Theorie und Praxis der Netzentfaltungen als Grundlage für die Verifikation nebenläufiger Systeme

Stefan Römer

Februar 2000

Lehrstuhl für Informatik VII
der Technischen Universität München

Theorie und Praxis der Netzentfaltungen als Grundlage für die Verifikation nebenläufiger Systeme

Stefan Römer

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ–Prof. Dr. Dr. h.c. Wilfried Brauer
Prüfer der Dissertation:	1. Univ–Prof. Dr. Javier Esparza
	2. Univ–Prof. Dr. Walter Vogler, Universität Augsburg

Die Dissertation wurde am 8.2.2000 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 23.5.2000 angenommen.

Kurzfassung

Bei der Untersuchung von Zustandsräumen nebenläufiger und reaktiver Systeme tritt das bekannte Problem der Zustandsraum-Explosion zutage. Zahlreiche Methoden sind in den letzten Jahren entwickelt und in Werkzeugen implementiert worden, um über geschickte Codierungen des Zustandsraumes dennoch eine Exploration zu ermöglichen, z.B. BDDs, Kronecker-Algebren, das Ausnutzen von Symmetrien oder Netzentfaltungen.
Die (maximale) Entfaltung eines beschränkten Petri–Netzes ist eine halb- geordnete Darstellung, in der in einem einzigen (im Allgemeinen unendlichen) Objekt alle möglichen Abläufe des modellierten Systems repräsentiert werden. Für Fragen wie Erreichbarkeit oder Verklemmung ist bereits ein gewisses endliches Anfangsstück der maximalen Entfaltung ausreichend. Dieses Präfix enthält, in einer kompakten Repräsentation, Informationen über alle erreichbaren Zustände. McMillan hat einen Algorithmus zur Konstruktion eines endlichen Anfangsstücks der Netzentfaltung beschrieben, das groß genug ist, dass darin alle erreichbaren Markierungen enthalten sind.
Die Arbeit beschreibt einige Verbesserungen dieses Algorithmus, die unter anderem garantieren, dass die Größe des generierten Präfixes die Zahl der erreichbaren Markierungen (bis auf eine Konstante) nicht übersteigt. Den Schwerpunkt hierbei bilden Betrachtungen zur effizienten Realisierung des Verfahrens, die eine Untersuchung von Systemen mit Zustandsräumen in der Größenordnung mehrerer Zehnerpotenzen in überschaubarer Zeit gestattet. Die dieser Implementierung zugrundeliegenden Datenstrukturen, Funktionen und Heuristiken werden beschrieben.
In einem weiteren Schwerpunkt wird aufgezeigt, wie mit Hilfe der Entfaltungstechnik Eigenschaften von Programmen, die in einer parallelen Hochsprache spezifiziert sind, untersucht werden können. Syntax und Semantik dieser Sprache, DB$(PN)^2$, werden entwickelt, deren Datentypen einen (potenziell) unendlichen Wertebereich aufweisen dürfen. Die anschließenden Betrachtungen beschreiben einen Algorithmus, der zu einem in dieser Sprache geschriebenen Programm ein endliches Präfix der Entfaltung konstruiert.
Die Präfix–Generierung ist zentrales Modul einer Verifikationsumgebung zur Analyse nebenläufiger Systeme. Das konstruierte Präfix dient beispielsweise als Eingabe verschiedener nachgeschalteter Model–Checker basierend auf unterschiedlichen temporalen Logiken. Das Zusammenspiel dieser einzelnen Komponenten wird erläutert.
Implementierungsaspekte und experimentelle Ergebnisse einer konkreten, effizienten Umsetzung der vorgestellten Algorithmen runden die Arbeit ab.

So eine Arbeit wird eigentlich nie fertig,
man muss sie für fertig erklären,
wenn man nach Zeit und Umständen
das Möglichste getan hat.

J. W. v. GOETHE ÜBER „IPHIGENIE“,
ITALIENISCHE REISE, 16.3.1787

Danksagung

Für die Betreuung dieser Arbeit, zahlreiche Anregungen, Verbesserungsvorschläge und auch kritische Anmerkungen möchte ich mich bedanken bei Prof. Javier Esparza und Prof. Walter Vogler.
Ein besonderer Dank geht an Bram und die aktive VIM–Gemeinde für einen exzellenten Texteditor. Was ein hervorragendes Werkzeug wert ist, weiss man erst zu schätzen, wenn man größere Aufgaben wie die vorliegende Arbeit damit zu bewältigen hat.
Text und Abbildungen sind erstellt worden mit Hilfe des Satzsystems LaTeX 2_ε, unter Zuhilfenahme einiger Zusatzpakete wie dem KOMA–Skript und den PSTricks–Makros; mit letzteren sind sämtliche Illustrationen entstanden, unter Verwendung eigens geschriebener Programme und Makros zur weitgehenden Automatisierung bei der Darstellung von Netzgraphen. Allen beteiligten Autoren gilt mein Dank für ihren Einfallsreichtum, den Blick für das Notwendige und insbesondere die kostenfreie Bereitstellung dieser Werkzeuge.
Nicht zuletzt Dank an meine Familie, Freunde und Kollegen – für Aufmunterung und Zuspruch, Verständnis und manch willkommene Ablenkung während meiner Schaffensphase. Die aufgeschobenen Festivitäten werden alsbald nachgeholt.

München, Februar 2000 Stefan Römer

Hinweis

Die Textgestaltung dieser Arbeit orientiert sich an den Maßgaben der Neuregelung der deutschen Rechtschreibung der Kultusministerkonferenz vom 1.12.1995. An manchen Stellen werden zulässige Nebenformen anstatt der empfohlenen Hauptformen verwendet. In einigen Fällen ist der Autor von den dortigen Vorschlägen jedoch bewusst abgewichen; *Graphen* werden in dieser Arbeit weiterhin nicht geadelt.

Inhaltsverzeichnis

Kapitel 1

„Wenn man recht liest, so entfaltet sich in unserem Innern eine wirkliche, sichtbare Welt nach den Worten.“
NOVALIS, FRAGMENTE

Einführung

Die Informationstechnologie ist in den vergangenen Jahren in zahlreichen Bereichen rasant vorangeschritten: zum einen ist die Hardware der Rechner immer schneller, miniaturisierter und damit komplexer geworden, zum anderen werden aber auch die auf den Rechnern laufenden Anwendungen immer umfangreicher und aufwändiger – nicht zuletzt, weil die Hardware dies zulässt. Mit steigender Komplexität wächst jedoch auch die Wahrscheinlichkeit von Fehlern, da kein Hersteller solcher Komponenten garantieren kann, in der Testphase der Entwicklung wirklich alle möglichen Systemzustände überprüft zu haben[1].

Ein wichtiges Konzept bei der Beschleunigung von Hard– und Software stellt die Verteilung von Ressourcen dar. Einzelne Systemkomponenten erhalten Teilaufgaben, die sie allein oder im Verbund mit anderen Komponenten bearbeiten. Die Ergebnisse dieser lokalen Berechnungen werden untereinander ausgetauscht, indem die beteiligten Systeme, Maschinen oder Prozesse miteinander kommunizieren. Informationen können dabei über Kanäle oder gemeinsame Variablen gewechselt werden.

In der Informatik befasst man sich schon seit längerem mit der (mathematischen) Modellierung verteilter Systeme – inzwischen gibt es eine Vielzahl von Methoden, sie formal zu beschreiben und im Anschluss ihre bestimmte Eigenschaften wie Korrektheit oder Verklemmungsfreiheit zu untersuchen. Häufig wird versucht, klassische Verfahren aus der Analyse sequenzieller Systeme zu übertragen. Gewisse Phänomene verteilter Systeme lassen sich jedoch von sequenziellen Modellen nicht erfassen, andere Probleme ergeben sich durch die Größe, die eine vollständige Verhaltensbeschreibung einnimmt – die Grenzen des Testens sind hier oft schnell erreicht.

[1] Die Software–Hersteller wissen dies sehr gut, deshalb ist in ihren Lizenzverträgen sogar regelmäßig ein Haftungs– und Gewährleistungsausschluß enthalten für den Fall, dass sich ihre Produkte während des Betriebs als fehlerhaft herausstellen sollten, siehe z. B. [Hül94].

Um dem letzteren Problem, der sogenannten Zustandsraum–Explosion, zu begegnen, wird u. a. versucht, geschickte Codierungen des Zustandsraumes nebenläufiger Systeme zu finden. Bekannte Vertreter hierfür sind, neben anderen, *Binary Decision Diagrams* (kurz BDDs) [Bry86] oder Netzentfaltungen [McM92].
Mit Unterstützung des Rechners wird damit eine effiziente Überprüfung von Systemeigenschaften möglich, wenn auch – selbst bei immer höherem Speicherausbau – im Allgemeinen nur für entweder verhältnismäßig kleine Teilkomponenten oder aber stark von der Wirklichkeit abstrahierte Modelle. Mit wachsendem Interesse halten diese Verfahren jedoch auch Einzug in industrielle Fertigungsprozesse, beispielsweise in der Entwicklung von Mikroprozessoren[2].

Die vorliegende Arbeit verwendet als grundlegenden Formalismus zur Beschreibung nebenläufiger Systeme Petri–Netze [Pet62]. Zur Darstellung des dynamischen Systemverhaltens wird eine sogenannte Halbordnungssemantik herangezogen und als Basisobjekt für die Verifikation eingesetzt: die Netzentfaltung, anschaulich gewonnen durch das Entfalten der möglichen Systemabläufe.
Der Vorteil von Netzentfaltungen gegenüber anderen Verhaltensdarstellungen, z. B. Transitionssystemen, ist der hohe Informationsgehalt, der insbesondere bei Verfahren zur Verifikation genutzt werden kann: ein Halbordnungs–basierter Ansatz beschreibt die Semantik der elementaren Aktionen des Systems mit Hilfe einer Kausal–Relation, die darüber Auskunft gibt, ob zwei Aktionen unabhängig voneinander ausführbar sind, oder ob eine Aktion der anderen vorausgehen muss. Daneben bilden Entfaltungen Konflikte und räumliche Verteiltheit unmittelbar ab. Da die Entfaltung eines Netzsystems alle erreichbaren Zustände repräsentiert, können mit ihr alle Erreichbarkeitsprobleme gelöst werden.
Unglücklicherweise sind Netzentfaltungen im Allgemeinen unendlich große Objekte. McMillan hat in [McM92] einen Algorithmus zur Konstruktion eines endlichen Anfangsstücks der Netzentfaltung beschrieben, das bereits alle erreichbaren Markierungen enthält, jedoch in vielen Fällen erheblich kleiner ist als der in ihm implizit codierte Zustandsraum. Damit eignet sich dieses Anfangsstück, vollständiges Präfix genannt, sehr gut für die Verifikation.
Gegenüber symbolischen Model–Checking–Methoden wie BDD–basierten [BCM$^+$92, Hu95] haben Präfixe der Netzentfaltung den Vorteil, dass keinerlei zusätzliche Informationen für eine effiziente Konstruktion notwendig sind wie etwa Variablenordnungen[3] oder Heuristiken. Mit Hilfe ein und desselben Präfixes können alle untersuchba-

[2] Die Bestrebungen in diese Richtung dürften nicht zuletzt durch einen – von der Firma Intel selbst entdeckten – Fehler in der FPU der ersten Pentium–Prozessoren vorangetrieben worden sein[Int94]. Dieser Fehler löste 1994/95 die bislang größte (und teuerste) Umtauschaktion der PC–Geschichte aus [Hüs95].

[3] Für die Berechnung von Präfixen muss zwar auch eine Ordnung (auf den Transitionen) festgelegt werden, diese hat jedoch – verglichen mit der Variablenordnung bei BDDs – nur einen geringen Einfluss auf die Präfixgröße.

ren Eigenschaften überprüft werden; inzwischen gibt es Model–Checker für verschiedene temporale Logiken [Esp93, Wal98], die auf die Netzentfaltung zurückgreifen. Erste experimentelle Untersuchungen haben aufgezeigt, dass die Entfaltungstechnik ihre Vorteile bei stark nebenläufigen Systemen ausspielen kann, während BDDs bei stärker sequenziellen Systemen vorzuziehen sind. Über reine Stellen / Transitions–Netze hinaus ist die Entfaltungstechnik inzwischen auch auf andere Modelle übertragen worden, darunter Netze mit Lesekanten [VSY98], zeitbehaftete Netze [BF99], sowie kommunizierende Transitionssysteme [ER99] und eine CSP–verwandte Prozess–Algebra [LB99]. Einen Überblick bezüglich Veröffentlichungen im Zusammenhang mit Netzentfaltungen und ihren Anwendungen bietet eine umfassende Bibliographie [Unf99].

Die Arbeit beschreibt einige Verbesserungen des Algorithmus zur Präfix–Generierung, die unter anderem garantieren, dass die Größe des generierten Präfixes die Zahl der erreichbaren Zustände (bis auf eine Konstante) nicht übersteigt. Den Schwerpunkt hierbei bildet die Beschreibung einer effizienten Implementierung des Verfahrens, die eine Untersuchung von Systemen mit Zustandsräumen in der Größenordnung mehrerer Zehnerpotenzen in überschaubarer Zeit gestattet. Diese konkrete Umsetzung ist inzwischen Teil mehrerer Werkzeuge zur Modellierung, Analyse und automatischen Verifikation nebenläufiger Systeme [PEP98, CPN99, SSE99]. In zahlreichen Veröffentlichungen sind mit dieser Implementation Benchmarks und Vergleichsstudien angestellt worden, darunter [Röm95, ERV96, Bes96, HD96, MR97, Hel98, ERV00, Mel98, HDS99, ER99, Hel99b].

Im PEP–Tool [PEP98, MRE96], einer Petri–Netz–basierten Entwicklungs– und Programmierumgebung, ist der Präfix–Generator Teil des zentralen Model–Checkers [Esp93, Esp94, Gra95, Gra97]. Dieses Werkzeug unterstützt unterschiedliche Eingabesprachen zur Modellierung nebenläufiger Systeme: zur strukturierten, übersichtlichen Eingabe steht eine Hochsprachen–ähnliche Programm–Notation, B(PN)^2[BH93], zur Verfügung; darüber hinaus wird ein Fragment von SDL [SDL92] erkannt, einer recht weit verbreiteten Modellierungssprache. Auf Ebene der Petri–Netze werden High–Level(HL)–Netze in einer speziellen Ausprägung zur Modellierung angeboten: die sogenannten M–Netze [BFF$^+$95]. Daneben können Systeme in einer einfachen Automatendarstellung beschrieben werden, für deren Beschriftungen die gleiche Syntax verwendet wird wie für M–Netze. Ebenfalls möglich ist die direkte Spezifikation von Systemen als Stellen / Transitions(S / T)–Netze. Als weitere, in der Praxis jedoch seltener genutzte Möglichkeit steht die Modellierung als Terme einer CCS[Mil80]–ähnlichen Prozess–Algebra [BDH92] zur Verfügung. Für die Netz– und Automatendarstellungen sind in PEP spezielle graphische Editoren enthalten; die Programmnotationen werden mit Hilfe von Texteditoren eingegeben,
Die beschriebenen Eingabesprachen sind in einer Hierarchie angeordnet: die in das Werkzeug integrierten Verifikationstechniken arbeiten auf 1–sicheren S / T–Netzsyste-

men, somit müssen alle Darstellungen in diese Netze übersetzt werden; dabei kann es notwendig sein, stufenweise Übersetzungen in mehrere der Eingabeformate vorzunehmen. Beispielsweise wird ein in B(PN)^2 spezifiziertes Programm zunächst in eine entsprechende HL–Netz–Darstellung umgewandelt, dieses Netz wird anschließend in ein S/T–Netz aufgefaltet[4]. Alternativ kann das B(PN)^2–Programm in einen Term der Prozess–Algebra transformiert werden und von dort in ein S/T–Netz. Dieses Netz dient anschließend als Eingabe des Präfix–Generators, der den Zustandsraum codiert für die nachfolgende Verifikation.

In [LG93, Röm93] wird ein Ansatz vorgestellt, der zu einem prozess–algebraischen Term direkt kompositionell dessen halbgeordnete Verhaltensbeschreibung in Form eines Prozess–Netzes erzeugt. Auf diese Weise kann ein Übersetzungsvorgang eingespart werden. Für eine praktische Umsetzung bringt dieses kompositionelle Verfahren jedoch einen entscheidenden Nachteil: es wird stets die maximale Entfaltung eines Systems erzeugt – bzw. der Teil davon, den der Speicherausbau zulässt. In Anbetracht der allgemeinen Unendlichkeit von Entfaltungen ist der kompositionelle Ansatz in dieser Form somit nicht praktikabel. Die Kriterien, die bei der Präfix–Generierung nach McMillan verantwortlich sind für die Terminierung bei Erreichen eines vollständigen, d. h. alle Systemzustände repräsentierenden Anfangsstücks der Entfaltung, lassen sich auf die kompositionelle Konstruktion nicht ohne weiteres übertragen.
Deshalb wird in der vorliegenden Arbeit ein etwas anderer Ansatz verfolgt, eine halbgeordnete Verhaltensbeschreibung von Systemen auf direkterem Wege zu gewinnen. Der Kontrollfluss–Anteil eines gegebenen B(PN)^2–ähnlichen Programms wird zunächst – über die Prozess–Algebra – übersetzt in ein 1–sicheres Netz. Die prozess–algebraische Darstellung wird hierbei nicht explizit gespeichert, sie ist über die Kompositionalität direkt in den Parser der Eingabesprache integriert. Zu diesem Netz wird anschließend, zusammen mit den Informationen über die Programmvariablen und ihre Datentypen, mit einer Version des Algorithmus von McMillan ein endliches, vollständiges Präfix der Netzentfaltung generiert.
Da die tatsächlich vorkommenden Variablenwerte somit erst während der Konstruktion des Präfixes ermittelt werden, ist es sogar möglich, potenziell unendliche Datenbereiche anzugeben. Die bislang in [PEP98] verwendeten Verfahren ließen dies nicht zu, da in den Variablendarstellungen alle theoretisch möglichen Werteänderungen vorgesehen werden mussten. In HL–Netzen stellt dies durch Verwendung symbolischer Marken kein Problem dar, spätestens bei der Entfaltung eines HL–Netzes in ein S/T–Netz wird jedoch jede Werteänderung einzeln repräsentiert. Große Wertebereiche ziehen damit einen erheblichen Speicherverbrauch nach sich, der die Verifikation in einzelnen Fällen stark einschränken oder sogar praktisch unmöglich machen kann [5].

[4] Die *Auffaltung* eines HL–Netzes in ein S/T–Netz ist zu unterscheiden von der *Entfaltung* eines S/T–Netzes in ein Prozess–Netz.

[5] Über konkrete Schwierigkeiten bei der Umsetzung eines Algorithmus in B(PN)^2 aufgrund der nur einge-

Im Rahmen dieser Entwicklungen sind auch einige Änderungen an der Programm–Notation B(PN)^2 vorgenommen worden: Variablen können nur global deklariert werden, außerdem ist der Parallelоperator nur auf oberster Ebene zugelassen. Die Sprache ist damit vornehmlich geeignet zur Modellierung kommunizierender sequenzieller Systeme, damit ist ein breites Spektrum von Anwendungen abgedeckt. Als weitere Änderungen sind zusätzliche Konstrukte wie Sprünge und ein **if**–Kommando aufgenommen worden [RM96]; darüber hinaus ist das Angebot der unterstützten Datentypen erweitert worden um Strukturen wie Felder und Tupel. Insgesamt ist es mit diesen Änderungen möglich, minimale Netze zu einem Programm zu erzeugen, für die anschließende Verifikation ist dieser Umstand besonders vorteilhaft. Die geänderte Programm–Notation wird im in dieser Arbeit DB(PN)^2 genannt, *Distributed B(PN)*^{2}.

Die Arbeit stellt die Integration ausgesuchter Module zu einer DB(PN)^2–basierten Verifikationsumgebung vor. Mit ihr können nebenläufige Systeme mit geringem Aufwand modelliert und anschließend untersucht werden. Auf Basis des vollständigen Präfixes der Netzentfaltung sind mittlerweile eine Reihe effizienter Verifikationsverfahren entwickelt und implementiert worden, darunter Algorithmen zur Erkennung von Verklemmungen [McM95, MR97, Mel98, Hel99b] und zur Untersuchung von Erreichbarkeitsfragen [Hel99b, SE00]. Daneben stehen Model–Checker zur Verfügung, welche die Spezifikation von beliebigen Eigenschaften erlauben, die sich mit Hilfe von temporal–logischen Formeln ausdrücken lassen. Unterstützt werden sowohl Branching-Time–Logiken [Esp94, Gra97], als auch Linear–Time–Logiken [Wal98, Wal00, HNW98].

Abbildung 1.1 gibt einen graphischen Überblick über den Zusammenhang zwischen den oben beschriebenen Modellen und Darstellungen (Eingabesprache, Algebren, Netze), die in dieser Arbeit betrachtet werden: der Kontrollfluss–Anteil eines gegebenen DB(PN)^2–Programms wird übersetzt in einen prozess–algebraischen Term; dieser Term wird automatisch übersetzt in ein entsprechendes 1–sicheres Petri–Netz. Die Angaben in den linken Hälften der Blöcke bezeichnen Instanzen eines durchgehenden, einfachen Beispiels: zwei verschiedene nebenläufige Zuweisungen an dieselbe ganzzahlige Variable *i*. Bei der Übersetzung des Programms werden die deklarierten (globalen) Variablen zusammen mit ihren Wertebereichen extrahiert und gesondert verwaltet. Beide Informationen, Netz und Variablen, dienen als Eingabe für die Berechnung des Präfixes der Entfaltung, die das Verhalten des durch das Programm modellierten Systems beschreibt. Die Beschriftungen an den Stellenknoten geben hier die jeweils aktuellen Variablenwerte an; die Transitionen sind mit den elementaren Programmaktionen versehen. Mit diesem Präfix (und zusätzlichen, hier nicht näher ausgeführten Informationen aus dem ursprünglichen Modell, wie Bezeichnernamen oder Kontrollpunkte im Programm) kann anschließend die Analyse des Programms mittels Verifikation erfolgen.

schränkt vorhandenen skalaren Datentypen berichtet [Mer96].

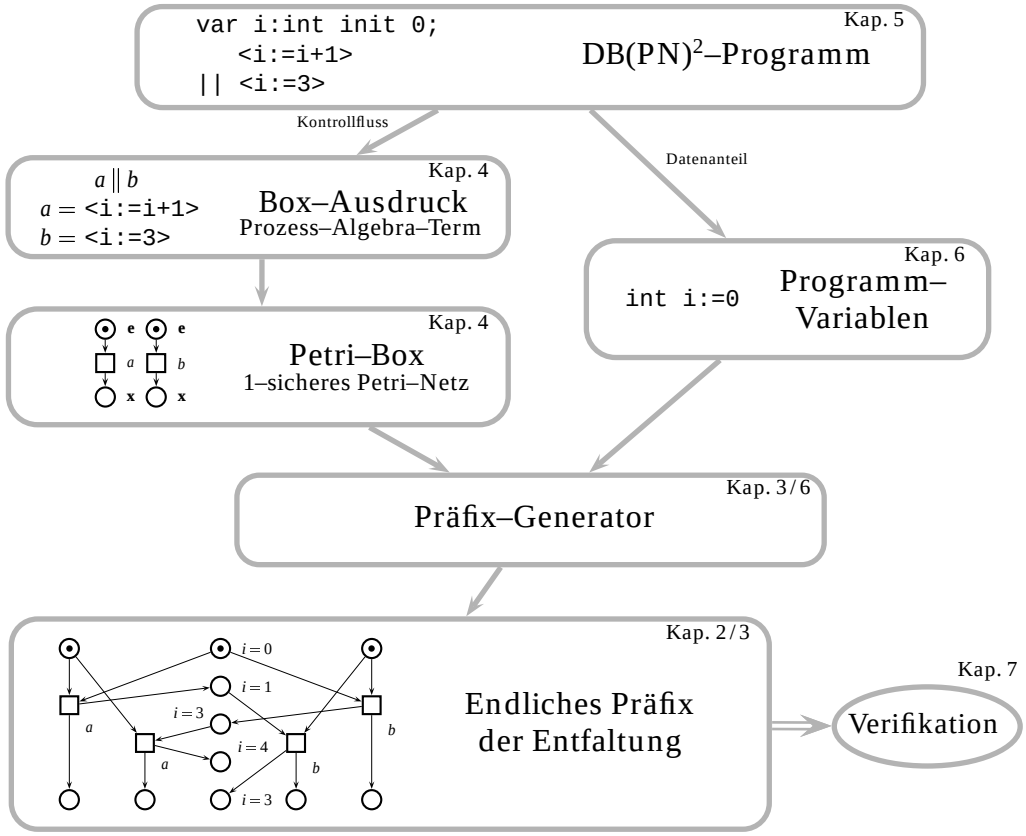


Abbildung 1.1: Übersicht.

Die Kapitelangaben in den Blöcken von Abb. 1.1 verweisen auf die Stellen im theoretischen Teil dieser Arbeit, an denen die entsprechenden Themen erörtert werden. Interessante Gesichtspunkte der prototypischen Umsetzung werden in den praktischer ausgelegten Kapiteln 8, 9 und 10 diskutiert, die nicht in diese Graphik aufgenommen wurden. Der Schwerpunkt der Arbeit liegt auf der Darstellung der praktischen Umsetzung der Verfahren; für die theoretischen Ausführungen notwendige Beweise werden deshalb nur in solchen Fällen angegeben, dass diese abweichende Notationen bzw. Beweisführungen aufweisen, oder ein Beweis zentrale Bedeutung besitzt. Andernfalls sind Quellen für entsprechende Beweise zitiert.

Im Einzelnen ist die Arbeit wie folgt gegliedert: Kapitel 2 fasst die wichtigsten mathematischen und netztheoretischen Notationen und Sachverhalte zusammen, die für die kommenden Ausführungen benötigt werden. Verschiedene Formen der Verhaltensbeschreibung sowie der Prozess–Begriff auf Netzen werden hier definiert.
Die Präfix–Generierung ist Gegenstand von Kapitel 3: der Algorithmus von McMillan

wird Schritt für Schritt entwickelt, daran anschließend werden mögliche Verbesserungen des Verfahrens beschrieben.
Das Kapitel 4 ist dem Box–Kalkül gewidmet, also Box–Ausdrücken und ihrer Netz–Semantik, den Petri–Boxen. Die Programm–Notation DB(PN)^2 wird in Kapitel 5 vorgestellt. Neben einer Beschreibung ihrer Syntax und Semantik wird sie verglichen mit dem ihr verwandten B(PN)^2.
Kapitel 6 führt die Ergebnisse aus den Kapiteln 3 bis 5 zusammen und beschreibt die Konstruktion endlicher Präfixe für DB(PN)^2–Programme.
In Kapitel 7 werden Verifikationstechniken im Überblick zusammengestellt. Insbesondere werden dort existierende Verfahren basierend auf dem endlichen Präfix diskutiert.
Implementationsaspekte der in den vorangegangenen Kapiteln präsentierten Verfahren sind in Kap. 8 zu finden. Neben (allgemeinen) Datenstrukturen und Basisoperationen für Petri–Netze werden dort hauptsächlich Einzelheiten der Umsetzung des Präfixgenerators und des Tests auf Verklemmungsfreiheit sowie des DB(PN)^2–Übersetzers beschrieben. Die vorgestellten Techniken sind keineswegs nur auf die hier konkret beschriebenen Probleme anwendbar, vielmehr können sie als Anregung bei der effizienten Umsetzung beliebiger Petri–Netz–basierter Algorithmen angesehen werden. Diverse Komplexitätsbetrachtungen runden dieses Kapitel ab.
In Kapitel 9 wird eine kleine Verifikationsumgebung vorgestellt, die die Analyse nebenläufiger Systeme mit halbordnungsbasierten Verfahren gestattet. Programmbeispiele und experimentelle Ergebnisse der prototypischen Implementierung der Algorithmen sind in Kapitel 10 zusammengestellt.
Kapitel 11 fasst die Arbeit zusammen und liefert einen kleinen Ausblick auf mögliche Erweiterungen. Ein Anhang enthält eine Aufstellung der vollständigen Syntax der Eingabesprachen sowie die Aufrufparameter der beschriebenen Programme. Daneben sind dort das Literaturverzeichnis und ein Index der verwendeten Symbole und wichtigsten Begriffe zu finden.

Kapitel 2

„Felix, qui potuit rerum cognoscere causas.“
(Glücklich, wer den Grund der Dinge
zu erkennen vermocht hätte.)
VERGIL, GEORGICA, 2, 490

Grundlagen

Dieses Kapitel führt einige mathematische und netztheoretische Notationen und Begriffe ein, die in dieser Arbeit benutzt werden. Zunächst stellt Abschnitt 2.1 die verwendeten Schreibweisen für Menge, Relationen und Ordnungen zusammen.

Das für diese Arbeit zentrale Konzept ist das der Petri–Netze, ursprünglich begründet von Carl Adam Petri in [Pet62]. Petri–Netze besitzen eine wohldefinierte Struktur, die in verschiedener Weise interpretiert werden kann; mit ihnen lassen sich auf einfache und anschauliche Weise sowohl der strukturelle (statische) Anteil nebenläufiger Systeme modellieren, als auch deren dynamisches (Ablauf–)Verhalten. Das Modell der Petri–Netze wird, zusammen mit einigen statischen Eigenschaften, einführend in Abschnitt 2.2 beschrieben. Für ein vertiefendes Studium der Theorie der Petri–Netze sei auf Textbücher wie [Rei85, Bau90] verwiesen.

Abschnitt 2.3 erweitert Petri–Netze um eine bewegliche Markierung. Ähnlich endlichen Automaten, mit denen Petri–Netze sehr nah verwandt sind, wird in diesem Zusammenhang von (erreichbaren) Zuständen gesprochen. Wichtige Begriffe, die auf markierten Netze definiert werden, sind Lebendigkeit oder Verklemmung.

Das dynamische Verhalten nebenläufiger Systeme kann auf verschiedene Weise beschrieben werden. Abschnitt 2.4 stellt zwei dieser Methoden gegenüber: Interleaving– und Halbordnungs–Semantiken. Letztere werden eingehender untersucht, insbesondere wird der Begriff *Prozess* eines Netzes definiert. Für den Rest der Arbeit wird eine halbgeordnete Prozess–Semantik zur Verhaltensbeschreibung verwendet.

2.1 Mathematische Notationen

Logik Die folgenden Symbole werden in ihrer vertrauten Bedeutung verwendet: $\wedge$ (und), $\vee$ (oder), $\neg$ (nicht), $\Rightarrow$ (Implikation), $\Leftrightarrow$ (Äquivalenz), $=$ (Gleichheit), $\neq$ (Ungleichheit), $\exists x : P(x)$ (es existiert ein x, das das Prädikat $P(x)$ erfüllt), $\forall x : P(x)$ (für alle x gilt $P(x)$). Es gelten die üblichen Prioritäten der Operatoren (Aufzählung in absteigender Bindungsstärke): $\neg$; $=, \neq$; $\wedge$; $\vee$; $\Rightarrow, \Leftrightarrow$; $\forall, \exists$.

Mengen Mit $\mathbb{N} = \{0, 1, 2, \dots\}$ wird die Menge der natürlichen Zahlen bezeichnet, $\mathbb{N}^+ = \{1, 2, 3, \dots\}$ steht für die Menge natürlicher Zahlen ohne die 0. Die Menge der ganzen Zahlen $\{0, 1, -1, 2, -2, \dots\}$ wird $\mathbb{Z}$ geschrieben. Die leere Menge wird $\emptyset$ abgekürzt. Für eine Menge X bezeichnet $x \in X$ ($x \notin X$) das *(Nicht–)Enthaltensein* des Elements x in X. Mit $|X|$ wird die *Kardinalität* von X bezeichnet, für endliche Mengen entspricht sie der Anzahl der Elemente in X. Mengen werden durch die Notation $\{x \mid P(x)\}$ charakterisiert. Für zwei Mengen X, Y bezeichnet $X \subseteq Y$ ($X \subset Y$) die (echte) *Teilmenge* von X in Y. Die *Potenzmenge* von Y, $2^Y = \{X \mid X \subseteq Y\}$, fasst alle Teilmengen von Y zusammen. Binäre Operationen auf Mengen sind *Vereinigung* ($X \cup Y$), *Durchschnitt* ($X \cap Y$) und *Differenz* ($X \setminus Y$), übertragen auf eine Familie $(X_i)_{i \in I}$ von Mengen werden Vereinigung bzw. Durchschnitt geschrieben als $\bigcup_{i \in I} X_i$ bzw. $\bigcap_{i \in I} X_i$. $X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$ bezeichnet das *Produkt* von X und Y. Zwei Mengen X und Y heißen *disjunkt*, falls $X \cap Y = \emptyset$ gilt.

Abbildungen Eine *Abbildung* f von X nach Y wird $f : X \to Y$ geschrieben. Für $x \in X$ bezeichnet $f(x)$ das *Bild* von x. Die *Umkehrung* von f wird f^{-1} geschrieben; f^{-1} ist im Allgemeinen keine Abbildung. Für $y \in Y$ definiere $f^{-1} = \{x \mid f(x) = y\}$; ist f^{-1} eine Abbildung, so wird sie *Umkehrabbildung* von f genannt. Falls nicht anders vermerkt, wird für eine Menge $X' \subseteq X$ die Erweiterung von f definiert als $f(X') = \bigcup_{x \in X'} f(x)$, für $Y' \subseteq Y$ bzw. die Erweiterung von f^{-1} als $f^{-1}(Y') = \bigcup_{y \in Y'} f^{-1}(y)$. Für $X' \subseteq X$ bezeichnet $f|_{X'} : X' \to Y$ die *Restriktion* von f auf X'. Eine Abbildung $f : X \to Y$ heißt *injektiv*, falls sie $\forall x, y \in X : f(x) = f(y) \Rightarrow x = y$ erfüllt. Gilt $f(X) = Y$, so heißt f *surjektiv*. Ist f injektiv und surjektiv, dann wird f auch *bijektiv* genannt.

Multimengen Eine *Multimenge* über X ist eine Abbildung $\mu : X \to \mathbb{N}$; μ heißt *endlich*, wenn die Menge $\{n \in \mathbb{N} \mid \mu(n) \neq 0\}$ endlich ist. Die Menge der (endlichen) Multimengen über X wird mit $\mathcal{M}(X)$ ($\mathcal{M}_\mathcal{F}(X)$) gekennzeichnet. Jede Menge $Y \subseteq X$ kann als spezielle Multimenge $Y : X \to \{0, 1\}$ interpretiert werden, umgekehrt kann jede Multimenge, für die $\forall x \in X : \mu(x) \leq 1$ gilt, als Menge aufgefasst werden. Seien $\mu_1, \mu_2 \mathcal{M}$ zwei Multimengen und $x \in X$, dann werden die gewöhnlichen mengentheoretischen Operationen wie folgt übertragen: $(\mu_1 \cup \mu_2)(x) = \max(\mu_1(x), \mu_2(x))$ (Vereinigung),

$(\mu_1 \cap \mu_2)(x) = \min(\mu_1(x), \mu_2(x))$ (Durchschnitt), $(\mu_1 \setminus \mu_2)(x) = \max(\mu_1(x) - \mu_2(x), 0)$ (Differenz), $(\mu_1 + \mu_2)(x) = \mu_1(x) + \mu_2(x)$ (Summe).

Sprachen Sei A ein endliches *Alphabet* von Symbolen. Eine Sequenz $w = a_1 a_2 a_3 ...$ mit $a_i \in A, i \in I$ heißt *Wort* über A; $|w|$ gibt die *Länge*von w an; $\epsilon \notin A$ kennzeichnet das leere Wort. Für $n \in \mathbb{N}$ bezeichnet $A^n = \{a_1 a_2 \dots a_n \mid a_i \in A, 1 \leq i \leq n\}$ die Menge aller Wörter der Länge n über A. Die Menge A^* aller Wörter beliebiger (endlicher) Länge über A inklusive ϵ wird *Sprache* genannt. Die Sprache $A^+ = A^* \setminus \{\epsilon\}$ ist die Menge der nicht–leeren Wörter über A. Für zwei Wörter $w_1, w_2 \in A^*$ bezeichnet $w_1 w_2$ (gelegentlich auch $w_1 \cdot w_2$ geschrieben) die *Konkatenation* von w_1 und w_2; w_1 heißt dann *Präfix*und w_2 *Suffix* von $w_1 \cdot w_2$ Die Anzahl der Vorkommen eines Symbols a in einem Wort w wird $\#_a(w)$ geschrieben.

Relationen Für eine Menge $R \subseteq X \times X$ wird das Paar (X, R) *Relation* über X genannt. Wenn X unmittelbar aus dem Kontext hervorgeht, wird statt (X, R) manchmal nur R geschrieben. Die Schreibweisen $(x, y) \in R$ und xRy sind gleichwertig. Für eine Relation (X, R) und eine Menge $X' \subseteq X$ ist die *Restriktion* von R auf X' definiert als $R|_{X'} = \{(x, y) \mid x, y \in X' \wedge xRy\}$.
Eine Relation (X, R) heißt

- *reflexiv*, wenn $\forall x \in X : xRx$;
- *irreflexiv*, wenn $\forall x \in X : \neg(xRx)$;
- *symmetrisch*, wenn $\forall x, y \in X : xRy \Rightarrow yRx$;
- *antisymmetrisch*, wenn $\forall x, y \in X : xRy \wedge yRx \Rightarrow x = y$;
- *transitiv*, wenn $\forall x, y, z \in X : xRy \wedge yRz \Rightarrow xRz$;
- *linear*, wenn $\forall x, y \in X : xRy \vee yRx \vee x = y$.

Die *transitive* Hülle R^+ einer Relation R ist die kleinste transitive Relation, die R umfasst. Die *reflexive transitive Hülle* wird mit R^* bezeichnet. Eine Relation R heißt *Äquivalenzrelation*, wenn R reflexiv, symmetrisch und transitiv ist. Eine Abbildung $h : (X, R) \to (Y, S)$ heißt *Homomorphismus*,falls für alle $x_1, x_2 \in X$ gilt: $h(x_1 R x_2) = h(x_1) S h(x_2)$; h heißt *Isomorphismus*,falls h zusätzlich bijektiv ist.

Ordnungen *Geordnete Mengen* spielen in dieser Arbeit eine wichtige Rolle. Einerseits können die Abläufe von verteilten Systemen als spezielle Ordnungaufgefasst werden, andererseits wird auf der Menge der Abläufe eine Präfixrelation definiert – ebenfalls eine Ordnung. Neben diesen beiden Anwendungen treten Ordnungen an einigen weiteren Stellen auf.
Sei X eine Menge und (X, R) eine Relation über X. (X, R) heißt

- *(reflexive) Halbordnung* auf X, wenn R reflexiv, antisymmetrisch und transitiv ist;
- *(strikte oder echte) Halbordnung* auf X, wenn R irreflexiv und transitiv ist;

- (reflexive, strikte) *Totalordnung*, wenn (X, R) eine (reflexive, strikte) Halbordnung ist und R zudem linear.

Zur Kennzeichnung von Ordnungsrelationen werden spezielle Symbole verwendet, die bereits syntaktisch die Ordnung ausdrücken sollen: $<$ $(\leq)$ bzw. $>$ $(\geq)$; $\prec$ $(\preceq)$; $\sqsubset$ $(\sqsubseteq)$ sind Beispiele für zusammengehörende strikte (reflexive) Ordnungen.
Sei $(X, \prec)$ eine Ordnung. Für ein Element $x \in X$ definiert $\downarrow x = \{y \in X \mid y \prec x\}$ die Vorgängermenge und $\uparrow x = \{y \in X \mid x \prec y\}$ die Nachfolgermenge von x; die Erweiterung auf eine Menge $Y \subseteq X$ ist definiert durch $\downarrow Y = \bigcup_{y \in Y} \downarrow y$ bzw. $\uparrow Y = \bigcup_{y \in Y} \uparrow y$. Für $Y, Z \subseteq X$ definiert $[Y, Z] = (\uparrow Y) \cap (\downarrow Z)$ das Intervall zwischen Y und Z. *Minimum* bzw. *Maximum* einer Menge $Y \subseteq X$ sind definiert als $\mathrm{Min}(Y) = \{y \in Y \mid \nexists x \in Y : x \prec y\}$ bzw. $\mathrm{Max}(Y) = \{y \in Y \mid \nexists x \in Y : y \prec x\}$.
Die Relation li auf X ist definiert als $x \,\mathrm{li}\, y$, wenn $x \leq y \vee y \leq x$. $Y \subseteq X$ heißt li–Menge bezüglich $\prec$, wenn $\forall x, y \in Y : x \,\mathrm{li}\, y$. Y heißt *Linie*, wenn Y maximale li–Menge ist. Eine Ordnung $(X, \prec)$ heißt *fundiert*, wenn gilt: $\forall x \in X \forall$ Linien $L : L \cap \downarrow x \in \mathbb{N}$.

Graphen Ein gerichteter *Graph* ist ein Tupel $G = (V, E)$ bestehend aus einer Menge V von *Knoten* und einer Menge $E \subseteq V \times V$ von geordneten Knotenpaaren, den *Kanten*. In Darstellungen werden die Knoten als Kreise, die Kanten als Pfeile gezeichnet. G heißt *bipartit*, falls V aus zwei disjunkten Knotenmengen besteht: $V = V_1 \cup V_2$. Ein *Pfad* in einem Graph ist eine Folge von Knoten $v_1 v_2 \ldots v_n, n \geq 1$, so dass es eine Kante $(v_i, v_{i+1}) \in E$ für alle $1 \leq i \leq n$ gibt; die *Länge* dieses Pfades ist $n - 1$. Gilt $v_1 = v_n$, so ist der Pfad ein *Zyklus*. Ein Graph heißt *azyklisch*, falls er keinen Zyklus enthält. Ein Graph $G_v = ((V, E), S)$ heißt *verwurzelt*, falls (V, E) ein Graph und $S \in V$, die *Wurzel*, ein ausgezeichneter Startknoten ist, an dem der Graph aufgehängt ist.

2.2 Petri–Netze

Das Konzept der Petri–Netze ist eingeführt worden in [Pet62]; die hier verwendeten Notationen lehnen sich weitgehend an die in [BF87] eingeführten Schreibweisen an. In seiner allgemeinen, einfachsten Ausprägung, den Stellen/Transitions–Netzen, ist der Netzbegriff wie folgt definiert.

Definition 2.1 *Petri–Netz*

Ein *Petri–Netz* (kurz *Netz*) ist ein bipartiter Graph $\mathcal{N} = (S, T, W)$ mit disjunkten Knotenmengen von *Stellen* S und *Transitionen* T, sowie einer *Gewichtsfunktion* $W : ((S \times T) \cup (T \times S)) \to \mathbb{N}$, die Stellen und Transitionen durch gewichtete, gerichtete Kanten abwechselnd miteinander verbindet.

Ein Netz $\mathcal{N}$ heißt endlich, wenn $|S \cup T| \in \mathbb{N}$ ist. ■2.1

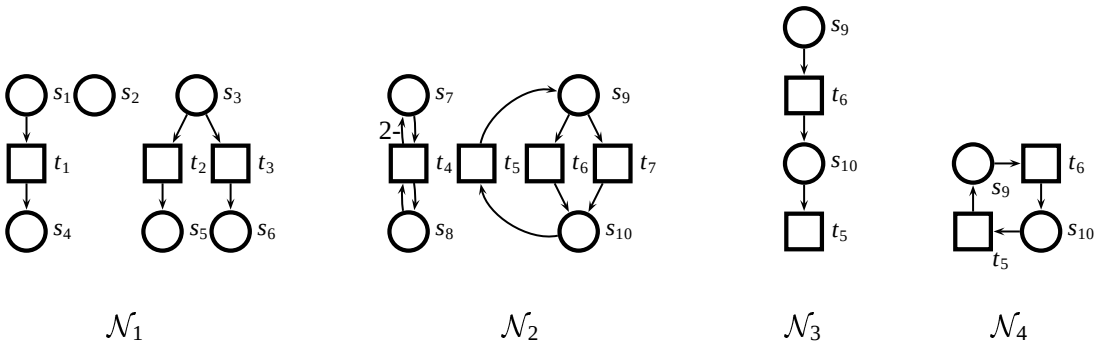


Abbildung 2.1: Vier Petri–Netze.

Petri–Netze dienen beispielsweise zur Beschreibung des Informationsflusses in einem nebenläufigen System. Sie bilden einen gerichteten Graphen, der in den Darstellungen durch Kreise für die Stellenknoten und Quadrate[1] für die Transitionsknoten repräsentiert wird. Die gerichteten Kanten führen jeweils von einem Element der einen zu einem Element der anderen Knotenmenge; das Kantengewicht wird, falls es größer als 1 ist, als Zahlenwert an die jeweilige Kante geschrieben; ein Kantengewicht von 0 wird durch Weglassen der entsprechenden Kante angezeigt. Abbildung 2.1 zeigt beispielhaft vier Netze. Die Knoten der Netze sind zur Identifizierung mit Namen versehen, gleiche Namen bezeichnen hierbei gleiche Elemente.

Aus formalen Gründen wird im Folgenden die Gewichtsfunktion bei Netzen, die nur einfache Kantengewichte aufweisen, d. h. es gilt

$$W : \big((S \times T) \cup (T \times S) \big) \to \{ 0, 1 \} ,$$

als Flussrelation F geschrieben:

$$F \subseteq \big((S \times T) \cup (T \times S) \big) .$$

Diese beiden Darstellungen stehen durch die Äquivalenz

$$W(x, y) = 1 \;\Leftrightarrow\; (x, y) \in F$$

in Beziehung.

Definition 2.2 *Vor–, Nachbereich*

Sei $\mathcal{N} = (S, T, W)$ ein Netz und sei $x \in S \cup T$ ein Knoten von $\mathcal{N}$.

Die Menge ${}^\bullet x = \{ y \in S \cup T \mid W(y, x) > 0 \}$ heißt *Vorbereich* (engl. *preset*),
die Menge $x^\bullet = \{ y \in S \cup T \mid W(x, y) > 0 \}$ *Nachbereich* (*postset*) von x.

Die Erweiterung auf Mengen $X \subseteq S \cup T$ ist gegeben durch:

$${}^\bullet X = \bigcup_{x \in X} {}^\bullet x \quad \text{bzw.} \quad X^\bullet = \bigcup_{x \in X} x^\bullet .$$

■2.2

[1] Manche Autoren verwenden auch (ausgefüllte) Rechtecke.

Für das Netz $\mathcal{N}_1$ aus Abbildung 2.1 gilt beispielsweise $s_3^\bullet = \{t_2, t_3\}$, ${}^\bullet s_3 = \emptyset$ und ${}^\bullet\{t_1, t_2\} = \{s_1, s_3\}$.
Die Stellenmengen der in dieser Arbeit betrachteten Petri–Netze seien stets nicht–leer ($S \neq \emptyset$), außerdem gelte für Transitionen die sogenannte *T–Eingeschränktheit*

$$\forall t \in T : {}^\bullet t \neq \emptyset \neq t^\bullet ,$$

d. h. Transitionen besitzen stets ein– und ausgehende Kanten; zudem seien die Vor– und Nachbereiche von Transitionen stets endlich. Netz N_3 von oben erfüllt diese Bedingung beispielsweise nicht.
Es werden nun einige weitere strukturelle Eigenschaften von Netzen eingeführt.

Definition 2.3 *Schlinge*

Sei (S, T, W) ein Netz. Zwei Knoten $s \in S$, $t \in T$ sind durch eine *Schlinge* miteinander verbunden, falls $s \in t^\bullet$ und $t \in s^\bullet$ gilt. ■2.3

Schlingen sind anschaulich die kürzest möglichen Zyklen in Netzen. Im Beispielnetz $\mathcal{N}_2$ sind t_4 und s_7 sowie t_4 und s_8 jeweils durch Schlingen miteinander verbunden.

Definition 2.4 *Disjunkte Netze*

Zwei Netze $\mathcal{N} = (S, T, W)$ und $\mathcal{N}' = (S', T', W')$ heißen *disjunkt*, falls ihre Knotenmengen disjunkt sind, also $(S \cup T) \cap (S' \cup T') = \emptyset$ gilt. ■2.4

Netze $\mathcal{N}_1$ und $\mathcal{N}_2$ aus Abb. 2.1 sind disjunkt; die Netze $\mathcal{N}_2$, $\mathcal{N}_3$ und $\mathcal{N}_4$ sind dagegen untereinander nicht disjunkt, da sie z. B. alle eine mit s_9 bezeichnete Stelle besitzen.

Definition 2.5 *Isomorphe Netze*

Seien $\mathcal{N} = (S, T, W)$ und $\mathcal{N}' = (S', T', W')$ zwei Netze, dann heißen $\mathcal{N}$ und $\mathcal{N}'$ zueinander *isomorph*, wenn es einen Isomorphismus $\iota : S \cup T \to S' \cup T'$ gibt mit $W(s,t) = W'(\iota(s), \iota(t))$ und $W(t,s) = W'(\iota(t), \iota(s))$ für alle $s \in S$ und $t \in T$.
■2.5

Anschaulich können isomorphe Netze durch geeignetes Umbenennen der Knoten ineinander überführt werden.

Definition 2.6 *Teilnetz*

Seien $\mathcal{N} = (S, T, W)$ und $\mathcal{N}' = (S', T', W')$ zwei Netze. $\mathcal{N}'$ heißt *Teilnetz* von $\mathcal{N}$, geschrieben $\mathcal{N}' \trianglelefteq \mathcal{N}$, wenn gilt

$$S' \subseteq S, \quad T' \subseteq T \quad \text{und} \quad W' = W|_{(S' \times T') \cup (T' \times S')} .$$

■2.6

Im obigen Beispiel ist $\mathcal{N}_4$ ein Teilnetz von $\mathcal{N}_2$, jedoch ist $\mathcal{N}_3$ kein Teilnetz von $\mathcal{N}_2$, da im Netz $\mathcal{N}_3$ keine Kante von t_5 nach s_9 existiert.

Netze lassen sich oft zerlegen in unabhängige Teilnetze. Zusammenhängende, voneinander isolierte Netzkomponenten modellieren dann nebenläufige, autonome Systemanteile, die mit den übrigen Komponenten nicht kommunizieren.

Definition 2.7 *Isolation, Komponente*

Sei $\mathcal{N} = (S, T, W)$ ein Netz und sei $\mathcal{N}' = (S', T', W')$ ein Teilnetz von $\mathcal{N}$.

(a) Ein Knoten $x \in S \cup T$ heißt *isoliert*, falls ${}^\bullet x \cup x^\bullet = \emptyset$ gilt.

(b) $\mathcal{N}'$ heißt *isoliertes Teilnetz* von $\mathcal{N}$, wenn für alle $x \in S' \cup T'$ gilt:
   (i) ${}^\bullet x \subseteq (S' \cup T')$ (keine Kanten führen in des Teilnetz hinein) und
   (ii) $x^\bullet \subseteq (S' \cup T')$ (keine Kanten führen aus dem Teilnetz heraus).

(c) $\mathcal{N}'$ heißt *(zusammenhängende) Komponente* von $\mathcal{N}$, wenn $\mathcal{N}'$ ein minimales isoliertes Teilnetz von $\mathcal{N}$ ist, d. h., wenn es kein echtes Teilnetz $\mathcal{N}'' \lhd \mathcal{N}'$ gibt, das in $\mathcal{N}'$ isoliertes Teilnetz ist. ■2.7

In Abbildung 2.1 ist die Stelle s_2 im Netz $\mathcal{N}_1$ isoliert; das Teilnetz $(\{s_2\}, \emptyset, \emptyset)$ ist somit eine Komponente von $\mathcal{N}_1$, das insgesamt aus drei Komponenten besteht; das Teilnetz $(\{s_1, s_2, s_4\}, \{e_1\}, \{(s_1, t_1), (t_1, s_4)\})$ ist zwar in $\mathcal{N}_1$ isoliert, aber keine Komponente desselben, da es weiter zerlegbar ist. Aufgrund der T–Eingeschränktheit können in den Netzen dieser Arbeit lediglich Stellen oder Teilnetze isoliert sein.

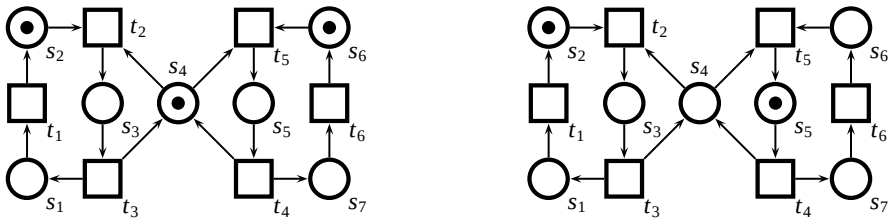
2.3 Netzsysteme

Neben den vorgestellten statischen Beziehungen zwischen Stellen und Transitionen können dynamische Eigenschaften eines Systems mit Netzen modelliert werden, mit anderen Worten Zustandsänderungen. Dazu wird eine veränderliche Markierung auf den Stellenkonten eingeführt, die den aktuellen Zustand des Systems widerspiegelt. Weiterhin wird eine Schaltregel (Transitionsregel) aufgestellt, die festlegt, wie einzelne Markierungen in Nachfolgemarkierungen übergehen.

Definition 2.8 *System*

Ein *(Netz–)System* ist ein Quadrupel $\Sigma = (S, T, W, M_0)$, wobei $\mathcal{N} = (S, T, W)$ das zugrunde liegende Netz ist und $M_0 : S \to \mathbb{N}$ die Anfangsmarkierung bezeichnet.

Das System Σ ist endlich, wenn das Netz $\mathcal{N}$ endlich ist. ■2.8



(a) System Σ_1.

(b) Σ_1 nach Schalten von t_5.

Abbildung 2.2: Beispiel für ein Netzsystem.

In dieser Arbeit werden alle modellierten Ausgangssysteme als endlich vorausgesetzt; die Verhaltensbeschreibung dieser Systeme, die in Abschnitt 2.5 eingeführt und ebenfalls durch Netzsysteme dargestellt wird, kann hingegen unendlich sein. Bei allen betrachteten Systemen sind die Vor– und Nachbereiche von Transitionen stets endlich.

Formal ist eine Markierung eine Multimenge über den Stellen des Netzes, die durch Aufzählung der markierten Stellen in ihrer Vielfachheit notiert werden. Graphisch werden Markierungen durch Eintragen von Marken (engl. *token*) als schwarze Punkte in den Stellenknoten repräsentiert, oder, bei unüberschaubarer Markenzahl, durch einen entsprechenden Zahleneintrag. Das System in Abb. 2.2(a) hat die Startmarkierung $M_0 = \{s_1 \mapsto 0, s_2 \mapsto 1, s_3 \mapsto 0, s_4 \mapsto 1, s_5 \mapsto 0, s_6 \mapsto 1\} = \{s_2, s_4, s_6\}$.
In der vorliegenden Arbeit soll die Kapazität der Stellen, d. h. die Anzahl maximal aufzunehmender Marken, nicht beschränkt sein.

Eine Zustandsänderung wird durch das *Schalten* einer Transition modelliert: aus jeder Stelle des Vorbereichs einer (aktivierten, d. h. schaltbaren) Transition werden so viele Marken entfernt, wie das Gewicht der verbindenden Kante groß ist (d. h. die Transition ist nur in dem Fall aktiviert, dass ihre Stellen im Vorbereich je mit ausreichend vielen Marken bestückt sind). Danach werden in jeder Stelle des Nachbereichs der Transition entsprechend der Gewichtung der verbindenden Kante Marken hinzugefügt. Dieser dynamische Aspekt von Petri–Netzen wird in der folgenden Definition formalisiert.

Definition 2.9 *Aktivierte Transition; Schaltregel*

Seien $\Sigma = (S, T, W, M_0)$ ein System, $M : S \to \mathbb{N}$ eine Markierung und $t \in T$ eine Transition.

(a) *M aktiviert t* (geschrieben $M \xrightarrow{t}$), falls gilt: $\forall s \in {}^\bullet t : M(s) \geq W(s, t)$.
Mit anderen Worten, jede Stelle im Vorbereich von t trägt mindestens so viele Marken, wie das verbindende Kantengewicht groß ist.

(b) Eine unter M aktivierte Transition kann *schalten* und produziert eine Nachfolgemarkierung M' (Schreibweise: $M \xrightarrow{t} M'$), falls gilt:

$$M \xrightarrow{t} \ \wedge \ \forall s \in S : \ M'(s) \ = \ M(s) - W(s, t) + W(t, s).$$

Ein solches Schalten wird auch *Markierungsänderung* oder *Zustandsübergang* genannt. ■2.9

Die Markierung M_0 des Systems aus Abbildung 2.2(a) aktiviert sowohl die Transition t_2 als auch t_5. Teilbild (b) zeigt das System nach Schalten von t_5; die erreichte Markierung $M_1 = \{s_2, s_5\}$ aktiviert nun ihrerseits t_4.
Im Zusammenhang mit der reflexiven transitiven Hülle $\longrightarrow^*$ der Zustandsübergangs–Relation werden die folgenden Sprechweisen verwendet.

Notation 2.10 *Schaltsequenz; erreichbare Markierung*

Seien $\Sigma = (S, T, W, M_0)$ ein System, $M^0 \xrightarrow{t^1} M^1 \xrightarrow{t^2} M^2 \ldots M^{n-1} \xrightarrow{t^n} M^n$ mit $t^i \in T, M^j : S \to \mathbb{N}, n \in \mathbb{N}, 0 \leq i, j \leq n$ eine Folge von Markierungsänderungen.

Dann ist M^n von M^0 aus über die Transitionsfolge $\sigma = t^1 t^2 \ldots t^n$ *erreichbar*, geschrieben $M^0 \xrightarrow{\sigma} M^n$; die Folge $\sigma \in T^*$ wird *Schaltsequenz* genannt.
Für die leere Sequenz ϵ gilt: $M^i \xrightarrow{\epsilon} M^i$.

$\mathcal{R}_M^\Sigma \ = \ \{ M' : S \to \mathbb{N} \mid \exists \sigma \in T^* : M \xrightarrow{\sigma} M' \}$ bezeichnet die Menge aller von $M : S \to \mathbb{N}$ aus erreichbaren Markierungen bezüglich des Systems Σ, auch *Zustandsraum* genannt. ■2.10

Mit Hilfe des Erreichbarkeitsbegriffs lassen sich einige fundamentale Eigenschaften von Netzsystemen beschreiben.

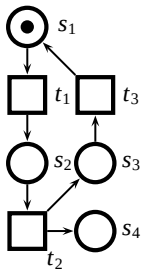
Definition 2.11 *Dynamische Eigenschaften von Systemen [Rei85]*

Sei $\Sigma = (S, T, W, M_0)$ ein System, und sei $n \in \mathbb{N}$.

(a) $M : S \to \mathbb{N}$ ist *n–beschränkt*, wenn $\forall s \in S : M(s) \leq n$ gilt.
Σ heißt *n–beschränkt*, wenn gilt: $\forall M \in \mathcal{R}_{M_0}^\Sigma : \ M$ ist n–beschränkt.
1–Beschränktheit wird auch *(1–)Sicherheit* bezeichnet.

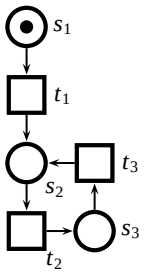
(b) $t \in T$ ist *lebendig*, wenn $\forall M \in \mathcal{R}_{M_0}^\Sigma \exists M' \in \mathcal{R}_M^\Sigma : M' \xrightarrow{t}$.
Σ ist *lebendig*, wenn $\forall t \in T : t$ ist lebendig.

(c) Eine Markierung $M : S \to \mathbb{N}$ heißt *tot*, wenn $\nexists t \in T : M \xrightarrow{t}$.
Σ besitzt eine *Verklemmung* (engl. *deadlock*), falls $\exists M \in \mathcal{R}_{M_0}^\Sigma : M$ ist tot, d. h. falls eine tote Markierung erreicht werden kann. ■2.11



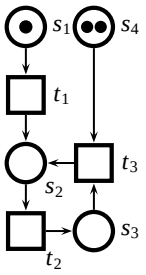
nicht beschränkt
lebendig
verklemmungsfrei

(a) Σ_2



sicher
nicht lebendig
verklemmungsfrei

(b) Σ_3



2–beschränkt
nicht lebendig
nicht verklemmungsfrei

(c) Σ_4

Abbildung 2.3: Dynamische Eigenschaften dreier Netzsysteme.

Sichere Netzsysteme sind eine sehr gut untersuchte Netzklasse [CEP95]. Sie besitzen eine nahe Verwandtschaft zu kommunizierenden endlichen Automaten und werden aus diesem Grund häufig eingesetzt zur Modellierung nebenläufiger Systeme.

Abbildung 2.3 zeigt drei Netzsysteme: Σ_2 ist in s_4 unbeschränkt, da mit jedem Durchlaufen des Zyklus eine zusätzliche Marke nach s_4 gelegt wird, die nie abgezogen werden kann. Σ_3 ist nicht lebendig, da t_1 nur ein einziges Mal schalten kann. In Σ_4 erlaubt s_4 über t_3 nur ein zweimaliges Durchlaufen der Schleife, die erreichbare Markierung $M' = \{s_3\}$ besitzt keine Nachfolgemarkierung, da sie keine Transition aktiviert.

Aus diesen Beispielen ergibt sich unmittelbar die Frage: wie weist man solche (und andere) dynamische Eigenschaften für beliebige Netzsysteme auf möglichst einfache und effektive Weise nach? Eine Möglichkeit bieten strukturelle Untersuchungen, beispielsweise über Invarianten oder Fallen [Sta90, DE95]; oft sind diese Methoden jedoch eher unvollständig.

Einen anderen Weg bietet die explizite Analyse der tatsächlich erreichbaren Markierungen: bei endlichem Zustandsraum kann eine erschöpfende Simulation bzw. die Generierung des kompletten Zustandsraumes vorgenommen werden, bei unendlich vielen Zuständen wird man sich mit dieser Methode auf eine endliche Teilmenge erreichbarer Markierungen einschränken müssen. Verschiedene Formalismen der Darstellung des Netzverhaltens beschreibt der nächste Abschnitt.

2.4 Verhaltensbeschreibungen von Netzen

Es gibt zwei allgemeine Sichten auf das dynamische Verhalten von Petri–Netzen: die erste Sichtweise, die sogenannte Interleaving– oder sequenzielle Semantik, betrachtet die Menge der möglichen Ausführungsfolgen des Systems. Die zweite Methode, kausale Netz–Semantiken, betrachtet halbgeordnete Abläufe (sogenannte Prozesse) des Systems. Beide Semantiken erlauben die Darstellung des Verhaltens als Menge von einzelnen Ausführungen (sogenanntes *Linear–Time*–Verhalten) oder vereinigt in einem einzigen Objekt (*Branching–Time*–Verhalten).

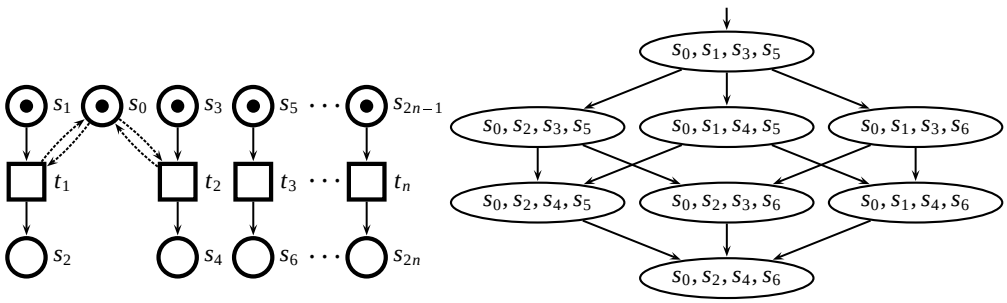
Interleaving–Semantiken

Die Interleaving–Semantik eines Netzsystems lässt sich einerseits beschreiben durch Aufzählen aller (maximalen, d. h. nicht verlängerbaren) Folgen von Markierungsänderungen. Eine andere Möglichkeit bietet die Aufstellung des Erreichbarkeitsgraphen. Die Knoten dieses Graphen bestehen aus den erreichbaren Markierungen, die Markierungsübergänge bilden die Kanten.

Definition 2.12 *Erreichbarkeitsgraph*

Sei $\Sigma = (S, T, W, M_0)$ ein Netzsystem.
Der verwurzelte, gerichtete und kantenbeschriftete Graph $\mathcal{RG}_\Sigma = ((V, E), M_0)$ mit $V = \mathcal{R}^\Sigma_{M_0}$ und $E = \{(M, t, M') \in \mathcal{R}^\Sigma_{M_0} \times T \times \mathcal{R}^\Sigma_{M_0} \mid M \xrightarrow{t} M'\}$ heißt *Erreichbarkeitsgraph* oder *Zustandsübergangsgraph* von Σ. ■2.12

In Abbildung 2.4(b) ist ein solcher Erreichbarkeitsgraph dargestellt. Ist ein System n–beschränkt, so gibt es nur endlich viele kombinatorische Möglichkeiten für Markierungen; die Anzahl der tatsächlich erreichbaren Markierungen liegt im Allgemeinen weit darunter, da längst nicht alle möglichen Markenbelegungen erreichbare Markierungen repräsentieren.
Mit dem Erreichbarkeitsgraphen steht ein einfach strukturiertes, endliches Objekt zur Verfügung, die Interleaving–Semantik erlaubt daher oft elegante, mit wenig Aufwand implementierbare Anwendungen. Ein schönes Beispiel hierfür ist der in [CES86] beschriebene Algorithmus zur Untersuchung von Systemeigenschaften, spezifiziert in einer modalen Logik.
Demgegenüber bringt diese Darstellung jedoch auch erhebliche Nachteile mit sich. Zum einen kann nicht zwischen Nebenläufigkeit und Nichtdeterminismus unterschieden werden, beispielsweise besitzt das System aus Abb. 2.4(a) stets den gleichen Erreichbarkeitsgraphen, unabhängig davon, ob die vier gestrichelt gezeichneten Kanten vorhanden sind oder nicht. Existieren diese Kanten, dann verursacht die einzelne Marke auf der gemeinsamen Stelle s_0 im Vorbereich von t_1, t_2 eine Konfliktsituation.



(a) System Σ_5. (b) $\mathcal{RG}_{\Sigma_5}$ $(n = 3)$.

Abbildung 2.4: Ein Netzsystem und sein Erreichbarkeitsgraph.

Neben diesem Informationsverlust ist ein – vor allem für praktische Anwendungen – meist schwerwiegenderer Nachteil das Problem der sogenannten *Zustandsraum–Explosion*. Durch die explizite Verwaltung aller erreichbaren Markierungen wächst, insbesondere bei einem hohen Grad an Nebenläufigkeit, die Größe des Erreichbarkeitsgraphen rasch an – die Anzahl der Zustände nebenläufiger Netzanteile ist ein multiplikativer Faktor. Ein extremes Beispiel zeigt Abb. 2.4(a), dessen Erreichbarkeitsgraph für eine Systemgröße von n genau 2^n Zustände umfasst, damit exponentiell größer ist als das Ausgangssystem.
In Anbetracht dieses recht einfach strukturierten Systems ist leicht vorstellbar, welche Probleme die Analyse wirklich interessanter, praktisch relevanter Netzsysteme aufwirft. Seit einiger Zeit versucht man deshalb, geschickte Codierungen des Zustandsraumes zu finden, so dass nur ein Ausschnitt der erreichbaren Zustände explizit gespeichert werden muss. Bekannte Verteter hierfür sind *Binary Decision Diagrams* (BDDs) [Bry86, BCM+92, Hu95], Netzentfaltungen [McM92] und Kronecker–Algebren [Kem96]. Weitere Techniken, der Zustandsraum–Explosion zu begegnen, sind das Ausnutzen von Symmetrien [Sta91, ES93] oder Halbordnungs–basierte Reduktionen, z. B. *Stubborn Sets* [Val92].

Halbordnungs–Semantiken

Das Verhalten von Netzsystemen lässt sich auf anschauliche Weise auch mit Hilfe von Abläufen beschreiben, diese stellen selbst wieder (spezielle) Petri–Netze dar, allerdings mit einfacherer Struktur. Beispielsweise enthalten sie keine Zyklen, den Knoten liegt demnach auf natürliche Weise eine Halbordnung zugrunde. Kausale Abhängigkeiten spiegeln sich unmittelbar in dieser Ordnung wider, Nichtdeterminismus wird durch Konflikte modelliert; Nebenläufigkeit wird durch das Fehlen dieser Relationen dargestellt. Räumlich getrennte Systemanteile finden sich wieder in unabhängigen Kompo-

nenten.
In der Literatur werden verschiedene Halbordnungs–basierte Netz–Semantiken vorgeschlagen, darunter Prozess–Semantiken [Rei85, BD87, BF88, Eng91], die die Umgebung von Transitionen, d. h. deren Vor– und Nachbereiche, erhalten, sowie *Executions* [Vog91], die das Schalten einer Transition quantitativ für jede lokal am Schaltvorgang beteiligte Stelle abbilden.
Beide Semantiken gestatten die Modellierung des Systemverhaltens als Menge einzelner Abläufe oder in Form eines einzigen Objekts, das die möglichen (verzweigenden) Abläufe vereint. Im Folgenden werden verzweigende Prozesse für die Darstellung der Netzentfaltung verwendet, wie auch in [McM92, ERV96]. Sie bieten eine natürliche Netz–Semantik, insbesondere für 1–sichere Netzsysteme.
Die Technik der Präfixbildung von Netzentfaltungen ist inzwischen auch auf Executions übertragen worden, [ERV00, Haa98]. Die einem Prozess entsprechende Execution–Darstellung enthält im Extremfall die doppelte Anzahl an Stellenknoten.
Struktur und Eigenschaften der Prozessen zugrunde liegenden Netze werden im nächsten Abschnitt beschrieben, anschließend wird der Prozess–Begriff formalisiert. Ihr Einsatz als kompakte Darstellung des Zustandsraumes eines Netzsystems wird Gegenstand von Kapitel 3 sein.

2.5 Occurrence–Netze

Dieser Abschnitt beschreibt, welche Gestalt die Netze besitzen, welche später als Grundlage für Prozess–Netze dienen. Dabei werden zwei Arten von Netzen unterschieden, die im Verzweigungsgrad der Stellenknoten differieren. Zunächst werden die im letzten Abschnitt erwähnten Relationen zwischen Knoten eines Netzes formal gefasst.

Definition 2.13 *Kausal–, Konflikt– und* co*–Relation*

Sei $\mathcal{N} = (S, T, W)$ ein Netz, und seien $x, x' \in S \cup T$ zwei Knoten von $\mathcal{N}$.

(a) x und x' stehen in *Kausal-Relation*, geschrieben $x \preceq x'$, falls es in $\mathcal{N}$ einen Pfad $x \ldots x'$ gibt. Die irreflexive Kausal–Relation wird mit $\prec$ notiert.

(b) x und x' stehen in *Konflikt*, geschrieben $x \# x'$, wenn gilt
$$\exists t, t' \in T,\ t \neq t',\ {}^\bullet t \cap {}^\bullet t' \neq \emptyset\ :\ (\ t \preceq x\ \wedge\ t' \preceq x'\ )\ ,$$
d. h. es gibt zwei verschiedene Pfade zu diesen Knoten, die in derselben Stelle beginnen und sich unmittelbar anschließend trennen.
Ein einzelner Knoten x steht in *Selbstkonflikt*, falls $x \# x$ gilt.

(c) x und x' stehen in co–Relation (engl. für *concurrent*, nebenläufig), kurz x co x', wenn sie weder in Kausal– noch in co–Relation stehen:
$$x \text{ co } x' \quad \text{gdw. } \neg\ (\ x \preceq x'\ \vee\ x' \preceq x\ \vee\ x \# x'\ )$$

■2.13

Der Begriff Kausal-Relation lässt sich sinnvoll nur in zyklenfreien Netzen festlegen. Aus diesem Grunde werden Netze, die in dieser Arbeit als Grundlage für die Verhaltensbeschreibung von Netzsystemen dienen, wie folgt definiert [Eng91].

Definition 2.14 *Occurrence–Netz, Kausalnetz*

Sei $\mathcal{O} = (B, E, F)$ ein Netz.

(a) $\mathcal{O}$ heißt *Occurrence–Netz*[2], falls es die folgenden Eigenschaften besitzt:

(i)	$F \subseteq (B \times E) \cup (E \times B)$	(lediglich einfache Kantengewichte),
(ii)	$F^+ \cap (F^{-1})^+ = \emptyset$	(Azyklizität),
(iii)	$\nexists e \in E : e \# e$	(keine Selbstkonflikte von Ereignissen),
(iv)	$\forall b \in B : \lvert {}^\bullet b \rvert \leq 1$	(Bedingung nicht rückwärts verzweigt).

In Occurrence-Netzen heißen die Elemente von B *Bedingungen* (engl. *conditions*) und die Elemente aus E *Ereignisse* (*events*).

(b) $\mathcal{O}$ heißt *Kausalnetz* , falls zusätzlich zu (i)–(iv) gilt

(v)	$\forall b \in B : \lvert b^\bullet \rvert \leq 1$	(Bedingungen nicht vorwärts verzweigt).

■2.14

Jedes Kausalnetz ist somit auch ein Occurrence–Netz. Abbildung 2.5(b) und (c) zeigt zwei Occurrence–Netze. Dagegen sind $\mathcal{O}_1$ und die Netze $\mathcal{N}_2$ und $\mathcal{N}_4$ aus Abb. 2.1 weder Occurrence– noch Kausalnetze.

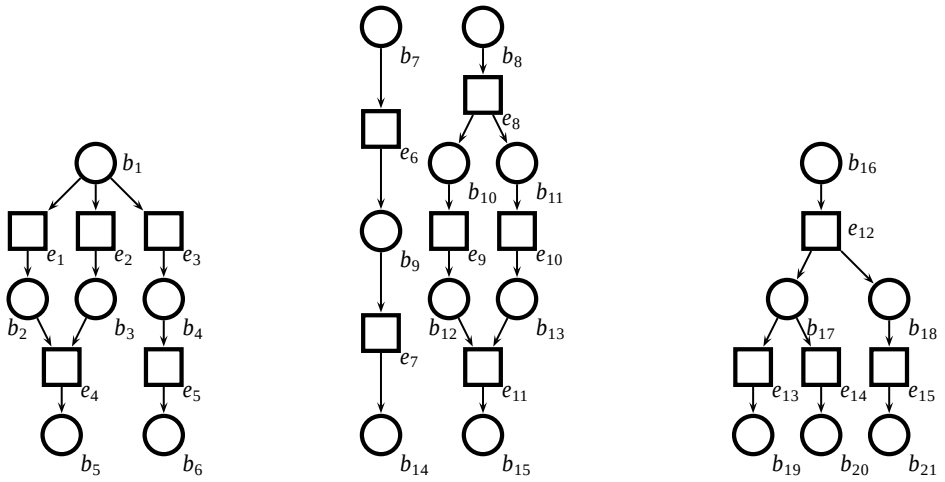
Wie oben angedeutet, liegt jedem Occurrence–Netz $\mathcal{O} = (B, E, F)$ wegen der Azyklizität eine Halbordnung $(X, \preceq) = (B \cup E, F^*)$ zugrunde, diese Ordnung ist zudem fundiert. Reflexivität und Transitivität von $\preceq$ folgen unmittelbar für die transitive Hülle F^* der Flussrelation.

Minimum bzw. Maximum von $\mathcal{O}$ werden direkt auf die Ordnung $\preceq$ zurückgeführt, sie sind definiert als $\mathrm{Min}(\mathcal{O}) = \mathrm{Min}(B)$ bzw. $\mathrm{Max}(\mathcal{O}) = \mathrm{Max}(B)$. Damit gilt beispielsweise für das Netz $\mathcal{O}_3$ in Abb. 2.5, dass $\mathrm{Min}(\mathcal{O}_3) = \{ b_{16} \}$ ist. Ebenfalls mit der Halbordnung ist der Begriff der Vorgängermenge anwendbar, so ist z. B. $\downarrow\{ b_{19}, b_{21} \} = \{ b_{16}, b_{17}, b_{18}, b_{19}, b_{21}, e_{12}, e_{13}, e_{15} \}$ und $\mathrm{Max}(\downarrow\{ b_{17} \}) = \{ b_{19}, b_{20} \}$.

In Kausalnetzen gibt es keine Konflikte, da in ihnen Bedingungen stets unverzweigt sind. Einen einzelnen Lauf des Systems betrachtend, hat an allen Bedingungen, die mehrfach verzweigenden Stellenknoten entsprechen[3], eine Entscheidung stattgefunden, welches Ereignis als nächstes geschaltet wird; die in dieser Situation ebenfalls möglichen Ereignisse sind mit dieser Entscheidung für diesen konkreten Ablauf ausgeschlossen worden.

[2] Es wird hier absichtlich die englische Bezeichnung verwendet, da es keine gute deutsche Entsprechung für diese Netzklasse gibt. Der am ehesten passende Begriff *Bedingungs–/Ereignis–Netz* definiert in [Rei85] bereits Netze mit anderen Eigenschaften.

[3] Wie diese Entsprechung von Bedingungen und Stellen aussieht, beschreibt der nächste Abschnitt.



(a) $\mathcal{O}_1$ (b) $\mathcal{O}_2$ (c) $\mathcal{O}_3$

Abbildung 2.5: Drei Netze: $\mathcal{O}_1$ ist kein Occurrence–Netz; $\mathcal{O}_2$ ist Kausalnetz; $\mathcal{O}_3$ ist Occurrence–Netz, aber kein Kausalnetz.

In den Beispielen aus Abb. 2.5 stehen die Bedingungen b_{19} und b_{20} des Netzes $\mathcal{O}_3$ in Konflikt; Ereignis e_4 von $\mathcal{O}_1$ steht in Selbstkonflikt, daher ist es kein Occurrence–Netz. Das Netz $\mathcal{O}_2$ hingegen enthält überhaupt keine Konflikte.

Zur Vereinfachung einiger Definitionen wird nachfolgend ein spezielles virtuelles Ereignis eingeführt.

Notation 2.15 *Leeres Ereignis*

Sei $\mathcal{O} = (B, E, F)$ ein Occurrence–Netz mit der Ordnung $(B \cup E, \prec)$. Das *leere Ereignis* $\perp \in E$ wird definiert mit der Eigenschaft $\forall x \in B \cup E \setminus \perp : \perp \prec x$.

■2.15

Die Tiefe eines Ereignisses ergibt sich aus der Länge des längsten Pfades vom Minimum zu diesem Ereignis. Sie ist folgendermaßen definiert.

Definition 2.16 *Tiefe eines Ereignisses*

Seien $\mathcal{O} = (B, E, F)$ ein Occurrence–Netz und $(B \cup E, \prec)$ die zugrunde liegende Halbordnung. Die *Tiefe*eines Ereignisses $e \in E$ ist definiert als

$$d(e) = \max\{|\{e_1, e_2, \ldots, e\}| \mid \{e_1, e_2, \ldots, e\} \subseteq {\downarrow}e \wedge e_1 \prec e_2 \prec \ldots \prec e\}$$

■2.16

Dem leeren Ereignis wird per definitionem die Tiefe 0 zugeordnet, $d(\perp) = 0$. Für die Bestimmung der Tiefe einer Bedingung wird unmittelbar auf das (eindeutige) Ereignis in dessen Vorbereich zurückgegriffen: es ist $d(b) = 0$ für $b \in \mathrm{Min}(\mathcal{O})$, bzw. $d(b) = d({}^\bullet b)$ für $b \in B \setminus \mathrm{Min}(\mathcal{O})$.

