eine ungerade Anzahl von Leadern stellt eine Konvergenzstufe dar. Das bedeutet, daß bei einer ungeraden Gesamtanzahl an Leadern im System gesichert ist, daß kein neuer Leader entsteht.

- *Schritt 4 (Beweis der Selbststabilisierung):* Wir beweisen, daß schließlich ein Leader immer wieder die Münze wirft und die Konvergenzstufe konstant bleibt. Wir beweisen, daß nun die niedrigste Konvergenzstufe erreicht ist, woraus die Wahl eines eindeutigen Leaders folgt.

Asynchroner Tokenaustausch

- *Schritt 1 (Basiseigenschaften):* Wir beweisen, daß irgendwann ein *semistabiler* Zustand erreicht wird: Jede Nachricht, die sich initial auf einem Kanal befand, ist entweder bereits empfangen oder nicht aktuell (d. h. sie wird nicht mehr empfangen).

- *Schritt 2 (Wechsel der Abstraktionsstufe):* Als wesentliche Abstraktion führen wir den Begriff der *Welle* ein. Jede Nachricht gehört zu einer Welle. Wir abstrahieren vom Empfang einer konkreten Nachricht und betrachten das Versenden und Empfangen von Wellen. Diese Abstraktion ist sehr wirkungsvoll, da sie ignoriert, daß eine Welle bereits empfangen werden kann, bevor jede einzelne Nachricht dieser Welle vollständig versendet wurde.

- *Schritt 3 (Konvergenzstufen):* Die Anzahl der Wellen bildet eine Stabilisierungsfunktion. Sie kann nicht zunehmen, daher bleibt sie in jedem Ablauf letztendlich konstant.

- *Schritt 4 (Beweis der Selbststabilisierung):* Wir beweisen, daß die Anzahl der Wellen schließlich konstant 1 ist. Daraus folgt die Selbststabilisierung direkt. Wir verwenden bei diesem Beweis eine spezifische Eigenschaft der zugrundeliegenden Datenstruktur.

Der Algorithmus von Brown, Gouda und Wu

- *Schritt 1 (Basiseigenschaften):* Wir beweisen die Deadlockfreiheit des Algorithmus: Jeder Ablauf ist unendlich.

- *Schritt 2 (Wechsel der Abstraktionsstufe):* Wir führen als wesentliche Abstraktion den Begriff des Tokens ein. Vorher haben wir den Effekt des Schaltens einer Transition als eine Zustandsänderung eines Agenten beschrieben, jetzt können wir sagen, daß beispielsweise ein Token an einen Nachbarn weitergereicht wurde.

- *Schritt 3 (Konvergenzstufen):* Die Anzahl der Token im System bildet eine Stabilisierungsfunktion, d. h. die Anzahl der Token nimmt nie zu.

- *Schritt 4 (Beweis der Selbststabilisierung):* Wir führen die *closure*-Eigenschaft auf eine Eigenschaft von Ketten (also der Struktur des Netzwerks) zurück, die offensichtlich gilt. Irgendwann gibt es ein Token, daß durch die Kette von Agenten wandert und dabei keine anderen Token trifft und alle Agenten besucht. Hieraus folgt die Selbststabilisierung.

5 Analyse der Fallstudien

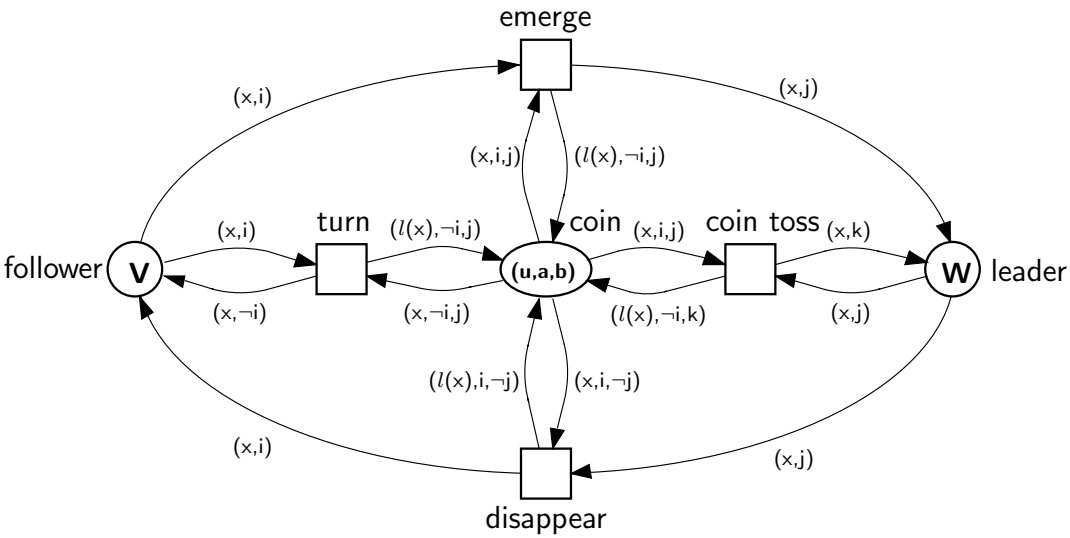
In diesem Kapitel analysieren wir unsere Algorithmen aus Kapitel 3. Konkret beweisen wir die Korrektheit der Petrinetzmodelle, indem wir eine formale Beziehung zwischen der temporallogischen Spezifikation und dem operationellen Systemnetz herleiten. Wir halten uns an das in Abschnitt 4.4 eingeführte Verifikationsschema.

In Abschnitt 5.1 verifizieren wir unseren neuen Algorithmus zur selbststabilisierenden Leader Election im unidirektionalen Ring. Zusätzlich analysieren wir die Zeitkomplexität unseres Algorithmus und beweisen seine Speicherminimalität. In Abschnitt 5.2 beweisen wir die Korrektheit unseres Algorithmus zum asynchronen Tokenaustausch. Eine klassische Komplexitätsanalyse kann bei diesem Algorithmus nicht durchgeführt werden, da unsere Fairneßannahme lediglich qualitativer Natur ist: Es werden lediglich Abläufe ausgeschlossen, die die Spezifikation nicht erfüllen. In Abschnitt 5.3 vergleichen wir einen neuen Beweis mit zwei existierenden Beweisen. Wir zeigen, wie durch geeignete Abstraktionen Beweisziele eines Algorithmus auf algorithmenunabhängige Beweisziele des zugrundeliegenden Netzwerks zurückgeführt werden können. Beim Beweis unseres Algorithmus zum verteilten Zählen in Abschnitt 5.4 wenden wir unser Verifikationsschema nicht an, da die wesentliche Abstraktion (der globale Zählerstand) bereits Bestandteil der Spezifikation ist.

Beim Detaillierungsgrad der Beweise orientieren wir uns am Ziel dieser Arbeit – den Entwurf selbststabilisierender Algorithmen methodisch zu unterstützen. Im Mittelpunkt steht die bei der Verifikation hilfreiche Abstraktion, die eine algorithmische Idee widerspiegelt. Wir vermeiden die in Abschnitt 1.1 diskutierte Gefahr, daß diese Abstraktion nicht zum gewählten Modell paßt, indem wir ausgehend vom operationalen Modell die Grundlagen der Abstraktion formal verifizieren. Wir zeigen also, daß das operationelle Modell eines Algorithmus seine algorithmische Idee widerspiegelt. Natürlich hat die Formalisierung auch Grenzen: Wir werden beispielsweise nicht die Korrektheit von Stelleninvarianten im einzelnen nachweisen. Jedoch ist durch die in Kapitel 4 eingeführten Techniken klar, wie die Korrektheit nachgerechnet werden kann.

5.1 Randomisierte Leader Election

In Abbildung 5.1 ist das Petrinetzmodell unseres Algorithmus zur randomisierten Leader Election in uniformen Ringen noch einmal angegeben. Die we-



spec (U,l) : unidirectional ring	var x : site
sort value = site × bool	var i, j, k : bool
sort coin = site × bool × bool	$V_1 \cup W_1 = U$
const V, W : set of values	$V_1 \cap W_1 = \emptyset$
const (u,a,b) : coin	$\exists x \in U : x = u$

Abbildung 5.1: Σ_4 – Randomisierte Leader Election im unidirektionalen Ring

sentlichen Eigenschaften haben wir wie folgt spezifiziert:

$$\Sigma_4 \models \Box(|\mathsf{coin}| = 1) \tag{5.1}$$

$$\Sigma_4 \models_{\mathcal{R}} \exists x \in U : \Diamond\,\Box(\mathsf{leader}_1 = [x]) \tag{5.2}$$

5.1.1 Verifikation

Die Gültigkeit der Sicherheitseigenschaft (5.1) ist offensichtlich:

Satz 5.1 $\Sigma_4 \models \Box(|\mathsf{coin}| = 1)$

Beweis: $|\mathsf{coin}| = 1$ ist eine Stelleninvariante von Σ_4. □

Die Verifikation der Lebendigkeitseigenschaft baut auf Lemmas auf, die zu der informellen Beschreibung in Abschnitt 3.1.1 korrespondieren. Später kombinieren wir diese Eigenschaften schrittweise.

Lemma 5.2 $\Sigma_4 \models \Box(\mathsf{leader}_1 + \mathsf{follower}_1 = U)$

Beweis: $\mathsf{leader}_1 + \mathsf{follower}_1 = U$ ist eine Stelleninvariante von Σ_4. □

Mit dieser Eigenschaft können wir sagen, daß in jedem Zustand jeder Agent entweder ein *Leader* oder ein *Follower* ist.

Lemma 5.3 $\Sigma_4 \models \forall x \in U : (\mathsf{coin}_1(x) \mapsto \mathsf{coin}_1(l(x))$

Beweis: Abbildung 5.2 zeigt einen Beweisgraph. Seine Knoten sind durch

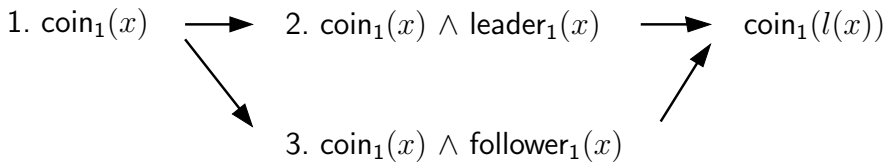


Abbildung 5.2: Ein Beweisgraph für $\mathsf{coin}_1(x) \mapsto \mathsf{coin}_1(l(x))$

folgende Argumente gerechtfertigt:

Knoten 1: Lemma 5.2 impliziert $\mathsf{leader}_1(x) \vee \mathsf{follower}_1(x)$.

Knoten 2: Entweder coin-toss ist aktiviert, oder disappear ist aktiviert. Muster 14 impliziert $\mathsf{coin}_1(l(x))$.

Knoten 3: Entweder turn ist aktiviert, oder emerge ist aktiviert. Muster 14 impliziert $\mathsf{coin}_1(l(x))$. □

Die Menge der Agenten ist endlich, und die Agenten bilden einen Ring. Daher impliziert Lemma 5.3, daß die Münze in Runden umhergereicht wird. Wir zählen im folgenden Runden eines Ablaufs: In jedem Zustand eines Ablaufs ist die Rundennummer die Anzahl der Ereignisse, bei denen Agent u die Münze weitergereicht hat.

Definition 5.4 (Runde) Sei $R = (C_0, C_1, \dots)$ ein Ablauf von Σ_4. Für einen Zustand C_i von R und $n \in \mathbb{N}$ definieren wir $C_i \models \mathit{round} = n$, gdw. $(n-1)|U| < i \leqslant n|U|$.

Von nun an werden wir nur noch Zustände *nach der ersten Runde der Münze* betrachten. Damit können wir uns auf die letzte Aktion beziehen, die ein Agent in einem Ablauf von Σ_4 ausführte. Zuerst betrachten wir einen Follower, der die Münze besitzt. Er kann entweder die Transition turn oder die Transition emerge schalten. Diese Entscheidung hängt von seinem eigenen Zustand und dem zweiten Wert der Münze ab. Daher bezeichnen wir den zweiten Wert der Münze als *f-Wert (Follower-Wert) der Münze*. In Formeln schreiben wir hierfür *f-coin*. Wir definieren den f-Wert eines Followers als den zweiten Wert der zugehörigen Marke auf der Stelle follower. Für einen Leader x definieren wir den f-Wert als den f-Wert, den die Münze hatte, als x das letzte Mal eine Aktion ausführte, also in der vorherigen Runde. In Formeln bezeichnen wir den f-Wert eines Agenten mit *f-val*(x).

Definition 5.5 (F-Wert) Sei $R = (C_0, C_1, \dots)$ ein Ablauf von Σ_4. Für einen Zustand C_i von R, $i \geqslant |U|$, und $k \in \mathbb{B}$ definieren wir:

i. $C_i \models$ *f-coin* $= k$, gdw. $C_i \models \mathsf{coin}_2 = [k]$;

ii. Für einen Follower x gilt $C_i \models$ *f-val*$(x) = k$, gdw. $C_i \models \mathsf{follower}(x, k)$;

iii. Für einen Leader x gilt $C_i \models$ *f-val*$(x) = k$, gdw. $\exists j \in \mathbb{N}, 0 < i - j \leqslant |U| :$ $C_j \models \mathsf{coin}_1 = [x] \wedge \mathsf{coin}_2 = [k]$.

Der F-Wert der Münze ändert sich unendlich oft:

Lemma 5.6 $\Sigma_4 \models \forall i \in \mathbb{B} : (\textit{f-coin} = i \mapsto \textit{f-coin} = \neg i)$

Beweis: Jedes Auftreten der Transitionen emerge oder coin toss ändert den f-Wert der Münze. Daher genügt es zu zeigen, daß immer wieder eine dieser Transitionen auftritt. Jedesmal, wenn eine der Transitionen turn oder disappear für einen Agenten x schaltet, endet x als ein Follower. Darüberhinaus gleicht der f-Wert von x dem f-Wert der Münze. Die Münze kreist zwischen den Agenten. Daher wird die Münze schließlich einen Follower erreichen, dessen f-Wert dem f-Wert der Münze gleicht. Nun ist emerge die einzig aktivierte Transition. □

Ob ein Leader disappear oder coin-flip schaltet, hängt nur von seinem eigenen Wert und dem dritten Wert der Münze ab. Daher nennen wir analog zu Definition 5.5 für jede Marke (x, i) auf leader i den *l-Wert (Leader-Wert)* von x. Für eine Marke (x, i, j) auf der Stelle coin nennen wir j den *L-Wert der Münze*.

Definition 5.7 (L-Wert) Für einen Zustand C von Σ_4 und $k \in \mathbb{B}$ definieren wir:

i. $C \models \textit{l-coin} = k$, gdw. $C \models \mathsf{coin}_3 = [k]$;

ii. Für $x \in U$ gilt $C \models \textit{l-val}(x) = k$, gdw. $C \models \mathsf{leader}(x, k)$.

Lemma 5.8 $\Sigma_4 \models \Diamond\, \exists x \in U : \mathsf{leader}(x, \textit{l-coin})$

Beweis: Die Behauptung folgt direkt aus Lemma 5.6. Wenn sich der f-Wert der Münze ändert, dann entsteht ein Zustand, der die Behauptung erfüllt. □

Lemma 5.9 $\Sigma_4 \models \Diamond\, \Box\, \exists x \in U : \mathsf{leader}(x, \textit{l-coin})$

Beweis: Lemma 5.8 sichert:

$$\Sigma_4 \models \Diamond\, \exists x \in U : \mathsf{leader}(x, \textit{l-coin})$$

Diese Eigenschaft ist stabil, d. h. es gibt keine Transition in Σ_4, die sie ungültig macht. Daher gilt die Behauptung. □

Lemma 5.10 $\Sigma_4 \models \textit{true} \mapsto \exists x \in U : (\mathsf{coin}_1(x) \wedge \mathsf{leader}(x, \textit{l-coin}))$

Beweis: Dieses Lemma folgt aus Lemma 5.9 und Lemma 5.3. Irgendwann wird es immer einen Leader x geben, dessen l-Wert dem l-Wert der Münze gleicht. Der l-Wert der Münze wird nur durch coin-toss geändert. Die Münze kreist zwischen den Agenten. Spätestens bei x tritt coin-toss auf. Dies impliziert die Behauptung. □

Korollar 5.11 $\Sigma_4 \models \exists x \in U : \textit{true} \mapsto (\mathsf{coin}_1(x) \wedge \mathsf{leader}(x, \textit{l-coin}))$

Beweis: folgt aus Lemma 5.10 und der Endlichkeit des Ringes. □

Wir haben bereits bewiesen, daß die Münze in Runden zwischen den Agenten umhergereicht wird. Nun definieren wir eine *Spur der Münze*:

Definition 5.12 (Spur der Münze) Für $x \in U$ und $\mathsf{coin}_1(x)$ nennen wir die Sequenz $(\textit{f-val}(x), \textit{f-val}(l(x)), \ldots, \textit{f-val}(l^{|U|-1}(x)), \textit{f-coin})$ die *Spur der Münze*.

In Abbildung 5.3 ist eine typische Spur der Münze für ein System mit sechs Agenten angegeben. Die Abkürzungen F und L beschreiben, daß der jeweilige Agent im Zustand follower bzw. leader ist.

Lemma 5.13 Sei $(\textit{f-val}(x), y)$ eine Teilsequenz der Spur der Münze. Dann gilt $\textit{f-val}(x) = y$, gdw. x ein Follower ist.

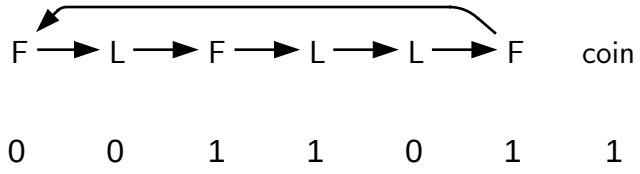


Abbildung 5.3: Eine typische Spur der Münze

Beweis: Die Behauptung folgt aus Lemma 5.6. Nur ein Leader ändert den f-Wert der Münze. □

Das nächste Lemma ist entscheidend für die Verifikation der Lebendigkeitseigenschaft, da es Konvergenzstufen (siehe Abschnitt 4.2) impliziert. Irgendwann (nach der ersten Runde) wird eine ungerade Anzahl von Leadern nie mehr größer.

Lemma 5.14 $\Sigma_4 \models \forall n \in \mathbb{N} : \Diamond\,\Box\big(|\mathsf{leader}| = 2n+1 \Rightarrow \Box(|\mathsf{leader}| \leqslant 2n+1)\big)$

Beweis: Sei $x \in U$ der Münzbesitzer, also $\mathsf{coin}_1(x)$. Nun kann die Anzahl der Leader nur dann größer werden, wenn x ein Follower ist und die Transition emerge auftritt. Sei j der f-Wert von x. Auf der Spur von x zur Münze ändert sich der Wert in der Sequence $2n+1$ mal, wegen der ungeraden Anzahl der Leader und Lemma 5.13. Deshalb gilt $\mathit{f\text{-}coin} = \neg j$, und somit ist emerge nicht aktiviert. □

Definition 5.15 (Konvergenzstufe) Sei R ein Ablauf von Σ_4. Für einen Zustand C von R mit $C \models \mathit{round} > 1$ definieren wir $C \models \mathit{stair} = n$, gdw. $C \models |\mathsf{leader}| = 2n$ oder $C \models |\mathsf{leader}| = 2n+1$.

Die Konvergenzstufe nimmt ab, wenn die Anzahl der Leader gerade ist und die Transition disappear auftritt:

Lemma 5.16 $\Sigma_4 \models \forall x \in U\ \forall n,k \in \mathbb{N} : (\mathit{stair} = n \wedge |\mathsf{leader}| = 2k \wedge \mathsf{coin}_1(x) \wedge \mathsf{leader}(x, \neg \mathit{l\text{-}coin})) \mapsto \mathit{stair} = n-1$

Beweis: folgt direkt aus obiger Definition. □

Eine Konvergenzstufe nimmt nie zu:

Lemma 5.17 $\Sigma_4 \models \forall n \in \mathbb{N} : \Box(\mathit{stair} = n \Rightarrow \Box\, \mathit{stair} \leqslant n)$

Beweis: Die Behauptung folgt direkt aus Lemma 5.14, zusammen mit der Beobachtung, das jedes Auftreten einer Transition die Anzahl der Leader um höchstens eins erhöht. □

Nun haben wir alle Voraussetzungen für den eigentlichen Beweis der Lebendigkeitseigenschaft bewiesen. Alle obigen Lemmas gelten unabhängig von einer Randomisierungsannahme. Im nächsten Teil des Beweises werden wir zeigen, daß die Randomisierungsannahme für die Transition coin toss die Wahl eines eindeutigen Leaders gewährleistet. Bevor wir die Randomisierungsannahme formalisieren, überblicken wir die verbleibenden Beweisschritte. Der Einfachheit halber nehmen wir im folgenden an, daß die Anzahl der Agenten größer als zwei ist. In Abbildung 5.4 ist ein Beweisgraph für die Eigenschaft $\Sigma_4 \models_{\mathcal{R}} \Diamond\Box(stair = 0)$ angegeben.

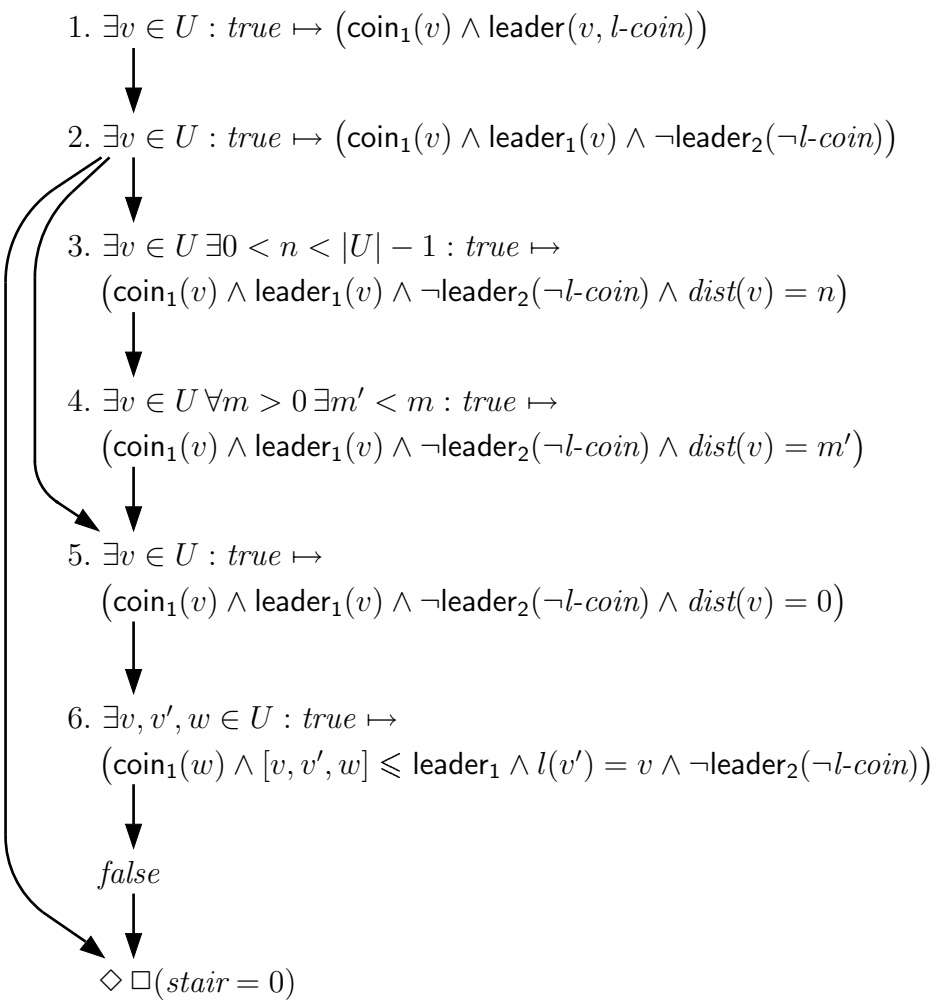


Abbildung 5.4: Ein Beweisgraph für $\Sigma_4 \models_{\mathcal{R}} \Diamond\Box(stair = 0)$

Wir beschreiben die einzelnen Beweisschritte des Beweisgraphen zuerst informell. Korollar 5.11 besagt, daß in jedem Ablauf die Münze unendlich oft

von einem Leader $v \in U$ geworfen wird (Knoten 1). Darüberhinaus ist das System Σ_4 ein System mit endlich vielen erreichbaren Zuständen. Also gibt es auch einen Leader $v \in U$, der die Münze unendlich oft in einem bestimmten Zustand C wirft.

In Lemma 5.20 werden wir zeigen, daß in jedem randomisierten Ablauf schließlich ein Zustand erreicht wird, in dem v die Münze werfen kann und jeder Leader denselben l-Wert hat (Knoten 2).

Von jetzt an unterscheiden wir drei Fälle: Erster Fall, die Konvergenzstufe ist 0. In diesem Fall sind wir fertig. Anderenfalls gilt, daß es mehr als einen Leader gibt. Wir definieren den Abstand von v zum nächsten Leader in Richtung rechts (Notation: $\mathit{dist}(v)$). Zweiter Fall, der rechte Nachbar von v ist ein Leader (Knoten 5). Dritter Fall, der rechte Nachbar von v ist ein Follower (Knoten 3).

In Lemma 5.23 werden wir beweisen, daß im dritten Fall auch ein Zustand mit geringerem Abstand unendlich oft auftritt (Knoten 4). Induktion über die Größe von $\mathit{dist}(v)$ führt dann zu Knoten 5.

Analog zu den Argumenten, die zu Knoten 2 führten, werden wir in Lemma 5.25 zeigen, daß unendlich oft ein Zustand C' erreicht wird, in dem ein dritter Leader w die Münze wirft und v sowie sein rechter Nachbar v' den gleichen l-Wert haben (Knoten 6).

Vom Zustand C' aus werden in einem randomisierten Ablauf schließlich v und v' beide zu Followern. Dies verringert die Konvergenzstufe und ist ein Widerspruch zur Unendlichkeit der Häufigkeit des Auftretens von C' (Lemma 5.26). Dies schließt den Beweis ab.

Nun fixieren wir die Randomisierungsannahme (vgl. Abschnitt 2.2.2):

Lemma 5.18 Seien $A = \{(\mathsf{coin\ toss}, \beta) : \beta(\mathsf{k}) = 0\}$ und $A' = \{(\mathsf{coin\ toss}, \beta) : \beta(\mathsf{k}) = 1\}$. Dann ist $\mathcal{R} = (A, A')$ eine Randomisierungsannahme für Σ_4.

Beweis: A und A' sind disjunkte Mengen von Aktionen von Σ_4. Nach Definition 2.41 genügt es zu zeigen, daß in jedem erreichbaren Zustand von Σ_4 gilt:

1. Zu jeder aktivierten Aktion $a \in A$ existiert eine aktivierte Aktion $a' \in A'$ und anders herum:

   Zu jeder Aktion $a = (\mathsf{coin\ toss}, \beta_a) \in A$ gibt es eine Aktion $a' = (\mathsf{coin\ toss}, \beta_{a'}) \in A'$, so daß $\beta_a(x) = \beta_{a'}(x)$ für alle Variablen außer k gilt. Offensichtlich ist a' aktiviert, falls a aktiviert ist. Die entgegengesetzte Richtung ist symmetrisch.

2. Wenn zwei Aktionen $a, a' \in A \cup A'$ aktiviert sind, dann stehen sie in Konflikt:

   Zwei aktivierte Aktionen teilen immer eine Ressource auf der Stelle coin. Wegen Satz 5.1 ist diese Ressource eindeutig. □

Notation 5.19 Im folgenden benutzen wir eine abkürzende Schreibweise für Sequenzen von Münzwurfergebnissen, in der wir jedem Resultat seine Häufigkeit zuweisen. Wir schreiben beispielsweise $0^3 10$ für die Sequenz $(0, 0, 0, 1, 0)$.

Lemma 5.20 Knoten 1 des Beweisgraphen in Abbildung 5.4 gilt.

Beweis: In jedem randomisierten Ablauf R von Σ_4 gibt es unendlich viele Zustände, in denen v die Münze werfen kann. Darüberhinaus ist Σ_4 ein System mit endlich vielen Zuständen. Also gibt es einen Zustand C, der in R unendlich oft auftritt, mit $C \models \mathsf{coin}_1(v) \wedge \mathsf{leader}(v, \textit{l-coin})$.

Sei m die Anzahl der Leader in C außer v, die den l-Wert 0 haben. Wir wenden Definition 2.44 an mit der endlichen Sequenz von Münzwurfresultaten 0^{m+1}. Mit vorbestimmten Münzwurfresultaten verhält sich Σ_4 deterministisch.

In der nächsten Runde werden nur v und die m anderen Leader die Münze werfen. Daher führt diese Sequenz zu einem Zustand C', in dem v die Münze hat und jeder andere Leader denselben l-Wert 0 hat, d. h. $C' \models \mathsf{coin}_1(v) \wedge \mathsf{leader}_1(v) \wedge \neg\mathsf{leader}_2(1)$. Deshalb gilt $R \models \textit{true} \mapsto C$ impliziert $R \models_{\mathcal{R}} C \mapsto C'$. Daraus folgt die Behauptung wegen der Transitivität von $\mapsto$. □

Für einen Agent x bezeichnet der *Abstand* ($\textit{dist}(x)$) die Anzahl der aufeinanderfolgenden Follower in Richtung rechts:

Definition 5.21 (Abstand) Für einen Zustand C von Σ_4, $n \in \mathbb{N}$ und einen Agent $x \in U$ gilt $C \models \textit{dist}(x) = n$, gdw. $n < |U|$ und $C \models \exists y \in U : \mathsf{leader}_1(y) \wedge l^{n+1}(y) = x \wedge \forall m \in \mathbb{N}, 1 < m \leqslant n : \mathsf{follower}_1(l^n(y))$.

Abbildung 5.5 veranschaulicht die obige Definition: Der Abstand von v ist 2 (da $l^3(y) = v$). Wiederum steht L für einen Leader und F für einen Follower.