Im Rahmen der Implementierung ermöglicht das folgende Lemma eine konstruktive Berechnung der Tiefe eines Ereignisses; sinnvollerweise wird hierzu in jedem Ereignisknoten dessen Tiefe gespeichert, damit sie nicht immer wieder neu berechnet werden muss.

Lemma 2.17 *Rekursive Ermittlung der Tiefe*

Sei $\mathcal{O} = (B, E, F)$ ein Occurrence–Netz und $e \in E$ ein Ereignis mit Tiefe $d(e) \in \mathbb{N}$. Es gilt

$$d(e) = \begin{cases} 1 & \text{falls } e \in Min(\mathcal{O})^\bullet, \\ \max\{d(e') \mid e' \in {}^{\bullet\bullet}e\} + 1 & \text{sonst}. \end{cases}$$

Beweis: Induktion über die Vorgängerpfade von e. ■2.17

Die folgende Definition betrachtet Mengen von Elementen, die untereinander paarweise in co–Relation stehen.

Definition 2.18 *co–Menge; Schnitt*

Sei $\mathcal{O} = (B, E, F)$ ein Occurrence–Netz.

(a) Eine Menge $c \subseteq X$ heißt co–Menge, falls gilt: $\forall x_1, x_2 \in c, x_1 \neq x_2 : x_1 \,\mathrm{co}\, x_2$.

(b) Ein *Schnitt* (engl. *cut*) $c \subseteq X$ ist eine maximale co–Menge, d. h. es gilt zusätzlich $\forall z \in X \setminus c \;\; \exists x \in c : \neg(x \,\mathrm{co}\, z)$.

(c) Ein Schnitt $c \subseteq X$ heißt *B–Schnitt*, falls zusätzlich $c \subseteq B$ gilt. ■2.18

Da in dieser Arbeit endliche Ausgangssysteme vorausgesetzt wurden, sind Schnitte ebenfalls stets endliche Mengen. Ein Schnitt durch das Netz $\mathcal{O}_2$ (Abb. 2.5) ist beispielsweise $\{b_9, e_9, e_{10}\}$. Die Mengen $\{b_{19}, b_{21}\}$ und $\{b_{20}, b_{21}\}$ stellen B–Schnitte durch das Occurrence–Netz $\mathcal{O}_3$ dar.

In Kausalnetzen teilt ein Schnitt c das Netz anschaulich in genau zwei Teile: wegen der Maximalität von c steht jeder Knoten $x \in X$ in Kausal–Relation mit mindestens einem Element des Schnittes (da es keine Konflikte gibt), d. h. es gilt $x \preceq c \vee c \prec x$; damit „liegt“ x entweder vor oder nach dem Schnitt (der Schnitt selbst kann als Grenzbereich einem der beiden Teile zugeordnet werden).

2.6 Prozesse von Netzsystemen

Für eine präzise Darstellung des Verhaltens eines Systems muss eine Beziehung zwischen Systemnetz und Occurrence–Netz hergestellt werden. Die folgende Definition beschreibt eine strukturerhaltende Abbildung zwischen Netzen.

Definition 2.19 *Netzhomomorphismus*

Seien $\mathcal{N}_1 = (S_1, T_1, W_1)$ und $\mathcal{N}_2 = (S_2, T_2, W_2)$ zwei Netze, wobei in $\mathcal{N}_1$ lediglich einfache Kantengewichte vorkommen, d. h. $W_1(x, x') \leq 1$ für Knoten $x, x' \in S_1 \cup T_1$. Eine Abbildung $h : S_1 \cup T_1 \to S_2 \cup T_2$ mit

(i) $h(S_1) \in \mathcal{M}_{\mathcal{F}}(S_2)$ und $h(T_1) \in \mathcal{M}_{\mathcal{F}}(T_2)$ (h konserviert Knotentypen) und

(ii) $\forall e \in T_1 \, \forall s \in S_2 : \; (\; W_2(s, h(e)) = |h^{-1}(s) \cap {}^\bullet e| \; \wedge \; W_2(h(e), s) = |h^{-1}(s) \cap e^\bullet| \;)$
(h konserviert lokale Umgebungen von Transitionen)

heißt *Netzhomomorphismus* von $\mathcal{N}_1$ und $\mathcal{N}_2$. ■2.19

Die Erweiterung der Abbildung h auf Sequenzen bzw. Mengen von Knoten erfolgt elementweise. Für $x_1, x_2, \ldots, x_n \in S_1 \cup T_1$ gilt

$$\begin{array}{rcll} h(x_1 x_2 \ldots x_n) & = & h(x_1)h(x_2)\ldots h(x_n) & \in (S_2 \cup T_2)^n \\ h(\{x_1, x_2, \ldots, x_n\}) & = & \{h(x_1), h(x_2), \ldots, h(x_n)\} & \in \mathcal{M}_{\mathcal{F}}(S_2 \cup T_2) \end{array}$$

Das Bild einer Knotenmenge aus $\mathcal{N}_1$ ist also eine Multimenge von Knoten aus $\mathcal{N}_2$. Die Erweiterung der Umkehrung h^{-1} wird analog definiert.
Mit Hilfe des obigen Netzhomomorphismus stellt die folgende Definition den Bezug zwischen einem Netzsystem und einem Occurrence-Netz her.

Definition 2.20 *Prozess–Axiome [BF88, Eng91]*

Seien $\Sigma = (S, T, W, M_0)$ ein System, $\mathcal{O} = (B, E, F)$ ein Occurrence–Netz und $\pi : B \cup E \to S \cup T$ ein Netzhomomorphismus.

(a) $(\mathcal{O}, \pi)$ heißt *kausaler Prozess* von Σ, wenn gilt

(o) $\mathcal{O}$ ist ein Kausalnetz.

(i) $\mathrm{Min}(\mathcal{O})$ ist ein B–Schnitt von $\mathcal{O}$.

(ii) $\forall s \in S$: $M_0(s) = |\pi^{-1}(s) \cap \mathrm{Min}(\mathcal{O})|$
($\mathrm{Min}(\mathcal{O})$ korrespondiert mit der Startmarkierung M_0).

(iii) $\forall e \in E : |E \cap [\mathrm{Min}(\mathcal{O}), \{e\}]| \in \mathbb{N}$
($\mathcal{O}$ ist $\mathrm{Min}(\mathcal{O})$–diskret, d. h. für jedes Ereignis besteht die Vormenge aus endlich vielen Ereignissen).

(b) $\mathcal{B} = (\mathcal{O}, \pi)$ heißt *(verzweigender) Prozess* (engl. *branching process*) von Σ, falls (i)–(iii) gelten und außerdem

(iv) $\forall e_1, e_2 \in E : ((^\bullet e_1 = {}^\bullet e_2 \wedge \pi(e_1) = \pi(e_2)) \Rightarrow e_1 = e_2)$
(In jeder Situation kann zu einer Transition von Σ das entsprechende Ereignis in $\mathcal{O}$ höchstens einmal gewählt werden; Transitionen werden nicht dupliziert).

■2.20

Die natürliche Startmarkierung eines Prozesses legt auf jede Bedingung $b \in \mathrm{Min}(\mathcal{O})$ genau eine Marke; in den graphischen Repräsentationen in dieser Arbeit wird sie weggelassen, da Der Platz in den Knoten bei den graphischen Darstellungen kann somit für die Repräsentation der Abbildung π verwendet werden. Hervorzuheben ist, dass Prozesse stets sichere Systeme darstellen.
Für eine ausführliche Erläuterung von Definition 2.20 sei auf [BF88, Eng91] verwiesen. Sie soll hier lediglich anhand eines Beispiels verdeutlicht werden. Dazu ist in Abb. 2.6(a) das System Σ_6 angegeben. Teilbilder (b) und (c) der gleichen Abbildung zeigen zwei mögliche (nicht–verzweigende) Prozesse von Σ_6. Die Funktion π wird dadurch gekennzeichnet, dass $\pi(x)$ für ein Element x des Prozess–Netzes in den x repräsentierenden Knoten geschrieben wird. Beispiele für verzweigende Prozesse des gleichen Systems sind in Abb. 2.6(d) und (e) dargestellt.

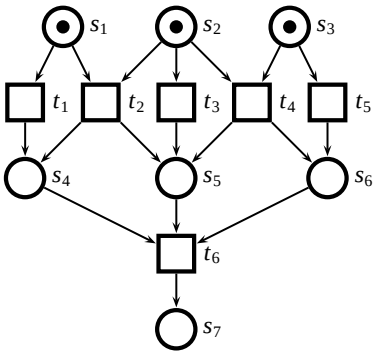
Kausale Prozesse dienen zur Modellierung *einzelner* Durchläufe eines Systems; Konflikte und somit Nichtdeterminismus werden durch die Angabe einer Menge einzelner kausaler Prozesse beschrieben. Mit verzweigenden Prozessen lassen sich unterschiedliche Ausführungen in einem einzigen Objekt, einschließlich nichtdeterministischer Alternativen modellieren.
Verschiedene verzweigende Prozesse des gleichen Netzsystems unterscheiden sich darin, wie weit sie das Verhalten desselben „auffalten“. Die folgende Definition formalisiert eine Ordnung auf diesen Anfangsstücken von Prozessen eines Systems.

Definition 2.21 *Präfix eines Prozesses [Eng91, ERV96]*

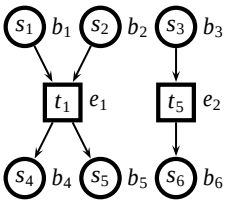
Seien $\mathcal{B} = (\mathcal{O}, \pi)$ und $\mathcal{B}' = (\mathcal{O}', \pi')$ mit $\mathcal{O}' = (B', E', F')$ verzweigende Prozesse eines Netzsystems Σ. Prozess $\mathcal{B}'$ heißt *Präfix* von $\mathcal{B}$, geschrieben $\mathcal{B}' \trianglelefteq_p \mathcal{B}$, falls er die folgenden Eigenschaften besitzt:

(i)	$\mathcal{O}' \trianglelefteq \mathcal{O}$	(Teilnetz);
(ii)	$\mathrm{Min}(\mathcal{O}) \subseteq \mathcal{O}'$	(Startmarkierung enthalten);
(iii)	$\forall b' \in B' : {}^\bullet b' \subseteq E'$ ($^\bullet$ in $\mathcal{O}$)	(Bedingungs–Vorbereich konserviert);
(iv)	$\forall e' \in E' : {}^\bullet e' \cup e'^\bullet \subseteq B'$ ($^\bullet$ in $\mathcal{O}$)	(Ereignis–Umgebungen konserviert);
(v)	$\pi' = \pi\vert_{B' \cup E'}$	(Restriktion der Beschriftungen).

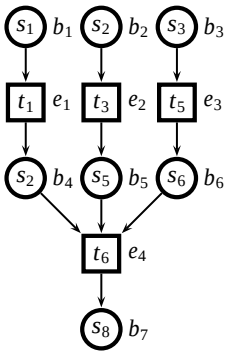
■2.21



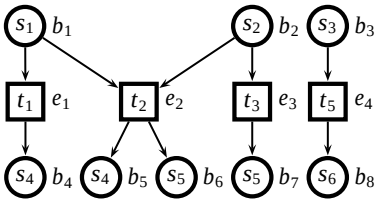
(a) System Σ_6



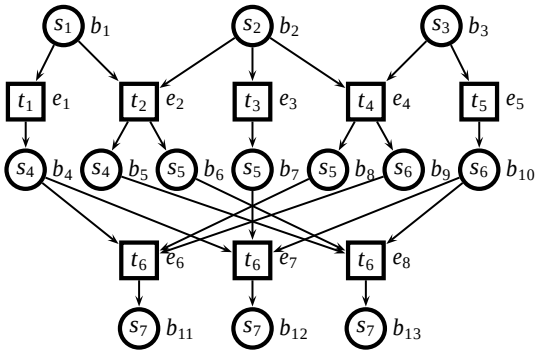
(b) kausaler Prozess von Σ_6



(c) kausaler Prozess von Σ_6



(d) verzweigender Prozess von Σ_6



(e) (max.) verzweigender Prozess von Σ_6

Abbildung 2.6: Ein System und einige seiner Prozesse.

In [Eng91] wird gezeigt, dass die Menge der Präfixe eines Netzsystems einen vollständigen *Verband* bezüglich der Ordnung $\trianglelefteq_p$ bildet. Dieser Verband besitzt einen (bis auf Isomorphie eindeutigen *maximalen* verzweigenden Prozess, eben gerade die *(maximale) Entfaltung*, die im Folgenden mit $\mathcal{B}_m$ oder $\mathsf{Unf}(\Sigma)$ bezeichnet wird.
Anschaulich kann man sich die Entfaltung als eine Schleifen auflösende Auffaltung des Systems vorstellen, diese ist im Allgemeinen unendlich. In der Entfaltung sind damit alle (auch unverzweigten Kausal–) Prozesse von Σ enthalten. Abbildung 2.6(e) stellt die Entfaltung des Systems aus Abb. 2.6(a) dar, welche in diesem Fall endlich ist.
Nachdem der statische Bezug zwischen Systemnetz und Prozess–Netz hergestellt ist, folgt nun die Übertragung des dynamischen Aspekts.

Definition 2.22 *Repräsentation einer Markierung*

Seien Σ ein Netzsystem, $\mathcal{B} = ((B, E, F), \pi)$ ein verzweigender Prozess von Σ. Markierung $M : S \to \mathbb{N}$ ist in $\mathcal{B}$ *repräsentiert*, falls $\exists$ B–Schnitt $c \subseteq B : \pi(c) = M$.
■2.22

Mit dieser Definition und Ergebnissen aus [BF88, Eng91] lässt sich der folgende fundamentale Zusammenhang zwischen Markierungen eines Netzsystems und B–Schnitten durch zugehörige Präfixe der Entfaltung herleiten.

Lemma 2.23 *Erreichbare Markierungen und ihre Repräsentationen*

Seien $\Sigma = (S, T, W, M_0)$ ein Netzsystem, $\mathcal{B} \trianglelefteq_p \mathcal{B}_m$ ein Präfix der Entfaltung von Σ. Für eine Markierung $M : S \to \mathbb{N}$ gilt:

(a) M ist in $\mathcal{B}$ repräsentiert $\Rightarrow$ $M \in \mathcal{R}^{\Sigma}_{M_0}$.

(b) $M \in \mathcal{R}^{\Sigma}_{M_0}$ $\Rightarrow$ M ist in $\mathcal{B}_m$ repräsentiert. ■*2.23*

Damit ist jede erreichbare Markierung in der maximalen Entfaltung als erreichbarer B–Schnitt enthalten – und umgekehrt; dies stellt eine sehr schwache Äquivalenzrelation dar. In einem endlichen Präfix sind ebenfalls nur erreichbare Markierungen repräsentiert, lediglich ungewiss ist, ob in diesem konkreten Präfix *alle* erreichbaren Markierungen repräsentiert sind.
Da in einem beschränkten Netzsystem nur endlich viele verschiedene Markierungen erreichbar sind, folgt unmittelbar, dass es eine endliche Approximation der maximalen Entfaltung geben muss, in der alle erreichbaren Markierungen repräsentiert sind. Welche Eigenschaften eine solche Approximation besitzen muss und wie sie konstruktiv berechnet werden kann, ist Gegenstand des nächsten Kapitels.

Kapitel 3

„ Ἀμείνω δ' αἴσιμα πάντα."
(Besser bei allem ist Ordnung.)
HOMER, ODYSSEE, GESANG VII, 310

Konstruktion von endlichen Präfixen

Nachdem der Begriff des endlichen Präfixes der Netzentfaltung eines beschränkten Netzsystems mathematisch gefasst und für beschränkte Netzsysteme die Existenz eines alle erreichbaren Markierungen repräsentierenden Anfangsstücks nachgewiesen worden ist, wird nun ausgeführt, auf welche Weise es konstruktiv berechnet werden kann. Nähert man sich diesem Problem zunächst auf naive Weise, so müssten für eine effektive Konstruktion eines solchen Präfixes alle erreichbaren Markierungen dahingehend überprüft werden, ob diese im konstruierten Präfix repräsentiert sind. Um diese Vergleiche effizient gestalten zu können, wären hierzu alle erreichbaren Schnitte durch das Präfix in irgendeiner Form explizit zu speichern – damit wäre die Konstruktion eines Präfixes im Platzverbrauch genauso teuer wie die des entsprechenden Erreichbarkeitsgraphen.
Die Schlüsselbeobachtung von McMillan [McM92, McM95] war es, dass bei weitem nicht alle erreichbaren Schnitte für Vergleiche herangezogen werden müssen. Vielmehr genügt es, diejenigen Schnitte zu betrachten, die von der *Geschichte* einzelner Ereignisse induziert werden, d. h. der minimalen Schaltfolge von Ereignissen, die zum Eintritt eines einzelnen Ereignisses notwendig sind. Auf diese Weise muss im Allgemeinen nur eine kleine Teilmenge der tatsächlich erreichbaren Markierungen gespeichert werden.
Anschaulich muss bei Auffinden einer bereits repräsentierten Markierung die Konstruktion anschließend nur noch hinter einem bestimmten der beiden Schnitte, die dieser Markierung entsprechen, fortgesetzt werden. Die Auswahl dieses Schnittes basiert auf der Festlegung einer speziellen Ordnung. Mehrere mögliche Ordnungen werden in diesem Kapitel untersucht.

Ein Algorithmus zur Prozess–Generierung von beschränkten Netzsystemen wird in Abschnitt 3.1 vorgestellt; dieser wird später in 3.4 derart modifiziert und verfeinert, dass er ein endliches Präfix der Entfaltung generiert. Im Gegensatz zu [McM95] erfolgt

die Darstellung des Algorithmus in einer etwas allgemeineren Form, um die anschließenden Verbesserungen einfacher beschreiben zu können. Die für die Konstruktion notwendigen Begriffe werden in Anlehnung an [ERV96, ERV00] eingeführt.
Neben dem Begriff der *Konfiguration* wird vor allem das *Cut-off–Kriterium* definiert, welches die Terminierung des Verfahrens beeinflusst. Daneben werden essenzielle Kriterien diskutiert, die ein gut geartetes Präfix erfüllen sollte, um für die Untersuchung von Zustandsräumen geeignet zu sein: Vollständigkeit und Minimalität.
In der ursprünglichen Version der Algorithmus von McMillan [McM95] kann ein generiertes Präfix der Entfaltung exponentiell kleiner als der Zustandsraum des Systems sein. Im ungünstigsten Fall kann es jedoch vorkommen, dass ein sogar exponentiell größeres Präfix erzeugt wird. Abschnitt 3.5 zeigt Lösungen für diverse Systemklassen auf, dieses Problem zu umgehen.

3.1 Induktive Prozess–Generierung

Die Ordnung $\trianglelefteq_p$ auf Präfixen liefert eine intuitive Idee für die Konstruktion von Entfaltungen: ausgehend vom minimalen Element des Verbandes, dem verzweigenden Prozess, der die Startmarkierung repräsentiert, wird dieser durch Hinzufügen von Ereignissen und Bedingungen Schritt für Schritt entlang der Ordnung $\trianglelefteq_p$ *erweitert*.
Unter möglichen Erweiterungen werden dabei im Folgenden Mengen von Ereignissen der maximalen Entfaltung verstanden, deren Vorbereich komplett im bereits konstruierten Teil der Entfaltung enthalten ist, die selbst jedoch noch nicht Elemente dieses bislang konstruierten Teils sind.
Der nachstehende Algorithmus formalisiert die Idee der automatischen Generierung von Entfaltungen; er berechnet auf induktive Weise die maximale Entfaltung eines beliebigen *n*–beschränkten Netzsystems.

Algorithmus 3.1 *Konstruieren der Entfaltung eines beschränkten Netzsystems*

Eingabe: Beschränktes Netzsystem $\Sigma = (\mathcal{N}, M_0)$.

Ausgabe: Entfaltung $\mathsf{Unf}(\Sigma)$.

```
1     begin
2       Initialisiere Unf durch Übertragen der Startmarkierung M_0;
3       erw := Menge der möglichen Erweiterungen von Unf;
4       while erw ≠ ∅ do
5         Wähle aus erw ein Ereignis e aus und füge es zu Unf hinzu, mit-
6         samt neuen Bedingungen, die dem Nachbereich π(e)• entsprechen;
7         erw := Menge der möglichen Erweiterungen von Unf
8       endwhile;
9       return Unf
10    end                                                          ■3.1
```

Von einer konkreten Datenstruktur für die Verwaltung von Netzen wird an dieser Stelle bewusst abstrahiert, darauf wird später in Kapitel 8 ausführlich eingegangen.

Die Initialisierung in Zeile 2 des Algorithmus ist folgendermaßen zu verstehen: jede anfänglich markierte Stelle des Systems wird – entsprechend der Zahl der Marken oft – durch Einfügen einer Bedingung in das zunächst leere Prozess–Netz übernommen. *Unf* besteht nach dieser Anweisung also lediglich aus isolierten Bedingungen, eine für jede Marke von M_0; die Menge der Ereignisse ist leer, ebenso die Flussrelation.
Mit der Berechnung der Menge der *möglichen Erweiterungen* (Zeilen 3 und 7) werden solche Ereignisse ermittelt, die noch nicht darin enthalten sind und die den bereits konstruierten Teil des Prozesses derart expandieren würden, dass wieder ein verzweigender Prozess gemäß Def. 2.20 entstünde.
Die gefundenen Ereignisse werden jedoch vorerst noch nicht in *Unf* eingefügt – dies geschieht erst nach und nach[1] in der Hauptschleife, Zeilen 4 bis 8.
In dieser Schleife wird der bereits konstruierte Teil der Entfaltung induktiv erweitert durch das Einfügen jeweils eines Ereignisses e aus der Menge *erw*. Außerdem wird dessen lokaler Nachbereich erzeugt, also Bedingungen für jede Stelle $s \in \pi(e)^\bullet$, je so viele, wie das verbindende Kantengewicht $W(\pi(e), s)$ groß ist. Die Menge der möglichen Erweiterungen wird nach jedem Einfügen aktualisiert (Zeile 7).
Die Schleife wird beendet, wenn alle erweiternden Ereignisse in die Entfaltung eingebaut worden sind. Im Allgemeinen terminiert der beschriebene Algorithmus damit nicht, nämlich gerade dann, wenn das übergebene Netzsystem unendliche Schaltsequenzen besitzt.

Eine wichtige Nebenbedingung muss zu diesem Algorithmus getroffen werden: ist die zu berechnende Entfaltung nicht endlich, könnte es passieren, dass lediglich entlang einiger Systemkomponenten entfaltet wird (in einer Art Tiefensuche), andere Komponenten jedoch nicht in die Generierung der Entfaltung einbezogen werden. Aus diesem Grund muss zusätzlich eine Fairness–Annahme getroffen werden, dass sämtliche Komponenten beim Entfalten Berücksichtigung finden. Diese Fairness wird später bei der Erzeugung eines vollständigen Präfixes garantiert, indem die Entfaltung in einer Art Breitensuche abgeschritten wird: das aus der Menge *erw* ausgewählte Ereignis (Zeile 5 in Algorithmus 3.1) wird stets minimale Tiefe bezüglich den Ereignissen aus *erw* besitzen.

Ähnlich, wie in [BD87] das Zusammenfallen der Kausalnetze von axiomatischer und induktiver Prozess–Definition nachgewiesen wird, lässt sich beweisen, dass die Algo-

[1] Der Grund für das sukzessive Einfügen einzelner Ereignisse ist rein technischer Natur, im Vorgriff auf Algorithmus 3.12, der eine in bestimmtem Sinne geordnete Konstruktion verlangt. In Alg. 3.1 könnten die in der Menge *erw* gesammelten Ereignisse in Zeile 5/6 alternativ in einem einzigen Schritt in *Unf* eingebaut werden.

rithmus 3.1 zugrunde liegende induktive Prozess–Konstruktion tatsächlich Prozesse im Sinne der axiomatischen Definition 2.20 erzeugt. Dieser Beweis wird hier nicht geführt.

McMillan hat in [McM92] einen Algorithmus beschrieben, welcher induktiv für beliebige beschränkte Systeme ein endliches Anfangsstück der maximalen Entfaltung konstruiert. Dieses Präfix ist groß genug, das Verhalten des Systems konsistent und vollständig zu beschreiben, d. h. insbesondere sind darin alle erreichbaren Markierungen codiert. Im Allgemeinen ist es aber wesentlich kleiner als der Erreichbarkeitsgraph.

Die Grundidee des Verfahrens ist die gleiche wie in Alg. 3.1: das Präfix der Entfaltung wird, ausgehend von der Startmarkierung, induktiv berechnet durch Hinzufügen einzelner Ereignisse und deren Nachbereich. Um jedoch eine Terminierung zu garantieren, wird ein sogenanntes *Cut-off–Kriterium* hinzugezogen: mit jedem neu eingefügten Ereignis wird überprüft, ob die von ihm induzierte erreichbare Markierung (bezogen auf das Netzsystem) schon anderswo im bereits konstruierten Teil des Präfixes von einem Ereignis induziert wird. Ist dies nicht der Fall, wird mit dem Einfügen fortgefahren; andernfalls jedoch haben die von den beiden betrachteten Ereignissen induzierten Schnitte – übertragen auf die maximale Entfaltung – eine isomorphe Nachfolgermenge (anschaulich ihre *Zukunft*). Konsequenterweise wird die Konstruktion lediglich hinter einem (bestimmten) der beiden Ereignisse fortgesetzt; das andere wird als Cut-off–Ereignis markiert.

Ausgehend von dieser informellen Beschreibung der Präfix–Generierung werden im folgenden Abschnitt die notwendigen Vorarbeiten zur Formalisierung des Algorithmus geleistet. Ein wichtiger Punkt wird hierbei die Bestimmung eines geeigneten Cut–off–Kriteriums sein, über das die Terminierung der Präfix–Generierung kontrolliert wird.

3.2 Konfigurationen und induzierte Schnitte

Zwei zentrale Begriffe im Zusammenhang mit Prozessen sind *(B–)Schnitte* und *Konfigurationen*. Ein B–Schnitt durch ein Prozess–Netz entspricht einer erreichbaren Markierung des Prozesses, die, vermöge der Abbildung π, mit einer entsprechenden Markierung im zugrunde liegenden Netzsystem korrespondiert [BF88, Eng91].

Während ein B–Schnitt aus einer Menge von Bedingungen geformt wird, bestehen Konfigurationen aus Ereignissen. Eine Konfiguration beschreibt, ähnlich einem kausalen Prozess, einen einzelnen möglichen Ablauf, ein mögliches Schalten von Ereignissen des Prozess–Netzes – und damit implizit (über π) eine mögliche Schaltfolge von Transitionen im zugrunde liegenden Netzsystem.

Definition 3.2 *(Lokale) Konfiguration*

Sei $\mathcal{O} = (B, E, F)$ ein Prozess–Netz.

(a) Eine Menge $C \subseteq E$ von Ereignissen heißt *Konfiguration*, falls gilt

(i) $\downarrow C = C$ (C ist kausal abgeschlossen).
(ii) $\forall e, e' \in C : \neg(e \# e')$ (C ist konfliktfrei).

(b) $\mathcal{C}_{\mathcal{O}} = \{ C \subseteq E \mid C \text{ ist Konfiguration} \}$ bezeichne die Menge der Konfigurationen von $\mathcal{O}$.

(c) Für ein Ereignis $e \in E$ definiert $[e] = E \cap \downarrow e$ dessen *lokale Konfiguration*[2].

■3.2

Das virtuelle Ereignis $\perp$ ist Element jeder Konfiguration, damit ist insbesondere $C = \{\perp\}$ eine Konfiguration, für die gilt $[\perp] = \emptyset$. Durchschnitt bzw. Vereinigung von Konfigurationen liefern wieder eine Konfiguration. Die lokale Konfiguration $[e]$ enthält gerade alle Ereignisse (und nur diese, inklusive e), die zum Eintritt von e bereits geschaltet haben müssen, bildlich gesprochen die *Geschichte* von e.
Im Prozess aus Abb. 3.1(c) sind $\{\perp, e_3, e_4, e_7, e_8\}$ und $\{\perp, e_1, e_4, e_5\}$ Beispiele für Konfigurationen, $[e_{15}] = \{\perp, e_3, e_7, e_{11}, e_{15}\}$ ist eine lokale Konfiguration. $\{\perp, e_3, e_6, e_7\}$ ist keine Konfiguration (Bedingung (i) verletzt), ebenso $\{\perp, e_1, e_4, e_5, e_9\}$ (wegen $e_4 \# e_9$). Da das virtuelle Ereignis Element jeder Konfiguration ist, wird es in den folgenden Darstellungen nicht jedesmal aufgezählt.

Endliche Konfigurationen und Schnitte sind eng miteinander verbunden: das Schalten der Ereignisse einer Konfiguration induziert einen B–Schnitt durch das Prozess–Netz. Im Netz aus Abb. 3.1(c) wird von der Konfiguration $C_x = \{e_3, e_4, e_7, e_8\}$ der Schnitt $c_x = \{b_1, b_{10}, b_{11}\}$ induziert. Ein solcher Schnitt lässt sich folgendermaßen konstruktiv berechnen.

Notation 3.3 *Die Abbildungen* Cut *und* Mark[3]

Sei $\mathcal{B} = (\mathcal{O}, \pi)$ ein verzweigender Prozess und sei $C \in \mathcal{C}_{\mathcal{O}}$ eine endliche Konfiguration. Definiere

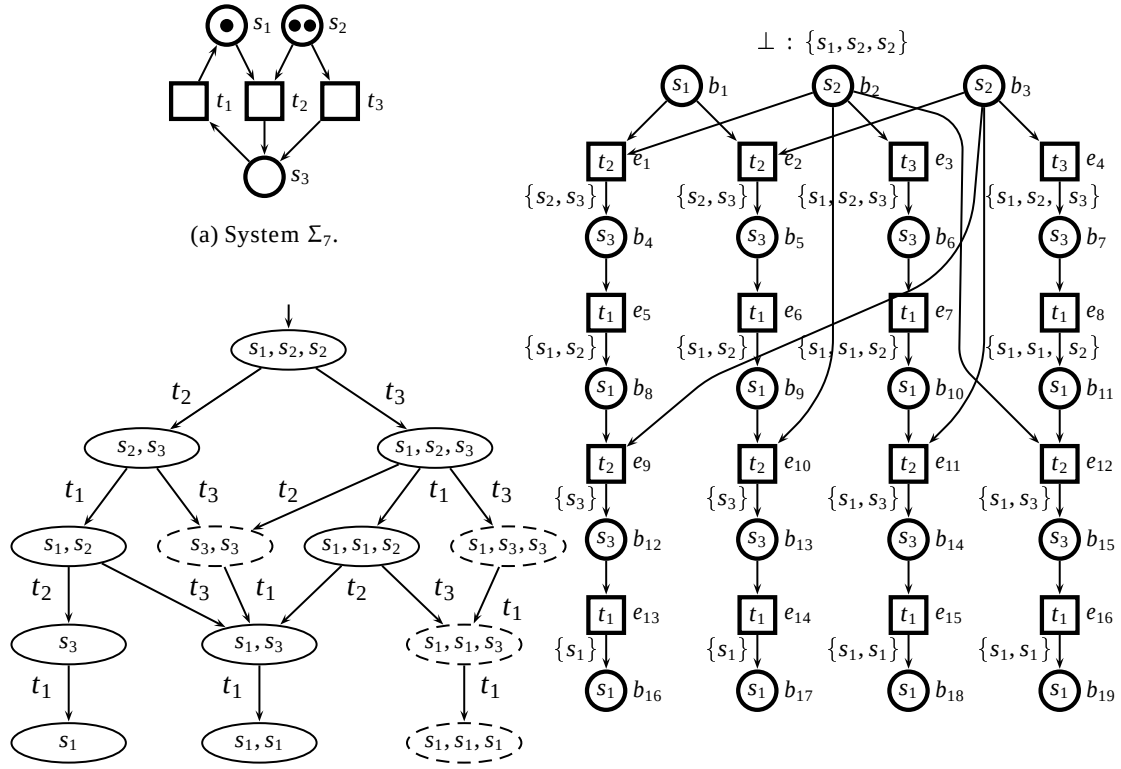
(a) $\mathsf{Cut}(C) = (\mathrm{Min}(\mathcal{O}) \cup C^\bullet) \setminus {}^\bullet C$

(b) $\mathsf{Mark}(C) = \pi(\mathsf{Cut}(C))$ ■3.3

Angewendet auf das obige Beispiel ergibt sich:

$$\mathsf{Cut}(C_x) = (\{b_1, b_2, b_3\} \cup \{b_6, b_7, b_{10}, b_{11}\}) \setminus \{b_2, b_3, b_6, b_7\} \;=\; c_x.$$

[2] Offensichtlich gilt $[e] \in \mathcal{C}_{\mathcal{O}}$, da kein Ereignis von $[e]$ in Selbstkonflikt steht.
[3] Die Funktion Mark wird in [McM92] FinalState genannt.



(a) System Σ_7.

(b) Erreichbarkeitsgraph $\mathcal{RG}_{\Sigma_7}$.

(c) Maximaler verzweigender Prozess $\mathcal{B}_m$ von Σ_7.

Abbildung 3.1: Ein System, sein Erreichbarkeitsgraph und die Entfaltung.

Für eine Konfiguration C bildet $\mathsf{Mark}(C)$ den induzierten Schnitt ab auf das Netzsystem und beschreibt dort die entsprechende erreichbare Markierung. In Abb. 3.1(c) ist unter jedem Ereignis e die Menge $\mathsf{Mark}([e])$ angegeben. Zum Vergleich ist in Teilbild (b) der Erreichbarkeitsgraph des gleichen Systems gezeigt. Die darin gestrichelt gekennzeichneten Zustände werden in der Entfaltung nicht von einer lokalen Konfiguration induziert.

Notation 3.4 *M–Äquivalenz*

Sei $\mathcal{B}$ ein verzweigender Prozess. Zwei Konfigurationen $C, C' \in \mathcal{C}_\mathcal{B}$ heißen *M–äquivalent*, geschrieben $C \equiv_M C'$, falls $\mathsf{Mark}(C) = \mathsf{Mark}(C')$ gilt. ■3.4

Beispiele für Abb. 3.1(c) sind: $[e_{15}] \equiv_M [e_{16}]$ und $\{e_4, e_4, e_7\} \equiv_M \{e_3, e_4, e_8\}$.

Definition 3.5 *Suffix eines Prozesses bezüglich einer Konfiguration*

Seien $\mathcal{B} = ((B, E, F), \pi)$ ein verzweigender Prozess und $C \in \mathcal{C}_\mathcal{B}$ eine Konfiguration. Mit $Y = \{ y \in \uparrow \mathsf{Cut}(C) \mid \forall e \in C : \neg(e \# y) \}$ definiert

$$\Uparrow(C, \mathcal{B}) = ((B|_Y, E|_Y, F|_{Y \times Y}), \pi|_Y)$$

das durch C induzierte *Suffix* in $\mathcal{B}$. ■3.5

In Abb. 3.1(c) ergibt sich daraus beispielsweise

$$\Uparrow(\{e_3, e_4, e_7, e_8\}, \mathcal{B}_m) = ((\{b_1, b_{10}, b_{11}\}, \emptyset, \emptyset), \{b_1 \mapsto s_1, b_{10} \mapsto s_1, b_{11} \mapsto s_1\})$$

und für die lokale Konfiguration $[e_5]$ liefert $\Uparrow([e_5], \mathcal{B}_m)$ den Prozess, der sich aus den Knoten $b_8, e_9, b_{12}, e_{13}, b_{16}$ und $b_3, e_4, b_7, e_8, b_{11}$ zusammensetzt.

Mit anderen Worten beschreibt das Suffix $\Uparrow(\mathcal{B}, C)$ das Netz jenseits des Schnittes $\mathsf{Cut}(C)$, d. h. die erreichbaren Knoten in dessen verzweigender *Zukunft*. Dieses Netz ist selbst wieder ein verzweigender Prozess, dessen Minimum gerade von $\mathsf{Cut}(C)$ geformt wird. Übertragen auf das zugrunde liegende Netzsystem ergibt sich der folgende Zusammenhang.

Satz 3.6 *Konfigurationen induzieren Prozesse [ERV00]*

Seien $\Sigma = (\mathcal{N}, M_0)$ ein Netzsystem und $\mathcal{B} \trianglelefteq_p \mathcal{B}_m$ ein beliebiges Präfix der Entfaltung von Σ. Für jede Konfiguration $C \in \mathcal{C}_\mathcal{B}$ gilt

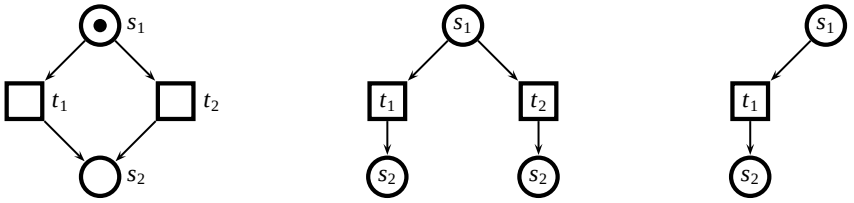
(a) $\Uparrow(C, \mathcal{B})$ ist ein verzweigender Prozess des Systems $(\mathcal{N}, \mathsf{Mark}(C))$.

(b) $\Uparrow(C, \mathcal{B}_m)$ ist isomorph zu $\mathsf{Unf}((\mathcal{N}, \mathsf{Mark}(C)))$. ■*3.6*

Dieser Isomorphismus stellt sicher, dass das Verhalten jenseits M–äquivalenter Konfigurationen äquivalent ist. Für eine komprimierte Darstellung reicht somit die einmalige Aufnahme des Verhaltens jenseits eines die Markierung M induzierenden Schnittes aus.

3.3 Vollständige endliche Präfixe

Zum Ende von Abschnitt 2.6 wurde darauf hingewiesen, dass in einer hinreichenden endlichen Approximation der Entfaltung alle erreichbaren Markierungen repräsentiert sein müssen. Damit auf dem Präfix später beliebige Eigenschaften des Systems überprüft werden können, ist diese Bedingung allein jedoch nicht ausreichend.
Ein Beispiel hierzu zeigt das Netzsystem in Abb. 3.2: die Stelle s_2 des Systems aus (a) wird markiert durch das Schalten entweder von t_1 oder t_2. Die Entfaltung (b) spiegelt



(a) Netzsystem Σ_8. (b) Entfaltung $\mathsf{Unf}(\Sigma_8)$. (c) Präfix $\mathsf{Pre}(\mathsf{Unf}(\Sigma_8))$.

Abbildung 3.2: Ein Netzsystem, seine Entfaltung und ein alle erreichbare Markierungen enthaltendes Präfix.

diesen Umstand wider. Im Präfix (c) sind alle erreichbaren Markierungen repräsentiert, nämlich $\{s_1\}$ und $\{s_2\}$. Allerdings büßt dieses Präfix gegenüber der Entfaltung an Information ein, nämlich, dass t_2 von der Startmarkierung aktiviert wird. Diese Eigenschaft kann offensichtlich auch nicht auf andere Weise aus dem Präfix (c) gewonnen werden.
Um diesen Informationsverlust zu vermeiden, muss sichergestellt werden, dass sich alle im System unter einer Markierung aktivierten Transitionen in einem Präfix wiederfinden. Auf das Prozess–Netz übertragen müssen also alle einen erreichbaren Schnitt *erweiternden* Ereignisse Berücksichtigung finden, wobei der Begriff Erweiterung wie folgt definiert wird.

Definition 3.7 *Erweiterung einer Konfiguration*

Seien $\mathcal{B} = ((B, E, F), \pi)$ ein verzweigender Prozess und $C \in \mathcal{C}_\mathcal{B}$ eine endliche Konfiguration.

(a) Eine Menge $\emptyset \neq X \subseteq E \setminus C$ von Ereignissen heißt *Erweiterung* von C, wenn $C \cup X \in \mathcal{C}_\mathcal{B}$ gilt. Ist X eine Erweiterung von C, dann wird mit $C \oplus X$ die um X *erweiterte Konfiguration* C notiert (und umgekehrt).

(b) $\mathcal{C}_\mathcal{B}^C = \{ C \oplus X \mid X \subseteq E \setminus C \}$ bezeichnet die Menge der erweiterten Konfigurationen von C bezüglich $\mathcal{B}$. ■3.7

Aus (a) folgt für zwei Konfigurationen $C, C' \in \mathcal{C}_\mathcal{B}$ unmittelbar:

$$C \subset C' \Rightarrow \exists \text{ Erweiterung } X \neq \emptyset : C \oplus X = C'$$

Wie bereits im Algorithmus 3.1 zur Berechnung der maximalen Entfaltung angedeutet (Menge *erw* in Zeile 3 und 7), wird das Prozess–Netz schrittweise über das Einfügen von erweiternden Ereignissen konstruiert.
An ein im obigen Sinne gut geartetes Präfix der Entfaltung werden zusammenfassend folgende Anforderungen gestellt.

Definition 3.8 *Vollständiger Prozess*

Seien $\Sigma = (S, T, W, M_0)$ ein Netzsystem und $\mathcal{B} = ((B, E, F), \pi)$ ein verzweigender Prozess von Σ.
$\mathcal{B}$ ist *vollständig*, falls gilt $\forall M \in \mathcal{R}^{\Sigma}_{M_0} \exists C \in \mathcal{C}_{\mathcal{O}}$ mit

(i) $\mathsf{Mark}(C) = M$ und

(ii) $\forall t \in T: M \xrightarrow{t} \Rightarrow \exists e \in E \setminus C : C \oplus \{e\}$ ist Konfiguration $\wedge\, \pi(e) = t$.

■3.8

Die Entfaltung eines Systems ist offensichtlich stets vollständig.

Das elementweise Erweitern des Präfixes muss an irgendeiner Stelle abbrechen, um ein endliches, vollständiges Anfangsstück zu erhalten. Dazu werden, wie im letzen Abschnitt hergeleitet, die lokalen Konfigurationen auf M–Äquivalenz hin überprüft – ist sie erfüllt, werden die beteiligten Konfigurationen daraufhin verglichen, welche die jeweils „größere“ ist, hinter der die weitere Berechnung abgebrochen werden kann. Für die Vergleiche wird eine Ordnung auf den Konfigurationen benötigt, die anschaulich mit der Konstruktion des Präfixes „mitwächst“: bei vorhandener M–Äquivalenz sollen später eingefügte Ereignisse bezüglich dieser Ordnung „größere“ lokale Konfigurationen garantieren als ihre Vorgänger.

Notation 3.9 *Der Isomorphismus $I^{C_2}_{C_1}$ auf Konfigurationen [Esp94, ERV96]*

Seien C_1 und C_2 mit $C_1 \equiv_M C_2$ zwei M–äquivalente Konfigurationen. Nach Satz 3.6 sind $\Uparrow(C_1, \mathcal{B})$ und $\Uparrow(C_2, \mathcal{B})$ beide isomorph zu $\mathsf{Unf}((\mathcal{N}, M))$, und folglich isomorph zueinander. Ein solcher Isomorphismus wird im Folgenden mit $I^{C_2}_{C_1}$ bezeichnet. ■3.9

Hierbei ist der Isomorphismus $I^{C_2}_{C_1}$ im Allgemeinen nur für sichere Systeme eindeutig. Mit Hilfe dieses Isomorphismus wird das Aussehen geeigneter Ordnungen für die Präfix–Generierung folgendermaßen festgelegt.

Definition 3.10 *Adäquate Ordnung*

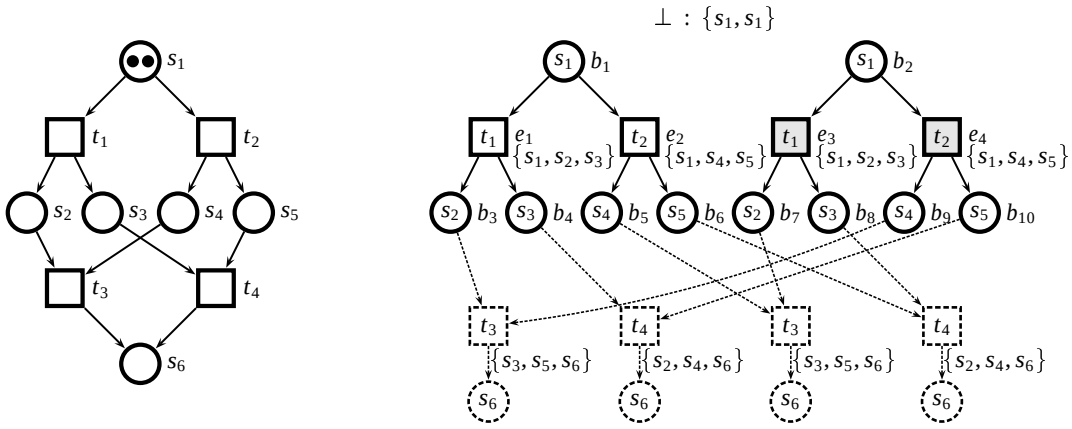
Sei $\mathcal{B}_m = ((B_m, E_m, F_m), \pi_m)$ die Entfaltung eines Netzsystems. Eine Ordnung $(\mathcal{C}_{\mathcal{B}_m}, \sqsubset)$ auf den endlichen Konfigurationen von $\mathcal{B}_m$ heißt *adäquat*, falls für beliebige $C_1, C_2 \in \mathcal{C}_{\mathcal{B}_m}$ gilt:

(i) $\sqsubset$ ist fundiert;

(ii) $C_1 \subset C_2 \Rightarrow C_1 \sqsubset C_2$;

(iii) $C_1 \sqsubset C_2 \wedge C_1 \equiv_M C_2 \Rightarrow \forall$ endl. Erw. $X \subseteq E_m \setminus C_1 : C_1 \oplus X \sqsubset C_2 \oplus I^{C_2}_{C_1}(X)$.

■3.10



(a) System Σ_9.

(b) Entfaltung von Σ_9.

Abbildung 3.3: Beispiel für die Notwendigkeit von Bedingung (ii) in Def. 3.11(a).

Im Einzelnen verlangt Bedingung (ii), dass die Ordnung $\sqsubset$ die Mengeninklusion verfeinert; (iii) fordert, dass endliche Erweiterungen von Konfigurationen die Ordnung $\sqsubset$ erhalten. Der Begriff der adäquaten Ordnung dient nun zur Formulierung des zentralen Terminierungs–Kriteriums der Präfix–Generierung.

Definition 3.11 *Cut-off–Ereignis*

Seien $\mathcal{B}_m$ die Entfaltung eines Netzsystems und $(\mathcal{C}_{\mathcal{B}_m}, \sqsubset)$ eine adäquate Ordnung. Sei weiterhin $\mathcal{B} = ((B, E, F), \pi) \trianglelefteq_p \mathcal{B}_m$ ein Präfix von $\mathcal{B}$.

(a) $e \in E$ wird *Cut-off–Ereignis* von $\mathcal{B}$ bezüglich $\sqsubset$ genannt, falls es eine lokale Konfiguration $[e'] \in \mathcal{C}_{\mathcal{B}}$ gibt mit
   (i) $[e'] \equiv_M [e]$ und
   (ii) $[e'] \sqsubset [e]$.

(b) Ist e ein Cut-off–Ereignis, so wird ein e' mit obigen Eigenschaften ein zu e *korrespondierendes Ereignis* genannt.

(c) $E_{cutoff} \subseteq E$ bezeichnet die Menge aller Cut-off–Ereignisse von $\mathcal{B}$. ■3.11

Das in Abbildung 3.3 dargestellte Netzsystem illustriert die Notwendigkeit von Bedingung (a)(ii) dieser Definition: Das System Σ ist 2–beschränkt, Teilbild (b) zeigt seine Entfaltung. Der Anteil des Netzes mit durchgezogenen Linien kennzeichnet ein Präfix, das bei Weglassen von Bedingung (ii) generiert wird. Die beiden grau unterlegten Ereignisse e_3 bzw. e_4 wurden als Cut-off–Ereignisse identifiziert, da die von ihnen induzierten Markierungen äquivalent sind mit denen von e_1 bzw. e_2. Damit sind jedoch

für die weitere Konstruktion eines vollständigen Präfixes notwendige Ereignisse ausgeschlossen, da sie als Cut-off deklariert sind.
Die an dieser Stelle korrektere Wahl von e_2 und e_3 (bzw. alternativ e_1 und e_4) zu Cut-off–Ereignissen erforderte einen Blick in die Zukunft, Kombinationen welcher Ereignisse später für eine Konstruktion noch benötigt werden.
Für 1–sichere Systeme besteht das gleiche Problem; ein entsprechendes Beispiel ist in [ERV00] enthalten.

3.4 Der Algorithmus zur Präfix–Generierung

Der folgende Algorithmus erzeugt für ein beschränktes Netzsystem ein vollständiges, endliches Präfix seiner Entfaltung. Genauer wird hiermit eine Familie von Algorithmen beschrieben, parametrisiert über die konkrete Wahl der Ordnung $\sqsubset$ in Zeile 6.

Algorithmus 3.12 *Konstruktion eines vollständigen Präfixes eines beschränkten Netzsystems*

Eingabe: Beschränktes Netzsystem $\Sigma = (\mathcal{N}, M_0)$.

Ausgabe: Endliches, vollständiges Präfix $\mathsf{Pre}(\mathsf{Unf}(\Sigma))$ der Entfaltung.

```
1   begin
2     Initialisiere Pre durch Übertragen der Startmarkierung M_0;
3     erw := Menge der möglichen Erweiterungen von Pre;
4     cut-off := ∅;
5     while erw ≠ ∅ do
6       Wähle aus erw ein Ereignis e mit [e] bzgl. ⊏ minimal;
7       if [e] ∩ cut-off = ∅ then
8         Füge e zu Pre hinzu, mitsamt allen Bedingungen,
9           die dem Nachbereich π(e)• in Σ entsprechen;
10        erw := Menge der möglichen Erweiterungen von Pre;
11        if e ist Cut-off–Ereignis von Fin then
12          cut-off := cut-off ∪ {e}
13        endif
14      else
15        erw := erw \ {e}
16      endif
17    endwhile;
18    return Pre
19  end                                                    ■3.12
```

Das Testen der Cut-off–Bedingung in Zeile 11 erfolgt mit Hilfe von Def. 3.11(a); dazu wird überprüft, ob das Präfix ein Ereignis enthält, dessen Konfiguration zu derjenigen von e M–äquivalent ist.

Satz 3.13 *Endlichkeit und Vollständigkeit des generierten Präfixes [ERV96]*

Seien $\Sigma = (S, T, W, M_0)$ ein beschränktes Netzsystem und $\sqsubset$ eine adäquate Ordnung. Das von Algorithmus 3.12 erzeugte Präfix $\mathcal{B} = \mathsf{Pre}(\mathsf{Unf}(\Sigma)) = ((B, E, F), \pi)$ der Entfaltung von Σ ist (a) endlich und (b) vollständig.

Beweis: (a) Endlichkeit von $\mathcal{B}$:

(i) Die Vor– und Nachbereiche von Ereignissen sind endlich: $\forall e \in E : |{}^\bullet e| \in \mathbb{N} \ \wedge \ |e^\bullet| \in \mathbb{N}$. Diese Tatsache ergibt sich unmittelbar aus der Endlichkeit von S, in Verbindung mit der Bijektion zwischen ${}^\bullet e$ und ${}^\bullet\pi^{-1}(e)$ bzw. $e^\bullet$ und $\pi^{-1}(e)^\bullet$ (Definition 2.20).

(ii) Die Tiefe eines Ereignisses $e \in E$ lässt sich abschätzen mit Hilfe der Anzahl erreichbarer Zustände: es gilt $d(e) \leq |\mathcal{R}^\Sigma_{M_0}| + 1$.
Da B–Schnitte durch $\mathcal{B}$ erreichbaren Markierungen entsprechen, muss jede Folge von Ereignissen $e_1 \prec e_2 \prec \ldots \prec e_{|\mathcal{R}^\Sigma_{M_0}|+1}$ zwei Ereignisse $e_i \prec e_j$ enthalten, deren Konfigurationen die gleiche Markierung induzieren, also $e_i \equiv_M e_j$. Wegen $e_i \prec e_j$ gilt $[e_i] \subset [e_j]$, und damit $[e_i] \sqsubset [e_j]$. Somit wird in diesem Fall e_j als Cut-off–Ereignis erkannt und die Konstruktion des Präfixes hinter e_j nicht fortgesetzt.

(iii) Für $k \in \mathbb{N}$ bezeichne $E_k = \{e \in E \mid d(e) \leq k\}$ die Menge der Ereignisse mit Tiefe maximal k. Dann ist die Menge E_k endlich.
Beweis durch Induktion über k. Induktionsanfang: $E_0 = \emptyset$ ist endlich. Induktionsschluss: Sei E_k endlich, damit ist wegen (i) ebenfalls $E_k^\bullet$ endlich. Nach der Konstruktion des Präfixes gilt ${}^\bullet E_{k+1} \subseteq E_k^\bullet \cup \mathrm{Min}(\mathcal{B})$; die Menge ${}^\bullet E_{k+1}$ ist also endlich. Nach Definition 2.20(iv) werden Transitionen im Prozess–Netz nicht dupliziert, somit ist auch E_{k+1} selbst endlich.

Insgesamt besteht $\mathcal{B}$ wegen (i) aus endlich vielen Bedingungen und wegen (ii), (iii) aus endlich vielen Ereignissen.

(b) Vollständigkeit von $\mathcal{B}$: die Eigenschaften von Def. 3.8 sind nachzuweisen.

(i) Jede erreichbare Markierung von Σ ist in $\mathcal{B}$ repräsentiert, formal $\forall M \in \mathcal{R}^\Sigma_{M_0} \ \exists C \in \mathcal{C}_\mathcal{B}$ mit $\mathsf{Mark}(C) = M$.
Es bezeichne $\mathsf{Unf}(\Sigma) = ((B_m, E_m, F_m), \pi_m)$ die Entfaltung von Σ. Sei $M \in \mathcal{R}^\Sigma_{M_0}$, d. h. $\exists C_1 \in \mathcal{C}_{\mathcal{B}_m} : \mathsf{Mark}(C_1) = M$. Falls $C_1 \notin \mathcal{C}_\mathcal{B}$, dann enthält diese Konfiguration ein Ereignis $e_1 \in E_{cutoff}$ und eine Erweiterung $X_1 \subseteq E_m$ mit $C_1 = [e_1] \oplus X_1$. Nach Def. 3.11(a) (Cut-off–Ereignis) folgt: $\exists [e'] \in \mathcal{C}_\mathcal{B} : [e'] \equiv_M [e]$ und $[e'] \sqsubset [e]$.
Betrachte die Konfiguration $C_2 = [e'] \oplus I^{C_2}_{C_1}(X_1)$. Da die Ordnung $\sqsubset$ von endlichen Erweiterungen erhalten wird, folgt $C_2 \sqsubset C_1$, außerdem gilt $\mathsf{Mark}(C_2) = M$. Falls $C_2 \notin \mathcal{C}_\mathcal{B}$ ist, kann dieses Verfahren wiederholt werden für eine Konfiguration $C_3 = [e_3] \oplus I^{C_3}_{C_2}(X_2)$ mit $C_3 \sqsubset C_2$

und $\mathsf{Mark}(C_3) = M$ u. s. w. Wegen der Fundiertheit von $\sqsubset$ kann diese Iteration nur endlich oft durchgeführt werden und terminiert somit in einer Konfiguration $C \in \mathcal{C}_\mathcal{B}$.

(ii) $\forall M \in \mathcal{R}^\Sigma_{M_0} \exists C \in \mathcal{C}_\mathcal{B}$ mit $\forall t \in T: \; M \xrightarrow{t} \; \Rightarrow \quad \exists e \in E \setminus C : C \oplus \{e\} \wedge \pi(e) = t$.
Wenn eine Transition t in Σ aktiviert wird, dann durch eine erreichbare Markierung M; nach (i) ist diese in $\mathcal{B}$ repräsentiert. Sei nun $C \in \mathcal{C}_\mathcal{B}$ eine Konfiguration mit $\mathsf{Mark}(C) = M$, überdies sei C bzgl. $\sqsubset$ minimal. Falls C ein Cut-off–Ereignis enthält, dann lässt sich mit einem iterativen Verfahren wie in (i) eine Konfiguration $C' \sqsubset C$ ermitteln, für die ebenfalls $\mathsf{Mark}(C') = M$ gilt, im Widerspruch zur Minimalität von C. Somit kann C kein Cut-off–Ereignis enthalten, demnach muss es eine Konfiguration $C \oplus \{e\}$ geben, für deren erweiterndes Ereignis $\pi^{-1}(e) = t$ gilt.

■3.13

McMillan hat in [McM92] die Ordnung $\sqsubset$ über die Größe der lokalen Konfigurationen festgelegt.

Definition 3.14 *Die Ordnung $\sqsubset_{McM}$*

Sei $\mathcal{B}$ die Entfaltung eines Netzsystems, und seien $C_1, C_2 \in \mathcal{C}_\mathcal{B}$ Konfigurationen. Definiere die Ordung $(\mathcal{C}_\mathcal{B}, \sqsubset_{McM})$ mit

$$C_1 \sqsubset_{McM} C_2 \;\text{ gdw. }\; |C_1| < |C_2|.$$

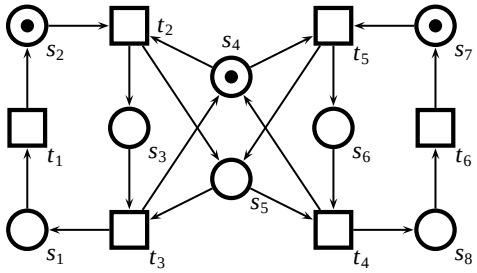
■3.14

Satz 3.15

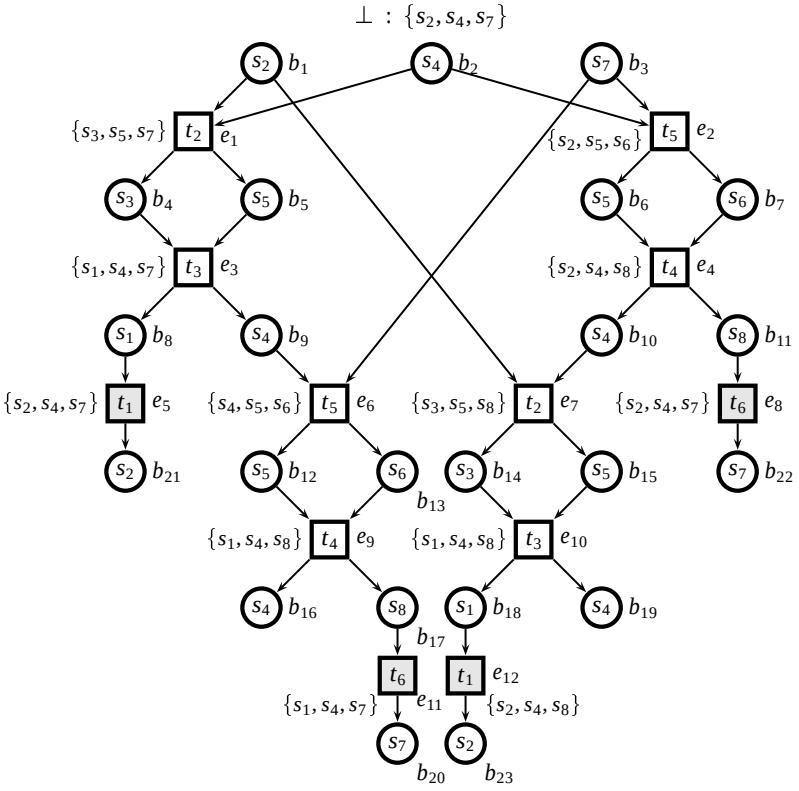
Die Ordnung $\sqsubset_{McM}$ ist adäquat.

Beweis: Die Eigenschaften von Def. 3.10 sind leicht nachzuprüfen: (i) $\sqsubset_{McM}$ ist fundiert, da $<$ fundiert ist. (ii) Inklusion: seien $C_1, C_2 \in \mathcal{C}_\mathcal{B}$; aus $C_1 \subset C_2$ folgt direkt $|C_1| < |C_2|$. (iii) Erhalt endlicher Erweiterungen: seien $C_1 \sqsubset_{McM} C_2$ und X Erweiterung von C_1; da $I^{C_2}_{C_1}$ bijektiv ist, gilt $|X| = |I^{C_2}_{C_1}(X)|$, und damit folgt $|C_1| < |C_2| \Rightarrow |C_1 \oplus X| < |C_1 \oplus I^{C_2}_{C_1}(X)|$. ■3.15

In Abbildung 3.4 ist ein lebendiges Netzsystem angegeben, eine einfache Lösung des Problems des gegenseitigen Ausschlusses. Teilbild (b) zeigt das mit Algorithmus 3.12 und der Ordnung $\sqsubset_{McM}$ konstruierte endliche Präfix seiner Entfaltung. Die Indizes der Knotennamen geben die Reihenfolge des Einfügens in das Präfix an. Der Zustandsraum

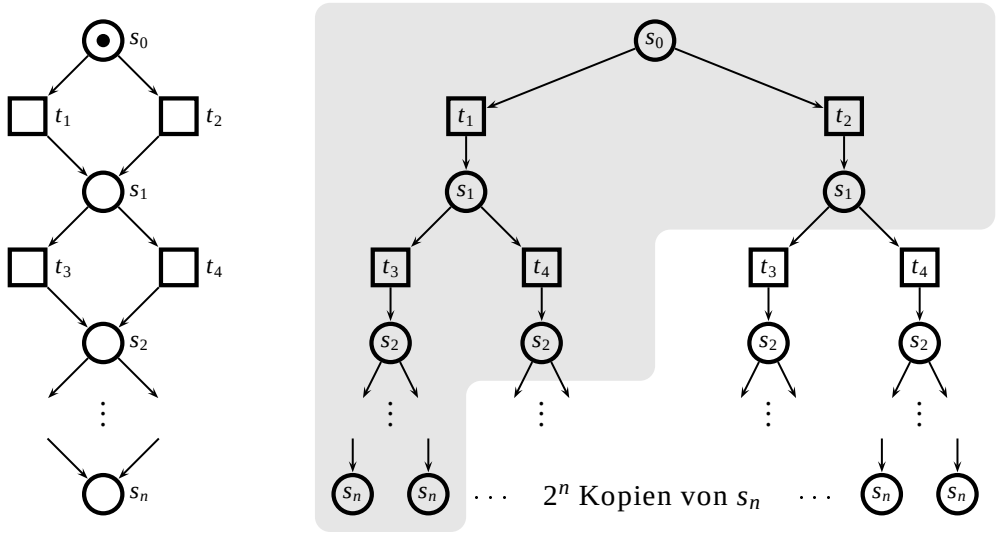


(a) System Σ_{10}.



(b) McMillan–Präfix von Σ_{10}.

Abbildung 3.4: Beispiel für eine Präfix–Generierung nach Algorithmus 3.12.



(a) Netzsystem Σ_{11}.

(b) Entfaltung $\mathcal{B}_m$ von Σ_{11}.

Abbildung 3.5: Beispiel einer Entfaltung, die exponentiell größer ist als $|\mathcal{R}^{\Sigma}_{M_0}|$.

von Σ_{10} umfasst 14 erreichbare Markierungen; das Präfix besteht aus 12 Ereignissen, von denen vier Cut-off–Ereignisse sind: e_5, e_8, e_{11} und e_{12}.

Unglücklicherweise führt gerade die Bedingung (ii) der Definition 3.11(a) (Cut-off–Ereignis) bei der Ordnung $\sqsubset_{McM}$ zu einem unangenehmen Seiteneffekt. Bei genauer Betrachtung des letzten Beispiels (Entfaltung des Systems Σ_{10} in Abb. 3.4) fällt auf, dass die Ereignisse e_9 und e_{10} beide die gleiche Markierung $\{s_1, s_4, s_8\}$ induzieren, jedoch keines von beiden als Cut–off–Ereignis erkannt wird.
Wie bereits an anderer Stelle erwähnt, kann der Algorithmus von McMillan aufgrund dieser Eigenschaft in einigen Fällen ein unverhältnismäßig großes Präfix generieren. Ein extremes Beispiel[4] hierfür ist in Abb. 3.5 angegeben, ein einfaches, parametrisiertes Netz. Mit der Ordnung $\sqsubset_{McM}$ wird das in Teilbild (b) dargestellte Präfix generiert – dieses fällt mit der maximalen Entfaltung zusammen und hat exponentielle Größe im Vergleich zum Ausgangssystem. Darüber hinaus ist es sogar exponentiell größer als der Erreichbarkeitsgraph, der genau $n + 1$ Zustände umfasst.
Der Grund für diese „Explosion“ liegt in der ungünstigen Wahl des Cut-off–Kriteriums: da lediglich die Größen der Konfigurationen verglichen werden, kann im vorliegenden Beispiel kein Cut-off–Ereignis ermittelt werden – alle jeweils M–äquivalenten Konfigurationen besitzen gleiche Kardinalität.

[4] Beispiel gefunden von A. Kishinevsky und A. Taubin

Der grau unterlegte Bereich in Abb. 3.5(b) kennzeichnet ein mögliches vollständiges Präfix der Entfaltung von Σ_{11}. Dieses besitzt wie Σ_{11} und $\mathcal{R}_{M_0}^{\Sigma_{11}}$ lediglich lineare Größe in Ahhängigkeit von n.
Im nächsten Abschnitt wird beschrieben, wie sich durch Verfeinern der Ordnung $\sqsubset_{McM}$ Verbesserungen in der Präfix–Generierung erreichen lassen, die diesen Effekt vermeiden.

3.5 Verfeinerte adäquate Ordnungen

Die wichtigste Eigenschaft für die Präfix–Generierung ist Adäquatheit der verwendeten Ordnung auf Konfigurationen. Somit erfüllt Algorithmus 3.12 mit der Ordnung $\sqsubset_{McM}$. die gestellte Aufgabe. Für praktische Anwendungen, insbesondere im Rahmen der Verifikation verteilter Systeme, ist neben der Vollständigkeit des Präfixes ein anderes Kriterium von Bedeutung: seine Größe. Eine wünschenswerte Eigenschaft ist damit die Minimalität des erzeugten Netzes. In den folgenden Unterabschnitten werden einige Ordnungen für zunehmend speziellere Systemklassen betrachtet, die die Größe des Präfixes beeinflussen. Abschließend erfolgt eine Diskussion der Minimalität.

3.5.1 Eine adäquate Ordnung für *n*–beschränkte Netzsysteme

Offensichtlich reicht ein Vergleich der Kardinalitäten lokaler Konfigurationen allein nicht aus, sie fein genug voneinander zu unterscheiden und damit ein unnötiges Aufblähen der Präfixgröße zu verhindern. Gesucht ist also ein Kriterium, Konfigurationen gleicher Größe aufgrund ihrer Struktur in vernünftiger Weise ordnen zu können. Eine naheliegende Möglichkeit bietet das Betrachten der Beschriftungen der Ereignisse von Konfigurationen.

Definition 3.16 *Linearisierung*

Sei $(X, \preccurlyeq)$ eine Halbordnung und sei $Y \in \mathcal{M}_{\mathcal{F}}(X)$ eine endliche Multimenge von Elementen aus X. Die *Linearisierung* von Y, $\mathsf{Lin}_{\preccurlyeq}(Y) = y_1 y_2 \ldots y_{|Y|}$, ist eine Permutation der Elemente von Y mit der Eigenschaft

$$\forall 1 \leq i, j \leq |Y| : i < j \Rightarrow y_i \preccurlyeq y_j. \qquad \blacksquare 3.16$$

Anschaulich entspricht die Linearisierung einer Sortierung bezüglich $\preccurlyeq$; ist diese Ordnung darüber hinaus total, dann existiert zu jeder Menge eine eindeutige Linearisierung; für nicht–totale Ordnungen muss die Linearisierung dagegen nicht eindeutig sein, da untereinander nicht vergleichbare Elemente in keiner festen Reihenfolge angeordnet werden können.
Bezogen auf Prozess–Netze liefert die Linearisierung $\mathsf{Lin}_{\preceq}([e])$ einer lokalen Konfiguration eine mögliche Schaltsequenz zum Eintritt von e. In Abb. 3.1(c) gilt z. B. $\mathsf{Lin}_{\preceq}([e_{14}]) =$

$e_2e_6e_{10}e_{14}$. Wird die Linearisierung einer beliebigen Konfiguration abgebildet auf das ursprüngliche System Σ, $\pi(\mathsf{Lin}_{\preceq}(C))$, so ergibt sich die entsprechende Schaltsequenz in Σ, also auf Modellierungsebene. Im obigen Beispiel ist somit $\pi(\mathsf{Lin}_{\preceq}([e_{14}])) = t_2t_1t_2t_1$[5]. In netzbasierten Werkzeugen werden Linearisierungen dazu benutzt, Fehlerpfade auszugeben. Diese geben Aufschluss darüber, welche Schaltfolgen z. B. zu einer spezifikationsverletzenden Situation geführt haben, sie können damit unmittelbar zur Fehlerlokalisierung und –beseitigung eingesetzt werden. In Netz–Simulatoren können Linearisierungen zur Visualisierung von Schaltsequenzen genutzt werden.

Notation 3.17 *Geordnete Transitionenfolge*

Sei $((B, E, F), \pi)$ die Entfaltung eines beschränkten Netzsystems (S, T, W, M_0), sei weiterhin $(T, \ll)$ eine totale Ordnung auf den Transitionen. Für eine Menge $E' \subseteq E$ von Ereignissen definiere

$$\phi(E') = \mathsf{Lin}_{\ll}(\pi(E'))$$

■3.17

In den Netzsystemen dieser Arbeit wird $(T, \ll)$ von der natürlichen Ordnung der Indizes der Transitionsnamen abgeleitet, also $t_i \ll t_j$ gdw. $i < j$ für $t_i, t_j \in T$.
Wegen der Totalität von $\ll$ ist die Abbildung ϕ eindeutig, Transitionen können allerdings mehrfach vorkommen. Beispielsweise gilt für die Entfaltung aus Abb. 3.1(c) auf Seite 34: $\phi([e_{13}]) = \mathsf{Lin}_{\ll}(t_2t_1t_2t_1) = t_1t_1t_2t_2$.

Für den Vergleich endlicher Sequenzen bieten sich lexikographische Ordnungen an. Diese lassen sich allgemein derart unterschieden, ob die Länge der Sequenz in den Vergleich einbezogen wird oder nicht.

Definition 3.18 *Lexikographische Ordnungen*

Seien $(A, \ll)$ eine Totalordnung über einem endlichen Alphabet A und $v, w \in A^*$ zwei Wörter über A.

(a) Die lexikographische Ordnung $(A^*, <_{lex_\ll})$ ist definiert durch

$$v <_{lex_\ll} w \text{ gdw. } \exists i \in \mathbb{N}^+ : (\forall 1 \leq j < i : v_j = w_j) \wedge v_i \ll w_i$$

(b) Die Silex[6]–Ordnung $(A^*, <_{silex_\ll})$ ist definiert durch

$$v <_{silex_\ll} w \text{ gdw. } |v| < |w| \vee (|v| = |w| \wedge v <_{lex_\ll} w)$$

■3.18

[5] Das virtuelle Ereignis $\perp$ wird von π abgebildet auf des leere Wort ϵ.

[6] kurz für *size–lexicographic order*.

Mit der gewöhnlichen alphabetischen Ordnung $a \ll b \ll c$ ist z. B. $abc <_{lex_\ll} ac$, aber $ac <_{silex_\ll} abc$. Hervorzuheben ist, dass von diesen beiden Ordnungen lediglich $<_{silex_\ll}$ fundiert ist, $<_{lex_\ll}$ hingegen nicht.
Mit Hilfe geordneter Transitionsfolgen wird nun eine weitere Ordnung auf Konfigurationen für Algorithmus 3.12 definiert.

Definition 3.19 *Die Ordnung $\sqsubset_b$*

Seien $\Sigma = (S, T, W, M_0)$ ein beschränktes Netzsystem und $(T, \ll)$ eine beliebige totale Ordnung auf den Transitionen. Sei $\mathcal{B}_m$ die Entfaltung von Σ, seien weiterhin $C_1, C_2 \in \mathcal{C}_{\mathcal{B}_m}$ zwei Konfigurationen. Definiere

$$C_1 \sqsubset_b C_2 \quad \text{gdw.} \quad \begin{array}{ll} & |C_1| < |C_2| \\ \vee & |C_1| = |C_2| \wedge \phi(C_1) <_{lex_\ll} \phi(C_2). \end{array}$$

■3.19

Offensichtlich ist $\sqsubset_b$ eine Verfeinerung von $\sqsubset_{McM}$; die Adäquatheit von $\sqsubset_b$ folgt unmittelbar aus den darin verwendeten Ordnungen, siehe auch [ERV00].
Die Ordnung $\sqsubset_b$ hätte äquivalent auch direkt über die Silex–Ordnung ausgedrückt werden können: $C_1 \sqsubset_b C_2 \Leftrightarrow \phi(C_1) <_{silex_\ll} \phi(C_2)$. Die Gründe, dies nicht zu tun, sind zweierlei: neben der unmittelbareren Darstellung der Verfeinerung von $\sqsubset_{McM}$ erlaubt die Definition über $<_{lex}$ eine effizientere Implementation. Die Größe einer Konfiguration $[e]$ lässt sich in linearer Zeit, $O(|[e]|)$, während der Präfix–Konstruktion ermitteln, die Berechnung von ϕ ist – insbesondere bei wachsendem Präfix – eine vergleichsweise teure Operation, da alle Elemente der zu vergleichenden lokalen Konfigurationen betrachtet werden müssen. Deshalb gilt es, diese aufwändigeren Vergleiche soweit wie möglich zu vermeiden.

Garantiert nun die Ordnung $\sqsubset_b$ die Vergleichbarkeit aller Konfigurationen, um einer Degeneration von Präfixen entgegenzuwirken? Das Netzsystem in Abb. 3.4 zeigt, dass dies im Allgemeinen immer noch nicht der Fall ist: die lokalen Konfigurationen $[e_9] = \{e_1, e_3, e_6, e_9\}$ und $[e_{10}] = \{e_2, e_4, e_7, e_{10}\}$ sind M–äquivalent ($M = \{s_1, s_4, s_8\}$). Es sind $|[e_9]| = |[e_{10}]|$ und

$$\phi([e_9]) = \mathsf{Lin}_\ll(\{t_2, t_3, t_5, t_4\}) = t_2 t_3 t_4 t_5$$

$$\phi([e_{10}]) = \mathsf{Lin}_\ll(\{t_5, t_4, t_2, t_3\}) = t_2 t_3 t_4 t_5$$

Die Konfigurationen $[e_9]$ und $[e_{10}]$ sind somit bezüglich $\sqsubset_b$ nicht vergleichbar, e_{10} kann nicht als Cut–off–Ereignis erkannt werden.

3.5.2 Eine totale adäquate Ordnung für sichere Netzsysteme

Wie könnte nun die Ordnung $\sqsubset_b$ noch weiter verfeinert werden? Das letzte Beispiel vermittelt eine intuitive Idee: anstatt die Transitionsfolge $\pi(C)$ einer Konfiguration C zu sortieren, besteht auch die Möglichkeit, die Ereignisse von C gemäß der Halbordnung des Netzes aufzuzählen, und anschließend die Sequenz ihrer Beschriftungen zu betrachten, also $\pi(\mathsf{Lin}_{\prec}(C))$. Im Beispiel führt dies zum Vergleich der Sequenzen $t_2t_3t_5t_4$ und $t_5t_4t_2t_3$, der mit einer geeigneten Ordnung zu entscheiden ist.
Unglücklicherweise sind diese Sequenzen im Allgemeinen nicht eindeutig, da $\prec$ nicht total ist. Damit sind sie für Vergleichsoperationen in dieser Form ungeeignet. Die Linearisierung kann jedoch eindeutig gemacht werden, beispielsweise durch separates Sortieren der Beschriftungen ungeordneter, also nebenläufiger Ereignisse. Der folgende Algorithmus ermittelt die Anteile ungeordneter Ereignisse schichtweise und konkateniert sie anschließend entsprechend der Ordnung $\prec$ aufsteigend. In der Literatur ist diese Darstellung unter dem Namen Foata–Normalform bekannt [Die90].