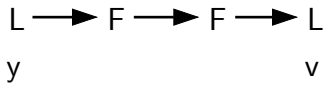


Abbildung 5.5: $\textit{dist}(v) = 2$

Lemma 5.22 Knoten 2 des Beweisgraphen in Abbildung 5.4 gilt.

Beweis: Wir betrachten wiederum einen randomisierten Ablauf R von Σ_4, einen Agent $v \in U$ und einen Zustand C, der unendlich oft auftritt, mit $C \models \mathsf{coin}_1(v) \wedge \mathsf{leader}_1(v) \wedge \neg\mathsf{leader}_2(\neg \mathit{l\text{-}coin})$.

Nun betrachten wir drei Fälle für C: $\mathit{dist}(v) = n-1$, $\mathit{dist}(v) = 0$ und $0 < \mathit{dist}(v) < n-1$. Wegen obiger Definition gilt einer dieser Fälle für C.

Der erste Fall impliziert $C \models \mathit{stair} = 0$. Zusammen mit Lemma 5.17 impliziert dies $R \models_{\mathcal{R}} \Diamond\Box(\mathit{stair} = 0)$.

Der zweite und der dritte Fall sind durch die Knoten 5 bzw. 3 im Beweisgraphen repräsentiert. □

Lemma 5.23 Knoten 3 des Beweisgraphen in Abbildung 5.4 gilt.

Beweis: Wir betrachten wiederum einen randomisierten Ablauf R von Σ_4, einen Agent $v \in U$, $n \in \mathbb{N}$ und einen Zustand C, der unendlich oft auftritt, mit $C \models \mathsf{coin}_1(v) \wedge \mathsf{leader}_1(v) \wedge \neg\mathsf{leader}_2(\neg \mathit{l\text{-}coin}) \wedge \mathit{dist}(v) = n$.

Da C unendlich oft in R auftritt und die Konvergenzstufe nie zunimmt (Lemma 5.17) bleibt die Konvergenzstufe konstant, d. h. $C \models \exists k \in \mathbb{N} : \Box(\mathit{stair} = k)$. Wegen Definition 2.44 wird v irgendwann den Wert $\neg \mathit{l\text{-}coin}$ im Zustand C werfen. In der nächsten Runde wird kein anderer Leader die Münze werfen, da alle anderen Leader einen l-Wert haben, der sich von dem der Münze unterscheidet.

Wir betrachten den Zustand, in dem die Münze bei dem Leader v' angelangt ist, der sich in C am weitesten links von v befindet. Dieser Leader wird die Transition $\mathsf{disappear}$ schalten. Danach ist wegen Lemma 5.16 die Anzahl der Leader gerade. Darüberhinaus gilt $\mathit{l\text{-}val}(v') = \mathit{l\text{-}coin}$. Nun betrachten wir die Spur der Münze, ausgehend vom Follower $l(v')$. Wegen der geraden Anzahl der Leader gilt $\mathit{f\text{-}val}(v') = \mathit{f\text{-}val}(l(v'))$. Dies impliziert $\mathit{f\text{-}val}(l(v')) = \mathit{f\text{-}coin}$. Daher wird $l(v')$ die Transition emerge schalten und zum Leader werden. Dies reduziert den Abstand von v. □

Lemma 5.24 Knoten 4 des Beweisgraphen in Abbildung 5.4 gilt.

Beweis: Die Gültigkeit folgt trivial mit $m = 1$. □

Lemma 5.25 Knoten 5 des Beweisgraphen in Abbildung 5.4 gilt.

Beweis: Wir betrachten wiederum einen randomisierten Ablauf R von Σ_4, einen Agent $v \in U$ und einen Zustand C, der unendlich oft auftritt, mit $C \models \mathsf{coin}_1(v) \wedge \mathsf{leader}_1(v) \wedge \neg\mathsf{leader}_2(\neg \mathit{l\text{-}coin}) \wedge \mathit{dist}(v) = 0$.

Dies impliziert, daß ein Leader v' existiert mit $l(v') = v$. Nun unterscheiden wir zwei Fälle:

Zuerst betrachten wir den Fall $C \models |\mathsf{leader}| > 2$. Sei w der nächste Leader in Richtung links von v aus gesehen. Wegen Definition 2.44 wird v irgendwann im Zustand C den Münzwert $\neg$*l-coin* werfen. Somit wird w die Münze erhalten, was den Beweis für diesen Fall abschließt.

Anderenfalls gilt $C \models |\mathsf{leader}| = 2$. Dann ist $l(v)$ ein Follower (da $|U| > 2$). Wir wenden Definition 2.44 an mit $(\neg$*l-coin*$)^3$ als endlicher Sequenz von Münzwurfresultaten. Wenn wir die Spur der Münze betrachten, können wir analog zu Lemma 5.23 beweisen, daß $l(v)$ ein Leader wird. Darüberhinaus wird kein anderer Agent ein Leader. Nach den drei Münzwürfen (von v, v' und wiederum v), besitzt $l(v)$ die Münze. Mit $w = l(v)$ impliziert dies die Behauptung. □

Lemma 5.26 Knoten 6 des Beweisgraphen in Abbildung 5.4 gilt.

Beweis: Wir betrachten wiederum einen randomisierten Ablauf R von Σ_4, Agenten $v, v', w \in U$ und einen Zustand C, der unendlich oft auftritt, mit $C \models \mathsf{coin}_1(w) \wedge [v, v', w] \leqslant \mathsf{leader}_1 \wedge l(v') = v \wedge \neg\mathsf{leader}_2(\neg$*l-coin*$)$.

Mit Definition 2.44 gilt, daß irgendwann w in C den Wert *l-coin* wirft. Danach werden v und v' beide die Transition $\mathsf{disappear}$ schalten. Dadurch ist die Konvergenzstufe verringert. Dies ist ein Widerspruch zur Unendlichkeit des Auftretens von C. □

Schließlich beweisen wir die wesentliche Eigenschaft von Σ_4: In jedem randomisierten Ablauf von Σ_4 existiert ein Agent $x \in U$, der zum eindeutigen Leader gewählt wird.

Satz 5.27 $\Sigma_4 \models_{\mathcal{R}} \exists x \in U : \Diamond\Box(\mathsf{leader}_1 = [x])$

Beweis: Der Beweisgraph in Abbildung 5.4 liefert:

$$\Sigma_4 \models_{\mathcal{R}} \Diamond\Box(\mathit{stair} = 0) \tag{5.3}$$

Daraus folgt:

$$\Sigma_4 \models_{\mathcal{R}} \Diamond\Box(|\mathsf{leader}| = 0 \vee |\mathsf{leader}| = 1) \tag{5.4}$$

Weiterhin gilt wegen Lemma 5.9:

$$\Sigma_4 \models \Diamond\Box(\exists x \in U : \mathsf{leader}(x, \textit{l-coin})) \tag{5.5}$$

Daher gilt:

$$\Sigma_4 \models_{\mathcal{R}} \Diamond\Box(\exists x \in U : \mathsf{leader}_1(x) \wedge |\mathsf{leader}| = 1) \tag{5.6}$$

Daraus folgt:

$$\Sigma_4 \models_{\mathcal{R}} \Diamond\Box(\exists x \in U : \mathsf{leader}_1 = [x]) \tag{5.7}$$

In Σ_4 gibt es keine Aktion, die zwei Leader in einem einzigen Schritt austauscht. Deshalb:

$$\Sigma_4 \models_{\mathcal{R}} \exists x \in U : \Diamond\Box(\mathsf{leader}_1 = [x]) \tag{5.8}$$

□

5.1.2 Komplexität

In diesem Abschnitt analysieren wir die Zeit- und Speicherkomplexität von Σ_4. Zur Analyse der Zeitkomplexität führen wir unseren Algorithmus auf einen bekannten Algorithmus und ein bekanntes Resultat aus [MOY96] und [IJ90] zurück. Wir beweisen danach, daß der Speicherbedarf von Σ_4 (zwei Bits pro Agent) minimal ist.

Satz 5.28 Die erwartete Stabilisierungszeit von Σ_4 beträgt $O(n^2)$ Runden.

Beweis: Die Spur der Münze kann als eine *selbststabilisierende digitale Uhr* (vgl. die *digital clock* in [MOY96]) betrachtet werden. Daher können wir die gleichen Argumente wie in [MOY96] verwenden, um die Zeitanalyse auf ein Standard-*Random-Walk*-Problem [IJ90] zurückzuführen. Dies impliziert die Behauptung. □

Unser Algorithmus Σ_4 benötigt zwei Bits pro Agent. Wir beweisen im folgenden die Minimalität dieses Speicherplatzbedarfs, d. h. ein Bit pro Agent genügt nicht.

Theorem 5.29 Es gibt keinen uniformen selbststabilisierenden Leader Election-Algorithmus für unidirektionale Ringe, der auf Tokenaustausch basiert, konstanten Speicherplatz benötigt und nur ein Bit pro Agent benutzt.

Beweis: Wir nehmen die Existenz eines Algorithmus A an, der die Voraussetzungen erfüllt.

Da A konstanten Speicherplatz hat, gibt es ein $k \in \mathbb{N}$, so daß k die Größe des Speicherplatzes (in Bits) des Tokens ist. Also gibt es in A für einen Agenten höchstens 2^{k+1} mögliche Zustandsübergänge (durch 2^k mögliche Zustände des Tokens und das Bit des Agenten selbst). Das Bit eines Agenten beschreibt, ob der Agent ein Leader L oder ein Follower F ist (da ein Agent sonst nicht lokal entscheiden kann, ob er ein Leader ist).

Unter den 2^k möglichen Zuständen des Tokens gibt es m, $m > 0$, Zustände, die es einem Leader ermöglichen[1], ein Follower zu werden. Anderenfalls könnte ein Anfangszustand ohne Follower nicht stabilisiert werden.

Analog dazu gibt es m', $m' > 0$, Zustände des Tokens, die einem Follower ermöglichen, ein Leader zu werden. Anderenfalls könnte ein Anfangszustand ohne Leader nicht stabilisiert werden.

Wir betrachten einen stabilen Zustand C_S in einem Ring von n Agenten $(p_0, \ldots, p_{n-1})$, so daß p_0 der Leader ist und das Token besitzt. In diesem Zustand gibt es genau einen Leader und $n-1$ Follower. In jedem Ablauf von A, der in C_S beginnt, ist keine Aktion aus $m \cup m'$ aktiviert (Stabilität von C_S). Wir betrachten nun das Verhalten des Tokens. Das Token kreist im Ring. Es wird eine unendliche zyklische Sequenz von Agenten treffen, deren Zustand L oder F ist. Diese Sequenz ist beginnend bei p_0 die folgende: $\Big(L, \underbrace{F, \ldots, F}_{n-1}\Big)^*$.

Da jeder Agent nur ein Bit hat und die Agenten uniform sind, beschreibt eine Sequenz vollständig die möglichen Zustandsübergänge des Tokens. Mit dieser Sequenz wird das Token nie eine Aktion aus $m \cup m'$ aktivieren.

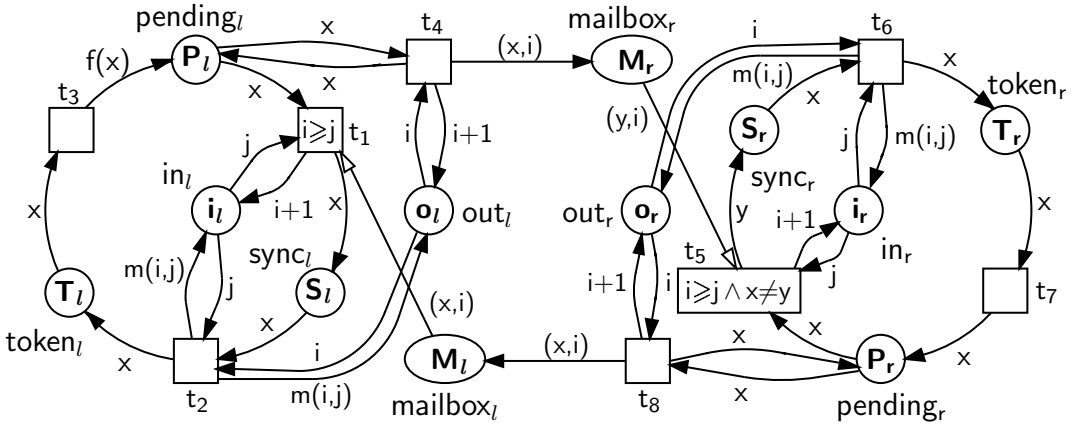
Nun betrachten wir einen Ring, der aus $2n$ Agenten besteht, mit einem erweiterten Anfangszustand C_S, so daß $C_S(p_{n+i}) = C_S(p_i)$, für $0 \leqslant i < n$. Es gibt zwei Leader p_0 und p_n. Aus der Sicht des Tokens jedoch hat sich die Sequenz $\Big(L, \underbrace{F, \ldots, F}_{n-1}\Big)^*$ nicht geändert. Also kann das Token nicht zwischen beiden Systemen unterscheiden. Keine Aktion aus $m \cup m'$ wird jemals aktiviert werden, und somit wird sich das erweiterte System nicht selbst stabilisieren. Das ist ein Widerspruch zur Annahme. □

5.2 Asynchroner Tokenaustausch

In Abbildung 5.6 ist das Petrinetzmodell unseres neuen Algorithmus für den asynchronen Tokenaustausch noch einmal angegeben. Zusätzlich haben wir den Transitionen explizit Namen gegeben. Die wesentlichen Eigenschaften haben wir wie folgt spezifiziert:

$$\Sigma_7 \models \mathit{true} \mapsto (|\mathsf{token}_l| > 0) \tag{5.9}$$

[1] Für einen deterministischen Algorithmus A könnte hier „erzwingen“ statt „ermöglichen“ stehen; wir wollen jedoch nicht ausschließen, daß A ein nichtdeterministischer oder ein randomisierter Algorithmus ist.



sort 3val	var x, y : 3val
sort message = 3val × nat	var i, j : nat
fct f : 3val ⟶ 3val	f(0) = 1
const 0, 1, 2 : 3val	f(1) = 2
const P_l, S_l, T_l, P_r, S_r, T_r : multiset of 3val	f(2) = 0
const i_l, o_l, i_r, o_r : nat	∃ x ∈ 3val : $P_l + S_l + T_l$ = [x]
const M_l, M_r : finite multiset of messages	∃ x ∈ 3val : $P_r + S_r + T_r$ = [x]

Abbildung 5.6: Σ_7 – Asynchroner Tokenaustausch

$$\Sigma_7 \models \mathit{true} \mapsto (|\mathsf{token}_r| > 0) \tag{5.10}$$

$$\Sigma_7 \models \Diamond\,\Box(|\mathsf{token}_l + \mathsf{token}_r| \leqslant 1) \tag{5.11}$$

Für den Beweis der Eigenschaften (5.9) und (5.10) zeigen wir zunächst, daß in jedem Ablauf von Σ_7 gilt:

1. Über beide Kanäle werden unendlich oft Nachrichten versendet.

2. Von beiden Kanälen werden unendlich oft Nachrichten empfangen.

Die geforderten Eigenschaften (5.9) und (5.10) folgen dann direkt. Wir beginnen mit einigen Stelleninvarianten:

$$\Sigma_7 \models \Box\,|\mathsf{pending}_l + \mathsf{sync}_l + \mathsf{token}_l| = 1 \tag{5.12}$$

$$\Sigma_7 \models \Box\,|\mathsf{pending}_r + \mathsf{sync}_r + \mathsf{token}_r| = 1 \tag{5.13}$$

$$\Sigma_7 \models \Box\, |\mathsf{in}_l| = 1 \tag{5.14}$$

$$\Sigma_7 \models \Box\, |\mathsf{out}_l| = 1 \tag{5.15}$$

$$\Sigma_7 \models \Box\, |\mathsf{in}_\mathsf{r}| = 1 \tag{5.16}$$

$$\Sigma_7 \models \Box\, |\mathsf{out}_\mathsf{r}| = 1 \tag{5.17}$$

In Lemma 5.32 zeigen wir eine grundlegende Lebendigkeitseigenschaft: Jeder Agent kommt immer wieder in den Zustand pending. Wir beweisen dies nur für den linken Agenten; die Anwendung der Beweisschritte auf den rechten Agenten ist analog.

Lemma 5.30 $\Sigma_7 \models \mathsf{sync}_l(x) \mapsto \mathsf{token}_l(x)$

Beweis: Wir wenden eine Kombination aus Muster 9 und Muster 10 an. Die Invarianten (5.14) und (5.15) sichern, daß t_2 aktiviert ist. Invariante (5.12) sichert, daß Transitionen t_4 und t_1 nicht schalten. □

Lemma 5.31 $\Sigma_7 \models \mathsf{token}_l(x) \mapsto \mathsf{pending}_l(f(x))$

Beweis: Wir wenden Muster 2 an. □

Lemma 5.32 $\Sigma_7 \models \mathit{true} \mapsto \exists\, x \in \{0,1,2\} : \mathsf{pending}_l(x)$

Beweis: Die Behauptung folgt aus Invariante (5.12), Lemma 5.31, Lemma 5.32 sowie der Transitivität von $\mapsto$ (Lemma 4.20). □

Lemma 5.33 $\Sigma_7 \models \mathit{true} \mapsto \exists\, x \in \{0,1,2\} : \mathsf{pending}_\mathsf{r}(x)$

Beweis: analog zu Lemma 5.32. □

Wir zeigen nun, daß über beide Kanäle unendlich oft Nachrichten versendet werden.

Lemma 5.34 $\Sigma_7 \models \mathsf{out}_l(i) \mapsto (\exists\, j \in \mathbb{N} : \mathsf{out}_l(j) \wedge j > i)$

Beweis: Zuerst stellen wir fest, daß jede Transition im Nachbereich von out_l den Wert von out_l nicht verringert. Jede Aktion von t_4 erhöht den Wert von out_l sogar. Wir führen den Beweis nun indirekt. Angenommen, in einem Ablauf von Σ_7 würde t_4 nur endlich oft schalten.

Wegen Lemma 5.32 gibt es immer wieder einen Zustand, in dem t_4 aktiviert ist:

$$\Sigma_7 \models \mathit{true} \mapsto \exists\, x \in \{0,1,2\} : \mathsf{pending}_l(x)$$

Hieraus folgt (wegen der Endlichkeit des Domains $\{0,1,2\}$):

$$\Sigma_7 \models \exists\, x \in \{0,1,2\} : \mathit{true} \mapsto \mathsf{pending}_l(x)$$

Da nach unserer Annahme t_4 nur endlich oft schaltet, muß die im Konflikt stehende Transition t_1 unendlich oft schalten. Wenn in einem Zustand $\mathsf{pending}_l(x)$ die Transition t_1 schaltet, dann gilt danach $\mathsf{sync}_l(x)$. Wegen Lemma 5.30 und 5.31 gilt somit auch:

$$\Sigma_7 \models \forall\, x \in \{0,1,2\} : \mathit{true} \mapsto \mathsf{pending}_l(x)$$

Dies bedeutet, daß t_1 unendlich oft im Modus $x = 0$, unendlich oft im Modus $x = 1$ und unendlich oft im Modus $x = 2$ schaltet. Da t_8 die einzige Transition ist, die Marken auf $\mathsf{mailbox}_l$ legt, muß auch t_8 unendlich oft mit jeder Belegung von x schalten. Die Stelle $\mathsf{pending}_r$ wird von t_7 und t_8 gefüllt. Da t_8 die Marke auf $\mathsf{pending}_r$ nicht verändert und somit keinen anderen Modus für t_8 ermöglicht, muß auch t_7 unendlich oft schalten. Wegen der benötigten Ressourcen auf der Stelle token_r muß auch t_6 unendlich oft schalten. Wegen sync_r muß t_5 unendlich oft schalten, und wegen $\mathsf{mailbox}_r$ muß t_4 unendlich oft schalten. Dies ist ein Widerspruch zur Annahme, daß t_4 nur endlich oft schaltet. Also schaltet in jedem Ablauf t_4 tatsächlich unendlich oft. Dies impliziert unsere Behauptung. □

Lemma 5.35 $\Sigma_7 \models \mathsf{out}_r(i) \mapsto (\exists\, j \in \mathbb{N} : \mathsf{out}_r(j) \wedge j > i)$

Beweis: Analog zu Lemma 5.34. □

Die im Beweis von Lemma 5.34 benutzte Argumentation kann auch (mit einigem Aufwand) formalisiert werden. Wir haben über Ereignisse und Zustände gleichzeitig argumentiert. Eine Logik dazu wurde in [KV98a, KV99] vorgeschlagen. Hier haben wir darauf verzichtet, da die Gültigkeit des Lemmas ohnehin einfach nachzuvollziehen ist.

Wir zeigen im folgenden, daß immer wieder Nachrichten empfangen werden. Konkret zeigen wir, daß immer wieder Nachrichten mit einem anderen Wert versendet werden:

Lemma 5.36 $\Sigma_7 \models \mathsf{mailbox_r}(x,i) \mapsto (\exists\, j \in \mathbb{N} : \mathsf{mailbox_r}(f(x),j) \wedge j > i)$

Beweis: Von Lemma 5.34 wissen wir bereits, daß $\mathsf{t_4}$ unendlich oft schaltet. Die gewünschte Eigenschaft kann also nur dann verletzt werden, wenn es eine Variablenbelegung $k' \in \{0,1,2\}$ für x gibt, mit der $\mathsf{t_4}$ nur endlich oft schaltet. Dann gibt es auch eine Variablenbelegung, die x einen Wert k zuordnet, so daß gilt: $\mathsf{t_4}$ schaltet unendlich oft in einem Modus, der der Variable x den Wert k zuordnet und nur endlich oft in einem Modus, der x den Wert $f(k)$ zuordnet. Wir nehmen die Existenz eines solchen Ablaufs an und führen die Annahme zum Widerspruch.

Sei o. B. d. A. $k = 0$. Also schaltet $\mathsf{t_4}$ nur endlich oft im Modus $x = 1$. $\mathsf{t_5}$ schaltet daher nur endlich oft im Modus $y = 1$. Daher schalten auch $\mathsf{t_6}$ und $\mathsf{t_7}$ nur endlich oft im Modus $x = 1$. Dann schaltet aber auch $\mathsf{t_5}$ nur endlich oft im Modus $x = 1$. Also gibt es ein Suffix des Ablaufs, in dem die Transitionen $\mathsf{t_5}$, $\mathsf{t_6}$ und $\mathsf{t_7}$ nicht mehr im Modus $x = 1$ schalten.

Wegen Lemma 5.33 gilt irgendwann im Suffix $\exists x \in \{0,1,2\} : \mathsf{pending_r}(x)$. Im Fall $x = 1$ wird irgendwann $\mathsf{t_5}$ im Modus $y \neq 1$ schalten (da wir Fairneß für $\mathsf{t_5}$ gefordert haben). Dies führt schließlich zum zweiten Fall, in dem $\mathsf{pending_r}(x) \wedge x \neq 1$ gilt. Somit schaltet $\mathsf{t_8}$ nur endlich oft im Modus $x = 1$. Somit schalten auch $\mathsf{t_1}$, $\mathsf{t_2}$ und $\mathsf{t_3}$ nur endlich oft im Modus $x = 1$. Also gibt es ein Suffix des Ablaufs, in dem $\mathsf{t_1}$, $\mathsf{t_2}$ und $\mathsf{t_3}$ nicht mehr im Modus $x = 1$ schalten.

Da gemäß unserer Annahme $\mathsf{t_4}$ unendlich oft im Modus $x = 0$ schaltet, gilt im Suffix irgendwann $\mathsf{pending}_l(0)$. Nun unterscheiden wir zwei Fälle:

1. $\mathsf{t_1}$ schaltet unendlich oft im Modus $x = 0$. Dies führt zu $\mathsf{pending}_l(1)$. Da $\mathsf{t_1}$ im Suffix nicht mehr im Modus $x = 1$ schaltet, schaltet $\mathsf{t_4}$ unendlich oft im Modus $x = 1$. Dies ist ein Widerspruch zur Annahme.

2. $\mathsf{t_1}$ schaltet endlich oft im Modus $x = 0$. Dann gibt es ein Suffix, in dem $\mathsf{t_1}$ nicht mehr im Modus $x = 0$ schaltet. Da $\mathsf{t_4}$ unendlich oft im Modus $x = 0$ schaltet, kann $\mathsf{t_4}$ nur endlich oft im Modus $x = 2$ schalten. Wie oben können wir nun auch zeigen, daß $\mathsf{t_8}$ nur endlich oft im Modus $x = 2$ schaltet. Da $\mathsf{t_8}$ auch nur endlich oft im Modus $x = 1$ schaltet, schaltet $\mathsf{t_8}$ nur im Modus $x = 0$ unendlich oft. Das widerspricht aber der Fairneßannahme für $\mathsf{t_1}$. □

Lemma 5.37 $\Sigma_7 \models \mathsf{mailbox}_l(x,i) \mapsto (\exists\, j \in \mathbb{N} : \mathsf{mailbox}_l(f(x),j) \wedge j > i)$

Beweis: Analog. □

Der Beweis ähnelt dem Beweis von Lemma 5.34. Jedoch haben wir in diesem Beweis unsere Fairneßannahme ausgenutzt. Lemma 5.34 gilt auch ohne Fairneßannahmen. Die Argumentationskette über (fast) alle Transitionen hinweg ist typisch für Kommunkationsprotokolle (vgl. z. B. Beweise des Alternating-Bit-Protokolls in [Hai85, Gou85, AUWY82]).

Satz 5.38 $\Sigma_7 \models \mathit{true} \mapsto (|\mathsf{token}_l| > 0)$

Beweis: Wegen Lemma 5.36 schaltet $\mathsf{t_1}$ in jedem Ablauf unendlich oft. Daher gilt auch:

$$\Sigma_7 \models \mathit{true} \mapsto \exists x \in \{0,1,2\} : \mathsf{sync}_l(x)$$

Mit Lemma 5.30 folgt die Behauptung. □

Satz 5.39 $\Sigma_7 \models \mathit{true} \mapsto (|\mathsf{token}_\mathsf{r}| > 0)$

Beweis: Analog zum Beweis von Satz 5.38. □

Intuitiv gibt es in jedem Ablauf eine erste Stabilisierungsphase, in der sich die Kanäle „einschwingen“. Konkret bedeutet dies, daß irgendwann alle Nachrichten einen kleineren Zeitstempel haben als der zugehörige out-Zähler und jede aktuelle Nachricht (deren Zeitstempel größer oder gleich dem in-Zähler ist) einen bezüglich dieses Kanals eindeutigen Zeitstempel hat. Wir nennen einen Zustand mit diesen Eigenschaften *semistabil*.

Definition 5.40 (Semistabil) Ein Zustand C von Σ_7 heißt *semistabil* (Notation $C \models \mathit{semistabil}$), gdw. gilt:

i. $C \models (\mathsf{out}_l(i) \wedge \mathsf{mailbox}_\mathsf{r}(x,j)) \Rightarrow i > j$,

ii. $C \models (\mathsf{out}_\mathsf{r}(i) \wedge \mathsf{mailbox}_l(x,j)) \Rightarrow i > j$,

iii. $C \models (\mathsf{mailbox}_\mathsf{r}(x,i) \wedge \mathsf{in}_\mathsf{r}(j) \wedge i > j) \Rightarrow \mathsf{mailbox}_{\mathsf{r}2}[i] = 1$ und

iv. $C \models (\mathsf{mailbox}_l(x,i) \wedge \mathsf{in}_l(j) \wedge i > j) \Rightarrow \mathsf{mailbox}_{l2}[i] = 1$

Wir werden in Satz 5.46 beweisen, daß jeder Ablauf von Σ_7 schließlich einen semistabilen Zustand erreicht. Vorher führen wir noch eine hilfreiche Notation ein: $\mathit{mmax}(M)$ bezeichnet den maximalen Zeitstempel einer endlichen Multimenge von Nachrichten.

Notation 5.41 Für eine endliche Multimenge M der Sorte message bezeichnen wir mit $\mathit{mmax}(M)$ die kleinste natürliche Zahl, so daß für alle $(x,i) \in M$ gilt: $i \leqslant \mathit{mmax}(M)$.

Lemma 5.42 $\Sigma_7 \models \Box\, \forall i > mmax(\mathsf{M_r}) : \big((\mathsf{out}_l(i) \wedge \mathsf{mailbox}_{\mathsf{r2}}(j)) \Rightarrow i > j\big)$

Beweis: Initial gilt diese Zusicherung. $\mathsf{t_4}$ erhält die Gültigkeit. $\mathsf{t_2}$ kann den out-Zähler nicht verringern. $\mathsf{t_5}$ kann den maximalen Zeitstempel auf $\mathsf{mailbox_r}$ nicht erhöhen. Alle anderen Transitionen sind irrelevant. □

Lemma 5.43 $\Sigma_7 \models \Box\, \forall i > mmax(\mathsf{M}_l) : \big((\mathsf{out_r}(i) \wedge \mathsf{mailbox}_{l2}(j)) \Rightarrow i > j\big)$

Beweis: Analog zu Lemma 5.42. □

Lemma 5.44 $\Sigma_7 \models \Box\, \forall i > mmax(\mathsf{M_r}) : \mathsf{out}_l[i] + \mathsf{mailbox}_{\mathsf{r2}}[i] \leqslant 1$

Beweis: Initial gilt diese Zusicherung. $\mathsf{t_4}$ erhält zusammen mit (5.17) und Lemma 5.42 die Gültigkeit der Zusicherung. $\mathsf{t_2}$ erhält wegen Lemma 5.42 die Gültigkeit der Zusicherung. Keine andere Transition legt Marken auf out_l oder $\mathsf{mailbox_r}$. □

Lemma 5.45 $\Sigma_7 \models \Box\, \forall i > mmax(\mathsf{M}_l) : \mathsf{out_r}[i] + \mathsf{mailbox}_{l2}[i] \leqslant 1$

Beweis: Analog zu Lemma 5.44. □

Satz 5.46 $\Sigma_7 \models \Diamond\, \Box$ *semistabil*

Beweis: Wir beweisen separat die vier Teileigenschaften der Zustandseigenschaft *semistabil* (vergleiche Def. 5.40). Die Gültigkeit der Behauptung ergibt sich dann durch eine allgemeine Regel unserer temporalen Logik (siehe Lemma 4.20).