Algorithmus 3.20 *Foata–Normalform einer Konfiguration*

Eingabe: Konfiguration $C \in \mathcal{C}_{\mathcal{B}}$ eines verzweigenden Prozesses $\mathcal{B}$.
Ausgabe: Foata–Normalform $\mathsf{FNF}(C)$.

```
1   begin
2     FNF := ∅;
3     while C ≠ ∅ do
4       FNF := FNF · Min(C);
5       C := C \ Min(C)
6     endwhile;
7     return FNF
8   end
```
■3.20

Damit kann die Ordnung $\sqsubset_b$ auf folgende Weise verfeinert werden:

Definition 3.21 *Die Ordnung $\sqsubset_t$*

Sei $\Sigma = (S, T, W, M_0)$ ein sicheres Netzsystem und sei $(T, \ll)$ eine beliebige totale Ordnung auf den Transitionen. Sei $\mathcal{B}_m = ((B, E, F), \pi)$ die Entfaltung von Σ und seien $C_1, C_2 \in \mathcal{C}_{\mathcal{B}_m}$ Konfigurationen. Die Ordnung $(\mathcal{B}_m, \sqsubset_t)$ ist definiert durch

$$\begin{aligned} C_1 \sqsubset_t C_2 \quad \text{gdw.} \quad & |C_1| < |C_2| \\ \vee \quad & |C_1| = |C_2| \wedge \phi(C_1) <_{lex_{\ll}} \phi(C_2) \\ \vee \quad & \phi(C_1) = \phi(C_2) \wedge \mathsf{FNF}(C_1) <_{FNF\text{-}lex_{\ll}} \mathsf{FNF}(C_2) \end{aligned}$$

mit

$$E_1 \cdot E_2 \cdot \ldots \cdot E_m <_{FNF\text{-}lex_{\ll}} E'_1 \cdot E'_2 \cdot \ldots \cdot E'_n$$
$$\text{gdw. } \exists 1 \leq i \leq m : (\forall 1 \leq j < i : \phi(E_j) = \phi(E'_j)) \ \wedge \ \phi(E_i) <_{lex_{\ll}} \phi(E'_i)$$

wobei $E_i, E'_j \subseteq E; 1 \leq i \leq m; 1 \leq j \leq n$ für $m, n \in \mathbb{N}^+$. ■3.21

Die Ordnung $\sqsubset_t$ ist jedoch nur adäquat für 1–sichere Netzsysteme. Beim Nachweis des Erhalts der Ordnung bei endlichen Erweiterungen (Punkt (iii) von Def. 3.10) wird auf die Eigenschaft sicherer Systeme zurückgegriffen, dass für beliebige Bedingungen $b_1, b_2 \in B$ mit $b_1 \mathsf{co}\, b_2$ stets $\pi(b_1) \neq \pi(b_2)$ gilt. Der ausführliche Beweis ist in [ERV00] enthalten.
Eine naheliegende Frage ist, ob die zweite Alternative in Definition 3.21 verzichtbar ist, also ob vielleicht auch

$$C_1 \sqsubset'_t C_2 \quad \text{gdw.} \quad |C_1| < |C_2| \vee |C_1| = |C_2| \wedge \phi(\mathsf{FNF}(C_1)) <_{lex_{\ll}} \phi(\mathsf{FNF}(C_2))$$

eine adäquate Ordnung beschreibt. Das System in Abbildung 3.6 begründet allerdings die Notwendigkeit dieser Bedingung für die Adäquatheit von $\sqsubset_t$; sie wird auch unmittelbar für deren Beweis benötigt.
Um das einzusehen, werden beispielhaft die Konfigurationen $C_1 = \{e_1, e_2\}$ und $C_2 = \{e_3, e_4\}$ im dargestellten Prozess betrachtet. Offensichtlich gilt $|C_1| = |C_2|$ und $C_1 \equiv_M C_2$, außerdem ist $\mathsf{FNF}(C_1) = \{e_1, e_2\} <_{FNF\text{-}lex_{\ll}} \{e_3\} \cdot \{e_3\} = \mathsf{FNF}(C_2)$ infolge des Vergleichs der Minima: $t_3 t_4 <_{lex_{\ll}} t_5$. Konfiguration C_1 ist durch e_5 mit $\pi(e_5) = t_1$ erweiterbar, es folgt

$$\begin{aligned} \mathsf{FNF}(C_1 \oplus \{e_5\}) = \{e_1, e_2\} \cdot \{e_5\} >_{FNF\text{-}lex_{\ll}} \{e_3, e_6\} \cdot \{e_4\} &= \mathsf{FNF}(C_2 \oplus \{e_6\}) \\ &= \mathsf{FNF}(C_2 \oplus \{I_{C_1}^{C_2}(e_5)\}) \end{aligned}$$

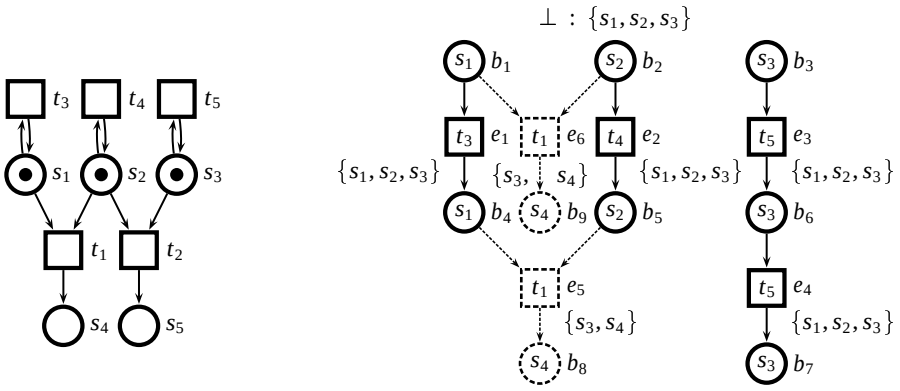
wegen

$$\phi(\{e_1, e_2\}) = t_3 t_4 >_{lex_{\ll}} t_1 t_5 = \phi(\{e_3, e_6\}).$$

Die Ordnung $\sqsubset'_t$ wird also nicht von allen endlichen Erweiterungen erhalten. Bei Hinzunahme der zweiten Alternative von Def. 3.21 wird dieser Fall abgefangen aufgrund von

$$\phi(C_1) = t_1 t_3 t_4 <_{lex_{\ll}} t_1 t_5 t_5 = \phi(C_2).$$

Ein etwas größeres Beispiel für die Präfix–Generierung mit Hilfe der Ordnung $\sqsubset_t$ ist in Abbildung 3.7 auf Seite 50 zusammengestellt. Gezeigt ist eine Modellierung der *Cyclic Scheduler* nach R. Milner [Mil80] in Form kommunizierender Automaten, siehe Teilbild (a). Dabei sind im allgemeinen Fall n Scheduler ringförmig angeordnet (hier wird der konkrete Fall $n = 3$ betrachtet), so dass jeweils Scheduler i und $(i+1) \bmod n$, $0 \leq i < n$, miteinander kommunizieren. Jedem der n Scheduler ist ein separater

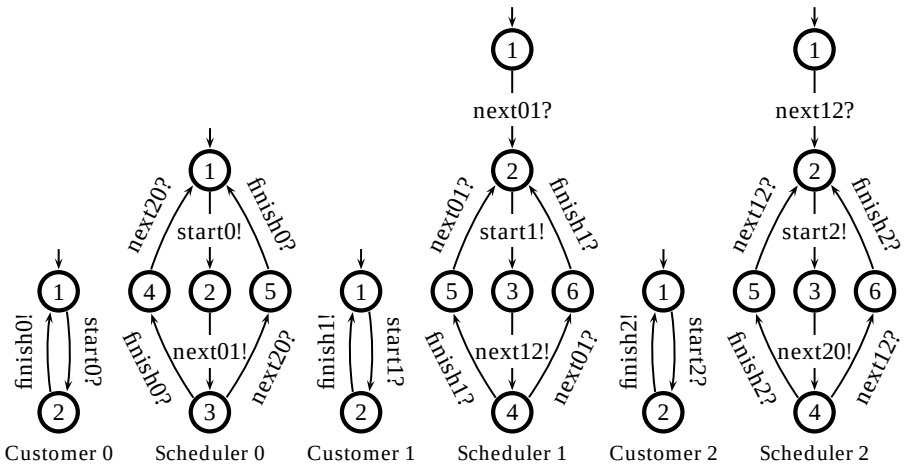


(a) System Σ_{12}. (b) Ein Prozess von Σ_{12}.

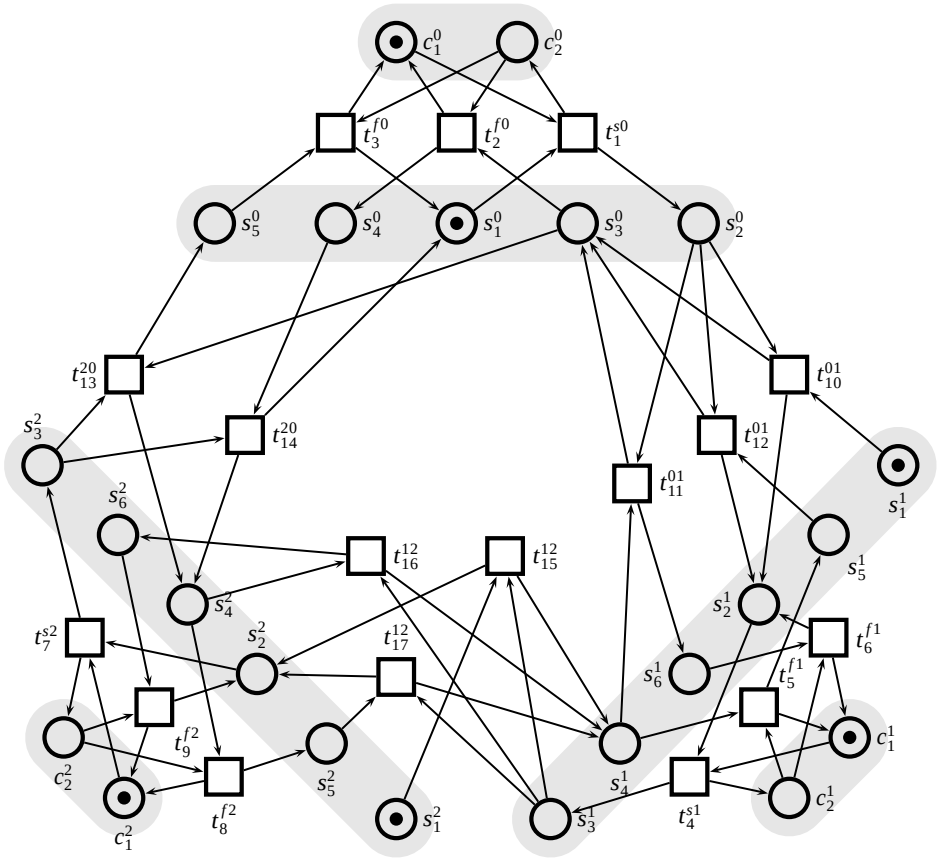
Abbildung 3.6: Beispiel für die Notwendigkeit der 2. Disjunktion bei der Ordnung $\sqsubset_t$.

Customer–Prozess vorgeschaltet. Kommunikation ist in den Beschriftungen der Zustandsübergänge angedeutet durch die aus CSP [Hoa78] entlehnte Notation ch! und ch?: auf einen Kommunikationskanal ch kann schreibend (!) bzw. lesend (?) zugegriffen werden; durch die Synchronisation (Verschmelzung) zweier jeweils komplementärer Kommunikationsbeschriftungen wird der Nachrichtenaustausch zwischen den einzelnen Automaten modelliert.

In Teilbild (b) ist das der Automatendarstellung entsprechende Petri–Netz gezeigt, das bereits vollständig synchronisiert ist. Aus Platzgründen sind die Beschriftungen der Knoten des Netzes abgekürzt worden: in den Stellenknoten steht *s* für *Scheduler* und *c* für *Customer*; die hochgestellten Indizes geben die Nummern des jeweiligen Schedulers bzw. Customers wieder, die tiefgestellten Indizes entsprechenden den jeweiligen Zustandsnummern in der Automatendarstellung. Die Transitionen sind alle mit *t* bezeichnet, deren tiefgestellte Indizes lediglich eine fortlaufende Aufzählung angeben. Die hochgestellten Bezeichner stehen abkürzend für die jeweiligen Kommunikationsaktionen: „f*j*“ steht für „finish*j*“, „s*j*“ für „start*j*“, sowie „*jk*“ für „next*jk*“, jeweils für $0 \leq j, k \leq 2$.
Diese Beschriftungen werden auch im mit der Ordnung $\sqsubset_t$ konstruierten Präfix verwendet, das in Abbildung 3.8 dargestellt ist. Das Präfix besteht aus insgesamt 52 Bedingungen und 23 Ereignissen; das entsprechende mit der Ordnung $\sqsubset_{McM}$ erzeugte Präfix enthält 94 Bedingungen und 44 Ereignisse und ist damit fast doppelt so groß. Aus Gründen der Unübersichtlichkeit (die Knoten eines Netzes dieser Größe lassen sich kaum mehr vernünftig auf einer Seite anordnen) wird das zweite Präfix hier nicht dargestellt.



(a) Cyclic Scheduler (der Größe 3), modelliert als kommunizierende Automaten.



(b) System Σ_{12}: vollständig synchronisierte Petri–Netz–Darstellung von (a).

Abbildung 3.7: Cyclic Scheduler nach Milner [Mil80].

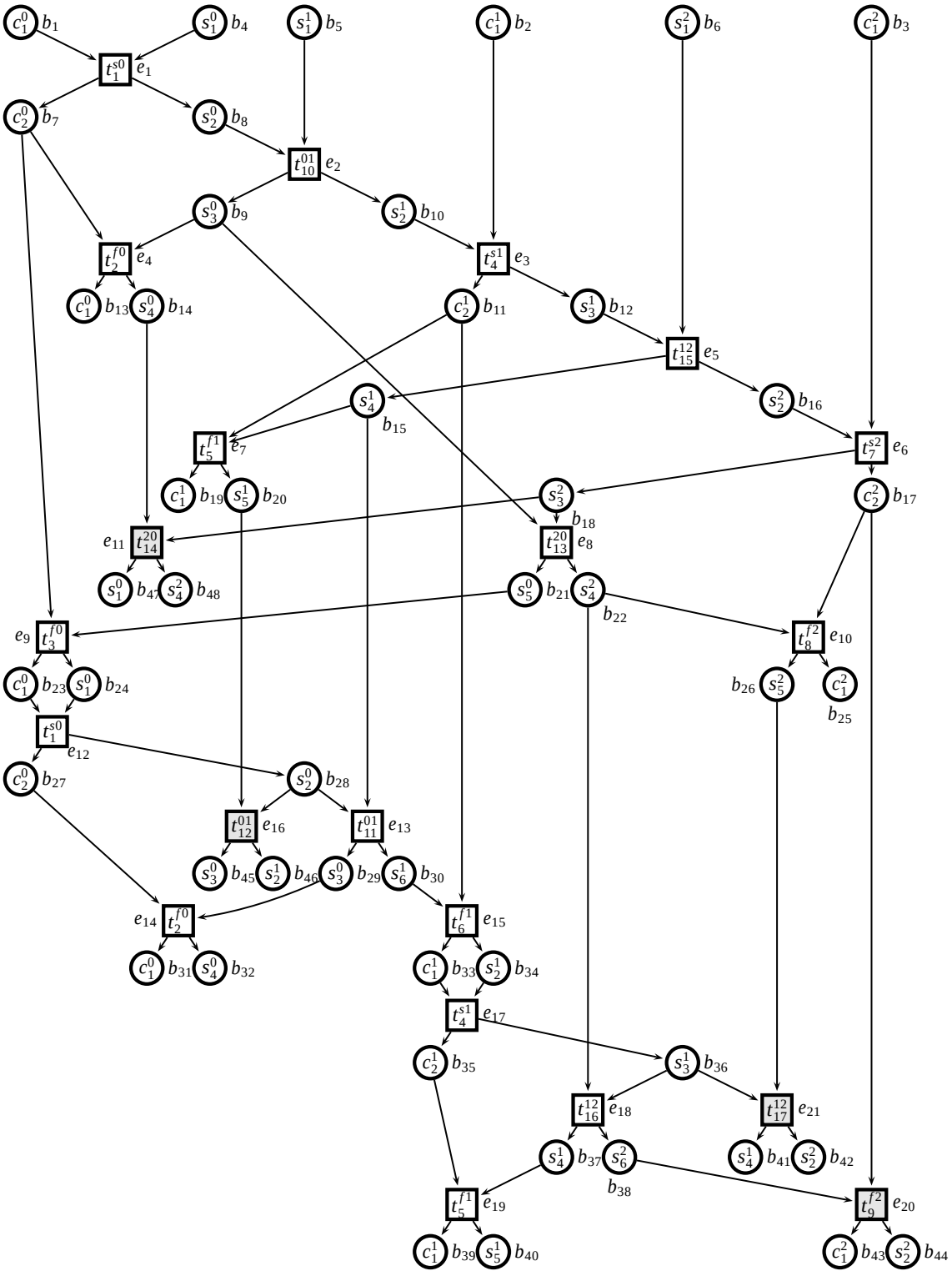


Abbildung 3.8: Präfix von Σ_{13} aus Abb. 3.7(b), mit Ordnung $\sqsubset_t$ generiert.

Mit Hilfe der 1–Sicherheit lässt sich noch eine andere Eigenschaft nachweisen: die Ordnung $\sqsubset_t$ ist total [ERV96]. Daraus ist die folgende Größenabschätzung bezüglich des generierten Präfixes herleitbar.

Satz 3.22 *Abschätzung der Präfixgröße bei totaler Ordnung $\sqsubset$ [ERV96]*

Sei $Pre(Fin(\Sigma)) = ((B, E, F), \pi)$ das von Algorithmus 3.12 mit der totalen adäquaten Ordnung $(\mathcal{C}_{Fin}, \sqsubset)$ generierte Präfix eines Systems Σ. Dann gilt $|E \setminus E_{cutoff}| \leq |\mathcal{R}^{\Sigma}_{M_0}|$.

Beweis: Da die Ordnung $\sqsubset$ als total vorausgesetzt wird, sind alle Konfigurationen untereinander vergleichbar. Die Konstruktion von Algorithmus 3.12 garantiert (Zeile 6), dass Ereignisse entlang dieser Ordnung in das Präfix aufgenommen werden. Für jedes neu einzufügende Ereignis e mit $\exists e' : [e'] \equiv_M [e]$ folgt unmittelbar $[e'] \sqsubset [e]$. Damit wird e als Cut-off–Ereignis identifiziert.

Jede erreichbare Markierung $M \in \mathcal{R}^{\Sigma}_{M_0}$ kann somit nur maximal einmal von einem Nicht–Cut-off–Ereignis induziert werden, alle weiteren Vorkommen von M führen zu Cut-off–Ereignissen. Folglich ist $|E \setminus E_{cutoff}| \leq |\mathcal{R}^{\Sigma}_{M_0}|$. ■3.22

Es ist nicht bekannt, ob für jede Klasse von Netzen eine totale adäquate Ordnung existiert. Für den praktischen Einsatz sind 1–sichere Netzsysteme jedoch eine der relevantesten Netzklassen, so dass die meisten der auf Netzentfaltungen basierenden Verifikationstechniken inzwischen diese Netzklasse voraussetzen. In der in Kapitel 8 beschriebenen Implementation des McMillan–Algorithmus werden ebenfalls nur sichere Netze unterstützt. Diese Einschränkung erlaubt eine Reihe von Vereinfachungen, die unmittelbare Effizienzsteigerungen gegenüber einer Realisierung auf Basis beliebig beschränkter Systeme nach sich ziehen.

Dennoch bleibt festzuhalten, dass die Reduktion auf 1–sichere Systeme eine echte Einschränkung gegenüber dem Original–Algorithmus von McMillan darstellt. Aus diesem Grund haben verschiedene Autoren die totale Ordnung $\sqsubset_t$ auf n–beschränkte Systeme übertragen [ERV00, Haa98]. Beide benutzen zur Darstellung der Halbordnungssemantik die gegen Ende von Kapitel 2.4 erwähnten *Executions*.
Der in [ERV00] beschriebene Ansatz einer totalen Ordnung für beschränkte Systeme ist ebenfalls im Kontext der in Kapitel 8 diskutierten Datenstrukturen und Algorithmen implementiert worden; experimentelle Ergebnisse von Tests auf Verklemmungsfreiheit beschränkter Systeme sind in [Mel98], Kapitel 3.6 enthalten.

In der Literatur gibt es mittlerweile weitere Abwandlungen der Ordnungen $\sqsubset_b$ und $\sqsubset_t$, z. B. [HNW98, Hel99a]. Das Ziel solcher alternativen Ordnungen ist nicht unbedingt Minimalität des generierten Präfixes; diese wird manchmal sogar bewusst aufgegeben, um andere wünschenswerte Eigenschaften des Präfixes zu erzielen.

3.5.3 Totale adäquate Ordnungen für SND–Systeme

Die in diesem Abschnitt untersuchte Ordnung ist ursprünglich in [ER99] vorgestellt worden für ein einfaches Modell kommunizierender Automaten, sogenannte synchrone Produkte von Transitionssystemen [Arn94]. Dieses Modell ist auf Petri–Netze unmittelbar übertragbar, dort ist es unter dem Namen *SND–Systeme* bekannt: *State Net Decomposable Systems*, in Zustandsnetze zerlegbare Netzsysteme [BF87].

Definition 3.23 *Zustandsnetz, SND–Netz*

Sei $\mathcal{N} = (S, T, F)$ ein Netz mit einfachen Kantengewichten.

(a) $\mathcal{N}$ heißt *Zustandsnetz* (oder *S–Graph*),falls gilt: $\forall t \in T : |{}^\bullet t| = 1 = |t^\bullet|$.

(b) $\mathcal{N}$ heißt *SND–Netz* wenn es $n \in \mathbb{N}^+$ Zustandsnetze $\mathcal{N}_i = (S_i, T_i, F_i)$, $1 \leq i \leq n$ gibt mit

$$S = \bigcup_{i=1}^{n} S_i\,, \qquad T = \bigcup_{i=1}^{n} T_i\,, \qquad F = \bigcup_{i=1}^{n} F_i\,,$$

und der Disjunktheits–Bedingung $\forall 1 \leq i, j \leq n, i \neq j : S_i \cap S_j = \emptyset$.

■3.23

Infolge der Disjunktheit der Stellenmengen der Zustandsmengen sind auch deren Flussrelationen untereinander disjunkt. Anschaulich bestehen SND–Netze aus n „Komponenten“, die jeweils aus Zustandsnetzen gebildet werden; gleichbenannte Transitionen der einzelnen Komponenten synchronisieren und verschmelzen auf diese Weise miteinander[7]. In SND–Netzen sind somit Stellenknoten eindeutig einem einzelnen Zustandsnetz zuzuordnen[8], Transitionen hingegen können an mehreren Zustandsnetzen beteiligt sein; wegen Def. 3.23(a) gilt $|{}^\bullet t| = |t^\bullet|$ für alle $t \in T$, wobei im Vor– bzw. Nachbereich einer Transition t mit $|{}^\bullet t| > 1$ je genau eine Stelle aus jeder an der Synchronisation von t beteiligten Komponente enthalten ist. Als Spezialfall ist auch jedes Zustandsnetz ein SND–Netz. In Abbildung 3.9 sind zwei (markierte) SND–Netze gezeigt; die einzelnen Zustandsnetze sind durch unterschiedliche Grauwerte hervorgehoben.

Definition 3.24 *SND–System*

Sei $\mathcal{N} = (S, T, F)$ ein SND–Netz, das aus n Zustandsnetzen $\mathcal{N}_i = (S_i, T_i, F_i)$, $1 \leq i \leq n$, besteht.

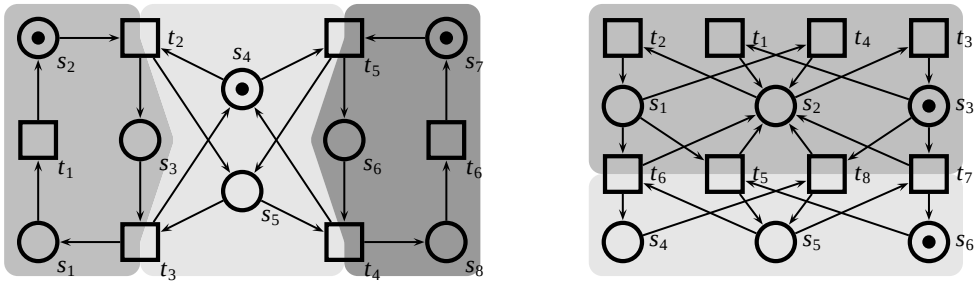
(a) Eine Markierung M von $\mathcal{N}$ heißt *regulär*, falls $\forall 1 \leq i \leq n : \sum_{s \in S_i} M(s) = 1$ gilt, d. h. jedes Zustandsnetz genau eine Marke besitzt.

(b) $\Sigma = (\mathcal{N}, M_0)$ heißt *SND–System*, falls M_0 reguläre Markierung von $\mathcal{N}$ ist.

■3.24

[7] Diese Verschmelzung wird ganauso durchgeführt wie im Beispiel der *Cyclic Scheduler* in Abb. 3.7.

[8] Umgekehrt muss die Zerlegung eines SND–Netzes in S–Graphen nicht eindeutig sein, jedoch ist es, mit einer lebendigen Markierung M versehen, stets in die gleiche Zahl $\sum_{s \in S} M(s)$ von S–Graphen zerlegbar.



(a) Σ_{14}: Gegenseitiger Ausschluss.

(b) Σ_{15}: FIFO–Puffer der Kapazität 2.

Abbildung 3.9: Beispiele für SND–Systeme.

Satz 3.25 *Regularität erreichbarer Markierungen in SND–Systemen*

Sei $\Sigma = (S, T, F, M_0)$ ein SND–System, das aus n Zustandsnetzen $\mathcal{N}_i = (S_i, T_i, F_i)$, $1 \leq i \leq n$ besteht. Dann gilt $\forall M \in \mathcal{R}^{\Sigma}_{M_0}$: M ist regulär.

Beweis: Sei $M' \in \mathcal{R}^{\Sigma}_{M_0}$ eine beliebige reguläre Markierung, und sei $M' \xrightarrow{t} M$ ein möglicher Zustandsübergang.

Dann gilt für alle Zustandsnetze $\mathcal{N}_i$ mit $t \in S_i$, dass $\exists s' \in {}^\bullet t \cap S_i : s' \in M'$; s' ist nach Def. 3.23 eindeutig bestimmt und wegen der Regularität von M' ist keine andere der Stellen $S_i \setminus \{s'\}$ markiert. Nach dem Schalten von t wird die Marke auf die eindeutige Stelle $s \in t^\bullet \cap S_i$ gelegt, es verbleibt also genau eine Marke in dieser Komponente.

Innerhalb der Zustandsnetze $\mathcal{N}_i$ mit $t \notin S_i$ verändert das Schalten von t die aktuelle Markenbelegung nicht. Somit ist M insgesamt regulär. ■3.25

Als unmittelbare Folge dieses Satzes sind SND–Systeme damit stets sicher. Bezogen auf einen Prozess von Σ folgt daraus, dass jeder B–Schnitt genau eine Bedingung aus jeder der n Komponenten enthält. Diese Schnitte sind somit darstellbar als n–stellige Vektoren.

Innerhalb einer Komponente existieren wegen Def. 3.23(a), zusammen mit der 1–Sicherheit von SND–Systemen, keine nebenläufig aktivierten Transitionen. Übertragen auf einen verzweigenden Prozess eines SND–Systems können daher zwei Ereignisse, die der gleichen Komponente angehören, nicht in co–Relation stehen. Damit gilt der folgende Zusammenhang (Beweis durch Induktion über die Struktur des Prozess–Netzes).

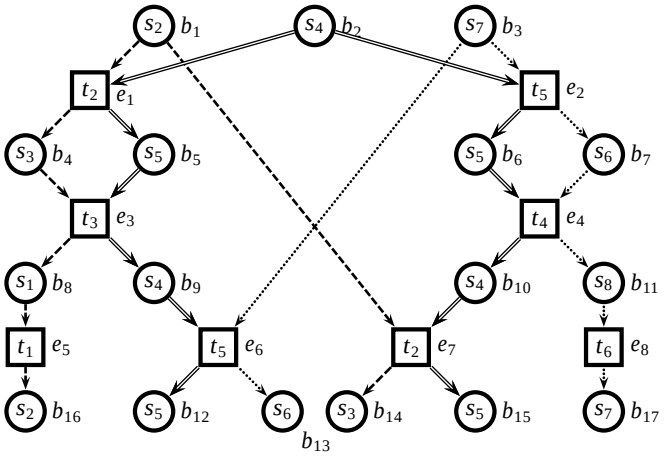


Abbildung 3.10: Verzweigender Prozess von Σ_{14}.

Satz 3.26 *Vergleichbarkeit von i–Ereignissen*

Sei $(\,(B, E, F'),\ \pi\,)$ ein verzweigender Prozess eines aus n Zustandsnetzen $(\,S_i, T_i, F_i\,)$, $1 \leq i \leq n$, bestehenden SND–Systems. Dann gilt für zwei Ereignisse $e_1, e_2 \in E \cap \pi^{-1}(T_i)$ einer Komponente i: $e_1 \preceq e_2 \ \vee\ e_2 \preceq e_1 \ \vee\ e_1 \# e_2$. ■*3.26*

In Abbildung 3.10 ist ein verzweigender Prozess des SND–Systems Σ_{14} angegeben. Der obige Sachverhalt wird in dieser Graphik durch die unterschiedliche Schraffur der Kanten hervorgehoben, welche die Zugehörigkeit zu den drei Zustandsnetzen kennzeichnet. Über b_1 stehen beispielsweise e_1 und e_7 in der ersten Komponente in Konflikt; e_1, e_3 und e_6 sind kausal abhängig in der zweiten Komponente.
Wie schon im letzten Abschnitt bei der totalen Ordnung $\sqsubset_t$ wird nachfolgend eine Abbildung definiert, die für eine Menge von Ereignissen die (über den Homomorphismus π) entsprechende Menge von Transitionen ermittelt und sortiert. Für SND–Systeme wird diese Abbildung zunächst lokal für jedes Zustandsnetz *i* vorgenommen und dann komponentenweise zusammengefasst.

Definition 3.27 *i–Sicht auf Ereignisse*

Sei $\Sigma = (\,S, T, F, M_0\,)$ ein SND–System bestehend aus *n* Zustandsnetzen $\mathcal{N}_i = (\,S_i, T_i, F_i\,), 1 \leq i \leq n$, und sei $\mathcal{O} = (\,(B, E, F'),\ \pi\,)$ ein Prozess-Netz von Σ.

Für eine Ereignis–Menge $E' \subseteq E$ und $1 \leq i \leq n$ ist die *i–Sicht* auf E' definiert durch

$$V_i(E') = \pi(\mathsf{Lin}_{\prec}(E' \cap \pi^{-1}(T_i)))$$

Das *n*–Tupel $\mathbf{V}(E') = (V_1(E'), \ldots, V_n(E'))$ vereint die lokalen Sichten auf E'.

■3.27

Die i–Sicht projiziert Ereignisse, gemäß der Halbordnung des Prozess–Netzes sortiert, auf die Transitionen der i–ten Komponente. Für eine Konfiguration $C \in \mathcal{C}_{\mathcal{O}}$ beschreibt $V_i(C)$ also gerade die entsprechende Schaltfolge in Zustandsnetz i, die zu dieser Konfiguration beigetragen hat. Die Linearisierung ist eindeutig, da unter den Ereignissen einer lokalen Sicht gemäß Def. 3.26 keine nebenläufig aktivierten Ereignisse enthalten sein können.
Bezogen auf das Prozess–Netz in Abb. 3.10 ergeben z. B. für die lokale Konfiguration $[e_6] = \{e_1, e_3, e_6\}$ die folgenden i–Sichten: $V_1([e_6]) = t_2t_3t_5$, $V_2([e_6]) = t_2t_3t_5$ und $V_3([e_6]) = t_5$, zusammengefasst also $\mathbf{V}([e_6]) = (t_2t_3t_5, t_2t_3t_5, t_5)$. Ein anderes Beispiel ist $\mathbf{V}([e_8]) = (\epsilon, t_5t_4, t_5t_4t_6)$.

Die folgende Eigenschaft der Abbildung $\mathbf{V}$ ist die Kernbeobachtung, welche später die Definition einer einfachen totalen adäquaten Ordnung auf Präfixen für SND–Systeme erlauben wird.

Satz 3.28 *Injektivität von* $\mathbf{V}$ *[ER99]*

Sei $(\,(B, E, F'),\ \pi\,)$ *ein verzweigender Prozess eines aus n Zustandsnetzen* $(\,S_i,\ T_i,\ F_i\,)$, $1 \leq i \leq n$, *bestehenden SND–Systems. Für Konfigurationen* $C_1, C_2 \in \mathcal{C}_{\mathcal{B}}$ *gilt*

$$\mathbf{V}(C_1) = \mathbf{V}(C_2) \Rightarrow C_1 = C_2$$

Beweis: Seien $C_1, C_2 \in \mathcal{C}_{\mathcal{B}}$ mit $\mathbf{V}(C_1) = \mathbf{V}(C_2)$. Es wird $C_1 = C_2$ gezeigt über $C_1 = C = C_2$ mit $C = C_1 \cap C_2$. Wegen der Symmetrie der Gleichheit wird hier nur die Richtung $C_1 = C$ bewiesen, und zwar durch Widerspruch:
Angenommen, $C \subset C_1$, d. h. $\exists e_1 \in C_1 \setminus C : C \oplus \{e_1\}$. Dann muss es für die Transition $t = \pi(e_1)$ eine Komponente i geben mit $t \in T_i$. Nach Def. 3.27 ist (wegen der Ordnung $\prec$) $V_i(C) \cdot t$ ein Präfix der Sicht $V_i(C_1)$, und damit, wegen $\mathbf{V}(C_1) = \mathbf{V}(C_2)$, auch Präfix von $V_i(C_2)$. Also $\exists e_2 \in C_2 : C \oplus \{e_2\} \wedge \pi(e_2) = t$. Für die Ereignisse e_1, e_2 gilt offensichtlich $\pi({}^\bullet e_1) = {}^\bullet t = \pi({}^\bullet e_2)$, daneben sind ${}^\bullet e_1$ und ${}^\bullet e_2$ co–Mengen; wegen $C \oplus \{e_1\} \wedge C \oplus \{e_2\}$ ist außerdem ${}^\bullet e_1 \cup {}^\bullet e_2$ co–Menge. Da in SND–Prozessen in jeder co–Menge maximal eine Bedingung aus jeder Komponente enthalten sein kann, muss ${}^\bullet e_1 = {}^\bullet e_2$ gelten, damit folgt $e_1 = e_2$, also $e_1 \in C_2$, in Widerspruch zur anfänglichen Wahl $e_1 \in C_1 \setminus C = C_1 \setminus C_2$.
■3.28

Damit ist jede Konfiguration C eindeutig charakterisiert durch das Tupel seiner lokalen Sichten $\mathbf{V}(C)$. Im Folgenden wird eine Ordnung $\Subset$ auf Sichten hergeleitet, die zusammen mit Algorithmus 3.12 für die Präfix–Generierung von SND–Systemen benutzt wird.
Der Wertebereich der Abbildung $\mathbf{V}$ sind n–Tupel von Transitionsfolgen über den einzelnen Netzkomponenten, kurz T–Vektoren über $\mathbf{T}^* = T_1^* \times \ldots \times T_n^*$. Auf dieser Menge werden lexikographische Ordnungen in Anlehnung an Def. 3.18 wie folgt definiert.

Definition 3.29 *Lexikographische Ordnungen auf* $\mathbf{T}^*$

Seien $(T, \ll)$ eine Totalordnung über den Transitionen eines SND–Systems $\Sigma = (S, T, F, M_0)$ und $\mathbf{v}, \mathbf{w} \in \mathbf{T}^*$ zwei T–Vektoren.

(a) Die Silex–Ordnung $(\mathbf{T}^*, \Subset_{silex_1})$ ist definiert durch

$$\mathbf{v} \Subset_{silex_1} \mathbf{w} \text{ gdw. } \exists 1 \leq i \leq n : (\forall 1 \leq j < i : \mathbf{v}_j = \mathbf{w}_j) \wedge \mathbf{v}_i <_{silex_\ll} \mathbf{w}_i$$

(b) Die Silex–Ordnung $(\mathbf{T}^*, \Subset_{silex_2})$ ist definiert durch

$$\begin{aligned} \mathbf{v} \Subset_{silex_2} \mathbf{w} \quad \text{gdw.} \quad & (\exists 1 \leq i \leq n : (\forall 1 \leq j < i : |\mathbf{v}_j| = |\mathbf{w}_j|) \wedge |\mathbf{v}_i| < |\mathbf{w}_i|) \\ \vee \quad & ((\forall 1 \leq i \leq n : |\mathbf{v}_i| = |\mathbf{w}_i|) \\ & \wedge \exists 1 \leq i \leq n : (\forall 1 \leq j \leq i : \mathbf{v}_j = \mathbf{w}_j) \wedge \mathbf{v}_i <_{lex_\ll} \mathbf{w}_i) \end{aligned}$$

■3.29

Die beiden Ordnungen unterscheiden sich darin, zu welchem Zeitpunkt die explizite Untersuchung von Transitionssequenzen vorgenommen wird. Ordnung $\Subset_{silex_1}$ benötigt diese zur Feststellung der komponentenweisen Gleichheit. Vergleiche bezüglich der Ordnung $\Subset_{silex_2}$ betrachten zunächst nur die Kardinalitäten, also numerische Werte. Stimmen die Kardinalitäten für alle Komponenten überein, dann werden auch die Transitionssequenzen untersucht.

Ein kleines Beispiel soll die Anwendung dieser Ordnungen verdeutlichen. Dazu werden die beiden fünfstelligen T–Vektoren $\mathbf{v} = (t_2t_3, t_1t_{10}, t_2, t_1, t_3t_{10})$ und $\mathbf{w} = (t_2t_4, t_1t_{10}, t_2, t_1, t_{10}t_4)$ betrachtet[9]. Für den Vergleich von $\mathbf{v}$ und $\mathbf{w}$ bezüglich $\Subset_{silex_1}$ werden zunächst die Längen der Sequenzen der jeweils ersten Komponente verglichen; diese Größen betragen für beide Vektoren zwei, der anschließende direkte Vergleich der ersten Transitionssequenzen ergibt $t_2t_3 <_{lex_\ll} t_2t_4$, damit ist $\mathbf{v} \Subset_{silex_1} \mathbf{w}$.
Beim Vergleich von $\mathbf{v}$ und $\mathbf{w}$ bezüglich der Ordnung $\Subset_{silex_2}$ werden zunächst lediglich die Längen der Sequenzen komponentenweise verglichen. Im vorliegenden Beispiel sind diese stets gleich, der Längenvektor ergibt sich zu $(2, 2, 1, 1, 2)$. Damit muss die Entscheidung wiederum durch direkten Sequenz–Vergleich gefällt werden, wie oben liefert bereits der Vergleich in der ersten Komponente, dass $\mathbf{v}_1 <_{lex_\ll} \mathbf{w}_1$ ist , somit gilt ebenfalls $\mathbf{v} \Subset_{silex_2} \mathbf{w}$. In diesem Beispiel führt der Vergleich für beide Ordnungen zufällig zum gleichen Ergebnis.

Über die eben definierten Ordnungen auf T–Vektoren lassen sich unmittelbar Ordnungen auf den Konfigurationen der Entfaltung eines SND–Systems bestimmen.

[9] Diese Vektoren sind dem Präfix aus Abb. 3.12 auf Seite 59 entnommen, dort gilt $\mathbf{v} = \mathbf{V}([e_{10}])$ und $\mathbf{w} = \mathbf{V}([e_{11}])$.

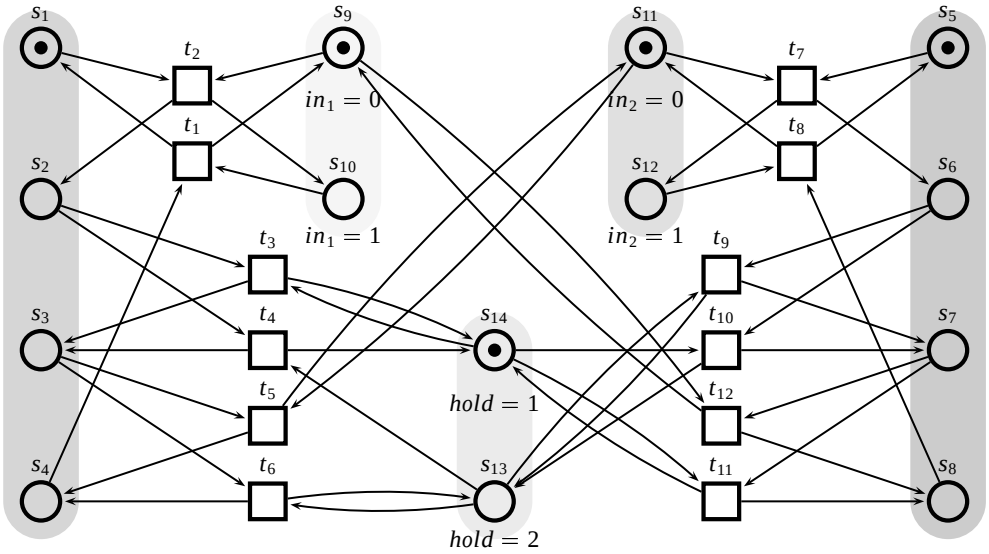


Abbildung 3.11: Σ_{16}: SND–System der Mutex–Lösung nach Peterson.

Definition 3.30 *Die totalen adäquaten Ordnungen $\sqsubset_{SND_1}$ und $\sqsubset_{SND_2}$*

Sei $\Sigma = (S, T, F, M_0)$ ein SND–System und sei $(T, \ll)$ eine totale Ordnung auf den Transitionen. Seien weiterhin $\mathcal{B}_m = \mathsf{Unf}(\Sigma)$ und $C_1, C_2 \in \mathcal{C}_\mathcal{B}$ zwei Konfigurationen. Die Ordnungen $(\mathcal{C}_\mathcal{B}, \sqsubset_{SND_1})$ und $(\mathcal{C}_\mathcal{B}, \sqsubset_{SND_2})$ sind definiert als

(a) $C_1 \sqsubset_{SND_1} C_2$ gdw. $\mathbf{V}(C_1) \Subset_{silex_1} \mathbf{V}(C_2)$

(b) $C_1 \sqsubset_{SND_2} C_2$ gdw. $\mathbf{V}(C_1) \Subset_{silex_2} \mathbf{V}(C_2)$

■3.30

Adäquatheit und Totalität der Ordnungen $\sqsubset_{SND_1}$ und $\sqsubset_{SND_2}$ werden in [ER99] nachgewiesen. Ihr Beweis basiert auf den Eigenschaften $C_1 \subseteq C_2 \Rightarrow \mathbf{V}(C_1) \Subset \mathbf{V}(C_2)$ und $\mathbf{V}(C \oplus E) = \mathbf{V}(C) \cdot \mathbf{V}(E)$, wobei „$\cdot$“ die komponentenweise Konkatenation von T–Vektoren kennzeichnet.

Auf den ersten Blick scheinen die SND–Ordnungen noch keine wirklichen Vorteile zu bieten gegenüber der auf sicheren Netzen ebenfalls totalen Ordnung $\sqsubset_t$, solange für die Berechnung der Abbildung $\mathbf{V}$ weiterhin die komplette lokale Konfiguration eines Ereignisses berechnet werden muss.
Die Struktur von SND–Systemen erlaubt jedoch eine weniger aufwändige Ermittlung von B–Schnitten, diese können lokal zusammengesetzt werden aus den Schnitten der unmittelbaren Vorgängerknoten. Für die formale Definition dieser Operation wird als nächstes die komponentenbezogene Tiefe von Bedingungen eingeführt.

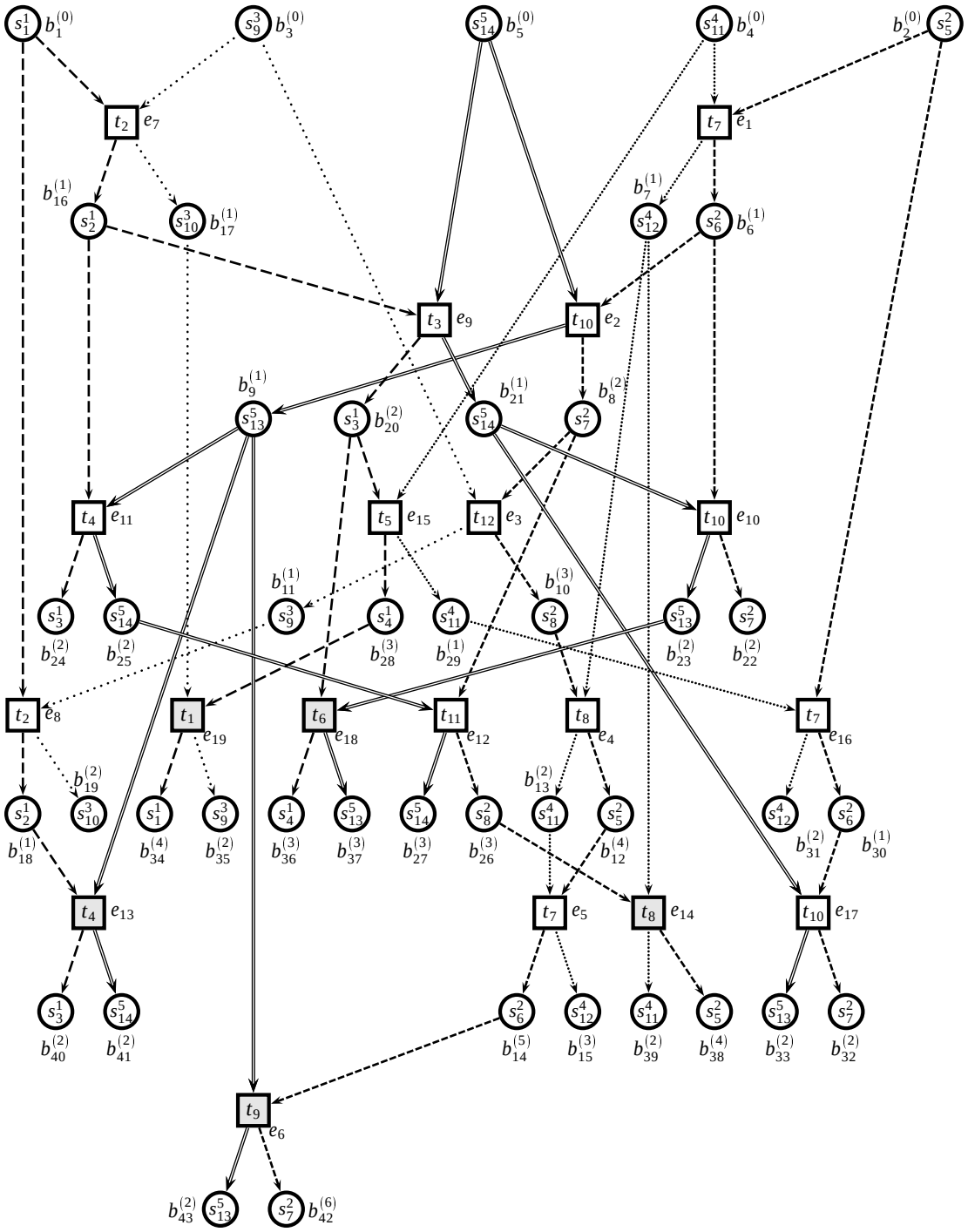


Abbildung 3.12: Präfix des Systems Σ_{16} aus Abb. 3.11, mit Ordnung $\sqsubset_{SND_1}$ generiert.

Definition 3.31 *i–Tiefe von Bedingungen*

Sei $\mathcal{B} = ((B, E, F'), \pi)$ ein verzweigender Prozess eines aus n Zustandsnetzen $\mathcal{N}_i = (S_i, T_i, F_i)$, $1 \leq i \leq n$, bestehenden SND–Systems. Die *i–Tiefe* einer Bedingung $b_i \in B \cap \pi^{-1}(S_i)$ der i–ten Netzkomponente wird rekursiv definiert durch

$$d_i(b_i) = \begin{cases} 0 & \text{falls } b_i \in \text{Min}(\mathcal{B}), \\ d_i(b_i') + 1 & \text{sonst, wobei } b_i' \in {}^{\bullet\bullet}b_i \cap \pi^{-1}(S_i) \end{cases} \qquad \blacksquare 3.31$$

Globale Tiefe d und i–Tiefe stimmen im Allgemeinen nicht überein, jedoch gilt stets $d_i(b) \leq d(b)$. Abbildungen 3.11 und 3.12 zeigen ein SND–System, das das Problem des wechselseitigen Ausschlusses zweier nebenläufiger Prozesse in der Lösung von Peterson [Pet81] modelliert, sowie ein Präfix der Entfaltung, welches mit der Ordnung $\sqsubset_{SND_1}$ konstruiert wurde. Die Indizes der Bedingungen und Ereignisse spiegeln die Reihenfolge ihres Einfügens in das Präfix wider. Die *i*–Tiefen der Bedingungen sind in Klammern hochgestellt an jede Bedingung geheftet. Die unterschiedlichen Linien drücken die Zugehörigkeit der Kanten (und damit der Knoten) zu den fünf Zustandsnetzen dieses Systems aus.

Der folgende Zusammenhang zwischen den Bedingungen einzelner, von Konfigurationen induzierter Schnitte und den Elementen des von der Vereinigung der Konfigurationen induzierten Schnittes gestattet später eine lokale Ermittlung der Abbildung Cut, und damit auch von Mark.

Lemma 3.32 *Zusammensetzung von B–Schnitten [ER99]*

Sei $\mathcal{B}$ ein verzweigender Prozess eines aus n Zustandsnetzen bestehenden SND–Systems. Seien $C^1, \ldots, C^k, C \in \mathcal{C}_\mathcal{B}$ $(k \in \mathbb{N}^+)$ Konfigurationen mit $\bigcup_{j=1}^{k} C^j = C$. Abkürzend beschreibe $\mathbf{c}^j = Cut(C^j)$ sowie $\mathbf{c} = Cut(C)$. Dann gilt komponentenweise für $1 \leq i \leq n$: $\mathbf{c}_i$ ist die eindeutige Bedingung $b_i \in \{\mathbf{c}_i^1, \ldots, \mathbf{c}_i^k\}$[10] mit maximaler i–Tiefe $d_i(b_i)$.

$\blacksquare$*3.32*

Die Elemente der Menge $\{\mathbf{c}_i^1, \ldots, \mathbf{c}_i^k\}$ können total geordnet werden, weil sie sowohl der gleichen Komponente i angehören (Halbordnung), als auch im Nachbereich der Ereignisse der Konfiguration C liegen (Konfliktfreiheit); zudem ist ihr Vorkommen in einer Menge einmalig. Damit sind die Bedingungen $\mathbf{c}_i$ jeweils eindeutig bestimmt.

[10] Hierbei bezeichnet $\mathbf{c}_i^j$ innerhalb des Schnittes $\mathbf{c}^j$ die (eindeutige) Bedingung in der i–ten Komponente.

Satz 3.33 *Lokale Berechnung von Cut*

Sei $\mathcal{B} = ((B, E, F'), \pi)$ Präfix der Entfaltung eines SND–Systems. Für ein Ereignis $e \in E$ ist $\mathsf{Cut}([e])$ wie folgt konstruierbar:

(i) Berechne $\mathbf{c} = \mathsf{Cut}(\bigcup_{e' \in {}^\bullet\bullet e} [e'])$ mit Hilfe von Lemma 3.32.

(ii) Füge die Bedingungen $b \in e^\bullet$ hinzu: falls b in Komponente i, dann setze $\mathbf{c}_i = b$.

■*3.33*

Ein Beispiel soll diese Konstruktion verdeutlichen: für das Ereignis e_4 des Präfixes aus Abb. 3.12 gilt ${}^\bullet\bullet e_4 = \{e_1, e_3\}$. Die von diesen Ereignissen induzierten Schnitte sind $\mathsf{Cut}([e_1]) = (b_1^{(0)}, b_6^{(1)}, b_3^{(0)}, b_7^{(1)}, b_5^{(0)})$ und $\mathsf{Cut}([e_3]) = (b_1^{(0)}, b_{10}^{(3)}, b_{11}^{(1)}, b_7^{(1)}, b_9^{(1)})$. Die hochgestellten Zahlen geben hierbei wiederum die i–Tiefen der Bedingungen an, in diesem Fall sind alle i–Tiefen des zweiten Vektors größer oder gleich denen des ersten Vektors. Der Nachbereich von e_4 setzt sich aus b_{12} (Komponente 2) und b_{13} (Komponente 4) zusammen. Diese Bedingungen werden in den obigen zweiten Vektor eingesetzt, damit ergibt sich $\mathsf{Cut}([e_4])$ zu $(b_1^{(0)}, b_{12}^{(4)}, b_{11}^{(1)}, b_{13}^{(2)}, b_9^{(1)})$.

Das System zur Modellierung des gegenseitigen Ausschlusses nach Peterson aus Abbildung 3.11 sowie das mit Hilfe der Ordnung $\sqsubset_{SND_1}$ konstruierte vollständige Präfix seiner Entfaltung stellen zusammenfassend ein Anwendungsbeispiel von Algorithmus 3.12 für ein SND–System dar. Die unterschiedlich punktierten Kantenzüge im Netz des Präfixes geben wiederum die Zugehörigkeit der daran hängenden Knoten zu den fünf Komponenten des Ausgangssystems an. Der Markenfluss lässt sich auf diese Weise sehr leicht nachvollziehen, auch wenn das Netz eher unübersichtlich scheint. Die Bedingungs– und Ereignisknoten sind in der Vertikalen angeordnet bezüglich ihrer globalen Tiefe.
Die Indizes in den Namen der Knoten geben wieder die Reihenfolge an, in der die Knoten in das Präfix aufgenommen werden. Verglichen mit Präfixen, die mit der Ordnung $\sqsubset_t$ konstruiert sind, fällt auf, dass die Knoten nicht notwendigerweise schichtweise in das Präfix eingefügt werden. Bedingt durch die komponentenweise Betrachtung der Sichten beim Vergleich mittels der Ordnung $\sqsubset_{SND_1}$ kann es vorkommen, dass zunächst in einer Art Tiefensuche entlang einzelner Komponenten erweiternde Ereignisse erzeugt werden, und dabei die Erweiterung anderer Komponenten zunächst vernachlässigt wird. Das ist beispielsweise für die Ereignisse $e_1, e_2, \ldots, e_6$ zu beobachten.
Die Konstruktion mit Hilfe der Ordnung $\sqsubset_t$ verläuft gleichmäßiger entlang der Tiefe. Ein weiterer Grund ist hierfür ist, dass bei der Ordnung $\sqsubset_t$ die Größe der lokalen Konfiguration Berücksichtigung findet, die mit der Tiefe (bei niedrigem Rückwärts–Verzweigungsgrad der Transitionen) korreliert.

In Kapitel 8 wird ausgeführt, auf welche Weise die Präfix–Generierung für SND–Systeme effizient implementiert werden kann. Kapitel 10 wird verschiedene der hier und

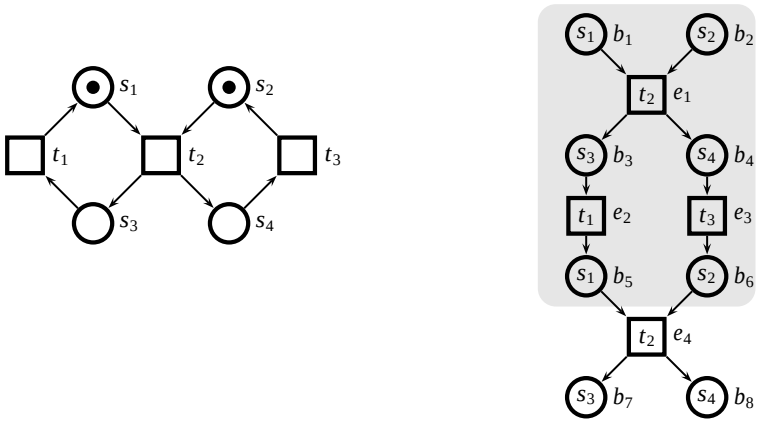
in den letzten Abschnitten beschriebenen Ordnungen anhand experimentell ermittelter Ergebnisse vergleichend gegenüberstellen. Es wird sich zeigen, dass die speziellen Ordnungen für SND–Systeme in vielen Fällen zu einer kürzeren Konstruktionszeit des Präfixes führen kann als die Verwendung der Ordnung $\sqsubset_t$.

3.6 Minimalität von Präfixen

Der Vorstellung verschiedener Ordnungen $\sqsubset$, die zu unterschiedlichen Präfixgrößen führen, sollen sich nun einige Betrachtungen zur *Minimalität* der generierten Präfixe anschließen. Abbildung 3.13 zeigt dazu ein einfaches Netzsystem; das mit Algorithmus 3.12 und der Ordnung $\sqsubset_t$ generierte Präfix ist daneben dargestellt. Der grau unterlegte Teil entspricht dem minimalen vollständigen Präfix, somit erzeugt der Algorithmus offensichtlich in diesem Sinne keine minimalen Präfixe.

Warum ist das so? Um festzustellen, dass das Präfix in Abb. 3.13(b) vollständig ist, muss bekannt sein, dass die Anfangsmarkierung $M_0 = \{ s_1, s_2 \}$ noch einmal an anderer Stelle im Präfix als Markierung einer nicht–lokalen Konfiguration, nämlich $\mathsf{Mark}(\{ t_1, t_3 \}) = M_0$, vorkommt. Für jede im Präfix codierte Markierung müsste diese Information zusätzlich verwaltet werden, da a priori nicht bekannt ist, auf welche der Markierungen später noch einmal Bezug genommen wird. Zu dem erheblichen Speicheraufwand – im schlechtesten Fall wären alle erreichbaren Markierungen zu merken – gesellen sich die Kosten der Ermittlung dieser Markierungen. Letztlich müssten mit jedem Einfügen eines neuen Ereignisses alle möglichen B–Schnitte durch das bereits konstruierte Präfix ermittelt werden, der Aufwand hierfür ist wiederum exponentiell in der Größe des Präfixes.

In [Hel99a] wird eine Methode zur Minimierung von Präfixen beschrieben, das Cut-off–Kriterium aus Definition 3.11 wird dazu folgendermaßen abgeändert: e ist Cut-off–Ereignis, wenn es eine (nicht notwendigerweise lokale) Konfiguration C gibt mit $C \equiv_M [e]$ und $C \sqsubset [e]$. Die dazu erforderlichen Berechnungen zur Identifizierung der nicht–lokalen Konfigurationen im Präfix werden mit Methoden des *Constraint Programming* durchgeführt. Eine prototypische Implementierung, die ein bereits erzeugtes Präfix auf diese Weise nachträglich minimiert, benötigt für kleine Präfixe (um 100 Ereignisse) etwa die gleiche, im Extremfall (Netze um 6000 Ereignisse) bis zum 60–fachen der Zeit, die für die eigentliche Präfix–Generierung notwendig war. Dieser Aufwand muss oft sogar dann aufgebracht werden, wenn keinerlei Minimierung gegenüber dem Verfahren mit der ursprünglichen Cut-off–Kriterium erzielt werden kann. Da das Präfix im Rahmen der Verifikation nur einmal berechnet werden muss, kann sich die Minimierung durchaus auszahlen; die Frage, ob und welcher Form das minimierte Präfix Vorteile bringt, ist Gegenstand weiterer, noch nicht abgeschlossener Untersuchungen.



(a) Ein System Σ_{17} (b) Präfix $\mathsf{Pre}(\mathsf{Unf}(\Sigma_{17}))$

Abbildung 3.13: Ein Netzsystem und sein Präfix nach Algorithmus 3.12 (Ordnung $\sqsubset_t$).

In einem anderen Sinne sind die mit dem (unveränderten) Algorithmus 3.12 generierten Präfixe durchaus minimal. Während der Konstruktion werden lediglich erreichbare Markierungen gespeichert, die mit *lokalen* Konfigurationen korrespondieren. Das ist besonders bedeutend bei der Untersuchung von Systemen mit einem hohen Grad an Nebenläufigkeit, weil die Zahl der lokalen Konfigurationen (diese wird bestimmt durch die Anzahl der Ereignisse im Präfix) im Allgemeinen um ein Vielfaches geringer ist als die Zahl der erreichbaren Zustände. Somit benötigt ihre Speicherung weit weniger Platz als das Verwalten des Zustandsraumes in Form des Erreichbarkeitsgraphen.

Kapitel 4

„Die Natur muss irgendein arithmetisch–
geometrisches Koordinatensystem besitzen,
da in ihr alle Arten von Modellen vorkommen. “
RICHARD BUCKMINSTER FULLER

Box–Algebra und Petri–Boxen

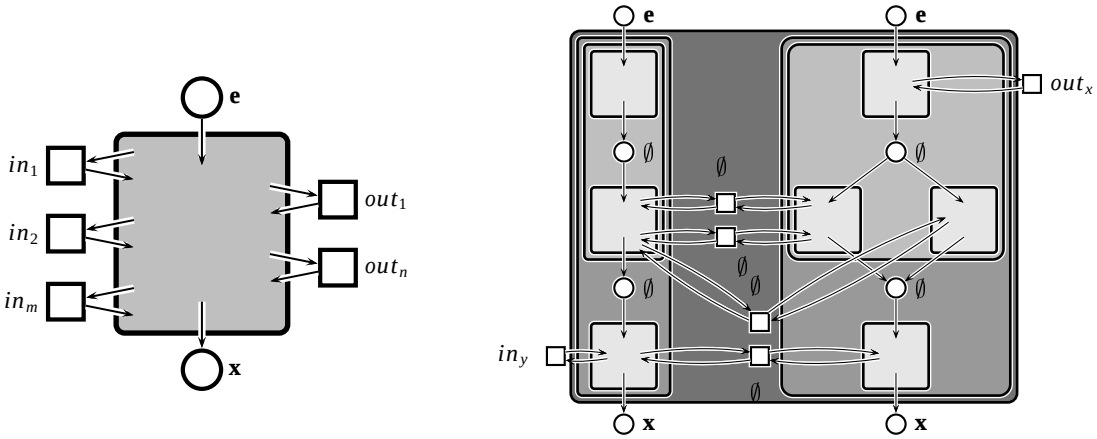
Nach der Vorstellung der formalen Grundlagen sowie der Herleitung und Beschreibung von Algorithmen zur Generierung vollständiger Präfixe soll dieses Verfahren in den nächsten Kapiteln in einen größeren Kontext gestellt werden, nämlich die Einbettung in eine Umgebung zur automatischen Verifikation von nebenläufigen, reaktiven Systemen; darunter werden solche verstanden, die aus mehreren Komponenten bestehen und miteinander kommunizieren, aber auch mit der Umgebung des Systems interagieren. In dieser Umgebung wird die Erzeugung des vollständigen Präfixes der Entfaltung das zentrale Modul sein, an das sich die verschiedenen Analysemethoden anschließen.

Die größeren Netzbeispiele aus dem letzten Kapitel, wie etwa der *Cyclic Scheduler* oder der gegenseitige Ausschluss nach Peterson haben vielleicht einen Eindruck vermittelt, dass die Modellierung nebenläufiger Systeme allein mit Hilfe von einfachen Stellen / Transitions–Netzen schnell unübersichtlich werden kann. Während der Erstellung und Änderung eines solchen Systems kommt dazu der Aspekt der Fehleranfälligkeit. Um die Eingabe der zu untersuchenden Systeme zu erleichtern, wird in dieser Arbeit auf Modellierungsseite eine spezielle Sprache angeboten, die sich semantisch leicht auf Petri–Netze abbilden lässt, deren Syntax jedoch mehr an vertraute Hochsprachen angelehnt ist. Dadurch kann auf einen speziellen graphischen Editor verzichtet werden, der für die Darstellung von Netzen oder Automaten notwendig wäre.
Bevor diese Sprache im nächsten Kapitel im Detail vorgestellt wird, soll zunächst der theoretische Unterbau entwickelt werden, der ihr zugrunde liegt. Dazu wird eine spezielle Klasse von Petri–Netzen definiert, die, zusammen mit geeigneten Operationen, eine kompositionelle Konstruktion von Netzen gestattet. Die auf diese Weise konstruierten Netze können später mit den vorgestellten Algorithmen entfaltet werden.

Petri–Netze sind – aufgrund fehlender Mittel zur Strukturierung – bekannt als ein nicht–kompositionelles Systemmodell. In der Literatur sind deswegen in der Vergangenheit verschiedene Ansätze vorgeschlagen worden, Teilnetze strukturiert zu größeren Netzen zusammenzusetzen. Dabei werden folgende Techniken unterschieden:

- Verschmelzen bzw. Synchronisieren von Transitionen verschiedener Teilnetze miteinander, z. B. [Maz88]; dies entspricht im Wesentlichen einer Modellierung synchroner Kommunikation.
- Verschmelzung einzelner Stellen von Teilnetzen, [Vog92, Val94]; hierdurch wird beispielsweise Kommunikation über globale Variablen modelliert.
- Einfügen zusätzlicher Kanten zwischen den Stellen und Transitionen der Teilnetze, [SB94]; dies entspricht im Wesentlichen asynchroner Kommunikation.
- Ersetzen von Transitionen oder ganzen Teilnetzen durch andere Netze, sogenannte Verfeinerung, [BGV90]. Auf diese Weise wird eine Abstraktion von Netzen erzielt.

Darüber hinaus ist versucht worden, prozess–algebraische Strukturierungstechniken auf Petri–Netze zu übertragen, beispielsweise aus CSP [Hoa78] oder CCS [Mil80]; ein Überblick über die Historie und verschiedene Ansätze ist u. a. in [BC91, KEB93] enthalten.
Einer dieser Vertreter ist der sogenannte *Box–Kalkül* aus [BDH92], später verfeinert und erweitert in zahlreichen Veröffentlichungen, darunter [BDE93, Dev93, KEB94]. Dieser Kalkül verwendet gleich mehrere der oben aufgezählten Techniken zur Netzkonstruktion.
Der Box–Kalkül ist mit der Intention entwickelt worden, Programmiersprachen, die Nebenläufigkeit unterstützen (beispielsweise *Occam*, [May83]), auf Petri–Netze abbilden zu können. Auf diese Weise soll die (formale) Entwicklung nebenläufiger Programme vereinfacht werden, da Programmdesign, Programmstruktur und Programmverifikation unmittelbar in Verbindung gebracht werden können.
Eine sehr wichtige Anforderung bei der Gestaltung des Kalküls stellte daher der Aspekt der *Kompositionalität* dar, d. h., den syntaktischen Operationen auf Programmebene sollten (über einen entsprechenden Homomorphismus) wohldefinierte semantische Operationen auf Netzebene, den sogenannten *Petri–Boxen*, gegenüberstehen.
Der Name Petri–*Box* rührt dabei von dem Umstand her, dass für die eigentliche Komposition einzelner Teilnetze nur deren Schnittstellen nach außen interessant sind, nicht jedoch ihre innere Struktur (*Black-Box*–Prinzip), siehe auch Abb. 4.1(a).
Die Schnittstelle eines Box–Netzes wird von dessen Eingangs– und Ausgangsstellen sowie nicht–leer beschrifteten Transitionen gebildet. Über die Stellenknoten wird die strukturelle Komposition vorgenommen, die Operationen wie Sequenz, Alternative



(a) Schema einer Petri–Box.

(b) Beispiel einer Komposition von Boxen.

Abbildung 4.1: Prinzipielle Idee der Petri–Boxen.

und Schleife umfasst. Die Transitionen der Netze sind mit Kommunikationsbeschriftungen versehen, über die die Synchronisation der nebenläufigen Prozesse stattfindet. In Abb. 4.1(b) ist schematisch die Komposition einer Box aus einzelnen Boxen dargestellt; die Konstruktion erfolgt von innen nach außen (zunehmender Grauwert).
In seiner ursprünglichen Definition enthält der Box–Kalkül darüber hinaus hierarchische Konstrukte wie Verfeinerung und Rekursion [BDE93] und weitere spezielle Operationen (z. B. Umbeschriftung). In der vorliegenden Arbeit jedoch wird nur ein ausgesuchtes Fragment des in [BDH92] beschriebenen Kalküls verwendet. Die vielleicht augenfälligste Änderung betrifft die Einschränkung des Parallelоperators, der hier nur auf oberster Ebene zugelassen ist. Neben den Vorschlägen, die in [EB96] unter dem Namen *Small Box Calculus* veröffentlicht wurden, sind einige zusätzliche Operationen definiert worden [RM96], mit denen sich die Größe der konstruierten Netze weiter minimieren läßt.
Die eigentliche Petri–Box–Algebra besteht aus zwei Teilen: um unabhängig von einer bestimmten Hochsprache zu sein, wird der aktuelle Programmtext zunächst übersetzt in eine CCS–ähnliche Prozess–Notation, die von konkreten Aktionen abstrahiert. Die syntaktische Domäne dieser Notation bilden die sogenannten *Box–Ausdrücke*; sie werden in Abschnitt 4.1 näher erläutert.
Die semantische Domäne stellen die oben erwähnten *Petri–Boxen* dar, diese bestehen aus beschrifteten Netzen und werden in 4.3 eingeführt. Abschnitt 4.4 beschreibt alle notwendigen Operationen auf Petri–Boxen, mit denen diese Netze kompositionell zu größeren Netzen verbunden werden können. Mit Hilfe der Operationen wird schließlich die Semantik der Box–Ausdrücke in 4.5 festgelegt.

Die Anbindung einer konkreten Programmiersprache an den Box–Kalkül wird, wie erwähnt, Gegenstand von Kapitel 5 sein. Mit Hilfe der hier beschriebenen Techniken ist es leicht möglich, eine Anbindung auch an eine andere als die hier beschriebene Hochsprache vorzunehmen.

4.1 Syntax der Box–Ausdrücke

Für die Annotation der atomaren Bestandteile einer parallelen Programmiersprache werden auf Ebene der Box–Algebra spezielle Mengen von Bezeichnern benötigt; sie dienen später zur Beschriftung der Transitionen.

Definition 4.1 *Konjugation*

Sei $\mathcal{A}$ ein abzählbares Alphabet von Bezeichnern, den *Aktionsnamen*, auf dem eine Bijektion $\widehat{\ }: \mathcal{A} \to \mathcal{A}$ mit den folgenden Eigenschaften definiert ist:

$$\widehat{a} \neq a \text{ und } \widehat{\widehat{a}} = a \quad \text{für alle } a \in \mathcal{A}.$$

Diese Bijektion wird *Konjugation* genannt; a und $\widehat{a}$ sind *Konjugierte* voneinander. ■4.1

Aktionsnamen sind also stets paarweise vorhanden; sie werden notiert durch kleine lateinische Buchstaben aus der ersten Hälfte des Alphabets, beispielsweise $\mathcal{A} = \{a, \widehat{a}, b, \widehat{b}, g, \widehat{g}\}$.
Die hinter der Menge $\mathcal{A}$ stehende Idee ist, dass konjugierte Aktionsnamen a und $\widehat{a}$ synchronisieren können (analog wie in CCS, [Mil89]; dort werden die Konjugierten durch Überstrich gekennzeichnet). Normalerweise können die Elemente der Menge $\mathcal{A}$ frei gewählt werden, wobei dann jeweils für jeden Aktionsnamen angegeben werden muss, welches die dazu Konjugierte ist.
Andererseits kann auch die Kommunikation über einen Kanal (d. h. eine gerichtete Verbindung bestimmter Kapazität zwischen nebenläufigen Prozessen) modelliert werden (wie z. B. in CSP [Hoa78] oder B(PN)^2 [BH93]). Bezeichne *ch* einen solchen Kommunikationskanal, dann sind die Befehle *ch*!*val* (mit der Bedeutung: sende den Wert *val* über Kanal *ch*) sowie *ch*?*val* (lese genau den Wert *val* von Kanal *ch*) in diesem Sinne Konjugierte voneinander.
Neben den Aktionsnamen wird ein – zu $\mathcal{A}$ disjunktes – Alphabet $\mathcal{S}$ von *Sprungmarken* benötigt, auf dem ebenfalls die obige Bijektion definiert ist. Die Elemente von $\mathcal{S}$ werden hier mit kleinen griechischen Buchstaben notiert: $\beta, \gamma, \kappa, \ldots$ Die Vereinigung der beiden Bezeichnermengen ergibt die Menge der Kommunikationsbeschriftungen, $\mathcal{L} = \mathcal{A} \cup \mathcal{S}$.

Mengen $\alpha \subseteq \mathcal{L}$ von Aktionsnamen und / oder Sprungmarken werden *Aktionsmengen* genannt. Im Gegensatz zu [BDH92] werden in dieser Arbeit nur einfache Mengen betrachtet, keine Multimengen.
Bei der Angabe einelementiger Aktionsmengen werden im Folgenden die Mengenklammern häufig weggelassen: a anstelle $\{a\}$. Insbesondere ist die leere Menge $\emptyset$ eine (interne) Aktionsmenge, die der unsichtbaren τ–Aktion von CCS entspricht. Aktionsmengen stellen die elementaren Bestandteile der Box–Algebra dar, deren Syntax nachfolgend beschrieben ist.

Definition 4.2 *Syntax der Box–Ausdrücke*

Seien $a \in \mathcal{A}$ ein Aktionsname, $\alpha \subseteq \mathcal{A}$ eine Aktionsmenge und $\gamma \in \mathcal{S}$ eine Sprungmarke. Die Syntax der Box–Ausdrücke E ist induktiv definiert durch

E	::=	P	*Parallele Komponenten*
	|	$[\, a : \mathsf{E} \,]$	*Scoping (über Aktionsname a)*
P	::=	K	*Sequenzielle Komponente (frei von Sprungmarken)*
	|	P ‖ P	*Nebenläufige Komposition*
K	::=	S	*Sequenzielle Komponente*
	|	$[\, \gamma : \mathsf{K} \,]$	*Scoping (über Sprungmarke γ)*
S	::=	α	*Elementare Aktion*
	|	**stop**	*Abbruch*
	|	S ; S	*Sequenzielle Komposition*
	|	S [] S	*Alternative*
	|	$[\, \mathsf{S} * \mathsf{S} \,]$	*Schleife*
	|	$[\, \mathsf{S} \xrightarrow{\gamma} \,]$	*Sprung*
	|	$[\, \xrightarrow{\widehat{\gamma}} \mathbf{e} \,]$	*Sprungziel: zum Start des Teilausdrucks*
	|	$[\, \xrightarrow{\widehat{\gamma}} \mathbf{x} \,]$	*Sprungziel: zum Ende des Teilausdrucks*

■4.2

Bei den Operatoren bindet „ ; “ stärker als „ [] “. Zur Verdeutlichung des Vorrangs einzelner Operatoren können Klammern verwendet werden. Die Menge der (wohlgeformten) Box–Ausdrücke wird nachfolgend mit DBEXPR bezeichnet.
Beispiele für Box–Ausdrücke sind $[a : [\widehat{a} * \mathbf{stop}] \parallel a; b; a]$ oder $[a : [b : [c : \{\widehat{a}, b\};\ \parallel \widehat{a}]]]$.

Die vier Nichtterminale E, P, K und S der in Def. 4.2 angegebenen Grammatik strukturieren einen jeden Box–Ausdruck: von innen nach außen betrachtet spezifizieren Teilausdrücke S sequenzielle Programme, die aus elementaren Aktionen mit Hilfe der Konstrukte Sequenz, Alternative, Schleife sowie Sprüngen zusammengesetzt sind. In der nächsten Ebene K werden alle Sprünge geschlossen, d. h. Ausgangspunkte von Sprüngen und ihre Sprungziele werden zusammengeführt; nachdem alle Sprünge aufgelöst

worden sind, enthält die sequenzielle Komponente K keine Sprungmarken mehr. Die einzelnen sequenziellen Komponenten K werden parallel komponiert zu einem parallelen Programm, das jedoch noch unsynchroniesiert ist. Auf oberster Ebene werden schließlich alle durch die elementaren Aktionen angegebenen Kommunikationen ausgeführt, indem zueinander Konjugierte verschmolzen werden.

Den einzelnen syntaktischen Konstrukten liegt die folgende Bedeutung zugrunde: Elementare Aktionsmengen repräsentieren die entsprechenden atomaren, ausführbaren Aktionen des Programmes. Der Ausdruck **stop** hingegen führt keine Aktion aus, er bewirkt ein Halten und somit eine Verklemmung des entsprechenden Prozesses.
Der Operator „ ; “ steht für die Hintereinanderausführung, und „ [] “ die nichtdeterministische Auswahl. Die Schleife $[L * T]$ beschreibt das kein–, ein– oder mehrmalige Durchlaufen des Schleifenrumpfes L, gefolgt vom einmaligen Ausführen des terminierenden Ausdrucks T; dieses Konstrukt ist somit verhaltensäquivalent zum entsprechenden rekursiven Ausdruck $R = (L; R) [] T$ aus CCS. Im Gegensatz zu [BDH92] wird hier nicht zwingend eine Initialisierung der Schleife gefordert; das dort benutzte Konstrukt kann jedoch auf einfache Weise emuliert werden: $[I * L * T] = I; [L * T]$.
Mit dem Ausdruck $[\alpha \stackrel{\gamma}{\rightarrow}]$ werden Sprünge spezifiziert: für die leere Aktionsmenge $\alpha = \emptyset$ finden diese unbedingt statt, andernfalls bedingt, d. h. in Abhängigkeit von den Aktionen der Aktionsmenge α. Das Sprungziel wird durch Ausdrücke der Form $[\stackrel{\gamma}{\rightarrow} z]$ spezifiziert: für $z = \mathbf{e}$ wird zurück an den Eintrittspunkt eines Teilausdrucks, für $z = \mathbf{x}$ an dessen Terminierungspunkt gesprungen. Wo genau diese Zielpunkte lokalisiert sind, wird später bei der Beschreibung der Semantik deutlich. Die in Sprung– und Sprungziel–Ausdrücken verwendeten Sprungmarken γ, $\widehat{\gamma}$ sind Konjugierte voneinander, sie werden über den Scoping–Operator $[\gamma : \mathsf{K}]$ miteinander verschmolzen. Allgemein kann das Scoping aufgefasst werden als Hintereinanderausführung von Synchronisation (Verschmelzen konjugierter Kommunikationsbeschriftungen) und Restriktion (Ausblenden von Kommunikationsbeschriftungen).
Schließlich erlaubt der $\|$ –Operator die parallele Komposition von Ausdrücken; anders als in [BDH92] darf er nicht beliebig verschachtelt auftreten, sondern ist nur auf oberster Ebene zugelassen.
Die hier vorgestellte Algebra kann somit zur Modellierung von nebenläufigen Systemen verwendet werden, die aus sequenziellen, miteinander kommunizierenden Komponenten bestehen. Das Scoping über Aktionsnamen ist ebenfalls ausschließlich auf oberster Ebene zugelassen, für die Modellierung bedeutet das die ausschließliche Deklaration globaler Variablen.
Der Grund für diese syntaktischen Einschränkungen ist technischer Natur: einerseits garantieren sie die Konstruktion endlicher Netzsysteme; diese für die Präfix–Generierung essenzielle Voraussetzung bietet die Box–Algebra in der ursprünglichen Definition an vielen Stellen nicht. Auf der anderen Seite erlaubt der hier vorgestellte Kalkül eine vereinfachte, in gewissem Sinne minimale Konstruktion von Netzen. Neben der

meist intuitiveren Darstellung der Programme als Netzgraph ergeben sich durch die geringere Netzgröße praktische Vorteile bei der anschließenden Verifikation.

4.2 Beschriftete Netze

Den Definitionsbereich der Box–Algebra bilden die sogenannten Petri–Boxen, eine spezielle Klasse von Stellen / Transitions–Netzen. Sie basieren auf den in Kapitel 2.2 vorgestellten Netzsystemen, hinsichtlich ihrer Struktur besitzen sie einige zusätzliche Eigenschaften, daneben sind die Knoten der Netze mit speziellen Beschriftungen versehen. Gegenüber den ursprünglichen Definitionen im Box–Kalkül [BDH92] sind wiederum einige Änderungen vorgenommen worden, vorwiegend Vereinfachungen, auf die an den entsprechenden Stellen hingewiesen wird.
Die Transitionsknoten erhalten als Beschriftungen Aktionsmengen, die aus den im letzten Abschnitt beschriebenen Aktionsnamen und Sprungmarken bestehen. Über diese Kommunikationsbeschriftungen wird die Synchronisation von Transitionen (und damit Aktionen) kontrolliert. Die Stellenknoten erhalten eine davon abweichende Beschriftung, die die *Lage* der jeweiligen Stelle im Netz kennzeichnet. Wie schon in der einführenden Abbildung 4.1 angedeutet worden ist, dienen nicht–leere Stellen– wie Transitionsbeschriftungen als eine Art Kommunikations–Schnittstelle, über die aus einzelnen Netzen größere Netze kompositionell konstruiert werden können.

Definition 4.3 *Beschriftetes Netz*

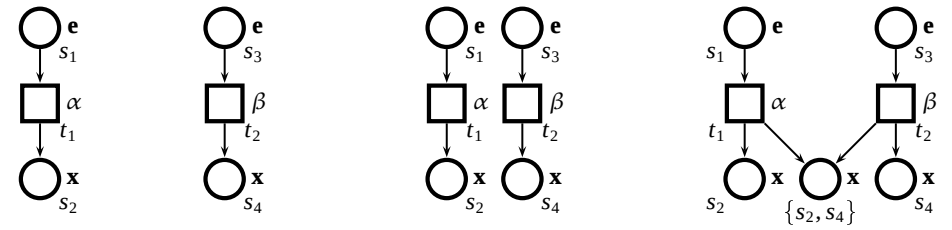
Ein be*schriftetes (Petri–)Netz* ist ein Quadrupel $\Lambda = (S, T, F, \lambda)$, wobei (S, T, F) ein Petri–Netz ist und λ eine Beschriftungsfunktion mit

$$\begin{array}{lcl} \lambda: & S & \rightarrow \quad \{\mathbf{e}, \emptyset, \mathbf{x}\} \\ \lambda: & T & \rightarrow \quad \mathcal{L} \end{array}$$ ■4.3

Stellen, die mit **e** beschriftet sind, heißen *Eingangsstellen*; mit **x** beschriftete Stellen werden *Ausgangsstellen* genannt. Die übrigen, mit $\emptyset$ beschrifteten Stellen sind *interne* Stellen. Ebenso heißen mit $\emptyset$ beschriftete Transitionen *intern*,alle anderen werden *Kommunikations–* oder *beobachtbare Transitionen* genannt. In den graphischen Darstellungen dieser Arbeit werden die Beschriftungen an die Netzknoten geschrieben.

Notation 4.4 *Ein–/Ausgangsstellen beschrifteter Netze*

Sei $\Lambda = (S, T, F, \lambda)$ ein beschriftetes Netz.
${}^\bullet\Lambda = \{s \in S \mid \lambda(s) = \mathbf{e}\}$ bezeichnet die Menge der Eingangsstellen,
$\Lambda^\bullet = \{s \in S \mid \lambda(s) = \mathbf{x}\}$ die Menge der Ausgangsstellen von Λ. ■4.4



(a) Λ_1 (b) Λ_2 (c) $\Lambda_3 = \Lambda_1 \sqcup \Lambda_2$ (d) $\Lambda_3 \boxplus (\{s_2, s_4\}, \mathbf{x})$

Abbildung 4.2: Operationen auf beschrifteten Netzen.

Die natürliche Anfangsmarkierung eines beschrifteten Netzes ist diejenige, bei der nur die Eingangsstellen jeweils mit genau einer Marke belegt sind und sonst keine Stelle markiert ist, also $M_0 = {}^\bullet\Lambda$. $\Sigma(\Lambda) = (\Lambda, M_0)$ bezeichne im Folgenden das dem beschrifteten Netz Λ zugrunde liegende System mit der natürlichen Anfangsmarkierung M_0. In den graphischen Darstellungen werden Markierungen oft nicht explizit angezeigt, da es in diesem Kapitel vornehmlich um die kompositionelle Konstruktion von Netzen geht, nicht um deren Schalten.
Trägt jede Ausgangsstelle (und keine andere) genau eine Marke, so ist dies der Endzustand (Terminierung); dieser muss nicht notwendigerweise vom Anfangszustand aus über eine Folge von schaltenden Transitionen erreichbar sein.

Auf beschrifteten Netzen sind eine Reihe von Operationen definiert, die Manipulationen anhand der Beschriftungen erlauben. Für die Definitionen dieses Abschnitts seien zwei disjunkte beschriftete Netze $\Lambda = (\ S, T, F, \lambda\ )$ und $\Lambda' = (\ S', T', F', \lambda'\ )$ vorausgesetzt, für die neben den üblichen Disjunktheitsbedingungen auch $S \cap T' = \emptyset = S' \cap T$ und $(S \cup T') \cap (S' \cup T) = \emptyset$ gelte. Die Disjunktheit der Netze bezieht sich ausdrücklich nicht auf die Beschriftungen.

Definition 4.5 *Vereinigung disjunkter Netze*

Die *Vereinigung* von Λ und Λ' ist definiert als

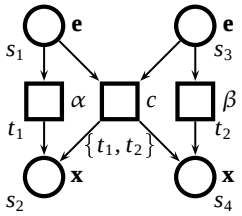
$$\Lambda \sqcup \Lambda' \ = \ (\ S \cup S',\ T \cup T',\ F \cup F',\ \lambda \cup \lambda'\ ) \qquad \blacksquare 4.5$$

Anschaulich werden bei der Vereinigung die einzelnen beschrifteten Netze nebeneinandergelegt zu einem einzigen Netz, ein Beispiel zeigt Abbildung 4.2(a)–(c).
Einzelne Netzknoten können über die nächste Operation zusammengefasst werden.

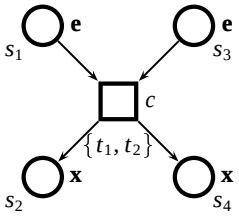
Definition 4.6 *Multiplikation von Knoten*

Seien R, R' Mengen von Stellen (bzw. Transitionen). Definiere die *Multiplikation* als

$$R \boxtimes R' \ = \ \{\, \{\, r, r' \,\} \mid r \in R \ \wedge\ r' \in R' \,\}\ . \qquad \blacksquare 4.6$$



(a) $\Lambda_4 = \Lambda_3 \boxplus (\{\{t_1, t_2\}\}, \{\{t_1, t_2\} \mapsto \{c\}\})$



(b) $\Lambda_5 = \Lambda_4 \boxminus \{t_1, t_2\}$

Abbildung 4.3: Operationen auf beschrifteten Netzen (Fortsetzung).

Die Multiplikation erfüllt die Kommutativität ($R \boxtimes R' = R' \boxtimes R$) und kann somit als symmetrisches kartesisches Produkt angesehen werden. Für das beschriftete Netz Λ_3 aus Abb. 4.2(c) gilt z. B. $\{s_1\} \boxtimes \{s_3\} = \{s_1, s_3\}$ und $\{t_1\} \boxtimes \{t_2\} = \{t_1, t_2\}$.

Die so erhaltenen Namen sind für $r \neq r'$ verschieden von den Namen der bereits im Netz vorhandenen Knoten[1], mit diesen Namen versehene Knoten somit disjunkt zu den existierenden Knoten.

Die folgenden beiden Definitionen legen die Hinzunahme auf diese Weise gewonnener Stellen bzw. Transitionen fest; jeder neue Knoten erhält außerdem eine Beschriftung, die von den Beschriftungen der an der Multiplikation beteiligten Knoten abweichen kann.

Definition 4.7 *Hinzufügen einer Stelle*

Seien $s = \{s_1, s_2\} \in S \boxtimes S, s \notin S$ eine Stelle und $l \in \{\mathbf{e}, \emptyset, \mathbf{x}\}$ eine Beschriftung dieser Stelle. Das Netz der *Stellenaddition* $\Lambda \boxplus (\{s\}, l)$ ist gegeben durch

$$(S \cup \{s\},\ T,\ F \cup (\{s\} \times (s_1 \cup s_2)^\bullet) \cup ({}^\bullet(s_1 \cup s_2) \times \{s\}),\ \lambda \cup \{s \mapsto l\}). \quad \blacksquare 4.7$$

Abbildung 4.2(d) zeigt die Addition der Stelle $\{s_2, s_4\}$ zum beschrifteten Netz Λ_3. Die mit dieser neuen Stelle verbundenen Kanten werden unmittelbar von den Knoten übernommen, die an der Multiplikation beteiligt sind.

Analog zum Einfügen von Stellen ist das Hinzufügen von Transitionen definiert. Allgemeiner wird hierbei gleich eine Menge neuer Transitionen addiert.

[1] Diese Verschiedenheit gilt unter der Voraussetzung, dass ein mit Hilfe derselben Multiplikation gewonnener Knoten nicht bereits in das Netz eingefügt wurde. In [BDH92] ist auf beschrifteten Netzen zusätzlich eine Äquivalenzrelation definiert, die solche Duplikate von Knoten durch Wahl eines kanonischen Repräsentanten einer Klasse „ausfiltert“, siehe auch die Bemerkungen in Kap. 4.6.

Definition 4.8 *Hinzufügen einer Menge von Transitionen*

Sei $\Lambda = (S, T, F, \lambda)$ ein beschriftetes Netz. Außerdem seien $U \subseteq T \boxtimes T$ eine Menge von (zu $S \cup T$ disjunkten) Transitionen und $\lambda_U : U \to \mathcal{L}$ eine Beschriftung dieser Transitionen. Definiere die *Addition von Transitionen* durch

$$\Lambda \boxplus (U, \lambda_U) = (S, T \cup U, F \cup F_U, \lambda \cup (\lambda_U|_U))$$

mit

$$\begin{aligned} F_U = {} & \{(s, \{u_1, u_2\}) \in S \times U \mid F \cap \{(s, u_1), (s, u_2)\} \neq \emptyset\} \\ \cup {} & \{(\{u_1, u_2\}, s) \in U \times S \mid F \cap \{(u_1, s), (u_2, s)\} \neq \emptyset\} \end{aligned}$$ ■4.8

Ein einfaches Anwendungsbeispiel für diese Operation zeigt Abbildung 4.3(a). Die letzte Definition dieses Abschnitts beschreibt das Löschen von Elementen aus einem Netz.

Definition 4.9 *Entfernen von Knoten*

Sei $\Lambda = (S, T, F, \lambda)$ ein beschriftetes Netz und sei $R \subseteq S \cup T$. Definiere

$$\Lambda \boxminus R = (S \backslash R, T \backslash R, F|_{((S \cup T) \backslash R) \times ((S \cup T \backslash R)}, \lambda|_{(S \cup T) \backslash R}) \; .$$ ■4.9

Sollen aus dem beschrifteten Netz Λ_4 aus Abb. 4.3 die Transitionen t_1 und t_2 gelöscht werden, dann ergibt sich das Netz Λ_5. Gleichzeitig mit den Knoten werden auch alle daran hängenden Kanten aus dem Netz genommen.
Das Entfernen von Knoten ist in beliebiger Reihenfolge möglich, d. h. für $R_1, R_2 \subseteq S \cup T$ gilt $(\Lambda \boxminus R_1) \boxminus R_2 = \Lambda \boxminus (R_1 \cup R_2) = (\Lambda \boxminus R_2) \boxminus R_1$ [BDH92].

Nach diesen Vorbereitungen kann nun die Definition der Petri–Boxen sowie der einzelnen Operationen auf ihnen vorgenommen werden.

4.3 Petri–Boxen

Dieser Abschnitt definiert, um welche Art von Objekten es sich bei Boxen handelt und wie deren Basiselemente aussehen. Wie zuvor bereits angedeutet, dienen beschriftete Netze als Grundlage für Boxen.

Definition 4.10 *Petri–Box*

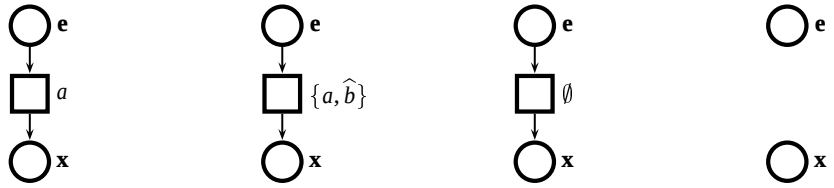
Eine *Petri–Box* (kurz *Box*) B ist ein beschriftetes Netz, das die folgenden Bedingungen erfüllt:

(i) ${}^\bullet B \neq \emptyset$ (B besitzt mindestens eine Eingangsstelle);

(ii) $B^\bullet \neq \emptyset$ (B besitzt mindestens eine Ausgangsstelle);

(iii) $\forall s \in B^\bullet : s^\bullet = \emptyset$ (Es führen keine Kanten aus Ausgangsstellen).

■4.10



(a) B_a (b) $B_{\{a,\widehat{b}\}}$ (c) $B_{\emptyset}$ (d) $B_{\mathbf{stop}}$

Abbildung 4.4: Vier elementare Boxen.

Daneben gilt insbesondere für alle Transitionen einer Box die T–Eingeschränktheit. Eine Box B ist endlich, wenn das zugrunde liegende Netz endlich ist. Mit DBOX wird im Folgenden die Domäne der Boxen bezeichnet, die die Elemente der Box–Algebra bilden.
Der hier vorgeschlagene Ansatz weicht etwas ab von den ursprünglichen Definitionen aus [BDH92]. Zum einen ist es erlaubt, dass Eingangsstellen eingehende Kanten besitzen; diese Eigenschaft wird später für die einfachere Darstellung von Schleifen und Sprüngen benötigt. Daneben können Transitionen (temporär) leere Vor– bzw. Nachbereiche besitzen. Diese Verletzung der T–Eingeschränktheit stellt lediglich ein Zwischenstadium bei der Komposition von Boxen dar, bei vollständig synchronisierten Netzen ist diese Bedingung wieder erfüllt.

Zunächst werden nun die Basiselemente dieser Algebra definiert. Danach folgen die Operationen, mit denen, ausgehend von den Basiselementen, Boxen kompositionell zusammengesetzt werden können.

Definition 4.11 *Elementare Kommunikations–Box*

Für eine Aktionsmenge $\alpha \subseteq \mathcal{A}$ wird eine elementare Box B_α wie folgt definiert:

$$B_\alpha \;=\; (\,\{\,s_e,\, s_x\,\},\ \{\,t\,\},\ \{\,(s_e,\, t),\ (t,\, s_x)\,\},\ \{\,s_e \mapsto \mathbf{e},\ t \mapsto \alpha,\ s_x \mapsto \mathbf{x}\,\}\,)$$

■4.11

Sprungmarken sind hierbei absichtlich als Elemente von Aktionsmengen elementarer Boxen ausgeschlossen; sie werden später innerhalb spezieller Boxen verwendet. Für den hier betrachteten Ansatz werden nur solche Transitionsbeschriftungen betrachtet, bei denen in einer Aktionsmenge nicht zugleich ein Kommunikationsbeschriftung und deren Konjugierte vorkommen; für $a \in \mathcal{A}$ gilt also stets: $a \in \alpha \Rightarrow \widehat{a} \notin \alpha$. Diese gegenüber den Originaldefinitionen [BDH92] vorgenommene Einschränkung garantiert wiederum die Endlichkeit der konstruierten Boxen.

Eine weitere elementare[2] Box beschreibt die nachstehende Definition.

Definition 4.12 ***stop***–*Box*

Die **stop**–Box wird definiert durch

$$B_{\mathbf{stop}} = (\{ s_e, s_x \}, \emptyset, \emptyset, \{ s_e \mapsto \mathbf{e}, s_x \mapsto \mathbf{x} \}) \qquad \blacksquare 4.12$$

Abbildung 4.4 zeigt Ausprägungen elementarer Petri–Boxen. Die Box $B_\emptyset$ terminiert, ohne einen weiteren Effekt erzielt zu haben; sie wird manchmal auch mit $B_{\mathbf{skip}}$ gekennzeichnet, da sie die Semantik des gleichnamigen *Occam*–Befehls beschreibt.

4.4 Box–Operatoren

In diesem Abschnitt wird eine Auswahl der wichtigsten Operationen auf Boxen definiert, soweit sie für die hier verwendete Semantik benötigt werden.

4.4.1 Nebenläufigkeit und Alternative

Die nebenläufige Komposition zweier Boxen greift unmittelbar auf die disjunkte Vereinigung von Netzen zurück.

Definition 4.13 *Nebenläufige Komposition von Boxen*

Seien B_1 und B_2 zwei o. B. d. A. disjunkte[3] Boxen. Die *nebenläufige Komposition* von B_1 und B_2 ist definiert als

$$B_1 \parallel B_2 = B_1 \sqcup B_2$$

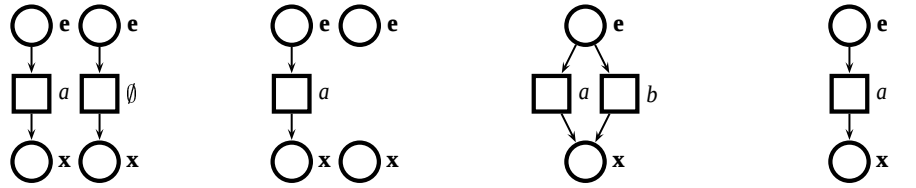
$\blacksquare 4.13$

Die Operation $\parallel$ ist kommutativ und assoziativ [BDH92]. Abbildungen 4.5(a) und 4.5(b) zeigen Beispiele für die nebenläufige Komposition von Boxen.

Bei der alternativen Komposition zweier Boxen werden die Eingangsstellen beider Netze multipliziert, sowie, unabhängig davon, die Ausgangsstellen der beiden Netze.

[2] Die **stop**–Box ist ableitbar mittels Restriktion, $B_{\mathbf{stop}} = B_a$ **rs** a, und somit keine elementare Box im eigentlichen Sinne.

[3] Die Netze der Boxen können durch eine einfache Umbenennung der Knoten disjunkt gemacht werden.



(a) $B_a \parallel B_{\mathbf{skip}}$ (b) $B_a \parallel B_{\mathbf{stop}}$ (c) $B_a [] B_b$ (d) $B_a [] B_{\mathbf{stop}}$

Abbildung 4.5: Parallele und alternative Komposition einfacher Boxen.

Definition 4.14 *Alternative Komposition von Boxen*

Seien B_1, B_2 zwei (o. B. d. A.) disjunkte Boxen. Die *Alternative* oder *Auswahl* von B_1 und B_2 ist definiert als

$$\begin{aligned} B_1 [] B_2 \quad = \quad & (B_1 \sqcup B_2) \quad \boxplus \quad ({}^\bullet B_1 \boxtimes {}^\bullet B_2, \mathbf{e}) \\ & \boxplus \quad (B_1^\bullet \boxtimes B_2^\bullet, \mathbf{x}) \\ & \boxminus \quad ({}^\bullet B_1 \cup {}^\bullet B_2 \cup B_1^\bullet \cup B_2^\bullet) \end{aligned}$$

■4.14

Der Kontrollfluss teilt sich für die jeweiligen Alternativen nichtdeterministisch auf und wird anschließend wieder zusammengeführt. Die Operation [] ist ebenfalls kommutativ und assoziativ [BDH92]. In Abb. 4.5(c) und 4.5(d) sind Beispiele für die alternative Komposition von Boxen dargestellt.

4.4.2 Sequenz und Iteration

Bei der sequenziellen Komposition zweier Boxen werden die Ausgangsstellen der ersten Box mit den Eingangsstellen der zweiten Box multipliziert.

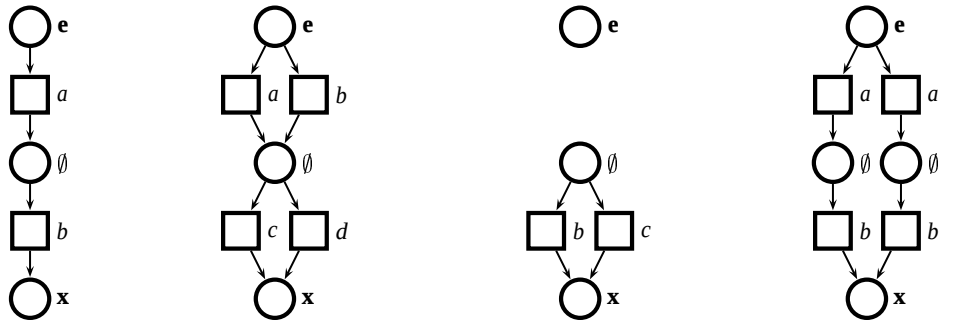
Definition 4.15 *Sequenzielle Komposition von Boxen*

Seien B_1 und B_2 zwei (o. B. d. A.) disjunkte Boxen. Dann definiert

$$\begin{aligned} B_1 \,;\, B_2 \quad = \quad & (B_1 \sqcup B_2) \quad \boxplus \quad (B_1^\bullet \boxtimes {}^\bullet B_2, \emptyset) \\ & \boxminus \quad (B_1^\bullet \cup {}^\bullet B_2) \end{aligned}$$

die sequenzielle Komposition von B_1 und B_2. ■4.15

Die sequenzielle Komposition ist assoziativ, d. h. es gilt $B_1;(B_2;B_3) = B_1;B_2;B_3 = (B_1;B_2);B_3$ [BDH92]. Durch die Multiplikation der Ausgangsstellen der ersten Box mit den Eingangsstellen der zweiten wird sichergestellt, dass die erste Box terminiert haben



(a) $B_a; B_b$ (b) $(B_a \,[]\, B_b); (B_c \,[]\, B_d)$ (c) $B_{\mathbf{stop}}; (B_b \,[]\, B_c)$ (d) $B_a; B_b \,[]\, B_a; B_b$

Abbildung 4.6: Beispiele für die sequenzielle Komposition von Boxen.

muss, bevor die zweite betreten werden kann. Beispiele für die Sequenz von Boxen zeigt Abbildung 4.6.

Eine der Sequenz verwandte Operation stellt die Iteration von Boxen dar, die wiederholte Hintereinanderausführung. Die durch den Ausdruck $B = [\, B_L * B_T \,]$ symbolisierte Bedeutung ist die folgende: der Schleifenrumpf B_L kann beliebig oft ausgeführt werden (auch keinmal), danach ist genau eine Ausführung der Box B_T möglich. Das der Box B_L entsprechende Teilnetz ist in Form einer Schleife in den B darstellenden Repräsentanten eingebunden;
Bei der Komposition der entsprechenden Boxen werden die Ein– und Ausgangsstellen der Box B_L zusammen mit den Eingangsstellen der Box B_T multipliziert.

Definition 4.16 *Iteration von Boxen*

Seien B_L und B_T zwei (o. B. d. A. disjunkte) Boxen. Die *Iteration* von B_L und B_T ist definiert durch

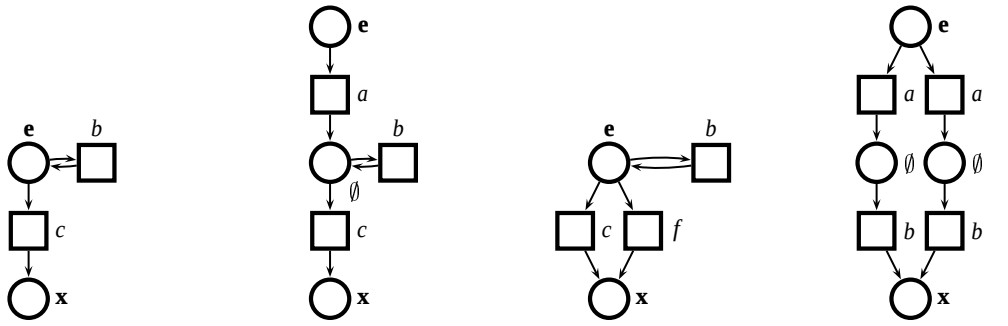
$$[\, B_L * B_T \,] \;=\; (\, B_H \sqcup B_T \,) \boxplus (\, B_H^\bullet \boxtimes {}^\bullet B_T, \mathbf{e} \,) \boxminus (\, B_H^\bullet \cup {}^\bullet B_T \,)$$

mit

$$B_H \;=\; B_L \boxplus (\, {}^\bullet B_L \boxtimes B_L^\bullet, \mathbf{e} \,) \boxminus (\, {}^\bullet B_L \cup B_L^\bullet \,).$$

■4.16

Die in [BDH92] vorgeschlagene dreiteilige Iterations–Konstruktion $[B_I * B_L * B_T]$ enthält eine zusätzliche Box B_I zur Initialisierung der Schleife. Auf diese Weise wird garantiert, dass die Eingangsstellen einer Box niemals eingehende Kanten aufweisen. In praktischen Anwendungen wird die Box B_I dabei häufig mit $B_{\mathbf{skip}}$ belegt, wenn keine



(a) $[B_b * B_c]$ (b) $B_a; [B_b * B_c]$ (c) $[B_b * B_c [] B_f]$ (d) $B_a; B_b [] B_a; B_b$

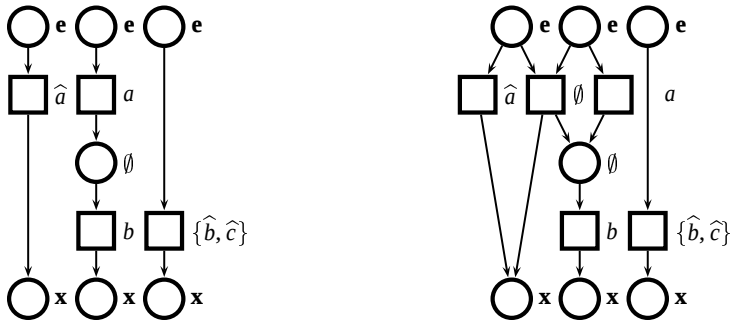
Abbildung 4.7: Beispiele für die Iteration von Boxen.

besondere Initialisierung erforderlich ist, und führt somit zu einer unnötigen Vergrößerung des Netzes.
Deshalb wird hier die etwas einfachere Schleifensyntax verwendet, die allerdings im Zusammenwirken mit der alternativen Komposition zu Mehrdeutigkeiten führen kann. Abbildung 4.7(c) zeigt ein Beispiel: das Netz $[B_b * B_c] [] B_f$ ist isomorph zu demjenigen von $[B_b * (B_c [] B_f)]$. Der erste Ausdruck erlaubt also ein Ausführen von B_f, nachdem die Schleife wenigstens einmal durchlaufen worden ist, eine mit Sicherheit ungewollte Semantik. Um das zu vermeiden, wird festgelegt, dass Schleifen nicht uninitialisiert in einer Alternative vorkommen dürfen.[4] In Definition 4.2 ist dieser Fall nicht explizit aufgenommen, damit die Beschreibung der Syntax nicht unnötig kompliziert wird.

4.4.3 Scoping

Mit der Synchronisation von Boxen wird als nächstes ein unärer Operator betrachtet, bei dem die Beschriftung der Transitionen von Bedeutung ist. Die für die Synchronisation relevanten Beschriftungen sind dabei (nicht–leere) Kommunikationsaktionen.
Zwei Transitionen eines Netzes können synchronisieren, falls die Beschriftung der einen Transition eine Kommunikationsbeschriftung enthält, zu der in der Beschriftung der anderen Transition deren Konjugierte vorhanden ist. Als Ergebnis der Synchronisation wird eine neue Transition in das Netz eingefügt. Die Kanten der neuen Transition von bzw. zu Stellen werden von den beiden synchronisierenden Transitionen übernommen, die ihrerseits unverändert im Netz verbleiben. Die eingefügte Transition erhält als Beschriftung eine Aktionsmenge, die sich aus der Vereinigung der beiden Ausgangstran-

[4] Die im nächsten Kapitel beschriebene automatische Übersetzung von Programmen in Petri–Boxen garantiert diese Eigenschaft aufgrund der Syntax der verwendeten Programm–Notation. Eine Befehlsfolge der Art < a:= 1> [] **do** ... **od** ist dort nicht erlaubt.



(a) $B^1 = B_{\widehat{a}} \parallel B_a; B_b \parallel B_{\{\widehat{b},\widehat{c}\}}$ (b) $B^2 = B^1$ **sy** a

Abbildung 4.8: Beispiel für die Anwendung der Synchronisation.

sitionen ergibt, abzüglich des an der Synchronisation beteiligten Paares von Kommunikationsbeschriftungen.

Definition 4.17 *Synchronisation einer Box bezüglich einer Kommunikationsbeschriftung*

Seien $B = (S, T, F, \lambda)$ eine Box und $a \in \mathcal{L}$ eine Kommunikationsbeschriftung. Die *Synchronisation* von B bezüglich a ist definiert als B **sy** $a = B \boxplus (U_a, \lambda_{U_a})$ mit

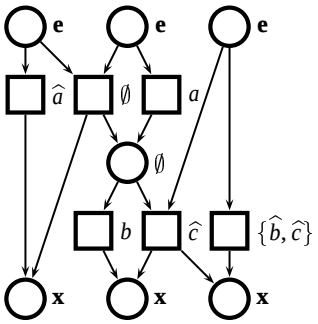
$$U_a = \{\, \{t, t'\} \in T \boxtimes T \mid a \in \lambda(t) \wedge \widehat{a} \in \lambda(t') \,\}$$

und

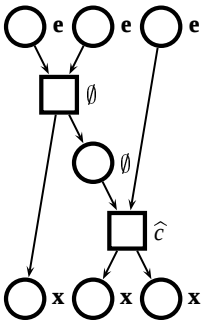
$$\lambda_{U_a} = \{\, \{t, t'\} \mapsto (\lambda(t) \cup \lambda(t')) \setminus \{a, \widehat{a}\} \,\} \mid \{t, t'\} \in U_a \,\}.$$

■4.17

In Abbildung 4.8 (b) und (c) sind Beispiele für die Synchronisation gezeigt. Würde die Box B^3 anschließend bezüglich c synchronisiert werden, so bliebe sie unverändert; zwar besitzen mehrere Transitionen den Aktionsnamen $\widehat{c}$ in ihrer Beschriftung, jedoch ist keine Transition des Netzes mit c beschriftet.
Eine Synchronisation über Aktionsnamen innerhalb derselben sequenziellen Komponente ist mit diesen Definitionen zwar technisch möglich, macht jedoch wenig Sinn, da Transitionen erzeugt werden, die niemals aktiviert werden können. Innerhalb von Komponenten wird das Synchronisieren über Sprungmarken die Zusammenführung von Sprüngen und Sprungzielen ermöglichen und ist somit beabsichtigt. Ein Synchronisieren von Sprungmarken über sequenzielle Komponenten hinweg wiederum ist zu vermeiden; anschaulich könnten auf diese Weise sequenzielle Prozesse in die Rümpfe von anderen Prozessen springen, was sicher nicht beabsichtigt sein kann. Die Syntax der Box–Ausdrücke stellt diese Eigenschaft sicher, indem Sprungmarken lokal für je-



(a) $B^3 = B^2$ **sy** b



(b) $B^4 = (B^3$ **rs** $a)$ **rs** $b = [\{a,b\} : B^1]$

Abbildung 4.9: Beispiele für Synchronisation und Restriktion (Forsetzung).

de sequenzielle Komponente synchronisiert werden, bevor die Komponenten parallel komponiert werden.
Die hier vorgestellte Synchronisationsoperation ist vergleichbar mit derjenigen kommunizierender Automaten, und damit wesentlich einfacher als die in [BDH92] definierte, welche auf Multimengen basiert und dadurch Kantengewichte größer als eins und sogar unendliche Netze produzieren kann.

Als weiterer unärer Operator entfernt die Restriktion alle Transitionen aus einem Box–Netz, deren Aktionsmenge eine bestimmte Kommunikationsbeschriftung – konjugiert oder unkonjugiert – enthält.

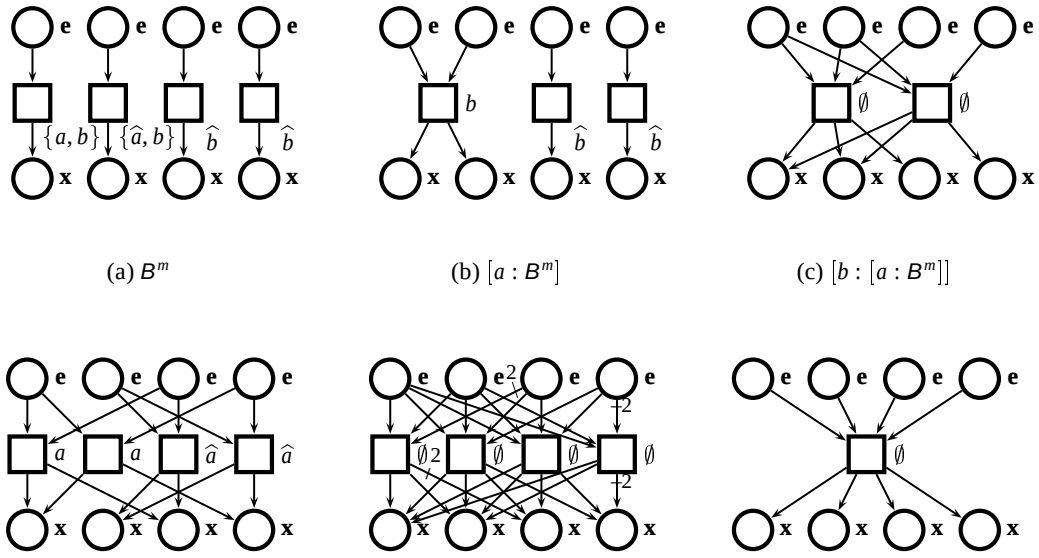
Definition 4.18 *Restriktion einer Box bezüglich einer Kommunikationsbeschriftung*

Seien $B = (\, S,\, T,\, F,\, \lambda\,)$ eine Box und $a \in \mathcal{L}$ eine Kommunikationsbeschriftung. Sei weiterhin

$$T^a \;=\; \{\, t \in T \mid \lambda(t) \cap \{\, a, \widehat{a}\,\} \neq \emptyset\,\}$$

die Menge der a oder $\widehat{a}$ in der Beschriftung tragenden Transitionen.
Dann definiert B **rs** $a = B \boxminus T^a$ die *Restriktion* von B bezüglich a. ■4.18

Abbildung 4.9(b) zeigt, in Fortführung des letzten Beispiels, eine Anwendung dieser Operation; sie greift unmittelbar auf das Löschen von Transitionen zurück.
Die Synchronisation mit nachgeschalteter Restriktion über demselben Bezeichner wird Scoping genannt.



(a) B^m (b) $[a : B^m]$ (c) $[b : [a : B^m]]$

(d) $[b : B^m]$ (e) $[a : [b : B^m]]$ (f) $[\{a, b\} : B^m]$

Abbildung 4.10: Beispiel zur Erfordernis von Multimengen.

Notation 4.19 *Scoping einer Box bezüglich einer Kommunikationsbeschriftung*

Sei B eine Box und $a \in \mathcal{L}$ eine Kommunikationsbeschriftung. Das *Scoping* von B bezüglich a ist definiert als $[\, a : B \,] = (\, B \textbf{ sy } a \,) \textbf{ rs } a$. ■4.19

In [BDH92] wird gezeigt, dass alle diese Operationen, Synchronisation, Restriktion und damit Scoping, kommutativ sind, d. h. es gilt insbesondere $[a : [b : B]] = [b : [a : B]]$ für $a, b \in \mathcal{L}$. Aus diesem Grund wird für das Scoping über mehrelementige Aktionsmengen nachfolgend die abkürzende Schreibweise $[\{a, b\} : B]$ verwendet.
Die Kommutativität wird in [BDH92] jedoch für den allgemeinen Fall nachgewiesen, dass Aktionsmengen aus *Multimengen* über Kommunikationsbeschriftungen bestehen. Abb. 4.10 zeigt ein Beispiel für die Notwendigkeit von Multimengen: die Box $B^m = B_{\{a,b\}} \parallel B_{\{\widehat{a},b\}} \parallel B_{\widehat{b}} \parallel B_{\widehat{b}}$, wird in (b) zunächst über a, danach über b synchronisiert, Teilbild (c). Ganz offensichtlich ist diese Box verschieden von derjenigen in (e), die durch Synchronisation von B^m erst über b (d) und dann erst über a entstanden ist. Richtigerweise muss also die linke Transition der Box in (b) mit der Multimenge $\{b, b\}$ beschriftet werden, um bei der anschließenden Synchronisation über b das Verschmelzen mit *beiden* mit $\widehat{b}$ beschrifteten Transitionen zu ermöglichen. Das Ergebnis dieser korrekten Synchronisation ist in Teilbild (f) dargestellt.

Auf den ersten Blick ist vielleicht nicht ganz einsichtig, weshalb die – etwas unübersichtliche – Box $[a : [b : B^m]]$ in Teilbild (e) ebenfalls eine richtig synchronisierte Box darstellt. Es wird später nachgewiesen werden, dass die mit Hilfe der Operationen auf Boxen erzeugten Netzsysteme stets 1–sicher sind. Damit können über Kanten, die ein Gewicht größer als eins besitzen, keine Marken wandern; die daran hängenden Transitionen sind also nie aktiviert. In der Box aus (e) können somit wegen der Kantengewichte zwei die ganz linke und ganz rechte Transition mitsamt der verbundenen Kanten aus dem Netz genommen werden. Übrig bleiben die beiden mittleren Transitionen, die die gleiche Beschriftung sowie denselben Vor– wie Nachbereich besitzen. Wann immer also die eine Transition aktiviert ist, ist es auch die andere – und umgekehrt. Ein anschließendes Schalten einer der beiden Transitionen führt zu derselben Nachfolgemarkierung wie sie die jeweils andere produzierte. Bezogen auf das Ausgangsnetz B^m ist es letzlich gleichgültig, welche mit b beschriftete Transition mit welcher mit $\widehat{b}$ beschrifteten synchronisiert. Somit kann eine dieser Transitionen ebenfalls aus dem Netz entfernt werden, ohne am Verhalten etwas zu ändern. Die Behandlung solcher Knotenduplikate wird später in Abschnitt 4.6 eingehender diskutiert.
Außerhalb der Konstruktion der Synchronisation (bzw. des Scopings) werden Multimengen im vorgestellten Box–Kalkül nicht benötigt, deshalb ist die Synchronisation hier etwas einfacher (und intuitiver) definiert als in [BDH92], und Transitionsbeschriftungen werden auf einfache Mengen beschränkt. Bei der praktischen Umsetzung wird die Kommutativität ausgenutzt, das Scoping wird für jede Transition des Kontrollflusses getrennt in einem einzigen Schritt gleichzeitig für alle in der Beschriftung enthaltenen Aktionsnamen ausgeführt. Das ist einfach realisierbar, weil innerhalb des Kontrollflusses eines regelmäßig nur unkonjugierte Aktionsnamen verwendet werden. Die dazu konjugierten Namen sind in den die Variablen repräsentierenden Datenfluss–Prozessen zu finden, die später beschrieben werden.

4.4.4 Sprünge

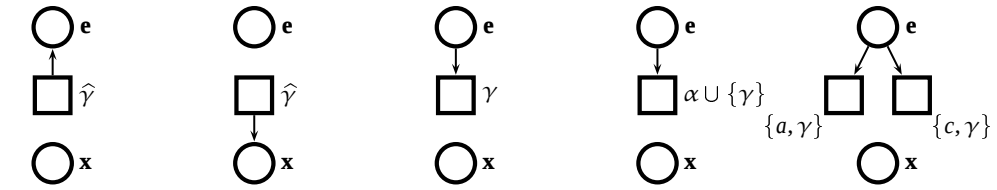
Zur Beschreibung von *Sprüngen* werden zwei Operatoren benötigt: einer für Sprungziele und ein dazu komplementärer für die Sprunganweisungen.

Definition 4.20 *Sprungzieloperator*

Seien $\gamma \in \mathcal{S}$ eine Sprungmarke und $z \in \{\mathbf{e}, \mathbf{x}\}$ eine Stellenbeschriftung. Die Box des *Sprungzieloperators* ist definiert durch

$$[\overset{\gamma}{\rightarrow} z] = (\{s_\mathrm{e}, s_\mathrm{x}\}, \{t\}, \{(t, s_z)\}, \{s_\mathrm{e} \mapsto \mathbf{e}, t \mapsto \{\gamma\}, s_\mathrm{x} \mapsto \mathbf{x}\})$$

■4.20



(a) $[\stackrel{\widehat{\gamma}}{\rightarrow}\ \mathbf{e}]$ (b) $[\stackrel{\widehat{\gamma}}{\rightarrow}\ \mathbf{x}]$ (c) $[B_{\emptyset}\ \stackrel{\gamma}{\rightarrow}]$ (d) $[B_{\alpha}\ \stackrel{\gamma}{\rightarrow}]$ (e) $[B_a [] B_c\ \stackrel{\gamma}{\rightarrow}]$

Abbildung 4.11: Boxen für Sprungziele (a),(b) und Sprünge (c)–(e).

Auf diese Weise werden also zwei Operationen definiert: der Operator $[\stackrel{\gamma}{\rightarrow}\ \mathbf{e}]$ erlaubt den Sprung zurück an die Eingangsstelle einer Box, $[\stackrel{\gamma}{\rightarrow}\ \mathbf{x}]$ ermöglicht den Vorwärtssprung an die Ausgangsstelle der entsprechenden Box, jeweils in Abhängigkeit von der Sprungmarke γ.
In Abbildung 4.11(a) und (b) sind diese beiden Netze dargestellt. Sprungziele müssen in einer sequenziellen Komponente eindeutig sein, daher darf dieselbe Sprungmarke nur maximal einmal pro Komponente in einem Sprungzieloperator verwendet werden.

Das Gegenstück zum Sprungziel, der Ausgangspunkt eines Sprungs, wird wie folgt konstruiert.

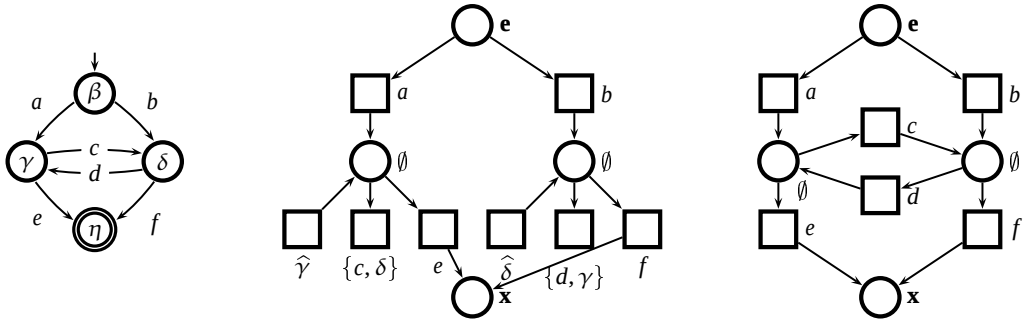
Definition 4.21 *Sprungoperator*

Seien $B = (S, T, F, \lambda)$ eine Box und $\gamma \in \mathcal{S}$ eine Sprungaktion. Die Box des *Sprungoperators* ist definiert durch $[B\ \stackrel{\gamma}{\rightarrow}] = (S, T, F \setminus ({}^\bullet(B^\bullet) \times B^\bullet), \lambda')$ mit

$$\lambda'(x) = \begin{cases} \lambda(x) \cup \{\gamma\} & \text{falls } x \in {}^\bullet(B^\bullet), \\ \lambda(x) & \text{sonst} \end{cases}$$

für $x \in S \cup T$. ■4.21

Anschaulich werden die zu Ausgangsstellen führenden Kanten entfernt und die Beschriftungen der Transitionsknoten dieser Kanten um die jeweilige Sprungmarke erweitert. Nur für den Fall, dass die Box B terminiert, kann der Sprung somit ausgeführt werden. Die Zusammenführung von Sprüngen und Sprungzielen erfolgt über die Sprungmarken durch Scoping. Sprünge über sequenzielle Komponenten hinweg sind dabei nicht möglich und werden ausgeschlossen.
In Abbildung 4.11(c)-(e) sind Beispiele für Sprung–Boxen dargestellt: Teilbild (c) zeigt die Box eines unbedingten Sprungs, die Box in (d) ist bedingt durch die Aktionsmenge



(a) Automat (b) Box B' (c) Box $B'' = [\{\gamma, \delta\} : B']$

Abbildung 4.12: Beispiel für die Anwendung von Sprüngen.

α. Die in Teilbild (e) dargestellte Box illustriert, in welcher Form eine nicht–elementare Boxen, hier die Box einer einfachen Alternative, durch den Sprungoperator verändert werden, indem alle Kanten zur Ausganggsstelle entfernt werden.
Allgemein sind Sprünge – aus Gründen der Übersichtlichkeit und Wartbarkeit – in der Informatik bei Programmiersprachen ein unbeliebtes syntaktisches Konstrukt[5], ihre Unterstützung in Prozess–Algebren ist zumindest unüblich. Hier soll nun nicht das Für und Wider von Sprüngen diskutiert werden, sondern kurz motiviert werden, warum und wann der (wohlüberlegte und sparsame) Einsatz von Sprüngen sinnvoll und erforderlich sein kann.
Dazu zeigt Abbildung 4.12(a) einen einfachen endlichen Automaten mit vier Zuständen und sechs Zustandsübergängen. Die Zustände γ und δ bilden über die Übergänge *c* und *d* eine Schleife, wobei beide Zustände weitere Ein– und Ausgangskanten besitzen; eine solche Konstruktion lässt sich strukturell nicht mit den bislang vorgestellten Operationen beschreiben[6].

[5] Vielfach sind dies Reminiszenzen aus Zeiten des berüchtigten „Spaghetti–Codes“ von Sprachen wie FORTRAN oder BASIC, [Dij68]. Diese Argumentation konnte jedoch weitgehend entkräftet worden, [Knu74, Rub87]. Moderne „wohlstrukturierte“ Sprachen wie JAVA unterbinden die Verwendung von `goto` mittlerweile absichtlich; allerdings werden Sprung–Direktiven wie `break`, `continue` und `return` auch weiterhin unterstützt, die teilweise sogar, mit Anweisungen wie `outer` verknüpft, bezeichnet verwendet werden können.

[6] Eine nicht–strukturelle Möglichkeit der Darstellung solcher Schleifen bietet die Simulation eines endlichen Automaten mit nur einem Zustand. Eine zusätzlich einzuführende Programmvariable speichert den jeweils aktuellen Zustand des simulierten Automaten. Eine solche Lösung wird in [PEP98] zur Darstellung beliebiger, nicht in B(PN)^2 bzw. der Box–Algebra aus [BDH92] ausdrückbarer Automatenstrukturen verwendet. Die dort implementierte Transformation ist allerdings nicht einbettbar, d. h., es muss stets der

Eine Lösung der Art $E = a; X[]b; Y$ mit $X = c; Y[]e$ und $Y = d; X[]f$ (CCS–Notation) erfordert die Operation Rekursion. Allerdings führte dies bei der Übersetzung in eine Box zu einem unendlichen Netz.
Mit Hilfe von Sprüngen lässt sich dieser Automat über den folgenden – noch nicht synchronisierten – Ausdruck modellieren:

$$B' = B_a; ([\xrightarrow{\widehat{\gamma}} \mathbf{e}][][B_c \xrightarrow{\delta}][]B_e) \,[]\, B_b; ([\xrightarrow{\widehat{\delta}} \mathbf{e}][][B_d \xrightarrow{\gamma}][]B_f)$$

Die zugehörige Box zeigt Abbildung 4.12(b). Das Netz nach Synchronisation über die Sprungmarken γ und δ ist daneben in Teilbild (c) dargestellt. Die Verwandtschaft zum ursprünglichen Automaten in (a) ist deutlich zu erkennen; die Übersetzung liefert eine minimale Petri–Netz–Darstellung.
Solche wechselseitigen Verweise wie im obigen Beispiel sind keineswegs unüblich bei der Spezifikation von Prozessen. Das nächste Kapitel wird aufzeigen, dass derartige Strukturen sogar im Box–Kalkül selbst verwendet werden – bei der Darstellung des Datenflusses durch sogenannte Datenboxen.

4.5 Semantik der Box–Ausdrücke

Mit Hilfe der in den letzten Abschnitten beschriebenen Operationen auf Boxen wird nun eine kompositionelle Netz–Semantik für Box–Ausdrücke definiert: jedem Term E der Box–Algebra aus Abschnitt 4.1 wird eine Box $\mathsf{DBox}(\mathsf{E})$ zugeordnet.

Definition 4.22 *Semantik von Box–Ausdrücken*

Seien $\mathsf{E}_1, \mathsf{E}_2 \in \mathsf{DBEXPR}$ Box–Ausdrücke, $a \in \mathcal{A}$ ein Aktionsname, $\gamma \in \mathcal{S}$ eine Sprungaktion, $\alpha \subseteq \mathcal{A}$ eine Aktionsmenge und $z \in \{\mathbf{e}, \mathbf{x}\}$ eine Stellenbeschriftung. Dann wird der semantische Homomorphismus $\mathsf{DBox} : \mathsf{DBEXPR} \to \mathsf{DBOX}$ definiert als

$$\begin{array}{lcl}
\mathsf{DBox}([a : \mathsf{E}_1]) & = & [a : \mathsf{DBox}(\mathsf{E}_1)] \\
\mathsf{DBox}(\mathsf{E}_1 \parallel \mathsf{E}_2) & = & \mathsf{DBox}(\mathsf{E}_1) \parallel \mathsf{DBox}(\mathsf{E}_2) \\
\mathsf{DBox}([\gamma : \mathsf{E}_1]) & = & [\gamma : \mathsf{DBox}(\mathsf{E}_1)] \\
\mathsf{DBox}(\alpha) & = & B_\alpha \\
\mathsf{DBox}(\mathbf{stop}) & = & B_{\mathbf{stop}} \\
\mathsf{DBox}(\mathsf{E}_1 \,;\, \mathsf{E}_2) & = & \mathsf{DBox}(\mathsf{E}_1) \,;\, \mathsf{DBox}(\mathsf{E}_2) \\
\mathsf{DBox}(\mathsf{E}_1 \,[]\, \mathsf{E}_2) & = & \mathsf{DBox}(\mathsf{E}_1) \,[]\, \mathsf{DBox}(\mathsf{E}_2) \\
\mathsf{DBox}([\mathsf{E}_1 * \mathsf{E}_2]) & = & [\mathsf{DBox}(\mathsf{E}_1) * \mathsf{DBox}(\mathsf{E}_2)] \\
\mathsf{DBox}([\mathsf{E}_1 \xrightarrow{\gamma}]) & = & [\mathsf{DBox}(\mathsf{E}_1) \xrightarrow{\gamma}] \\
\mathsf{DBox}([\xrightarrow{\gamma} z]) & = & [\xrightarrow{\gamma} z]
\end{array}$$

■4.22

gesamte Ausdruck als Automat simuliert werden. Das obige Beispiel wird z. B. übersetzt in eine Petri–Box der Größe 27 Stellen und 20 Transitionen.

Einige Eigenschaften dieser Netz–Semantik werden im Folgenden hergeleitet.

Satz 4.23 *Sequenzielle Box–Ausdrücke beschreiben S–Systeme*

Sei $K \in DBEXPR$ ein sequenzieller Box–Ausdruck. Dann ist der markierte Netzgraph von $DBox(K)$ ein S–System.

Beweis: Der Ausdruck K ist nach Def. 4.2 und wegen der Kommutativität des Scopings auflösbar in $\mathsf{K} = [\mathcal{S}_\mathsf{S} : \mathsf{S}]$, wobei S eine (unsynchronisierte) sequenzielle Komponente beschreibt und $\mathcal{S}_\mathsf{S} \subseteq \mathcal{S}$ die in S vorkommenden Sprungmarken bezeichnet. Sei weiterhin $\mathsf{DBox}(\mathsf{K}) = (S, T, F, \lambda)$.

(a) Es gilt: $\forall t \in T : |{}^\bullet t| = 1 = |t^\bullet|$.

Elementare Boxen erzeugen lediglich Transitionen mit einelementigen Vor– und Nachbereichen. Die Operationen Alternative, Sequenz und Iteration erhalten diese Eigenschaft: jeder Stellenaddition schließt sich ein Entfernen der an der Stellenmultiplikation beteiligten Knoten an, die Kanten zu bzw. von Transitionen werden anschaulich nur „umgebogen“ auf die neue Stelle.

Einzig Sprungziel– und Sprungoperatoren können leere Vor– bzw. Nachbereiche bei Transitionen erzeugen, dies jedoch stets komplementär für konjugierte Sprungmarken. Das anschließende Scoping verschmilzt diese Transitionen, so dass auch sie einelementige Vor– und Nachbereiche besitzen. Offene Sprünge, d. h. solche, bei denen die konjugierte Sprungmarke fehlt, werden durch die beim Scoping durchgeführte Restriktion aus dem Netz entfernt.

(b) Die Startmarkierung von $\mathsf{DBox}(\mathsf{K})$ ist regulär.

Die Konstruktion von elementaren Boxen und Sprungzielen garantiert, dass ihre Netze genau eine Eingangsstelle besitzen. Die Operationen Alternative und Iteration erhalten diese Eigenschaft, da sie über die Stellenmultiplikation und –addition genau eine Eingangsstelle in das Netz einfügen und alle bisherigen Eingangsstellen entfernen. Bei der Sequenz wird die Eingangsstelle der ersten Box übernommen, die Eingangsstelle der zweiten Box verschmilzt mit der Ausgangsstelle der ersten. Der Sprungoperator hinterlässt Eingangsstellen unverändert. Über strukturelle Induktion über S folgt somit unmittelbar $|{}^\bullet\mathsf{DBox}(\mathsf{S})| = 1$. Da das Scoping Stellenknoten nicht manipuliert, gilt ebenfalls $|{}^\bullet\mathsf{DBox}(\mathsf{K})| = 1$. Die Anfangsmarkierung von $\mathsf{DBox}(\mathsf{K})$ ist somit regulär. (Analog lässt sich herleiten, dass $\mathsf{DBox}(\mathsf{K})$ genau eine Ausgangsstelle besitzt.)

■4.23

Die disjunkte Vereinigung von S–Systemen liefert wieder ein S–System. Mit Hilfe der Definition des Scopings ergibt sich unmittelbar der folgende Zusammenhang.

Lemma 4.24 *Petri–Boxen sind SND–Systeme*

Sei $E \in DBEXPR$ ein Box–Ausdruck. Dann beschreibt $DBox(E)$ ein SND–System.

■*4.24*

Damit sind Petri–Boxen insbesondere 1–sicher. Daneben lassen sich die in Kapitel 3.5.3 vorgestellten Vereinfachungen bezüglich der Präfix–Generierung anwenden, insbesondere der weitgehende Verzicht auf die explizite Berechnung lokaler Konfigurationen.

4.6 Vergleich der Box–Kalküle

Nach der Diskussion des im Rahmen dieser Arbeit verwendeten (abgewandelten) Box–Kalküls soll ein kleiner Vergleich mit dem Kalkül aus [BDH92] dieses Kapitel beschließen. Zusammengefasst betreffen die wichtigsten vorgenommenen Änderungen die folgenden Punkte:

- Zulassen des Paralleloperators nur auf oberster Ebene.
- Weglassen einiger entbehrlicher Konstrukte (Verfeinerung, Rekursion, Umbenennung).
- Einführen neuer Operatoren zur Modellierung von Sprüngen.
- Vereinfachter Schleifenoperator (Fortfall eines zwingenden Initialisierungsteils).[7]
- Transitionsbeschriftungen bestehen lediglich aus einfachen Mengen anstatt Multimengen von Kommunikationsbeschriftungen.

Diese Änderungen sind vorgenommen worden, um eine einfache Netz–Semantik zu erhalten, die Netze geringer Größe konstruiert. Daneben wird auf diese Weise der Erhalt der SND–Eigenschaft sichergestellt; diese Eigenschaft stand beim Design zunächst gar nicht im Vordergrund – sie kann sogar leicht rückgängig gemacht werden, da der Box–Kalkül in jedem Fall die Konstruktion 1–sicherer Netze garantiert, für die es eine totale Ordnung für die Präfix–Generierung gibt.
Da sich bei der Suche nach geeigneten adäquaten Ordnungen eine in vielen Fällen vereinfachte Berechnung des Präfixes für SND–Systeme herauskristallisiert hat, bot sich die Einschränkung aauf diese Netzklasse geradezu an. Erweiterungen wie etwa das Sprungkonzept sind ausdrücklich nicht auf das Vorhandensein von SND–Systemen angewiesen sondern sind auch für beliebige 1–sichere Netzsysteme möglich [RM96].

[7] Die hier definierte Netz–Semantik von Schleifen hat zudem, durch das Erlauben von Kanten, die in Eingangsstellen weisen, eine sehr einfache und intuitive Gestalt; für das Schleifenkonstrukt in [BDH92] existieren drei verschiedene Semantiken, die sich in der Größe der entsprechenden Netze unterscheiden.

Ein bislang nicht erwähnter Unterschied betrifft das Zusammenfassen „ähnlich“aussehender Boxen zu Klassen. In [BDH92] wird eine Äquivalenzrelation auf Boxen definiert, die Stellen– bzw. Transitionsduplikate berücksichtigt. Zwei Stellen (bzw. Transitionen) *duplizieren* einander, wenn sie die gleiche Beschriftung tragen und außerdem ihre Vor– und Nachbereiche (inklusive der Kantengewichte) exakt übereinstimmen. Boxen, die sich lediglich durch duplizierende Knoten unterscheiden, werden derselben Äquivalenzklasse zugeordnet. Der kanonische Repräsentant einer jeden Äquivalenzklasse ist diejenige Box, in der es keine Knotenduplikate gibt.
Auf den hier definierten Kalkül lässt sich diese Äquivalenzklassenbildung (zumindest für Stellenknoten) nicht ohne weiteres übertragen, da dadurch eine wichtige Eigenschaft verloren ginge, nämlich die Garantie, SND–Systeme zu erhalten. Würden beispielsweise mehrere isolierte, gleich beschriftete Stellen als Duplikate behandelt und die Boxen unifiziert, verlöre man damit die Zuordnung einer Stelle zu einer bestimmten Netzkomponente.
Die Entfernung von Transitionsduplikaten ist jedoch durchaus vorteilhaft, um die Netzgröße möglichst gering zu halten. Da solche lediglich bei der Konstruktion der Synchronisation entstehen können (und zwar nur bei der Synchronisation über Aktionsnamen, nicht jedoch bei Sprungmarken), ist dieser Sonderfall in den theoretischen Betrachtungen aus Gründen der Vereinfachung nicht gesondert aufgeführt worden, da das Problem bei der hier vorgenommenen Konstruktion des Präfixes keine Rolle spielt.
Die Synchronisation über Aktionsnamen wird hier nämlich erst bei der Berechnung erweiternder Ereignisse des Präfixes vorgenommen, bis zu diesem Zeitpunkt bleiben die Box des Kontrollflusses und die Boxen, die die Werte der Variablenänderungen repräsentieren und somit die konjugierten Aktionsnamen enthalten, unsynchronisiert. Das Vermeiden von Duplikaten von Ereignissen wird auf die gleiche Weise zugesichert, wie in Alg. 3.12 garantiert wird, dass jedes Ereignis nur einmal in das Präfix aufgenommen wird.

Kapitel 5

„Enita non sunt multiplicanda praeter necessitatem.“
(Dinge sollen nicht komplizierter
als notwendig gemacht werden.)
WILLIAM OF OCCAM, 1285–1349

„Man sollte die Dinge so einfach machen wie möglich
– aber nicht einfacher.“
ALBERT EINSTEIN, 1879–1955

Die Modellierungssprache DB(PN)^2

Der im letzten Kapitel beschriebene Box–Kalkül ist bereits in seiner urprünglichen Form mit der Zielsetzung entwickelt worden, eine kompositionelle Netz–Semantik für parallele Hochsprachen zu spezifizieren. In [HHB92] wurde mit ihm eine Netz–Semantik für eine Teilmenge der Sprache *Occam* [May83] beschrieben. Daneben ist auf den Box–Kalkül aufsetzend eine einfache abstrakte Programm–Notation entwickelt worden, genannt B(PN)^2 (ausgeschrieben BPNPN für *Basic Petri Net Programming Notation* [BH93]). Erweiterungen und Änderungen der Sprache B(PN)^2 werden in diesem Kapitel diskutiert, mit dem Ziel, eine weithin universelle Notation zu erhalten, für die, über die beschriebene Box–Algebra, eine für die Präfix–Generierung geeignete Netz–Semantik definiert werden kann.

B(PN)^2 unterstützt grundlegende Programmierparadigmen wie atomare Aktion, Sequenz, Alternative, Prozess–Synchronisation und Schleife. Parallelität von Aktionen ist beliebig verschachtelt zugelassen, also auch innerhalb von Prozessen. Variablen können global oder lokal für einen Block deklariert werden. Darüber hinaus existieren für diese Sprache Erweiterungen in Richtung hierarchischer Konstrukte, beispielsweise durch die Entwicklung eines einfachen Prozedurkonzepts [Jen94].
Interprozess–Kommunikation kann in B(PN)^2 modelliert werden über globale Variablen oder über aus CSP [Hoa78] entlehnte Kanäle, die in FIFO– oder LIFO–Reihenfolge benutzbar sind. Insgesamt ist die Syntax von B(PN)^2 kompakt und übersichtlich, sie eignet sich für ein weites Feld von Modellierungen nebenläufiger Systeme.

B(PN)^2 ist eine der zentralen Eingabesprachen des Werkzeugs PEP (*Programming Environment based on Petri nets*, eingedeutscht *Petri–Netz basierte Entwicklungs– und Programmierumgebung*, [BF95, MRE96, PEP98]). Daneben wird diese Sprache auch vom *Model–Checking–Kit* [SSE99] unterstützt.
Das folgende Programm liefert ein einfaches Beispiel für die B(PN)^2–Syntax. Modelliert sind zwei nebenläufige Prozesse, die auf eine gemeinsame Variable *semaphor* zugreifen können, um über diese das Betreten eines kritischen Abschnitts zu signalisieren.

Programmtext 5.1 *Semaphor–Beispiel in B(PN)^2*

```
1   begin
2     var semaphor: {0..1} init 1;
3     var crit_sect_1: {0..1} init 0;
4     var crit_sect_2: {0..1} init 0;
5     begin
6       do <true> enter
7         <'semaphor > 0 and semaphor' = 'semaphor – 1>;
8         <crit_sect_1' = 1>; <crit_sect_1' = 0>;
9         <semaphor' = 'semaphor + 1>; repeat
10      od
11    end
12    ||
13    begin
14      do <true> enter
15        <'semaphor > 0 and semaphor' = 'semaphor – 1>;
16        <crit_sect_2' = 1>; <crit_sect_2' = 0>;
17        <semaphor' = 'semaphor + 1>; repeat
18      od
19    end
20  end                                                    ■5.1
```

Atomare Aktionen sind bei dieser Notation in spitze Klammern eingebettet. Auf den ersten Blick ungewöhnlich könnten die Apostrophe vor bzw. nach Variablennamen erscheinen. Sie geben an, ob innerhalb des Ausdrucks der Wert einer Variablen *vor* (bei vorangestelltem Apostroph) oder *nach* (Apostroph nachgestellt) Ausführung der entsprechenden Aktion betrachtet werden soll. So beschreibt etwa der Ausdruck in Zeile 7, *semaphor*' = '*semaphor*– 1, ein Dekrementieren der Variablen *semaphor*.

Neben den oben aufgezählten Merkmalen besitzt B(PN)^2 auch (natürliche) Beschränkungen. Beispielsweise werden als Datentypen lediglich einfache skalare Typen über einem Ausschnitt der ganzen Zahlen unterstützt. Alle anderen Typen müssen auf diese Darstellung abgebildet werden.
In der Syntax von B(PN)^2 werden unterschiedliche Konzepte durch die gleichen syntaktischen Konstrukte beschrieben, beispielsweise werden Schleifen und Fallunterscheidung beide durch ein **do**–**od**–Konstrukt ausgedrückt, und asynchrone wie synchrone

Kommunikation beide über Kanäle (unterschiedlicher Kapazität). Dieses Vorgehen ermöglicht eine übersichtliche und einfache Semantik. Probleme entstehen jedoch dann, wenn man Programmierkonzepte verwenden möchte, die nicht direkt in der Syntax enthalten sind.
Durch die Zusammenlegung von Schleifen und Fallunterscheidungen in einem Konstrukt ist ein bedingtes Verlassen oder Wiederholen eines Schleifenrumpfes, das in Sprachen wie C oder Pascal durch **break** bzw. **continue** erfolgt, in B(PN)^2 nicht direkt möglich. Einzig die Einführung einer zusätzlichen Variablen erlaubt dies; neben der damit verbundenen Unübersichtlichkeit hat dies auch Auswirkungen auf die Größe des erzeugten Netzes.
Andere Probleme, nämlich die im letzten Kapitel im Zusammenhang mit dem Fehlen von Sprüngen angesprochenen, treten auf bei der Modellierung synchronisierender höherer Automaten (darunter werden solche verstanden, bei denen die Zustandsübergänge mit Termen über Variablen beschriftet sind); diese finden in industriellen Anwendungen häufig Verwendung. B(PN)^2 ist für einen solchen Einsatz ungeeignet, da sich nicht alle möglichen Zustandsübergänge strukturell beschreiben lassen.

Der Nutzen einer abstrakten Beschreibungssprache für allgemeine Automatenstrukturen hat sich u. a. bei der Bearbeitung einer konkreten Fallstudie [Sán96] gezeigt. Leider erfordern die Eingabe und das spätere Verändern von Automatendarstellungen oft spezielle graphische Editoren. Obwohl diese Editoren inzwischen häufig verschiedene Grade der Hierarchisierung und Abstraktion zulassen, bleibt die graphische Darstellung von Systemen, insbesondere wenn sie eine bestimmte Größe überschreiten, recht unübersichtlich; Ausdrucke nehmen oft die Größe von Postern ein oder erinnern gar an „Tapeten“. Eine Programm–Notation bietet den Vorteil, mit einem beliebigen Editor schnell veränderbar und leicht reproduzierbar zu sein.

Das vorliegende Kapitel versucht sich einiger der oben genannten Punkte anzunehmen. Dazu wird eine Variante von B(PN)^2 entworfen, die auf der einen Seite etwas Ballast (bezüglich der ins Auge gefassten Präfix–Generierung) abwirft, auf der anderen Seite neue, flexible Konstrukte einführt. Teile dieser Änderungen, wie etwa die Beschränkung des Parallel–Operators auf die oberste Ebene und die Vereinfachung von Schleifen, wurden unter dem Namen *Small Box Calculus* in [EB96] beschrieben; Verbesserungen bezüglich der Modellierung von Automatenstrukturen sind in [RM96] vorgeschlagen worden.
Das modifizierte B(PN)^2 wird im Folgenden *Distributed* B(PN)^2 genannt, kurz DB(PN)^2. Ein wichtiges Ziel beim Design dieser Sprache ist gewesen, in einem gewissen Sinne minimale Netze zu einem Programm zu erzeugen, um die anschliessende Verifikation zu vereinfachen bzw. überhaupt erst zu ermöglichen.
Im weiteren Verlauf wird zunächst die Syntax von DB(PN)^2–Programmen vorgestellt, Unterschiede zu B(PN)^2 werden dabei nur am Rande aufzeigt. Im Anschluss daran

wird die (Netz–)Semantik von DB(PN)^2 beschrieben, die auf den im letzten Kapitel eingeführten Operationen auf Petri–Boxen (für den Kontrollfluss) fußt. Während bislang hauptsächlich der Kontrollfluss betrachtet worden ist, wird hier nun auch die Netz–Semantik von Variablen vorgestellt.

5.1 Syntax von DB(PN)^2

Ein DB(PN)^2–Programm besteht aus einer (endlichen) Zahl nebenläufiger Prozesse. Jeder dieser Prozesse stellt wiederum ein sequenzielles Programm dar, das aufgebaut ist aus atomaren Aktionen, die über Konstrukte wie Sequenz, Alternative und Schleife miteinander kombiniert werden. Den Prozessen vorangestellt ist ein gemeinsamer Daklarationsteil, in dem Variablen und Konstanten definiert werden.
Die folgenden Abschnitte präsentieren die Syntax von DB(PN)^2–Programmen, aufgeteilt in den Kontrollfluss, den Deklarationsteil sowie die Struktur der atomaren Aktionen. In diesem Kapitel werden nur die zum konzeptuellen Verständnis essenziellen Sprachbestandteile diskutiert, die vollständige Syntax von DB(PN)^2 (beispielsweise das Aussehen von Kommentaren) ist in Anhang A.1 aufgeführt. Dort sind auch Hinweise zur konkreten maschinenlesbaren Syntax enthalten, die etwas von der hier verwendeten abweicht.

5.1.1 Kontrollfluss

Die Syntax von DB(PN)^2 wird in den folgenden Abschnitten als kontextfreie Grammatik spezifiziert. Fettgedruckte Wörter und Zeichen stehen für Terminale, kursiv gesetzte Bezeichner sind die nicht–terminalen Symbole der Grammatik.
DB(PN)^2–Programme besitzen die folgende Struktur:

program	::=	*decl* **;** *par-block* | *par-block*
par-block	::=	*seq-block* | *seq-block* **‖** *par-block*
seq-block	::=	*command*
command	::=	**begin** *command* **end** | *action* | **skip** | **abort** | *jump*
	|	*command* **;** *command* | **if** *alt-set* **fi** | **do** *alt-set* **od**
alt-set	::=	*action* | *action jump* | *action* **;** *command* | *alt-set* **[]** *alt-set*
action	::=	**<** *expr* **>** | **<** *label* **:** *expr* **>**
jump	::=	**goto** *label* | **break** | **continue**
label	::=	*identifier*

Der Aufbau von Deklarationen *decl* wird in Abschnitt 5.1.2 behandelt, und Ausdrücke *expr* anschließend in Abschnitt 5.1.3.
Ein erstes DB(PN)^2–Programm ist nachfolgend angegeben. Die Schlüsselwörter sind wieder durch Fettdruck hervorgehoben.

Programmtext 5.2 *Der Mutex–Algorithmus nach Knuth [Ray86] in DB(PN)²*

```
1   var c1,c2: card init 0;
2   var k: card init 1;
3     do <true>
4       <l1: c1:= 1>;
5       if <l2: k = 1> goto l3 [] <k# 1> fi; /* # steht für ≠ */
6       if <c2# 0> goto l2 [] <c2 = 0> fi;
7       <l3: c1:= 2>;
8       if <c2 = 2> goto l1 [] <c2# 2> fi;
9       <k:= 1>;
10        /* Kritischer Abschnitt 1 */
11      <k:= 2>;
12      <c1:= 0>
13    od
14  ||
15    do <true>
16      <m1: c2:= 1>;
17      if <m2: k = 2> goto m3 [] <k# 2> fi;
18      if <c1# 0> goto m2 [] <c1 = 0> fi;
19      <m3: c2:= 2>;
20      if <c1 = 2> goto m1 [] <c1# 2> fi;
21      <k:= 2>;
22        /* Kritischer Abschnitt 2 */
23      <k:= 1>;
24      <c2:= 0>
25    od                                                    ■5.2
```

Auch ohne Kenntnis der noch nicht beschriebenen Bestandteile sollte die Funktionsweise dieses Programms leicht nachvollziehbar sein; die Syntax orientiert sich weitgehend an geläufigen Sprachen. Einige Erläuterungen zur Bedeutung der Kontrollfluss–Direktiven von DB(PN)²:

- Jede Alternative eines **if**– bzw. **do**–Konstrukts wird eingeleitet von einer atomaren Aktion, dem sogenannten Wächter (engl. *guard*), ähnlich wie beim `alt`–Befehl von *Occam*.

  Kann bei Eintritt eines **if** oder **do** mehr als ein Wächter zu **true** ausgewertet werden, wird einer von ihnen nicht–deterministisch ausgewählt und die entsprechende Alternative betreten.

- Ist für ein **if** oder **do** unter den Wächtern keiner zu **true** auswertbar, so verharrt der Prozess am Beginn dieses Befehls, bis (vielleicht) einer von ihnen erfüllt wird.

  Es gibt kein Äquivalent für das aus anderen Sprachen bekannte **else**. Der nicht von anderen Wächtern abgedeckte Fall muss explizit als eigene Alternative mit

einem entsprechenden Wächter spezifiziert werden. Dabei muss dem Wächter kein weiteres Kommando folgen, wie das obige Programmbeispiel an mehreren Stellen zeigt.

- Innerhalb eines **do**–**od**–Konstrukts bewirkt der Befehl **break** einen Abbruch, **continue** einen Wiedereintritt der Schleife. Bei ineinander verschachtelten Schleifen beziehen sich diese Befehle auf des jeweils innerste Schleifenkonstrukt.

  In einer **do**–Schleife wird jede Alternative implizit mit einem **continue** beendet.[1]

- Ein außerhalb einer Schleife platziertes **break** gestattet darüber hinaus das (bedingte) Abbrechen eines Prozesses, verbunden mit einem Sprung ans Ende des jeweiligen Prozesses.[2] Im Gegensatz dazu erlaubt der **abort**–Befehl das Anhalten eines Prozesses; dieses geht unmittelbar einher mit einer Verklemmung des jeweiligen Prozesses.

- In einem alternativen Zweig (*alt-set*) sind die syntaktisch verschiedenen Befehlsfolgen *action jump* und *action* **;** *jump* auch semantisch zu unterscheiden: der erste Befehl beschreibt einen bedingten Sprung (der ununterbrechbar in einem Schritt ausgeführt wird), der zweite hingegen einen Wächter gefolgt von einem unbedingten Sprung (zwischen diesen Befehlen kann eine Aktion eines nebenläufigen Prozesses zur Ausführung gelangen).

- Sprungziele (*label* **:**) werden innerhalb atomarer Aktionen spezifiziert[3]. Sprünge sind dabei nur innerhalb einzelner sequenzieller Prozesse möglich, nicht jedoch Prozess–übergreifend. Derselbe Bezeichner für ein Sprungziel kann daher in verschiedenen Prozessen verwendet werden, auch wenn dies aus Gründen der Lesbarkeit nur in Ausnahmefällen praktiziert werden sollte.

5.1.2 Datentypen und Deklarationsteil

Variablen können in DB(PN)^2 nur global, Prozess–übergreifend deklariert werden. Ihre Definition erfolgt zu Beginn des Programms, zusammen mit den Konstanten, im Deklarationsteil. Die folgende Grammatik beschreibt dessen Aufbau.

---

1 In B(PN)^2–Programmen muss *jede* Alternative eines **do**–**od**–Konstrukts explizit abgeschlossen werden entweder mit **repeat** (Wiedereintritt; entspricht **continue**) oder mit **exit** (Verlassen der Schleife; wie **break**).

2 Nicht erlaubt hingegen ist ein **continue** außerhalb von Schleifen.

3 Der Grund für diese auf den ersten Blick ungewohnte Schreibweise ist die in Kapitel 4.4.4 beschriebene Netz–Semantik von Sprüngen, die auf die Synchronisation von Transitionen zurückgeht.

decl	::=	**var** *ident-list* **:** *type-set init*
	|	**var** *ident-list* **:** **array** *number* **of** *type-set init*
	|	**var** *ident-list* **:** **chan** *capacity* **of** *type-set*
	|	**var** *ident-list* **:** **stack** *capacity* **of** *type-set*
	|	**const** *ident-list* **=** *constant*
	|	*decl* **;** *decl*
ident-list	::=	*identifier* | *ident-list* **,** *identifier*
type-set	::=	*type* | *type* **x** *type*
capacity	::=	*number* | ω
type	::=	*dim-range* | **{** *range* **}** | **card** | **int** | **bool**
range	::=	*sub-range* | *range* **,** *sub-range*
sub-range	::=	*constant* | *constant* **..** *constant*
init	::=	**init** *constant* | **init (** *const-list* **)** | **init [** *const-list* **]** | ϵ
const-list	::=	*constant* | *const-list* **,** *constant*

Jede Variable darf nur einmal deklariert werden. Die Syntax von Bezeichnern und numerischen Konstanten entspricht derjenigen, die in vertrauten Hochsprachen verwendet wird; ihre exakte Definition für maschinenlesbare Programme ist im Anhang erläutert.
Als vordefinierte skalare Datentypen unterstützt DB(PN)^2 Boolesche Wahrheitswerte, **bool** $=\{$**false** , **true** $\}$, sowie die Zahlenbereiche **int** $\subset \mathbb{Z}$ und **card** $\subset \mathbb{N}$.[4] Es können auch beliebige Teilmengen der skalaren Typen durch Aufzählung bestimmt werden. Datentypen zur Aufnahme einzelner Symbole oder von Zeichenketten sind nicht vorhanden.
Variablen müssen in DB(PN)^2 stets mit einem konkreten Wert initialisiert werden. Im Rahmen der Verifikation wird auf diese Weise das erzeugte Netzsystem möglichst klein gehalten. Wird die explizite Initialisierung einer Variablen weggelassen, so wird sie mit dem Wert 0 bzw. **false** vorbelegt, bei Aufzählungstypen mit dem jeweils minimalen Element ihres Definitionsbereichs.

Einfache Datentypen können zusammengefasst werden mit Hilfe von eindimensionalen Feldern (**array**) bzw. Tupeln (entsprechen Verbundtypen zur Aufnahme zweier frei wählbarer skalarer Datentypen). Der erste Eintrag eines Feldes trägt den Index 0, ebenso wird die erste Komponente eines Tupels mit 0 angesprochen, die zweite mit 1. Beispiele für diese Datentypen liefern die folgenden Deklarationen:

var *vektor*: **array** 5 **of int init [** $-1, 0, 1, 2, 3$ **] ;**
var *tup*: **{** 3**..**7**,** 11**}** **x** **bool** **;**

[4] Die Implementation wurde auf einer 32–Bit–Architektur vorgenommen, es ist **card** $= \{0, 1, \ldots, 2^{32}-1\}$ und **int** $= \{-2^{31}, \ldots, 2^{31}-1\}$.

Wegen des fehlenden Initialisierungsteils wird die Variable *tup* hierbei mit dem Wertepaar $(3, \textbf{false})$ vorbesetzt.

Die Deklaration von Kanälen erfolgt über die Schlüsselwörter **chan** (definiert einen FIFO–Kanal, d. h., der zuerst auf den Kanal geschriebene Wert wird auch als erstes wieder ausgelesen) und **stack** (für einen LIFO–Kanal; Lesen von Werten erfolgt in der umgekehrten Reihenfolge wie das Schreiben). Für jeden Kanal kann seine maximale Aufnahmekapazität festgelegt werden; dabei dient die Kapazität 0 zur Modellierung von synchroner Kommunikation, der Wert ω kennzeichnet unbeschränkte, (potenziell) unendliche[5] Kapazität. Kanäle können nicht initialisiert werden, standardmäßig sind sie zu Programmbeginn leer.

Neben Variablen können im Deklarationsteil globale numerische Konstanten definiert werden, auf diese Weise ist ein einfaches Parametrisieren und Skalieren von Programmen möglich.[6] Beide Deklarationsarten können in beliebiger Reihenfolge vorgenommen werden, Konstanten dürfen ebenfalls in Variablen– oder Konstantendeklarationen verwendet werden, allerdings müssen sie stets vor ihrer jeweils ersten Benutzung definiert worden sein.

Die Implementation, d. h. DB(PN)^2–Parser und Auswertungsfunktionen von Ausdrücken, ist offen gehalten für spätere Erweiterungen durch zusätzliche Datentypen, beispielsweise *n*–Tupel oder Listen.

5.1.3 Atomare Aktionen

Elementare Anweisungen in DB(PN)^2 unterscheiden sich etwas von denen üblicher (imperativer) Programmiersprachen. Sie bestehen aus atomaren Aktionen in Gestalt Boolescher Ausdrücke, die in spitze Klammern eingeschlossen sind. Eine atomare Aktion bildet eine unteilbare, von anderen Prozessen nicht unterbrechbare Einheit. Damit ist beispielsweise eine einfache Beschreibung sogenannter *Test–and–Set*–Befehle möglich, wie etwa im (B(PN)^2–)Semaphor–Programmbeispiel 5.1.

Aus operationaler Sicht kann eine atomare Aktion kann nur dann ausgeführt werden, falls der in Klammern eingeschlossene Ausdruck als Ganzes zu **true** ausgewertet wird. Bezogen auf die später zugrunde gelegte Petri–Netz–Semantik bedeutet dies, dass die entsprechende Transition schalten kann.

Die Syntax elementarer Ausdrücke ist in der folgenden Grammatik zusammengefasst.

[5] Rechnerbedingt gibt es hierbei natürliche Beschränkungen, abhängig von der Größe des verfügbaren Speichers. In Programmtexten wird **omega** für ω geschrieben.

[6] In Anhang A.1 wird als weiteres Strukturmittel der Parametrisierung die Syntax von Makros vorgestellt. Da es sich bei diesen Makros lediglich um eine Textersetzung handelt, die keine eigene Netz–Semantik benötigt, werden sie nicht hier zusammen mit der Sprachbeschreibung eingeführt.

expr	::=	*assignment* | *bool-expr* | *expr* **,** *expr*
assignment	::=	*identifier index* **:=** *rvalue* | *identifier* **!** *rvalue*
bool-expr	::=	**(** *bool-expr* **)** | *bool-simple* | *bool-expr bool-op bool-expr*
	|	**not** *bool-expr* | *num-expr comp-op num-expr*
rvalue	::=	*value* | **(** *value* **,** *value* **)**
value	::=	*bool-expr* | *num-expr*
comp-op	::=	**=** | **#** | **<** | **>** | **<=** | **>=**
bool-op	::=	**and** | **or** | **=** | **#**
num-expr	::=	**(** *num-expr* **)** | *num-simple* | *num-expr num-op num-expr*
	|	**–** *num-expr*
num-op	::=	**+** | **–** | $*$ | **/** | **mod**
bool-simple	::=	*identifier index* | *identifier* **?** | *identifier* **??** | *identifier* **!!**
	|	**true** | **false**
num-simple	::=	*identifier index* | *number*
index	::=	**[** *num-expr* **]** | **(** *num-expr* **)** | ϵ

Das Symbol **#** steht hierbei für ungleich. In Zuweisungen müssen die Datentypen der verwendeten Variablen und Ausdrücke zueinander kompatibel sein.[7]
Der Wert eines (Teil–)Ausdrucks ergibt sich bei Bei Booleschen Ausdrücken (*bool-expr*) unmittelbar. Eine Anweisung wie <$v = 0$> lässt die Programmausführung solange vor diesem Befehl verharren, bis v den Wert 0 annimmt. Zuweisungen der Art <$v := v + 1$> oder <c**!** 5> werden zu **true** ausgewertet, wenn der Wert des Ausdrucks rechts vom **:=** bzw. **!** innerhalb des Definitionsbereiches der Variablen auf der linken Seite liegt.[7]

Innerhalb einer atomaren Aktion kann jede Variable höchstens eine Werteänderung erfahren; eine Anweisung wie <v**:=**3**,**v**:=**v**–**1> ist in DB(PN)^2 nicht möglich. Ausdrücke werden auch in keiner fest vorgegebenen Reihenfolge ausgewertet, bis auf die Tatsache, dass die Evaluation des Ausdrucks auf der rechten Seite einer Zuweisung vor der linken durchgeführt wird. Das letzte Beispiel hätte also auch <v**:=**v**–**1**,** v**:=**3> geschrieben werden können, womit das Verbot solcher Konstrukte anschaulich begründet ist.