Eigenschaft (i). Wegen Lemma 5.34 gilt:

$$\Sigma_7 \models \Diamond\big(\mathsf{out}_l(i) \wedge i > mmax(\mathsf{M_r})\big) \tag{5.18}$$

Zusammen mit Lemma 5.42 folgt:

$$\Sigma_7 \models \Diamond\, \Box\big((\mathsf{out}_l(i) \wedge \mathsf{mailbox_r}(x, j)) \Rightarrow i > j\big) \tag{5.19}$$

Eigenschaft (ii) gilt analog wegen Lemma 5.35 und Lemma 5.43.
Eigenschaft (iii). Wegen Lemma 5.36 gilt:

$$\Sigma_7 \models \Diamond\big(\mathsf{in_r}(j) \wedge j > mmax(\mathsf{M_r})\big) \tag{5.20}$$

Zusammen mit Lemma 5.44 folgt:

$$\Sigma_7 \models \Diamond\, \Box\big((\mathsf{mailbox_r}(x, i) \wedge \mathsf{in_r}(j) \wedge i > j) \Rightarrow \mathsf{mailbox}_{\mathsf{r2}}[i] = 1\big) \tag{5.21}$$

Eigenschaft (iv) gilt analog wegen Lemma 5.37 und Lemma 5.45.
Diese vier Eigenschaften implizieren die Behauptung. □

Im nächsten Teil des Beweises führen wir eine Abstraktion ein. Zur Illustration dieser Abstraktion stellen wir uns einen Ablauf von Σ_7 vor, in dem irgendwann auf $\mathsf{mailbox_r}$ zwei Nachrichten $(0, 27)$ und $(0, 28)$ liegen, von denen eine empfangen wird. Egal welche Nachricht empfangen wird, die andere wird in jedem Fall nicht empfangen (da der rechte Agent abwechselnd Nachrichten mit verschiedenen Werten empfängt). Wir abstrahieren daher vom Empfang konkreter Nachrichten. Wir sagen, daß $(0, 27)$ und $(0, 28)$ zu einer *Welle* gehören. Es werden also Wellen gesendet und empfangen. Jede Welle hat einen Wert ($0, 1$ oder 2).

Um die Definition von Wellen zu vereinfachen, definieren wir zuerst *last* als den Wert, der jeweils von einem Agenten als aktuellster Wert empfangen wurde:

Definition 5.47 (*last*) Für einen Zustand C von Σ_7 und $x \in \{0, 1, 2\}$ definieren wir $C \models last_r = x$, falls $C \models \mathsf{sync_r}(x) \vee \mathsf{token_r}(x) \vee \mathsf{pending_r}(x)$. Weiterhin definieren wir $C \models last_l = x$, falls $C \models \mathsf{sync}_l(x) \vee \mathsf{token}_l(x) \vee \mathsf{pending}_l(f(x))$.

Definition 5.48 (x-Welle auf einem Kanal) Für einen Zustand C von Σ_7 und $x \in \{0, 1, 2\}$ heißt ein abgeschlossenes Intervall $I = [a, b]$ natürlicher Zahlen *x-Welle auf* $\mathsf{mailbox_r}$, falls gilt:

i. $a > mmax(\mathsf{M_r})$,

ii. $C \models \mathsf{mailbox_r}(x, a) \wedge \mathsf{mailbox_r}(x, b)$,

iii. $C \models \forall\, a \leqslant i \leqslant b : (\mathsf{mailbox_r}(y, i) \Rightarrow x = y)$,

iv. $C \models \mathsf{in_r}(j) \Rightarrow \big(b \geqslant j \vee (\mathsf{out}_l(j+1) \wedge \mathsf{pending}_l(x))\big)$,

v. $C \models (\mathsf{in_r}(j) \wedge last_r(x)) \Rightarrow \exists\, k \in \mathbb{N} : \exists\, y \neq x : (j \leqslant k < a \wedge \mathsf{mailbox_r}(y, k))$, und

vi. I ist maximal bezüglich i. bis v.

Wir definieren *x-Wellen auf* $\mathsf{mailbox}_l$ analog, d. h. wie oben, nur mit vertauschten Indizes.

Da die Definition 5.48 sehr technisch ist, erläutern wir die einzelnen Bestandteile im folgenden. Eine x-Welle ist ein maximales Intervall, daß im Ablauf erzeugt wurde, also initial nicht vorhanden ist (i.). Das minimale und das maximale Element entsprechen beide einer Nachricht mit dem Wert x und einem entsprechenden Zeitstempel (ii.). Wir fordern in iii., daß keine „falsche“ Nachricht im Intervall liegt. Wir fordern nicht, daß jeder Zeitstempel zwischen

a und b auch durch eine Nachricht vertreten ist, da es tatsächlich während der Stabilisierungsphase zu „Lücken“ innerhalb eines solchen Intervalls kommen kann. Bedingung iv. und v. beschreiben, daß wir nur aktuelle Wellen betrachten, also solche, die irgendwann empfangen werden können. Hier sind einige Spezialfälle zu betrachten: Die Welle könnte noch fortgesetzt werden (iv.), oder sie wurde bereits empfangen (v.). Bedingung vi. fordert die bereits erwähnte Maximalität.

Wir beschreiben jetzt informell den weiteren „Fahrplan“ unseres Beweises. Wir betrachten hierzu eine 0-Welle, die mit $\mathsf{t_5}$ empfangen wurde. Sie wird auf $\mathsf{sync_r}$ durch eine 0 repräsentiert. Im weiteren Ablauf wird sie auf $\mathsf{token_r}$ durch eine 0 repräsentiert. Schließlich wird sie auf $\mathsf{pending_r}$ durch eine 0 solange repräsentiert, wie die Werte auf $\mathsf{out_r}$ und $\mathsf{in_r}$ sich nicht unterscheiden. Danach erscheint sie als 0-Welle auf $\mathsf{mailbox}_l$, usw. Eine Welle kann inaktuell werden, wenn sie vollständig gesendet wurde und der Zielagent seinen in-Zähler erhöht. Wir werden zeigen, daß in jedem Ablauf von Σ_7 irgendwann die Anzahl der (aktuellen) Wellen konstant ist. Darüberhinaus zeigen wir, daß die Anzahl der Wellen schließlich konstant 1 ist. Hieraus können wir dann die Stabilisierung schließen. Wir formalisieren nun die Repräsentation einer Welle bei den Agenten. Wir definieren diese Repräsentation derart, daß sich der Wert einer Welle nur an einer Stelle ändert – beim Schalten von $\mathsf{t_3}$.

Definition 5.49 (x-Welle bei einem Agenten) In einem Zustand C von Σ_7 und $x \in \{0,1,2\}$ *hat der rechte Agent eine x-Welle*, falls gilt:

$$C \models \mathsf{sync_r}(x) \vee \mathsf{token_r}(x) \vee \big(\mathsf{pending_r}(x) \wedge \exists\, n \in \mathbb{N} : \mathsf{out_r}(n) \wedge \mathsf{in_r}(n)\big)$$

Der linke Agent hat eine x-Welle, falls gilt:

$$C \models \mathsf{sync}_l(x) \vee \mathsf{token}_l(x) \vee \big(\mathsf{pending}_l(x) \wedge \exists\, n \in \mathbb{N} : \mathsf{out}_l(n) \wedge \mathsf{in}_l(n)\big)$$

Definition 5.50 (Wellenanzahl) Für einen Zustand C von Σ_7 ergibt die Summe aller Wellen auf $\mathsf{mailbox_r}$ und $\mathsf{mailbox}_l$, erhöht um jeweils 1 für jeden Agenten, der eine Welle hat, die *Wellenanzahl*. Für eine Wellenanzahl von n notieren wir $C \models \mathit{waves} = n$.

Lemma 5.51 $\Sigma_7 \models \exists\, n \in \mathbb{N} : \Diamond\Box(\mathit{waves} = n)$

Beweis: Wegen Satz 5.46 gilt, daß in jedem Ablauf irgendwann ein semistabiler Zustand erreicht wird. Wir zeigen für jede Transition, daß ihr Schalten in einem semistabilen Zustand die Wellenanzahl nicht erhöht. Beim Schalten von $\mathsf{t_1}$ in einem semistabilen Zustand wird wegen Definition 5.48 die Wellenanzahl auf $\mathsf{mailbox}_l$ um 1 verringert. Der linke Agent hat nach dem Schalten eine

Welle (Definition 5.49). Also steigt die Wellenanzahl nicht an. Beim Schalten von t_2 und t_3 bleibt die Wellenanzahl konstant. Beim Schalten von t_4 ist nur der Fall interessant, in dem t_4 direkt nach t_3 schaltet, da nur in diesem Fall eine neue Welle auf $\mathsf{mailbox}_\mathsf{r}$ erzeugt werden kann. Dann gilt aber vor dem Schalten auch, daß der in- und der out-Zähler des linken Agenten gleich sind. Also hatte vorher der linke Agent eine Welle (Definition 5.49), und die Wellenanzahl bleibt konstant. Der Beweis für die anderen Transitionen geht analog. Die Behauptung folgt nun aus der Fundiertheit der natürlichen Zahlen.

Theorem 5.52 $\Sigma_7 \models \Diamond\,\Box(|\mathsf{token}_l + \mathsf{token}_\mathsf{r}| \leqslant 1)$

Beweis: Wegen Lemma 5.51 gibt es in jedem Ablauf ein $n \in \mathbb{N}$, so daß schließlich die Anzahl der Wellen konstant n ist. Dies impliziert, daß jede Welle, die gesendet wird, schließlich auch empfangen wird. Wegen Lemma 5.36 versendet der linke Agent immer wieder nacheinander 0-Wellen, 1-Wellen und 2-Wellen. Wir betrachten nun eine Situation, in der der linke Agent im Zustand $\mathsf{pending}_l$ ist und eine 0-Welle $[a, b]$ vollständig versendet hat. Es gilt also $\mathsf{out}_l = [b + 1]$. Weiterhin gilt, daß der Wert von in_l nicht kleiner als a ist.

Er wird also als nächste Aktion die Transition t_1 im Modus $x = 0$ schalten, also eine 0-Welle empfangen. In diesem Zustand gibt es eine Anzahl n von Wellen auf dem Weg von $\mathsf{pending}_l$ über $\mathsf{mailbox}_\mathsf{r}$, sync_r, $\mathsf{token}_\mathsf{r}$, $\mathsf{pending}_\mathsf{r}$ und $\mathsf{mailbox}_l$. Konkret gilt $(n \bmod 3) = 1$, da die Wellen nacheinander gesendet werden und ihren Wert auf diesem Weg nicht ändern. Wir werden indirekt beweisen, daß die Wellenanzahl 1 ist.

Angenommen, die Wellenanzahl ist ungleich 1. Dann ist sie also mindestens 4. Wir werden den Fall 4 betrachten, da sich der allgemeine Fall hieraus leicht kanonisch ableiten läßt. Es gibt also die 0-Welle $[a, b]$ auf $\mathsf{mailbox}_\mathsf{r}$, eine 2-Welle, eine 1-Welle und eine weitere 0-Welle, die wir zur Unterscheidung $0'$-Welle nennen.

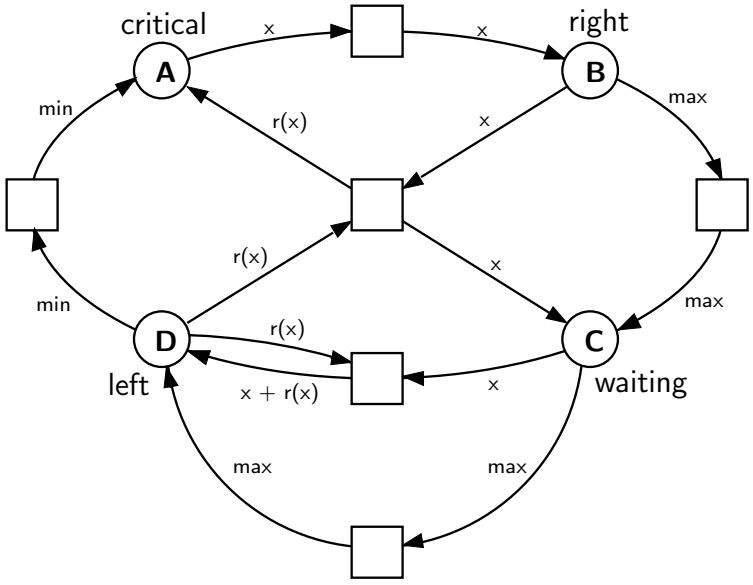
Nachdem der rechte Agent die $0'$-Welle versendet hat, ist der out-Zähler des rechten Agenten größer als der in-Zähler des linken Agenten. Nachdem der rechte Agent die 1-Welle versendet hat, ist sein out-Zähler größer als $b + 1$, da sich die Zähler des linken Agenten beim Empfang der $0'$-Welle synchronisieren und demnach der Wert des in-Zählers des linken Agenten vor dem Empfang der 1-Welle nicht kleiner als $b + 1$ ist.

Nach dem Empfang der 2-Welle synchronisieren sich die beiden Zähler des rechten Agenten. Also gilt vor dem Empfang der 0-Welle, daß der in-Zähler des rechten Agenten größer als $b + 1$ ist. Damit kann aber die 0-Welle nicht empfangen werden. Dies ist ein Widerspruch. Daher ist die Anzahl der Wellen gleich 1. Dies impliziert mit Definition 5.50 die Behauptung. □

Im Beweis wurden auch die Auswirkungen der Synchronisation deutlich: Drei Wellen mußten vom rechten Agenten empfangen werden, um zu garantieren, daß sein in-Zähler mindestens den ursprünglichen Wert des out-Zählers des linken Agenten hat. Mit nur zwei verschiedenen Wellenwerten kann die Eindeutigkeit der Wellen nicht garantiert werden.

5.3 Der Algorithmus von Brown, Gouda und Wu

In Abbildung 5.7 ist das Petrinetzmodell des Algorithmus noch einmal angegeben. Die wesentlichen Eigenschaften haben wir wie folgt spezifiziert:



spec (U,r,min,max) : unidirectional chain var x : site
const A, B, C, D : multiset of sites A + B + C + D = U

Abbildung 5.7: Σ_8 – Tokenaustausch in Ketten

$$\Sigma_8 \models \forall x \in U : true \mapsto \mathsf{critical}(x) \tag{5.22}$$

$$\Sigma_8 \models \Diamond \Box \, |\mathsf{critical}| \leqslant 1 \tag{5.23}$$

Jeder Agent befindet sich immer in einem der vier Zustände critical, right, waiting, left:

Lemma 5.53 $\Sigma_8 \models \Box(\mathsf{critical} + \mathsf{right} + \mathsf{waiting} + \mathsf{left} = U)$

Beweis: $\mathsf{critical} + \mathsf{right} + \mathsf{waiting} + \mathsf{left} = U$ ist eine Stelleninvariante von Σ_8. □

Σ_8 ist deadlockfrei:

Lemma 5.54 $\Sigma_8 \models \Box\, \neg \mathit{lockout}$

Beweis: *lockout* impliziert, daß kein Agent im Zustand critical ist. Weiterhin ist min nicht im Zustand left. max ist nicht im Zustand right oder waiting. Also ist max im Zustand left und min in einem der Zustände waiting oder right. Daher gibt es zwei benachbarte Agenten $x, r(x) \in U$, so daß $\mathsf{waiting}(x) \vee \mathsf{right}(x)$ und $\mathsf{left}(r(x))$ gilt. In jedem Fall ist eine Transition aktiviert. Also gilt *lockout* in keinem Zustand. □

Notation 5.55 Für einen Agenten $x \in U \setminus \{min\}$ bezeichnet $l(x)$ den linken Nachbarn von x, d. h. $r(l(x)) = x$.