Der Zugriff auf einzelne Elemente einer Feldvariablen erfolgt in gewohnter Notation mit eckigen Klammern: für die im letzten Abschnitt deklarierte Variable *vektor* ist beispielsweise *vektor***[**0**]:=** 7 eine mögliche Zuweisung. Auf die gleiche Weise erfolgt der Zugriff auf Komponenten eines Tupels, nur werden hierbei runde statt eckige Klammern verwendet; Beispiel: $tup(1)$**:= not** $tup(1)$.

Wie schon im einleitenden Semaphor–Beispiel dieses Kapitels, Programm 5.1, angedeutet, verwendet die ursrüngliche Sprachdefinition von B(PN)^2 [BH93] in Ausdrücken

[7] Bei Inkompatibilität werden beim späteren Scoping über den Aktionsnamen die entsprechenden Transitionen mangels passender Konjugierten via Restriktion entfernt, dadurch kommt es an dieser Stelle zu einer Verklemmung.

explizit die Vor– und Nachwerte von Variablen, d. h., es werden die Werte unterschieden, die eine Variable *vor* bzw. *nach* Ausführung der entsprechenden atomaren Aktion besitzt. Diese werden dort durch Apostroph notiert.
Aus verschiedenen Gründen ist diese Schreibweise nicht in DB(PN)^2 übernommen worden: Zum einen wird damit die Evaluation von Ausdrücken erheblich vereinfacht, da Vor– und Nachwerte auf der rechten Seite einzelner (Un–)Gleichungen und Zuweisungen nicht gemischt vorkommen können. Daneben wird auch der (in B(PN)^2 durchaus beabsichtigte) Nichtdeterminismus ausgeschlossen, der Ausdrücke wie $< x' = y' >$ ermöglicht. Bei Variablen mit sehr kleinem Definitionsbereich lässt sich ein solcher Befehl vielleicht noch übersetzen, bei den in dieser Arbeit betrachteten Datentypen jedoch nicht mehr.
Auf der anderen Seite sind DB(PN)^2–Programmtexte leichter zu lesen, da für Zuweisungen die vertraute Notation **:=** verwendet wird, die sich vom Vergleichsoperator abhebt. Intern wird in DB(PN)^2 das Konzept der Vor– und Nachwerte jedoch beibehalten, wie später bei der Beschreibung der Netz–Semantik von deutlich wird.[8]

Datenaustausch zwischen Prozessen kann über globale Variablen oder Kanäle erfolgen. Die Syntax der Kanal–Kommunikation ist angelehnt an CSP [Hoa78] und *Occam*: Sei *c* als Kanalvariable deklariert, dann kennzeichnet *c***!** 4 das Schreiben der Zahl 4 auf Kanal *c*; diese Anweisung wird nur dann zu **true** ausgewertet, falls der Wert auch geschrieben werden kann, also der Kanal noch nicht voll ist.
Umgekehrt wird das Lesen von einem Kanal wird mit dem Operator **?** hinter der Kanalvariablen notiert: der Ausdruck *v***:=** *c***?** beschreibt das Auslesen des aktuellen Datums aus dem Kanal *c* in die Variable *v*. Im Gegensatz zu *Occam* muss der eingelesene Wert nicht zwingend einer Variablen zugewiesen werden, sondern kann wie ein Lesezugriff auf eine Variable in einem Ausdruck vorkommen. Kommt ein solcher Term mehrfach in einer atomaren Aktion vor, so ist stets derselbe Zugriff auf den Kanal gemeint: der Ausdruck < *v***:=** *c***? +** *c***? >** weist der Variablen *v* das Doppelte des zuletzt vom Kanal *c* gelesenen Wertes zu.

Der momentane Zustand eines Kanals kann auch einzeln abgefragt werden mit Hilfe der unären Booleschen Operatoren **??** und **!!**. Der Ausdruck *c***??** wird zu **true** ausgewertet gdw. der Kanal *c* leer ist (entspricht einem is_empty(*c*)). Andererseits ist der Ausdruck *c***!!** genau dann **true**, wenn die Kapazität von Kanal *c* erschöpft ist (wie is_full(*c*)). Diese unären Operationen sind nur definiert für Kanäle mit einer Kapazität größer als null.

Damit ist die Beschreibung der wesentlichen syntaktischen Elemente von DB(PN)^2 abgeschlossen; die Menge der syntaktisch korrekten DB(PN)^2–Programme wird im fol-

[8] Es ist also auf relativ einfache Weise möglich, Syntax und Implementation (Parser) derart zu modifizieren, dass Variablen in der Schreibweise mit Vor– und Nachwerten verwendet werden können.

genden ebenfalls DB(PN)^2 geschrieben. Im Anhang ist ihre vollständige Syntax aufgeführt. Neben den hier behandelten Konstrukten ist es beispielsweise möglich, einfache, textersetzende Makros zu definieren, mit denen sich mehrfach verwendete Befehlsfolgen abkürzen lassen. Damit sind DB(PN)^2-Programme leicht für verschiedene Anzahlen gleichartiger Prozesse skalierbar. Eine andere Erweiterung betrifft das Setzen von Kontrollpunkten für die Verifikation der Programme; sie werden in Kapitel 9.2 erläutert.

5.2 Netz–Semantik von DB(PN)^2

Analog zur Definition der Syntax wird nachfolgend die Semantik von DB(PN)^2 separat beschrieben für den Kontroll– und Datenfluss von Programmen. Die Semantik für den Kontrollfluss setzt unmittelbar auf den im letzten Kapitel definierten Operationen auf Petri–Boxen auf.
In Anlehnung an [Mil80, Mil89] werden Variablen(deklarationen) in DB(PN)^2 modelliert durch parallele Komposition spezieller, wiederum sequenzieller Daten–Prozesse, sogenannter Daten–Boxen. Sie beschreiben die möglichen Änderungen der Werte einzelner Variablen. Der aktuelle Zustand einer Datenbox spiegelt den momentanen Variablenwert wider.
Die Datentypen Feld und Tupel können interpretiert werden als gleichzeitiges Beschreiben bzw. Lesen mehrerer Variablen skalaren Typs; sie sind somit durch verallgemeinerte Datenboxen darstellbar. Auch die Kommunikation von Prozessen über Kanäle (festzulegender Kapazität) kann mit Daten–Prozessen beschrieben werden, die das Schreiben und Lesen von Werten modellieren; diese Prozesse werden Kanalboxen genannt.

Variablenzugriffe, beispielsweise innerhalb der Berechnung numerischer Ausdrücke oder in Zuweisungen, werden beschrieben durch Synchronisation von Transitionen des Kontrollflusses mit Transitionen in den entsprechenden Daten– bzw. Kanalboxen, die die beteiligten Variablen darstellen.

5.2.1 Kontrollfluss

Die Semantik des Kontrollflusses von DB(PN)^2–Programmen fußt auf den in Kapitel 4 eingeführten Box–Ausdrücken und deren Kompositionellen Operatoren. DB(PN)^2–Programme sind somit darstellbar als 1–sichere SND–Systeme.
Für die folgenden Betrachtungen wird die Menge der in einer sequenziellen Programmkomponente *par-block* vorkommenden Sprungmarken mit $\mathcal{S}_{\textit{par-block}}$ abgekürzt. Bei der Übersetzung in Box–Ausdrücke werden die benutzerdefinierten Marken übernommen; nachfolgend werden sie durch γ symbolisiert.

Daneben benötigt die Konstruktion von Schleifen und Prozess–Terminierung eindeutige interne Sprungmarken. Diese generischen Marken werden hier mit β (für **break**) und κ (für **continue**) bezeichnet.

Definition 5.3 *Box–Semantik von DB(PN)^2 (Kontrollfluss)*

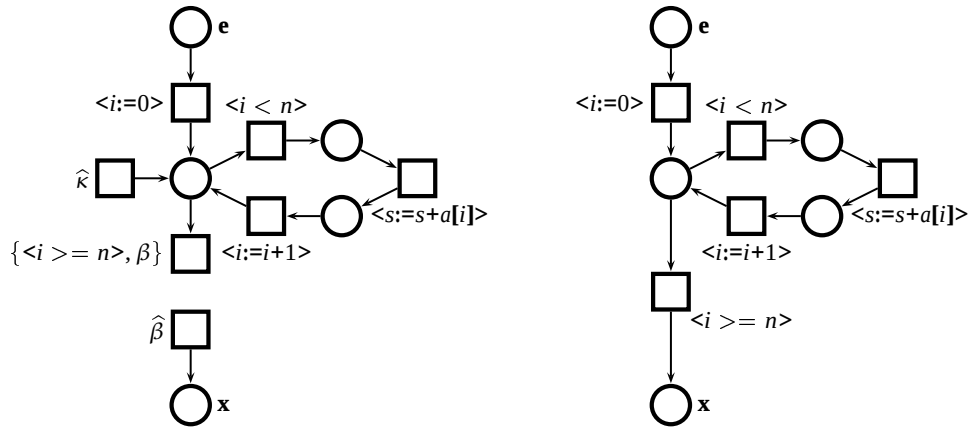
Die Semantikfunktion DExpr : DB(PN)^2 $\rightarrow$ DBEXPR bildet Programmfragmente kompositionell auf Box–Ausdrücke ab. Sie ist induktiv wie folgt definiert.

$$
\begin{aligned}
\mathsf{DExpr}(\ \textit{seq-block} \parallel \textit{par-block}\ ) &= \mathsf{DExpr}(\ \textit{seq-block}\ ) \parallel \mathsf{DExpr}(\ \textit{par-block}\ ) \\
\mathsf{DExpr}(\ \textit{seq-block}\ ) &= [\ \mathcal{S}_{\textit{seq-block}} \cup \{\beta\} : \mathsf{DExpr}(\ \textit{seq-block}\ ) [] [\stackrel{\widehat{\beta}}{\rightarrow} \mathbf{x}]\ ] \\
\mathsf{DExpr}(\ <\textit{expr}>\ ) &= \mathsf{DExpr}(\ \textit{expr}\ ) \\
\mathsf{DExpr}(\ <\gamma\textbf{:}\ \textit{expr}>\ ) &= \mathsf{DExpr}(\ \textit{expr}\ ) [] [\stackrel{\widehat{\gamma}}{\rightarrow} \mathbf{e}] \\
\mathsf{DExpr}(\ \textbf{skip}\ ) &= \emptyset \\
\mathsf{DExpr}(\ \textbf{abort}\ ) &= \textbf{stop} \\
\mathsf{DExpr}(\ \textit{command}_1\ \textbf{;}\ \textit{command}_2\ ) &= \mathsf{DExpr}(\ \textit{command}_1\ )\ \textbf{;}\ \mathsf{DExpr}(\ \textit{command}_2\ ) \\
\mathsf{DExpr}(\ \textbf{if}\ \textit{alt-set}\ \textbf{fi}\ ) &= \mathsf{DExpr}(\ \textit{alt-set}\ ) \\
\mathsf{DExpr}(\ \textbf{do}\ \textit{alt-set}\ \textbf{od}\ ) &= [\ \{\kappa, \beta\} : [\mathsf{DExpr}(\ \textit{alt-set}\ ) [] [\stackrel{\widehat{\kappa}}{\rightarrow} \mathbf{e}]\ *\ [\stackrel{\widehat{\beta}}{\rightarrow} \mathbf{x}]\ ]\ ] \\
\mathsf{DExpr}(\ \textit{alt-set}_1\ []\ \textit{alt-set}_2\ ) &= \mathsf{DExpr}(\ \textit{alt-set}_1\ ) [] \mathsf{DExpr}(\ \textit{alt-set}_2\ ) \\
\mathsf{DExpr}(\ <\textit{expr}>\ \textbf{goto}\ \gamma\ ) &= [\mathsf{DExpr}(\ \textit{expr}\ ) \stackrel{\gamma}{\rightarrow}] \\
\mathsf{DExpr}(\ <\textit{expr}>\ \textbf{continue}\ ) &= [\mathsf{DExpr}(\ \textit{expr}\ ) \stackrel{\kappa}{\rightarrow}] \\
\mathsf{DExpr}(\ <\textit{expr}>\ \textbf{break}\ ) &= [\mathsf{DExpr}(\ \textit{expr}\ ) \stackrel{\beta}{\rightarrow}] \\
\mathsf{DExpr}(\ \textbf{goto}\ \gamma\ ) &= [\emptyset \stackrel{\gamma}{\rightarrow}] \\
\mathsf{DExpr}(\ \textbf{continue}\ ) &= [\emptyset \stackrel{\kappa}{\rightarrow}] \\
\mathsf{DExpr}(\ \textbf{break}\ ) &= [\emptyset \stackrel{\beta}{\rightarrow}]
\end{aligned}
$$

■5.3

Die Semantik von Deklarationen und Ausdrücken ist in dieser Definition noch nicht enthalten; das wird in den beiden nächsten Abschnitten nachgeholt.

Die Übersetzung von Programmkonstrukten in die entsprechende Box ist aufgrund der verwendeten Box–Operationen leicht nachvollziehbar. Besondere Erwähnung verdienen die mit einem Scoping aufgelösten Konstrukte: sequenzielle Komponenten und Schleifen. Die Semantik von ersteren führt für jeden Block eine individuelle Sprungmarke β ein, verbunden mit einem Sprungziel zum Ende des Blockes. Damit wird das vorzeitige Terminieren eines Blockes mit dem **break**–Kommando ermöglicht. Auf ähnliche Weise führt die Semantik von Schleifen Sprungziele zum Schleifenanfang ($\widehat{\kappa}$) und Schleifenende ($\widehat{\beta}$) ein.
Die (un–)bedingten Sprunganweisungen bedienen sich der dazu konjugierten Sprungmarken, auf diese Weise werden mit Hilfe des Scopings Sprünge und Sprungziele zusammengeführt.



(a) Box vor dem Scoping

(b) Box nach dem Scoping

Abbildung 5.1: Box–Semantik des Beispielprogramms P.

Zur Verdeutlichung dieser Übersetzung dient ein beispielhafter Programmausschnitt, der die Einträge eines numerischen Feldes a in der Variablen s aufsummiert:

$$
\begin{array}{ll}
P = & \texttt{<}i\texttt{:=0>;} \\
& \textbf{do} \quad \texttt{<}i < n\texttt{>;<}s\texttt{:=}s\texttt{+}a\texttt{[}i\texttt{]>;<}i\texttt{:=}i\texttt{+1>} \\
& \;[\!] \quad \texttt{<}i >= n\texttt{>}\ \textbf{break} \\
& \textbf{od}
\end{array}
$$

Diese Schleife entspricht der Befehlsfolge **for** i:= 0 **to** $n-1$ **do** s:= $s + a[i]$; in Pascal–Notation. Unter Anwendung von Def. 5.3 ergibt sich $\mathsf{DExpr}(P)$ zu

$$\texttt{<}i\texttt{:=0>;}[\{\kappa,\beta\} : [(\texttt{<}i < n\texttt{>;<}s\texttt{:=}s\texttt{+}a\texttt{[}i\texttt{]>;<}i\texttt{:=}i\texttt{+1>} \;[\!]\; [\texttt{<}i >= n\texttt{>} \xrightarrow{\beta}] \;[\!]\; [\xrightarrow{\hat{\kappa}} \mathbf{e}]) * [\xrightarrow{\hat{\beta}} \mathbf{x}]]]$$

Die Semantik von DB(PN)^2–Ausdrücken wird erst im nächsten Abschnitt definiert, deshalb dienen hier vorerst die atomaren Aktionen des Programms als Beschriftungen. Abbildung 5.1(a) zeigt die entsprechende Petri–Box vor dem Scoping[9] über den Sprungmarken der Schleife. Teilbild (b) zeigt die Box nach dem Scoping; die nicht verwendeten Sprungaktionen sind durch die Restriktion entfernt worden.

[9] Die sequenzielle Komposition der ersten Aktion <i:=0> mit dem Rest des Programmfragments findet tatsächlich erst nach dem Scoping statt.

5.2.2 Atomare Aktionen

Stand in der bisherigen Diskussion des Box–Kalküls und der Sprache DB(PN)^2 eher der Kontrollfluss, also das Programm–Gerüst im Vordergrund, so wird nun die dynamische Verhalten von Variablen(–Änderungen) genauer untersucht. Zunächst wird die Semantik atomarer Aktionen betrachtet; grundlegende Idee bei ihrer Übersetzung ist es, sie als Boolesche Ausdrücke zu interpretieren. Variablenbelegungen, die eine atomare Aktion zu **true** auswerten lassen, können als elementare Boxen dargestellt werden. Erfüllen dieses Kriterium mehrere verschiedene Variablenbelegungen, so werden diese mit dem Auswahl–Operator miteinander verknüpft.
Für die formale Definition wird das Aussehen atomarer Werteänderungen wie folgt festgelegt.

Notation 5.4 *Werteänderungen von Nicht–Kanalvariablen*

Sei v eine beliebige Variable eines DB(PN)^2–Programms. Mit $\mathbb{D}_v$ wird im Folgenden der Definitionsbereich von v bezeichnet, der sich unmittelbar aus der Deklaration von v ergibt (*type-set*).

Die Änderung einer (Nicht–Kanal–)Variablen vom Wert $i \in \mathbb{D}_v$ *vor* Ausführung einer Aktion in den Wert $j \in \mathbb{D}_v$ *nach* ihrer Ausführung wird mit v_k^j notiert.

■5.4

Für $j = k$ ändert die Variable v ihren Wert nicht; dies ist u. a. bei lesenden Zugriffen der Fall. Die Menge der Werteänderungen von v wird abgekürzt mit $\mathcal{A}_v = \{v_k^j \mid j,k \in \mathbb{D}_v\}$[10] $\subset \mathcal{A}$. Die Elemente aus $\mathcal{A}_v$ dienen als Kommunikationsbeschriftungen bei der Übersetzung von DB(PN)^2–Programmen in Box–Ausdrücke (und Boxen).
Beispielweise kann für das Programm **var** i : **int init** 4; <i:=i+3> die atomare Aktion in den Box–Ausdruck DExpr(<i:=i+3>) $= i_7^4$ übersetzt werden.[11] Auf Netzebene ist dies eine elementare Box, deren einzige Transition mit dem Aktionsnamen i_7^4 beschriftet ist. Greift eine atomare Aktion auf mehrere Variablen zu, dann besteht die Beschriftung aus einer Menge von Aktionsnamen.

Kanalvariablen werden etwas anders aufgelöst. Hier stehen keine Werteänderungen im Vordergrund, stattdessen werden die gelesenen bzw. geschriebenen Werte betrachtet. Sei c eine Kanalvariable, dann werden für den Wert $k \in \mathbb{D}_c$ die entsprechenden Aktionsnamen notiert als $c?k$ (Lesen) bzw. $c!k$ (Schreiben), also ähnlich wie schon im Programmtext. Zusätzlich werden für Kanäle Aktionsnamen $c??$ und $c!!$ benötigt, die, wiederum durch die identische Schreibweise angedeutet, die Tests von Kanälen beschreiben. Die Menge der für eine Kanalvariable benötigten Aktionsnamen ist damit $\mathcal{A}_c = \{c?k, c!k, c??, c!! \mid k \in \mathbb{D}_c\} \subset \mathcal{A}$.

[10] Diese Menge enthält nur unkonjugierte Aktionsnamen.

[11] Dies ist eine etwas vereinfachte Darstllung, die noch vom Datenfluss abstrahiert.

Wie nun die einer atomaren Aktion zugeordneten Aktionsnamen konstruktiv aus dem Programmtext gewonnen werden können, formalisiert die folgende Definition.

Definition 5.5 *Evaluierung von DB(PN)^2–Ausdrücken*

Seien v eine Variable skalaren Typs, a eine Feldvariable der Größe n, t eine Tupelvariable und c eine Kanalvariable eines DB(PN)^2–Programms. Die Auswertungsfunktion $\mathsf{Eval} : \text{DB(PN)}^2 \to 2^{(2^{\mathcal{A}} \times (\mathbf{bool} \cup \mathbf{int}))}$ ist induktiv wie folgt definiert.

$$
\begin{aligned}
\mathsf{Eval}(\mathit{expr}_1\,\mathbf{,}\,\mathit{expr}_2) &= \{(\alpha_1 \cup \alpha_2, \mathbf{true}) \mid (\alpha_1, \mathbf{true}) \in \mathsf{Eval}(\mathit{expr}_1) \\
&\quad \wedge (\alpha_2, \mathbf{true}) \in \mathsf{Eval}(\mathit{expr}_2)\} \\
\mathsf{Eval}(v\,\mathbf{:=}\,\mathit{rvalue}) &= \{(\{v_k^j\}, \mathbf{true}) \mid j \in \mathbb{D}_v \wedge k \in \mathsf{Eval}(\mathit{rvalue})\} \\
\mathsf{Eval}(t\,\mathbf{(}\,\mathit{num\text{-}expr}\,\mathbf{)\!:=}\,\mathit{rvalue}) &= \{(\{t_{(k_0,k_1)}^{(j_0,j_1)}\} \cup \alpha, \mathbf{true}) \mid (j_0,j_1),(k_0,k_1) \in \mathbb{D}_t \\
&\quad \wedge (\alpha, i) \in \mathsf{Eval}(\mathit{num\text{-}expr}) \wedge i \in \{0,1\} \\
&\quad \wedge k_i \in \mathsf{Eval}(\mathit{rvalue})\} \\
\mathsf{Eval}(a\,\mathbf{[}\,\mathit{num\text{-}expr}\,\mathbf{]\!:=}\,\mathit{rvalue}) &= \{(\{a_{[k_0,\ldots,k_i,\ldots,k_{n-1}]}^{[j_0,\ldots,j_{n-1}]}\} \cup \alpha, \mathbf{true}) \mid \\
&\qquad [j_0,\ldots,j_{n-1}], [k_0,\ldots,k_{n-1}] \in \mathbb{D}_a \\
&\qquad \wedge (\alpha, i) \in \mathsf{Eval}(\mathit{num\text{-}expr}) \wedge 0 \le i < n \\
&\qquad \wedge k_i \in \mathsf{Eval}(\mathit{rvalue})\} \\
\mathsf{Eval}(c\mathbf{!}\mathit{rvalue}) &= \{(\{c!k\}, \mathbf{true}) \mid k \in \mathbb{D}_c \wedge k \in \mathsf{Eval}(\mathit{rvalue})\} \\
\mathsf{Eval}(v) &= \{(\{v_k^j\}, j) \mid j,k \in \mathbb{D}_v \wedge (\mathrm{free}(v) \Rightarrow j = k)\} \\
\mathsf{Eval}(t\,\mathbf{(}\,\mathit{num\text{-}expr}\,\mathbf{)}) &= \{(\{t_{(k_0,k_1)}^{(j_0,j_1)}\} \cup \alpha, j_i) \mid (j_0,j_1),(k_0,k_1) \in \mathbb{D}_t \\
&\quad \wedge (\alpha, i) \in \mathsf{Eval}(\mathit{num\text{-}expr}) \wedge i \in \{0,1\} \\
&\quad \wedge (\mathrm{free}(t) \Rightarrow (j_0,j_1) = (k_0,k_1))\} \\
\mathsf{Eval}(a\,\mathbf{[}\,\mathit{num\text{-}expr}\,\mathbf{]}) &= \{(\{a_{[k_0,\ldots,k_{n-1}]}^{[j_0,\ldots,j_i,\ldots,j_{n-1}]}\} \cup \alpha, j_i) \mid \\
&\qquad [j_0,\ldots,j_{n-1}], [k_0,\ldots,k_{n-1}] \in \mathbb{D}_a \\
&\qquad \wedge (\alpha, i) \in \mathsf{Eval}(\mathit{num\text{-}expr}) \wedge 0 \le i < n\} \\
&\quad \wedge (\mathrm{free}(a) \Rightarrow \forall l \in \{0,\ldots,n-1\} : j_l = k_l)\} \\
\mathsf{Eval}(c\mathbf{?}) &= \{(\{c?k\}, k) \mid k \in \mathbb{D}_c\} \\
\mathsf{Eval}(c\mathbf{??}) &= \{(\{c??\}, \mathbf{true})\} \\
\mathsf{Eval}(c\mathbf{!!}) &= \{(\{c!!\}, \mathbf{true})\} \\
\mathsf{Eval}(\mathit{un\text{-}op}\ \mathit{some\text{-}expr}) &= \{(\alpha, \mathit{un\text{-}op}\ v) \mid (\alpha, v) \in \mathsf{Eval}(\mathit{some\text{-}expr})\} \\
\mathsf{Eval}(\mathit{some\text{-}expr}\ \mathit{bin\text{-}op}\ \mathit{some\text{-}expr}) &= \{(\alpha_1 \cup \alpha_2, v_1\ \mathit{bin\text{-}op}\ v_2) \mid \\
&\qquad (\alpha_1, v_1) \in \mathsf{Eval}(\mathit{some\text{-}expr}) \\
&\qquad \wedge (\alpha_2, v_2) \in \mathsf{Eval}(\mathit{some\text{-}expr})\} \\
\mathsf{Eval}(\mathit{some\text{-}const}) &= \{(\emptyset, \mathit{some\text{-}const})\}
\end{aligned}
$$

Hierbei bezeichnet *bin-op* einen binären Operator (also *bool-op* , *num-op* oder *comp-op*), *un-op* einen unären Operator (**–** oder **not**) und *some-expr* einen Booleschen (*bool-expr*) bzw. numerischen Ausdruck (*num-expr*). Weiterhin steht *some-const* für eine Boolesche (**false**, **true**) oder numerische (*number*) Konstante. ■5.5

Die Abbildung Eval weist dabei jedem Ausdruck eine Menge von Tupeln zu; jedes dieser Tupel enthält einen elementaren Box–Term (genauer eine Menge von Werteübergängen, also Aktionsnamen) und den unter dieser repräsentierten Variablenbelegung berechneten Wert des Ausdrucks.
Das bei der Auswertung von Nicht–Kanalvariablen verwendete Prädikat *free* gibt an, ob die untersuchte Variable frei im *gesamten* Ausdruck auftritt, d. h., ob ihr kein neuer Wert zugewiesen wird. In diesem Fall findet keine Werteänderung statt, Vor– und Nachwert sollten übereinstimmen.
Die Sprache B(PN)^2 [BH93] fordert bei der Modellierung gerade in diesem letzten Punkt mehr Überlegung; eine intensive Auseinandersetzung mit dem Konzept der Vor– und Nachwerte ist Voraussetzung, sollen möglichst kompakte Darstellungen generiert werden. Beispielsweise kann eine unüberlegt geschriebene Anweisung wie **<***z***':='***x***+'***y***>** schon bei relativ kleinen Definitionsbereichen von etwa 10 Werten pro Variable den Rechner bei der Verifikation des Programms einige Zeit (womöglich ungewollt) beschäftigen. Für das obige Beispiel werden nämlich alle potenziellen Nachwerte für x und y berechnet, obwohl vielleicht gar keine Werteänderung beabsichtigt war.
DB(PN)^2 nimmt beim Programmdesign etwas von dieser Arbeit ab, indem bei der Übersetzung automatisch die tatsächlich veränderten Variablen erkannt werden. Dabei mag zwar etwas an Flexibilität eingebüßt werden, aber es entstehen keine unschönen Seiteneffekte wie die eben angedeuteten.

Bei der Übersetzung eines DB(PN)^2–Programms werden die eine atomare Aktion erfüllenden Variablenbelegungen bzw. –änderungen alternativ miteinander verknüpft. Damit ist Definition 5.3, Box–Semantik von DB(PN)^2, auf die folgende Weise für Deklarationen erweiterbar:

$$\mathsf{DExpr}(\,expr\,) = \bigsqcup_{(\alpha,\mathbf{true})\in \mathsf{Eval}(expr)} \alpha$$

Die Semantik von Ausdrücken soll durch einige Beispiele veranschaulicht werden, die nachstehende Variablendeklarationen voraussetzen.

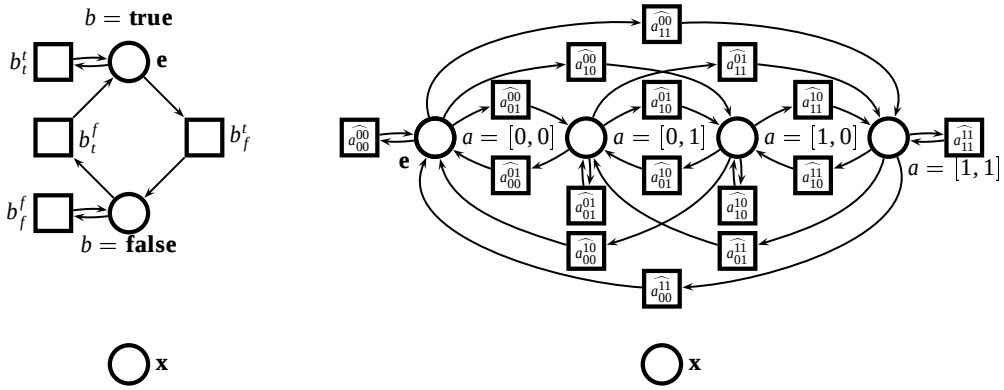
var i : **{**0**..**2**};** **var** c : **chan** 2 **of {**0**,** 1**};** **var** f : **array** 2 **of {**0**,** 1**};**

$$\begin{aligned}
\mathsf{DExpr}(5{=}5) &= \emptyset \\
\mathsf{DExpr}(i{:=}c?{+}2) &= \{i_2^0, c?0\} \sqcup \{i_2^1, c?0\} \sqcup \{i_2^2, c?0\} \\
\mathsf{DExpr}(i{:=}c?{+}3) &= \mathbf{stop} \\
\mathsf{DExpr}(f\mathbf{[}1\mathbf{]:=}2{-}\,f\mathbf{[}0\mathbf{]}) &= f_{[0,1]}^{[0,0]} \sqcup f_{[0,1]}^{[0,1]} \sqcup f_{[1,0]}^{[1,0]} \sqcup f_{[1,0]}^{[1,1]} \\
\mathsf{DExpr}(c!f\mathbf{[}i\mathbf{]}{+}1) &= \{i_0^0, f_{[0,0]}^{[0,0]}, c!1\} \sqcup \{i_1^1, f_{[0,0]}^{[0,0]}, c!1\} \sqcup \{i_0^0, f_{[0,1]}^{[0,1]}, c!1\} \sqcup \{i_1^1, f_{[0,1]}^{[0,1]}, c!2\} \\
&\quad \sqcup \{i_0^0, f_{[1,0]}^{[1,0]}, c!2\} \sqcup \{i_1^1, f_{[1,0]}^{[1,0]}, c!1\} \sqcup \{i_0^0, f_{[1,1]}^{[1,1]}, c!2\} \sqcup \{i_1^1, f_{[1,1]}^{[1,1]}, c!2\}
\end{aligned}$$

Die Größe der Definitionsbereiche der Variablen sind absichtlich klein gewählt worden, die Beispiele machen dennoch bereits ein Hauptproblem dieser Art von Übersetzung deutlich: Infolge der expliziten Berechnung und Verwaltung aller kombinatorisch möglichen Werteänderungen, die einen Ausdruck erfüllen, beansprucht diese Darstellung sehr viel Speicherplatz, insbesondere bei großen Definitionsbereichen. Für die spätere Synchronisation mit den Daten–Prozessen wird noch zusätzlich Platz benötigt.
Eine Möglichkeit der kompakteren Darstellung bieten hier High–Level–Petri–Netze, bei denen die Variablenübergänge durch symbolische Beschriftungen vorgenommen werden. Die Marken dieser Netze sind nicht mehr nur quantitativ spezifizierbar (Markenzahl auf einem Stellenknoten), sondern auch qualitativ (der Typ einer „gefärbten“ Marke spiegelt beispielsweise den aktuellen Variablenwert wider). Der Box–Kalkül bietet hierfür eine spezielle Netzklasse an, die M–Netze [BFF$^+$95] genannt werden. Sie sind in den letzten Jahren in zahlreichen Veröffentlichungen untersucht worden.
Verschiedene Werkzeuge [PEP98, SSE99] unterstützen bereits die M–Netz–Semantik bei der Übersetzung von B(PN)^2–Programmen oder Automatendarstellungen, bislang allerdings vornehmlich auf Modellierungsebene. Effiziente Algorithmen zur Untersuchung des Zustandsraums der auf diese Weise modellierten Systeme existieren bislang kaum. Für die Verifikation z. B. mit Hilfe von Model–Checkern müssen diese Netze daher in einfache (1–sichere) Stellen-/Transitions–Netze aufgefaltet werden – auf diesem Wege werden wiederum alle potenziellen Werteübergänge explizit erzeugt.
Aus diesem Grund wird in dieser Arbeit von vornherein auf Low–Level–Netze in Form von Petri–Boxen zurückgegriffen. Allerdings werden bei der Konstruktion der Netz–Semantik eines Programmes die Ausdrücke atomarer Aktionen nicht explizit in der obigen Form expandiert, sondern es werden nur die tatsächlich in Programmläufen vorkommenden Werteänderungen bei der Konstruktion des vollständigen Präfixes ermittelt und gespeichert. Das nächste Kapitel wird dies ausführlich erörtern.

5.2.3 Deklarationen und Daten–Prozesse

Zur Repräsentation der aktuellen Variableninhalte während der Programmausführung werden spezielle Daten–Prozesse eingeführt.
Im Gegensatz zur eher formalen Definition der Kontrollfluss–Semantik erfolgt die Beschreibung des Datenflusses an vielen Stellen mehr informell und durch Angabe konkreter Beispiele und Netze. Ihre exakte Definition als Box–Ausdruck wäre sehr unübersichlich und wenig anschaulich, zudem wird diese Darstellungsform nicht für die hier beschriebene Übersetzung benötigt.
Die (Netz–)Semantik einer Variablendeklaration $decl_v$ beschreibt die möglichen Werteübergänge von v. Die Transitionen sind mit Elementen der Menge $\widehat{\mathcal{A}_v}$ beschriftet, also gerade mit den entsprechenden konjugierten Aktionsnamen des Kontrollflusses. Datenboxen werden parallel komponiert mit dem Kontrollfluss; über das Scoping werden die Paare später zusammengeführt.



(a) DExpr(**var** b : **bool init true**) (b) DExpr(**var** $a : \{0,1\}$**init** $[0,0]$)

Abbildung 5.2: Beispiele für Datenboxen[12]*.*

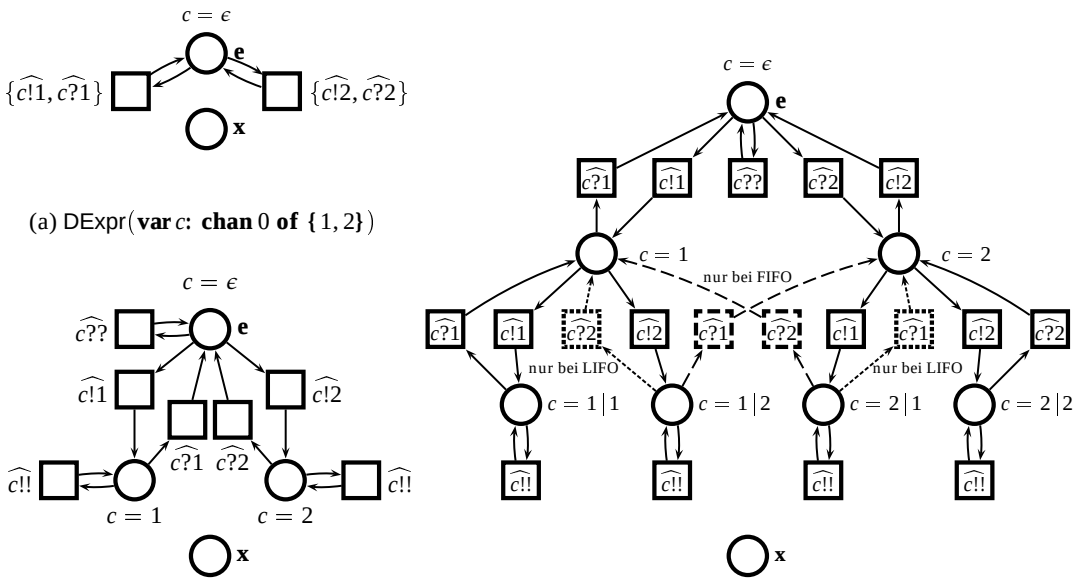
Abbildung 5.2 zeigt die Datenbox einer Booleschen Variablen, die mit **true** initialisiert ist, sowie die Box einer Feldvariablen. Die Boxen für beliebige andere Nicht–Kanalvariablen sind daraus unmittelbar ableitbar. Die Eingangsstelle dieser Boxen ist jeweils derjenige Stellenknoten, der den Initialwert der Variablen repräsentiert.
Damit ist Definition 5.3, die Semantik von DB(PN)^2, auf die folgende Weise für Deklarationen erweiterbar.

$$\mathsf{DExpr}(\, \mathit{decl}_v ; \mathit{par\text{-}block}\,) = [\, \mathcal{A}_v : \mathsf{DExpr}(\, \mathit{decl}_v\,) \,\|\, \mathsf{DExpr}(\, \mathit{par\text{-}block}\,)\,]$$

Kanalboxen werden auf ähnliche Weise konstruiert wie Datenboxen. Die Menge der Kommunikationsaktionen einer Kanalvariablen c ist $\mathcal{A}_c = \{c!k, c?k \mid k \in \mathbb{D}_c\} \cup \{c!!, c??\}$. Neben den Schreib– und Leseaktionen sind also auch die Tests enthalten. Abbildung 5.3 zeigt Ausprägungen für LIFO– und FIFO–Kanäle über einem zweiwertigen Datentyp mit Kapazitäten 0 und 1 (die Semantik für beide Kanaltypen ist hierbei gleich) sowie der Kapazität 2. Für den letzten Fall sind in Teilbild 5.3(c) die Netze für beide Typen in einem einzigen Netz zusammengefasst, da sie sich nur geringfügig unterscheiden: LIFO–Kanäle enthalten die beiden kurzgestrichelten Transitionen, FIFO–Kanäle die mit länger gestrichelten Linien gezeichneten Transitionen in der Mitte.

Die Netz–Semantik von Kanalboxen weicht ab von der Darstellung, die beispielsweise

[12] In Kapitel 4.4.4 ist bereits darauf hingewiesen worden, dass sich Datenboxen im Allgemeinen nur mit Hilfe von Sprungoperatoren als Box–Ausdrücke angeben lassen, um die obige endliche Netzdarstellung zu erhalten.



(a) DExpr(**var** *c*: **chan** 0 **of** {1, 2})

(b) DExpr(**var** *c*: **chan** 1 **of** {1, 2})

(c) DExpr(**var** *c*: **chan** / **stack** 2 **of** {1, 2})

Abbildung 5.3: Beispiele für Kanalboxen.

in [BH93] gewählt worden ist: dort ist darauf geachtet worden, kompakte Boxen mit einer möglichst kleinen Anzahl von mit Aktionsnamen beschrifteten Transitionen vorzuschlagen. Allerdings sind einige zusätzliche unsichtbare (mit $\emptyset$ beschriftete) Transitionen notwendig, die bei der hier gewählten Form entfallen können. Auf diese Weise wird in [BH93] die Zahl von Synchronisationen während des Scopings von Kontroll– und Datenfluss weitgehend gering gehalten.
Die in dieser Arbeit vorgeschlagenen Netze erfüllen andere Anforderungen: die obigen Kanalboxen stellen wiederum Zustandsnetze dar; Komposition und Scoping mit dem Kontrollfluss erhalten somit die Eigenschaften der hier vorgestellten Boxen, es sind weiterhin SND–Systeme. Der aktuelle Zustand eines Kanals lässt sich unmittelbar an einem Stellenknoten ablesen – anstatt einer Menge von Stellen bei der in [BH93] gewählten Darstellung.

Aus dem letzten Kapitel ist bekannt, dass eine Box terminiert, wenn gerade alle Ausgangsstellen – und nur diese – mit einer Marke belegt sind. Bei genauerer Betrachtung der oben definierten Daten– und Kanalboxen wird auffallen, dass diese in der dargestellten Form den Endzustand niemals erreichen können. Jedes modellierte System würde somit ungewollte Verklemmungen aufweisen, sicherlich eine nicht gewünschte

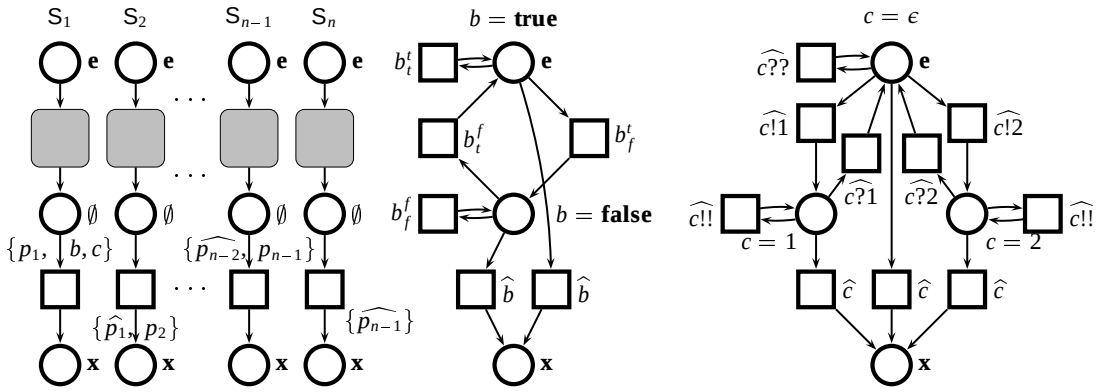


Abbildung 5.4: Schema der Variablen–Deinitialisierung.

Eigenschaft im Rahmen der Verifikation.
Die Lösung besteht darin, Variablen am Programmende zu deinitialisieren. Dieses Vorgehen ist bereits in [BH93] verwendet worden, dort ist es von noch größerer Bedeutung wegen der Möglichkeit der Deklaration lokaler Variablen.
Die Deinitialisierung kann auf einfache Weise erfolgen: in den entsprechenden Daten– bzw. Kanalboxen wird jede nicht mit **x** beschriftete Stelle über eine Transition mit der Ausgangsstelle verbunden. Diese Transitionen werden beschriftet mit dem konjugierten Namen der Variablen (ohne Werteübergang). Abbildung 5.4 zeigt diese Konstruktion am Beispiel je einer einfachen Daten– und Kanalbox, beide rechts im Bild. Jede Aktionsmenge $\mathcal{A}_v$ einer Variablen $v \in \mathsf{Var}(P)$ eines DB(PN)^2–Programms P enhält also zusätzlich zu den bisherigen Ausführungen den Aktionsname v.
Der Kontrollfluss muss nun seinerseits sicherstellen, dass die Deinitialisierung erst bei Erreichen seines Endzustands eingeleitet wird. Dabei wird etwas anders vorgegangen als in [BH93], wo die konjugierten Aktionen zur Variablen–Deinitialisierung sequentiell an den Kontrollfluss komponiert werden. Zum einen muss hier die SND–Eigenschaft der Boxen erhalten werden, zum anderen soll das zu berechnende Präfix möglichst minimale Tiefe besitzen.
Deshalb wird an jede der $n \geq 1$ sequenziellen Programmkomponenten S_i, $1 \leq i \leq n$, des Kontrollflusses eine elementare Deinitialisierungs–Box DB_i sequenziell komponiert (S_i;DB_i), die folgendes Aussehen besitzt:

$$DB_i = \begin{cases} \{p_1 \mid n > 1\} \cup \mathsf{Var}(P) & \text{falls } i = 1, \\ \{\widehat{p_i}, p_{i+1}\} & \text{falls } 1 < i < n, \\ \{\widehat{p_n}\} & \text{falls } i = n, n > 1. \end{cases}$$

Die (eindeutigen) Aktionsnamen $\mathcal{A}_p = \{p_1, \ldots, p_{n-1}\}$ dienen zur Synchronisation der sequenziellen Komponenten unteinander zwecks gemeinsamer Variablen–Deinitiali-

sierung; in Abbildung 5.4 ist im linke Teil schematisch das Netz eines auf diese Weise erweiterten Kontrollflusses gezeigt, die grauen Blöcke abstrahieren von konkreten sequentiellen Komponenten.
Die hinzugefügten Boxen bzw. Transitionen werden in einem abschließenden Scoping über dem gesamten Programm zusammengeführt. Damit ist die Semantik von DB(PN)^2 aus Definition 5.3 abschließend auf die folgende Weise zu vervollständigen:

$$\begin{aligned} \mathsf{DExpr}(\textit{program}) &= [\,\mathcal{A}_p : \mathsf{DExpr}(\textit{decl};\textit{par-block})\,] \\ \mathsf{DExpr}(\textit{seq-block}_i) &= [\,\mathcal{S}_{\textit{seq-block}_i} \cup \{\beta\} : \mathsf{DExpr}(\textit{seq-block}_i)[][\overset{\widehat{\beta}}{\rightarrow}\mathbf{x}]\,];B_i \end{aligned}$$

Die untere Regel ist zu verstehen als Ersatz der zweiten Tabellenzeile von Def. 5.3.

5.3 Beispiel einer Programm–Übersetzung

Zum Abschluss der Sprachbeschreibung führt dieser Abschnitt die vollständige Übersetzung eines DB(PN)^2–Programms in die entsprechende Petri–Box exemplarisch vor, wiederum für Petersons Lösung zum gegenseitigen Ausschluss, für die Abbildung 3.11 eine einfache, direkte Modellierung als Petri–Netz zeigte.
In DB(PN)^2 kann dieser Algorithmus folgendermaßen codiert werden.

Programmtext 5.6 *Der Mutex–Algorithmus nach Peterson [Pet81] in DB(PN)^2*

```
1   var in1,in2:bool init false;
2   var hold:{1,2}init 1;
3     do <in1:=true>; <hold:=1>;
4                     if <not in2 or hold=2> fi;
5                       /* Kritischer Abschnitt 1 */
6                     <in1:=false>
7     od
8   ||
9     do <in2:=true>; <hold:=2>;
10                    if <not in1 or hold=1> fi;
11                      /* Kritischer Abschnitt 2 */
12                    <in2:=false>
13    od                                              ■5.6
```

Die beiden **if**–Anweisungen in Zeile 4 bzw. 10 besitzen keinen alternativen Zweig, sie modellieren damit ein aktives Warten wie ein **await**–Befehl. Das Einbetten des Tests in **if** ... **fi** dient lediglich der Hervorhebung dieses aktiven Wartens; die Semantik des Tests ist die gleiche mit und ohne das **if**, da jede atomare Aktion nur bei Erfüllen des enthaltenen Ausdrucks ausgeführt werden kann.
In einem ersten Schritt wird zunächst der Kontrollfluss dieses Programms übersetzt. Abbildung 5.5 zeigt am linken bzw. rechten Bildrand den ersten bzw. zweiten Prozess

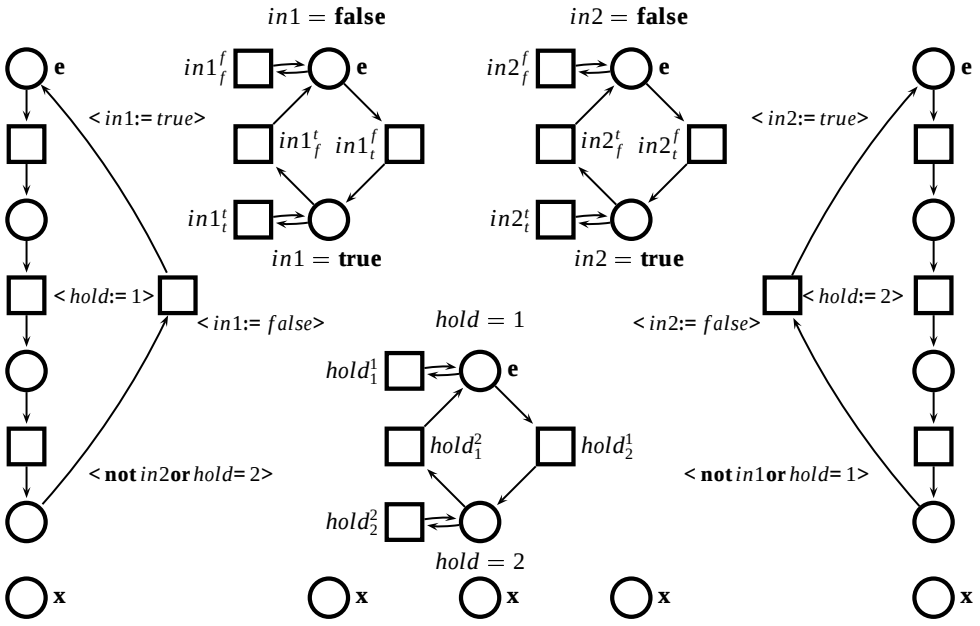


Abbildung 5.5: Netz–Semantik von Programm 5.6: Kontrollfluss und Datenboxen.

als Box. Die atomaren Aktionen sind noch nicht aufgelöst worden, die Transitionen sind deshalb mit Ausschnitten aus dem Programmtext beschriftet. In der Bildmitte sind die Datenboxen der deklarierten Variablen dargestellt.

Als nächstes wird die Semantikfunktion auf die atomaren Aktionen angewendet, daraus resultiert die in Abb. 5.6 dargestellte Box.

Die Auswertung der Anweisungen ermittelt die möglichen Werteänderungen der Variablen, alle in Frage kommenden Lösungen werden alternativ in das Netz eingefügt. Daher ist diese Darstellung auch um einige Transitionen größer als das Netzsystem aus Abb. 3.11. Bei der automatischen Übersetzung wird nicht ermittelt, dass zum Ende der (Endlos–)Schleifen der Wert der Variablen *in1* bzw. *in2* wieder auf **false** gesetzt wird und somit eine Werteänderung $in1_t^t$ für die Aktion **<** *in*1**:= true >** nie stattfinden kann (analog für *in2*).

Auch die gestrichelt gezeichneten Transitionen sind infolge der Auswahl über alle den Ausdruck **< not** *in*1 **or** *hold***=** 1 **>** (für den zweiten Prozess entsprechend) erfüllenden Werteänderungen eingefügt worden. Da innerhalb dieser Disjunktion jedoch nur lesende Zugriffe auf Nicht–Kanalvariablen erfolgen, kann die gestrichelte Alternativen entfallen und die Disjunktion damit in eine strukturelle Exklusiv–Oder–Verknüpfung aufgelöst werden.

Abbildung 5.7 zeigt schließlich die Box nach dem Scoping über alle Aktionsnamen. Die Deinitialisierung der Variablen ist aus dieser Konstruktion wiederum herausgehalten

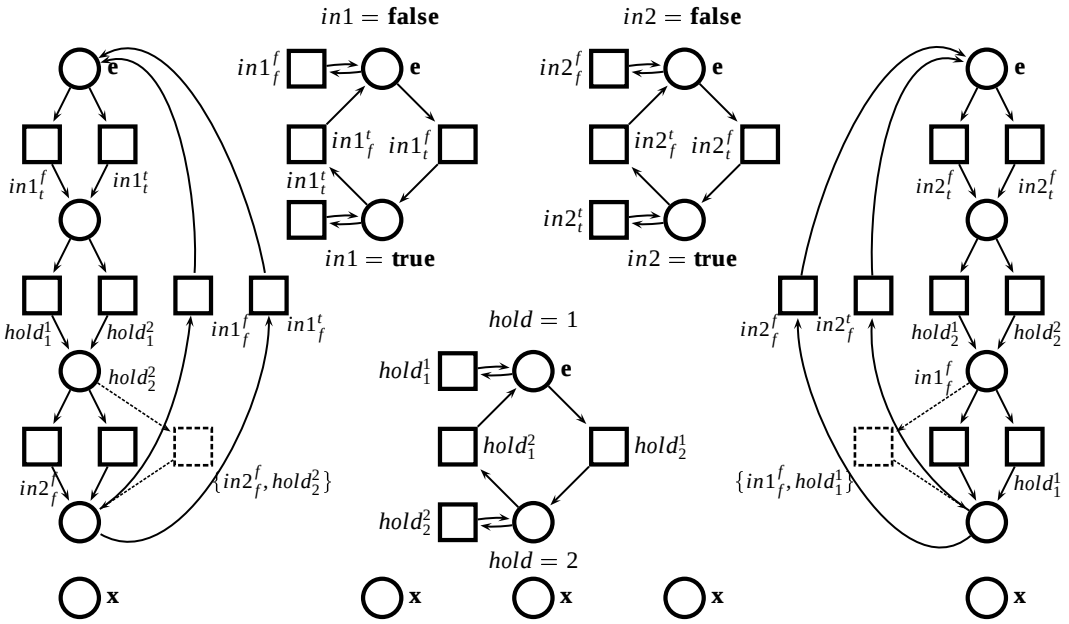


Abbildung 5.6: Auflösen der atomaren Aktionen der Box aus Abb· 5.5.

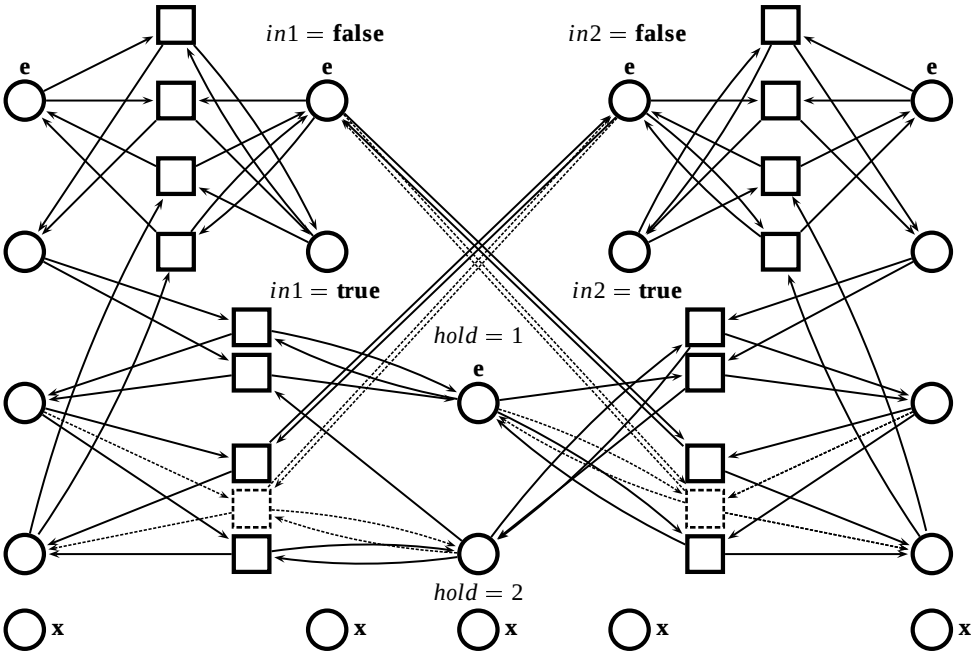


Abbildung 5.7: Scoping der Box aus Abb. 5.6 über den enthaltenen Aktionsnamen.

worden, da sie in diesem Beispiel nicht zum Tragen kommt. Die beiden Benutzerprozesse spiegeln reaktive Systeme wider, die in einer Endlosschleife immer wieder ausgeführt werden und nicht terminieren.

Damit sind alle Vorbereitungen abgeschlossen, eine parallele Hochsprache, DB(PN)^2, samt ihrer soeben definierten Netz–Semantik zusammenzuführen mit der Technik der Netzentfaltungen aus Kapitel 3. Das wird Gegenstand der nachfolgenden Ausführungen sein.
Die hier beschriebene Übersetzung in Netze ist auch auf andere parallele Hochsprachen übertragbar. Weiterhin ist die Beschränkung auf SND–Systeme eher technischer Natur, um die Konstruktion des Präfixes möglichst effizient ausführen zu können. Die Sprache B(PN)^2 [BH93] und der ursprüngliche Box–Kalkül aus [BFF$^+$95] mit seiner Garantie der Erzeugung 1–sicherer Netze bieten hier einen etwas allgemeineren Zugang, der jedoch gerade aufgrund dieser Allgemeinheit beim Einsatz in der Verifikation praktisch relevanter Systeme nicht selten zu Problemen führt.

Kapitel 6

„Tempora mutantur, nos et mutamur in illis!“
(Die Zeiten ändern sich, und wir uns mit ihnen!)
KAISER LOTHAR I.

„Die Zeit verwandelt uns nicht,
sie entfaltet uns nur.“
MAX FRISCH, TAGEBUCH

Präfix–Generierung für DB(PN)^2–Programme

Nach der Vorstellung einer Programmnotation zur Modellierung nebenläufiger, reaktiver Systeme sowie einer geeigneten Netz–Semantik für diese Sprache untersucht dieses Kapitel, wie für die so erhaltenen Netze die Berechnung eines Präfixes ihrer Entfaltung auf möglichst geschickte Weise berechnet werden kann.

Das naheliegende Vorgehen besteht darin, die einem DB(PN)^2–Programm zugeordnete Petri–Box, ein 1–sicheres Netzsystem, direkt als Eingabe für Algorithmus 3.12 zu verwenden (unter Ausnutzen der SND–Eigenschaft könnten dabei die entsprechenden speziellen Ordnungen bei der Präfix–Generierung Verwendung finden). Dieser Weg wird auch in den derzeitig verfügbaren Werkzeugen beschritten, die auf dem Box–Kalkül aufsetzen [PEP98, SSE99]. Allerdings wird dort bei der Übersetzung von Programmen stets noch eine Zwischendarstellung erzeugt, nämlich High–Level–Petri–Netze (genauer M–Netze, [BFF$^+$95]), oder alternativ, jedoch seltener verwendet, Ausdrücke der Box–Algebra.
Ein Problem für die Präfix–Generierung ist dabei häufig die Größe der Low–Level–Netzdarstellung, welche insbesondere beeinflusst wird vom modellierten Datenfluss durch die explizite Verwaltung aller Werteänderungen der deklarierten Variablen.

Einige Verbesserungen bezüglich dieser Netzgröße lassen sich bereits während der Modellierungsphase des Systems durch sorgfältiges Programmdesign erreichen. Beispielsweise hilft die explizite Initialisierung von Variablen, eine ganze Reihe von Teilprozessen aus der Berechnung des Präfixes herauszuhalten. Weiterhin ist eine Beschränkung auf möglichst kleine Wertebereiche hilfreich (ein schönes Beispiel hierfür liefert

die Modellierung fehlertoleranter Systeme in [Mer96]), die weitgehende Vermeidung von Nichtdeterminismen oder die Einschränkung der Zahl möglicher Werteänderungen bei der Formulierung von Ausdrücken innerhalb atomarer Aktionen. Insgesamt ist also eine intensive Auseinandersetzung mit der hinter der Sprache stehenden Netz–Semantik erforderlich, um das System für die eigentliche Verifikation vorzubereiten.

Für den Box–Kalkül wird in [LG93, Röm93, Kou94] ein alternativer Weg der Erzeugung der Prozess–Semantik aufgezeigt: Halbordnungs–Semantiken werden dort auf direktem Wege für Box–Ausdrücke festgelegt, sowohl auf Basis von Kausal– wie auch von Occurrence–Netzen. Dabei wird die Kompositionalität der Box–Operatoren auf natürliche Weise ausgenutzt, Box–Prozesse werden analog wie Boxen für Box–Ausdrücke definiert. Die Konsistenz der so auf direktem Wege berechneten Entfaltung mit der auf indirektem Wege gewonnenen (d. h. Übersetzung eines Box–Ausdrucks in eine Box und anschließendes Entfalten dieser Box) wird in [LG93] nachgewiesen.
Im Rahmen der Verifikation besitzt dieses Vorgehen jedoch einen schwer wiegenden Nachteil: es wird stets die maximale Entfaltung berechnet, und diese ist im Allgemeinen unendlich. Ein Übertragen der Cut-off–Eigenschaft, die das Erzeugen eines vollständigen, endlichen Präfixes garantiert, ist auf die kompositionelle Konstruktion von Prozessen nicht ohne weiteres möglich. Die Präfix–Generierung nach dem McMillan–Algorithmus erfolgt anschaulich durch sukzessives, schichtweises Entfalten des Prozess–Netzes, eine solche Vorgehensweise ist bei der kompositionellen Konstruktion im Allgemeinen nicht möglich. Beispielsweise kann ein abschließendes Scoping über den Aktionsnamen einer Programmvariablen erfordern, dass Transitionen mit niedriger Tiefe verschmolzen werden müssen – zum Zeitpunkt des Scopings liegt jedoch bereits der komplett entfaltete Kontrollfluss vor, und der kann unendlich groß sein.
Um ein endliches Teilstück der Entfaltung überhaupt mit dem Rechner kompositionell konstruieren zu können, müssen daher (oft künstliche) Schranken vorgegeben werden, die beispielsweise die maximale Iterationstiefe bestimmen, bis zu der eine Schleife entfaltet wird. Auf diese Weise kann jedoch nicht garantiert werden, dass in dem konstruierten Teilnetz der Entfaltung jede erreichbare Markierung repräsentiert ist.

Dieses Kapitel präsentiert ein Verfahren, das zwischen den beiden oben dargestellten angesiedelt ist, und das es darüber hinaus zulässt, Variablen mit sehr großen Definitionsbereichen zu deklarieren, da nur die tatsächlich in Ausführungen vorkommenden Werte gespeichert werden (die Zahl der möglichen Werteänderungen muss auch hier überschaubar bleiben, sonst kann ein solches Netz nicht verwaltet werden).
Eine Version des Algorithmus 3.12, die Berechnung eines vollständigen Präfixes der Entfaltung, bildet dabei den zentralen Teil, wobei hauptsächlich die Box des Kontrollflusses des modellierten Systems als Eingabe dient. Die Kompositionalität der Box–Operatoren wird in der Weise genutzt, dass das Scoping über Aktionsnamen erst sehr spät, nämlich bei der Erzeugung erweiternder Ereignisse durchgeführt wird. Die für

das Scoping notwendigen Datenfluss–Informationen werden dabei aus den Variablendeklarationen und dem bereits konstruierten Teil des Präfixes gewonnen. Die Berechnung des Netzgraphen wird dabei rein konstruktiv vorgenommen, in dem Sinne, dass kein Knoten, der bereits erzeugt worden ist, später wieder gelöscht wird (im Gegensatz zur sukzessiven Ausführung von Scoping als Synchronisation gefolgt von Restriktion).

DB(PN)^2–Programme werden mit der in dieser Arbeit vorgestellten Netzsemantik in SND–Systeme übersetzt. Diese Tatsache wird auch bei der Präfix–Generierung an mehreren Stellen ausgenutzt, unter anderem durch die Wahl einer der adäquaten SND–Silex–Ordnungen. Da bei der Übersetzung von Programmen in Netze Variablenzugriffe modelliert werden, wäre ein interessanter alternativer Ansatz gewesen, für die Präfix–Generierung die in [VSY98] beschriebene Ordnung für Netze mit Lesekanten zu verwenden. Unglücklicherweise sind die hier betrachteten Netzsysteme jedoch nicht persistent im dort beschriebenen Sinne.

6.1 Grundstruktur der DB(PN)^2–Präfix–Generierung

Die Präfix–Generierung eines (syntaktisch korrekten) DB(PN)^2–Programms beginnt mit der Erfassung der deklarierten Variablen. Für jede Variable wird deren Name, ihr Definitionsbereich sowie, bei Nicht–Kanal–Variablen, der initiale Wert gespeichert. Während des weiteren Parsens des Programmtextes wird der Kontrollfluss des Programms kompositionell als Petri–Box konstruiert; hierbei finden die in den letzten beiden Kapiteln beschriebenen Techniken Verwendung. Die atomaren Aktionen werden dabei unverändert[1] an die Transitionen der elementaren Boxen geheftet. Lediglich Sprungmarken werden isoliert, weil sie für die Konstruktion des Kontrollflusses benötigt werden.
Ist der komplette Programmtext gelesen und die Box des Kontrollflusses aufgestellt worden, kann die Präfix–Generierung für das modellierte System starten. Sie erfolgt weithin wie in Algorithmus 3.12 angegeben, hier jedoch lediglich entlang des Netzes des Kontrollflusses. Änderungen am Algorithmus gibt es nur bei der Berechnung erweiternder Ereignisse.

Zur Veranschaulichung soll wiederum das im letzten Kapitel betrachtete Beispiel des Peterson–Algorithmus 5.6 dienen. Abbildung 6.1 zeigt im linken Teil die Box des Kontrollflusses dieses Programms. Die Namensgebung der Netzknoten erklärt sich folgendermaßen: hochgestellte Indizes geben die Nummer des Prozesses an (1 bzw. 2), die

[1] In der Implementierung werden atomare Aktionen tatsächlich in Form eines Syntaxbaumes gespeichert, um die Evaluation zu beschleunigen. Darüber hinaus werden die in der Aktion vorkommenden Variablen separat für jede Transition verwaltet, und zwar unterteilt in Variablen, die einen Schreib– bzw. nur Lesezugriffe erfahren.

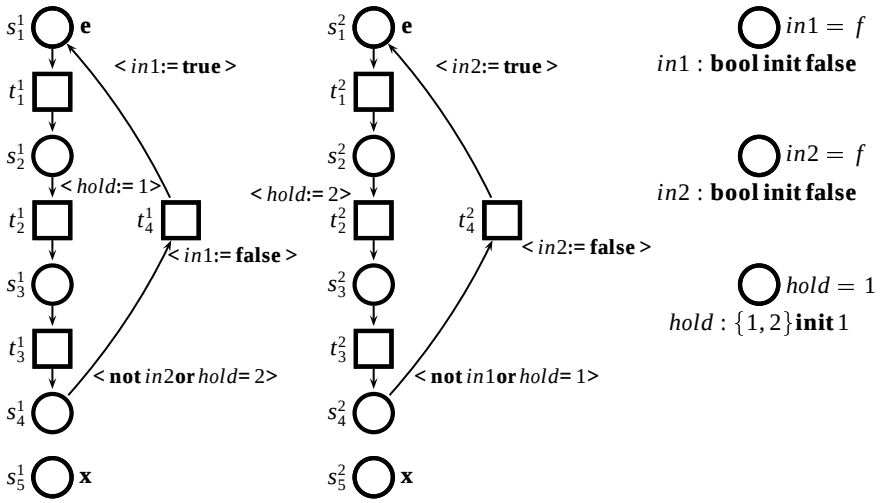


Abbildung 6.1: Kontrollfluss und Variablendeklarationen von Programm 5.6.

tiefgestellten eine fortlaufende Nummer innerhalb des jeweiligen Prozesses. Diese Notation ist zu unterscheiden von der für Werteänderungen von Variablen. Rechts im Bild sind die deklarierten Variablen dargestellt, zusammen mit den Startzuständen. Diese Stellenknoten werden (symbolisch) benötigt für das Übertragen der Startmarkierung in das zu erzeugende Präfix, dem ersten Schritt in Algorithmus 3.12. Aus Platzgründen werden die Booleschen Konstanten hier nur mit ihrem Anfangsbuchstaben geschrieben. Abbildung 6.2 zeigt ein mit der Ordnung $\sqsubset_{SND_1}$ erzeugtes Präfix der Entfaltung dieses Programms.[2]

Wie werden dabei nun das Präfix erweiternde Ereignisse ermittelt? Von der Startmarkierung aus sind z. B. die Transitionen t_1^1 und t_1^2 erreichbar; das noch nicht durchgeführte Scoping eingerechnet, sind sie dann aktiviert, wenn ihre atomaren Aktionen zu **true** ausgewertet werden können. Für t_1^1 ist das die Aktion **<** $in1$**:= true >**, die einzige Variable dieses Ausdrucks ist $in1$, die einen Schreibzugriff erfährt. Diese Aktion kann also stets ausgeführt werden (solange sich der zugewiesene Wert innerhalb des Definitionsbereichs der Variablen befindet).
Bezogen auf das Prozess–Netz gilt somit für ein Ereignis e, das mit t_1^1 beschriftet wird: Es müssen Bedingungsknoten im bereits berechneten Teil des Präfixes ermittelt werden, die der Datenbox von $in1$ entsprechen, und die in co–Relation stehen mit dem Vorbereich ${}^\bullet e$ (dies sind Bedingungen des Kontrollflusses). Für den allgemeinen Fall

[2] Bei der Konstruktion wird bereits, wo dies möglich ist, die Ersetzung von **or** in ein Exklusiv–Oder vorgenommen (abschaltbare Option). Die im synchronisierten Netz aus Abb. 5.7 gestrichelt dargestellten Teile haben deshalb keine Entsprechung in diesem Präfix. Außerdem ist festzustellen, dass es keine mit $in1_f^f$ bzw. $in2_f^f$ beschrifteten Ereignisse in diesem Präfix gibt, da die entsprechenden Transitionen im Netz aus Abb. 5.6 / 5.7 nie aktiviert werden.

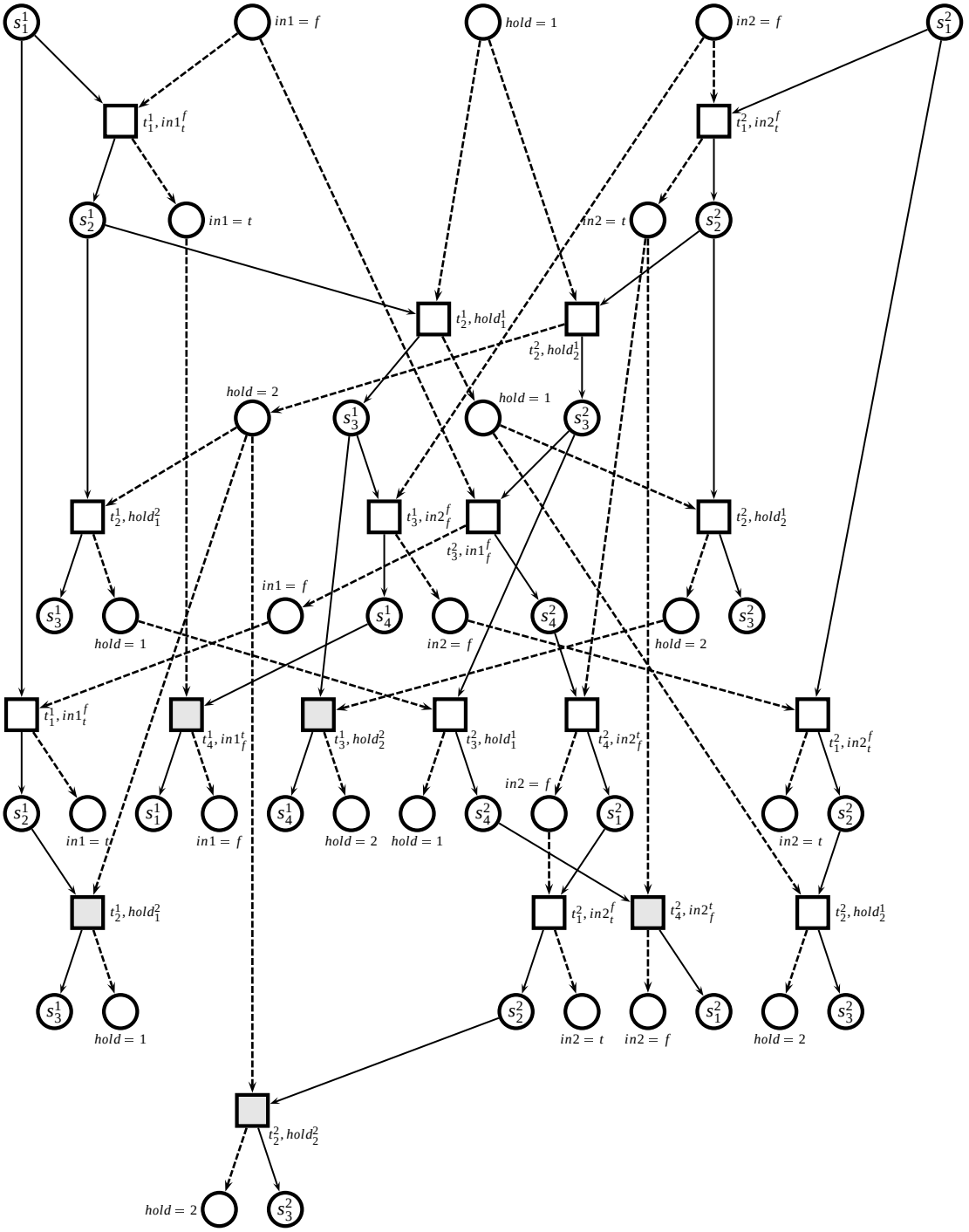


Abbildung 6.2: Präfix der Netzentfaltung des Systems aus Abb. 6.1.

muss zudem gelten, dass die Auswertung der atomaren Aktion mit den gefundenen Variablenwerten **true** ergibt; dies schließt ein, dass rechte Seiten von Zuweiungen oder Kanal–Schreibbefehlen innerhalb des Definitionsbereiches der Variablen liegen. Können solche Bedingungen ausfindig gemacht werden (wie in diesem Beispiel), ist ein erweiterndes Ereignis gefunden und kann in die Menge *erw* aus Algorithmus 3.12 aufgenommen werden.
Auf die gleiche Art und Weise erfolgt die Berechnung aller erweiternden Ereignisse. Beim Einfügen eines Ereignisknotens e in das Präfix berechnet sich dessen Nachbereich wie folgt: Der Kontrollfluss–Anteil ergibt sich unmittelbar aus dem Nachbereich der Transition $\pi(e)$; Bedingungen des Datenflusses werden für jede beteiligte Variable gewonnen durch Auswertung der atomaren Aktion – es werden Bedingungen erzeugt, die den Variablenwerten nach Ausführen von $\pi(e)$ entsprechen. In Abbildung 6.2 sind die aus dem Kontroll– bzw. Datenfluss resultierenden Kanten mit durchgehenden bzw. gestrichelten Linien gezeichnet.

Die Konstruktion des Präfixes allein entlang des Kontrollflusses ist möglich, da die Daten– und Kanalboxen eine sehr einfache Struktur besitzen (im Gegensatz zu [BH93]): jede Transition ist mit mindestens einem Aktionsnamen beschriftet, der mit Teilen des Kontrollflusses synchronisiert. Es gibt also keine Datenfluss–Transitionen, die – anschaulich gesprochen – „nebenläufig zum Kontrollfluss“ aktiviert sind.
Der umgekehrte Fall hingegen ist durchaus möglich: Transitionen des Kontrollflusses, die nicht mit dem Datenfluss synchronisieren müssen. Dies trifft auf alle (konstanten) Ausdrücke zu, die auf keine Variable zugreifen, etwa $<$**true**$>$ oder $<2=1+1>$. Die Übertragung solchen Ausdrüchen entsprechender Ereignisknoten in das Präfix kann auf die gleiche Weise wie im ursprünglichen Algorithmus durchgeführt werden, da bei ihnen kein Scoping stattfindet. Atomare Aktionen, die nicht zu **true** ausgewertet werden können, wie etwa $<$**false**$>$ oder $<4\#4>$, werden selbstverständlich nicht in das Präfix aufgenommen.

6.2 Eine totale Ordnung auf virtuellen Transitionen

Die grundsätzliche Struktur der DB(PN)^2–Präfix–Generierung ist im letzten Abschnitt aufgezeigt worden, einige wichtige Einzelheiten sind dabei ausgelassen worden und sollen nachfolgend diskutiert werden.
Für die Berechnung von endlichen, vollständigen Präfixen von Netzentfaltungen ist im vorderen Teil der Arbeit aufgezeigt worden, welche Kriterien ein Ereignis erfüllen muss, um als Cut-off erkannt zu werden. Wichtiges Instrument hierbei ist die Vergleichbarkeit der Ereignisse bezüglich der Ordnung $\sqsubset$. Insbesondere sind verschiedene Ordnungen vorgestellt worden, die zu einer Verbesserung gegenüber der von McMillan vorgeschlagenen führen. Allen diesen verbesserten Ordnungen ist gemein, dass sie ei-

ne totale Ordnung $\ll$ auf den Transitionen des Ausgangssystems voraussetzen. Eine solche wird also auch für die bei der DB(PN)^2–Präfix–Generierung verwendeten Ordnungen $\sqsubset_{SND_i}$ benötigt.

Allerdings liegt bei der hier betrachteten Konstruktion das Ausgangssystem zum einen unsynchronisiert vor, zum anderen wird der Datenfluss nicht explizit in Form von Netzen gespeichert. Zu Beginn der Präfix–Berechnung sind somit weder die eigentlich zu entfaltenden Transitionen bekannt, noch deren Zahl.
Es muss also ein Weg gefunden werden, die virtuellen und erst während der Präfix–Generierung tatsächlich ermittelten Transitionen aufzuzählen. Ruft man sich Def. 3.29 in Erinnerung, so ist zu erkennen, dass bei den SND–Silex–Ordnungen bereits eine komponentenweise totale Ordnung $\ll$ der Transitionen ausreicht. Eine solche Ordnung wird nachfolgend festgelegt.

Die Transitionen des Kontrollflusses lassen sich leicht ordnen, da das entsprechende Netz vor der Präfix–Generierung komplett vorliegt: die Knoten werden in beliebiger Reihenfolge mit einer fortlaufenden Nummer versehen, und zwar komponentenweise, d. h. einzeln für jeden Prozess. In Abbildung 6.1 ist dies bereits für den Peterson–Algorithmus so durchgeführt worden.
Beim Datenfluss spiegelt eine Daten– bzw. Kanalbox einer Variablen gerade eine Komponente des Netzsystems wider. Die Ordnung der lokalen (virtuellen) Transitionen einer Datenbox kann entsprechend den Werteänderungen vorgenommen werden. Für skalare –, Tupel– bzw. Feldvariablen (mit Feldgröße n) können die Transitionen der entsprechenden Datenboxen mit folgenden Ordnungen eindeutig sortiert werden:

$$
\begin{array}{rcll}
\widehat{v_b^a} & \ll & \widehat{v_d^c} & \text{gdw.} \quad a < c \vee (a = c \wedge b < d) \\
\overbrace{t_{(b_1,b_2)}^{(a_1,a_2)}} & \ll & \overbrace{t_{(d_1,d_2)}^{(c_1,c_2)}} & \text{gdw.} \quad a_1a_2 <_{lex_<} c_1c_2 \vee (a_1a_2 = c_1c_2 \wedge b_1b_2 <_{lex_<} d_1d_2) \\
\overbrace{a_{[b_1,\ldots,b_n]}^{[a_1,\ldots,a_n]}} & \ll & \overbrace{a_{[d_1,\ldots,d_n]}^{[c_1,\ldots,c_n]}} & \text{gdw.} \quad a_1 \ldots a_n <_{lex_<} c_1 \ldots c_n \\
 & & & \quad \vee (a_1 \ldots a_n = c_1 \ldots c_n \wedge b_1 \ldots b_n <_{lex_<} d_1 \ldots d_n)
\end{array}
$$

Innerhalb Boolescher Datenbereiche wird dabei **false** $<$ **true** vereinbart.

Bei Kanalboxen werden ebenfalls die Werte–, besser Zustandsänderungen für die Definition einer Ordnung herangezogen; für einen Kanal c reichen die Aktionsnamen $\widehat{c!\,v}, \widehat{c?\,v}, \widehat{c!!}, \widehat{c??}$ nicht aus, da sie mehrdeutig Transitionen zugeordnet sein können. Zustandsänderungen bestehen bei Zugriffen auf Kanalvariablen aus den Inhalten des Kanals vor bzw. nach Schalten der Transition. In Abbildung 6.3 sind die Transitionen einer zweiwertigen Kanalvariablen der Kapazität 2 entsprechend mit $c^{\text{vorher}}_{\text{nachher}}$ beschriftet.

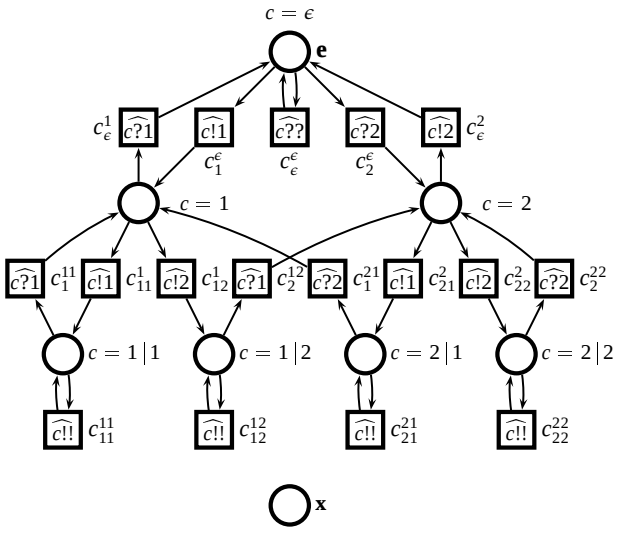


Abbildung 6.3: Ordnung der Transitionen der Kanalbox zu ***var*** *c:* ***chan*** 2 ***of*** *{*1, 2*}*.

Über diese Namen können die Transitionen wie folgt total geordnet werden:

$$c_v^u \ll c_x^w \quad \text{gdw.} \quad u <_{silex_<} w \ \vee \ (u = w \wedge v <_{silex_<} x)$$

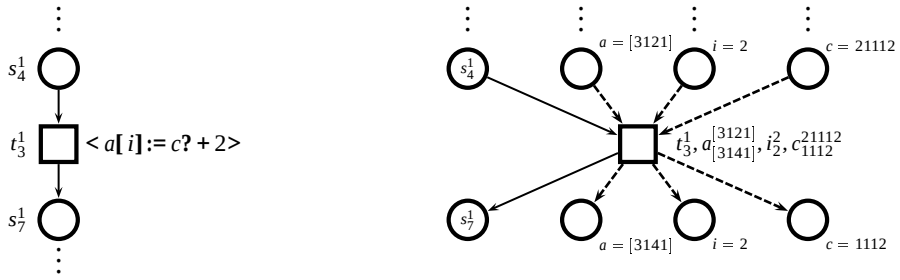
Im obigen Beispiel ergibt sich damit nachstehende Sortierung:

$$c_\epsilon^\epsilon \ll c_1^\epsilon \ll c_2^\epsilon \ll c_\epsilon^1 \ll c_{11}^1 \ll c_{12}^1 \ll c_\epsilon^2 \ll c_{21}^2 \ll c_{22}^2$$
$$\ll c_1^{11} \ll c_{11}^{11} \ll c_2^{12} \ll c_{12}^{12} \ll c_1^{21} \ll c_{21}^{21} \ll c_2^{22} \ll c_{22}^{22}$$

Falls andere adäquate Ordnungen als $\sqsubset_{SND_i}$ für die Präfix–Generierung von Programmen verwendet werden sollen, kann ebenfalls auf die soeben definierten Ordnungen $\ll$ zurückgegriffen werden. Daten– und Kanalboxen bilden in jedem Fall eigene Komponenten; diese können mit einer laufenden Nummer versehen werden. Für die Ermittlung einer globalen totalen Ordnung $\ll$ über allen Transitionen werden selbige mit einem Namen versehen, der sich aus einer komponentenweisen Konkatenation der lokalen Namen ergibt. Diese globalen Namen können dann lexikographisch sortiert werden.

6.3 Evaluation atomarer Aktionen

Zentrale Bedeutung bei der DB(PN)^2–Präfix–Generierung kommt der Auswertung von Ausdrücken atomarer Aktionen zu, über die die Ermittlung erweiternder Ereignisse erfolgt. Mit Hilfe des Netzes des Kontrollflusses wird festgestellt, welche Transitionen *t*



(a) Ausschnitt aus Box.

(b) Entsprechender Ausschnitt aus Präfix.

Abbildung 6.4: Schema der Ermittlung eines erweiternden Ereignisses.

für eine Erweiterung in Frage kommen. Dies erfolgt durch Abgleich der maximalen Bedingungen des konstruierten Teils des Präfixes über die Abbildung π mit den entsprechenden Stellenknoten im Kontrollfluss.
Bei den so ermittelten Transitionen t wird die angeheftete atomare Aktion inspiziert und die im Ausdruck vorkommenden Variablen werden festgestellt. Abbildung 6.4(a) zeigt hierzu schematisch ein Beispiel für die Anweisung **< *a*[*i*] := *c*? + 2>** , die Teil eines nicht näher spezifizierten Programmes sei. Die enthaltenen Variablen sind *a*, *i* und *c*, die wie folgt deklariert sind:

var *i*: **int** ; **var** *a*: **array** 4 **of int** ; **var** *c*: **chan** ω **of int** ;

Für diese drei Variablen müssen im bereits konstruierten Teil des Präfixes Bedingungsknoten ermittelt werden, die über die Abbildung π mit einer (virtuellen) Stelle der entsprechenden Daten– bzw. Kanalbox beschriftet sind. Werden die zu einer Netzkomponente gehörenden Knoten mit einer eindeutigen Nummer versehen (wie dies bei Verwendung der SND–Ordnungen zur Berechnung der lokalen *i*–Sichten bereits geschieht), ist dies leicht durchführbar.
Anschließend müssen die möglichen Kombinationen von Bedingungsknoten – eine oder mehrere[3] Bedingungen aus dem Maximum des Präfixes, die mit dem Kontrollfluss korrespondieren, sowie je eine Bedingung aus der Entfaltung des Datenflusses der beteiligten Variablen – daraufhin getestet werden,

- ob sie eine co–Menge bilden und
- ob die Auswertung des Ausdrucks unter der gefundenen Variablenbelegung **true** liefert.

[3] Der Fall, dass mehrere Bedingungen des Kontrollflusses im Vorbereich eines erweiternden Ereignisses liegen, kommt nur bei der Verwendung von Kanälen der Kapazität 0 vor sowie im Rahmen der Variablen–Deinitialisierung.

Falls beide Bedingungen zutreffen, kann ein entsprechendes Ereignis in die weitere Konstruktion des Präfixes eingehen, wie es beispielhaft Abbildung 6.4(b) zeigt. Die Beschriftung der Bedingungen im Nachbereich des neuen Ereignisses ergeben sich gemäß der Auswertung des Ausdrucks; für jede beteiligte Variable wird dabei eine Bedingung eingefügt, auch wenn diese ihren Wert nicht ändert. Auf diese Weise wird das Scoping über Aktionsnamen während der Konstruktion des Präfixes durchgeführt. Zu beachten ist, dass die SND–Struktur des Netzes gewahrt bleibt.

6.4 Programmvariablen und Deinitialisierung

Bislang unberücksichtigt bei der Konstruktion des Präfixes der Netzentfaltung eines DB(PN)^2–Programms ist die Deinitialisierung von Variablen geblieben, deren Notwendigkeit in Kapitel 5.2.3 begründet wurde. Beim Beispiel des Peterson–Algorithmus ist eine Deinitialisierung nicht erforderlich, da es sich um ein reaktives System handelt, die einzelnen Prozesse befinden sich in Endlosschleifen.
Um die Konstruktion des Präfixes innerhalb der Hauptschleife einfach zu halten und keinen Sonderfall für die Deinitialisierung zu schaffen, wird ein kleiner Kunstgriff vorgenommen: Im Netz des Kontrollflusses wird im Deinitialisierungsteil, siehe Abbildung 5.4 auf Seite 110, bereits vor Beginn der Entfaltungsprozedur ein Scoping über den Aktionsnamen p_i, $1 \leq i < n$, durchgeführt; dabei entsteht eine einzelne Transition, die beschriftet ist mit je einem Deinitialisierungs–Aktionsnamen $\widehat{v}$ für jede vorkommende Programmvariable v. Diese Transition kann bei der Suche nach erweiternden Ereignissen auf die gleiche Weise berücksichtigt werden wie die übrigen Transitionen des Kontrollflusses (auch wenn die Beschriftung syntaktisch keiner atomare Aktion entspricht).

Damit ist theoretische Beschreibung der Berechnung vollständiger, endlicher Präfixe von Netzsystemen abgeschlossen. Als nächstes sollen die mehr praktischen Aspekte dieses Verfahrens beleuchtet werden, dazu zeigt zunächst das folgende Kapitel Anwendungen auf, in denen die Entfaltungsmethode für die Untersuchung von nebenläufigen Systemen genutzt wird. Später wird in 8 ausführlich auf konkrete Aspekte bei der Implementierung der in den letzten Kapiteln beschriebenen Algorithmen eingegangen.

Kapitel 7

„Je vertrauter und alltäglicher eine Verhaltensweise ist,
desto problematischer wird ihre Analyse.“
DESMOND MORRIS
LIEBE GEHT DURCH DIE HAUT (1972), 10

Verifikation auf Basis des Präfixes

Nachdem in den letzten Kapiteln ausführlich die Konstruktion von endlichen, vollständigen Präfixen der Netzentfaltung beschrieben worden ist, soll nun untersucht werden, auf welche Weise ein solches Präfix bei der Verifikation von verteilten Systemen genutzt werden kann. Unter Verifikation wird dabei der Vorgang des Überprüfens von Systemen verstanden, in dem Sinne, ob die Eigenschaften, die ursprünglich spezifiziert wurden, vom Modell und dessen Design auch tatsächlich erfüllt werden.

Grundsätzlich werden zwei Methoden der Verifikation unterschieden. Eine einfache Möglichkeit bietet die Untersuchung lediglich einer geeigneten Teilmenge möglicher Ausführungen; eine solche Teilmenge kann z. B. durch Simulationen gewonnen werden. Eigenschaften werden dann aufgrund dieser empirisch ermittelten Daten überprüft. Diese Methode ist damit vergleichbar mit dem systematischen Testen eines Programms.
Die aufwändigere Variante besteht in der Untersuchung des gesamten Zustandsraumes, aus technischer Sicht ist sie somit nicht anwendbar für sehr große Systeme wegen des Problems der Zustandsraum–Explosion.

Aus diesem Grunde ist die Simulation auch die traditionelle, weiter verbreitete Vorgehen. Erst in den letzten Jahren ist das Interesse größer geworden, formale Methoden einzusetzen. Dieses Kapitel wird sich ganz den Verifikationsmethoden widmen, die den gesamten Zustandsraum für die Überprüfung von Eigenschaften zu Rate ziehen, wobei dieser in Form des vollständigen Präfixes der Netzentfaltung vorliegt. Es werden hier nur Verfahren für 1–sichere Netzsysteme diskutiert, da dies die in der Praxis am häufigsten verwendete Netzklasse darstellt, die ein weites Spektrum von Anwendungen abdeckt. Schließlich garantiert der Einsatz der Sprache DB$(PN)^2$ eine

1–sichere Netzsemantik, so dass alle diese Verfahren für die Verifikation von DB(PN)^2–Programmen herangezogen werden können.
In [Lam77] wird eine Unterteilung von Eigenschaften vorgenommen in Sicherheits– und Lebendigkeits–Eigenschaften. Sicherheitseigenschaften beschreiben danach Aussagen der Art „Nichts Schlechtes wird jemals eintreten“, das Erreichen unerwünschter Zustände kann damit ausgeschlossen werden. Dazu zählen Eigenschaften wie Erreichbarkeit (etwa beim gegenseitigen Ausschluss) und Verklemmungsfreiheit, aber auch Schleifeninvarianten, (partielle) Korrektheit oder, auf Netzebene, die Lebendigkeit einzelner Transitionen. Lebendigkeitsaussagen wiederum charakterisieren Aussagen der Form „Etwas Gutes wird irgendwann eintreten“. Beispiele hierfür sind Terminierung, das Beantworten von Anfragen sowie das Vermeiden des Aushungerns einzelner Prozesse.

Im weiteren Verlauf werden diverse Algorithmen vorgestellt, die Erreichbarkeit von Systemzuständen und Verklemmungsfreiheit untersuchen. Trotz der offenbar deckungsgleichen Funktionalität hat jedes einzelne dieser Verfahren seine Berechtigung. Keiner der Algorithmen kann als universell in dem Sinne angesehen werden, dass mit ihm allein das entsprechende Problem für beliebig strukturierte Netze stets auf die effizienteste Weise gelöst werden kann.
Zum Nachweis frei spezifizierbarer Eigenschaften werden Model–Checker unterschiedlicher Logiken vorgestellt. Alle beschriebenen Algorithmen sind bereits implementiert und Bestandteil verschiedener Werkzeuge zur Modellierung und Analyse verteilter Systeme. Die mehr praktisch orientierten Aspekte der Analyseverfahren werden später in Kapitel 9 näher beleuchtet.

7.1 Erreichbarkeit von Markierungen

Das Erreichbarkeitsproblem besteht allgemein darin, festzustellen, ob ein Netzsystem einen bestimmten Zustand einnehmen kann, d. h., ob eine bestimmte Markierung durch Schalten einer Folge von Transitionen von der Startmarkierung aus erreicht werden kann, siehe auch Notation 2.10. Während die Komplexität des allgemeinen Erreichbarkeitsproblems PSPACE–vollständig ist in der Größe des Systems [EN94], ist es lediglich NP–vollständig in der Präfix–Größe.

Bei den zu untersuchenden Markierungen können zwei Arten unterschieden werden: globale Systemzustände und Teilmarkierungen. Bei ersteren bestimmt die zu untersuchende Markierung vollständig, welche Stellen markiert sein sollen und welche nicht. Teilmarkierungen hingegen geben an, welche Stellen mindestens gleichzeitig in einem Zustand markiert sein sollen oder nicht, über andere Stellen wird keine Aussage getroffen.

Nachfolgend werden verschiedene Ansätze diskutiert, das Erreichbarkeitsproblem für sichere Netzsysteme auf Basis des vollständigen Präfixes zu lösen. Als Ergebnis der Analyse kann neben einer reinen Ja / Nein–Aussage häufig die gefundene Schaltsequenz von Interesse sein.

Erreichbarkeitsanalyse während der Präfix–Generierung McMillan hat in [McM92] als Anwendungsbeispiel der Entfaltungstechnik ein einfaches Vorgehen zur Lösung allgemeiner Erreichbarkeitsprobleme vorgeschlagen: durch Hinzufügen einer neuen Transition t zum Netzsystem $\Sigma = (S, T, W, M_0)$, die genau dann aktiviert wird, wenn die gesuchte (Teil–)Markierung $M \subseteq S$ erreicht wird, es ist also ${}^\bullet t = M$. Wenn die Entfaltung (und damit ein vollständiges Präfix) ein mit t beschriftetes Ereignis enthält, ist die gesuchte (Teil–)Markierung als erreichbar erkannt worden.
Ist lediglich das Erreichen einer bestimmten (Teil–)Markierung von Interesse, so kann die Präfix–Generierung bereits unmittelbar nach Erzeugen eines entsprechend beschrifteten Ereignisses abgebrochen werden, die Erreichbarkeit wäre damit nachgewiesen. Bei Nicht–Erreichen würde das vollständige Präfix konstruiert.[1]
Die Forderung, dass bestimmte Stellen unmarkiert im gesuchten Zustand vorkommen sollen, kann mit diesem Verfahren im Allgemeinen nicht berücksichtigt werden, da über das Aktivieren von t nicht nachweisbar ist, dass bestimmte Stellen gleichzeitig unmarkiert sind.
Eine Lösung dieses Problems bietet die folgende Strategie: Jedes sichere System kann unter Erhalt des ursprünglichen Verhaltensi transformiert werden (in ein SND–System), indem zu jeder Stelle $s \in S$ deren Komplement s^c (falls noch nicht vorhanden) in das Netz aufgenommen wird, das sich durch Umdrehen der Kanten ergibt; lediglich an Schlingen beteiligte Kanten werden nicht für das Komplement übernommen, siehe auch [SE00].
Auf diese Weise können diejenigen Stellen, die in der gesuchten Markierung M unmarkiert sein sollen, einfach durch Angabe des jeweils komplementären Knotens spezifiziert werden. Für die Untersuchung kann auf die Präfix–Generierung mit Hilfe einer der SND–Ordnungen zurückgegriffen werden. Die Schaltsequenz, die zu M führt, kann leicht über $\pi(\mathsf{Lin}_\prec([e]))$ gewonnen werden, wobei e das zuletzt in das Präfix eingfügte Ereignis (mit $\pi(e) = t$) ist.

Die Aufnahme der zusätzlich erforderlichen Stellen–Komplemente zieht die Konstruktion eines größeren Präfixes nach sich. Für die Suche nach einer einzelnen, ganz bestimmten Markierung könnten vereinfachend nur die Komplemente solcher Stellen eingefügt werden, die für diese Markierung von Belang sind. Soll die Frage der Erreichbarkeit innerhalb eines festen Netzsystems universell für eine beliebige Zahl von

[1] Diese einfache Methode zur Überprüfung der Erreichbarkeit ist direkt in die Implementation des Präfix–Generators integriert, siehe auch Anhang A.2 (Option `-X`).

Markierungen beantwortet werden, ist das wiederholte Konstruieren des Präfixes im Allgemeinen kein effizientes Vorgehen. Für diesen Fall sind Algorithmen geeigneter, die auf einem einmal generierten Präfix die Untersuchungen vornehmen können; solche werden als nächstes beschrieben.

Erreichbarkeitsanalyse mit Hilfe graphentheoretischer Überlegungen Aus den Überlegungen aus Kapitel 3.2 zu Konfigurationen und Schnitten ist bekannt, dass jeder B–Schnitt durch das Präfix einer erreichbaren Markierung entspricht. Ebenso lässt sich leicht nachweisen, dass nicht–maximale co–Mengen von Bedingungen erreichbare Teilmarkierungen repräsentieren, da sie sich zu B–Schnitten erweitern lassen [NPW80, BF88]. Von dieser Beobachtung ausgehend lässt sich ein einfacher Algorithmus formulieren, mit dem sich die Frage der Erreichbarkeit testen lässt.

Eine Markierung $M = \{s_1, \ldots, s_n\} \subseteq S$ ist genau dann von M_0 aus erreichbar, wenn die Elemente eines der n–Tupel aus $\pi^{-1}(s_1) \times \ldots \times \pi^{-1}(s_n)$ eine co–Menge bilden. Zum Auffinden einer solchen Menge müssen lediglich alle Kombinationen von Bedingungen auf die co–Mengen–Eigenschaft hin getestet werden.[2] Das Erreichbarkeitsproblem ist somit reduzierbar auf das Finden einer Clique in einem n–partiten Graph. Ist die co–Relation des Präfixes in Form einer Matrix[3] gespeichert, so dass in $O(1)$ die co–Eigenschaft für zwei Knoten entschieden werden kann, dann beträgt die Zeit–Komplexität dieses Algorithmus $O((\frac{|B|}{n})^{|M|} \cdot \frac{|M|^2}{2})$, also Anzahl möglicher Tupel multipliziert mit der Zahl der maximal notwendigen co–Tests.
Wiederum lässt sich dieses Verfahren nur für Markierungen anwenden, die lediglich aus markierten Stellen bestehen. Sollen in der Markierung auch Zusicherungen über unmarkierte Stellen enthalten sein, muss das Netz wie zuvor beschrieben um Stellen–Komplemente erweitert werden.

Erreichbarkeitsanalyse mit Hilfe von Logik–Programmen Einen Logik–basierten Ansatz zur Erreichbarkeitsanalyse auf Präfixen beschreibt Heljanko in [Hel99b]. Grundidee hierbei ist es, die Netzstruktur des Präfixes zusammen mit der zu untersuchenden Markierung als eine Reihe logischer Klauseln aufzufassen und anschließend zu überprüfen, ob es ein (stabiles) Modell gibt, welches das so erhaltene Logik–Programm erfüllt. Im Grunde lässt sich das Erreichbarkeitsproblem somit reduzieren auf das Erfüllbarkeitsproblem (SAT). Für die Ermittlung der Modelle wird auf Methoden der (regelbasierten) Constraint–Programmierung und die dort vorhandenen Werkzeuge zurückgegriffen.

[2] Ein solcher Algorithmus ist in abgewandelter Form bereits für die Konstruktion des Präfixes notwendig: bei der Berechnung möglicher erweiternder Ereignisse. Siehe hierzu auch die Ausführungen in Kapitel 8.2.5 und darüber hinaus [Mel98, SE00].

[3] Siehe hierzu Kapitel 8.2.4.

Das Verfahren ist gleichsam für Teil– wie für vollständige Markierungen anwendbar. Die Stellenknoten, für die die (Nicht–)Erreichbarkeit geprüft werden soll, sind nachfolgend mit $M^+ \subseteq S$ für die erreichbaren und $M^- \subseteq S$ für die nicht erreichbaren disjunkt aufgeteilt. Untersucht wird damit das Erreichen der Zusicherung $< M^+, M^- >$.
Die logischen Klauseln besitzen allgemein die Form $h \leftarrow a_1, \ldots, a_n$, wobei die a_i propositionale Atome darstellen, die mit den entsprechend benannten Knoten des Netzsystems bzw. des Präfixes korrespondieren und zur Vereinfachung mit ihnen identifiziert werden. Sind alle Atome a_i, $1 \leq i \leq n$, in einem Modell, dann auch das Atom h. Atome können auch negiert auf der rechten Seite vorkommen, dies wird durch Überstrich (z. B. $\overline{a_i}$) notiert.
Vereinfachend wird im Folgenden eine Mengenschreibweise für Gruppen von Atomen verwendet: die Menge $\{a_1, \ldots, a_n\}$ steht also für $a_1, \ldots, a_n$, die Negation einer Menge wird dabei auf alle Elemente angewendet: $\overline{\{a_1, \ldots, a_n\}}$ bezeichnet die Atome $\overline{a_1}, \ldots, \overline{a_n}$. Darüber hinaus werden einige Hilfsatome benötigt, die anzeigen, dass Ereignisse $e \in E$ nicht in der gesuchten Schaltfolge vorkommen; sie werden nachfolgend *ne* geschrieben. Als weitere Abkürzung werden einige spezielle Schreibweisen verwendet, die sich einfach in die obige allgemeine Form von Klauseln umwandeln lassen [Hel99b]: Die linke Seite einer Klausel kann leer sein, in diesem Fall soll die Integrität der Atome auf der rechten Seite sichergestellt werden. Auf der rechten Seite symbolisiert das Voranstellen der Ziffer 2 vor eine Menge von Atomen (wie etwa in $h \leftarrow 2\{a_1, \ldots, a_n\}$), dass, wenn mindestens zwei der Atome a_i, $1 \leq i \leq n$, zu einem Modell gehören, dann auch h.
Der folgende Satz beschreibt, wie das Logik–Programm zur Untersuchung von Erreichbarkeitsfragen aussehen muss.

Satz 7.1 *Erreichbarkeitsanalyse auf dem Präfix mittels Logik–Programmierung [Hel99b]*

Sei $\Sigma = (\mathcal{N}, M_0)$ ein sicheres Netzsystem und sei $(\mathcal{O}, \pi) = \mathsf{Pre}(\mathsf{Unf}(\Sigma))$ mit $\mathcal{O} = (B, E, F)$ vollständiges Präfix der Entfaltung von Σ. Die (partielle) Markierung (Zusicherung) $< M^+, M^- >$ ist genau dann von M_0 aus erreichbar, wenn das folgende Logik–Programm $\mathcal{P}$ ein stabiles Modell besitzt.

$$
\left.\begin{array}{rcl} e_i & \leftarrow & {}^{\bullet\bullet}e_i, \overline{ne_i} \\ ne_i & \leftarrow & \overline{e_i} \end{array}\right\} \text{ für alle } e_i \in E \setminus E_{cutoff} \quad (1)
$$

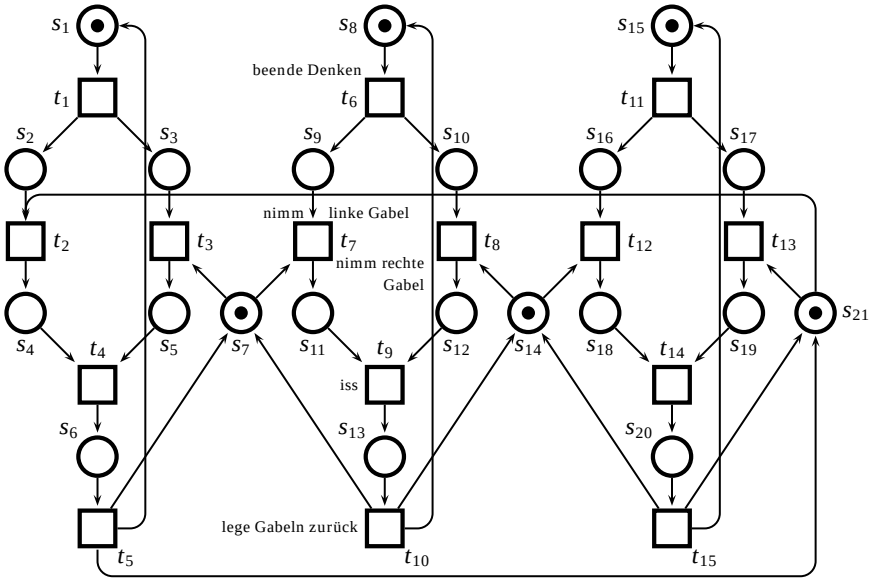
$$
\leftarrow \; 2(b_i^\bullet \setminus E_{cutoff}) \qquad \text{für alle } b_i \in B \text{ mit } |b_i^\bullet \setminus E_{cutoff}| \geq 2 \quad (2)
$$

$$
\left.\begin{array}{rcl} b_i & \leftarrow & {}^\bullet b_i, \overline{(b_i^\bullet \setminus E_{cutoff})} \\ \pi(b_i) & \leftarrow & b_i \end{array}\right\} \text{ für alle } b_i \in B \text{ mit } \pi(b_i) \in M^+ \cup M^- \wedge {}^\bullet b_i \in E_{cutoff} \quad (3)
$$

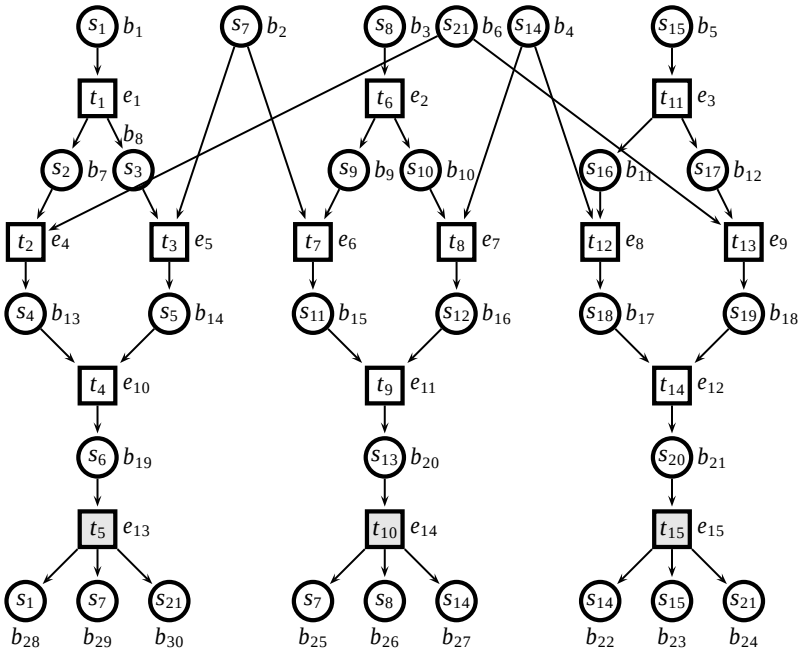
$$
\leftarrow \; \overline{s_i} \qquad \text{für alle } s_i \in M^+ \quad (4)
$$

$$
\leftarrow \; s_i \qquad \text{für alle } s_i \in M^- \quad (5)
$$

Für jedes stabile Modell $\Delta_\mathcal{P}$ von $\mathcal{P}$ gibt die Restriktion $C = \Delta_\mathcal{P}|_E$ eine Konfiguration an (d. h. es ist $C \in \mathcal{C}_\mathcal{O}$), die die Zusicherung $< M^+, M^- >$ impliziert. Mit anderen Worten gilt $M^+ \subseteq \mathsf{Mark}(C)$ und $M^- \cap \mathsf{Mark}(C) = \emptyset$. ■*7.1*



(a) Netzsystem



(b) Präfix der Entfaltung

Abbildung 7.1: Beispiel: Die drei speisenden Philosophen.

Die Regeln (1) und (2) des Programms $\mathcal{P}$ beschreiben die Netzstruktur des Präfixes: Regeln (1) spiegeln die (partielle) Kausal– und (2) die (partielle) Konfliktrelation wider. Die Constraints (3) geben die Schaltregel der lokalen Umgebung der gesuchten Markierung sowie den (partiellen) Netzhomomorphismus π an. Schließlich spezifizieren die Constraints (4) und (5) die Zusicherung $< M^+, M^- >$. Bei der Aufstellung des Logik–Programms brauchen Cut-off–Ereignisse und deren Nachbereiche nicht berücksichtigt zu werden, da alle erreichbaren Markierungen bereits im übrigen Teil des Präfixes repräsentiert sind.
Ein Beispiel soll diese Konstruktion verdeutlichen. Dazu wird wiederum auf das Netzbeispiel aus Abbildung 7.1 zurückgegriffen. Das Logik–Programm zur Untersuchung der Erreichbarkeit der Zusicherung $< \{s_1, s_3, s_9\}, \{s_{10}\} >$ hat die folgende Gestalt:

$$
\begin{array}{lcl}
e_1 & \leftarrow & \overline{ne_1} \\
e_2 & \leftarrow & \overline{ne_2} \\
e_3 & \leftarrow & \overline{ne_3} \\
e_4 & \leftarrow & e_1, \overline{ne_4} \\
e_5 & \leftarrow & e_1, \overline{ne_5} \\
e_6 & \leftarrow & e_2, \overline{ne_6} \\
e_7 & \leftarrow & e_2, \overline{ne_7} \\
e_8 & \leftarrow & e_3, \overline{ne_8} \\
e_9 & \leftarrow & e_3, \overline{ne_9} \\
e_{10} & \leftarrow & e_4, e_5, \overline{ne_{10}} \\
e_{11} & \leftarrow & e_6, e_7, \overline{ne_{11}} \\
e_{12} & \leftarrow & e_8, e_9, \overline{ne_{12}}
\end{array}
\qquad
\begin{array}{lcl}
ne_1 & \leftarrow & \overline{e_1} \\
ne_2 & \leftarrow & \overline{e_2} \\
ne_3 & \leftarrow & \overline{e_3} \\
ne_4 & \leftarrow & \overline{e_4} \\
ne_5 & \leftarrow & \overline{e_5} \\
ne_6 & \leftarrow & \overline{e_6} \\
ne_7 & \leftarrow & \overline{e_7} \\
ne_8 & \leftarrow & \overline{e_8} \\
ne_9 & \leftarrow & \overline{e_9} \\
ne_{10} & \leftarrow & \overline{e_{10}} \\
ne_{11} & \leftarrow & \overline{e_{11}} \\
ne_{12} & \leftarrow & \overline{e_{12}}
\end{array}
\qquad
\begin{array}{lcl}
 & \leftarrow & e_5, e_6 \\
 & \leftarrow & e_7, e_8 \\
 & \leftarrow & e_4, e_9
\end{array}
\qquad
\begin{array}{lcl}
b_7 & \leftarrow & e_1, \overline{e_4} \\
b_8 & \leftarrow & e_1, \overline{e_5} \\
b_9 & \leftarrow & e_2, \overline{e_6} \\
b_{10} & \leftarrow & e_2, \overline{e_7} \\
s_2 & \leftarrow & b_7 \\
 & \leftarrow & \overline{s_2} \\
s_3 & \leftarrow & b_8 \\
 & \leftarrow & \overline{s_3} \\
s_9 & \leftarrow & b_9 \\
 & \leftarrow & \overline{s_9} \\
s_{10} & \leftarrow & b_{10} \\
 & \leftarrow & s_{10}
\end{array}
$$

Die Kosten für das Aufstellen des Logik–Programm sind linear in der Größe des Präfixes; dabei können Optimierungen vorgenommen werden, z. B. durch Entfernen mehrfacher Vorkommen der gleichen Klausel. Das Problem, ein stabiles Modell für $\mathcal{P}$ zu ermitteln, ist NP–vollständig, jedoch gibt es effiziente, rechnergestützte Werkzeuge zur Berechnung [Hel99b].
Ein stabiles Modell des obigen Programms ist $\{e_1, e_2, e_3, ne_4, ne_5, ne_6, e_7, ne_8, ne_9, ne_{10}, ne_{11}, ne_{12}, b_7, s_2, b_8, s_3, b_9, s_{10}\}$. Die damit errechnete Konfiguration $\{e_1, e_2, e_3, e_7\}$ korrespondiert mit der Schaltfolge $t_1 t_6 t_{11} t_8$, die zur gesuchten Markierung führt.

7.2 Auffinden von Verklemmungen

Das Vermeiden von Verklemmungen ist eine der zentralen Anforderungen an verteilte, nebenläufige Systeme. Nachfolgend werden einige Verfahren angegeben, mit denen sich Verklemmungen mit Hilfe des Präfixes der Netzentfaltung ermitteln lassen. Wie schon das Erreichbarkeitsproblem ist auch das Ermitteln von Verklemmungen PSPACE–vollständig in der Größe des Ausgangssystems und NP–vollständig in der Größe des vollständigen Präfixes [McM92, McM95].

Der Algorithmus von McMillan In [McM92] wird ein Verfahren zur Ermittlung von Verklemmungen auf der Basis von Netzentfaltungen beschrieben. Die Idee des Algorithmus ist es, die Konfigurationen zu untersuchen. Die grundlegende Beobachtung besteht darin, dass ein System genau dann keine Verklemmung aufweist, wenn jede (globale) Konfiguration des Präfixes erweitert werden kann zu einer Konfiguration, die wenigstens ein Cut-off–Ereignis enthält. In diesem Fall nämlich kann die Konfiguration unendlich oft erweitert werden.
Mit anderen Worten: ein Netzsystem enthält genau dann eine Verklemmung, wenn es eine Konfiguration gibt, die in Konflikt mit jedem Cut-off-Ereignis der Entfaltung steht. Zu diesem Zweck wird der Begriff *Spoiler* eingeführt: die Menge der Spoiler eines Ereignisses e_c ist definiert als $S_{e_c} = \{e \in E \mid e \# e_c\} \cap [e_c]^{\bullet\bullet}$, sie enthält alle Ereignisse, die in unmittelbarem Konflikt stehen mit einem Ereignis aus der lokalen Konfiguration von e_c.
Der Algorithmus versucht eine solche Konfiguration konstruktiv über ein *Branch-and-bound*–Verfahren mittels Backtracking zu ermitteln. Die berechnete Menge S enthält am Schluss diejenigen Spoiler, die mit jedem Cut-off–Ereignis des Präfixes in Konflikt stehen. Existiert eine solche nicht–leere Menge S, dann kann sie erweitert werden zu einer Konfiguration, die zu einer Verklemmung des Systems führt; mittels Linearisierung ergibt sich eine entsprechende Schaltsequenz zum Eintritt dieser Verklemmung, diese ist $\pi(\mathsf{Lin}_{\prec}(S))$. Terminiert die Berechnung andererseits mit $S = \emptyset$, dann ist das System verklemmungsfrei.
Der Algorithmus besitzt die folgende Struktur:

Algorithmus 7.2 *Test auf Verklemmungsfreiheit [McM92]*

Eingabe: Präfix $\mathcal{B} = (B, E, F)$ mit Cut-off–Ereignissen $E_{cutoff} \subseteq E$.
Ausgabe: Menge S, die alle zu E_{cutoff} in Konflikt stehenden Spoiler enthält.

```
1    S := ∅; E_c := E_cutoff;
2    while E_c ≠ ∅ do
3      e_c := Element aus E_c mit der kleinsten Zahl an Spoilern;
4      if e_c besitzt Spoiler then
5        Wähle ein Ereignis e aus den Spoilern von e_c;
6        S := S ∪ {e};
7        Lösche alle Ereignisse in Konflikt zu S
8      else
9        Springe zurück zur letzten Wahl eines Spoilers
10         und stelle die gelöschten Ereignisse wieder her
11     endif
12   endwhile                                              ■7.2
```

Der Algorithmus ist im schlechtesten Fall exponentiell in der Größe des Präfixes (Zahl der Ereignisse), bedingt durch das Backtracking. Durch das Betrachten der Ereignisse

mit minimaler Spoilerzahl kann die Zahl der in Frage kommenden Ereignisse rasch eingeschränkt werden. Im Mittel wird die Berechnung durch diese einfache Heuristik erheblich beschleunigt. Eine effizente Umsetzung dieses Algorithmus wird in [MR97] mit dem nachfolgend beschriebenen Verfahren verglichen.

Ermittlung von Verklemmungen mittels Linearer Programmierung In [MR97] wird ein linear–algebraischer Ansatz vorgestellt, der auf Basis des endlichen Präfixes der Entfaltung ein Netzsystem auf Verklemmungsfreiheit hin testet. Das Verfahren überträgt die sogenannte *Markierungsgleichung* [Mur89, DE95] auf die azyklische Struktur des Präfixes zur Untersuchung von Erreichbarkeitsfragen.[4]
Sei $\Sigma = (\mathcal{N}, M_0)$ mit $\mathcal{N} = (S, T, F)$ ein sicheres Netzsystem und M mit $M_0 \xrightarrow{\sigma} M$ eine erreichbare Markierung. Quantitativ lassen sich die Markenänderungen für jede einzelne Stelle $s \in S$ erfassen als $M(s) = M_0(s) + \sum_{t \in {}^\bullet s} \#_t(\sigma) - \sum_{t \in s^\bullet} \#_t(\sigma)$. In Matrixschreibweise ergibt sich $M = M_0 + \mathbf{N} \cdot \vec{\sigma}$, hierin bezeichnet $\mathbf{N}$ die *Inzidenzmatrix* von $\mathcal{N}$, d. h. eine $S \times T$–Matrix mit ganzzahligen Einträgen $\mathbf{N}(s,t) = F(s,t) - F(t,s)$ für alle $s \in S$ und $t \in T$; der *Parikh–Vektor* $\vec{\sigma} = (\#_{t_1}(\sigma), \ldots, \#_{t_{|T|}}(\sigma))^\top$ (Spaltenvektor) beschreibt die Anzahl der Vorkommen der einzelnen Transitionen in der Schaltsequenz σ. Infolge der einfachen Addition und Subtraktion von ein– und ausgehenden Kantengewichten können Schlingen in dieser Darstellung nicht unterschieden werden vom Fehlen einer Verbindung. Dieses Problem besteht in azyklischen Netzen nicht.
Falls die Markierung M von M_0 aus erreichbar ist, dann hat das Gleichungssystem $M = M_0 + \mathbf{N} \cdot X$ mit $X \geq \vec{0}$ ganzzahlig, die sogenannte *Markierungsgleichung*, wenigstens eine Lösung (nämlich gerade $X = \vec{\sigma}$). Die Rückrichtung dieser Aussage gilt nur für den Fall azyklischer Netze [Mur89, Mel98], es handelt sich hier also um ein voll entscheidbares Verfahren.

Satz 7.3 *Test auf Verklemmungsfreiheit mittels linearer Programmierung [MR97, Mel98]*

Sei $\Sigma = (\mathcal{N}, M_0)$ ein sicheres Netzsystem und sei $(\mathcal{O}, \pi) = \mathsf{Pre}(\mathsf{Unf}(\Sigma))$ mit $\mathcal{O} = (B, E, F)$ vollständiges Präfix der Entfaltung von Σ. Das System Σ ist genau dann verklemmungsfrei, wenn das folgende Ungleichungssystem keine Lösung besitzt:

Variablen: M, X: $\{0,1\}$–Vektoren der Dimension $|B|$ bzw. $|E|$.

$$\begin{array}{rcll} M & = & Min(\mathcal{O}) + \mathbf{O} \cdot X & \\ \sum_{b \in {}^\bullet e} M(b) & \leq & |{}^\bullet e| - 1 & \text{für alle } e \in E \\ X(e) & = & 0 & \text{für alle } e \in E_{cutoff} \end{array}$$

■*7.3*

[4] In [Mel98] wird ein linear–algebraischer Ansatz vorgeschlagen, mit dem sich die Erreichbarkeit von (Teil-) Markierungen mit Hilfe des vollständigen Präfixes untersuchen lässt. Das Verfahren führt im Allgemeinen jedoch zu relativ schlechten Laufzeiten aufgrund einer ungünstigen Struktur des Ungleichungssystems, deshalb ist es im letzten Abschnitt nicht näher betrachtet worden.

Die Constraints $\sum_{b \in {}^\bullet e} M(b) \leq |{}^\bullet e| - 1$ begründen sich aus der Tatsache, dass bei einer Verklemmung kein Ereignis aktiviert wird, da die entsprechende Markierung keinen Vorbereich eines Ereignisses vollständig überdeckt.
Für das Beispiel der speisenden Philosophen aus Abb. 7.1 kann nach Satz 7.3 das folgende Ungleichungssystem formuliert werden.

$$
\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
M(b_1) &=& 1 - X(e_1) & M(b_{16}) &=& X(e_7) - X(e_{11}) & M(b_1) &\leq& 0 \\
M(b_2) &=& 1 - X(e_5) - X(e_6) & M(b_{17}) &=& X(e_8) - X(e_{12}) & M(b_3) &\leq& 0 \\
M(b_3) &=& 1 - X(e_2) & M(b_{18}) &=& X(e_9) - X(e_{12}) & M(b_5) &\leq& 0 \\
M(b_4) &=& 1 - X(e_7) - X(e_8) & M(b_{19}) &=& X(e_{10}) - X(e_{13}) & M(b_6) + M(b_7) &\leq& 1 \\
M(b_5) &=& 1 - X(e_3) & M(b_{20}) &=& X(e_{11}) - X(e_{14}) & M(b_2) + M(b_8) &\leq& 1 \\
M(b_6) &=& 1 - X(e_4) - X(e_9) & M(b_{21}) &=& X(e_{12}) - X(e_{15}) & M(b_2) + M(b_9) &\leq& 1 \\
M(b_7) &=& X(e_1) - X(e_4) & M(b_{22}) &=& X(e_{15}) & M(b_4) + M(b_{10}) &\leq& 1 \\
M(b_8) &=& X(e_1) - X(e_5) & M(b_{23}) &=& X(e_{15}) & M(b_4) + M(b_{11}) &\leq& 1 \\
M(b_9) &=& X(e_2) - X(e_6) & M(b_{24}) &=& X(e_{15}) & M(b_6) + M(b_{12}) &\leq& 1 \\
M(b_{10}) &=& X(e_2) - X(e_7) & M(b_{25}) &=& X(e_{14}) & M(b_{13}) + M(b_{14}) &\leq& 1 \\
M(b_{11}) &=& X(e_3) - X(e_8) & M(b_{26}) &=& X(e_{14}) & M(b_{15}) + M(b_{16}) &\leq& 1 \\
M(b_{12}) &=& X(e_3) - X(e_9) & M(b_{27}) &=& X(e_{14}) & M(b_{17}) + M(b_{18}) &\leq& 1 \\
M(b_{13}) &=& X(e_4) - X(e_{10}) & M(b_{28}) &=& X(e_{13}) & M(b_{19}) &\leq& 0 \\
M(b_{14}) &=& X(e_5) - X(e_{10}) & M(b_{29}) &=& X(e_{13}) & M(b_{20}) &\leq& 0 \\
M(b_{15}) &=& X(e_6) - X(e_{11}) & M(b_{30}) &=& X(e_{13}) & M(b_{21}) &\leq& 0 \\
\multicolumn{9}{c}{} \\
X(e_{13}) &=& 0 & X(e_{14}) &=& 0 & X(e_{15}) &=& 0
\end{array}
$$

Die ersten beiden Spalten beschreiben die Markierungsgleichung des Präfixes (je eine Gleichung pro Bedingung), die Ungleichungen rechts codieren die Constraints des Tests auf Verklemmungsfreiheit (je Ereignis eine Ungleichung). In der abgesetzten unteren Zeile stehen schließlich die Constraints für den Ausschluss von Cut-off–Ereignissen. Diese Ungleichungssysteme können effizient mit Werkzeugen der gemischt–ganzzahligen Programmierung ausgewertet werden. Dabei werden die Lösungsvektoren

$$
\begin{aligned}
M &= \{b_7, b_9, b_{11}, b_{14}, b_{16}, b_{18}\} \\
X &= (1,1,1,0,1,0,1,0,1,0,0,0,0,0,0)^\top
\end{aligned}
$$

ermittelt, das System weist also Verklemmungen auf.

Ermittlung von Verklemmungen mit Hilfe von Logik–Programmierung Das im letzten Abschnitt beschriebene Logik–basierte Verfahren zur Erreichbarkeitsanalyse kann auf einfache Weise modifiziert werden, um damit Verklemmungen zu erkennen. Lediglich die Constraints (3)–(5) aus Satz 7.1 müssen hierzu ausgetauscht werden.

Satz 7.4 *Finden möglicher Verklemmungen mittels Logik–Programmierung [Hel98, Hel99b]*

Sei $\Sigma = (\mathcal{N}, M_0)$ ein sicheres Netzsystem und sei $(\mathcal{O}, \pi) = \mathsf{Pre}(\mathsf{Unf}(\Sigma))$ mit $\mathcal{O} = (B, E, F)$ vollständiges Präfix der Entfaltung von Σ. Das System Σ besitzt genau dann

eine erreichbare Verklemmung, wenn das folgende Logik–Programm $\mathcal{P}$ ein stabiles Modell besitzt.

$$\left.\begin{array}{rcll} e_i & \leftarrow & {}^{\bullet\bullet}e_i, \overline{ne_i} & \\ ne_i & \leftarrow & \overline{e_i} & \end{array}\right\} \text{für alle } e_i \in E \setminus E_{cutoff} \quad (1)$$

$$\leftarrow \quad 2(b_i^\bullet \setminus E_{cutoff}) \qquad \text{für alle } b_i \in B \text{ mit } |b_i^\bullet \setminus E_{cutoff}| \geq 2 \quad (2)$$

$$b_i \leftarrow {}^\bullet b_i, \overline{(b_i^\bullet \setminus E_{cutoff})} \qquad \text{für alle } b_i \in B \text{ mit } b_j^\bullet \neq \emptyset \quad (3)$$

$$\leftarrow \quad {}^\bullet e_i \qquad \text{für alle } e_i \in E \quad (4)$$

Für jedes stabile Modell $\Delta_\mathcal{P}$ von $\mathcal{P}$ gibt die Restriktion $C = \Delta_\mathcal{P}|_E \in \mathcal{C}_\mathcal{O}$ eine Konfiguration an, die zur Verklemmung führt, und $\pi(Lin_\prec(C))$ ist die entsprechende Schaltsequenz in Σ. ■*7.4*

Die geänderten Constraints (3) bzw. (4) spezifizieren die Schaltregel bzw. die Erreichbarkeit und damit Lebendigkeit aller Ereignisse.
Bezogen auf das Beispiel aus Abb. 7.1 kann das folgende Logik–Programm aufgestellt werden, wobei zusätzlich die ersten vier Spalten des Beispiels aus Kap. 7.1 auf Seite 131 übernommen werden müssen, die den Klauseln zu (1) und (2) entsprechen.

$b_1 \leftarrow \overline{e_1}$
$b_2 \leftarrow \overline{e_5}, \overline{e_6}$
$b_3 \leftarrow \overline{e_2}$
$b_4 \leftarrow \overline{e_7}, \overline{e_8}$
$b_5 \leftarrow \overline{e_3}$
$b_6 \leftarrow \overline{e_4}, \overline{e_9}$
$b_7 \leftarrow e_1, \overline{e_4}$
$b_8 \leftarrow e_1, \overline{e_5}$
$b_9 \leftarrow e_2, \overline{e_6}$
$b_{10} \leftarrow e_2, \overline{e_7}$

$b_{11} \leftarrow e_3, \overline{e_8}$
$b_{12} \leftarrow e_3, \overline{e_9}$
$b_{13} \leftarrow e_4, \overline{e_{10}}$
$b_{14} \leftarrow e_5, \overline{e_{10}}$
$b_{15} \leftarrow e_6, \overline{e_{11}}$
$b_{16} \leftarrow e_7, \overline{e_{11}}$
$b_{17} \leftarrow e_8, \overline{e_{12}}$
$b_{18} \leftarrow e_9, \overline{e_{12}}$
$b_{19} \leftarrow e_{10}$
$b_{20} \leftarrow e_{11}$

$b_{21} \leftarrow e_{12}$
$\leftarrow b_1$
$\leftarrow b_3$
$\leftarrow b_5$
$\leftarrow b_6, b_7$
$\leftarrow b_2, b_8$
$\leftarrow b_2, b_9$
$\leftarrow b_4, b_{10}$
$\leftarrow b_4, b_{11}$
$\leftarrow b_6, b_{12}$

$\leftarrow b_{13}, b_{14}$
$\leftarrow b_{15}, b_{16}$
$\leftarrow b_{17}, b_{18}$
$\leftarrow b_{19}$
$\leftarrow b_{20}$
$\leftarrow b_{21}$

Ein stabiles Modell dieses Programms ist $\{e_1, e_2, e_3, ne_4, e_5, ne_6, e_7, ne_8, e_9, ne_{10}, ne_{11}, ne_{12}, b_7, s_2, b_9, s_9, b_{11}, s_{16}, b_{14}, s_5, b_{16}, s_{12}, b_{18}, s_{19}\}$. Die damit errechnete Konfiguration $\{e_1, e_2, e_3, e_5, e_7, e_9\}$ korrespondiert mit der Schaltfolge $t_1 t_6 t_{11} t_3 t_8 t_{13}$, die zu einer verklemmenden Markierung führt.

7.3 Model–Checking

Während die in den letzten beiden Abschnitten behandelten Verifikationsmethoden bestimmte, fest definierte Eigenschaften auf effiziente Weise überprüft haben, soll nun eine flexiblere, universellere Methode diskutiert werden: das Model–Checking bezeichnet eine automatisierte Technik, Korrektheitseigenschaften von Systemen nachzuweisen; von der Natur her sind dies meist sicherheitskritische, reaktive Systeme.

Eigenschaften können innerhalb von Model–Checkern mit Hilfe von Logiksprachen als (temporal–logische) Formeln spezifiziert werden. Ein deduktiver Kalkül oder ein Algorithmus nimmt dann die Verifikation dieser Eigenschaften auf dem modellierten System vor. Wie solche Logiken konkret aussehen, wird später in diesem Kapitel erörtert.
Vollständig automatisierte Model–Checking–Verfahren terminieren mit einer Ausgabe, ob die überprüfte Eigenschaft gilt oder nicht[5]. Im Falle des Nicht–Erfüllens kann oft eine Ausführungsfolge als Gegenbeispiel ausgegeben werden. Diese ist nützlich für die Fehlersuche und –korrektur. Neben den vollautomatischen gibt es auch halbautomatische Verfahren, die mit Antworten der Art Ja / Nein, oder aber „weiß nicht“ terminieren, wobei letzteres Resultat keine genaue Aussage zulässt und dann manuell durch Spezifikation anderer Eigenschaften weitere Untersuchungen erfordert. Halbautomatische Verfahren sollen hier jedoch nicht betrachtet werden.
In manchen Fällen kann es vorteilhafter sein, eine negative Aussage bezüglich der Gültigkeit von Eigenschaften zu erhalten. Das Gegenbeispiel kann dann nämlich Aufschluss darüber geben, ob ein Modellierungs– oder Formulierungsfehler vorliegt, oder gar, ob die angegebene Ausführungsfolge tatsächlich im realen System möglich ist. Im ersteren Fall können die Fehler leicht beseitigt werden durch Änderungen an Modell oder Formel, im letzteren Fall liegt ein Systemfehler vor, der entsprechen korrigiert werden muss. Ein großer Vorteil der automatischen Verifikation besteht darin, dass kein oder kaum zusätzlicher Aufwand vonnöten ist für die erneute Untersuchung des veränderten Modells.

Die ersten Model–Checker arbeiteten noch auf dem vollständigen Erreichbarkeitsgraphen, z. B. [CES86]; wegen des Problems der Zustandsraum–Explosion konnten somit nur relativ kleine Systeme getestet werden. Mit dem Aufkommen geschickter Codierungen des Zustandsraums, wie etwa BDDs oder dem in dieser Arbeit behandelten Präfix der Netzentfaltung, sind auch effiziente (symbolische) Model–Checking–Verfahren auf diesen Codierungen entwickelt worden. Zwei dieser implementierten Verfahren auf Präfixen werden im Anschluss vorgestellt, wobei für ausführliche Details auf das Studium der angegebenen Literatur verwiesen wird. Zunächst wird kurz aufgezeigt, worin sich die beiden Verfahren unterscheiden: in der Eingabesprache für die logischen Formeln.

Temporale Logiken In Kapitel 2.4 sind im Rahmen der halbgeordneten Darstellung des dynamischen Verhaltens von verteilten Systemen zwei Beschreibungsformen vorgestellt worden: kausale Prozesse dienen der Modellierung einzelner Ausführungen, andererseits erlauben verzweigende Prozesse die Modellierung mehrerer Abläufe in

[5] Voraussetzung für die Terminierung ist selbstverständlich ein endlicher Zustandsraum des Systems. Weiterhin können Platz– oder Zeitbeschränkungen eine erfolgreiche Untersuchung unmöglich machen.

einem Objekt. Analog zu dieser Einteilung lassen sich auch temporale Logiken klassifizieren: *Branching–Time–Logiken* interpretieren Eigenschaften auf *Zeitbäumen*, einzelne Zeitinstanzen können hierbei mehrere Nachfolger besitzen.
Auf der anderen Seite findet bei *Linear–Time–Logiken* die Untersuchung bezüglich einzelner *Zeitlinien* (auch Pfade in der Auffaltung des Erreichbarkeitsgraphen des Systems) statt, jede Zeitinstanz kann maximal einen Nachfolger haben, es wird also über konkreten einzelnen (möglicherweise unendlichen) Abläufen argumentiert. Zur Berücksichtigung von Nichtdeterminismen ist es hier notwendig, alle möglichen Zeitlinien in Betracht zu ziehen; bei den Branching–Time–Logiken hingegen genügt ein einziger Zeitbaum (der Ableitungsbaum), um die gesamte mögliche Zukunft darzustellen.
Allgemein eignen sich Branching–Time–Logiken zur Formulierung von Sicherheitseigenschaften. Mit Linear–Time–Logiken lassen sich neben Sicherheits– auch Lebendigkeitsaussagen formulieren, außerdem erlaubt die Zeitlinien–behaftete Argumentation ein Treffen von Fairness–Annahmen.

Ein Halbordnungs–basierter Model–Checker für eine einfache Branching–Time–Logik
Eine der ersten praktischen Anwendungen, die nach der Veröffentlichung des Algorithmus von McMillan auf dem Präfix der Netzentfaltung entwickelt worden ist, ist der Model–Checker aus [Esp93, Esp94, Gra95, Gra97]. Dieser akzeptiert eine einfache Branching–Time–Logik, S_4 [HC68], die eine Untermenge von CTL darstellt: Neben den Booleschen Konstanten und logischen Operatoren wie $\wedge$ und $\neg$ sowie daraus abgeleiteten Operatoren ($\vee, \Rightarrow, \Leftrightarrow$) unterstützt diese Logik als temporalen Operator lediglich $\diamond$ (entspricht **EF** in CTL; Bedeutung: „irgendwann“) sowie das daraus ableitbare $\Box$ (**AG** in CTL, hat die Bedeutung von „immer“).
Die Natur der Logik S_4 (wie auch CTL) erlaubt eine direkte Überprüfung der spezifizierten Eigenschaften auf dem Erreichbarkeitsgraphen mittels Traversierung. Dieser Model–Checker verwendet das vollständige Präfix, in dem die erreichbaren Markierungen nur einer codierten Form vorliegen. Eine spezielle *Shift*–Operation ermöglicht das Verlängern von Konfigurationen über das Präfix hinaus. Zur Identifizierung besionderer Zustände werden Methoden der Linearen Programmierung eingesetzt.

Ein Halbordnungs–basierter Model–Checker für eine Linear–Time–Logik Einen LTL–Model–Checker auf dem Präfix der Netzentfaltung beschreibt Wallner in [Wal98, Wal00]. Die Syntax von LTL [Pnu77] erlaubt Formeln, die neben Booleschen Konstanten und den üblichen logischen Operatoren die temporalen Operatoren **G** (*generally*), **F** (*finally*) und **U** (*until*) enthalten. Lediglich der *nexttime*–Operator **X** wird vom Model–Checker nicht unterstützt.
Dieser LTL–Model–Checker basiert auf dem automatentheoretischen Ansatz [VW86]: Ein beschrifteter Büchi–Automat [Büc62] wird konstruiert, der die von der zu überprüfenden LTL–Formel akzeptierten Abläufe akzeptiert. Dieser Automat wird mit dem

Präfix synchronisiert; der entstehende, sogenannte Produktautomat hat die Eigenschaft, dass er die leere Sprache genau dann akzeptiert, wenn das Ausgangssystem ein Modell der negierten LTL–Formel ist. Das Leerheits–Problem für einen Automaten kann mittels graphentheoretischer Algorithmen effizient beantwortet werden.
Die Konstruktion des Produktautomaten ist nicht trivial, da die unendlichen Abläufe des Systems auf komplizierte Weise im Präfix codiert sind. Um sie zu ermitteln, muss ein bestimmter Graph konstruiert werden, der im schlimmsten Fall exponentiell größer sein kann als das Präfix.

Kapitel 8

„Programmierer sind immer von Komplexitäten umgeben; wir können dies nicht vermeiden. Wenn unser grundlegendes Werkzeug, die Sprache, in der wir unsere Programme entwickeln und codieren, auch kompliziert ist, so wird die Sprache selbst Teil des Problems und nicht der Problemlösung.“
C. A. R. HOARE

Implementationsaspekte

In den vorangegangenen Kapiteln ist die Theorie der Netzentfaltungen formal hergeleitet und umrissen worden, zusammen mit möglichen Anwendungen. Die restlichen Kapitel dieser Arbeit widmen sich nunmehr den praktischen Aspekten und Ergebnissen im Rahmen konkreter Implementierungen auf Rechnern. Die vorgestellten Algorithmen und Verfahren stehen in diversen Analyse–Werkzeugen zur Verfügung, das nächste Kapitel stellt eine Übersicht dieser Werkzeuge vor.

Aus praktischen Überlegungen beschränkt sich im Folgenden die Darstellung der Entfaltungstechnik auf die sicherer Netzsysteme, auch wenn Implementierungen für beschränkte Netzsysteme existieren. Historisch gesehen bedingte der Model–Checker aus [Esp93, Esp94, Gra95, Gra97] eine erste konkrete, vor allem effiziente Implementierung des McMillan–Algorithmus im Rahmen der Arbeiten am Werkzeug PEP [BF95]. Für diesen Model–Checker reichen sichere Systeme aus, zudem fußt der gesamte Box–Kalkül auf dieser Netzklasse. Der größte Teil der bislang veröffentlichten Arbeiten zu Anwendungen von Netzentfaltungen bezieht sich auf sichere Systeme.

McMillan nahm für die Präsentation seiner Ergebnisse eine prototypische Implementierung des Algorithmus in der (interpretierten) Sprache LISP vor [McM92]. Als Anwendungsbeispiel führte er u. a. das Modell eines *Distributed Mutual Exclusion*–Schaltkreises (kurz DME) an: für die Präfix–Generierung eines DME der Zellengröße 9 (das entsprechende Netzsystem besteht aus 604 Stellen und 490 Transitionen) ermittelte er eine Rechenzeit von 19000 Sekunden ermittelt[1], also etwas mehr als fünf Stunden; das Präfix

[1] Leider sind die konkrete Rechnerplattform sowie der verwendete LISP–Interpreter nicht bekannt. Die angegebenen Zeiten sind somit nicht wirklich miteinander vergleichbar, sondern sollen nur eine ungefähre Größenvorstellung liefern.

(mit der Ordnung $\sqsubset_{McM}$ konstruiert) besteht aus 18316 Bedingungen und 5337 Ereignissen, davon 81 Cut-off.
Eine erste PEP–Implementierung des McMillan–Algorithmus, vorgenommen von zwei Studenten der Universität Hildesheim im Jahre 1994 in der Sprache *C*, reichte nicht an diese Zeiten heran: die Präfix–Generierung eines in B(PN)[2] modellierten Mutex–Algorithmus nach Peterson (bestehend aus 37 Stellen und 101 Transitionen; das Präfix ist 429 Bedingungen und 212 Ereignisse groß) dauerte länger als einen Tag. Diese Implementierung orientierte sich sehr stark an Listen–orientierten Datenstrukturen (wie auch schon in McMillans Originalarbeit), jedoch.
Für einen Einsatz in einem Verifikationswerkzeug waren derart hohe Rechenzeiten nicht akzeptabel, deshalb hat der Autor eine komplett neue Implementation vorgenommen, aus Gründen der Effizienz ebenfalls in Standard–*C* [KR78]. Für die Darstellung von Petri–Netzen ist eine spezielle Datenstruktur entwickelt worden, zusammen mit einer Bibliothek grundlegender Operationen zur Manipulation dieser Netze. Nach und nach wurden – und werden immer noch – zahlreiche Verbesserungen an Algorithmus und Implementierung vorgenommen, von denen die wichigsten in den Abschnitten dieses Kapitels vorgestellt werden. Mit dieser Version konnte das Präfix für das oben genannte Mutex–System in Bruchteilen von Sekunden (auf der gleichen Maschine) berechnet werden. Die Präfix–Generierung für das DME–9–System dauerte 132 Sekunden [MR97] auf einer SPARC 20/712.
Die Hervorhebung der Rechenzeit ist keineswegs Spielerei. Im Rahmen der automatischen Verifikation sind effiziente Werkzeuge unabdingbar. Im Verbund mit dem heutzutage üblichen (und voraussehbar weiter steigenden) Hauptspeicherausbau und Prozessortakt erlauben sie die Analyse immer größerer und damit praktisch relevanter Systeme in vernünftiger Zeit.
Für BDD–basierte Verifikationsalgorithmen gibt es inzwischen mehrere Bibliotheken unterschiedlicher Effizienz; teilweise besitzen diese sogar identische Funktionsprototypen und sind somit direkt austauschbar.

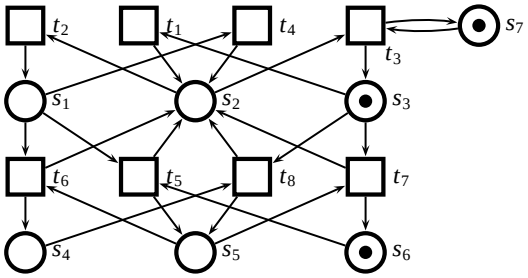
Die Spezialisierung auf die Entfaltung 1–sicherer Systeme erlaubt eine besondere Optimierung der zeitkritischsten Komponente der Präfix–Generierung, nämlich der Ermittlung möglicher erweiternder Ereignisse[2].
An zahlreichen Stellen dieser Implementierung wird das Vorhandensein sicherer Netze implizit verwendet, bis hinein in die Datenstrukturen, die lediglich einfache Kantengewichte und Markenbelegungen (maximal eine Marke pro Stellenknoten) voraussetzen. Damit sind Kanten und Marken als einfache Boolesche Wahrheitswerte zu behandeln.
Ausgehend von der Beschreibung der verwendeten Datenstrukturen für die Verwaltung von Petri–Netzen beleuchtet das Kapitel die zentralen Funktionen der Präfix–

[2] Komplexitätstheoretisch betrachtet ist das Problem POSSIBLE EXTENSIONS in der Größe des *n*–beschränkten Netzsystems und des Präfixes NP–vollständig; den formalen Beweis hierfür führt K. Heljanko durch Reduktion auf das Problem 3SAT in einem derzeit noch unveröffentlichten Manuskript.

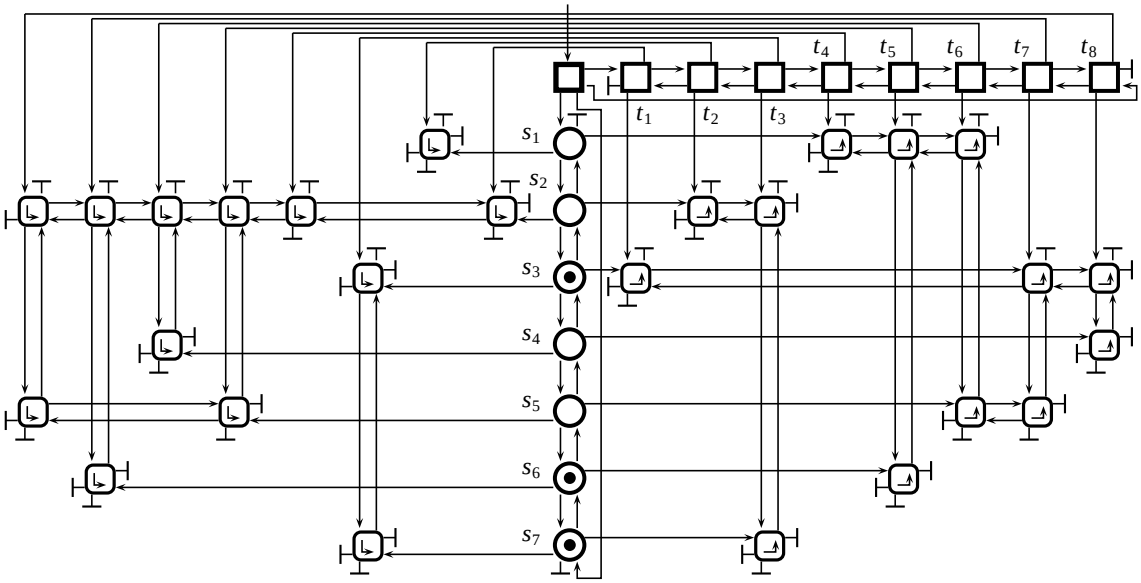
Generierung. Hierzu gehören beispielsweise die Ermittlung der co–Relation, die Bestimmung von Cut-off–Ereignissen oder die Berechnung möglicher erweiternder Ereignisse.
Im Anschluss daran werden einige interessante Aspekte der Umsetzung des DB(PN)^2–Übersetzers erläutert. Als eine konkrete Anwendung der Entfaltungstechnik wird abschließend die Implementierung eines einfachen Algorithmus zur Erkennung von Verklemmungen erläutert.

8.1 Eine universelle Datenstruktur für Petri–Netze

Für die Speicherung und Manipulation von Petri–Netzen ist eine universelle Datenstruktur entwickelt worden, die einen schnellen Zugriff auf die Knoten erlaubt [Röm93]. Damit ist die eine effiziente Umsetzung vieler Algorithmen auf Netzen möglich, nicht nur der hier beschriebenen.
Die Grundidee dieser Datenstruktur ist recht einfach, sie orientiert sich an der dem Netz zugrunde liegenden Inzidenzmatrix, um den Netzgraph zu verwalten. Da im Allgemeinen die Knoten eines Netzes keinen allzu hohen Verzweigungsgrad aufweisen, ist die Inzidenzmatrix nur schwach besetzt. Deshalb werden nur die tatsächlich Kanten im Netz repräsentierenden Einträge gespeichert. Abbildung 8.1(a) zeigt hierzu ein Beispiel: ein Netzsystem mit Knoten unterschiedlichen Verzweigungsgrads. In Abb. 8.1(b) ist die interne Repräsentation dieses Systems in der verwendeten Datenstruktur graphisch dargestellt, eine verfeinerte Form von Adjazenzlisten [Sed88].
Die Stellenknoten sind hierbei in einer doppelt verketteten Liste angeordnet; sie ist vertikal angeordnet in der Mitte des Bildes zu sehen. Die doppelte Verkettung erlaubt ein einfaches Einfügen und Löschen einzelner Knoten; diese Technik findet auch bei den anderen Listen Verwendung.
Zu jedem Stellenknoten werden dessen lokale Informationen gespeichert, also Name der Stelle, aktuelle Markenzahl, Beschriftungen, Nummer der Komponente (bei SND–Systemen) etc. Je nach verwendeter Netzklasse können hier, wie auch in den später beschriebenen Knoten, zusätzliche Daten abgelegt werden.
Die Transitionsknoten sind in einer eigenen, doppelt verketteten Liste vereinigt; diese ist horizontal angeordnet im Bild oben rechts dargestellt. Ebenfalls werden die eine einzelne Transition betreffenden Informationen lokal im entsprechenden Knoten gespeichert.
Der Zugriff auf die Stellen– und Transitionenlisten erfolgt über einen zentralen Kopfknoten, der Zeiger auf die jeweils ersten und letzen Elemente der beiden Listen enthält. Über diesen Kopfknoten ist ein einzelnes Netz im Speicher referenzierbar.
Die Kanten von Stellen nach Kanten (im Folgenden kurz S–T–Kanten) bzw. von Transitionen nach Stellen (T–S–Kanten) werden in zwei getrennten Listen verwaltet. Einerseits wird infolge der kreuzweise gekoppelten Listen – im Unterschied zu einfachen



(a) Beispielsystem.



(b) Interne Netzsystem–Darstellung mit doppelt verketteten Listen.

Abbildung 8.1: Interne Repräsentation von Netzen.

Adjazenz–Listen – jeder Knoten nur ein einziges Mal gespeichert. Andererseits erlaubt diese Trennung einen unmittelbaren Zugriff auf den Vor– bzw. Nachbereich einer Stelle oder Transition. Würden die Listenstrukturen links und rechts der Stellenknoten übereinandergelegt in einer einzigen Struktur verwaltet, müssten bei jedem Betrachten eines Vor– oder Nachbereichs die nicht passenden Knoten von neuem durch einen Test gefiltert werden.
In Abb. 8.1(b) ist der Block der T–S-Kanten bezüglich der Transitionen gespiegelt gegenüber der S–T–Kantenblock dargestellt.
Zu jedem Knoten werden wiederum dessen lokale Daten gespeichert, dazu gehört, sofern vorhanden, z. B. das Kantengewicht. Darüber hinaus werden die Referenzen auf den jeweiligen Stellen– und Transitionsknoten gespeichert. Die zusätzlichen Zeiger sind in den Kantenknoten der Abbildung durch abgewinkelte Pfeile dargestellt, die die Richtung der Verbindung anzeigen. Dieser Kniff erlaubt ein effizientes Abschreiten von Pfaden durch ein Netz, sowohl entlang der Vorbereiche wie auch der Nachbereiche.
Bei der Darstellung sicherer Netze wird auf ein Speichern der Kantengewichte verzichtet; die Existenz eines Kantenknotens impliziert das Gewicht eins.

Ein Nachteil der Aufteilung der Kantenmatrix ist das erschwerte Erkennen von Schlingen. Für die in dieser Arbeit betrachteten Netze und Probleme ist diese Einschränkung vernachlässigbar; Occurrence–Netze sind definitionsgemäß azyklisch, damit ist die Gesamtzahl der Kantenknoten eines Prozess–Netzes in beiden Darstellungen identisch. Bei der Umsetzung von Algorithmen wie [ES92, SE00] kann dieser Punkt jedoch Bedeutung erlangen.
Der Speicherverbrauch der geteilten Listenblöcke ist nur unwesentlich höher als bei Verwaltung in einer einzelnen Struktur (solange die Zahl der Schleifen gering ist): ein zusätzlicher Zeiger in jedem Stellen– und Transitionsknoten, außerdem je ein kompletter Kantenknoten pro Schlinge.

Basierend auf dieser Datenstruktur steht eine Bibliothek mit häufig benötigten, universellen Operationen zur Verfügung. Zu den wichtigsten Funktionen gehören:

- Einfügen neuer Knoten und Kanten;
- Auffinden bestimmter Knoten und Kanten;
- Löschen einzelner Knoten und Kanten bzw. ganzen Netzen;
- Operationen auf beschrifteten Netzen aus Kapitel 4, darunter Duplizieren von Stellen oder Transitionen, Vereinigung von Netzen, Addition und Multiplikation von Knoten;
- Ein-/Ausgabe–Routinen zum Lesen und Schreiben von Netzen in diversen Formaten.

Einige Grundoperationen wie das Besuchen der unmittelbaren Nachbarschaft eines Knotens können unmittelbar über Zeigerreferenzen ausgedrückt werden.
Die Zeit–Komplexitäten der Grundoperationen ergeben sich unmittelbar aus der Datenstruktur. Beispielsweise benötigt das Löschen eines bestimmten Netzknotens x die Zeit $O(|{}^\bullet x| + |x^\bullet|)$, das Besuchen des Nachbereichs eines Knotens y ist in $O(|y^\bullet|)$ durchführbar. Der Test auf Leerheit eines Vorbereiches für eine Transition t beträgt $O(1)$.
Ein konkretes Beispiel aus der Implementation soll einen Eindruck vermitteln, in welcher Form Operationen auf dieser Datenstruktur umgesetzt werden können. Im Zusammenhang mit der Präfix–Generierung ist eine häufig benutzte Funktion das Besuchen der Ereignisknoten einer lokalen Konfiguration, z. B. im Rahmen der Berechnung der Funktionen Cut und Mark.
Eine lokale Konfiguration kann über eine Tiefensuche (z. B. [Sed88]) abgeschritten werden. Auf Netzebene übertragen kann ein solcher Algorithmus in Pseudocode–Notation folgendermaßen beschrieben werden.

Algorithmus 8.1 *Besuchen einer lokalen Konfiguration*

Eingabe: Occurrence–Netz $\mathcal{B} = (B, E, F)$; Ereignis $e \in E$.

```
1   procedure visit_config( e )
2     IncVisitCounter();
3     config_recur( e )
4   endproc
5
6   procedure config_recur( e )
7     MarkVisited( e );
8     /*  Aktionen für Ereignis e  */
9     for all b ∈ •e do
10      if •b ≠ ∅ then
11        {e'} := •b;
12        if ¬Visited(e') then
13          config_recur( e' )
14        endif
15      endif
16    endfor
17  endproc                                          ■8.1
```

Die Funktion IncVisitCounter bedarf einer kurzen Erläuterung. In den meisten Beschreibungen von Algorithmen zur Tiefensuche wird eine Initialisierungsphase vorangestellt, die alle „visited“–Einträge in den einzelnen Knoten zurücksetzt. Der dafür benötigte Aufwand ist linear in der Größe des Graphen, für Netzentfaltungen also die aktuelle Präfixgröße, $O(|E|)$.

Diese Initialisierung wird hier durch Verwenden einer globalen Variablen *visited* auf $O(1)$ verkürzt. Die „visited“–Einträge der Knoten sind nunmehr nicht mehr binären Datentyps, sondern numerisch, um den aktuellen Wert von *visited* aufzunehmen, nachdem ein Knoten besucht worden ist.
Der nachstehende Programmausschnitt zeigt eine Implementation dieses rekursiven Algorithmus in *C*, aufsetzend auf den vorher beschriebenen Datenstrukturen.

Programmtext 8.2 *Besuchen einer lokalen Konfiguration, C–Umsetzung von Algorithmus 8.1*

```
int visited = 0; /* globale Variable */

void config_recur( Event e ) {
    STEdge ce;
    TSEdge ce;

    e->visited = visited;

    /* hier Aktionen für Ereignis e */

    for ( ce = e->preset; ce; ce = ce->nextplace )
        if ( ( ts = ce->place->preset ) && ts->trans->visited != visited )
            config_recur( ts->trans );
}

void visit_config( Event e ) {
    ++visited;
    config_recur( e );
}
```
■8.2

Dieses Beispiel veranschaulicht, auf welch einfache Weise die vorgestellte Datenstruktur die Implementierung von Operationen auf Netzen unterstützt. Auch ohne spezielle Kenntnisse der Programmiersprache *C* sollte es möglich sein, die einzelnen Anweisungen der Pseudocode–Notation im Programm wiederzufinden. Es werden keine zusätzlichen Funktionen aufgerufen, lediglich *C*–Befehle auf unterster Ebene werden verwendet.

8.2 Präfix–Generierung für sichere Netzsysteme

Der im theoretischen Teil vorgestellte Algorithmus 3.12 zur Berechnung eines vollständigen Präfixes lässt sich leicht als korrekt beweisen, in dieser Form ist er jedoch nicht besonders effizient. Beispielsweise wird nach jedem Einfügen eines neuen Ereignisses die Menge *erw* der möglichen Erweiterungen komplett neu berechnet. Spezielle Probleme, wie etwa das Finden erweiternder Ereignisse, werden darin überhaupt nicht näher ausgeführt. In diesem Abschnitt werden daher ausgewählte Aspekte einer konkreten Implementation dieses Algorithmus diskutiert, wobei das Ausgangs–Netzsystem als

1–sicher vorausgesetzt wird. Die hier beschriebene Implementation wird inzwischen in verschiedenen Netz–basierten Simulations– und Analyse–Werkzeugen eingesetzt, darunter PEP [PEP98] CPN–AMI [CPN99] und Kit [SSE99].

8.2.1 Wahl der Datenstrukturen

Für die Darstellung des Netzsystems wie auch des Präfixes (Occurrence–Netz) wird die zuvor beschriebene Datenstruktur für Petri–Netze verwendet. Das Netzsystem wird aus einer externen Datei, in der das Netz in einer bestimmten textuellen Beschreibungssprache abgelegt ist, eingelesen und dabei im Speicher aufgebaut. Daran anschließend wird das Präfix der Entfaltung berechnet.
Diese beiden Netzaufbauten sind rein konstruktiver Natur, es werden sukzessive Knoten erzeugt und in das Netz eingepasst, jedoch keine Knoten gelöscht.

Aus Algorithmus 3.12 auf Seite 39 ist der Test in Zeile 7 (**if** $[e] \cap \mathit{cut\text{-}off} = \emptyset$) in der Implementation nicht vorhanden. Als mögliche Erweiterungen kommen nur solche Ereignisse in Betracht, die nicht hinter Cut-off–Ereignissen liegen; das ist auf einfache Weise erreichbar, indem hinter Cut-off–Ereignissen zunächst keine Bedingungen eingefügt werden, sondern erst dann, wenn das vollständige Präfix erstellt worden ist. Im Programm ist über eine Option zusätzlich einschaltbar, ob überhaupt Bedingungen hinter Cut-off–Ereignissen erzeugt werden sollen. Aus theoretischer Sicht garantiert das Vorhandensein dieser Bedingungsknoten lediglich, dass es sich bei dem Netz tatsächlich um einen verzweigenden Prozess handelt (wegen des Netzhomomomorphismus π, Def. 2.19). Für die Vollständigkeit des Präfixes liefern sie keinen Beitrag, da die von einem Cut-off–Ereignis induzierte Markierung bereits anderswo im Präfix gefunden werden kann. Daher sind sie entbehrlich, bei einer hohen Zahl von Cut-off–Ereignissen kann die Netzgröße und damit der Speicherbedarf auf diese Weise merklich reduziert werden.
Als Folge dieser Beobachtung brauchen neue Knoten lediglich am Ende der Listen eingefügt zu werden, die Netz–Datenstrukturen können also noch etwas vereinfacht werden durch Verwendung einfach verketteter Listen (mit Zeiger auf das Listenende).