Als nächstes definieren wir formal, wann ein Agent ein Token hat. Aus der informellen Beschreibung des Algorithmus ist bereits klar, daß ein Agent ein Token hat, wenn er sich im Zustand critical befindet. Die weiteren Zustände beschreiben die „Treppenlinie“, die wir bereits im Ablauf in Abbildung 3.19 beobachtet haben.

Definition 5.56 (Token) Sei $u \in U \setminus \{min, max\}$. Für einen Zustand C von Σ_8 gilt:

i. $C \models \mathit{token}(u)$ gdw. $C \models \mathsf{critical}(u) \vee \mathsf{right}(u) \vee (\mathsf{left}(u) \wedge \mathsf{waiting}(l(u)))$

ii. $C \models \mathit{token}(min)$ gdw. $C \models \mathsf{critical}(min) \vee \mathsf{right}(min) \vee \mathsf{left}(min)$

iii. $C \models \mathit{token}(max)$ gdw. $C \models \mathsf{critical}(max) \vee \mathsf{right}(max) \vee \mathsf{waiting}(max) \vee (\mathsf{left}(max) \wedge \mathsf{waiting}(l(max)))$

Wir schreiben im folgenden *token* für $\{x \in U : \mathit{token}(x)\}$.

Wir werden zeigen: Es gibt immer mindestens ein Token im System, und die Anzahl der Token im System nimmt nicht zu.

Lemma 5.57 $\Sigma_8 \models \Box\, |\mathit{token}| > 0$

Beweis: Jedes Schalten einer Transition von Σ_8 setzt die Existenz eines Tokens voraus. (Dies kann am Netz einfach syntaktisch überprüft werden.) Da jeder Ablauf von Σ_8 unendlich ist (Lemma 5.54), folgt die Behauptung. □

Lemma 5.58 $\Sigma_8 \models \forall k \in \mathbb{N} : \Box(|token| = k \Rightarrow \Box\,|token| \leqslant k)$

Beweis: Jedes Schalten einer Transition setzt die Existenz eines Tokens bei einem der beteiligten Agenten voraus. Wir können wiederum syntaktisch am Netz überprüfen, daß jedes Auftreten einer Transition mit zwei beteiligten Nachbarn x und $r(x)$ in einem Zustand resultiert, in dem einer der beteiligten Agenten kein Token hat. □

Lemma 5.57 besagt intuitiv, daß immer mindestens ein Token existiert. Lemma 5.58 besagt intuitiv, daß Token nie erzeugt werden. Also gibt es ein Token, das *unendlich oft bewegt wird* (da jeder Ablauf unendlich ist und jede Transition ein Token bewegt).

Wir zeigen, daß dieses Token sich intuitiv wie eine *Patrouille* verhält – es geht auf und ab in der Kette der Agenten und besucht dabei jeden einzelnen. Hierzu definieren wir zunächst die *Richtung eines Tokens*.

Definition 5.59 (Richtung eines Tokens) Für einen Zustand C eines Ablaufs von Σ_8 sagen wir, daß ein Agent $x \in U$ ein *Rechtstoken* hat, gdw. $C \models token(x) \wedge (\mathsf{critical}(x) \vee \mathsf{right}(x) \vee \mathsf{waiting}(x))$. Ein Agent x hat ein *Linkstoken*, gdw. $C \models token(x) \wedge \mathsf{left}(x)$.

Wir können nun die wesentlichen Lebendigkeitseigenschaften unseres Algorithmus beweisen:

Theorem 5.60 $\Sigma_8 \models \forall x \in U : true \mapsto \mathsf{critical}(x)$

Beweis: Lemma 5.57 und Lemma 5.58 implizieren, daß es in jedem Ablauf von Σ_8 ein Token gibt, daß unendlich häufig an einer Transition beteiligt ist. Daher wird dieses Token auch entweder unendlich oft ein Linkstoken oder unendlich oft ein Rechtstoken sein. Wir betrachten im folgenden das Verhalten dieses Tokens.

Wir betrachten zunächst das Verhalten eines Linkstokens: Ein Linkstoken existiert, wenn ein Agent $x \neq min$ im Zustand left ist und sein linker Nachbar $l(x)$ sich im Zustand $\mathsf{waiting}$ befindet. In diesem Fall besteht die einzige Möglichkeit, das Token zu bewegen darin, daß $l(x)$ in den Zustand left übergeht. Danach hat $l(x)$ das Token, da x es nicht mehr hat und das Token nicht verschwindet.

Das heißt, das Linkstoken wandert solange nach links, bis es beim Agenten min ankommt. min dreht die Richtung eines Linkstokens. Insbesondere ist min auch der einzige Agent, der die Richtung eines Tokens von links auf rechts dreht.

Falls das unendlich oft bewegte Token ein Rechtstoken im Zustand critical bei einem Agenten $x \in U$ ist, wird es zu einem Rechtstoken bei x im Zustand right. Für $x \neq max$ wird es danach ein Rechtstoken für $r(x)$ im Zustand critical (wiederum, weil das Token irgendwann bewegt wird). Ein Rechtstoken wandert also nach rechts, bis es bei max angelangt ist. Dort wird das Token zum Linkstoken, indem max über die Zustände right und waiting in den Zustand left übergeht.

Zusammengefaßt gibt es ein Token, das unendlich oft durch die Kette der Agenten bewegt wird und nur bei min oder max die Richtung wechselt. Bei jedem Durchlauf des Tokens nach rechts wird jeder Agent $x \in U$ unter anderem im Zustand critical(x) durchlaufen. Das impliziert die Behauptung. □

Theorem 5.61 $\Sigma_8 \models \Diamond \Box\, |\mathsf{critical}| \leqslant 1$

Beweis: Wir betrachten wiederum das unendlich oft bewegte Token aus Theorem 5.60. Wegen Lemma 5.53 gibt es höchstens ein Token pro Agent. Da das betrachtete Token auf seiner Patrouille keinen anderen Token begegnet und keine Token erzeugt werden können, gibt es keine weiteren Token. Dies ist in einer Kette von Agenten offensichtlich. Dies impliziert direkt die Behauptung. □

Die Beweise in [BGW89] und [Rei98]

Unser Beweis ist neu. Er folgt dem in Abschnitt 4.4 vorgeschlagenen Verifikationsschema. Der originale Beweis von Brown, Gouda und Wu benutzt zum Beweis der Stabilisierung eine Kombination aus Induktion und einer Stabilisierungsfunktion, welche beweist, daß irgendwann ein Grundzustand erreicht wird, in dem alle Agenten im Zustand left sind. Von dort aus wird wiederum induktiv mit derselben Stabilisierungsfunktion bewiesen, daß die Agenten reihum in den Zustand critical kommen, bis der Grundzustand wieder erreicht wird. Die Stabilisierungsfunktion bildet in eine lexikographische Ordnung von $|U|$-vielen natürlichen Zahlen ab.

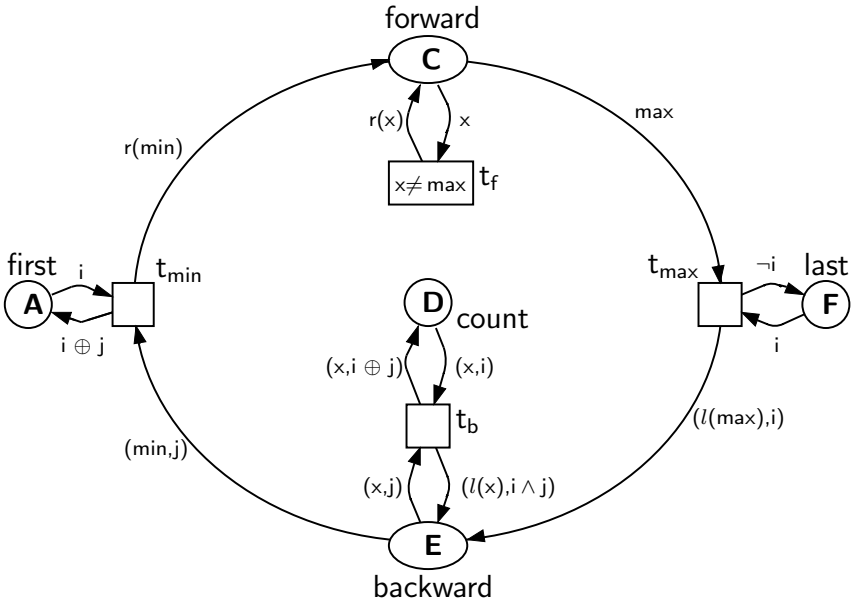
Reisig kommt ohne eine explizite Stabilisierungsfunktion aus. Er beweist die Stabilisierung induktiv über die Länge der Kette. (Intuitiv stabilisiert sich die Kette von rechts nach links.)

Durch unsere gewählte Abstraktion der sich in der Kette bewegenden Token verschieben wir die Induktion auf eine einfache Aussage im Netzwerk. So bedarf der Beweis, daß sich zwei Token treffen müssen, wenn eines von ihnen in der Kette patrouilliert, natürlich auch einer Induktion. Diese ist aber unabhängig von den konkreten Aktionen des Algorithmus und kann separat bewiesen werden.

Wir geben ebenfalls eine Stabilisierungsfunktion an ($|token|$). Sie ist jedoch einfacher als die von Brown, Gouda und Wu, da sie nur in die natürlichen Zahlen abbildet. Auch benutzen wir die Stabilisierungsfunktion anders. Während sie Brown, Gouda und Wu zum Nachweis nutzen, daß eine Transition irgendwann schaltet, weisen wir nach, daß bestimmte Transitionen irgendwann nicht mehr schalten (bzw. eine bestimmte Eigenschaft globaler Zustände nur endlich oft auftritt), konkret, daß Token nicht mehr verschwinden.

5.4 Verteilter Zähler

In Abbildung 5.8 ist das Petrinetzmodell unseres Algorithmus zum verteilten Zählen in Ketten noch einmal angegeben. Zusätzlich haben wir den Transi-



spec (U,r,l,min,max) : bidirectional chain
sort value = site × bool
const A, F : bool
const C : set of sites
const D, E : set of values
var x : site

var i, j : bool
$C \subseteq U \backslash \{min\}$
$D_1 = U \backslash \{min,max\}$
$E_1 \subseteq U \backslash \{max\}$
$\exists x \in U : C + E_1 = [x]$

Abbildung 5.8: Σ_{10} – Verteilter Zähler in einer Kette

tionen explizit Namen gegeben. Die wesentlichen Eigenschaften haben wir wie

folgt spezifiziert (α steht für den globalen Zählerstand, vergleiche Definition 3.8):

$$\Sigma_{10} \models \forall h, k \in \mathbb{N} : \Box((\alpha = h \wedge \alpha = k) \Rightarrow h = k) \tag{5.24}$$

$$\Sigma_{10} \models \forall k \in \{0, \ldots, 2^n\} : \mathit{true} \mapsto \alpha = k \tag{5.25}$$

$$\Sigma_{10} \models \Diamond \Box\, \alpha \leqslant 2^n \tag{5.26}$$

Die Sicherheitseigenschaft (5.24) gilt offensichtlich:

Satz 5.62 $\Sigma_{10} \models \forall h, k \in \mathbb{N} : \Box((\alpha = h \wedge \alpha = k) \Rightarrow h = k)$

Beweis: Durch die Definition von α entspricht jeder Zustand genau einer natürlichen Zahl. □

Für den Beweis der Lebendigkeitseigenschaften zeigen wir in Satz 5.65, daß jeder Ablauf von Σ_{10} unendlich ist. Hierzu leiten wir zunächst die Gültigkeit einiger Basiseigenschaften von Stelleninvarianten ab:

$$\Sigma_{10} \models \Box\, |\mathsf{forward} + \mathsf{backward_1}| = 1 \tag{5.27}$$

$$\Sigma_{10} \models \Box\, |\mathsf{first}| = 1 \tag{5.28}$$

$$\Sigma_{10} \models \Box\, |\mathsf{last}| = 1 \tag{5.29}$$

$$\Sigma_{10} \models \Box\, \mathsf{count_1} = U \setminus \{min, max\} \tag{5.30}$$

Die beiden folgenden Eigenschaften sind nicht mit einer Stelleninvariante beweisbar (da Eigenschaften, die mit einer Stelleninvarianten beweisbar sind, unsensibel gegen „Rückwärtsschalten“ einer Transition sind, siehe z. B. [Wal95]. Die Gültigkeit der folgenden Eigenschaft geht verloren, falls die Transition $\mathsf{t_f}$ im Modus $x = min$ rückwärts schaltet.):

Lemma 5.63 $\Sigma_{10} \models \Box\, \neg \mathsf{forward}(min)$

Beweis: Die Behauptung ist durch eine Zusicherung beweisbar. Im Anfangszustand gilt $C \subseteq U \setminus \{min\}$. Jede Marke, die von einer Transition auf die Stelle forward gelegt wird, ist im Wertebereich der Funktion r. Der Agent min ist nicht im Wertebereich der Funktion r. □

Lemma 5.64 $\Sigma_{10} \models \Box\, \neg \mathsf{backward}_1(max)$

Beweis: Analog zum Beweis von Lemma 5.63. □

Satz 5.65 $\Sigma_{10} \models \Box\, \neg lockout$

Beweis: Eigenschaft (5.27) impliziert zusammen mit Lemma 5.63 und Lemma 5.64:

$$\Sigma_{10} \models \Box\, \exists\, x \in U : \Big(\big(\mathsf{forward}(x) \wedge x \neq min\big) \vee \big(\mathsf{backward}_1(x) \wedge x \neq max\big)\Big) \tag{5.31}$$

Im einen Fall ist mit der Eigenschaft (5.29) eine der Transitionen $\mathsf{t_f}$ oder $\mathsf{t_{max}}$ aktiviert. Im anderen Fall ist mit der Eigenschaft (5.28) eine der Transitionen $\mathsf{t_b}$ oder $\mathsf{t_{min}}$ aktiviert. □

Im folgenden Teil unseres Beweises werden wir in Satz 5.68 eine wesentliche Lebendigkeitseigenschaft von Σ_{10} zeigen: Es gibt in jedem Ablauf immer wieder eine Nachricht auf $\mathsf{forward}$.

Lemma 5.66 $\Sigma_{10} \models \mathsf{backward}_1(x) \mapsto \mathsf{backward}_1(min)$

Beweis: Muster 11, angewendet auf $\mathsf{t_b}$, liefert:

$$\Sigma_{10} \models \big(\mathsf{backward}_1(x) \wedge x \neq min\big) \mapsto \mathsf{backward}_1(l(x))$$

Mit Induktion gilt daher auch die Behauptung. □

Lemma 5.67 $\Sigma_{10} \models \mathsf{backward}_1(min) \mapsto \mathsf{forward}(r(min))$

Beweis: Muster 12, angewendet auf $\mathsf{t_{min}}$, liefert die Behauptung. □

Satz 5.68 $\Sigma_{10} \models true \mapsto \exists\, x \in U : \mathsf{forward}(x)$

Beweis: Abbildung 5.9 zeigt einen Beweisgraph. Seine Knoten sind durch folgende Argumente gerechtfertigt:
Knoten 1: Eigenschaft (5.31).
Knoten 2: Lemma 5.66.
Knoten 3: Lemma 5.67. □

Im folgenden betrachten wir jedes Interleaving von Σ_{10} als eine Folge globaler Zählerstände, also als eine Folge natürlicher Zahlen. Wir untersuchen Eigenschaften dieser Zahlenfolge; insbesondere die Differenz zweier aufeinanderfolgender Zählerzustände.

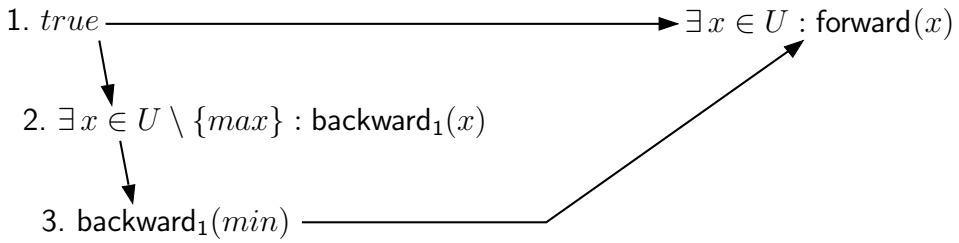


Abbildung 5.9: Ein Beweisgraph für $true \mapsto \exists\, x \in U$: forward(x)

Lemma 5.69 Für jeden Zustand C von Σ_{10}, mit $C \models \alpha = k$, und jeden Schritt $(C, (\mathsf{t_f}, \beta), C')$ gilt: $C' \models \alpha = k$.

Beweis: Das Schalten von $\mathsf{t_f}$ berührt keine Stelle, die zum Wert von α beiträgt. □

Lemma 5.70 Für jeden Zustand C von Σ_{10}, mit $C \models \alpha = k$, und jeden Schritt $(C, (\mathsf{t_b}, \beta), C')$ gilt: $C' \models \alpha = k$.

Beweis: Das Schalten von $\mathsf{t_b}$ im Modus $\beta(\mathsf{x}) = y$ ändert die Summe der (involvierten) lokalen Zählerstände $\alpha'(y) + \alpha'(l(y))$ nicht (vgl. Definition 3.8). Daher ändert sich auch nicht der globale Zählerstand α. □

Lemma 5.71 Für jeden Zustand C von Σ_{10}, mit $C \models \alpha = k$, und jeden Schritt $(C, (\mathsf{t_{max}}, \beta), C')$ gilt: $C' \models \alpha = k + 1$.

Beweis: Offensichtlich mit Definition 3.8. □

Lemma 5.72 Für jeden Zustand C von Σ_{10}, mit $C \models \alpha = k$, und jeden Schritt $(C, (\mathsf{t_{min}}, \beta), C')$ gilt: $C' \models \begin{cases} \alpha = k & \text{für } \beta(\mathsf{i}) + \beta(\mathsf{j}) < 2, \\ \alpha = k - 2^n & \text{sonst.} \end{cases}$

Beweis: Die einzige Möglichkeit zur Änderung des globalen Zählerstandes beim Schalten von $\mathsf{t_{min}}$ besteht darin, daß der Modus β den Variablen i und j jeweils 1 zuweist. Hierbei wird der globale Zählerstand um 2^n verringert. □

Lemma 5.69 bis 5.72 zusammengefaßt, gibt es drei Möglichkeiten für zwei aufeinanderfolgende Werte der betrachteten Zahlenfolgen:

1. beide Werte sind gleich,

2. der zweite Wert ist um 1 größer als der erste Wert, oder

3. der zweite Wert ist um 2^n kleiner als der erste Wert.

Wir verbinden nun Eigenschaften des Petrinetzmodells mit Eigenschaften der Zahlenfolgen:

Lemma 5.73 $\Sigma_{10} \models \Box(\mathsf{forward}(x) \Rightarrow \alpha < 2^n)$

Beweis: folgt aus Definition 3.8 und (5.27). □

Korollar 5.74 $\Sigma_{10} \models \Box(\alpha = 2^n \Rightarrow \mathsf{forward} = [\,])$

Beweis: folgt aus der Kontraposition von Lemma 5.73. □

Beim globalen Zählerstand $\alpha = 2^n$ kann $\mathsf{t_{max}}$ daher nicht schalten. Für diesen Fall reduziert sich die Anzahl der Möglichkeiten nachfolgender Zahlen in der Zahlenfolge auf die erste und die dritte Möglichkeit. Intuitiv kann also unser Zähler bei einem Zählerstand von 2^n nicht inkrementiert werden. Daher gilt offensichtlich, daß kein Zählerstand $\alpha > 2^n$ von einem Zählerstand $\alpha \leqslant 2^n$ erreichbar ist:

Lemma 5.75 $\Sigma_{10} \models \Box(\alpha \leqslant 2^n \Rightarrow \Box\,\alpha \leqslant 2^n)$

Beweis: Die Behauptung folgt aus den Lemmas 5.69 bis 5.72, Korollar 5.74 sowie der Fundiertheit der natürlichen Zahlen. □

Lemma 5.76 $\Sigma_{10} \models \Diamond\,\alpha \leqslant 2^n$

Beweis: Satz 5.68 impliziert:

$$\Sigma_{10} \models \Diamond\,\exists\, x \in U : \mathsf{forward}(x)$$

Mit Lemma 5.73 folgt die Behauptung. □

Theorem 5.77 $\Sigma_{10} \models \Diamond\,\Box\,\alpha \leqslant 2^n$

Beweis: Die Behauptung folgt mit Lemma 5.75 und Lemma 5.76 aus einer allgemeinen Regel unserer temporalen Logik (siehe Lemma 4.20). □

Die Gültigkeit der Lebendigkeitseigenschaft (5.25) führen wir wiederum auf eine Eigenschaft der von Σ_{10} generierten Zahlenfolgen zurück.

Lemma 5.78 $\Sigma_{10} \models \mathsf{forward}(x) \mapsto \mathsf{forward}(max)$

Beweis: Muster 13, angewendet auf $\mathsf{t_f}$, liefert:

$$\Sigma_{10} \models \big(\mathsf{forward}(x) \wedge x \neq max\big) \mapsto \mathsf{forward}(r(x))$$

Mit Induktion folgt die Behauptung. □

Das folgende Lemma sichert, daß der globale Zählerstand immer wieder inkrementiert wird:

Lemma 5.79 $\Sigma_{10} \models \mathit{true} \mapsto \mathsf{forward}(max)$

Beweis: Die Behauptung folgt aus Satz 5.68 und Lemma 5.79. □

Wir haben nun gezeigt, daß der globale Zählerstand in jedem Ablauf tatsächlich immer wieder inkrementiert wird. Zusammen mit der Stabilisierungseigenschaft, die durch Theorem 5.77 gegeben ist, folgt, daß jeder Zählerstand zwischen 0 und 2^n in jedem Ablauf immer wieder auftritt:

Theorem 5.80 $\Sigma_{10} \models \forall k \in \{0, \ldots, 2^n\} : \mathit{true} \mapsto \alpha = k$

Beweis: Die Behauptung folgt mit Induktion aus den Lemmas 5.69 bis 5.72, Theorem 5.77 und Lemma 5.79. □

6 Ausblick

In dieser Arbeit haben wir neue selbststabilisierende Algorithmen vorgestellt. Für die Formalisierung des operationellen Modells und die Verifikation der entscheidenden Eigenschaften der Modelle kombinierten wir algebraische Petrinetze und linear-time temporale Logik, angelehnt an das Vorbild der Distributed Algorithms' Working Notation (DAWN). Zwei der neuen Algorithmen sind von herausragender Bedeutung. Der Algorithmus zur randomisierten Leader Election im unidirektionalen Ring verbessert einen fehlerhaften Algorithmus aus der Literatur. Unser Algorithmus für den asynchronen Tokenaustausch ist der erste selbststabilisierende Algorithmus, der ohne Time-Out-Aktionen auskommt. Beide Algorithmen werden auch unabhängig von dieser Arbeit veröffentlicht (z. B. in [Ves00]).

Wesentliche Bestandteile der in dieser Arbeit vorgeschlagenen Methodik sind eine einheitliche Form der Darstellung selbststabilisierender Algorithmen für verschiedene Systemannahmen sowie die besondere Betonung und intuitive Darstellung einer algorithmischen Idee, die sich in halbgeordneten Abläufen und einer Abstraktion im Beweis widerspiegelt. Im Vergleich zu DAWN neu sind:

- die Integration von Randomisierungsannahmen in den Formalismus,
- erweiterte Beweisgraphen, die auch den Nachweis anderer Lebendigkeitseigenschaften als Leads-to-Eigenschaften erlauben,
- Beweismuster als syntaktische und algorithmenunabhängige Methode zum Überprüfen von Beweisen sowie
- ein Verifikationsschema für selbststabilisierende Algorithmen, das einerseits die wesentliche Abstraktion in den Mittelpunkt eines Beweises stellt[1] und andererseits Fehler bei der Umsetzung einer algorithmischen Idee in

[1] *„Abstraction is what it's all about"* lehrte mich Anish Arora über die Verifikation selbststabilisierender Algorithmen.

einen fertigen Algorithmus vermeidet (da Basiseigenschaften eines Algorithmus ausgehend vom operationellen Modell bewiesen werden müssen).

Wir beschreiben im folgenden Ansätze, die in dieser Arbeit entwickelten Methoden weiterzuentwickeln. Ein wesentliches Problem selbststabilisierender Algorithmen ist die inhärente Beweiskomplexität. Hier helfen geeignete kompositionale Beweistechniken. So könnte man die algorithmischen Ideen als eigene Algorithmen auffassen, sie modellieren und verifizieren. Dies ist im wesentlichen dasselbe, was wir bei einer Komplexitätsanalyse machen: ein Problem auf ein bekanntes (z. B. Random Walk) zurückführen. Durch geeignete, möglichst syntaktische, kompositionale Beweistechniken kann man dann analog die Korrektheit eines selbststabilisierenden Algorithmus nach dem Beweis seiner Basiseigenschaften auf die Korrektheit der algorithmischen Idee zurückführen. Ansätze für kompositionale Beweistechniken für Petrinetze gibt es ([Kin95, KP99]), eine anwendbare und an Fallstudien validierte Theorie steht bisher noch aus. Eine andere interessante Möglichkeit der Weiterentwicklung der vorgestellten Techniken besteht darin, das Beweisen mit Mustern weiter zu untersuchen. Hier haben wir die Basismuster ad-hoc gewählt. Eine Formalisierung lohnte sich im Rahmen dieser Arbeit nicht; sie erscheint jedoch als sehr vielversprechend, da die Idee von Mustern (die ursprünglich aus der objekt-orientierten Programmierung stammt) sich in allen Bereichen der Informatik zunehmend verbreitet. Sie hilft dabei, formale Methoden anwendbar darzustellen.

Wir enden mit drei kleineren Abgrenzungen zu DAWN. Nicht jedes Design-Prinzip von DAWN kann ohne weiteres auch auf selbststabilisierende Algorithmen übertragen werden. Dies betrifft zum einen die Komplexität der Beweise, die eine Abstraktion erfordert. DAWN-Beweise sind prinzipiell automatisch durch Theorem-Beweiser überprüfbar. Für unsere Beweise nehmen wir dies nur für den Beweis der Basiseigenschaften in Anspruch (mit Ausnahme des Algorithmus zum verteilten Zählen). Das Ziel der automatischen Überprüfbarkeit geben wir deswegen jedoch nicht auf, der Weg führt aber nur über kompositionale Beweistechniken. Ein zweiter Unterschied zu DAWN besteht darin, daß wir halbgeordnete Abläufe nicht für Beweise benutzen. Wir benutzen sie stark bei der eigentlichen Entwicklung eines Algorithmus. Rundenfehler sind beispielsweise viel deutlicher darstellbar als in Interleavings. Der Grund dafür, daß wir keine einfachen Halbordnungs-basierten Beweise für unsere Algorithmen gefunden haben, mag darin liegen, daß ein Ablauf eines selbststabilisierenden Algorithmus im stabilen Fall stark synchronisiert ist. Hier ist also typischerweise keine Nebenläufigkeit zu erwarten. (Selbst beim asynchronen Tokenaustausch hatten wir auf einer abstrakteren Ebene schließlich nur eine

Welle, die die Synchronisation erzeugt.) In der Phase der Stabilisierung argumentieren wir über Stabilisierungsfunktionen und über ein Verhalten im Unendlichen. Stabilisierungsfunktionen sind unabhängig von einer Interleaving- oder Halbordnungssemantik. Ein dritter Unterschied zu DAWN besteht darin, daß wir keine syntaktischen Restriktionen für unsere Petrinetzmodelle festlegen. In DAWN wurden bisher hauptsächlich Netzwerkalgorithmen betrachtet: Das verteilte System ist asynchron, und die Agenten kommunizieren über Nachrichtenaustausch. Hierfür gibt es syntaktische Restriktionen, die eine Implementierbarkeit des Algorithmus garantieren. Für selbststabilisierende Algorithmen gilt, daß selten zwei Algorithmen auf derselben Architektur arbeiten – wir haben in dieser Arbeit beispielsweise mehrere Interpretationen gemeinsamen Speichers kennengelernt. Wir können jedoch mit unseren Petrinetzmodellen alle üblichen Systemannahmen ausdrücken, ein weiteres den Entwurf selbststabilisierender Algorithmen unterstützendes Resultat aus der Flexibilität von Petrinetzen.

Literaturverzeichnis

[AB93] Y. Afek und G. M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7:27–34, 1993.

[AD97] J. Abello und S. Dolev. On the computational power of self-stabilizing systems. *Theoretical Computer Science*, 182:159–170, 1997.

[AG93] A. Arora und M. G. Gouda. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19:1015–1027, 1993.

[AKL+79] R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovász und C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *Proceedings of the 20th Annual IEEE Symposium on the Foundations of Computer Science*, Seiten 218–223, 1979.

[Ang80] D. Angluin. Local and global properties in networks of processors (extended abstract). In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, Seiten 82–93, 1980.

[Aro92] Anish Arora. *A foundation of fault-tolerant computing.* Ph.D. dissertation, University of Texas at Austin, 1992.

[AS85] B. Alpern und F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, Oktober 1985.

[AUWY82] A. V. Aho, J. D. Ullman, A. D. Wyner und M. Yannakakis. Bounds on the size and transmission rate of communication protocols. *Computers and Mathematics With Applications*, 8(3):205–214, 1982.