Während der Konstruktion des Präfixes erfolgt der Zugriff auf einzelne Netzknoten stets symbolisch über deren Speicheradressen. Zwar werden zu jedem Knoten auch die vom Benutzer vergebenen bzw. die von Modellierungswerkzeugen automatisch erzeugten Namen verwaltet; intern werden sie jedoch stets symbolisch referenziert. Bereits bei der Implementation einfacher netzbasierter Analysealgorithmen hat sich gezeigt, dass immer wieder durchzuführende Vergleiche von Zeichenketten eine signifikante Erhöhung der Rechenzeit nach sich ziehen.

Die Ordnung $\ll$ auf den Transitionen (und auch Stellen) des Netzsystems ergibt sich aus der Position der Knoten in den doppelt verketteten Listen. Diese Reihenfolge wird beim Einlesen des Netzes aus der externen Datei übernommen.
Die Darstellung der Abbildung π erfolgt durch einfache Verweise in den Bedingungs– bzw. Ereignisknoten des Prozess–Netzes auf die entsprechenden Stellen bzw. Transitionen des Netzsystems.
Die Umkehrabbildung π^{-1} wird benötigt für die kombinatorische Ermittlung der möglichen Erweiterungen des Präfixes, dazu müssen co–Mengen von Bedingungen berechnet werden. Die Abbildung π^{-1} wird nur für Stellen gebraucht. Da diese Umkehrabbildung nicht eindeutig ist, wird sie in Form linearer Listen an die einzelnen Stellenknoten geheftet, deren Elemente aus Verweisen auf die entsprechenden Bedingungsknoten bestehen. Mit jedem Einfügen einer neuen Bedingung in das Präfix werden diese Listen aktualisiert, sie sind absteigend nach ihrer identifizierenden Nummer, also dem Erzeugungsalter sortiert; die Kosten für das Einfügen sind somit $O(1)$.
Für die Entscheidungsfindung, ob ein Ereignis e die Cut-off-Eigenschaft besitzt, ist ein Vergleich der von $[e]$ induzierten Markierung $\mathsf{Mark}([e])$ mit den induzierten Markierungen der bereits im Präfix vorhandenen lokalen Konfigurationen notwendig. Zu diesem Zweck werden alle induzierten Markierungen gespeichert (maximal sind dies $|E \setminus E_{cutoff}|$ viele), und zwar jeweils als ein Feld von Referenzen auf Stellenknoten, siehe auch Abschnitt 8.2.6. Wie schon in [McM92] angeregt, werden diese Markierungen in einer Hash–Tabelle [Sed88] abgelegt, die Hash–Funktion bedient sich dabei der numerischen Referenzen der Stellenknoten.

8.2.2 Verwaltung erweiternder Ereignisse

Damit die Menge der erweiternden Ereignisse (Variable *erw* in Algorithmus 3.12) nicht in jedem Schleifendurchlauf komplett neu berechnet werden muss, wird sie in Form einer *Warteschlange* [Sed88] verwaltet. Diese Struktur ist bereits von McMillan in [McM92] vorgeschlagen worden.
Die einzelnen Ereignisse werden darin entsprechend der verwendeten Ordnung $\sqsubset$ aufgereiht, wobei das Kopfelement ein bezüglich der Ordnung minimales Element darstellt. Die in der Schlange enthaltenen Ereignisse sind bereits in das Präfix eingefügt worden, jedoch noch ohne Nachbereiche – zu diesem Zeitpunkt ist noch nicht bekannt, ob es sich jeweils um Cut-off–Ereignisse handelt oder nicht.
Zu jedem Ereignis wird die Größe seiner lokalen Konfiguration als Zahlenwert gespeichert, um die erforderlichen Vergleiche bezüglich der adäquaten Ordnung zu beschleunigen. Die Konfigurationen selbst werden – aus Platzgründen – nicht explizit verwaltet, mit dem Nachteil, dass bei den aufwändigeren Vergleichen die Konfigurationen komplett neu ermittelt werden müssen (über eine Funktion der Art von Algorithmus 8.1). Dieses Vorgehen ermöglicht einen sinnvollen Mittelweg zwischen Platz– und Zeitverbrauch.

Für die totalen Ordnungen $\sqsubset_t$ bzw. $\sqsubset_{SND_k}$ werden für die Vergleiche weitere Informationen zu den in der Warteschlange befindlichen Ereignisknoten e gespeichert, nämlich:

- bei der Ordnung $\sqsubset_t$ die Sequenz $\mathsf{Lin}_{\ll}(\pi([e]))$.

  In der Literatur wird diese Darstellung auch *Parikh–Vektor* genannt. Da dieser Vektor im Allgemeinen eine große Zahl Null–Einträge besitzt, wird die Sequenz in Form eines Feldes verwaltet, das lediglich die Nicht–Nulleinträge aufnimmt.
  Die einzelnen Feldeinträge bestehen aus zwei Komponenten: zum einen eine Referenz auf den Transitionsknoten, zum anderen die Anzahl der Vorkommen der jeweiligen Transition in der Sequenz. Innerhalb des Feldes sind die Einträge gemäß der Ordnung $\ll$ auf den Transitionen sortiert, die Dimension des Feldes ergibt sich aus der Zahl verschiedener Transitionen in der Sequenz. Für verschiedene Ereignisse müssen diese Felder also nicht die gleiche Dimension besitzen. Ein Vergleich zweier Sequenzen bzw. Felder ist in linearer Zeit in der Größe der Felder durchführbar, wobei der Vergleich bereits beim ersten Index mit ungleichen Einträgen abgebrochen werden kann.

- bei den Ordnungen $\sqsubset_{SND_k}$ ein numerisches n–Tupel $(|V_1([e])|, \ldots, |V_1([e])|)$, das die Größen der lokalen Sichten enthält.
  Diese n–Tupel werden ebenfalls als Felder gespeichert, jedoch mit fester Dimension n und lediglich numerischen Einträgen. Der Vergleich zweier solcher Tupel erfolgt auf die gleiche Weise wie oben und damit in linearer Zeit $O(n)$.
  Bei den SND–Ordnungen gestaltet sich die notwendige Tiefensuche einfacher, da sie entlang einer konkreten Komponente führt. Die Information über die Komponentenzugehörigkeit ist in den Stellenknoten in numerischer Form abgelegt.

Durch diese Vorab–Speicherung kann der überwiegende Teil der Vergleiche auf effiziente Weise ausgeführt werden. In Kapitel 10 werden die Anteile teurer Vergleiche bei der Präfix–Generierung für einige Beispiele zahlenmäßig erfasst. Dort sind auch quantitative Aussagen über die maximale Länge der Warteschlange zu finden.
In Versuchen hat sich das Einreihen neuer Ereignisse vom Ende der Schlange her für die meisten Netzsysteme als die vorteilhafteste Methode erwiesen; diese Art des Einfügens wird durch die schichtweise Konstruktion des Präfixes auch nahegelegt. Testreihen mit anderen Arten des Einfügens, beispielsweise vom Kopf her oder über ein binäres Suchen, waren zwar in Einzelfällen etwas schneller, führten aber sonst zu wesentlich ungünstigeren Zeiten als das Einsortieren vom Ende.
Eine Laufzeitverbesserung bei der Verwaltung der erweiternden Ereignisse ist aber vermutlich noch erzielbar. Strukturen, die hier Vorteile versprechen, sind der Einsatz einer Prioritätswarteschlange [Sed88] oder das nur grobe Einsortieren in Cluster, in denen Ereignisse mit gleicher Größe der lokalen Konfigurationen vereint werden. Diese Änderungen sind jedoch noch nicht implementiert, so dass eine abschließende Aussage noch nicht getroffen werden kann.

8.2.3 Hauptprogramm der Präfix–Generierung

Nach den verwendeten Datenstrukturen werden in den kommenden Abschnitten die zentralen Module der Präfix–Generierung vorgestellt. Der folgende Algorithmus liegt der Implementierung zugrunde, er beschreibt das Verfahren aus Alg. 3.12 auf etwas konstruktivere und effizientere Weise.

Algorithmus 8.3 *Erzeugen eines vollständigen Präfixes eines sicheren Netzsystems*

Eingabe: Endliches, sicheres Netzsystem $\Sigma = (S, T, F, M_0)$.
Ausgabe: Vollständiges Präfix $\mathsf{Pre}(\mathsf{Unf}(\Sigma)) = ((B, E, F), \pi)$ der Entfaltung.

```
1   procedure CalcPrefix()
2     /* Initialisierung der globalen Variablen: */
3     x := 0; y := 0; /* Zähler für Bedingungs- bzw. Ereignisknoten. */
4     B := ∅; E := ∅; F' := ∅;
5     cutoffs := ∅; queue := ϵ;
6     /* Übertragen der Anfangsmarkierung M0 in das Präfix: */
7     for all s ∈ M0 do
8       x := x + 1; B := B ∪ { b^x }; π(b^x) := s;
9       CalcPossibleExtensions(b^x) /* siehe Kap. 8.2.5 */
10    endfor;
11    /* Hauptschleife: */
12    while queue ≠ ϵ do
13      e := pop_first( queue );
14      if IsCutoff(e) then /* siehe Kap. 8.2.6 */
15        cutoffs := cutoffs ∪ { e }
16      else
17        for all s ∈ π(e)• do
18          x := x + 1; B := B ∪ { b^x }; π(b^x) := s; F' := F' ∪ { (e, b^x) };
19          CalcPossibleExtensions(b^x) /* siehe Kap. 8.2.5 */
20        endfor
21      endif
22    endwhile;
23    /* Bedingungen im Nachbereich von Cut-off–Ereignissen: */
24    for all e ∈ cutoffs do
25      for all s ∈ π(e)• do
26        x := x + 1; B := B ∪ { b^x }; π(b^x) := s; F' := F' ∪ { (e, b^x) }
27      endfor
28    endfor
29  endproc                                                    ■8.3
```

In die Menge *cutoffs* werden sukzessive die als Cut-off–Ereignisse erkannten Knoten als Referenzen auf Ereignisknoten eingefügt; diese Menge ist als lineare Liste realisiert, bei der neue Elemente am Listenanfang angefügt werden.
Mit dem Einfügen eines neuen Bedingungs– und Ereignisknotens sind eine Reihe weiterer Berechnungen verbunden, die aus Gründen der Übersichtlichkeit nicht alle explizit im obigen Algorithmus aufgeführt sind. Bei Einfügen einer Bedingung b^x betrifft dies:

- Aktualisierung der Abbildungen π und π^{-1}: Eintragen von $\pi(b^x)$ im Bedingungsknoten, sowie Aufnahme einer Referenz auf b^x in der π^{-1}–Liste der Stelle $\pi(b^x)$.
- Bei den SND–Ordnungen Ermittlung der i–Tiefe $d_i(b^x)$ gemäß Definition 3.31.

Vor dem Einsortieren eines möglichen erweiternden Ereignisses e_{erw} in die Warteschlange (dies erfolgt innerhalb der Funktion CalcPossibleExtensions(), die in Kapitel 8.2.5 ausführlich erklärt wird) werden die folgenden Daten ermittelt:

- Berechnung der Größe $|[e_{erw}]|$ der lokalen Konfiguration.

  Anders als bei der Tiefe $d()$ (Lemma 2.17) lässt sich diese Berechnung unglücklicherweise nicht effizient aus den direkten Vorgängerknoten herleiten, da lokale Konfigurationen nicht notwendig überschneidungsfrei sein müssen.

  Ein Beispiel für das Präfix in Abb. 8.2 auf Seite 154 soll diese Aussage verdeutlichen: für das Ereignis e_{10} ist $[e_{10}] = \{e_1, e_2, e_5, e_6, e_{10}\}$ und damit $|[e_{10}]| = 5$. Für die Vorgänger-Ereignisse ${}^{\bullet\bullet}e_{10} = \{e_5, e_6\}$ gilt $|[e_5]| = 3$ sowie $|[e_6]| = 4$. Aus diesen Zahlen allein – ohne Ermittlung der Größe der Schnittmenge $|[e_5] \cap [e_6]| = 3$ – läßt sich die Größe von $|[e_{10}]|$ nicht ermitteln.

  Mit Hilfe der Schnittmenge ergibt sich für e_{10} die Größe der lokalen Konfiguration zu $|[e_{10}]| = |[e_5]| + |[e_6]| - |[e_5] \cap [e_6]| + 1$ Diese Rechenregel lässt sich jedoch auf einfache Weise nicht verallgemeinern für den Fall, dass mehr als zwei lokale Konfigurationen nicht überschneidungsfrei sind.

  Eine Berücksichtigung der Schnittmengen erforderte außerdem entweder eine Speicherung der Konfigurationen, oder eine Speicherung der Kardinalität der Konfigurations–Schnittmengen für alle Paare von Ereignissen. Deshalb wird dieser Weg in der Implementierung nicht beschritten.

  Vielmehr wird die Berechnung von $|[e_{erw}]|$ im Rahmen einer Tiefensuche vorgenommen. Liegt kein SND–System vor, wird sie integriert in die nachfolgend diskutierte Ermittlung von $\mathsf{Mark}([e_{erw}])$.

- Berechnung der induzierten Markierung $\mathsf{Mark}([e_{erw}])$.

  Bei Verwenden von SND–Ordnungen wird diese gemäß Satz 3.33 durchgeführt. Die übrigen Ordnungen erfordern wiederum eine Tiefensuche, da sich auch hier

die induzierte Markierung nicht auf einfache Weise aus den induzierten Markierungen der Vorgänger–Ereignisse ableiten lässt.

Mit jeder Übernahme eines Ereignisses e^y aus der Warteschlange in das Präfix (Zeile 17 von Algorithmus 8.3) sind folgende Aktionen verbunden:

- Test, ob e^y die Cut-off–Eigenschaft erfüllt; siehe hierzu Kap. 8.2.6.
- Ist e^y kein Cut-off–Ereignis, dann werden sukzessive die Bedingungen im Nachbereich $e^{y\bullet}$ erzeugt. Mit jeder neuen Bedingung werden die daraus resultierenden erweiternden Ereignisse ermittelt, siehe auch Kap. 8.2.5.
- Aktualisieren der co–Relation (genauer: der co–Matrix) gleichzeitig für alle Bedingungen in $e^{y\bullet}$. Verschiedene Verfahren der Berechnung werden in Kap. 8.2.4 diskutiert.

Wie bereits erwähnt, werden die Bedingungen im Nachbereich von Cut-off–Ereignissen erst nach Berechnung des kompletten Präfixes erzeugt.

8.2.4 Berechnung der co–Relation

Die co–Relation wird bei der Emittlung erweiternder Ereignisse benötigt: Mengen von Bedingungen müssen daraufhin überprüft werden, ob sie eine co–Menge bilden, so dass sie den Vorbereich eines neuen Ereignisses bilden können. In diesem Abschnitt werden drei Methoden zum Testen von co–Mengen untersucht:

- Wiederholte Tiefensuche zum Aufspüren von Konflikten und Kausalitäten. Es werden keine Daten der co–Relation gespeichert
- Speicherung der kompletten co–Relation in Form einer Matrix (aus Symmetriegründen muss nur die untere Dreiecksmatrix verwaltet werden); Ermittlung der Einträge durch Tiefensuche mit anschließender Vorwärtssuche.
- Speicherung der co–Matrix wie vorher, jedoch „lokale“ Ermittlung der Einträge aus den unmittelbaren Vorgängerknoten.

Alle drei Formen der Berechnung sind implementiert worden. Ihre Vor– und Nachteile werden in den folgenden Abschnitten diskutiert.

Überprüfen der co–Eigenschaft durch Tiefensuche

Bei diesem Verfahren wird die co–Relation nicht explizit im Speicher gehalten. Stattdessen dient die Menge von Bedingungen, für die die co–Mengen–Eigenschaft überprüft werden soll, als Ausgangspunkt für eine Tiefensuche. Wird im Zuge dieser Tiefensuche ein Konflikt oder eine kausale Abhängigkeit unter zwei der Ausgangs–Bedingungen festgestellt, dann bilden die Bedingungen keine co–Menge.
Der folgende Algorithmus formalisiert diese Idee. Im Gegensatz zu Alg. 8.1 wird die Tiefensuche hier iterativ statt rekursiv durchgeführt.

Algorithmus 8.4 *Testen, ob eine Menge von Bedingungen co–Menge ist*

Eingabe: Präfix $\mathcal{B} = (B, E, F)$; Menge von Bedingungen $B' \subseteq B$.
Ausgabe: **true**, falls $b_1 \,\mathsf{co}\, b_2$ für alle $b_1, b_2 \in B'$, sonst **false**.

```
1   Boolean function is_co_set( B' ):
2     EmptyStack();
3     IncVisitCounter();
4     for all b ∈ B' do
5       MarkVisited(b);
6       if •b ≠ ∅ ∧ ¬Visited(•b) then
7         MarkVisited(•b); PushEvent(•b)
8       endif
9     endfor;
10    while StackNotEmpty() do
11      e := PopEvent();
12      for all b ∈ •e do
13        if Visited(b) then return False endif;
14        MarkVisited(b);
15        if •b ≠ ∅ ∧ ¬IsMarked(•b) then
16          MarkVisited(•b); PushEvent(•b)
17        endif
18      endfor
19    endwhile;
20    return True
21  endfunc                                          ■8.4
```

Der Vorteil dieses Algorithmus ist, dass kein zusätzlicher Speicherplatz benötigt wird. Der Stack zur Aufnahme von Ereignissen muss maximal $|E|$ Einträge fassen können.
Nachteil des Verfahrens ist der vergleichsweise hohe Zeitverbrauch, insbesondere müssen bei fehlenden Konflikten und Kausalitäten, d. h., die Bedingungen bilden tatsächlich eine co–Menge, ihre gesamten Geschichten abgeschritten werden, im ungünstigsten Fall ergibt sich ein Aufwand von $O(|B| + |E|)$.

In der Implementierung wird deshalb eine einfache Heuristik zur Beschleunigung eingesetzt. Für alle Bedingungen im Nachbereich des zuletzt eingefügten Ereignisses wird die co–Relation einmalig berechnet und in Form eines Bit–Vektors abgelegt. Hierzu wird eine kombinierte Tiefen– und Vorwärtssuche durchgeführt, wobei die Tiefensuche sowieso notwendig ist für die Berechnung der Kardinalität der lokalen Konfiguration, so dass nur die Vorwärtssuche zusätzlich in Rechnung gestellt werden muss.
Dieser Bit–Vektor liefert ein schnelles Ausschluss–Kriterium für Bedingungen, die nicht in co–Relation zur aktuell eingefügten Bedingung stehen und damit nicht zu einer co–Menge im Vorbereich eines erweiternden Ereignisses beitragen können. Nach Ende des Schleifendurchlaufs, d. h. nachdem alle Bedingungen im Nachbereich des letzten Ereignisses behandelt worden sind, wird der Bit–Vektor wieder gelöscht.

Aufstellen der co–Matrix durch kombinierte Tiefen–/Vorwärtssuche

Eine andere, von der Rechenzeit her günstigere, dafür aber recht speicherintensive Lösung bietet das Verwalten der kompletten co–Relation. Dies erfolgt sinnvollerweise in bitweiser Repräsentation für jedes Paar von Bedingungen. Die co–Relation lässt sich in einer Matrix darstellen, deren Spalten und Zeilen von den Bedingungen indiziert sind; wegen der Symmetrie genügt es, die untere Dreiecksmatrix zu speichern. Numerische Indizes ergeben sich aus den Positionsnummern der Bedingungsknoten in den linearen Listen, das Testen zweier Bedingungen auf die co–Eigenschaft hin ist somit – nach Erstellung der Matrix – in $O(1)$ ausführbar.
Abbildung 8.3 zeigt eine solche co–Matrix, die zum Prozess–Netz in Abb. 8.2 gehört. Da für die Konstruktion des Präfixes Bedingungen hinter Cut-off–Ereignissen nicht von Interesse sind, werden sie nicht gespeichert. Die Matrix wächst dynamisch mit jedem Einfügen einer neuen Bedingung in das Präfix um eine Zeile und Spalte. Die letzte Zeile enthält Informationen darüber, mit welchen Bedingungen des bereits konstruierten Teil des Präfixes die zuletzt eingefügte Bedingung in co steht.
Die grau hervorgehobenen Anteile der Matrix sind paarweise in den Zeilen identisch[3], aus der Tatsache heraus, dass Bedingungen im Nachbereich eines Ereignisses e untereinander in co stehen und außerdem mit all denjenigen Bedingungen, die zum Zeitpunkt des Einfügens von e mit e in co stehen. Somit genügt es, lediglich zu jedem Ereignis einer Zeile der Matrix zu speichern, die die Informationen für die Bedingungen im Nachbereich enthält. Im Allgemeinen übersteigt in Prozess–Netzen die Zahl der Bedingungen die der Ereignisse, so dass sich auf diese Weise eine erhebliche Speichereinsparung erzielen lässt gegenüber einem Ablegen in den Bedingungsknoten.

[3] Neben den grau unterlegten Bereichen stehen auch die minimalen Bedingungen b_1, b_2, b_3 untereinander in co und werden auf die beschriebene Weise behandelt: mit dem virtuellen Ereignis $\perp$ wird ein Bitvektor verknüpft, der die Informationen über die minimalen Bedingungen enthält. Dieser Vektor besteht somit regelmäßig aus $|\mathrm{Min}(\mathcal{B})|$ Einsen.

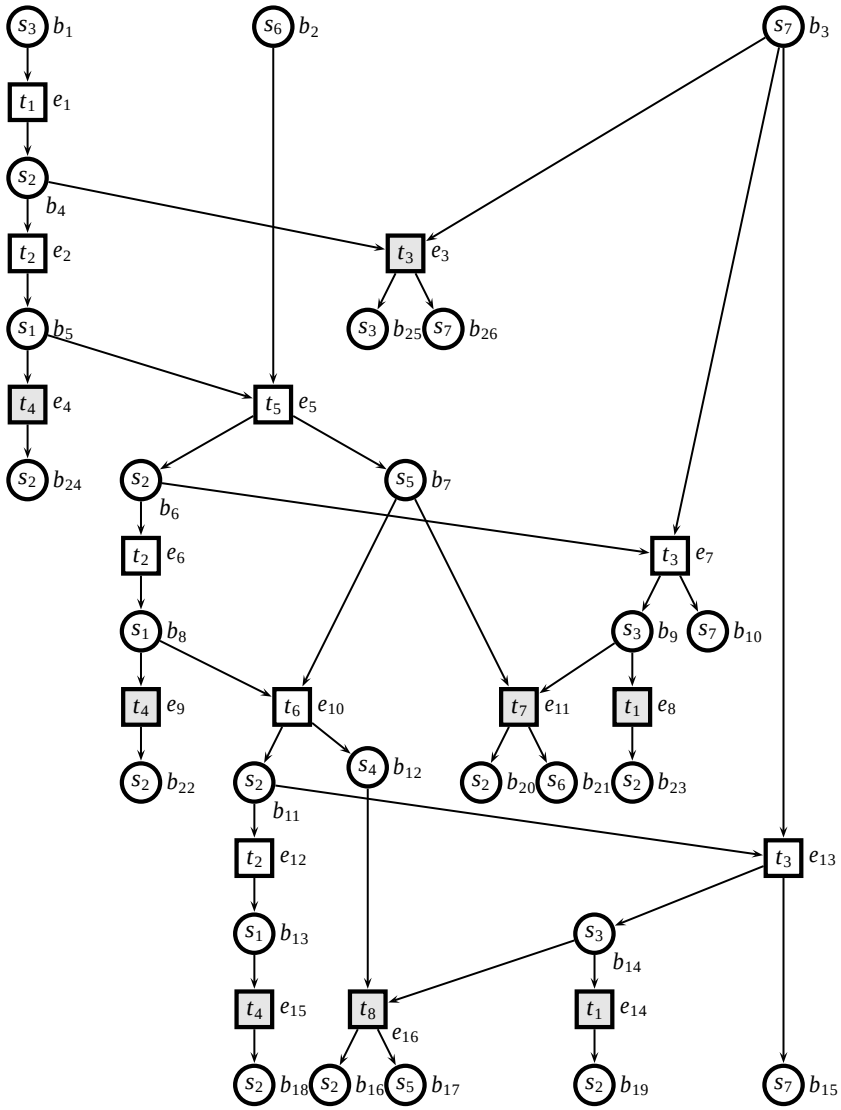


Abbildung 8.2: Vollständiges, mittels $\sqsubset_t$ generiertes Präfix des Systems aus Abb. 8.1(a).

	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}
b_1	–														
b_2	1	–													
b_3	1	1	–												
b_4	0	1	1	–											
b_5	0	1	1	0	–										
b_6	0	0	1	0	0	–									
b_7	0	0	1	0	0	1	–								
b_8	0	0	1	0	0	0	1	–							
b_9	0	0	0	0	0	0	1	0	–						
b_{10}	0	0	0	0	0	0	1	0	1	–					
b_{11}	0	0	1	0	0	0	0	0	0	0	–				
b_{12}	0	0	1	0	0	0	0	0	0	0	1	–			
b_{13}	0	0	1	0	0	0	0	0	0	0	0	1	–		
b_{14}	0	0	0	0	0	0	0	0	0	0	0	1	0	–	
b_{15}	0	0	0	0	0	0	0	0	0	0	0	1	0	1	–

Abbildung 8.3: co–Matrix des Präfixes aus Abb. 8.2.

Eine weitere Vereinfachung würde das Speichern der co–Relation für Ereignisse darstellen, schließlich ist der Vorbereich einer jeden Bedingung eindeutig durch (maximal) ein Ereignis bestimmt. Dieses Verfahren ist jedoch ungeeignet, da sich über die co–Relation von Ereignissen kein Rückschluss auf die co–Eigenschaft von Bedingungen ergibt. Ein Beispiel soll dies verdeutlichen: in Abb. 8.2 gilt $b_7 \,\mathsf{co}\, b_8$, andererseits sind ihre Vorbereiche ${}^\bullet b_7 = \{e_5\}$ und ${}^\bullet b_8 = \{e_6\}$ halbgeordnet, folglich gilt $\neg(e_5 \,\mathsf{co}\, e_6)$.

Trotz der Binärdarstellung ist der Speicherbedarf nicht zu unterschätzen, den die Matrix bei größeren Entfaltungen benötigt. Bei Verwenden von Bit–Vektoren mit minimaler Breite von 8 Bit werden im ungünstigsten Fall $\sum_{i=1}^{|B|} \lceil \log_8 i \rceil$ Bytes belegt – bei 8000 Bedingungen, einer mittleren Präfixgröße, sind das immerhin rund 30,5 MByte; bei einer Breite von 32 Bit noch etwas darüber. Deshalb erlaubt das implementierte Programm, die Zahl der Zeilen in der Matrix zu begrenzen. Ist dieser Wert überschritten, wird für die Ermittlung der co–Relation auf die zuvor beschriebene Methode der Tiefensuche zurückgegriffen, die zwar weniger effizient ist, dafür aber die Erzeugung größerer Präfixe gestattet.

Wie lassen sich nun die Einträge der co–Matrix ermitteln? Eine Lösung bietet wiederum eine Tiefensuche entlang der Ereignisse der lokalen Konfiguration. Auf diese Weise lässt sich jedoch nur die Relation zu Vorgängerknoten ermitteln. Der rückwärts gerichteten Tiefensuche muss sich also eine Vorwärtssuche anschließen, um die Beziehung zu den restlichen Knoten herauszufinden. Somit werden in jedem Fall alle Knoten des Graphen abgeschritten, der Aufwand beträgt demnach $O(|B| + |E|)$ (aktuelle Präfixgröße) für die Ermittlung der Werte einer Zeile der co–Matrix. Ein effizienteres Verfahren wird im nächsten Abschnitt beschrieben.

Aufstellen der co–Matrix durch Hinzuziehen lediglich direkter Vorgängerknoten

Die sukzessive Konstruktion des Präfixes wirft die praktisch relevante Frage auf, wie die dafür notwendigen Informationen ermittelt werden können, ohne sie immer wieder von neuem zu berechnen.
Die zuletzt beschriebene Methode der Ermittlung der Einträge der co–Matrix ist recht aufwändig, da stets der gesamte Graph des Prozess–Netzes abgeschritten werden muss. Bei der Bestimmung der Tiefe eines Ereignisses, Lemma 2.17, war es möglich, lokal vorhandene Informationen der Vorgängerknoten zu verwerten und zusammenzuführen. Eine ähnliche Möglichkeit der lokalen und damit effizienteren Berechnung gestattet auch die co–Relation. Den Ausgangspunkt liefert die folgende Beobachtung.

Satz 8.5 *Fortpflanzen der co–Relation [ER99]*

Sei $\mathcal{B} = ((B, E, F), \pi)$ ein Präfix der Entfaltung $\mathcal{B}_m = ((B_m, E_m, F_m), \pi_m)$ eines Netzsystems, und sei $e_m \in E_m \setminus E$ eine mögliche Erweiterung von $\mathcal{B}$. Für zwei Bedingungen $b_m \in e_m^\bullet$ ($^\bullet$ in $\mathcal{B}_m$) und $b \in B$ gilt:

$$b \; co \; b_m \;\Leftrightarrow\; b \notin {}^\bullet e_m \wedge \forall b' \in {}^\bullet e_m : b \; co \; b'$$

Beweis: Wegen $b_m \notin B$ und $\mathcal{B} \trianglelefteq_p \mathcal{B}_m$ kann $b_m \prec b$ nicht eintreten, es bleibt also zu zeigen: $\neg(b \prec b_m \vee b \# b_m) \;\Leftrightarrow\; b \notin {}^\bullet e_m \wedge \forall b' \in {}^\bullet e_m : b \,\mathsf{co}\, b'$, oder äquivalent: $b \prec b_m \vee b \# b_m \;\Leftrightarrow\; b \in {}^\bullet e_m \vee \exists b' \in {}^\bullet e_m : (b \prec b' \vee b' \prec b \vee b \# b')$.

Wenn die Äquivalenzen (i) $b \prec b_m \;\Leftrightarrow\; b \in {}^\bullet e_m \vee \exists b' \in {}^\bullet e_m : b \prec b'$ und (ii) $b \# b_m \;\Leftrightarrow\; \exists b' \in {}^\bullet e_m : (b' \prec b \vee b \# b')$ nachgewiesen werden können, folgt unmittelbar obige Aussage; (i) gilt offensichtlich, siehe Abb.8.4: $b_3, b_5 \prec b_m$.

(ii)$\Rightarrow$: $b \# b_m$, dann $\exists b_c \in B : b_c \ldots b$ und $b_c \ldots e_m b_m$ sind bis auf b_c disjunkte Pfade. Fall 1: $b' = b_c \in {}^\bullet e_m$, damit folgt $b' \prec b$, z. B. $b_8 \# b_m$ über b_4 in Abb. 8.4.

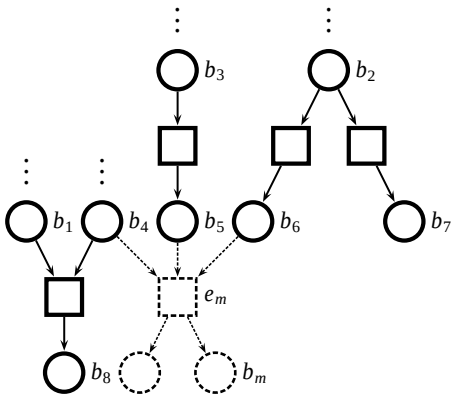


Abbildung 8.4: Zum Beweis von Satz 8.5.

Fall 2: $b' \notin {}^\bullet e_m$, dann $\exists$ Pfad $b_c \ldots b' e_m b_m$, und damit $b' \# b$, z. B. $b_7 \# b_m$ über b_2.
(ii)$\Leftarrow$: $\exists b' \in {}^\bullet e_m : b' \prec b$, also $\exists$ Pfade $b' \ldots e_m b_m$ und $b' \ldots b$, folglich ist $\overline{b \# b_m}$. Gelte andererseits $\exists b' \in {}^\bullet e_m : b \# b'$, dann gibt es zwei Pfade $b' e_m b_m$ und $b' \ldots b$, und damit $b \# b_m$ über b'. ■8.5

Die mit einem erweiternden Ereignis verbundene Zeile in der co–Matrix lässt sich also durch eine geschickte UND–Verknüpfung aus den Zeilen der unmittelbaren Vorgänger berechnen. Der folgende Algorithmus formalisiert diese Operation.

Algorithmus 8.6 *Aktualisieren der co–Matrix nach Einfügen eines Ereignisses*

Eingabe: Präfix $\mathcal{B} = ((B, E, F), \pi)$; zuletzt eingefügtes Ereignis $e \notin E_{cutoff}$, der Nachbereich von e ist noch nicht ins Netz eingefügt.

Ausgabe: BitVektor e.co enthält co–Informationen der Bedingungen in $e^\bullet$.

```
1   procedure update_co_local( e )
2     if e = ⊥ then
3       ⊥.co := NewBitVector(|Min(B)|);
4       ⊥.co[1..|Min(B)|] := 1 ... 1
5     else
6       bits := NewBitVector(|B| + |π(e)•|);
7       bits[1..|B| + |π(e)•|] := 1 ... 1;
8       for all b^k ∈ •e do
9         bits[1..k] := bits[1..k] bitand (•b^k).co[1..k];
10        for i := k + 1 to |B| do
11          if (•b^i).co[k] = 0 then
12            bits[i] := 0
13          endif
14        endfor;
15        bits[k] := 0
16      endfor;
17      e.co := bits
18    endif
19  endproc
```
■8.6

Mit ${}^\bullet b^k$ wird hierbei das eindeutige Ereignis im Vorbereich von b^k gekennzeichnet; für $b^k \in \mathrm{Min}(\mathcal{B})$ ist dies das virtuelle Ereignis $\perp$. Die äußere Schleife in den Zeilen 8 bis 16 untersucht der Reihe nach die Bedingungen im Vorbereich des neuen Ereignisses. Die eigentliche UND–Verknüpfung ist zweigeteilt: da lediglich die untere Diagonalmatrix gespeichert ist, können nur tatsächlich als Zeilen vorhandene Bitinformationen über effiziente byteweise Operationen verarbeitet werden. Die übrigen Bits sind nur spaltenweise zu erreichen, sie werden in der inneren Schleife, Zeile 10 bis 14, verarbeitet.

Der Aufwand für das Aktualisieren der co–Matrix beträgt somit $O(|B| \cdot |{}^\bullet e|)$, in der tatsächlichen Ausführung ist die Routine jedoch etwas schneller, da ein großer Teil der UND–Verknüpfung byteweise vorgenommen werden kann. Der Anteil der schneller zu verarbeitenden Bits ist umso größer, je geringer der Abstand der Ereignisse e und ${}^\bullet b^k$ in der Liste der Ereignisse ausfällt, mit anderen Worten, je später ${}^\bullet b^k$ in das Präfix aufgenommen wurde.

8.2.5 Ermittlung das Präfix erweiternder Ereignisse

Die Berechnung möglicher erweiternder Ereignisse ist der zeitkritischste Teil der Präfix–Generierung. Sie geht einher mit dem kombinatorischen Problem, (co–)Mengen von Bedingungen $B' \subseteq B$ zu bestimmen, für die $\pi(B') = {}^\bullet t$ für eine Transition $t \in T$ gilt.
Das folgende Programmfragment zeigt die entsprechende Stelle in McMillans Algorithmus (d. h. zur Präfix–Generierung für n–beschränkte Systeme mittels Ordnung $\sqsubset_{McM}$).

Algorithmus 8.7 *Ausschnitt aus dem Algorithmus von McMillan [McM92]*

Eingabe: Beschränktes Netzsystem $\Sigma = (S, T, F, M_0)$,
Ausgabe: Präfix $\mathcal{B} = ((B, E, F'), \pi)$ der Entfaltung $\mathcal{B}_m$ von Σ.

```
1   procedure Unfold( S,T,F,M0 ) /*  Hauptroutine  */
2     ⋮
3     if ¬ IsCutoff(e) then
4       for all s ∈ π(e)• do
5         Füge neue Bedingung b in Präfix ein mit •b = e und π(b) = s;
6         GenTrans({b},T);
7       endfor
8     endif
9     ⋮
10  endproc
11  procedure GenTrans( B',T ) /*  Berechnung erweiternder Ereignisse  */
12    if ∄t ∈ T : π(B') ⊆ •t then return endif;
13    if ¬ is_co_set(B') then return endif;
14    for all t ∈ T : π(B') = •t do
15      Füge ein neues Ereignis e in Präfix ein mit •e = B' und π(e) = t;
16      Reihe e in Warteschlange gemäß ⊏McM ein
17    endfor;
18    for all b ∈ B mit b älter als jedes Element in B' do
19      GenTrans(B' ∪ {b},T)
20    endfor
21  endproc
```

■8.7

Klammert man die Zeilen 12 und 13 aus, so berechnet die rekursive Prozedur GenTrans() sukzessive über der Menge der gegenwärtig im Präfix befindlichen Bedingungen alle die Teilmengen, die mindestens die zuletzt eingefügte Bedingung b (Zeile 6) enthalten. Es gibt jeweils $2^{|B|-1}$ solche Teilmengen für eine aktuelle Präfixgröße von $|B|$ Bedingungen.
Um diese exponentielle Zahl von Teilmengen zu reduzieren, hat McMillan mehrere Techniken vorgeschlagen:

- Es brauchen nur solche Teilmengen von Bedingungen betrachtet und anschließend expandiert zu werden, die, abgebildet auf das Ausgangssystem, im Vorbereich wenigstens einer Transition vorzufinden sind. Nur Teilmengen B' mit Kardinalität $|B'| \leq \max\{|{}^\bullet t| \mid t \in T\}$ müssen somit berücksichtigt werden. Dieser Test ist in Zeile 13 von Alg. 8.7 realisiert.

  Für eine praktische Anwendung des Algorithmus ist ein solcher Test – insbesondere für Systeme mit mehreren tausend Transitionen – nicht besonders effizient, da schlimmstenfalls bei jedem Aufruf von GenTrans(), also für jede kombinatorisch mögliche Teilmenge, $|T|$ viele Vergleiche ausgeführt werden müssten. Eine mögliche Verbesserung wäre eine Vorab–Berechnung und Speicherung der vorkommenden Transitions–Vorbereiche, zur Beschleunigung der Vergleiche sollten diese außerdem sinnvoll geordnet werden.

- Lediglich solche Teilmengen brauchen untersucht und expandiert zu werden, die wenigstens eine co–Menge bilden.

  Die co–Mengen–Eigenschaft ist essenziell, andernfalls würde kein Occurrence–Netz erzeugt (Verstoß gegen Def. 2.14(iii)). Der entsprechende Test ist in Zeile 14 von Alg. 8.7 zu finden.

Der Test in Zeile 19 (Finden der „ältesten“ Bedingung) ist relativ einfach umzusetzen, wenn die Bedingungen eine fortlaufende Nummer erhalten. Trotz dieser Techniken ist die Ermittlung erweiternder Ereignisse nach diesem Verfahren insgesamt recht ineffizient, da viele Kombinationen von Bedingungen berechnet werden, die als Vorbereich eines Ereignisses ausscheiden. Im Rahmen der Implementierung ist die Suche nach den Ereignissen deshalb im Vergleich zu Algorithmus 8.7 etwas auf den Kopf gestellt worden.
Anstatt die in Frage kommenden Teilmengen sukzessive zu generieren und mit den Transitions–Vorbereichen abzugleichen, werden von vornherein nur mögliche Mengen von Bedingungen dahingehend untersucht, ob sie eine co–Menge bilden.
Wie ist an diese Mengen auf konstruktive Weise heranzukommen? Gesucht werden mögliche neue Ereignisse im Nachbereich der zuletzt eingefügten Bedingung b. Potenzielle Kandidaten für solche Ereignisse sind demnach beschriftet mit einer Transition $t \in \pi(b)^\bullet$. Wie in Kap. 8.2.1 bereits erwähnt, wird die Umkehrabbildung π^{-1} zu jedem

Stellenknoten als Liste gespeichert. Die in Frage kommenden Mengen können also aus diesen Listen $\pi^{-1}(s)$ für $s \in {}^\bullet t$ kombinatorisch gewonnen werden.
Der folgende Algorithmus formalisiert diese Idee.

Algorithmus 8.8 *Erzeugen erweiternder Ereignisse*

Eingabe: Sicheres Netzsystem $\Sigma = (S, T, F, M_0)$, ein Präfix $\mathcal{B} = ((B, E, F'), \pi)$ der maximalen Entfaltung $\mathsf{Unf}(\Sigma)$; Bedingung $b \in B$ im Maximum des Präfixes. Globale Variablen y (Zähler der Ereignisse) und Warteschlange *queue*.

Ausgabe: $\mathcal{B}$ mit erweiternden Ereignissen im Nachbereich von b; Verweise auf die neuen Ereignisse sind in *queue* gemäß verwendeter Ordnung $\sqsubset$ eingereiht.

```
1   procedure CalcPossibleExtensions(b)
2     for all t ∈ π(b)• do
3       if |•t| = 1 then
4         /* Einfügen eines neuen Ereignisses mit seiner Eingangskante. */
5         y := y + 1; E := E ∪ { e^y }; π(e^y) := t; F' := F' ∪ { (b, e^y) };
6         insert_queue(queue, e^y)
7       elsif |•t| = 2 then
8         { s } := •t \ {π(b)};
9         for all b' ∈ π^{-1}(s) : b' co b do
10          y := y + 1; E := E ∪ { e^y }; π(e^y) := t;
11          F' := F' ∪ { (b, e^y), (b', e^y) };
12          insert_queue(queue, e^y)
13        endfor
14      else /* |•t| ≥ 3 */
15        n := |•t| − 1; { s_1, ..., s_n } := •t \ {π(b)};
16        /* K enthält die möglichen Kombinationen von Bedingungen, */
17        /* die in co–Relation mit der aktuellen Bedingung b stehen. */
18        K := { (b_1, ..., b_n) | b_i ∈ π^{-1}(s_i), b co b_i, 1 ≤ i ≤ n };
19        for all (b_1, ..., b_n) ∈ {(b_1, ..., b_n) ∈ K | b_i co b_j, 1 ≤ i < j ≤ n} do
20          y := y + 1; E := E ∪ { e^y }; π(e^y) := t;
21          F' := F' ∪ { (b, e^y), (b_1, e^y), ..., (b_n, e^y) };
22          insert_queue(queue, e^y)
23        endfor
24      endif
25    endfor
26  endproc
```

■8.8

In der äußeren Schleife werden alle Transitionen t des Netzsystems betrachtet, die im Nachbereich $\pi(b)^\bullet$ liegen. Da die Abbildungen π und π^{-1} gespeichert sind, ist ein di-

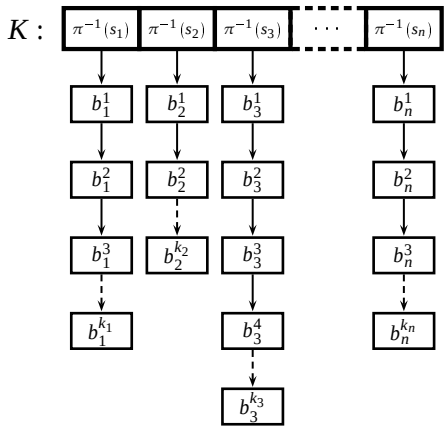


Abbildung 8.5: Realisierung der Menge K für ${}^\bullet t = \{\pi(b), s_1, s_2, \ldots, s_n\}$.

rekter Zugriff auf diesen Nachbereich unmittelbar möglich. In Abhängigkeit von der Größe des Vorbereichs von t wird nun die Suche nach co–Mengen durchgeführt.

- Für den einfachen Fall $|{}^\bullet t| = 1$ (Zeile 3) kann ein t entsprechendes Ereignis unmittelbar in das Präfix hinter b eingefügt werden; eine einzelne Bedingung muss nicht auf die co–Eigenschaft hin getestet werden.
- Der Fall $|{}^\bullet t| = 2$ (Zeile 7) erfordert eine Überprüfung der co–Eigenschaft: jede Bedingung $b' \in \pi^{-1}(s)$ wird daraufhin getestet, ob $b' \,\mathsf{co}\, b$ gilt. Ist dies der Fall, dann kann ein gemeinsames Ereignis im Nachbereich von b und b' eingerichtet werden.
- Im allgemeinen Fall, $|{}^\bullet t| \geq 3$ (Zeile 14), müssen alle Kombinationen potenzieller Bedingungen getestet werden, ob diese eine co–Menge bilden. Um die Zahl der Kombinationen von vornherein zu reduzieren, werden in einem ersten Schritt nur diejenigen Bedingungen in die Überlegung einbezogen, die in co mit b stehen. Dieser Test kann jeweils in $O(1)$ ausgeführt werden, wenn, wie in Kapitel 8.2.4 beschrieben, die co–Relation für Bedingung b zuvor als Bit–Vektor gespeichert wurde.

Die Menge K In Zeile 18/19 wird nicht komplett gespeichert, vielmehr wird für deren kombinatorische Ermittlung ein $(|{}^\bullet t| - 1)$–dimensionales Feld von Listen angelegt, wie es Abbildung 8.5 andeutet. Für jede Stelle $s' \in {}^\bullet t \setminus \pi(b)$ ist darin eine Liste von (Referenzen auf) Bedingungen $b' \in \pi^{-1}(s')$ enthalten, für die $b' \mathsf{co}\, b$ gilt.
Auf diese Weise wird eine einfache und effiziente Vorauswahl von möglichen Kandidaten getroffen. Die Berechnung der Kombinationen wird direkt auf dieser Datenstruktur

mit einem einfachen Backtracking–Algorithmus ausgeführt. Für die notwendigen Tests auf Einhaltung der co–Mengen–Eigenschaft wird auf die in Kapitel 8.2.4 diskutierten Verfahren zurückgegriffen; bei Verwenden einer co–Matrix sind dies einfache Abfragen von Bit–Vektoren.
Der Fall $|{}^\bullet(t)| = 2$ hätte auch dem allgemeinen Fall zugeordnet werden können. Da in einer Vielzahl der untersuchten Netzsysteme jedoch ein großer Anteil Transitionen beteiligt ist, die zweielementige Vorbereiche haben, lohnt sich eine Optimierung für diesen Spezialfall.

Die Zeitkomplexität eines Aufrufs dieser Prozedur errechnet sich im schlechtesten Fall zu

$$O((\max\{|s^\bullet| \mid s \in S\}) \cdot (\frac{|B|}{\xi})^{\xi} \cdot \frac{\xi^2}{2} \cdot t_{\text{insert\_queue}}),$$

wobei wieder $\xi = \max\{|{}^\bullet t| \mid t \in T\}$ wie vorher gilt. Der erste Faktor ergibt sich aus der Anzahl äußerer Schleifendurchläufe, der zweite Faktor gibt die maximale Anzahl von Kombinationen an Bedingungen wieder, die innerhalb der Menge K untersucht werden könnten, multipliziert mit der Anzahl der maximal notwendigen Tests der co–Relation, wobei hier die zeitgünstigste Variante vorausgesetzt wird, bei der ein Zugriff auf die komplette co–Matrix möglich ist. Schließlich bringt letzte Faktor $t_{\text{insert\_queue}}$ die für das Einfügen in die Schlange benötigte Zeit ein; diese hängt von mehreren Faktoren ab: der Größe des bereits berechneten Teils des Präfixes, der Länge der Schlange und nicht zuletzt von der Wahl der adäquaten Ordnung $\sqsubset$.
Diese obere Schranke wird nur erreicht in Netzsystemen mit einem hohen Rückwärtsverzweigungsgrad bei den Transitionsknoten und einem hohen sequenziellen Verzweigungsgrad innerhalb des Netzsystems. Unter diesen Voraussetzungen muss dann eine große Anzahl von Bedingungen auf die co–Eigenschaft hin überprüft werden.
Im Allgemeinen ist jedoch – wenn überhaupt – der Rückwärtsverzweigung nur einer kleinen Menge von Transitionen relativ groß. In durchschnittlichen Netzsystemen besitzt der weit überwiegende Teil der Transitionen Vorbereiche mit Größen kleiner oder gleich drei.

8.2.6 Überprüfen der Cut-off–Eigenschaft

Der Schlüssel zur Terminierung der Präfix–Generierung liegt im Auffinden der Cut-off–Ereignisse. Dazu sind im theoretischen Teil eine Reihe von Ordnungen $\sqsubset$ vorgestellt worden, die die diese Ereignisse durch Vergleich der lokalen Konfigurationen bzw. der lokalen Sichten ermitteln.
Um für ein Ereignis die in Def. 3.11 beschriebene Cut-off–Eigenschaft nachzuweisen, ist zunächst die von seiner lokalen Konfiguration induzierten Markierung im Ausgangssystem zu ermitteln. Hierzu wurde in Def. 3.3 die Funktion Mark eingeführt.

Die Implementation von Mark weicht etwas von dieser Definition ab, da zum Zeitpunkt des Überprüfens der Cut-off–Eigenschaft für ein Ereignis e selbiges noch keine Bedingungen im Nachbereich besitzt – diese werden ja gerade erst dann eingefügt, wenn e kein Cut-off–Ereignis ist. Aus diesem Grund wird Mark tatsächlich errechnet aus

$$\mathrm{Mark}(e) = \pi(e)^\bullet \cup \pi(\,(\,[e]^\bullet \cup \mathrm{Min}(\mathcal{B})\,) \setminus {}^\bullet[e]\,)$$

Der Nachbereich $[e]^\bullet$ ist dabei wiederum bezogen auf den bereits konstruierten Teil des Präfixes. Die Berechnung von $\mathsf{Mark}(e)$ erfolgt zusammen mit der Ermittlung von $||[e]||$ in derselben Tiefensuche.
Die so ermittelte Markierung wird verglichen mit den bislang von lokalen Konfigurationen induzierten Markierungen. Für einen raschen Vergleich werden alle Markierungen über eine Hash–Tabelle indiziert.
Zu Beginn der Hauptroutine des Präfix–Generators, Alg. 8.3, wird diese Hash–Tabelle initialisiert mit der Startmarkierung M_0, die mit der Konfiguration $[\bot]$ verbunden ist.

8.3 Präfix–Generierung für DB(PN)^2-Programme

In den letzten Abschnitten sind zahlreiche Details der Umsetzung des Präfix–Generators vorgestellt worden, wie sie sich in der Implementierung wiederfinden, die inzwischen Teil verschiedener Werkzeuge ist [PEP98, CPN99, SSE99]. Darüber hinaus besitzt das Programm diverse Erweiterungen, mit denen über das Netz des Präfixes hinaus zusätzliche Ausgaben für nachgeschaltete Programme erzeugt werden können, sowie zahlreiche Schalter, mit denen die Erzeugung des Präfixes direkt beeinflusst werden kann. Im Anhang A.2 sind die wichtigsten von ihnen tabellarisch mit einer kleinen Funktionsbeschreibung zusammengefasst.
Die in diesem Kapitel vorgestellten Einzelheiten gelten darüber hinaus auch für die Umsetzung des Präfix–Generators für beschränkte Netzsysteme nach [ERV00], der als Netzsemantik Executions verwendet, siehe auch Kapitel 2.4. Nicht zuletzt finden die beschriebenen Techniken bei der Implementierung des Präfix–Generators für DB(PN)^2–Programme Verwendung, lediglich die Berechnung erweiternder Ereignisse erfolgt dabei etwas anders.
Die Konstruktion der Petri–Box des Kontrollflusses eines DB(PN)^2–Programmes wird hier nicht in aller Ausführlichkeit erläutert. Für das Lesen eines Programms werden Standard–Werkzeuge benutzt (der Parser ist mit dem UNIX–Standardwerkzeug yacc erzeugt, dabei konnte unmittelbar die in der Arbeit verwendete Syntax verwendet werden), die Konstruktion von Box–Netzen auf Basis der kompositionellen Box–Operationen erfolgt mit Datenstrukturen und Techniken, die in [Röm93] beschrieben sind.
Die Datenstruktur zur Speicherung des Präfixes ist für die Bedingungen und Ereignisse erweitert worden, um die neu hinzugekommenen Informationen (Variablenwerte

und atomare Ausdrücke) aufnehmen zu können. Die Repräsentation von DB(PN)^2–Programmvariablen erfolgt in Form von Strukturen, in denen der Name der Variablen zusammen mit ihrem Definitionsbereich und dem Startwert abgelegt sind.

Kapitel 9

„Es macht Spaß, Computerprogramme zu schreiben, und es macht Spaß, gut geschriebene Computerprogramme zu lesen. Eine der größten Freuden des Lebens kann die Komposition eines Computerprogramms sein, von dem man weiß, die Lektüre wird anderen ebenso Freude bereiten wie einem selbst.“
DONALD E. KNUTH, LITERATE PROGRAMMING, CLSI 1992

Verifikationswerkzeuge

Die Entwicklung (nachweislich) fehlerfreier Software und Hardware für verteilte, reaktive Systeme ist immer noch eine Herausforderung und hat in den letzten Jahren zunehmend an Bedeutung gewonnen. Design und Analyse nebenläufiger Systeme mit Hilfe von Petri–Netzen ist dabei einer der bekanntesten formalen Ansätze in dieser Richtung. Eine Reihe von Werkzeugen, die die unterschiedlichsten Techniken zum Nachweis der Korrektheit von Programmen und Modellen einsetzen, sind dabei im Laufe der Zeit entstanden und werden ständig weiterentwickelt und verbessert. Führte man früher vorwiegend Simulationen aus, so kommen im Bereich der formalen Verifikation zunehmend Model–Checking–Verfahren zum Einsatz, die mit Hilfe verschiedener Ansätze dem Problem der Zustandsexplosion zu trotzen versuchen.

Einer dieser Ansätze ist die in dieser Arbeit diskutierte Technik der Analyse auf der Basis eines endlichen, vollständigen Präfixes der Netzentfaltung. Nach und nach wird diese Technik in Werkzeugen zur automatischen Verifikation nebenläufiger Systeme aufgenommen, oft nicht allein, sondern im Verbund mit (in Konkurrenz zu?) anderen Techniken zur Reduzierung des Zustandsraumes, etwa den weit verbreiteten BDD–basierten Verfahren.

In diesem Kapitel wird untersucht, in welchem Umfang die Technik der Netzentfaltungen bislang Einzug in Verifikationswerkzeuge gefunden hat und auf welche Weise ein Benutzer mit ihnen arbeiten kann. Die Arbeitsschritte beim Umgang mit zwei konkreten Werkzeugen, PEP [PEP98, MRE96] sowie (Model Checking) Kit [SSE99], werden dazu näher betrachtet. Zusammen decken sie das derzeit breiteste Spektrum an Präfix–basierten Methoden ab.

9.1 Vom Modell zur Verifikation

Beide Werkzeuge, PEP und Kit, bestehen intern weithin aus den gleichen Programm–Modulen, daher ist auch die Arbeit mit ihnen in großen Teilen identisch. Der Hauptunterschied liegt in der Bedienung: PEP besitzt eine graphische Oberfläche, das Kit präsentiert sich dem Benutzer als ein einzelnes ausführbares Programm, das über die Angabe von Optionen gesteuert wird; die eigentliche Verifikation geht dann automatisch vonstatten. Bei PEP müssen die auszuführenden Aktionen und Übersetzungsvorgänge explizit durch Anklicken (in der richtigen Reihenfolge) ausgelöst werden.
Dem Benutzer werden von beiden Werkzeugen eine Reihe von Eingabesprachen zur Modellierung von verteilten Systemen angeboten, gemeinsam unterstützt werden Stellen / Transitions–Netze und die Notation B(PN)^2 (und damit implizit auch der Box–Kalkül). Zudem erlauben sie (voneinander abweichende) Darstellugen kommunizierender sequenzieller Automaten.
Intern sind beide Werkzeuge modular aufgebaut, d. h. sie sind in Form von eigenständigen Programmen implementiert, die jeweils eine spezielle Aufgabe erfüllen. Jeder der Übersetzungsschritte wird von einem eigenen externen Programm ausgeführt, der Datenaustausch erfolgt über externe Dateien, in denen die Netze, Programme, Terme etc. in speziellen Formaten abgelegt sind. Die halbordnungsbasierte Verifikation von verteilten Systemen lässt sich bei beiden Werkzeugen in die folgenden Phasen gliedern:

1. Modellieren des zu untersuchenden Systems in einer der unterstützten Eingabesprachen. Im Folgenden wird davon ausgegangen, dass die Systeme als B(PN)^2–Programm oder in der Automatendarstellung spezifiziert werden. Bei Petri–Netzen kann direkt mit Phase 4. weitergemacht werden.

2. Übersetzen des eingegebenen Systems in ein High–Level–Netz (genauer M–Netz, [BFF$^+$95]). Alternativ kann das Programm in PEP auch in einen prozess–algebraischen Term der Box–Algebra übersetzt werden.

3. Auffalten des High–Level–Netzes (bzw. Übersetzen des Box–Ausdrucks) in ein Low–Level–Petri–Netz, d. h., in ein 1–sicheres Stellen / Transitions–Netz.

4. Berechnen des vollständigen Präfixes der Entfaltung des Low–Level–Netzes.

5. Durchführung der eigentlichen Verifikation, d. h. Starten des gewünschten Verifikationsmoduls, z. B. durch Wahl eines der Model–Checker, begleitet von der Eingabe einer zu überprüfenden Eigenschaft.

6. Warten auf das Resultat der Berechnung...

Nicht alle der Übersetzungsvorgänge 2. bis 4. müssen dabei explizit ausgelöst werden, in den Werkzeugen sind hierfür Schalter bzw. Ausführungsfolgen zur weitgehenden

Automatisierung vorhanden; die Übersetzungen sind aber notwendig. Die in Kapitel 6 vorgestellte Methode zur DB(PN)^2–Präfix–Generierung optimiert gerade diese Schritte. Es ist geplant, den DB(PN)^2–Präfix–Generator in eines dieser Werkzeuge zu integrieren.

9.2 Kontrollpunkte in DB(PN)^2–Programmen

Zur Referenzierung von Zuständen eines DB(PN)^2–Programms in erreichbaren Markierungen oder in Formeln für die Model–Checker müssen selbige mit einem symbolischen Bezeichner versehen werden. Als Möglichkeit bietet sich an, die intern verwendeten Namen der entsprechenden Stellenknoten in der Netz–Darstellung des Programmes zu verwenden, wie es etwa im PEP–Tool realisiert ist: Zustandsnamen werden dort in einem speziellen Editor per Mausklick auf zwei atomare Aktionen des Programmtextes ermittelt, indem – über ein eingebautes Referenzsystem – die Netzdarstellung des Programmes konsultiert und dort der entsprechende Stellenname ermittelt wird. Um diese Möglichkeit zu nutzen, muss also bereits die Netzdarstellung vorliegen. Wird das Programm geändert, so ändern sich mit den automatisch generierten Stellennamen auch die Kontrollpunkte; sie müssen also nach jeder Modifikation neu ermittelt werden.

In B(PN)^2 gestaltet sich die Zuordnung von Kontrollpunkten des Programms zu entsprechenden Stellenknoten in der Netzdarstellung nicht ganz trivial aufgrund der möglichen Schachtelung des Parallel–Operators, siehe auch [Rie94]. Deshalb müssen für die Spezifikation eines Programmzustands bisweilen zwei Aktionen herangezogen werden, je eine vor und nach dem Erreichen des gewünschten Zustands. Das Kit unterstützt eine einfache Erweiterung von B(PN)^2, die das Setzen von Kontrollpunkten in Form von Marken an Stellen erlaubt, an denen die Syntax auch atomare Aktionen gestattet. Dies ermöglicht eine in vielen Fällen ausreichende Möglichkeit (und ist gegenüber der anklickbaren PEP–Variante erheblich bedienungsfreundlicher), löst das Problem aber nicht in allen Fällen.

In DB(PN)^2 hingegen ist das Setzen von Kontrollpunkten recht einfach möglich, da die einzelnen Programm–Komponenten als SND–Systeme darstellbar sind und somit jede Komponente einen eindeutigen Zustand einnimmt, in dem, bezogen auf die Netzdarstellung, genau ein Stellenknoten markiert ist.
Kontrollpunkte in DB(PN)^2–Programmen werden gekennzeichnet durch Angabe von speziellen Bezeichnern zwischen den jeweiligen Anweisungen. Um sie eindeutig zu parsen, muss ihnen das Symbol **@** vorangestellt werden. Das folgende Programm zeigt ein Beispiel für das Setzen von Kontrollpunkten.

Programmtext 9.1 *Petersons Algorithmus [Pet81] in DB(PN)^2 mit Kontrollpunkten*

```
1   var in1,in2: bool init false;
2   var hold: {1,2} init 1;
3     do <in1:= true>; <hold:= 1>;
4                      if <not in2 or hold= 2> fi;
5                        @incs1 /* Kritischer Abschnitt 1 */
6                      <in1:= false>
7     od
8   ||
9     do <in2:= true>; <hold:= 2>;
10                     if <not in1 or hold= 1> fi;
11                       @incs2 /* Kritischer Abschnitt 2 */
12                     <in2:= false>
13    od                                                    ■9.1
```

Eine LTL–Formel zur Überprüfung der Einhaltung der Mutex–Eigenschaft beim obigen Programm kann damit folgendermaßen formuliert werden:

$$\mathbf{G}(\neg(\mathit{incs1} \wedge \mathit{incs2}))$$

Neben den Kontrollpunkten können auch Werte von Programmvariablen in Formeln spezifiziert werden, beispielsweise für obiges Beispiel die LTL–Formel $\mathbf{F}(\mathit{in1} = \mathbf{true})$ oder, im Rahmen eines anderen Tests, die zu untersuchende Markierung $\{\mathit{incs1}, \mathit{hold} = 1\}$. Bei der Ausgabe des Netzes des Präfixes wird die Abbildung π in einem Format geschrieben, das den (virtuellen) Stellenknoten des Datenflusses einen Namen gibt, der den jeweils eingenommenen Variablenwert oder Kanalzustand widerspiegelt.

9.3 Schnittstellen

Schon früh während der Implementierung des Präfix–Generators ist die Notwendigkeit entstanden, die erzeugten Netze abzuspeichern, um auf diesem Wege das Netz anderen Verifikationskomponenten zur Verfügung stellen zu können und die einzelnen Programmteile weithin unabhängig voneinander zu halten, da sie von verschiedenen Personen und zum Teil an verschiedenen Orten entwickelt wurden und werden. Diese Modularität der Werkzeugkomponenten hat sich über die Zeit bestens bewährt, da auf diesem Wege die einzelnen Programmteile von den jeweiligen Entwicklern am besten gepflegt werden können.
Für (allgemeine) Petri–Netze ist ein kompaktes Dateiformat entwickelt worden, das mehrere Anforderungen erfüllt: zum einen ist es textbasiert und erlaubt daher ein leichtes Überprüfen und manuelles Überprüfen und Ändern von Einträgen, zum anderen werden nur die wirklich notwendigen Informationen gespeichert, und das mit so wenig Platzbedarf wie irgend möglich. Dieses Dateiformat ist, mit einigen Erweiterun-

gen versehen, inzwischen das Standardformat von PEP [PEP98, MRE96][1] zur externen Netz–Repräsentation, durch die Verbreitung des Präfix–Generators und anderer Verifikationsmodule wird es mittlerweilse auch in anderen Werkzeugen unterstützt [CPN99, SSE99].
Im Rahmen der Darstellung von SND–Systemen ist dieses Format etwas erweitert worden, um zusätzlich zu jedem Stellenknoten die Komponentennummer speichern zu können.[2]
Die erzeugten Präfixe können darüber hinaus in zusätzlichen Formaten gespeichert werden, die zum einen – wegen der teilweise immensen Größe der Prozess–Netze – eine noch kompaktere, binär codierte Darstellung besitzen, und zum anderen eine Reihe von zusätzlichen Informationen enthalten, wie etwa Aufzählung der Cut-off– und dazu korrespondierender Ereignisse, Ausgabe der Abbildung πbzw. der co–Matrix, oder Aufzählung der globalen Konfigurationen des Präfixes (diese werden für den Model–Checker nach [Esp93, Gra95, Gra97] benötigt).
In Anhang A.2 werden die wichtigsten Parameter der Präfix–Generatoren aufgezählt und kurz beschrieben, darunter auch die Wahl der Ein– und Ausgabeformate.

9.4 Werkzeuge mit Netzentfaltern – eine Übersicht

Die auf dem McMillan–Algorithmus aufsetzende Technik der Netzentfaltungen hat mittlerweile Einzug gehalten in diverse Werkzeuge zur Modellierung, Simulation und Analyse verteilter Systeme. Nachfolgend werden einige von ihnen (in chronologischer Reihenfolge bezüglich der Integration von Präfix–Generatoren) aufgezählt und die angebotenen Funktionalitäten kurz beleuchtet. Bei allen diesen Werkzeugen handelt es sich um Entwicklungen aus dem akademischen Bereich, industrielle oder gar kommerzielle Entwicklungen auf diesem Sektor gibt es derzeit noch nicht. Soweit die Werkzeuge über das Internet verfügbar sind, ist in den angegebenen Referenzen ein Hinweis auf die entsprechenden Stellen enthalten. Ein allgemeinerer Vergleich von netzbasierten – nicht unbedingt die Entfaltungstechnik verwendenden – Werkzeugen ist in [Stö98] zu finden.

PEP Das Werkzeug PEP (*Petri–Netz–basierte Entwicklungs– und Programmierumgebung*) [PEP98, MRE96] hat bereits mehrfach in dieser Arbeit Erwähnung gefunden. Die Arbeiten an PEP fanden haupsächlich im Zeitraum 1993 bis 1998 statt, sie begannen an der Universität Hildesheim, später kamen die Humboldt–Universität zu Berlin sowie die

[1] Unter der in der Referenz [PEP98] angegebenen Web–Adresse von PEP kann eine ausführliche Beschreibung dieses Dateiformats heruntergeladen werden.

[2] Für den interessierten Leser, der das PEP–Dateiformat kennt: Komponentennummern werden über das mnemonische Kürzel C*n* (großes „C“) angegeben.

Universität Oldenburg hinzu. PEP ist das erste Werkzeug, das die Entfaltungstechnik m Zusammenhang mit einem Model–Checker unterstützt.
PEP besitzt eine graphische Benutzeroberfläche, in die (graphische) Editoren für die zahlreichen Eingabesprachen eingebaut sind, die zur Modellierung verteilter Systeme unterstützt werden. Zu diesen Sprachen gehört die Notation B(PN)^2, ein Fragment von SDL [SDL92], spezielle High– sowie Low–Level–Netze, eine Darstellung kommunizierender Automaten, außerdem die Prozess–Algebra des Box–Kalküls.
Auf Verifikationsseite sind zum einen Komponenten anderer Werkzeuge in die graphische Oberfläche von PEP integriert worden, hierzu zählen Analyseverfahren aus den Werkzeugen SPIN (ein Halbordnungs–basierter Model Checker, [Hol97]), INA (Untersuchung struktureller Eigenschaften von Netzen, [Sta92]) und SMV (BDD–basierter Model–Checker, [SMV98]). Zum anderen werden eine Reihe originärer Verfahren angeboten, beispielsweise ein Model–Checker für S–Netzsysteme oder auf Linearer Programmierung basierender Verfahren. Der in dieser Arbeit beschriebene Präfix–Generator wird in PEP vornehmlich genutzt für den in Kapitel 7.3 beschriebenen S4–Model–Checker.
Im Gegensatz zu den anderen hier aufgezählten Werkzeugen mit graphischer Oberfläche kann das Netz eines Präfixes in PEP nicht visualisiert werden. Für die in PEP angebotenen Analyseverfahren ist dies auch, schon allein wegen der Größe und Unübersichtlichkeit dieser Netze, nicht unbedingt notwendig.

SIS Ein interaktives Werkzeug für die Synthese, Analyse und Optimierung sequenzieller Schaltkreise ist SIS (*System for Sequential cIrcuit Synthesis*, [SSL$^+$92]), das seit etwa 1990 an der Berkeley Universität von Kalifornien entwickelt wird. In den ersten Versionen fand die Kommunikation mit dem Werkzeug über eine einfache Kommandosprache statt, inzwischen ist auch die Verwendung einer graphischen Oberfläche möglich, vornehmlich für die Anzeige der Netzgraphen.
Auf Eingabeseite können Systeme in speziellen Formaten zur Beschreibung von sequenziellen Schaltkreisen spezifiziert werden (sogenannte *State* – und *Signal Transition Graphs*), die sich auch für die Modellierung als Petri–Netze verwenden lassen. Die Hauptkomponenten dieses Werkzeugs unterstützen die Optimierung (z. B. Zustandsminimierung, *Retiming*), Synthese (z. B. Einfügen von *Delays*, Eliminieren von Gefahrensituationen (*Hazards*)) und Manipulation der eingegebenen Systeme.
Die Entfaltungstechnik ist in dieses Werkzeug 1995 integriert worden, und zwar zum Aufdecken von Verklemmungen nach dem Algorithmus von McMillan [McM92] aus Kapitel 7.2. Neben der ursprünglichen Ordnung $\sqsubset_{McM}$ kann alternativ auch eine optimierte Ordnung [KKT$^+$98] für die Konstruktion des Präfixes ausgewählt werden.

CPN-AMI Seit 1995 wird an der Universität Pierre & Marie Curie in Paris an CPN–AMI gearbeitet [CPN99], einer Petri–Netz–basierten CASE–Umgebung (*Computer Aided Soft-*

ware Enginieering). Eingabesprache ist eine spezielle High–Level–Netzdarstellung, eine graphische Oberfläche ermöglicht den Zugriff auf die verschiedenen Systemkomponenten. Der Schwerpunkt des Werkzeugs liegt in der Modellierungskomponente und der Simulation der Netze. Integriert in CPN–AMI sind außerdem diverse Verifikationsmöglichkeiten, die größtenteils auf Einbindung von Komponenten anderer Werkzeuge beruhen.

Für die Verifikation müssen die High–Level–Netze auch hier zunächst aufgefaltet werden in Low–Level–Netze. Diese können untersucht werden mit einer Reihe verschiedener Techniken, darunter strukturelle Invarianten (aus dem Werkzeug GreatSPN [Chi87]), Analyse des Erreichbarkeitsgraphen mit Methoden aus PROD [VHHP95] und GreatSPN2, darüber hinaus ist die Implementierung des in dieser Arbeit beschriebenen Präfix–Generators in das Werkzeug integriert. Leider schließen sich bislang keine weiterreichenden Verifikationsmethoden an, die auf dem Präfix aufsetzen; lediglich eine graphische Aufbereitung des Netzes des Prozess–Netzes ist in der derzeit aktuellen Version 2.4 vorgesehen.

VIP Das VIPtool (*Verification of Information systems by evaluating partially–ordered runs of Petri nets* [Fre98]) ist eine Gemeinschaftsarbeit der Universitäten Karlsruhe und Frankfurt. Hierbei handelt es sich um ein graphisch orientiertes, Halbordnungs–basiertes Werkzeug für die Eingabe, Simulation und Verifikation von speziellen Prädikate / Transitionen–Netzen, als Hauptanwendungsgebiet wird die Untersuchung komplexer industrieller Systeme adressiert.

Die Analysekomponente des Werkzeugs fußt neben einfachen Untersuchungen des Zustandsraumes insbesondere auf Methoden der Simulation. Die Entfaltungstechnik wird dabei genutzt zur Erzeugung halbgeordneter Abläufe, anhand derer dynamische Eigenschaften in Simulationsläufen visuell dargestellt werden können. In einer Datenbank werden die berechneten Abläufe gespeichert.

Selbst bei der Untersuchung relativ kleiner Systeme verlangt dieses Werkzeug vom Anwender sehr viel Geduld. Durch zahlreiche (teilweise abschaltbare) graphische Animationen dauern die Berechnungen oft unverhältnismäßig lange, obwohl die Implementierung der zeitkritischen Module in der Sprache C vorgenommen worden ist; die Umsetzung der übrigen Teile erfolgte in der interpretierten, objektorientierten Skript–Sprache Python.

Aus dem erzeugten Präfix werden die unverzweigten Ausführungen herausgefiltet und sind einzeln darstellbar, wobei die Knoten mit Hilfe eines (wiederum recht lansamen) automatischen Platzierungsalgorithmus übersichtlich angeordnet werden. Verschiedene Kriterien der Cut-off–Terminierung sind anwählbar, neben der Simulation sind über das Erkennen von Verklemmungen hinaus keine weiter reichenden Verifikationsmöglichkeiten auf dem berechneten Präfix vorhanden.

Modellierung und Analyse von Kommunikationssystemen mittels zeitbewerteter Occurrence–Netze An der Otto-von-Guericke-Universität Magdeburg wird seit 1997 die Entfaltungstechnik am Institut für Automatisierungstechnik im Rahmen der Entwicklung eines Werkzeugs eingesetzt, mit dem vorwiegend Kommunikationssysteme modular auf Basis von Occurrence–Netzen modelliert werden. Ziel ist eine qualitative und quantitative Untersuchung der Systeme.
Mit dem Werkzeug sollen sowohl der funktional korrekte Entwurf als auch die Dimensionierung bei vorgegebener Leistung in einem gesamtheitlichen Ansatz formal exakt und rechnergestützt ausführbar werden. Die Methode unterstützt die Berücksichtigung implementierungsspezifischer Details bei der Modellbildung. Dies eröffnet die Möglichkeit, Vorhersagen über das Verhalten der späteren Implementierung zu treffen bzw. dient bei schon vorhandenen Implementierungen des Kommunikationssystems zur Lokalisierung von Kapazitätsengpässen.
Für die Implementierung des Präfix–Generators wird die Sprache MATLAB verwendet. Interessanterweise hat sich gezeigt, dass für die implementierten Analyseverfahren die Ordnung $\sqsubset_{McM}$ am geeignetsten ist.

Model Checking Kit Konzeptionelle Idee hinter dem *Model Checking Kit* [SSE99] ist die Sammlung und Integration verschiedener, vorwiegend Petri–Netz–basierter Verifikationsverfahren in einem gemeinsamen Werkzeug, ohne dass der Benutzer die Inhomogenität (bedingt durch verschiedene Konzepte, Eingabesprachen und –formate der unterschiedlichen Implementierungen) bemerkt. Auf einfache Weise können so vergleichende Untersuchungen der verschiedenen Analysealgorithmen vorgenommen und die Vorteile eines jeden Verfahrens ausgenutzt werden.
Das Kit verwendet keine graphische Oberfläche, die Steuerung erfolgt durch Aufruf eines einzelnen Programms, zusammen mit der Angabe von Optionen zur Auswahl eines Verfahrens und den dazu benötigten Eingabedateien¸ die das modellierte System und eventuell zu testende Eigenschaften enthalten. Als Eingabesprachen stehen dabei eine spezielle Darstellung kommunizierender Automaten, die Notation B(PN)^2¸ sowie diverse Netzdateiformate der integrierten Werkzeuge zur Verfügung.
Netzentfaltungen werden in dieser Werkzeugsammlung für ein breites Spektrum von Verfahren genutzt: nahezu alle in Kapitel 7 vorgestellten Algorithmen zur Untersuchung von Verklemmungen und Erreichbarkeit sowie der LTL–Model–Checker können ausgewählt werden. Daneben unterstützt das Werkzeug weitere Analyseverfahren, etwa aus PROD [VHHP95] und SMV [SMV98].

Neben diesen teilweise abgeschlossenen oder in Weiterentwicklung befindlichen Werkzeugen arbeitet derzeit eine Gruppe an der Universität Edinburgh an einer eigenen Implementierung des Präfix–Generators. Zusammenfassend ist festzustellen, dass universelle Analyseverfahren, die die Technik der Netzentfaltungen verwenden, erst in relativ wenigen Werkzeugen zur Verfügung gestellt werden – verglichen etwa mit der

Zahl verfügbarer BDD–basierter Werkzeuge. In den nächsten Jahren wird sich ihre Zahl sicher vermehren, da das Interesse an Netzentfaltungen zunehmend wächst und immer effizientere Verifikationsalgorithmen entwickelt werden.
Etwas umfangreicher im Zusammenhang mit der praktischen Anwendung der Entfaltungstechnik ist bis dato die Zahl an Veröffentlichungen über Vergleichsstudien und Benchmarks. Darüber wird das nächste Kapitel berichten.

Kapitel 10

„Die größte Wahrscheinlichkeit der Erfüllung
lässt noch einen Zweifel zu;
daher ist das Gehoffte,
wenn es in der Wirklichkeit eintritt,
jederzeit überraschend.“
JOHANN WOLFGANG VON GOETHE
MAXIMEN UND REFLEXIONEN, 3

Experimentelle Ergebnisse

In den vorangegangenen Kapiteln dieser Arbeit ist die Berechnung eines vollständigen, endlichen Präfixes ausführlich in Theorie und Praxis diskutiert worden, daneben fanden sich Beschreibungen von Anwendungen und die Integration in Werkzeugen. Zum Abschluss sollen nun einige praktische Ergebnisse zeigen, wie sich dieses Verfahren im praktischen Einstz bewährt.
Zahlreiche Benchmarks und Vergleichsstudien sind in den letzten Jahren veröffentlicht worden, die direkt oder indirekt mit Hilfe der in der vorliegenden Arbeit beschriebenen Implementation angestellt wurden, darunter [Röm95, Bes96, HD96, MR97, Stö98, Hel98, Mel98, HDS99, Hel99a, Hel99b, ER99, Wal00]. Die dort präsentierten Ergebnisse und Schlussfolgerungen können und sollen hier nicht alle wiederholt werden.

Beispielsweise enthält [HDS99] die sehr interessante Fallstudie über die Modellierung einer industriell eingesetzten Produktionszelle, bestehend aus sechs Maschineneinheiten zur Bearbeitung von Stahlplatten. Verschiedene Verifikationswerkzeuge, denen unterschiedliche Techniken zugrunde liegen, wurden anhand dieses Modells miteinander verglichen, indem mit den jeweiligen Model–Checkern diverse Eigenschaften untersucht wurden. Der entfaltungsbasierte Branching–Time–Model–Checker schnitt dabei, zusammen mit dem LTL–Model–Checker aus PROD[1] [VHHP95], hervorragend ab gegenüber dem BDD–basierten Werkzeug SMV [SMV98][2].

[1] PROD verwendet die Technik der sturen Mengen zur Reduzierung der Größe des Erreichbarkeitsgraphen.
[2] Die Autoren führen dies teilweise darauf zurück, dass SMV ursprünglich nicht für die Analyse von Netzsystemen entwickelt wurde.

Andere Veröffentlichungen vergleichen die verschiedenen Methoden zum Auffinden von Verklemmungen oder zur Untersuchung von Erreichbarkeitsfragen miteinander (siehe Kapitel 7.1 und 7.2) und geben Kriterien an, unter welchen Umständen welche Methode einer anderen vorzuziehen ist, [MR97, Hel98, Hel99b].

In der vorliegenden Arbeit sind ausführlich Aspekte einer konkreten, effizienten Umsetzung der Präfix–Berechnung beschrieben worden. Aus diesem Grund sollen auch die experimentellen Ergebnisse ein wenig mehr von der technischen Seite das Algorithmus zeigen, insbesondere bezogen auf die bei der DB(PN)^2–Präfix–Generierung verwendeten SND–Ordnungen.

Für die Testreihen sind Beispiele aus [Cor94] verwendet worden, die schon zahlreichen Vergleichstests im Zusammenhang mit Netzentfaltungen als Grundlage dienten. Bei diesen Systemen handelt sich um ursprünglich in ADA codierte Problemstellungen aus verschiedenen Gebieten, die zum überwiegenden Teil in der Größe skalierbar sind. Die nachfolgende Aufstellung gibt einen Überblick: links der Name des modellierten Systems (das n gibt die Skalierung an), rechts eine kurze Erläuterung.