[Bes95] Eike Best. *Semantik.* Vieweg Verlag, 1995.

[BF88] Eike Best und César Fernández. *Nonsequential Processes*, *EATCS Monographs on Theoretical Computer Science* 13. Springer-Verlag, 1988.

[BGJ99] J. Beauquier, M. Gradinariu und C. Johnen. Memory space requirements for self-stabilizing leader election protocols. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, Seiten 199–208, Mai 1999.

[BGM93] J. E. Burns, M. G. Gouda und R. E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7:35–42, 1993.

[BGW89] G. M. Brown, M. G. Gouda und C. L. Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38:845–852, 1989.

[BKV99] T. Baar, E. Kindler und H. Völzer. Verifying intuition – ILF checks DAWN proofs. In *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*, Seiten 404–423, 1999.

[BP89] J. E. Burnes und J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, April 1989.

[BSW69] K. A. Bartlett, R. A. Scantlebury und P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12:260–261, 1969.

[CHR93] L. Cherkasova, R. R. Howell und L. E. Rosier. Bounded self-stabilizing Petri nets. In G. Rozenberg, Herausgeber, *Advances in Petri Nets 1993*, *Lecture Notes in Computer Science* 674, Seiten 26–50. Springer Verlag, 1993.

[Com97] Douglas Comer. *Computer Networks and Internets*. Prentice Hall, 1997.

[Des97] J. Desel. How distributed algorithms play the token game. In C. Freksa, M. Jantzen und R. Valk, Herausgeber, *Foundations of Computer Science — Potential, Theory, Cognition*, *Lecture Notes in Computer Science* 1337, Seiten 297–306. Springer-Verlag, 1997.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[Dij86] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1:5–6, 1986.

[DIM93] S. Dolev, A. Israeli und S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[DIM97] S. Dolev, A. Israeli und S. Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM Journal on Computing*, 26:273–290, 1997.

[DK98] J. Desel und E. Kindler. Proving correctness of distributed algorithms using high-level petri nets – a case study. In *Proceedings of the International Conference on Application of Concurrency to System Design*, Seiten 177–186, 1998.

[DKVW95] J. Desel, E. Kindler, T. Vesper und R. Walter. A simplified proof for a self-stabilizing protocol: A game of cards. *Information Processing Letters*, 54:327–328, 1995.

[DNR96] J. Desel, K.-P. Neuendorf und M.-D. Radola. Proving nonreachability by modulo-invariants. *Theoretical Computer Science. Special Volume on Petri Nets*, 153(1–2):49–64, Januar 1996.

[Dol00] Shlomi Dolev. *Self-Stabilization.* MIT-Press, 2000.

[EGL89] Hans-Dieter Ehrig, Martin Gogolla und Udo W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen.* Teubner, 1989.

[EM85] Hartmut Ehrig und Bernd Mahr. *Fundamentals of Algebraic Specifications 1. Equations and Initial Semantics*, *EATCS Monographs on Theoretical Computer Science* 6. Springer-Verlag, 1985.

[FD94] M. Flatebo und A. K. Datta. Two-state self-stabilizing algorithms for token rings. *IEEE Transactions on Software Engineering*, 20:500–504, 1994.

[FDG94] M. Flatebo, A. K. Datta und S. Ghosh. Self-stabilization in distributed systems. In T. L. Casavant und M. Singal, Herausgeber, *Readings in Distributed Computing Systems*, Kapitel 2, Seiten 100–114. IEEE Computer Society Press, 1994.

[Fel68] William Feller. *An Introduction to Probability Theory and its Applications.* Wiley, 1968.

[FR80] N. Francez und M. Rodeh. A distributed abstract data type implemented by a probabilistic communication scheme. In *Proceedings of the 21st Annual IEEE Symposium on the Foundations of Computer Science*, Seiten 373–379, 1980.

[GH91] M. G. Gouda und T. Herman. Adaptive programming. *IEEE Transactions on Software Engineering*, 17:911–921, 1991.

[GH96] M. G. Gouda und F. F. Haddix. The stabilizing token ring in three bits. *Journal of Parallel and Distributed Computing*, 35:43–48, 1996.

[GHR90] M. G. Gouda, R. R. Howell und L. E. Rosier. The instability of self-stabilization. *Acta Informatica*, 27:697–724, 1990.

[GM91] M. G. Gouda und N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.

[Gou85] M. G. Gouda. On "A simple protocol whose proof isn't". *IEEE Transactions on Communications*, 33(4):382–384, 1985.

[Gou87] M. G. Gouda. The stabilizing philosopher: Asymmetry by memory and by action. Technical Report TR-87-12, University of Texas at Austin, 1987.

[GSB94] R. Gupta, S. A. Smolka und S. Bhaskar. On randomization in sequential and distributed algorithms. *ACM Computing Surveys*, 26(1):7–86, März 1994.

[Hai85] B. T. Hailpern. A simple protocol whose proof isn't. *IEEE Transactions on Communications*, 33(4):330–337, 1985.

[Her90] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35:63–67, 1990.

[Her92] T. Herman. Self-stabilization: randomness to reduce space. *Distributed Computing*, 6:95–98, 1992.

[Her96] T. Herman. Self-stabilization bibliography: Access guide. *Chicago Journal of Theoretical Computer Science, Working Paper WP-1*, initiated November 1996. http://www.cs.uchicago.edu/cgi-bin/cjtcs/get/working-papers/1.html.

[IJ90] A. Israeli und M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, Seiten 119–131, 1990.

[IJ93] A. Israeli und M. Jalfon. Uniform self-stabilizing ring orientation. *Information and Computation*, 104:175–196, 1993.

[Jen92] Kurt Jensen. *Coloured Petri Nets*, *EATCS Monographs on Theoretical Computer Science* 1. Springer-Verlag, 1992.

[Kes88] J. L. W. Kessels. An exercise in proving self-stabilization with a variant function. *Information Processing Letters*, 29:39–42, 1988.

[Kin94] E. Kindler. Safety and liveness properties: A survey. *EATCS-Bulletin*, 53:268–272, 1994.

[Kin95] Ekkart Kindler. *Modularer Entwurf verteilter Systeme mit Petrinetzen*, *Edition VERSAL* 1. Bertz Verlag, 1995. Dissertation, Technische Universität München.

[KP93] S. Katz und K. J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993.

[KP99] E. Kindler und S. Peuker. Integrating distributed algorithms into distributed systems. *Fundamenta Informaticae*, 37(3):291–327, 1999.

[KR96] E. Kindler und W. Reisig. Algebraic system nets for modelling distributed algorithms. *Petri Net Newsletter*, 51:16–31, 1996.

[KRVW97] E. Kindler, W. Reisig, H. Völzer und R. Walter. Petri net based verification of distributed algorithms: An example. *Formal Aspects of Computing*, 9:409–424, 1997.

[KV96] E. Kindler und T. Vesper. Automatisch überprüfbare Beweistechniken für algebraische Petrinetze. In *3. Workshop Algorithmen und Werkzeuge für Petrinetze*, Seiten 33–38, 1996.

[KV98a] E. Kindler und T. Vesper. ESTL: A temporal logic for events and states. In *Proceedings of the 19th International Conference on Application and Theory of Petri Nets*, Seiten 365–384, 1998.

[KV98b] E. Kindler und H. Völzer. Flexibility in algebraic nets. In *Proceedings of the 19th International Conference on Application and Theory of Petri Nets*, Seiten 345–364, 1998. To appear in *Theoretical Computer Science*.

[KV99] E. Kindler und T. Vesper. ESTL: Some proof techniques. In W. van der Aalst, J.-M. Colom, F. Kordon, G. Kotsis und D. Moldt, Herausgeber, *Petri Net Approaches for Modelling and Validation*, Kapitel 4, Seiten 53–66. *LINCOM Studies in Computer Science*. LINCOM EUROPA, 1999.

[KW97] E. Kindler und R. Walter. Mutex needs fairness. *Information Processing Letters*, 62:31–39, 1997.

[Lam83] L. Lamport. Solved problems, unsolved problems and non-problems in concurrency, invited address. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, Seiten 63–67, 1983.

[Lam86] L. Lamport. The mutual exclusion problem: part II-statement and solutions. *Journal of the Association of the Computing Machinery*, 33:327–348, 1986.

[LEW96] Jacques Loeckx, Hans-Dieter Ehrich und Markus Wolf. *Specification of Abstract Data Types*. John Wiley & B. G. Teubner, New York, USA, 1996.

[LL90] L. Lamport und N. Lynch. Distributed computing: models and methods. In J. van Leeuwen, Herausgeber, *Handbook of Theoretical Computer Science. Volume B: Formal Models and Semantics*, Kapitel 18, Seiten 1157–1199. Elsevier, 1990.

[LPS81] D. Lehmann, A. Pnueli und J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proceedings of the 8th Annual International Colloquium on Automata, Languages and Programming*, Seiten 264–277, 1981.

[LR81] D. Lehmann und M. O. Rabin. On the advantage of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th Annual ACM Symposium on the Principles of Programming Languages*, Seiten 133–138, 1981.

[LS97] Y. Lakhnech und M. Siegel. Deductive verification of stabilizing systems. In *Proceedings of the 3rd Workshop on Self-Stabilizing Systems*, Seiten 201–216. Carleton University Press, 1997.

[MOOY92] A. Mayer, Y. Ofek, R. Ostrovsky und M. Yung. Self-stabilizing symmetry breaking in constant-space. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, Seiten 667–678, 1992.

[MOY96] A. Mayer, R. Ostrovsky und M. Yung. Self-stabilizing algorithms for synchronous unidirectional rings. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, Seiten 564–573, 1996.

[MP92] Zohar Manna und Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Specification.* Springer-Verlag, 1992.

[Mul89] Nicholas J. Multari. *Towards a Theory for Self-Stabilizing Protocols.* Ph.D. dissertation, University of Texas at Austin, 1989.

[PM96] W. Peng und K. Makki. Petri nets and self-stabilization of communication protocols. *Informatica*, 20:113–123, 1996.

[Pnu83] A. Pnueli. On the extremely fair treatment of probabilistic algorithms. In *Proceedings of the 15th Annual Symposium on the Theory of Computing*, Seiten 278–290, 1983.

[PSL97] A. Pogosyants, R. Segala und N. Lynch. Verification of the randomized consensus algorithm of Aspnes and Herlihy: a case study. In *Proceedings of 11th International Workshop on Distributed Algorithms*, Seiten 111–125, 1997.

[Rab76] M. O. Rabin. Probabilistic algorithms. In *Algorithms and Complexity: New Directions and Recent Results*, Seiten 21–39. Academic Press, 1976.

[Rao90] J. R. Rao. Reasoning about probabilistic algorithms. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, Seiten 247–264, 1990.

[Rao95] Josyula R. Rao. *Extensions of the UNITY methodology: compositionality, fairness, and probability in parallelism*, *Lecture Notes in Computer Science* 908. Springer-Verlag, 1995.

[Rei85] Wolfgang Reisig. *Petri Nets*, *EATCS Monographs on Theoretical Computer Science* 4. Springer-Verlag, 1985.

[Rei91] W. Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80:1–34, 1991.

[Rei98] Wolfgang Reisig. *Elements of Distributed Algorithms. Modeling and Analysis with Petri Nets*. Springer, 1998.

[Rei99] W. Reisig. Die Begriffe der Systemnetze und ihrer Termdarstellung. Unveröffentlichtes Manuskript zum Workshop „Luhme V“, 1999.

[RK97] W. Reisig und E. Kindler. Verification of distributed algorithms with algebraic Petri nets. In C. Freksa, M. Jantzen und R. Valk, Herausgeber, *Foundations of Computer Science — Potential, Theory, Cognition*, *Lecture Notes in Computer Science* 1337, Seiten 261–270. Springer-Verlag, 1997.

[RKV+96] W. Reisig, E. Kindler, T. Vesper, H. Völzer und R. Walter. Distributed algorithms for networks of agents. In W. Reisig und G. Rozenberg, Herausgeber, *Lectures on Petri Nets II: Applications*, *Lecture Notes in Computer Science* 1492, Seiten 331–385. Springer, 1996.

[Sch93] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25:45–67, 1993.

[Sie97] Michael Siegel. *Phased Design and Verification of Stabilizing Systems*. Ph.d. dissertation, Christian-Albrechts-Universität zu Kiel, 1997.

[SR87] E. Smith und W. Reisig. The semantics of a net is a net, an exercise in general net theory. In K. Voss, H. J. Genrich und G. Rozenberg, Herausgeber, *Concurrency and Nets*. Springer-Verlag, 1987.

[SS94] M. Siegel und F. A. Stomp. Extending the limits of sequentially phased reasoning. In *Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science*, Seiten 402–415, 1994.

[Sto93] F. A. Stomp. Structured design of self-stabilizing programs. In *Proceedings of the 2nd Israel Symposium on Theory and Computing Systems*, Seiten 167–176, 1993.

[Ves96] Tobias Vesper. Verifikation eines selbststabilisierenden Leader-Election-Algorithmus unter Ausnutzung verteilter Abläufe. Diplomarbeit, Humboldt-Universität zu Berlin, 1996.

[Ves00] T. Vesper. Randomized self-stabilizing leader-election. Submitted to *Journal of Parallel and Distributed Computing, Special Issue on Self-Stabilizing Distributed Systems*, 2000.

[Völ00] Hagen Völzer. Konspiration und Fairneß in fehlertoleranten Algorithmen. Dissertation. In Vorbereitung, 2000.

[Wal95] Rolf Walter. *Petrinetzmodelle verteilter Algorithmen. Beweistechnik und Intuition*, *Edition VERSAL* 2. Bertz Verlag, 1995. Dissertation, Humboldt-Universität zu Berlin.

[WWV+97] M. Weber, R. Walter, H. Völzer, T. Vesper, W. Reisig, S. Peuker, E. Kindler, J. Freiheit und J. Desel. DAWN: Petrinetzmodelle zur Verifikation verteilter Algorithmen. Informatik-Bericht 88, Humboldt-Universität zu Berlin, Dezember 1997.

[Yun99] Moti Yung, 1999. Personal communication.

[Zuc86] Lenore D. Zuck. *Past Temporal Logic*. Ph.D. dissertation, The Weizmann Institute of Science, 1986.

Index

Lebenslauf

Persönliche Daten

Name:	Tobias Vesper
Geburtsdatum:	2. März 1972 in Hennigsdorf

Schulausbildung

1990	Abitur, Berlin

Wissenschaftlicher Werdegang

1990–1996	Studium der Informatik an der Humboldt-Universität zu Berlin
1993–1994	Gaststudent bei S&W Medico Teknik A/S, Kopenhagen, Dänemark, (7 Monate)
März 1996	Diplom, Humboldt-Universität zu Berlin
seit April 1996	Wissenschaftlicher Mitarbeiter am Lehrstuhl von Prof. Reisig an der Humboldt-Universität zu Berlin, gefördert durch die Deutsche Forschungsgemeinschaft in der Forschergruppe „Petrinetz-Technologie“

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig ohne fremde Hilfe verfaßt zu haben und nur die angegebene Literatur und Hilfsmittel verwendet zu haben.

Tobias Vesper
6. Oktober 2000