ABP-1	Version des Alternating–Bit–Protokolls (6 Komponenten): zwei Benutzer, je ein Sender und Empfänger sowie zwei fehlerhafte Kanäle.
BDS-1	Ein Grenzverteidigungssystem: 15 Komponenten einfacher Struktur.
CYCLIC-n	Version von Milners *Cyclic Schedulers*: n Scheduler–Prozesse synchronisieren n Verbraucher ($2n$ Komponenten).
DAC-n	Eine einfache *Divide-and-Conquer*–Berechnung: bis zu n Prozesse arbeiten parallel daran ($n+1$ Komponenten).
DP-n	n speisende Philosophen teilen sich n Gabeln: Standard–Version mit Verklemmung ($2n$ Komponenten).
DPD-n	n speisende Philosophen (verklemmungsfrei): ein Token geht unter den Philosophen um; wer es besitzt, kann keine Gabel aufnehmen ($2n$ Komponenten).
DPFM-n	n speisende Philosophen (verklemmungsfrei): es gibt einen zentralen Gabel–Manager; beide Gabeln werden gleichzeitig aufgenommen ($n+1$ Komponenten).
DPH-n	n speisende Philosophen (verklemmungsfrei): ein Butler lässt maximal $n-1$ Philosophen gleichzeitig an den Tisch ($2n$ Komponenten).
ELEVATOR-n	Fahrstuhlsimulation: modelliert das Verhalten von n Fahrstühlen (insgesamt $n+3$ Komponenten) eines Hauses.

FURNACE-n	Verwaltung der Temperaturdaten von n Hochöfen ($2n + 6$ Komponenten).
GAS-NQ-n	Selbstbedienungs–Tankstelle mit n Tanksäulen, ohne Warteschlange ($n + 3$ Komponenten).
GAS-Q-n	Selbstbedienungs–Tankstelle mit n Tanksäulen und einer Warteschlange ($n + 3$ Komponenten).
HARTSTONE-n	Ein einfaches Protokoll: Ein Prozess startet und beendet n Arbeitsprozesse ($n + 1$ Komponenten).
KEY-n	Modellierung der Kommunikation zwischen einer Tastatur und einem Fenstermanager ($n + 5$ Komponenten).
MMGT-n	Verteilter Speichermanager für n Benutzer ($n + 4$ Komponenten).
OVER-n	Modell einer Autobahn: Verwaltung von n überholenden Autos ($2n + 1$ Komponenten).
RING-n	Token–Ring–Protokoll mit gegenseitigem Ausschluss für die beteiligten n Prozesse ($2n$ Komponenten).
RW-n	Datenbank, auf die n lesende und n schreibende Prozesse Zugriff haben ($2n + 1$ Komponenten).

Die ADA–Programme sind in mehrere andere Sprachen und Formate übersetzt worden, darunter eine Darstellung in Form sequenzieller, kommunizierender Automaten. Diese Automaten lassen sich auch als SND–Systeme auffasssen. In der obigen Liste ist jeweils die Zahl der Komponenten (Automaten) eines Systems mit angegeben.

In den Tabellen 10.1 und 10.2 wird die Konstruktion von Präfixen bei Verwendung der Ordnung $\sqsubset_{McM}$ bzw. $\sqsubset_t$ gegenübergestellt. Alle Experimente wurden auf derselben Maschine durchgeführt, einer SUN Ultra 60 mit 2 Prozessoren (je 295 MHz) und 1,5 GB RAM unter Solaris 2.7. Die Zeitangaben erfolgen durchgehend in Sekunden. Bei der Netzgröße der Präfixe ist zu beachten, dass keine Bedingungen hinter Cut-off–Ereignissen erzeugt werden.

Spalten 1–4 geben Auskunft über die Systemgröße (Anzahl Stellen und Transitionen) und die Zahl der Komponenten. Spalte 4 enthält die Größen der Zustandsräume; allgemein sind sie bei diesen Beispielen recht moderat. Systeme mit weit größeren Zustandsräumen werden z. B. in [MR97, Mel98] betrachtet.
In den Spalten 5–8 wird das mit der McMillan–Ordnung $\sqsubset_{McM}$ generierte Präfix charakterisiert: Anzahl Bedingungen (5) und Ereignisse (6), davon Cut-off (7). In Spalte 8 wird schließlich die für die Berechnung benötigte Zeit angegeben. Mit einem Strich „–“versehene Einträge konnten nicht ermittelt werden (Speicherüberlauf oder Abbruch nach mehrstündiger Rechnung).
Zum Vergleich dazu sind in Tabelle 10.2 die Werte bei Verwendung der Ordnung $\sqsubset_t$ aufgetragen (Spalten 9–12).

	Ausgangssystem				Präfix mit Ordnung $\sqsubset_{McM}$			
	$\|S\|$	$\|T\|$	Kompon.	Zustände	$\|B\|$	$\|E\|$	Cut-off	Zeit
Problem	1	2	3	4	5	6	7	8
ABP-1	43	95	6	113	345	259	88	0.03
BDS-1	53	59	15	36097	23991	35692	23590	277.53
CYCLIC-3	23	17	6	43	78	44	8	0.00
CYCLIC-6	47	35	12	639	606	361	64	0.06
CYCLIC-9	71	53	18	7423	4662	2834	512	2.93
CYCLIC-12	95	71	24	74264	36942	22555	4096	444.34
DAC-6	42	34	7	222	92	53	0	0.01
DAC-9	63	52	10	1790	167	95	0	0.01
DAC-12	84	70	13	14334	260	146	0	0.01
DAC-15	105	88	16	114686	371	206	0	0.03
DP-6	36	24	12	729	144	96	30	0.01
DP-8	48	32	16	6555	256	176	56	0.02
DP-10	60	40	20	48897	400	280	90	0.03
DPD-4	36	36	8	601	432	296	81	0.04
DPD-5	45	45	10	3489	1160	790	211	0.18
DPD-6	54	54	12	19861	2788	1892	499	0.94
DPD-7	63	63	14	109965	6372	4314	1129	6.07
DPFM-2	7	5	3	5	8	5	2	0.01
DPFM-5	27	41	6	13	37	41	25	0.01
DPFM-8	87	321	9	49	394	761	568	0.06
DPH-4	39	46	9	513	1786	1341	452	0.41
DPH-5	48	67	11	3113	121662	112461	51635	9223.93
DPH-6	57	92	13	16897	—	—	—	—
DPH-7	66	121	15	79927	—	—	—	—
ELEVATOR-1	63	99	4	158	303	263	100	0.02
ELEVATOR-2	146	299	5	1062	4600	4118	1632	3.22
ELEVATOR-3	327	783	6	7121	94957	85925	34562	4473.11
ELEVATOR-4	736	1939	7	43440	—	—	—	—
FURNACE-1	27	37	8	344	365	486	300	0.04
FURNACE-2	40	65	10	3778	11590	19020	13120	45.47
GAS-NQ-2	71	85	5	193	874	685	248	0.11
GAS-NQ-3	143	223	6	1770	48913	48805	24348	849.86
GAS-Q-1	28	21	4	19	39	23	4	0.00
GAS-Q-2	78	97	5	181	650	533	208	0.06
GAS-Q-3	284	475	6	1705	25933	27437	14460	218.75
HARTSTONE-75	377	227	76	153	527	302	1	0.08
HARTSTONE-100	502	302	101	203	702	402	1	0.16
MMGT-1	50	58	5	73	81	60	21	0.00
MMGT-2	86	114	6	817	1504	1305	554	0.33
MMGT-3	122	172	7	7703	45069	41551	19017	732.09
OVER-2	33	32	5	65	67	42	10	0.01
OVER-3	52	53	7	519	472	319	85	0.04
OVER-4	71	74	9	4175	2373	1729	545	0.73
OVER-5	90	95	11	33460	13596	10776	3981	45.93
RING-3	39	33	6	87	89	56	13	0.01
RING-5	65	55	10	1290	413	265	61	0.03
RING-7	91	77	14	17000	1307	819	169	0.22
RING-9	117	99	18	211528	3259	1986	361	1.25
RW-6	33	85	13	72	3938	11755	9792	5.98
RW-9	48	181	19	523	—	—	—	—
RW-12	63	313	25	4110	—	—	—	—

Tabelle 10.1: Ausgangsnetze und McMillan–Präfix der untersuchten Systeme.

	Präfix, mit Ordnung $\sqsubset_t$ erzeugt						
Problem	$\|B\|$ 9	$\|E\|$ 10	Cut-off 11	Zeit 12	WSmax 13	Vgl_{WS} 14	Gleichheit 15
ABP-1	225	167	56	0.01	28	194	–
BDS-1	5135	6330	3701	1.86	659	673247	680
CYCLIC-3	44	23	4	0.00	4	18	–
CYCLIC-6	98	50	7	0.01	4	42	–
CYCLIC-9	152	77	10	0.00	4	66	–
CYCLIC-12	206	104	13	0.01	4	90	–
DAC-6	92	53	0	0.00	8	92	–
DAC-9	167	95	0	0.00	11	197	–
DAC-12	260	146	0	0.01	14	338	–
DAC-15	371	206	0	0.02	17	515	–
DP-6	144	96	30	0.01	12	193	–
DP-8	256	176	56	0.01	16	418	–
DP-10	400	280	90	0.02	20	763	–
DPD-4	432	296	81	0.02	46	1353	–
DPD-5	1160	790	211	0.09	97	7356	–
DPD-6	2788	1892	499	0.37	201	35631	–
DPD-7	6372	4314	1129	1.65	405	161576	–
DPFM-2	8	5	2	0.00	2	3	–
DPFM-5	27	31	20	0.00	16	47	–
DPFM-8	102	209	162	0.01	91	1135	–
DPH-4	446	336	117	0.02	42	1500	22
DPH-5	1618	1351	547	0.17	107	21455	85
DPH-6	7776	7289	3407	3.11	374	429186	359
DPH-7	36144	37272	19207	101.44	1633	9749342	1389
ELEVATOR-1	181	157	59	0.01	35	544	–
ELEVATOR-2	920	827	331	0.07	132	7219	–
ELEVATOR-3	4235	3895	1629	0.97	503	115545	–
ELEVATOR-4	18070	16935	7337	23.96	2270	1863447	–
FURNACE-1	270	326	189	0.02	54	2803	5
FURNACE-2	1977	2767	1750	0.31	440	162667	60
GAS-NQ-2	246	169	46	0.01	20	382	–
GAS-NQ-3	1607	1205	401	0.17	128	13530	12
GAS-Q-1	35	21	4	0.00	4	20	–
GAS-Q-2	238	173	54	0.01	28	446	–
GAS-Q-3	1613	1297	490	0.20	148	19272	–
HARTSTONE-75	527	302	1	0.10	76	150	–
HARTSTONE-10	702	402	1	0.16	101	200	–
MMGT-1	79	58	20	0.00	11	59	–
MMGT-2	772	645	260	0.06	111	2592	13
MMGT-3	6625	5841	2529	2.61	779	159588	309
OVER-2	65	41	10	0.00	6	64	–
OVER-3	272	187	53	0.02	27	957	10
OVER-4	1097	783	237	0.11	84	13073	54
OVER-5	4936	3697	1232	1.28	363	327095	339
RING-3	75	47	11	0.00	6	70	–
RING-5	265	167	37	0.01	14	548	–
RING-7	655	403	79	0.05	27	2483	–
RING-9	1325	795	137	0.13	42	7961	–
RW-6	152	397	327	0.01	128	2103	–
RW-9	1060	4627	4106	1.31	1280	186097	–
RW-12	8240	49177	45069	168.77	11386	4644703	–

Tabelle 10.2: Mit der Ordnung $\sqsubset_t$ generierte Präfixe der Beispielsysteme.

Deutlich zu erkennen ist die Erzeugung eines kleineren Präfixes bei den Systemen CYCLIC-*n* und GAS-NQ-*n* infolge der Verwendung von $\sqsubset_t$. Viele andere Systeme profitieren ebenfalls, zudem können mit $\sqsubset_t$ sogar die Systeme entfaltet werden, bei denen die Ordnung $\sqsubset$ kein Ergebnis lieferte. In einigen Fällen trägt die totale Ordnung jedoch zu keiner Verbesserung bei, z. B. für DPD-*n*, HARTSTONE-*n*; in solchen Fällen sind bereits die Größen der lokalen Konfigurationen vergleichbar.

In der Tabelle sind außerdem die maximale Zahl von Ereignissen in der Warteschlange (kurz WS) angegeben (13), sowie die Anzahl der durchgeführten Vergleiche (14), die für das Einsortieren erweiternder Ereignisse in die Warteschlange getätigt wurden. Spalte 15 informiert schließlich über die Zahl der erfolglosen Vergleiche, nach denen die teurere Berechnung entlang der Foata–Normalform eingeleitet werden muss. Die teuren Vergleiche kommen nicht bei allen Beispielen vor, insgesamt ist ihr Anteil sogar verhältnismäßig gering.

Die Tabellen 10.3 bis 10.5 zeigen auf, wie die Verwendung von SND–Ordnungen die Präfix–Generierung für die gleichen Netzsysteme beeinflusst. In diesen Tabellen werden bei den skalierbaren Systemen nur die jeweils größten Vertreter betrachtet.
Über die in Kapitel 3.5.3 definierten Ordnungen $\sqsubset_{SND_1}$ (am Rand der Spalten ganz links abgekürzt durch „1“) und $\sqsubset_{SND_2}$ (kurz „2“) hinaus sind dabei einige zusätzliche, ebenfalls adäquate Ordnungen in die Untersuchungen einbezogen worden.
Die Zeichen „<“ bzw. „>“ symbolisieren, ob die verwendete totale Ordnung auf den Transitionen des Systems auf– („<“) oder absteigned („>“) sortiert. Weiterhin bezeichnet ein führendes „S“, dass bei diesen Ordnungen zunächst die Summen über den *i*–Tiefen verglichen werden. Kann der Vergleich damit nicht entschieden werden, wird mit der standardmäßigen Definition der entsprechenden SND_i–Ordnung (1 oder 2, in Abhängigkeit von „<“ bzw. „>“ auf– bzw. absteigend) fortgefahren. Durch diese Unterteilung können acht SND–Ordnungen unterschieden werden.
In den Tabellen sind die Spalten 16 bis 20 wie zuvor zu interpretieren; die Angabe der Zahl der Bedingungen ist diesmal aus Platzgründen weggelassen. Neu sind die Spalten 21, 22 und 23: $\mathrm{Vgl}_{\mathrm{Silex}}$ gibt die Zahl der komponentenbezogenen Vergleiche an, die im Rahmen der Silex–Ordnung durchgeführt werden. Dieser Wert muss also stets größer oder gleich dem Wert in Spalte 20 sein.
Spalte 22 enthält diejenige Zahl von Vergleichen von (21), die (im Allgemeinen) die aufwändigeren sind, da aufgrund einer Unvergleichbarkeit von Größen eine Tiefensuche zum Sequenzvergleich eingeleitet werden muss. Die Spalte 23 gibt innerhalb dieser teureren Vergleiche (22) wiederum die ganz billigen an, die daraus resultieren, dass diese unvergleichbare Größe und damit die Länge der zu vergleichenden Sequenzen null ist. Wie die ermittelten Werte in den Tabellen zeigen, kommt dieser Fall relativ häufig vor, insbesondere in der Anfangsphase der Präfix–Generierung, kostet aber kaum Zeit.

Problem		Präfix, mit Ordnung $\sqsubset_{SND_i}$ erzeugt							
		$\|E\|$	Cut-off	Zeit	maxQu	Vgl_{WS}	Vgl_{Silex}	Vgl_{Tief}	Vgl_{Null}
		16	17	18	19	20	21	22	23
ABP-1	1<	167	56	0.01	16	730	1559	1216	149
	2<	167	56	0.01	24	466	1676	653	87
	1>	167	56	0.01	16	1404	1727	429	64
	2>	167	56	0.01	16	1404	1912	196	19
	S1<	167	56	0.01	28	266	801	730	107
	S2<	167	56	0.01	28	290	1256	653	87
	S1>	167	56	0.01	24	234	730	694	96
	S2>	167	56	0.01	24	322	1435	653	87
BDS-1	1<	6929	3562	4.49	373	1109566	2539533	1748468	283906
	2<	6678	3167	2.73	426	1270042	3881397	83207	14103
	1>	9933	6457	2.26	101	377688	570958	208819	72967
	2>	9445	6176	1.89	103	354982	634090	27946	4464
	S1<	6691	3092	3.38	704	813120	372132	224398	38180
	S2<	6691	3092	3.22	712	830520	635165	82452	13355
	S1>	6691	3092	3.44	720	767037	335872	215255	36412
	S2>	6735	3112	3.40	692	771315	640920	77890	12722
CYCLIC-12	1<	104	13	0.01	5	216	700	507	55
	2<	104	13	0.01	5	216	921	78	0
	1>	93	13	0.01	23	1002	2542	1540	660
	2>	93	13	0.00	23	1002	2542	0	0
	S1<	104	13	0.01	4	93	344	298	55
	S2<	104	13	0.01	4	93	554	78	0
	S1>	104	13	0.01	4	90	277	243	55
	S2>	104	13	0.01	4	90	487	78	0
DAC-15	1<	206	0	0.01	17	1466	7488	6064	6022
	2<	206	0	0.01	18	1479	7829	223	196
	1>	206	0	0.02	17	1333	6122	4817	4789
	2>	206	0	0.01	17	1524	7489	119	105
	S1<	206	0	0.01	17	526	1506	1271	1244
	S2<	206	0	0.02	17	526	1715	223	196
	S1>	206	0	0.02	17	404	1193	1016	989
	S2>	206	0	0.02	17	404	1402	223	196
DP-10	1<	280	90	0.03	16	1617	4743	3153	1289
	2<	280	90	0.02	16	1816	5185	0	0
	1>	280	90	0.02	10	1456	4610	3154	2034
	2>	280	90	0.02	10	1456	4610	0	0
	S1<	280	90	0.02	20	799	982	662	372
	S2<	280	90	0.02	20	799	992	0	0
	S1>	280	90	0.02	20	765	857	578	288
	S2>	280	90	0.02	20	763	867	0	0
DPD-7	1<	4314	1129	1.76	191	334623	436189	213346	5551
	2<	4314	1129	1.49	226	303966	655930	30056	1182
	1>	4314	1129	1.32	23	56863	77896	23162	5009
	2>	4314	1129	1.30	23	56863	90465	3343	262
	S1<	4314	1129	1.65	415	176820	87222	42843	2230
	S2<	4314	1129	1.64	429	176862	200034	29153	1236
	S1>	4314	1129	1.63	403	161824	74552	41690	2127
	S2>	4314	1129	1.62	426	162883	199004	29153	1236

Tabelle 10.3: Mit diversen SND–Ordnungen generierte Präfixe der Beispielsysteme (1).

Problem		Präfix, mit Ordnung $\sqsubset_{SND_i}$ erzeugt: $\|E\|$ 16	Cut-off 17	Zeit 18	maxQu 19	Vgl_{WS} 20	Vgl_{Silex} 21	Vgl_{Tief} 22	Vgl_{Null} 23
DPFM-8	1<	209	162	0.01	51	4864	9910	6652	4449
	2<	209	162	0.02	40	3889	10186	217	116
	1>	209	162	0.01	101	11133	13027	1894	125
	2>	209	162	0.01	101	11133	13027	0	0
	S1<	209	162	0.01	94	1858	4153	2815	2012
	S2<	209	162	0.02	90	1659	5017	217	116
	S1>	209	162	0.01	91	1130	2582	1634	1241
	S2>	209	162	0.01	91	1226	3683	181	100
DPH-7	1<	19562	9925	35.94	1051	8655223	13806556	7923839	1046744
	2<	19306	9693	23.16	845	6847778	12896247	28548	5380
	1>	46483	26577	188.16	1515	35192066	36250643	1075894	55
	2>	45755	26092	58.65	2046	54062576	54777017	0	0
	S1<	19591	9790	38.01	1549	5538224	2174782	718668	137392
	S2<	19619	9817	35.38	1538	5515129	2265160	28537	5289
	S1>	42224	22757	144.57	2106	14896777	4901616	1229033	171604
	S2>	42902	23206	146.13	2128	15398260	5644680	39342	5300
ELEVATOR-4	1<	16935	7337	31.84	1028	4429764	7544996	4805614	689
	2<	16935	7337	25.60	1672	6405171	14462097	158828	232
	1>	16935	7337	77.20	2538	22602490	23805673	1222568	3665
	2>	16935	7337	44.69	2538	22602490	23840817	25504	920
	S1<	16935	7337	24.64	2188	1787550	1446442	598903	340
	S2<	16935	7337	23.65	2196	1894430	2290874	149748	232
	S1>	16935	7337	24.72	2186	1692752	1335768	521052	977
	S2>	16935	7337	23.72	2186	1831264	2087463	154683	944
FURNACE-2	1<	1934	1314	0.42	89	75127	229138	171347	38711
	2<	1992	1287	0.20	129	78746	227223	30354	6348
	1>	2787	1771	0.84	219	292740	418679	129503	27655
	2>	2769	1774	0.36	230	302895	439072	11379	1225
	S1<	1859	1194	0.20	212	74042	47312	37601	8770
	S2<	1859	1194	0.20	212	74074	58877	28182	6072
	S1>	1871	1206	0.19	212	79353	39524	31485	7007
	S2>	1871	1206	0.20	212	79517	51933	28257	6126
GAS-NQ-3	1<	1193	401	0.16	80	44726	56638	28721	7084
	2<	1193	401	0.14	88	43521	77440	3969	891
	1>	1239	401	0.14	118	70479	75719	5440	2845
	2>	1239	401	0.13	117	70191	75785	623	87
	S1<	1193	401	0.15	122	15617	9544	6140	1686
	S2<	1193	401	0.14	116	16811	19204	3937	891
	S1>	1233	401	0.15	128	13909	7779	4817	729
	S2>	1233	401	0.15	122	14366	14769	2864	487
GAS-Q-3	1<	1277	490	0.18	91	53622	67357	36838	8452
	2<	1281	490	0.15	95	55332	102821	4262	661
	1>	1301	490	0.14	90	60055	65044	5429	2857
	2>	1303	490	0.12	86	57391	63244	1029	111
	S1<	1281	490	0.16	158	22508	13049	7604	2292
	S2<	1281	490	0.16	158	24760	25203	4227	660
	S1>	1289	490	0.17	160	21452	11057	6169	854
	S2>	1289	490	0.17	158	22987	25063	4376	570

Tabelle 10.4: Mit diversen SND–Ordnungen generierte Präfixe der Beispielsysteme (2).

Problem		Präfix, mit Ordnung $\sqsubset_{SND_i}$ erzeugt: $\|E\|$ 16	Cut-off 17	Zeit 18	maxQu 19	Vgl_{WS} 20	Vgl_{Silex} 21	Vgl_{Tief} 22	Vgl_{Null} 23
	1<	402	1	0.11	101	10200	348351	338252	333300
	2<	402	1	0.07	101	10200	348551	1	0
	1>	402	1	0.46	101	10100	338450	328351	4950
HART–	2>	402	1	0.07	101	10100	338550	1	0
STONE–	S1<	402	1	0.08	101	5151	171701	166651	166650
100	S2<	402	1	0.06	101	5151	171801	1	0
	S1>	402	1	0.05	101	200	5051	4951	4950
	S2>	402	1	0.05	101	200	5151	1	0
	1<	5841	2529	2.92	396	684675	789751	432127	9780
	2<	5841	2529	2.21	610	974938	1505107	33411	1371
	1>	5770	2497	2.96	462	899866	942573	46003	19528
MMGT-3	2>	5770	2497	2.01	462	899866	954755	4329	354
	S1<	5841	2529	2.56	775	185435	112206	59644	2414
	S2<	5841	2529	2.53	791	205732	291607	33728	1371
	S1>	5819	2521	2.38	791	193607	133954	51878	2657
	S2>	5819	2521	2.30	759	210343	314414	31264	1299
	1<	3755	1285	1.30	120	149117	355939	227212	13705
	2<	3755	1285	0.91	161	194188	507754	22024	2509
	1>	5077	1714	1.73	103	196573	235924	45344	4437
OVER-5	2>	5077	1714	1.62	103	196847	245105	5564	611
	S1<	3703	1271	1.21	407	382650	82962	50651	4438
	S2<	3703	1271	1.16	407	384203	109591	21752	2511
	S1>	3701	1269	1.20	410	380207	72972	44923	3790
	S2>	3701	1269	1.15	410	381246	98520	20783	2312
	1<	507	142	0.09	37	6769	13462	11851	385
	2<	712	137	0.08	45	9698	26545	0	0
	1>	648	141	0.08	46	15533	18842	3600	238
RING-9	2>	796	137	0.08	74	21924	23451	0	0
	S1<	795	137	0.10	40	7780	3616	2338	147
	S2<	712	137	0.08	41	6890	3997	0	0
	S1>	796	137	0.10	42	7823	3412	2176	91
	S2>	796	137	0.09	42	7859	4753	0	0
	1<	49177	45069	304.57	8178	169682137	305914583	221853976	68101629
	2<	49177	45069	83.44	4106	119782284	283765468	292864	112640
	1>	49177	45069	1083.46	40986	1007316926	1007879410	562484	3246
RW-12	2>	49177	45069	508.76	40986	1007316926	1007879410	0	0
	S1<	49177	45069	63.82	12374	28295892	53721768	38806675	12697635
	S2<	49177	45069	31.87	11588	21887880	56868758	292864	112640
	S1>	49177	45069	34.53	11386	14644703	27632971	12992363	6477940
	S2>	49177	45069	32.60	11252	21549773	54337831	114687	47103

Tabelle 10.5: Mit diversen SND–Ordnungen generierte Präfixe der Beispielsysteme (3).

Ein Studium der Tabellen macht deutlich, dass Silex–Ordnung $\sqsubset_{SND_2}$ in allen Fällen zu einer schnelleren (oder wenigstens etwa gleich schnellen) Berechnung führt als $\sqsubset_{SND_1}$. Besonders schön zeigt sich das in den Beispielen DPH-7 und RW-12. Dieses Ergebnis deckt sich mit der Theorie, da die Ordnung $\sqsubset_{SND_2}$ zunächst mit den Größen der lokalen Sichten arbeitet, bevor ein Vergleich einzelner Elemente vorgenommen wird.
Bei der Frage nach auf– oder absteigender Sortierung der Transitionen lässt sich keine klare Präferenz aussprechen. In der überwiegenden Zahl der Systeme scheint die aufsteigende Ordnung günstiger zu sein; gute Beispiele hierfür sind wieder DPH-7 und RW-12, wohingegen KEY-3, OVER-5 und ELEVATOR-4 sich gerade anderherum verhalten. Hierbei ist sicher von Bedeutung, in welcher Reihenfolge die Knoten im übergebenen Netzsystem angeordnet sind.
Der vorgeschaltete Summenvergleich liefert in fast allen Beispielen eine Verbesserung der Laufzeit; lediglich die Beispiele BDS-1 und DPD-7 sind hier Ausreißer. Auch dieses Ergebnis ist theoretisch nachvollziehbar.

Nach diesen Beispielen scheint also „S2<“, d. h. Silex–Ordnung $\sqsubset_{SND_2}$ bei aufsteigender Sortierung der Transitionen und vorherigem Summenvergleich, eine günstige Ordnung für die Präfix–Generierung von SND–Systemen zu sein. In fast allen Fällen erlaubt diese Ordnung ebenfalls eine schnellere Berechnung als die Verwendung von $\sqsubset_t$; lediglich das Beispiel BDS-1 benötigt mit dieser Ordnung mehr Rechenzeit.

Kapitel 11

„Sed haec hactenus!“
(Für diesmal genug!)
CICERO, BUCH DER PFLICHTEN, 1, 39, 140

Zusammenfassung und Ausblick

In dieser Arbeit ist die Technik der Netzentfaltungen diskutiert worden als eine Möglichkeit, das Problem der Zustandsraum–Explosion bei der Untersuchung verteilter, reaktiver Systeme in den Griff zu bekommen. Ein vollständiges, endliches Präfix der maximalen Entfaltung enthält alle erreichbaren Markierungen des Systems, zudem liefert es Informationen über Nebenläufigkeit, Konflikte sowie kausale Abhängigkeiten.

Theoretische und praktische Ansätze sind vorgestellt worden, die Größe des Präfixes zu minimieren sowie seine Konstruktion effizient zu implementieren, so dass es mit Hilfe der aktuell verfügbaren Rechnerarchitekturen möglich ist, Präfixe von Systemen mit praktisch relevanter Problemstellung in überschaubarer Zeit zu berechnen. Dies gestattet eine effiziente automatische Verifikation des Ausgangssystems mittels verschiedener Verifikationsverfahren, insbesondere dem Model–Checking.
An mehreren Stellen der Arbeit sind über die bereits realisierten Verbesserungen hinaus weitergehende Modifikationen aufgezeigt worden. Diese umzusetzen wird Teil zukünftiger Untersuchungen sein, daneben wird auch die Integration spezieller Techniken in die Präfix–Generierung zu überprüfen sein, wie etwa das Ausnutzen von Symmetrien.

Um dem Anwender einen einfacheren und übersichtlicheren Zugang zu diesen Methoden zu ermöglichen, als es das direkte Modellieren von Systemen in Form von Petri–Netzen gestattet, ist eine einfache Programm–Notation, DB(PN)^2, vorgestellt worden. Gegenüber den direkt vergleichbaren Sprachen besitzt sie erweiterte Datenstrukturen und erlaubt einen lockereren Umgang mit den Definitionsbereichen von Variablen. Auch an anderen Stellen ist versucht worden, den Benutzer von der Last zu befreien, alle Einzelheiten der zugrunde liegenden Netzsemantik bei der Modellierung von Systemen kennen zu müssen. Durch das Zulassen von Sprüngen können darüber hinaus

beliebige Automatenstrukturen in DB(PN)^2 abgebildet werden. Die Netzsemantik dieser Sprache ist ausführlich erörtert worden, eine einfache Anpassung auch an andere Hochsprachen ist somit leicht möglich.
Bereits jetzt sind zahlreiche Erweiterungen der Sprache DB(PN)^2 geplant, die im Rahmen dieser Arbeit noch nicht umgesetzt werden konnten: um die Sprache mächtiger zu gestalten sollen beispielsweise Mehrfach–Zuweisungen der Art $(x, y) := (1, 2)$ unterstützt werden, außerdem sollen zusätzliche Datenstrukturen wie etwa Listen nachgerüstet werden.

Die vorgestellte, auf dem Box–Kalkül aufsetzende Netzsemantik ist so gewählt worden, dass die anschließende Erzeugung eines vollständigen Präfixes der Netzentfaltung möglichst einfach durchgeführt werden kann. Zentraler Gesichtspunkt dabei ist die Trennung des Kontroll– und Datenflusses von Programmen, beide werden erst bei der Erzeugung der entsprechenden Ereignisknoten im Präfix wieder zusammengeführt.
Andere Design–Aspekte sind eine möglichst minimale Größe der Ausgangsnetze, außerdem sind eine Reihe nicht unbedingt notwendiger Nicht–Determinismen vermieden worden, um die Konstruktion des Präfixes effizient vornehmen zu können. Die Frage der Effizienz war schließlich auch ein ausschlaggebender Grund, die Netzsemantik auf SND–Systeme einzuschränken. Ist dieser Punkt nicht so wichtig (etwa bei weiter steigenden Rechnerkapazitäten und Prozessortakten), kann die in dieser Arbeit beschriebene Übersetzung einer höheren parallelen Sprache auf einfache Weise verallgemeinert werden für den Fall 1–sicherer Petri–Boxen, die nicht notwendig SND–Systeme sein müssen. Damit könnten dann auch Systeme mit beliebig verschachtelter Parallelität modelliert werden, was im hier vorgestellten Ansatz nicht möglich ist.

Auf praktischer Seite ist eine effiziente Umsetzung des Präfix–Generators anhand zahlreicher Details einer konkreten Implementierung beschrieben worden. Die vorgestellte universelle Datenstruktur für Petri–Netze bewährt sich unterdessen hervorragend in Implementierungen, die den Netzgraphen wiederholt in verschiedenen Richtungen abschreiten oder zahlreiche Manipulationen an den Netzgraphen vornehmen, also nicht den konstruktiven Charakter der Präfix–Generierung aufweisen.
Der Präfix–Generator selbst ist mittlerweile Bestandteil diverser Werkzeuge zur Analyse verteilter Systeme. Zahlreiche Halbordnungs–basierte Verifikationsmethoden sind implementiert worden, die die Ausgabe(n) dieses Programms weiterverwenden, darunter verschiedene Ansätze zur Überprüfung erreichbarer Markierungen, zum Aufdecken von Verklemmungen, sowie effiziente Model–Checker verschiedener temporaler Logiken.

In etlichen Vergleichsstudien sind inzwischen Praktikabilität und Effizienz der Entfaltungsmethode nachgewiesen worden. Sie beweist gerade bei Systemen mit einem hohen Grad an Parallelität ihre Stärken, ein Feld, in dem der BDD–Ansatz oft in die Knie

geht. Anders herum lohnt sich der Einsatz von Netzentfaltungen bei stark sequenziellen Systemen kaum, da infolge der fehlenden Nebenläufigkeit nahezu alle erreichbaren Markierungen durch eine entsprechende *lokale* Konfiguration induziert werden müssen und somit das Präfix beinahe die Größe des Erreichbarkeitsgraphen annimmt. Bei sequenziellen Systemen zeigen wiederum BDDs Vorteile. In dieser Hinsicht ergänzen sich die beiden Ansätze geradezu.

In welche Richtungen wird es weitergehen? Ursprünglich von McMillan für die Beschreibung des Verhaltens beschränkter Stellen / Transitions–Netze definiert, sind seitdem Übertragungen der Entfaltungstechnik auf verschiedene andere Modelle vorgenommen worden: Petri–Netze mit Lesekanten [VSY98] und spezielle Zeit–Netze [BF99], aber auch Netz–freie Modelle wie kommunizierende Automaten [ER99] oder Prozess–Algebren [LB99]. Hier werden die Forschungen in den nächsten Jahren sicherlich Übertragungen auf weitere Modelle hervorbringen.
Die Erweiterung des Box–Kalküls um die Komponente Zeit ist in den letzten Jahren von mehreren Autoren in Angriff genommen worden [Lil98, BF99]. Für die Präfix–Generierung leiten sich hieraus unmittelbar Änderungen ab, eine Implementierung dieser Ideen im Rahmen des Werkzeugs PEP ist bereits geplant. Ob sich dieser Ansatz auch für eine Übertragung auf DB$(PN)^2$ lohnt, muss erst noch untersucht werden.
Auf Verifikationsseite sind bereits weitere Verbesserungen im Bereich der LTL–Model–Checker abzusehen, etwa eine Implementierung der Ideen aus [HNW99] oder der Ansatz aus [EH00], der auf der Definition einer neuen adäquaten Ordnung $\sqsubset$ beruht.

Anhang A

„That is the true beginning of our end.“
(Das ist der Anfang vom Ende.)
WILLIAM SHAKESPEARE
A MIDSUMMER NIGHT'S DREAM, V, 1, PROLOGUE

Programm–Dokumentation

Der Anhang fasst kurz technische Einzelheiten der wichtigsten Elemente von einigen der implementierten Module zusammen. Dazu gehört zum einen die vollständige Syntax der Eingabesprache DB(PN)^2, wie sie der Parser akzeptiert, gefolgt von einer Beschreibung zusätzlicher Sprachelemente, die im theoretischen Teil der Arbeit nicht behandelt worden sind.

Der Haupt–Algorithmus zur Präfix–Generierung steht in einem separaten Programm namens `ERVunfold` in mehreren Werkzeugen zur Verfügung [PEP98, CPN99, SSE99]. Ein eigener Abschnitt widmet sich den zur Verfügung stehenden Programmoptionen zur Beeinflussung des konstruierten Präfixes, die in weiten Teilen ebenfalls für den DB(PN)^2–Präfix–Generator `dbpn2unf` Gültigkeit haben.

A.1 Sprachumfang von DB(PN)^2

Die folgende Grammatik beschreibt die maschinenlesbare Syntax der Sprache DB(PN)^2, wie sie vom Programm `dbpn2unf` erkannt wird. Syntaktisch fehlerhafte Programmtexte werden von `dbpn2unf` unter Angabe entsprechender Fehlermeldungen zurückgewiesen.

Die Bedeutung der grundlegenden Sprachelemente (Kontrollfluss, Deklarationsteil und atomare Aktionen) ist in Kapitel 5 ausführlich erläutert worden und wird hier nicht wiederholt.

program	::=	*decl* ; *par-block* | *par-block*
par-block	::=	*proc-name command* | *proc-name command* **||** *par-block*
proc-name	::=	*identifier* **::** | ϵ
command	::=	**begin** *command* **end** | *action* | **skip** | **abort** | *jump*
	|	*command* ; *command* | **if** *alt-set* **fi** | **do** *alt-set* **od**
alt-set	::=	*action* | *action jump* | *action* ; *command* | *alt-set* **[]** *alt-set*
action	::=	< *expr* > | < *label* : *expr* >
jump	::=	**goto** *label* | **break** | **continue**
label	::=	*identifier*
decl	::=	**var** *ident-list* : *type-set init*
	|	**var** *ident-list* : **array** *number* **of** *type-set init*
	|	**var** *ident-list* : **chan** *capacity* **of** *type-set*
	|	**var** *ident-list* : **stack** *capacity* **of** *type-set*
	|	**const** *ident-list* = *constant*
	|	*decl* ; *decl*
ident-list	::=	*identifier* | *ident-list* , *identifier*
type-set	::=	*type* | *type* **x** *type*
capacity	::=	*number* | **omega**
type	::=	*dim-range* | { *range* } | **card** | **int** | **bool**
range	::=	*sub-range* | *range* , *sub-range*
sub-range	::=	*constant* | *constant* .. *constant*
init	::=	**init** *constant* | **init (** *const-list* **)** | **init [** *const-list* **]** | ϵ
const-list	::=	*constant* | *const-list* , *constant*
expr	::=	*bool-expr* | *assignment* | *expr* , *expr*
bool-expr	::=	**(** *bool-expr* **)** | *bool-simple* | *bool-expr bool-op bool-expr*
	|	**not** *bool-expr* | *num-expr comp-op num-expr*
assignment	::=	*identifier index* **:=** *rvalue* | *identifier* **!** *rvalue*
rvalue	::=	*value* | **(** *value* **,** *value* **)**
value	::=	*bool-expr* | *num-expr*
comp-op	::=	**=** | **#** | **<** | **>** | **<=** | **>=**
bool-op	::=	**and** | **or** | **=** | **#**
num-expr	::=	**(** *num-expr* **)** | *num-simple* | *num-expr num-op num-expr*
	|	**–** *num-expr*
num-op	::=	**+** | **–** | $*$ | **/** | **mod**
bool-simple	::=	*identifier index* | *identifier* **?** | *identifier* **??** | *identifier* **!!**
	|	**true** | **false**
num-simple	::=	*identifier index* | *number*
index	::=	**[** *num-expr* **]** | **(** *num-expr* **)** | ϵ
identifier	::=	$[\mathbf{a}-\mathbf{zA}-\mathbf{Z\_}][\mathbf{a}-\mathbf{zA}-\mathbf{Z0}-\mathbf{9\_}]^*$
constant	::=	*number* | **–** *number*
number	::=	$[\mathbf{0}-\mathbf{9}]^+$

Zwischen den Terminalsymbolen können an beliebiger Stelle und in beliebiger Zahl Leerzeichen, Tabulatoren, Zeilenenden sowie Kommentare zur Gliederung und Erhöhung der Lesbarkeit eingefügt werden. Kommentare werden wie in den Sprachen C bzw. C++ notiert: zeilenorientierte Kommentare beginnen mit **//** (zwei Schrägstriche) und reichen bis zum nächsten Zeilenende; geklammerte Kommentare werden in Paare von **/*** ... ***/**–Symbolen eingeschlossen. Beide Arten von Kommentaren können wechselseitig ineinander verschachtelt werden.
Für die Verifikation können außerdem Kontrollpunkte im Programmtext bestimmt werden, siehe Kapitel 9.2. Zwischen B(PN)²–Befehlen können hierzu beliebige Bezeichner, die mit einem **@** beginnen, eingefügt werden. Bei der Verifikation des Sytems mit den Model–Checkern oder Verfahren wie der Erreichbarkeitsanalyse können die Systemzustände über diese Kontrollpunkte referenziert werden.
Jedem (sequenziellen) Prozess kann ein Name zugewiesen werden, der, durch zwei Doppelpunkte **::** abgetrennt, dem ersten Befehl des jeweiligen Prozesses vorangestellt wird, also beispielsweise

var *i*: **int init** 1;
*proc*1:: <*i*:=*i*+3>
||
*proc*2:: <*i*:=*i*∗4>

Die Möglichkeit der Namensvergabe ist insbesondere interessant für die Verifikation. Bei der Erzeugung des vollständiges Präfixes der Entfaltung werden in allen Bedingungsknoten die Prozessnummern zur Identifizierung der Zugehörigkeit zu einer Komponente gespeichert und in den externen Dateiformaten mit abgelegt. Model–Checker wie der in [HNW99] beschriebene benötigen diese Informationen.

Zur Abkürzung mehrfach verwendeter Befehlsfolgen können einfache Text–Makros definiert werden, die folgendes Aussehen haben.

macro-def ::= **macro** *identifier parameters some-text* **endmacro**
parameters ::= **(** *macro-var-list* **)** | ϵ
macro-var-list ::= *identifier* **,** *macro-var-list* | *identifier*

Makrodefinitionen können an beliebiger Stelle vorkommen, an denen auch ein Kommentar möglich ist. Allerdings ist darauf zu achten, dass die Definition eines Makros stets vor seinem ersten Aufruf steht. Makrodefinitionen dürfen ineinander verschachtelt sein.[1] Eine erneute Verwendung des gleichen Namens für ein Makro überschreibt dessen vorherige Definition.

[1] Die maximale Schachtelungstiefe von Makrodefinitionen ist auf 10 voreingestellt, dieser Wert kann bei Bedarf vergrößert werden.

Die Bezeichner in der Parameterliste übernehmen die Funktion von Textvariablen, innerhalb der Makrodefinition werden sie referenziert durch ein vorangestelles Dollar–Symbol (ohne anderes Zeichen dazwischen). Sie können an beliebiger Stelle im Makrotext eingefügt werden, auch innerhalb von Zeichenketten eingebettet – es handelt sich um reine Textersetzung, die Textvariablen stehen in keiner Verbindung zu den Variablen des DB(PN)^2–Programms, Folgt einer Textvariablen ein Buchstabe oder eine Ziffer, so ist sie in geschweifte Klammern einzuschließen um sie vom folgenden Text abzugrenzen. Innerhalb der geschweiften Klammern können auch einfache arithmetische Ausdrücke stehen, die vor der Expandierung ausgewertet werden.
Die Syntax von Textvariablen ist damit die folgende.

macro-var ::= **$***identifier* | **${** *macro-expr* **}**
macro-expr ::= **(** *macro-expr* **)** | *number* | *identifier* | **–***macro-expr*
| *macro-expr num-op macro-expr*

Schließlich können Makros an beliebiger Stelle – nach und außerhalb ihrer Definition – aufgerufen werden durch Angabe ihres Namens, eventuell gefolgt von einer in runden Klammern eingeschlossenen Parameterliste.
Der folgende Programmtext zeigt ein Beispiel für die Anwendung von Makros.

Programmtext A.1 *Mutex–Algorithmus nach Knuth [Ray86] in DB(PN)^2 mit Makros (1)*

```
1   var c1,c2:card init 0;
2   var k:card init 1;
3   macro proc(i)
4     do <true>
5       <l1: c$i:=1>;
6       if <l2: k=$i> goto l3 fi;
7       if <c${3-i}#0> goto l2 fi;
8       <l3: c$i:=2>;
9       if <c${3-i}=2> goto l1 fi;
10      <k:=$i>;
11        /* Kritischer Abschnitt $i */
12      <k:=${3-i}>;
13      <c$i:=0>
14    od
15  endmacro
16  proc(1)||proc(2)                                    ■A.1
```

Hierbei sind die Prozesse mit Nummern versehen, die Variablen des Programmes werden daraus durch die in geschweiften Klammern spezifizierten Berechnungen gewonnen.
Eine alternative, vielleicht lesbarere Schreibweise des gleichen Programms mit Hilfe von Makros ist die nachstehende. Hierbei werden jedem Makroaufruf vier Parameter übergeben, die an den entsprechenden Textstellen ersetzt werden.

Programmtext A.2 *Mutex–Algorithmus nach Knuth [Ray86] in DB(PN)^2 mit Makros (2)*

```
1   var c1,c2: card init 0;
2   var k: card init 1;
3   macro proc( v1, j1, v2, j2 )
4     do < true >
5       < l1: $v1:= 1> ;
6       if < l2: k = $j1> goto l3 fi ;
7       if < $v2# 0> goto l2 fi ;
8       < l3: $v1:= 2> ;
9       if < $v2 = 2> goto l1 fi ;
10      < k:= $j1> ;
11        /* Kritischer Abschnitt $j1 */
12      < k:= $j2> ;
13      < $v1:= 0>
14    od
15  endmacro
16  proc( c1, 1, c2, 2 ) || proc( c2, 2, c1, 1 )                    ■A.2
```

A.2 Befehlsparameter und Optionen des Präfix–Generators

Der Haupt–Algorithmus zur Präfix–Generierung steht in einem separaten Programm namens `ERVunfold` neben der in Kapitel 9 beschriebenen Umgebung inzwischen in mehreren Werkzeugen zur Verfügung [PEP98, CPN99, SSE99]. Nachfolgend beschrieben sind die wichtigsten Programmoptionen, mit denen die Ein– und Ausgabe der Netze spezifiziert sowie die Art der Konstruktion des Präfixes beeinflußt werden kann.
Bis auf die Optionen zur Wahl der verschiedenen Eingabeformate sind diese Schalter auch für das Programm `dbpn2unf` anwendbar. Die Spezifikation der Datei mit dem DB(PN)^2–Programmtext erfolgt bei `dbpn2unf` mit der Option `-f=`*DB(PN)^2–Programm* oder alternativ durch Angabe lediglich des Dateinamens.

`-f=`*Netz–Datei*	Liest das zu entfaltendes Netzsystem aus *Netz–Datei*. Das Dateiformat für Netze ist im nächten Abschnitt beschrieben.
`-F=`*FSA–Datei*	Liest Eingabe aus der angegebenen *FSA–Datei*. Dieses Format beschreibt kommunizierende endliche Automaten und wird für alle Beispiele aus [Cor94] verwendet.
`-S=`*STG–Datei*	Liest Eingabe aus der angegebenen *STG–Datei*. In diesem Format, das u. a. von [SSL$^+$92] unterstützt wird, sind *Signal Transition Graphs* abgelegt, das sind Netz–ähnliche Strukturen, die bei der Verifikation von Schaltkreisen eingesetzt werden.

`-K`[*Schalter*]=*SND–Datei* Entfalten eines SND–Systems. Das Format der *SND–Datei* entspricht dem für allgemeine Netze bei Option `-f`, zusätzlich ist für jeden Stellenknoten die Zugehörigkeit zu den einzelnen Netzkomponenten durch eine Nummer spezifiziert. In den Präfix–Generator ist (noch) keine Routine zur automatischen Ermittlung der Komponenten integriert.
Über die optionalen *Schalter* lassen sich einige Parameter bei der Konstruktion des Präfixes beeinflussen:

- `1` oder `2` Wahl der SND–Ordnung $\sqsubset_{SND_1}$ bzw. $\sqsubset_{SND_2}$. Vorgabe ist die Ordnung $\sqsubset_{SND_1}$
- `a` oder `d` Festlegung, ob die Transitionen aufsteigend (`a` für *ascending*) oder absteigend (`d` für *descending*) bezüglich der Ordnung $\ll$ verwendet werden sollen. Voreinstellung ist aufsteigend; kann zusammen mit den anderen Schaltern angegeben werden.
- `s` Benutze die Summe über die lokalen *i*–Tiefen bei Vergleichen mit Hilfe der Ordnungen $\sqsubset_{SND_i}$. Nur bei gleichem Wert der Summen wird auf die aufwändigeren Vergleiche zurückgegriffen. Standardmäßig wird die Summe nicht betrachtet.

`-O`=*Netz–Datei* Ausgabe des Ausgangssystems nach *Netz–Datei*. Diese Option kann beispielsweise zur Konvertierung von STG– oder FSA–Dateien in Netzdateien genutzt werden.

`-U`=*Netz–Datei* Ausgabe des Präfixes in eine Netzdatei; es wird automatisch die Endung `.unf_net` angehängt. Für die Anzeige mit Hilfe von Netzeditoren wie dem aus [PEP98] werden die Knoten allerdings nicht mit graphischen Koordinaten versehen.

`-u`=*unf–Datei* Schreibt das konstruierte Präfix nach *unf–Datei*, die Endung `.unf` wird automatisch angehängt. Darin enthalten sind Informationen zur Netzstruktur des Präfixes, die Abbildung π sowie Angaben über die Cut-off–Ereignisse und jeweils ein dazu korrespondieres Ereignis (siehe hierzu auch Schalter `-l`). Außerdem kann eine Liste der globalen Konfigurationen enthalten sein (siehe auch Schalter `-c`).

	Dieses Dateiformat dient vornehmlich zur (manuellen) Analyse des erzeugten Netzes. Eine kompaktere Darstellung bietet der Schalter `-m`.
`-m=`*mci–Datei*	Schreibt das konstruierte Präfix in einer kompakten Binärdarstellung nach *mci–Datei*, die Endung `.mci` wird automatisch angehängt. Das *mci*–Format (kurz für *model checker input format*) wird insbesondere von den in den Werkzeugen vorhandenen Model–Checker–Programmen als Eingabe verwendet.
`-M=`*Matrix–Datei*	Ausgabe der co–Matrix für die Bedingungsknoten (untere Dreiecksmatrix) in binär codierter Darstellung nach *Matrix–Datei*; es wird automatisch die Endung `.bin_coco` angehängt.
`-n[=`*max–co*`]`	Vorgabe, bis zu welcher Präfixgröße (Zahl von Ereignissen) bei der Berechnung der co–Relation die co–Matrix gespeichert werden soll. Bei Überschreiten dieses Wertes wird die Berechnung der für die Ermittlung der erweiternden ereignisse notwendigen co–Mengen weniger effizient mit Tiefensuche durchgeführt. Standardmäßig gibt es keine interne Beschränkung. Wird kein expliziter Wert angegeben, wird eine Größe von 3000 Ereignissen angenommen.
`-@[@]`	Wahl der Ordnung $\sqsubset_t$ anstelle der voreingestellten Ordnung $\sqsubset_{McM}$. Bei Angabe des zweiten `@` wird für beschleunigte, aber speicherintensivere Berechnung der co–Matrix aus Kapitel 8.2.4 verwendet.
`-c`	Unterdrücke die Erzeugung der globalen Konfigurationen. Die Berechnung ist standardmäßig aktiviert, diese Konfigurationen werden benötigt für den S4–Model–Checker aus [Esp93].
`-C`	Unterdrückt das Einfügen von Bedingungen im Nachbereich von Cut-off–Ereignissen. Standardmäßig werden sie erzeugt.
`-x[=`*max–event*`]`	Beschränkt die Netzgröße des Präfixes auf *max–event* Ereignisse; wird die Zahl nicht angegeben, wird der Wert 10000 angenommen. Standardmäßig ist die Netzgröße intern nicht beschränkt, d. h. nur durch die zur Verfügung stehenden Ressourcen des jeweiligen Rechners.

`-l`	Speichert bei der Ausgabe des Präfixes (Schalter `-u` bzw. `-m`) den Namen des bezüglich der verwendeten Ordnung $\sqsubset$ jeweils letzten Ereignisses, das zu einem Cut-off–Ereignis korrespondiert. Standard ist der Bezug auf das jeweils erste solche Ereignis. Diese Information wird beispielsweise vom LTL–Model–Checker [Wal98, Wal00] für die Konstruktion der Zyklen benötigt.
`-I`	Verwendet die aktuelle Markierung des Netzsystems anstelle der Startmarkierung. Bei Netzsimulatoren, wie sie z. B. in [PEP98] integriert sind, kann das System geschaltet werden und mit einer anderen als der Startmarkierung gespeichert werden. Soll das Präfix beginnend mit dier Markierung generiert werden, dann ist diese Option hilfreich.
`-X=`*Transition*	Bricht die Präfix–Generierung ab, sobald ein Ereignis eingefügt worden ist, das mit dem Namen *Transition* beschriftet ist. Damit lassen sich einfache Erreichbarkeitsfragen schon während der Konstruktion des Präfixes lösen.
`-T[+[+]][=`*min–Datei*`]`	Ausgabe der Liste der nicht im Präfix repräsentierten Transitionen auf dem Bildschirm, bei Angabe von *min–Datei* in die entsprechende Datei, wobei die Endung `.min` angehängt wird. Der Schalter `+` entfernt die entsprechenden Transitionen aus dem ursprünglichen System, bei Angabe von `++` werden außerdem isolierte Stellenknoten entfernt. Das so veränderte Netzsystem kann mit der Option `-O` in eine Datei geschrieben werden.
`-q`	Zeigt während der Konstruktion des Präfixes den Status der aktuellen Berechnung: Netzgröße und Zahl der Cut-off–Ereignisse. Anzeige in einer Ausgabezeile, d. h. die neue Information überschreibt die vorherige.
`-p`	Wie Option `-q`, die Anzeige erfolgt jedoch für jeden Statusbericht in einer eigenen Zeile.
`-t`	Ausgabe der wichtigsten statistischen Informationen am Ender der Präfix–Berechnung auf dem Bildschirm. Hierzu gehören die Größe des Präfixes, Zahl der Cut-off–Ereignisse, benötigte Zeit (reine Konstruktionszeit ohne Einlesen des Netzes).
`-s`	Ausgabe erweiterter statistischer Größen. Zusätzlich zur Option `-t` werden Angaben gemacht zum Speicherverbrauch sowie zur Netzstruktur (z. B. maximale Größe eines Vorbereichs einer Transition).
`-v`	Anzeige der Versionsnummer des Programms.

Literaturverzeichnis

[Arn94] ARNOLD, A.: *Finite Transition Systems: Semantics of Communicating Systems.* Prentice–Hall, Englewood Cliffs, 1994.

[Bau90] BAUMGARTEN, B.: *Petri–Netze. Grundlagen und Anwendungen*. BI–Wissenschafts–Verlag, 1990.

[BC91] BOUDOL, G. und I. CASTELLANI: *Flow Models of Distributed Computations: Event Structures and Nets.* Rapport de Recherche, INRIA, Sophia Antipolis, Juli 1991.

[BCM+92] BURCH, J. R., E. M. CLARKE, K. L. MCMILLAN, D. L. DILL und L. J. HWANG: *Symbolic Model Checking:* 10^{20} *States and Beyond.* Information and Computation, 98(2):142–170, 1992.

[BD87] BEST, E. und R. DEVILLERS: *Sequential and Concurrent Behaviour in Petri Net Theory*. Theoretical Computer Science, 55:87–136, 1987.

[BDE93] BEST, E., R. DEVILLERS und J. ESPARZA: *General Refinement and Recursion for the Box Calculus.* In: ENJALBERT, P., A. FINKEL und K. W. WAGNER (Herausgeber): *STACS'93*, Band 665 der Reihe *LNCS*, Seiten 130–140. Springer–Verlag, 1993.

[BDH92] BEST, E., R. DEVILLERS und J. G. HALL: *The Box Calculus: A New Causal Algebra with Multi–Label Communication*. In: ROZENBERG, G. (Herausgeber): *Advances in Petri Nets 1992*, Band 609 der Reihe *LNCS*, Seiten 21–69. Springer–Verlag, 1992.

[Bes96] BEST, E.: *Partial Order Verification with PEP*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 29:305–328, 1996.

[BF87] BEST, E. und C. FERNÁNDEZ: *Notations and Terminology on Petri Nets.* Arbeitspapiere der GMD 195, Gesellschaft für Mathematik und Datenverarbeitung mbH, März 1987.

[BF88] BEST, E. und C. FERNÁNDEZ: *Nonsequential Processes – A Petri Net View*, Band 13 der Reihe *EATCS Monographs on Theoretical Computer Science.* Springer–Verlag, 1988.

[BF95] BEST, E. und H. FLEISCHHACK: *PEP: Programming Environment Based on Petri Nets*. Hildesheimer Informatik Bericht 14/95, Universtät Hildesheim, Mai 1995.

[BF99] BIEBER, B. und H. FLEISCHHACK: *Model Checking of Time Petri Nets Based on Partial Order Semantics*. In: BAETEN, J. C. M. und S. MAUW (Herausgeber): *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR) '99*, Band 1664 der Reihe *LNCS*, Seiten 210–225, Eindhoven, The Netherlands, August 1999. Springer–Verlag.

[BFF+95] BEST, E., H. FLEISCHHACK, W. FRACZAK, R. P. HOPKINS, H. KLAUDEL und E. PELZ: *An M–Net Semantics of B(PN)²*. In: DESEL, J. (Herausgeber): *Structures in Concurrency Theory (STRICT)*, Seiten 85–100. Springer–Verlag, Mai 1995.

[BGV90] BRAUER, W., R. GOLD und W. VOGLER: *A Survey of Behaviour and Equivalence Preserving Refinements of Petri Nets*. In: ROZENBERG, G. (Herausgeber): *Advances in Petri Nets 1990*, Band 483 der Reihe *LNCS*, Seiten 1–46. Springer–Verlag, 1990.

[BH93] BEST, E. und R. P. HOPKINS: *B(PN)² – a Basic Petri Net Programming Notation*. In: BODE, A., M. REEVE und G. WOLF (Herausgeber): *Proceedings of PARLE '93*, Band 694 der Reihe *LNCS*, Seiten 379–390. Springer–Verlag, 1993.

[Bry86] BRYANT, R. E.: *Graph–Based Algorithm for Boolean Function Manipulation*. IEEE Transactions on Computers, C–35(8):677–691, August 1986.

[Büc62] BÜCHI, J. R.: *On a Decision Method in Restricted Second Order Arithmetic*. In: *Proceedings of the International Congress on Logic, Method and Philosophical Science 1960*, Seiten 1–12, Stanford, 1962. Stanford University Press.

[CEP95] CHANG, A., J. ESPARZA und J. PALSBERG: *Complexity results for 1–safe nets*. Theoretical Computer Science, 147:117–136, 1995.

[CES86] CLARKE, E. M., E. A. EMERSON und A. P. SISTLA: *Automatic Verification of Finite–State Concurrent Systems Using Temporal Logic Specifications, Testing and Verification*. ACM Transactions on Programming Languages and Systems, 8(2):244–263, 1986.

[Chi87] CHIOLA, G.: *GreatSPN – A Tool for the Analysis of Generalised Stochastic Petri Nets*, 1987.

[Cor94] CORBETT, J. C.: *Evaluating Deadlock Detection Methods for Concurrent Software*. In: OSTRAND, T. (Herausgeber): *Proceedings of the 1994 International Symposium on Software Testing and Analysis – ISSTA '94*, Seiten 204–215. ACM–Press, 1994. Die Beispiele sind erhältlich unter `ftp://ftp.ics.hawaii.edu/pub/corbett/eval.tar.Z`.

[CPN99] *CPN–AMI Version 2.4, Handbook*, 1999. `http://www-src.lip6.fr/logiciels/framekit/cpn-ami.html`.

[DE95] DESEL, J. und J. ESPARZA: *Free Choice Petri Nets*. Cambridge University Press, 1995.

[Dev93] DEVILLERS, R.: *On a more Liberal Synchronisation Operator for the Petri Box Calculus*. Technischer Bericht TR–LIT–281, Université Libre de Bruxelles, Mai 1993.

[Die90] DIEKERT, V.: *Combinatorics on Traces*, Band 454 der Reihe *LNCS*. Springer–Verlag, 1990.

[Dij68] DIJKSTRA, E. W.: *Go to Statement Considered Harmful*. Communications of the ACM, 11(3):147–148, März 1968.

[EB96] ESPARZA, J. und G. BRUNS: *Trapping Mutual Exclusion in the Box Calculus*. Theoretical Computer Science, 153:95–128, 1996.

[EH00] ESPARZA, J. und K. HELJANKO: *A New Approach to LTL Model Checking Based on Net Unfoldings*, 2000. Papier in Vorbereitung.

[EN94] ESPARZA, J. und M. NIELSEN: *Decidability Issues for Petri Nets – a Survey*. Journal of Information Processing and Cybernetics, 30(3):143–160, 1994.

[Eng91] ENGELFRIET, J.: *Branching Processes of Petri Nets*. Acta Informatica, 28:575–591, 1991.

[ER99] ESPARZA, J. und S. RÖMER: *An Unfolding Algorithm for Synchronous Products of Transition Systems*. In: BAETEN, J. C. M. und S. MAUW (Herausgeber): *Proceedings of the 10^{th} International Conference on Concurrency Theory (CONCUR) '99*, Band 1664 der Reihe *LNCS*, Seiten 2–20, Eindhoven, The Netherlands, August 1999. Springer–Verlag.

[ERV96] ESPARZA, J., S. RÖMER und W. VOGLER: *An Improvement of McMillan's Unfolding Algorithm*. In: MARGARIA, T. und B. STEFFEN (Herausgeber): *Proceedings of the 2^{nd} International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) '96 (Passau, Germany)*, Band 1055 der Reihe *LNCS*, Seiten 87–106. Springer–Verlag, 1996.

[ERV00] ESPARZA, J., S. RÖMER und W. VOGLER: *An Improvement of McMillan's Unfolding Algorithm*. Akzeptiert zur Veröffentlichung in Formal Methods in System Design, 2000.

[ES92] ESPARZA, J. und M. SILVA: *A Polynomial–Time Algorithm To Decide Liveness of Bounded Free Choice Nets*. Theoretical Computer Science, 102:185–205, 1992.

[ES93] EMERSON, E. A. und A. P. SISTLA: *Symmetry and Model Checking*. In: COURCOUBETIS, C. (Herausgeber): *Proceedings of the* 5^{th} *International Conference on Computer–Aided Verification (CAV) '93 (Elouda, Greece)*, Band 697 der Reihe *LNCS*. Springer–Verlag, 1993.

[Esp93] ESPARZA, J.: *A Partial Order Approach to Model Checking*. Habilitationsschrift, Universität Hildesheim, Februar 1993.

[Esp94] ESPARZA, J.: *Model Checking Using Net Unfoldings*. Science of Computer Programming, 23:151–195, 1994.

[Fre98] FREYTAG, T.: *VIPtool – A Partial Order Based Simulation Tool for High–Level Petri Nets*. Universität Karlsruhe, 1998. `http://www.aifb.uni-karlsruhe.de/InfoSys/VIP/viptool/viptool.html`.

[Gra95] GRAVES, B.: *Ein Modelchecker für eine Linear–Time–Logik*. Diplomarbeit, Universität Hildesheim, Januar 1995.

[Gra97] GRAVES, B.: *Computing Reachability Properties Hidden in Finite Net Unfoldings*. In: *Proceedings of the* 17^{th} *FSTTCS*, Band 1346 der Reihe *LNCS*, Seiten 327–342. Springer–Verlag, 1997.

[Haa98] HAAR, S.: *Branching Processes of General S/T–Systems*. In: BURKHARD, H.-D., L. CZAJA und P. STARKE (Herausgeber): *Workshop Concurrency, Specification and Programming*, Band 110 der Reihe *Informatik–Berichte*, Seiten 88–97, Berlin, September 1998. Humboldt–Universität.

[HC68] HUGHES, G. E. und M. J. CRESWELL: *An Introduction to Modal Logic*. Methuen and Co., 1968.

[HD96] HEINER, M. und P. DEUSSEN: *Petri Net Based Design and Analysis of Reactive Systems*. In: *Proceedings of the Workshop on Discrete Event Systems (WODES) '96, Edinburgh*, Seiten 308–313, London, 1996. The Institution of Electrical Engineers.

[HDS99] HEINER, M., P. DEUSSEN und J. SPRANGER: *A Case Study in Design and Verification of Manufacturing System Control Software with Hierarchical Petri Nets*. The International Journal of Advanced Manufacturing Technology, 15:139–152, 1999.

[Hel98] HELJANKO, K.: *Deadlock Checking for Complete Finite Prefixes Using Logic Programs with Stable Model Semantics*. In: *Proceedings of the Workshop Concurrency, Specification & Programming (CS&P) '98*, Band 110 der Reihe *Informatik Bericht*, Seiten 106–115, Berlin, September 1998. Humboldt–Universität.

[Hel99a] HELJANKO, K.: *Minimizing Finite Complete Prefixes*. In: *Proceedings of the Workshop Concurrency, Specification & Programming (CS&P) '99*, September 1999. Akzeptiert zur Veröffentlichung.

[Hel99b] HELJANKO, K.: *Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1–Safe Petri Nets*. In: *Proceedings of the* 5^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) '99*, Band 1579 der Reihe *LNCS*, Seiten 240–254. Springer–Verlag, 1999.

[HHB92] HALL, J. G., R. P. HOPKINS und O. BOTTI: *A Basic Net Algebra for Program Semantics and its Application to occam*. In: ROZENBERG, G. (Herausgeber): *Advances in Petri Nets 1992*, Band 609 der Reihe *LNCS*. Springer–Verlag, 1992.

[Hül94] HÜLSMANN, H. B.: *Wenn der schöne Schein verblaßt – Rund um die Gewährleistung*. c't Magazin für Computertechnik, 8:52–53, 1994.

[HNW98] HUHN, M., P. NIEBERT und F. WALLNER: *Verification Based on Local States*. In: STEFFEN, B. (Herausgeber): *Proceedings of the* 4^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) '98 (Lisboa, Portugal)*, Band 1384 der Reihe *LNCS*, Seiten 36–51. Springer–Verlag, 1998.

[HNW99] HUHN, M., P. NIEBERT und F. WALLNER: *Model Checkig Logics for Communicating Sequential Agents*. In: THOMAS, W. (Herausgeber): *Proceedings of the* 2^{nd} *International Conference on Foundations of Software Science and Computation Structures (FOSSACS) '99 (Amsterdam, The Netherlands)*, Band 1578 der Reihe *LNCS*, Seiten 227–242. Springer–Verlag, März 1999.

[Hoa78] HOARE, C. A. R.: *Communicating Sequential Processes*. Communications of the ACM, 21(8):666–677, 1978.

[Hol97] HOLZMANN, G. J.: *The Model Checker Spin*. IEEE Transactions on Software Engineering, 23(5):279–295, 1997.

[Hüs95] HÜSKES, R.: *Intels Tauschaktion – Umtauschrecht für fehlerhafte Pentium–Prozessoren*. c't Magazin für Computertechnik, 2:17, 1995.

[Hu95] HU, A. J.: *Techniques for Efficient Formal Verification Using Binary Decision Diagrams*. Doktorarbeit, Stanford University, 1995.

[Int94] INTEL CORPORATION: *Pentium® Processor Replacement (FDIV) Program Information*, 1994. `http://www.intel.com/procs/support/pentium/fdiv`.

[Jen94] JENNER, L.: *Ein Low–Level Prozedurkonzept für B(PN)^2*. Diplomarbeit, Universität Hildesheim, 1994.

[KEB93] KOUTNY, M., J. ESPARZA und E. BEST: *Operational Semantics for the Petri Box Calculus*. Nummer 13/93 in *Hildesheimer Informatik–Bericht*. Universität Hildesheim, Oktober 1993.

[KEB94] KOUTNY, M., J. ESPARZA und E. BEST: *Operational Semantics for the Petri Box Calculus*. In: JONSSON, B. und J. PARROW (Herausgeber): *Proceedings of the 5th International Conference on Concurrency Theory (CONCUR '94)*, Band 836 der Reihe *LNCS*, Seiten 210–225. Springer–Verlag, August 1994.

[Kem96] KEMPER, P.: *Reachability Analysis Based on Structured Representations*. In: *Proceedings of the 17th International Conference on Application and Theory of Petri Nets (ATPN) '96 (Osaka, Japan)*, Band 1091 der Reihe *LNCS*. Springer–Verlag, Juni 1996.

[KKT+98] KONDRATYEV, A., M. KISHINEVSKY, A. TAUBIN, J. CORTADELLA, L. LAVAGNO und A. YAKOVLEV: *Identifying State Coding Conflicts in Asynchronous System Specifications Using Petri Net Unfoldings*. In: *Proceedings of the International Conference on Application of Concurrency to System Design (CSD) '98*, Seiten 152–163, Fukushima, Japan, März 1998. IEEE Computer Society.

[Knu74] KNUTH, D. E.: *Structured Programming with Go to Statements*. ACM Computing Surveys, 6(4):261–301, Dezember 1974.

[Kou94] KOUTNY, M.: *Partial Order Semantics of Box Expressions*. In: VALETTE, R. (Herausgeber): *Proceedings of the 15th International Conference on Application and Theory of Petri Nets (ATPN) '94*, Band 815 der Reihe *LNCS*, Seiten 318–337. Springer–Verlag, 1994.

[KR78] KERNIGHAN, B. W. und D. RITCHIE: *The C Programming Language*. Prentice Hall, 1978.

[Lam77] LAMPORT, L.: *Proving the Correctness of Multiprocess Programs*. IEEE Transactions on Software Engineering, 3(2):125–143, 1977.

[LB99] LANGERAK, R. und E. BRINKSMA: *A Complete Finite Prefix for Process Algebra.* In: HALBWACHS, N. und D. PELED (Herausgeber): *Proceedings of the* 11^{th} *International Conference on Computer–Aided Verification (CAV) '99 (Trento, Italy)*, Band 1633 der Reihe *LNCS*. Springer–Verlag, 1999.

[LG93] LINDE-GÖERS, H.-G.: *Compositional Partial Order Semantics of Petri Boxes.* Dissertation, Universität Hildesheim, Dezember 1993.

[Lil98] LILIUS, J.: *Efficient State Space Search for Time Petri Nets.* In: JANČAR, P. und M. KŘETÍNSKÝ (Herausgeber): *MFCS '98 Workshop on Concurrency*, Electronic Notes of Theoretical Computer Science. Elsevier, August 1998.

[May83] MAY, D.: *Occam.* In: *IEEE Colloquium on Software Tools for Hardware Design, No 98*, 1983.

[Maz88] MAZURKIEWICZ, A.: *Compositional Semantics of Pure Place/Transition Systems.* In: ROZENBERG, G. (Herausgeber): *Advances in Petri Nets 1988*, Band 340 der Reihe *LNCS*, Seiten 307–330. Springer–Verlag, 1988.

[McM92] MCMILLAN, K. L.: *Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits.* In: BOCHMANN, G. V. und D. K. PROBST (Herausgeber): *Proceedings of the* 4^{th} *International Conference on Computer–Aided Verification (CAV) '92*, Band 663 der Reihe *LNCS*, Seiten 164–174, Montreal, 1992. Springer–Verlag.

[McM95] MCMILLAN, K. L.: *A Technique of State Space Search Based on Unfolding.* Formal Methods in System Design, 6(1):45–65, 1995.

[Mel98] MELZER, S.: *Verifikation verteilter Systeme mittels linearer – und Constraint–Programmierung.* Dissertation, Technische Universität München, 1998.

[Mer96] MERKEL, S.: *Verification of Fault Tolerant Algorithms Using PEP.* SFB–Bericht 342 / 23 / 97 A, Technische Universität München, 1996.

[Mil80] MILNER, R.: *A Calculus of Communicating Systems*, Band 92 der Reihe *LNCS*. Springer–Verlag, 1980.

[Mil89] MILNER, R.: *Communication and Concurrency*. Prentice Hall, 1989.

[MR97] MELZER, S. und S. RÖMER: *Deadlock Checking Using Net Unfoldings.* In: GRUMBERG, O. (Herausgeber): *Proceedings of the* 9^{th} *International COnference on Computer–Aided Verification (CAV) '97 (Haifa, Israel)*, Band 1254 der Reihe *LNCS*, Seiten 352–363. Springer–Verlag, Juni 1997.

[MRE96] MELZER, S., S. RÖMER und J. ESPARZA: *Verification using PEP*. In: WIRSING, M. und M. NIVAT (Herausgeber): *Proceedings of Algebraic Methodology and Software Technology (AMAST) '96*, Band 1101 der Reihe *LNCS*, Seiten 591–594. Springer–Verlag, 1996.

[Mur89] MURATA, T.: *Petri Nets: Properties, Analysis and Applications*. Proceedings of IEEE, 77(4):541–580, 1989.

[NPW80] NIELSEN, M., G. PLOTKIN und G. WINSKEL: *Petri Nets, Event Structures and Domains*. Theoretical Computer Science, 13(1):85–108, 1980.

[PEP98] *PEP – Petrinetz–basierte Entwicklungs– und Programmierumgebung*, 1998. `http://theoretica.informatik.uni-oldenburg.de/~pep`.

[Pet62] PETRI, C. A.: *Kommunikation mit Automaten*. Schriften des Instituts für instrumentelle Mathematik, Bonn, 1962.

[Pet81] PETERSON, G. L.: *Myths about the Mutual Exclusion Problem*. Information Processing Letters, 12 / 3:115–116, 1981.

[Pnu77] PNUELI, A.: *The Temporal Logic of Programs*. In: *Proc. of the* 18^{th} *IEEE Symposium on Foundation of Computer Science (FOCS) '77*, Seiten 46–57, 1977.

[Ray86] RAYNAL, M.: *Algorithms for Mutual Exclusion*. North Oxford Academic Publisher Ltd., 1986.

[Rei85] REISIG, W.: *Petri Nets. An Introduction*, Band 4 der Reihe *EATCS Monographs on Theoretical Computer Science*. Springer–Verlag, 1985.

[Rie94] RIEMANN, R.-C.: *Eine temporale Logik für parallele Programme*. Diplomarbeit, Universität Hildesheim, Januar 1994.

[RM96] RÖMER, S. und S. MELZER: *Synchronisierende Automaten in PEP*. In: DESEL, J. und E. KINDLER UND A. OBERWEIS (Herausgeber): *Proceedings zum 3. Workshop Algorithmen und Werkzeuge für Petrinetze (AWPN) '96*, Forschungsberichte, Seiten 52–59. AIFB Universität Karlsruhe, Oktober 1996.

[Röm93] RÖMER, S.: *Implementierung einer kompositionellen Halbordnungssemantik für Petri–Boxen*. Diplomarbeit, Universität Hildesheim, 1993.

[Röm95] RÖMER, S.: *Ein effizienter Algorithmus zur Berechnung von Entfaltungen endlicher, 1–sicherer Petrinetze*. In: [BF95].

[Rub87] RUBIN, F.: *"GOTO Considered Harmful"Considered Harmful*. Communications of the ACM, ACM Forum, 30(3):195–196, März 1987.

[Sán96] SÁNCHEZ, B.: *Model Checking for State Graphs via Integer Programming*. Interner Bericht ZF SE 95/96 Sanchez–1, Siemens ZFE, 1996.

[SB94] SILBERTIN-BLANC, C.: *Cooperative Nets*. In: VALETTE, R. (Herausgeber): *Proceedings of the* 15^{th} *International Conference on Application and Theory of Petri Nets (ATPN) '94*, Band 815 der Reihe *LNCS*, Seiten 471–490. Springer–Verlag, 1994.

[SDL92] *SDL – Specification and Description Language*, 1992.

[SE00] SCHRÖTER, C. und J. ESPARZA: *Reachability Analysis Using Net Unfoldings*. Papier in Vorbereitung, 2000.

[Sed88] SEDGEWICK, R.: *Algorithms*. Addison–Wesley, Princeton University, 2nd Auflage, 1988.

[SMV98] *SMV – Version 2.5 (27.09.1998) – Versionen für SPARC (SunOS4, Solaris), Linux (Intel x86), Ultrix (DECstation) und MS Windows NT*, 1998. `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[SSE99] SCHRÖTER, C., S. SCHWOON und J. ESPARZA: *Model Checking Kit, Sammlung von netzbasierten Modellierungssprachen und Verifikationswerkzeugen*, 1999. `http://wwwbrauer.in.tum.de/gruppen/theorie/KIT`.

[SSL+92] SENTOVICH, E. M., K. J. SINGH, L. LAVAGNO, C. MOON, R. MURGAI, A. SALDANHA, H. SAVOJ, P. R. STEPHAN, R. K. BRAYTON und A. SANDIOVANNI-VINCENTELLI: *SIS: A System for Sequential Circuit Synthesis*. Department of Electrical Engineering and Computer Science, University of California, Berkeley, Mai 1992. `ftp://www.u-aizu.ac.jp/u-aizu/sis/sis-solaris2.tar.gz`.

[Sta90] STARKE, P. H.: *Analyse von Petri–Netz–Modellen*. B. G. Teubner–Verlag, Stuttgart, 1990.

[Sta91] STARKE, P. H.: *Reachability Analysis of Petri Nets Using Symmetries*. Journal of Mathematical Modelling and Simulation in Systems Analysis, 8(5/6):293–304, 1991.

[Sta92] STARKE, P. H.: *INA: Integrated Net Analyzer*. Handbuch, 1992.

[Stö98] STÖRRLE, H.: *An Evaluation of High–End Tools for Petri Nets*. Technischer Bericht Ludwig Maximilian Universität München, 1998.

[Unf99] *Bibliography on Unfoldings*, 1999. `http://wwwbrauer.informatik.tu-muenchen.de/gruppen/theorie/pom/pom.shtml`.

[Val92] VALMARI, A.: *A Stubborn Attack on State Explosion*. Formal Methods in System Design, 1:297–322, 1992.

[Val94] VALMARI, A.: *Compositional Analysis with Place-Ordered Subnets*. In: VALETTE, R. (Herausgeber): *Proceedings of the* 15^{th} *International Conference on Application and Theory of Petri Nets (ATPN) '94*, Band 815 der Reihe *LNCS*, Seiten 531–547. Springer–Verlag, 1994.

[VHHP95] VARPAANIEMI, K., J. HALME, K. HIEKKANEN und T. PYSSYSALO: *PROD reference manual*. Technischer BerichtB13, Helsinki university of technology, Digital Systems Laboratory, 1995.

[Vog91] VOGLER, W.: *Executions: a New Partial–Order Semantics of Petri Nets*. Theoretical Computer Science, 91:205–238, 1991.

[Vog92] VOGLER, W.: *Modular Construction and Partial Order Semantics of Petri Nets*, Band 625 der Reihe *LNCS*. Springer–Verlag, 1992.

[VSY98] VOGLER, W., A. SEMENOV und A. YAKOVLEV: *Unfolding and Finite Prefix for Nets with Read Arcs*. In: SANGIORGI, D. und R. DE SIMONE (Herausgeber): *Proceedings of the* 9^{th} *International Conference on Concurrency Theory (CONCUR) '98 (Nice, France)*, Band 1466 der Reihe *LNCS*, Seiten 501–516. Springer–Verlag, 1998.

[VW86] VARDI, M. Y. und P. WOLPER: *An Automata–Theoretic Approach to Automatic Program Verification*. In: *Proceedings of the* 1^{st} *IEEE Symposium on Logic in Computer Science (LICS) '86*, Seiten 322–331, Cambridge, 1986.

[Wal98] WALLNER, F.: *Model–Checking LTL Using Net Unfoldings*. In: HU, A. und M. VARDI (Herausgeber): *Proceedings of the* 10^{th} *International Conference on Computer–Aided Verification (CAV) '98 (Vancouver, BC, Canada)*, Band 1427 der Reihe *LNCS*, Seiten 207–218. Springer–Verlag, Juni 1998.

[Wal00] WALLNER, F.: *Automatic Verification with Net Unfoldings*. Dissertation, in Vorbereitung, Technische Universität München, 2000.

Index